237. Tours on a 4 x n playing board

Difficulty rating: 70 %.
Solution: 15836928. *Solved: Tue, 28 Feb 2017, 08:21.*
Math knowledge used: recurrence equations.
Programming techniques used: binary exponentiation.

One of my favourite problems ever. Like many others, at first it seems insurmountable, but it can be done, and in fact I consider that 70 % is too high a rating for this problem. The problem can be divided into three parts, and only the first one has any difficulty. The other two are reasonably easy, and in fact the third one is very straightforward. These three parts are: determinating the pattern, mathematical analysis to yield a workable formula, and coding the recursion.

The hardest part, as I mentioned, is determining the pattern. I don't remember spending that much time with this, but this is one of the cases where you'll need an Eureka moment and there's no real way to turn the lightbulb on aside from doing some experiments for smaller boards and/or reasoning about the structure. The pattern is like this: we can divide the whole "tour" into three sections. Two of these sections, which are related, move from the start and end squares towards the right end of the board; in fact, each one of the sections goes from the initial square to a square in the same row, and both sections end in the same column (i.e. if the starting squares are $(0, 0)$ and $(0, 3)$, then the final squares are $(i, 0)$ and $(i, 3)$ for a common value of i; these sections completely fill the board up to column i). The other one connects these two sections, and since it's the easier section, we can start from there. This section can have any width from 1 to ∞, and since it needs to completely fill a region with certain width x and fixed height 4, the way it works is by adding a zigzagging pattern: if the total width of the board is N and this section starts at column $i + 1$, it comprises seven segments: $(i + 1, 3) - (N, 3)$, $(N, 3) - (N, 2)$, $(N, 2) - (i + 1, 2)$, $(i + 1, 2) - (i + 1, 1)$, $(i + 1, 1) - (N, 1)$, $(N, 1) - (N, 0)$, and $(N, 0) - (i + 1, 0)$. For the special case where $i + 1 = N$ (width of just one square), there is instead one single line connecting $(i + 1, 3)$ and $(i + 1, 0)$. In any case, this "end" section can only have one shape, given the width. The other two sections are built in parallel, from three kind of different blocks:

- The first kind of block has any width greater or equal than two, and it consists of a "mushroom" protruding from the top row, while the bottom row stays flat. This "mushroom", having a known width x, must have its base in some two consecutive points, therefore such a block of width x might have $x - 1$ different configurations. If this seems

confusing, this is illustrated in columns 4 to 8 of the example pattern, with the "mushroom's base" in columns 7 and 8.

- The second kind of block is exactly like the first one, but reversed (the "mushroom" stems from the bottom).
- Finally, the last kind of block always has a width of exactly two, and it consists of two stumps of height 2 at each one of the rows. This is illustrated at columns 2 and 3 of the problem figure.

Note that all of these blocks start and end both end points at rows 0 and 3. I'm not sure if I could sketch a mathematical proof that these patterns cover all the possible cases, but as far as I know, and considering that I got the problem answer right, I would say that this is in fact correct.

Now, the next part, easier than the previous one, but still tricky. We need to create a mathematical model of this pattern. To do this, we start with a succession, $A(n)$, defined as the amount of ways to fill a board of width n using the three "initial" kinds of blocks. This can defined recursively, by adding either a block of length 2 from the third kind, or a block of length $m \geq 2$ from either the first or second kind. Each one of these can be $m-1$ cases, so in total there would be $2m-2$ cases. This reasoning yields this initial formula:

$$A(n) = A(n-2) + \sum_{m=2}^{\infty} (2m-2)\, A(n-m)\,.$$

The base cases would be $A(0) = 1$ (no board, trivial non-solution), and $A(n) = 0, \forall n < 0$. Unfortunately, this formula is not really workable because of the infinite sum there. But we can take the formula for $n-1$:

$$A(n-1) = A(n-3) + \sum_{m=2}^{\infty} (2m-2)\, A(n-1-m)\,;$$

and if we subtract $A(n) - A(n-1)$, we get:

$$A(n) - A(n-1) = A(n-2) - A(n-3) + \sum_{m=2}^{\infty} (2m-2)\, A(n-m) - \sum_{m=2}^{\infty} (2m-2)\, A(n-1-m)\,.$$

If we do a $m \to m-1$ replacement in the latest summatory, we get:

$$A(n) - A(n-1) = A(n-2) - A(n-3) + \sum_{m=2}^{\infty} (2m-2)\, A(n-m) - \sum_{m=3}^{\infty} (2m-4)\, A(n-m)\,.$$

This can be rewritten as

$$A(n) - A(n-1) = A(n-2) - A(n-3) + 2A(n-2) + \sum_{m=3}^{\infty} 2A(n-m),$$

or, much better:

$$A(n) = A(n-1) + 3A(n-2) - A(n-3) + \sum_{m=3}^{\infty} 2A(n-m).$$

We still have an infinite summatory, but we have reduced the degree of its terms, from one to zero. Repeating the same procedure will remove the summatory. We first note that

$$A(n-1) = A(n-2) + 3A(n-3) - A(n-4) + \sum_{m=3}^{\infty} 2A(n-1-m).$$

Subtracting the latest two equations, we get

$$\begin{aligned} A(n) - A(n-1) = A(n-1) + 2A(n-2) - 4A(n-3) + A(n-4) + \\ + \sum_{m=3}^{\infty} 2A(n-m) - \sum_{m=3}^{\infty} 2A(n-1-m). \end{aligned}$$

But the difference of the summatories is just $2A(n-3)$, yielding a reasonable formula:

$$A(n) = 2A(n-1) + 2A(n-2) - 2A(n-3) + A(n-4).$$

Theoretically one could solve this recurrence analytically, but the roots of the associated characteristic polynomial, $x^4 - 2x^3 - 2x^2 + 2x - 1$, are not integers. In fact there are two complex roots. This means that we can't use a closed formula if we want to stay in the domain of the integers. Too bad.

In any case, this is still not the final formula, since it can only describe the starting section. If we add the "final" section, we can determine $T(n)$. Considering that this final section can have any integer width greater or equal than one, and that given any width there is only a single possibility, the formula for $T(n)$ based on $A(n)$ is deceptively simple:

$$T(n) = \sum_{m=1}^{\infty} A(n-m).$$

Of course, we can't use this directly, but if we subtract two consecutive terms of this summation, we obviously get

$$T(n) - T(n-1) = A(n-1).$$

What this means is that the roots of the characteristic polynomial of the recurrence for T are the same as the ones for the characteristic polynomial (plus an additional 1, i.e. a constant). In fact, although it shouldn't necessarily be that way, in this particular problem it so happens that we can discard the additional root, and use the same recursion for T than the one we used for A (as far as I know, this happens because $T(0) = 0$):

$$T(n) = 2T(n-1) + 2T(n-2) - 2T(n-3) + T(n-4).$$

Of course, every recurrence needs its base cases. It's easy to determine that $T(1) = 1$, $T(2) = 1$, $T(3) = 4$ and $T(4) = 8$. This is a complete formula for $T(n)$, and we can start coding.

The final part of the problem is translating this recursion into something that can be computed in a reasonable time, like, say... logarithmic time? Of course, the standard way to compute these kind of recursions is to use binary exponentiation, using the transition matrix as the base. Given the problem input, $N = 10^{12}$, the solution is the first component of the vector resulting from this product:

$$S = \begin{pmatrix} 2 & 2 & -2 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}^{N-4} \begin{pmatrix} 8 \\ 4 \\ 1 \\ 1 \end{pmatrix}.$$

Of course, this is ridiculously fast. About 1,4ms in non-optimised code.

261. Pivotal Square Sums

Difficulty rating: 85 %.

Solution: 238890850232021. *Solved: Sun, 14 Nov 2021, 17:48.*

Math knowledge used: binary quadratic forms (includes convergents, Tonelli-Shanks, Hensel's Lemma), recursive successions, Faulhaber's formulas.

Programming techniques used: Alpertron (includes a prime sieve).

I can't believe I finally got to solve this. But here it is, after some very intense days of work observing patterns and carefully learn to prune, and when to use the full solver and when an easy recursive solution, I got this. I used a lot of code for this, falling into three different categories: the code for the Alpertron implementation, which I wrote about one year before solving this problem, more or less, and which I stopped using because I believed that had mistakes (my position right now is that is probably mostly correct, although duplicate solutions appear); the code for the problem itself, which is not so big; and a lot of intermediate code I used to analyse the results of the Alpertron implementation and extract patterns. The Alpertron code is definitely the biggest and most complicated subsystem I've written for Project Euler, much more so than Meissel-Lehmer and variants, but mostly I was adapting code from an existing implementation (and reverse engineering some details). Now, the analysis part was much harder because I was on my own. Patterns exist, and even if I used Alpertron on my final code, I know that I could have pressed further to find even more recursive patterns and avoid it altogether; fortunately, it wasn't necessary.

So, if a Project Euler problem has the steps of formulation, analysis and implementation, in this case the analysis used a lot of implementations itself, with feedback loops where, bit by bit, I unraveled additional patterns. Eventually the patterns where wide enough that I could rely on them and only use Alpertron for some special "base cases". And thus the problem could be solved in a comfortable 40 seconds of run time.

Ok, now for the math itself and these weird patterns. The basic formulation of the problem, given by the description, is this: find tuples of the form (k, m, n), with $m > 0$ and $k \leq n$, and which satisfy the following equality:

$$\sum_{i=k-m}^{k} i^2 = \sum_{i=n+1}^{n+m} i^2.$$

Also limited to $k \leq 10^{10}$, which is the problem limit. Simple enough. We can play with these summatories and rewrite it as

$$\sum_{i=0}^{k} i^2 - \sum_{i=0}^{k-m-1} i^2 = \sum_{i=0}^{n+m} i^2 - \sum_{i=0}^{n} i^2,$$

and then use Faulhaber's formula, $\displaystyle\sum_{i=0}^{x} i^2 = \frac{2x^3 + 3x^2 + x}{6}$. After an expansion and some simplifications, which were done a lot of time ago but which

were most probably done in Matlab, we arrive at a better equation:

$$(1+m)\,k^2 + \left(-m^2-m\right)k - mn^2 + \left(-m^2-m\right)n = 0.$$

This equation is not trivial at all. In fact, it's a pretty complicated one. It has infinite solutions, and their distribution is not at all simple. At this point, the closest equations I had a clue about were the binary forms solvable with Alpertron, but these equations are not that, since they have three variables and they are cubic. Ok, then the obvious solution is to iterate over one of them, treating it as a constant at each step and solving the resulting, simplified equation. This variable can't be k, because there would still exist cubic terms like $-m^2n$, and can't either be n because we have an mk^2 term. So it must be m. Ok, so we need to solve a binary quadratic form with the following values:

$$A = m+1, \qquad B = 0, \qquad C = -m,$$
$$D = -m^2-m, \quad E = -m^2-m, \quad F = 0.$$

There are three main questions: first, is this workable? That is, does this take a minute or ten hours? Second, which is the relationship between m and k? I need it to know when to stop iterating. Surely I don't need to solve 10^{10} equations. Third, do these solutions follow a reasonable pattern that I can take advantage of?

Now, the experiments start. After a few experiments I got a simple pattern that was good enough for most cases: start with the case $k = m$, $n = 0$ (this is not a valid solution, but it matches the equation and it can be used to kickstart the recursion), and apply the following recurrence:

$$\begin{aligned} k_{i+1} &= (2m+1)\,k_i + 2mn_i + m, \\ n_{i+1} &= (2m+2)\,k_i + (2m+1)\,n_i \quad . \end{aligned}$$

This is amazing, but unfortunately this problem would not be classified as 85 % if this was all there is to it. I tried several cases and these were the only solutions, until I tried $m = 49$ and found that there were additional solutions. After that I found several more cases, most of which seemed to be related to powerful numbers, most notably squares multiplied by small numbers (possibly 1), in such a way that $m+1$ was also a square multiplied by another small number. In the case $m = 49$, we have $m = 7^2$ and $m+1 = 2\cdot5^2$.

At this point my enthusiasm about a generic recurrence waned and I started working on an implementation of Alpertron. Fast forward a few weeks,

and I have my working implementation in Java (considering that I was sleep deprived at the time, I can't complain about the time I spent), but it's not identical to the original one, and sometime spurious (valid but duplicate) solutions appear. This made me suspect and I stopped working on this for a while. By this time 2020's Advent of Code had started and I had put Project Euler aside temporarily, and when I got back, all this Alpertron business seemed too intimidating and I still didn't have a 100 % clear idea about the pattern for the special cases (my idea was to find them using a Pell-like equation, $px^2 - qy^2 - 1 = 0$, for small values p and q; this wasn't very precise, as I learned later, and it still missed some particularly rare cases). So I just worked on different problems.

Fast forward almost a full year. I find problem 390, which can obviously be solved using Alpertron. I decide to try it and, to my surprise, it works. The equations were considerably simpler, and it manages to solve the problem, including calling the solver more than 70000 times (!), in less than one minute. This was very promising, so I decided to finally tackle 261. And so I started coding to find the cases where the basic recurrence is not valid.

My first large-scale experiment was quite successful. I identified a lot of numbers that didn't fit the base patterns, and they fell into two categories:

- Numbers of the form $m = x^2 - 1$ where x is an odd number.
- Spurious solutions fitting the weird and unclear pattern about squares multiplied by a small number. Iterating up to $m = 10^4$, I found these:
  - $m = 49 = 7^2$; $m + 1 = 2 \cdot 5^2$.
  - $m = 242 = 2 \cdot 11^2$; $m + 1 = 243 = 3^5$.
  - $m = 675 = 5^2 \cdot 3^3$; $m + 1 = 676 = 2^2 \cdot 13^2$.
  - $m = 1681 = 41^2$; $m + 1 = 1682 = 2 \cdot 29^2$.
  - $m = 2645 = 5 \cdot 23^2$; $m + 1 = 2646 = 2 \cdot 7^2 \cdot 3^3$.
  - $m = 4374 = 2 \cdot 3^7$; $m + 1 = 4375 = 7 \cdot 5^4$.
  - $m = 6727 = 7 \cdot 31^2$; $m + 1 = 6728 = 29^2 \cdot 2^3$.
- Also, for some of the $m = x^2 - 1$ I found irregularities. For most cases my Alpertron implementation found 12 solutions (for the weird solutions above there were always more, something between 18 and 36 and always a multiple of 6; more or less, 6 solutions resulted in a single valid one), but in some cases there were more than that:
  - $m = 288 = 3^2 \cdot 2^5$, $m + 1 = 289 = 17^2$ (24 solutions).

- $m = 1444 = 2^2 \cdot 19^2$; $m + 1 = 1445 = 5 \cdot 17^2$ (18 solutions).
- $m = 2400 = 3 \cdot 5^2 \cdot 2^5$; $m + 1 = 2401 = 7^4$ (24 solutions).
- $m = 9408 = 3 \cdot 7^2 \cdot 2^6$; $m + 1 = 9409 = 97^2$ (24 solutions).
- $m = 9800 = 5^2 \cdot 7^2 \cdot 2^3$; $m + 1 = 9801 = 11^2 \cdot 3^4$ (36 solutions).

The $m = x^2 - 1$ pattern was very promising, but the exceptions left me still dumbfounded. After looking at these numbers for a few minutes, I found a very interesting pattern that fit most of them:

- $m = 49 = 1 \cdot 7^2$; $m + 1 = 50 = 2 \cdot 5^2$.
- $m = 242 = 2 \cdot 11^2$; $m + 1 = 243 = 3 \cdot 9^2$.
- $m = 675 = 3 \cdot 15^2$; $m + 1 = 676 = 4 \cdot 13^2$.
- $m = 1444 = 4 \cdot 19^2$; $m + 1 = 1445 = 5 \cdot 17^2$.
- $m = 1681 = 1 \cdot 41^2$; $m + 1 = 1682 = 2 \cdot 29^2$.
- $m = 2645 = 5 \cdot 23^2$; $m + 1 = 2646 = 6 \cdot 41^2$.
- $m = 4374 = 6 \cdot 27^2$; $m + 1 = 4375 = 7 \cdot 25^2$.
- $m = 6727 = 7 \cdot 31^2$; $m + 1 = 6728 = 8 \cdot 29^2$.
- $m = 9800 = 8 \cdot 35^2$; $m + 1 = 9801 = 9 \cdot 33^2$.

Incredible! This is a reasonably simple pattern that fits *most* of these numbers, and the fact that there are two cases where the multipliers are 1 and 2 hints at a possible recursive solution. Indeed, let's make this formal: there is some multiplier a, and some numbers b and c, so that $m = ab^2$ and $m + 1 = (a + 1)\, c^2$. Well, this is another Alpertron-able binary quadratic form, right? We just need to treat a as a constant.

$$ab^2 + 1 = (a + 1)\, c^2 \Rightarrow ab^2 - (a + 1)\, c^2 + 1 = 0.$$

Good news! This equation is solved quite fast, and the solutions do follow a simple pattern, one for which I haven't found any exception. There is a base trivial solution, $b = 1$ and $c = 1$, and then there is a simple homogeneous recurrence:

$$\begin{aligned} b_{i+1} &= (2a + 1)\, b_i + (2a + 2)\, c_i. \\ c_{i+1} &= 2ab_i + (2a + 1)\, c_i. \end{aligned}$$

Note the similarities with the "standard" recursion! It's the same matrix, but inverted. Which is to be expected, since the equations are very similar (the main coefficients change from $(m+1, -m)$ to $(a, -(a+1))$, and instead of the linear coefficients there is a constant one). By the way, the solution $b = c = 1$ is not valid, but the first valid solution is $b = 4a+3$, $c = 4a+1$, which can be easily seen in the terms listed above. Wow, this is great! This gives us a list of terms that don't follow the recursion (so we can just call Alpertron for these values). Too bad that the list is incomplete. What happens to these terms?

- $m = 288$. Maybe $m = 2 \cdot 12^2$, $m + 1 = 17^2$?
- $m = 2400$. Maybe $m = 6 \cdot 400^2$, $m + 1 = 49^2 = 7^4$?.
- $m = 9408$. Maybe $m = 3 \cdot 56^2$, $m + 1 = 97^2$?.

All the squares in the $m + 1$ side are of the form $16x + 1$ for some x, does that count? Maybe so, but that's not the pattern I found. My next stop was to solve the equation for some $m = x^2 - 1$ (instead of for all m like I did on my first large-scale experiment), noting the cases where I got an unexpected result (i.e. my Alpertron gives me more than 12 solutions). Since x needed to be odd, I also considered $x = 2i + 1$ and noted the values of i. These were my results:

- $m = 288$; $x = 17$; $i = 9$.
- $m = 2400$; $x = 49$; $i = 25$.
- $m = 9408$; $x = 97$; $i = 49$.
- $m = 9800$; $x = 99$; $i = 50$.
- $m = 25920$; $x = 161$; $i = 81$.
- $m = 58080$; $x = 241$; $i = 121$.
- $m = 113568$; $x = 337$; $i = 169$.
- $m = 201600$; $x = 449$; $i = 225$.
- $m = 235224$; $x = 485$; $i = 243$.

Fuck! That's it. The values of i are *exactly* the values of $m+1$ for all the cases where there is at least one nontrivial solution. Oh, and yes, multiples of 16 plus one do appear, since if i is an odd square, say $i = (2j + 1)^2$ (I'm starting

to run out of letters for variables...), then $i^2 - 1$ is $16T_j$, where $T_j = \sum_{q=1}^{j} q$ is the jth triangular number.

Ok, this gave me a lot of ammunition. But now, after having separated the "special" cases for $m = x^2 - 1$, the "normal" ones all have the same set of 12 Alpertron solutions, indicating two recurring solutions (as opposed to a single one in the normal m values), so why not try to find a pattern for these solutions? I looked for it, and I found these base cases:

- $m = 8$: $k = 28 = 7 \cdot 4$, $n = 21 = 7 \cdot 3$.
- $m = 24$: $k = 132 = 22 \cdot 6$, $n = 110 = 22 \cdot 5$.
- $m = 48$: $k = 360 = 45 \cdot 8$, $n = 315 = 45 \cdot 7$.
- $m = 80$: $k = 760 = 76 \cdot 10$, $n = 684 = 76 \cdot 9$.

Ok! It seems that this base solution always follows this pattern. Given $m = x^2 - 1$ and $x = 2i + 1$ (therefore $m = 4i\,(i + 1)$):

$$\begin{aligned} k &= (m - i)\,(2i + 2) = (4i^2 + 3i)\,(2i + 2) = 8i^3 + 14i^2 + 6i, \\ n &= (m - i)\,(2i + 1) = (4i^2 + 3i)\,(2i + 1) = 8i^3 + 10i^2 + 3i. \end{aligned}$$

All right! This base solution is not valid since $k > n$ for all $i > 0$, but we can use it to iterate, using the exact same recursion as in the normal case (well, it comes from the same equation, so it's natural that it follows the same recursion). And now I finally have a complete scheme that works reasonably well, only needing to call Alpertron a handful of times. Here is the algorithm, at a high level. It keeps a set of valid values of k, a set of already passed values of m, and a set of special values of i (sorted by the associated value of a):

1. Iterate for integer values of a, starting from $a = 1$. Finish when a given value of a doesn't return any valid m.
    a) Use the recursion to get values of b and c.
    b) Iterate for every value of $m = ab^2$, finishing when a given m doesn't return any valid k.
        1) Solve for this value of m using Alpertron. Since the smallest solution always comes from $k = 2m\,(m - 1)$, we can return early if this value is above the limit of 10^{10}.
        2) Store all the valid values of k for this m.
        3) Store this m in the "already passed m" set.

4) Store $i = m+1$ in the "special i" set, indexing by the current a.

2. Store all the odd squares of the form $i = (2j+1)^2$ for $j \geq 1$ in the "special i" set. You can use 0 as an index. Although not so many are needed, I iterated until I found $i > 10^{10}$.

3. Now, iterate over all the values of i in the "special i" set. If any value of i doesn't return any valid k, advance to the next value of a.

   a) Use $m = 4i\,(i-1)$.

   b) Solve for this value of m using Alpertron. Again, stop early if m is too high.

   c) Store all the valid values of k for this m.

   d) Store this m in the "already passed m" set.

4. Now we are going to iterate over "standard" squares. So, iterate for values of i starting from $i = 1$, and stopping when a value doesn't result any valid k.

   a) Skip the current value of i if it was in the "special i" set.

   b) Use $m = 4i\,(i-1)$.

   c) Use both the "normal" and "special" recursions to calculate values of k.

   d) Store all the valid values of k for this m.

   e) Store this m in the "already passed m" set.

5. Finally, the "normal" iteration. Use all integer values of m, starting from 1 and ending when there isn't any valid value of k.

   a) Skip the current value of m if it was in the "already passed m" set.

   b) Use the "normal" recursion to calculate values of k.

   c) Store all the valid values of k for this m.

6. Finally! Sum all the values in the set of valid values of k to get the result. And yes, there are some repeated values, although they are very rare. $k = 684$ is found for $m = 4$ (with $n = 760$) and once again for $m = 18$ (with $n = k = 684$).

I could use longs for most of the algorithm. The only part requiring BigIntegers was the Alpertron implementation itself. I spent a couple additional hours because of a very silly mistake (I started iterating one of the loops from 2, but I needed the 1 as well, so I missed some solutions), but in the end I got the correct result, with a satisfying run time of 40 seconds, which is much less brute-forcey than I feared. I only needed to call Alpertron for 23 different values, ranging from 49 to 59535.

264. Triangle Centres

Difficulty rating: 85 %.

Solution: 2816417.1055. *Solved: Sat, 20 Nov 2021, 17:07.*

Math knowledge used: Euler line, cosine theorem, Hermite-Serret algorithm, "first prime" Erathostenes sieve (for divisor generation), square sum enumeration, Pythagoras theorem.

Programming techniques used: dynamic programming, binary exponentiation.

For an 85 %, this problem is *considerably* easier than titans like 261 or 275. In fact, my first approach worked on the first time, and even if not optimised, it takes somewhere around two minutes, which is not optimal but it's still very good, and still far below the hours or even days that I got for some really difficult problems.

The first thing to notice is that, if the circumcentre is at $(0, 0)$ and the orthocentre is at $(0, 5)$, then the Euler line is between both, so the barycentre is in the same line, at twice the distance from the orthocentre than to the circumcentre. This means that the barycentre is located exactly at $\left(\frac{5}{3}, 0\right)$, and this is all we need the orthocentre for. This position of the barycentre means that, if the triangle vertices are (a_x, a_y), (b_x, b_y) and (c_x, c_y), then the following relationships hold:

$$\begin{aligned} a_x + b_x + c_x &= 5, \\ a_y + b_y + c_y &= 0. \end{aligned}$$

Ok, this is great, but we still have four degrees of freedom, which is not good to iterate. So let's look at the other piece of information, the circumcentre at $(0, 0)$. This means that the three vertices are in a circumference with common radius, centred in the origin. This is actually even more useful, but using it is not so straightforward. Trying to find an expression of the circumcentre as a function of the vertices' coordinates, and using Alpertron or some other

magic, is very tempting. But it's not a good approach: the resulting equations are hellish. So, instead of solving a big equation system, we start from the circumcenter and find valid sets of vertices. As weird as it might sound, we are going to iterate over the circumference radius. In fact, we are going to iterate over the square of the circumference radius, which is trivially always an integer. For each value of r^2, we will find valid vertices, i.e. points which verify $x^2 + y^2 = r^2$, and from these sets, we will choose those which verify the barycentre conditions stated above.

This approach presents several questions. The most important one is: when do we stop iterating? And the second most important one is, probably, how do we efficiently enumerate vertices that fall both in the circumference and in lattice points? That's kind of the hardest part of this problem, which is still not so hard.

First, about the iteration stop point. Let's look at the bounds of the quotient perimeter/radius. The most "compact" triangle would be an equilateral triangle. We can determine the side of the triangle, a, with the cosine theorem:

$$a^2 = r^2 + r^2 - 2r^2 \cos \frac{2\pi}{3} = 3r^2 \Rightarrow a = \sqrt{3}r.$$

So the perimeter would be $P = 3\sqrt{3}r$. This implies a relationship $r^2 = \frac{P^2}{27}$ for the most compact triangle. Now, the least compact one would be two horizontal lines over the X axis, which an infinitesimal vertical side at either $(-r, 0)$ or $(r, 0)$. This implies a perimeter equal to $4r$, so the relationship would be $r^2 = \frac{P^2}{16}$ for this "triangle". This means that we have the following bounds for the value of r^2, which is the index of our main loop:

$$\frac{P^2}{27} \leq r^2 \leq \frac{P^2}{16}.$$

A few experiments confirm that the first bound is much more accurate for bigger triangles than the second one. In fact, we can safely choose 20 instead of 16 as the denominator; therefore, for the given limit $N = 10^5$, we "just" need to iterate for $r^2 \leq \frac{10^{10}}{20} = 5 \cdot 10^8$. In fact, the biggest r^2 is considerably lower: 343040465. Using this as a limit reduces the run time by around one third (from 117 seconds to 77), but it's kind of cheating, so I prefer using the other limit, which presumes that I don't yet know the solution of the problem :P. By the way, a condition that I did use is that n must be odd and a multiple of five, i.e $n \equiv 5 \pmod{10}$. This saves a lot of time, actually.

Very well. We will iterate up to $r^2 = 5 \cdot 10^8$. But what do we do at each iteration? We need to enumerate all the lattice points in such circumferences. Obviously we need to do it in very little time in order to have a semblance of efficiency. So we are going to use dynamic programming. For a given $n = r^2$, the first thing we do is decomposing n in prime factors (yes, this means a "first prime" sieve and the usage of my old DivisorHolder class). There are several situations:

- If the decomposition includes a primer factor of the form $4k + 3$ with an odd exponent, then the decomposition is impossible.
- If n is a prime number of the form $4k+1$, we use the Hermite-Serret algorithm, which is a surprisingly simple mix of modular exponentiation and Euclid's algorithm. This Hermite-Serret will find a single decomposition of the form $a^2 + b^2 = n$ with $a < b$.
- If $n = p^2$ for some prime $p = 4k + 3$, the only valid decomposition is $0^2 + p^2 = n$.
- If $n = 2^k$ for some $k > 1$, the only valid solution is $0^2 + \left(2^{k/2}\right)^2 = 2^k$ if k is even, or $\left(2^{(k-1)/2}\right)^2 + \left(2^{(k-1)/2}\right)^2 = 2^k$ if k is odd. This is not necessary, however, since even numbers are not actually used.
- In any other case, including cases like p^n for $p = 4k + 1$ or p^{2n} for $p = 4k+3$, we can decompose $n = n_1 \cdot n_2$ where n_1 and n_2 are values we have already decomposed. In this case, we iterate over the decompositions, so that if $n_1 = a^2 + b^2$ and $n_2 = c^2 + d^2$, we can do

  $$n = (ac + bd)^2 + (ad - bc)^2 .$$

  However, some additional care must be taken, because this might skip some solutions. This is mitigated by using the same formula again, but switching a and b. After doing this with all the valid pairs for n_1 and n_2, and taking care of possible duplicates (this happens, for example, when combining the solutions for p and p^2 to get the ones for p^3), we will get all the solutions for n. I believe that here there is room for a finer approach than mine, but my program is still quite fast even if not below the one minute rule.

Ok, so let's say that we are iterating for a given $n = r^2$, and we have a list of pairs (x_i, y_i) verifying $x_i^2 = y_i^2 = n$. This is a "base" list, which is convenient for the dynamic programming approach, but we need a "full" list including the negative signs and position swaps (i.e. for $n = 5 = 1^2 + 2^2$

we need both $(1, 2)$ and $(2, 1)$, and all the other 6 combinations that appear when introduce signs). So the first thing we will do is "unroll" this list of vertices, and when we have the full list, we will sort using the value of y (I could have used x, but y is slightly simpler because its equation includes a 0 instead of a 5). Then, we do the following to ensure that we find every solution exactly once:

- Iterate the list, sorted by y. We get a vertex a, and we are interested only in negative values of a, because we are going to iterate in such a way that $a_y \leq b_y \leq c_y$. Therefore, stop when we find $a_y \geq 0$.
- We want b_y to be bigger than a_y, so we iterate the same list or vertices, but starting from an index right above the one that gave us a. Now, we need $c_y = -a_y - b_y$, and at the same time, $b_y \leq c_y$. This means that $b_y \leq -\frac{a_y}{2}$. So we will halt the iteration of b as soon as we find a value of y above that limit.
- Ok, we have a and b. Now we look for a vertex c that satisfies $c_y = -a_y - b_y$, and $c_x = -a_x - b_x$. Some additional care is needed here in case c_y and b_y are the same (sorting additionally by x after y, and ensuring that $c_x > b_x$ if $c_y = b_y$, is enough). If we have found such a vertex, good news! Calculate the perimeter using Pythagoras theorem, and if it's below the limit, add it to the accumulator.

That's it. The difficult part is in enumerating the lattice points, which I didn't quite nail since I get duplicates, but anyway the solution is quite good and I believe that the best solutions from the problem thread are pretty close to this. Some people apparently missed the Hermite-Serret algorithm or the Euler line, and they had really slow brute force implementations that took hours.

If every 85 % was as easy as this, it would be even a bit disappointing. I would have rated this as somewhere between 50 % or 60 %.

283. Integer sided triangles for which the area/perimeter ratio is integral

Difficulty rating: 75 %.

Solution: 28038042525570324. *Solved: Sun, 21 Nov 2021, 07:40.*

Math knowledge used: Heronian triangles, "first prime" Erathostenes sieve (for divisor generation).

Programming techniques used: memoisation.

I don't consider this cheating at all, but yeah: there is a paper which basically gives away the solution. *Heronian triangles Whose Areas Are Integer Multiples of Their Perimeters*, by Lubomir Markov. It's a small paper and it's pretty direct, with a bunch of small formulas in the second page, under *Theorem 2*, which basically explains the solution step by step. Now, the solution proposed in the paper is mathematical, and we are interested in a more algorithmic approach, which takes advantage of recalculations and so on. So, this is it:

1. Let N be the limit of the problem, $N = 1000$ in our case.
2. We need a "last prime sieve" up to $16N^2$, actually pretty small.
3. From this sieve we are going to cache the decompositions of every number under $2N$, which we will use frequently. We could theoretically cache some others, but they aren't used so often. Although maybe there is a bit of time to be gained in exchange for more memory if all the decompositions are cached.
4. We also need a sieve of "V values". Given each value $u \in [1, 2N]$, we need a list of values in the range $\left[1, \left\lfloor \sqrt{3}u \right\rfloor\right]$ which are coprime with u. Superfast to find with a sieve, now that we have the decomposition of u from the previous step.
5. Now, iterate over all the values $m \in [1, N]$. For each m, iterate over every u that divides $2m$ (easily acquired through the decompositions we have), and for each u, iterate over every v in the "V values" cache for u. For each (u, v) pair we do:
   a) Decompose $M = 4m^2\,(u^2 + v^2)$. This is easily done: get the decomposition of $2m$, duplicate the prime powers to get the decomposition of $4m^2$, and then decompose $u^2 + v^2$ using the prime sieve and combine the results.
   b) Now, let $d_1 \cdot d_2$ be a decomposition of M. We find it easily by enumerating every divisor of M, d_1, and letting $d_2 = \dfrac{M}{d_1}$. Discard the cases where $d_1 > d_2$. We need, again, to iterate over every such decomposition. We also need the quotients $q_1 = \dfrac{d_1 + 2mu}{v}$ and $q_2 = \dfrac{d_2 + 2mu}{v}$ to be integers. For each one of these cases, we find a valid triangle:

$$a \;=\; q_1 + \frac{2mv}{u},$$

$$\begin{aligned} b &= q_2 + \frac{2mv}{u}, \\ c &= q_1 + q_2. \end{aligned}$$

Sometimes there are duplicates, so discard also the cases where $b > c$ (we would also want to discard $a > b$, but this is done automatically since $d_1 \leq d_2$). This condition is equivalent to $d_1 < v^2 \frac{2m}{u} - 2mu$.

c) That's it. No need to rescale, since the rescaled triangles are also going to be generated by this iterative process (that is, if $m = 1$ generates $a = 5$, $b = 12$, $c = 13$, then $m = 2$ generates $a = 10$, $b = 24$, $c = 26$, and it's not necessary to rescale manually). Add all the perimeters to get the result.

The code is roughly $O(N^2)$, or probably $O(N^2 \log^a n)$ for some low integer a, with a relatively high constant, meaning that the result for $N = 1000$ takes about four times the time for $N = 500$ (a bit more, actually), but it's still done in time (18 seconds for the proposed $N = 1000$).

This would be quite hellish without the paper. Apparently there is a result in Wolfram Alpha that gives some insight for Heronian triangles with *rational* quotients, but this is not enough, it might be complicated to iterate over, and apparently has some subtle mistakes. By the way: a Heronian triangle is just any triangle with rational perimeter and rational area, so of course the quotient is also a rational number, but this is a much wider set than the one we need for this problem.

292. Pythagorean Polygons

Difficulty rating: 65 %.

Solution: 3600060866. *Solved: Mon, 22 Nov 2021, 08:20.*

Math knowledge used: Pythagorean triples, Faulhaber formulas.

Programming techniques used: dynamic programming, meet-in-the-middle.

I say often that a typical Project Euler problem has three steps: formulation, analysis and coding. Here, as it is often the case, the formulation is given to us, and it's pretty simple. Usually there is a big amount of analysis, maybe with the usage of some arcane but specifically useful theorem, to reduce the formulation into something simple, and then the code has some additional complication. Here, the analysis stage is actually kind of short,

and most of the meat is in the coding, and in being careful about structures, pruning and so on.

My solution to this problem can be separated into four parts. First, we determine which segments can be a side of one of these "Pythagorean Polygons". Then, we build concatenations of these sides, forming each possible "quadrant". The third step consists of gluing these quadrants (somewhere between 0 and 4 of them) into incomplete polygons. Finally we add horizontal and vertical lines, as a glue of sorts, and the complete polygons are generated. Most of the operations, especially the later ones, can be done in bulk, so with a bit of pruning we can find all the cases very quickly. There isn't even a need of enumerating them one by one, since they come in groups. There are four key observations to be made:

- If we abstract every non-horizontal and non-vertical as an oblique broken line, a convex polygon is composed of at most 8 lines, like an octagon. Some of these might be missing, but there must be at least two of them.
- As a consequence of the previous point, there are four groups of non-horizontal or vertical segments, and in these groups, the slopes of the sides must follow certain ordering (basically, increasing the angle with respect the positive horizontal axis) in order to keep the polygon convex.
- There is a huge amount of symmetry to be exploited: not only in the generation of oblique broken lines, but also in the final enumeration.
- The horizontal and vertical lines can be added at the end, when the set of oblique sides is already determined.

OK, this is not the full story, but that's most of it. The remaining logic is more about the accidental complexity of having to prune carefully and so on. There are a few extra considerations to be made, and I will mention them as I explain the full procedure.

The first thing to do is to generate Pythagorean triples. We do this with the standard Pythagorean triple generation procedure; yes, the same one I've used everywhere. Given the limit of the problem, $N = 120$, a conservative upper bound for the maximum side length is $M = \frac{N}{\sqrt{2}} \approx 85$, so I only need to generate hypotenuses as big as M. There aren't many of them with such a low limit, and they can be generated almost sequentially, so this is ridiculously fast. Now, each of these can generate a few sides. If we have a triple with sides a, b and c, we will generate segments with

slope $\frac{b}{a}$, and also segments with slope $\frac{a}{b}$. For each side, we need to store the horizontal increment, the vertical increment, and the total length, and each side will be indexed by a slope. So, with this triangle $\{a, b, c\}$, we generate sides with $\{\Delta x = a, \Delta y = b, l = c\}$, $\{\Delta x = 2a, \Delta y = 2b, l = 2c\}$, $\{\Delta x = 3a, \Delta y = 3b, l = 3c\}$, and so on, until we hit $l > M$, which is too long and we can discard them; these sides will be filed under the slope $\frac{b}{a}$. But we are also going to need to generate the triangles $\{\Delta x = b, \Delta y = a, l = c\}$, $\{\Delta x = 2b, \Delta y = 2a, l = 2c\}$, $\{\Delta x = 3b, \Delta y = 3a, l = 3c\}$, and so on, filed under the slope $\frac{a}{b}$. There are just 16 Pythagorean triangles below the limit, so we have 32 groups of segments. So ends the first step.

Now we are going to get all the combinations of these segments that might be useful for us. That is, we need these combinations to have at most a segment from each slope, and the end point must be at a distance lower than M from the origin, so that the end can be reconnected with the origin without exceeding the maximum length (again, this is a conservative bound). We can generate all the combinations recursively, iterating from all the slopes, and moving from each slope to the higher ones. It's super fast and there aren't that many combinations anyway. Now, we will need an interesting structure. The final set of segment groups will be indexed first by total length (sorted!), and then by position (unsorted, although I wonder if we could so something clever here). I used a class named `BrokenLineCollection` which is basically this map, but with a ton of operations. For each length, and each position, I have an object of a class called `ConnectionSummary`, which has three fields: one, the total amount of connections from the origin to the given end point; two, a boolean indicating that this connection consists of a single side; and three, a counter indicating how many of these connections consist of a single line from the origin to the end point and another one from the end point to the origin, again. This doesn't make much sense now, and indeed this field will always be 0 at this stage, but we have a lot to do with this class, so this field will be useful.

Third stage! This map right now represents all the possibilities of a quadrant (not counting horizontal and vertical lines), and thanks to symmetry, we now that the other three quadrants have the exact same set of possibilities, just rotated. So, we combine the quadrants! Wait, there are a lot of possibilities! Each quadrant can be present or not, so there are $2^4 = 16$ possibilities! The symmetries will come in really handy right now. There are only six cases, actually:

- No quadrants. That is, no oblique lines. Only horizontal and vertical lines. In other word, rectangles! Of course we need to take these into account.
- Just one quadrant, First, second, third, fourth, we don't care. One line from the origin to the endpoint, and then a vertical and a horizontal line to connect back to the origin. This covers 4 of the 16 cases.
- Two consecutive quadrants! First and second, second and third, third and fourth, fourth and first one again. Another 4 possibilities. In this case there is a chance that we will only need a single long horizontal or vertical line to go back to the origin, but in most cases we will also need a short one in the other direction.
- Two non-consecutive quadrants! Either first and third, or second and fourth. We could get really close to the origin, since one side "steps back" from the other. We might even actually go back to the origin! In fact... if the oblique portions are each a single side, with the same length and the same slope, we get a fake polygon with no area, which we must not count! This is why we need this strange counter I mentioned above.
- Three quadrants! Again, 4 cases. Like three quarters of a circumference.
- All the quadrants! Oh, one single case. We might go back exactly to the origin. We might not. We don't have enough information yet.

Ok, how do we generate the combinations? It's as simple as it gets: combine all with all. Given a length and a position from the first quadrant, and a length and a position from the second quadrant, just add them. The connection summary is the product of both of these. If the sum of positions is the origin (this can only happen when adding the first and third quadrant, but it CAN happen), and each summary had a single line, we must add one to the counter of negated lines. Otherwise, for every combination of length and position that ends in the same point, just combine the connection summaries by adding all the counters. It's not possible to have a single line, since each combination has at least two segments now.

We need to apply this combination process several times. Aside from the starting object with the first quadrant information, we will need the combination of quadrants 1 and 2, which we will use to get the combinations of quadrants 1, 2 and 3, and with this, a combination with all the four quadrants. Finally we will need a combination of the first and third quadrant, and that's it. It's faster than it sounds because, at each step, if the current

total length is L and the end point position is (x, y), we can discard the result if $L + \left\lceil \sqrt{x^2 + y^2} \right\rceil > N$.

This is a bit bruteforcish, but it works. Now, the final stage: adding the horizontal and vertical lines. Let's say that after adding all the four quadrants we end at a point like $(-5, 4)$. We need to add a horizontal line of length 5 and a vertical line of length 4. Where? Well. The horizontal line goes either between the second and third quadrants (OK... in the usual geometrical terms, this means at the top of the y axis, i.e. between the first and second "standard" quadrant. Sorry for the confusion, but I'm not going to use these terms), which would push the end point further to the left, or between the fourth and the first one, pushing it to the right. So it must be in this latter case, so that we end at $(0, 4)$. As for the vertical line, it's more or less the same. Putting it between the first and second quadrant would move the end point upwards, and putting it between the third and fourth will move it downwards. We chose again the latter option, and we end at $(0, 0)$ at the cost of an additional length of 9. It can be seen clearly that there is only a way to add these horizontal and vertical "minimum" lines, but there is more. We can add an additional horizontal segment of length x to both the spaces between the second and third quadrant, and the fourth and first. These will cancel each other, so that the end point is again the origin. We can also do the same for vertical lines. Interesting. So, if the "smallest" possible polygon created using this set oblique sides has a total length of, say, 106, we can do this a total of 7 times, since each operation adds 2 to the total length, and $\frac{120 - 106}{2} = 7$. We can choose any amount of additions between 0 and 7, and for each amount, i, the possibilities are

$$CR_{2,i} = \binom{2 + i - 1}{i} = \binom{i + 1}{i} = \binom{i + 1}{1} = i + 1.$$

So, if we have up to x pairs of segments to add, the possibilities are given by Faulhaber's most basic formula:

$$\sum_{i=0}^{x} i + 1 = \sum_{i=1}^{x+1} i = \frac{(x + 1)(x + 2)}{2}.$$

We can abstract this with a function $f(x)$ that, given the total length of a "complete" polygon (i.e. after adding the minimum horizontal and vertical lines), having a length of x, returns the total combinations. It's just

$$f(x) = \frac{\left(\left\lfloor \frac{N-x}{2} \right\rfloor + 1\right)\left(\left\lfloor \frac{N-x}{2} \right\rfloor + 2\right)}{2}.$$

This gives us a complete algorithm to calculate the total amount of polygons in each subclass. Hooray!

- Iterate for every one of the six cases above.
- The case "no quadrants" is special. We actually have a minimum of length 4 (the smallest square of side 1), so the total amount is $f(4)$.
- For every other case, we proceed as follows. Iterate over all the cases, and for each one, add the minimum horizontal and vertical lengths. Then, if the total length is x, add a total of $f(x)$ to this group's result, multiplied times the cardinality of this connection summary object. Then, after this multiplication, subtract the counter of "negated" lines (in these cases, we have to subtract the base case without horizontal or vertical lines, but all the rest still count!).
- After having summed all the connections in each one of the six groups (or five, not counting the rectangles case), multiply times the symmetry factor (4 for the "consecutive" quadrants, 2 for the "opposite" quadrants, 1 for the "all quadrants").
- The sum of all this is the result of the problem.

It sounds more complicated than it actually is! I got it at my first try, and it makes me happy because I expected to have some missing cases or miscounting somehow, but no.

295. Lenticular holes

Difficulty rating: 75 %.
Solution: 4884650818. *Solved: 25 Nov 2021, 07:03.*
Math knowledge used: basic geometry, combinatorics, modular inverses, Bézout's Lemma, diophantine equations, inclusion-exclusion.
Programming techniques used: dynamic programming.

This is notably harder than other problems I've solved recently, like 292 or 385. It has a way to introduce small roadblocks everywhere. I expected to need Alpertron, but no, the only diophantine equations I needed to solve were simple, linear ones with two variables.

First, let's go to the meat. Study the characteristics of a lenticular hole. Are there different kinds, or something like that? Since the lenticular hole is delimited by two lattice points, the best starting point is the segment between these two points. These segments can't be vertical or horizontal, because in

that case they either contain a lattice point, thus invalidating the hole, or they come from a circle whose centre is not a lattice point, in whose case it also falls outside the definition. So, they are always oblique. Fantastic. Since a circle is a highly symmetrical figure, we can rotate these holes as much as we want. Because of this, I chose a reference frame where the segment is created in the positive direction, like moving from the first quadrant into the second one. That is: the segment starts in a point (x, y), and finishes in $(x - a, y + b)$ for some positive numbers a and b. This is enough to establish useful restrictions on these values, since these points must lie on a lattice-centred circumference. To make it as simple as possible, I fix $(0, 0)$ as the centre of the circle, which is actually enough to characterise all the circles we need, since we don't need to study the relative positions of the two circles. So, the first equation we have is:

$$R^2 = x^2 + y^2 = (x - a)^2 + (y + b)^2 \Rightarrow -2ax + 2by + a^2 + b^2 = 0.$$

This is already very informative. At the very least, we can see that a and b must be both odd, and coprime. First of all, we note that if they are not coprime, the segment contains lattice points, so it can't be used. This already discards the possibility that they are both even, since in that case they are obviously not coprime. But they can't also be one even and other odd. This is apparent from the equation itself, since $a^2 + b^2$ must be even and so a and b must have the same parity; but also, it makes sense because if one of (a, b) is even and the other is odd, then it means that one of $\{(x, y), (x - a, y + b)\}$ has an odd component and an even component, while the other has either two odds or two evens. The first implies an odd R^2, and the second one implies an even R^2, so they can't both happen at the same time.

Perfect! Since both a and b are odd, this means that a^2+b^2 is even. Which means that $\dfrac{a^2 + b^2}{2}$ is whole, and we can reduce the original equation to

$$ax - by = \frac{a^2 + b^2}{2}.$$

Since a and b are coprime, Bézout's lemma assures to us that this equation always has solutions, and in particular they are of the form

$$\begin{aligned} x &= bk + x_0, \\ y &= ak + y_0. \end{aligned}$$

We are interested in all the positive solutions, so we will find the smallest positive y_0, which guarantees that x_0 will be positive, since it will be the sum

of two positive numbers. We have

$$by \equiv -\frac{a^2+b^2}{2} \pmod{a},$$

therefore

$$y \equiv -b^{-1}\frac{a^2+b^2}{2} \pmod{a},$$

where b^{-1} is the inverse of b modulo a. Now, if we define

$$c = \frac{a^2+b^2}{2} \%a,$$

then we have

$$-\frac{a^2+b^2}{2} \equiv a - c \pmod{a}.$$

This means that we can calculate the smallest y as

$$y_0 = \left(b^{-1}\,(a-c)\right) \%a.$$

From the original equation we then get that

$$x_0 = \frac{1}{a}\left(\frac{a^2+b^2}{2} + by_0\right),$$

which is guaranteed to be an integer number. Ok, neat, we have the full list of solutions, all right? Not so fast. Not every value of k results in a valid solution, and now is where we need to start being very careful. Given some coordinates $x_k = bk + x_0$ and $y_k = ak + y_0$, we know that both (x_k, y_k) and $(x_k - a, y_k + b)$ lie in the same circumference of radius $R_k^2 = x_k^2 + y_k^2$, but this is not enough. The portion of the circle that falls between this chord and the circumference must not contain any lattice point, so we need to cull some values of k. So we obviously need to look for points that are immediately to the right of the chord, and make sure that they fall outside the circle. There is also an additional condition, a not so obvious one: the coordinate $x_k - a$ must be nonnegative, because otherwise we can be sure that $(0, y_k + b)$ will be inside the circle. By plugging this condition into the formula for x_k, we arrive to our first condition:

$$bk + x_0 - a \geq 0 \Rightarrow k \geq \left\lceil \frac{a - x_0}{b} \right\rceil.$$

This is not enough, though. This ensures that there is no lattice point inside the circle to the right of $(x_k - a, y_k + b)$, and by definition there can't be any to the right of (x_k, y_k), but we need to checks all the horizontal lines between y_k and $y_k + b$ (not included). Of course, you can also do this by checking the vertical lines between $x_k - a$ and x_k, but because of symmetry we can restrict ourselves to the cases $a \geq b$ (which, when including coprimality, translates to $a = b = 1$ plus $a > b$), it makes a bit more sense to check the vertical lines, which will be fewer. Or at least this was my initial reasoning.

Ok, easy task, right? The segment going from (x_k, y_k) to $(x_k - a, y_k + b)$ intersects each line $y = y_k + q$ in the points $\left(x_k - q\frac{a}{b}, y_k + q\right)$ for each value $q \in \{1, 2, \ldots, b-1\}$. So we can say that, given any of these vertical coordinates q, the closest lattice point which we don't want to have inside the circle is given by $(x_k + p, y_k + q)$ where $p = \left\lceil -q\frac{a}{b} \right\rceil$. Therefore, for each one of these lines, we get a new equation:

$$(x_k + p)^2 + (y_k + q)^2 \geq x_k^2 + y_k^2 = R_k^2.$$

Here, q will always be positive and p will always be negative (or maybe 0, but that doesn't actually happen since the slope is closer to the horizontal axis than to the vertical one). And yes, we need to use $\geq$ and not $>$ because the problem description says that there must not be any lattice point in the *interior* of the lenticular hole. Now, if we unroll these squares and conveniently remove the x_k^2 and y_k^2, we get this:

$$2x_k p + p^2 + 2y_k q + q^2 \geq 0,$$

which sounds like it's kind of easy to be fulfilled since we are pitting three positive numbers against a single negative one. In any case we want bounds for k, so the next step is to expand the formulas for x_k and y_k:

$$2\left(bk + x_0\right)p + p^2 + 2\left(ak + y_0\right)q + q^2 \geq 0.$$

Now we group the coefficients of k and the rest, giving

$$2\left(pb + qa\right)k \geq -p\left(2x_0 + p\right) - q\left(2y_0 + q\right).$$

Now, we need to handle signs with care. We know that $p = \left\lceil -q\frac{a}{b} \right\rceil$, and since a and b are coprimes, and $q < b$, the number inside the ceiling operator is not an integer. This means that $p \gneq -q\frac{a}{b}$, and since b is positive, we can do $pb > -qa$, and therefore $pb + qa > 0$. Hooray! So we can divide the inequation

by it, resulting in

$$k \geq \frac{-p\,(2x_0 + p) - q\,(2y_0 + q)}{2\,(pb + qa)}.$$

In purely integer terms, this means that

$$k \geq \left\lceil \frac{-p\,(2x_0 + p) - q\,(2y_0 + q)}{2\,(pb + qa)} \right\rceil,$$

and now we get a single lower bound for k for each value $q \in \{1, \ldots, b-1\}$ (no, we don't get any lower bound when $b = 1$; but we still need to apply the first condition we found). Clearly we are going for the most restrictive of all these bounds, since all of them must hold, but we would like to use some finesse, trying to get this most restrictive bound without iterating over all of them (I initially thought that we might need very big values of b. I was wrong. It turns out that the biggest complexity of this problem happens elsewhere. It's nice to get the right bound with a single calculation, though). Now, obviously this most restrictive value of k will happen when the lattice point is very close to the segment. And! basic modular arithmetic tells us that the values of $(-a, -2a, \ldots, -(b-1)\,a)$ modulo b are a reordenation of $(-1, -2, \ldots, -(b-1))$ modulo b. The closest point must be the one where the difference is smallest, and we know that there is a -1 out there! So we get $Q = a^{-1} \pmod{b}$, and $P = \left\lceil -Q\frac{a}{b} \right\rceil$. These are the optimal (that is, closest) values of p and q, so we plug them into the formula for the lower bound of k, and yes, that's the right result.

Summary of the bounds so far: we have a sequence of points given by $\{(x_k = bk + x_0, y_k = ak + y_0) : k \geq 0\}$, and we have some lower bounds for k (the combination of two conditions if $b > 1$, or a single one if $b = 1$) that guarantee that there aren't any lattice points between the segment that goes from (x_k, y_k) to $(x_k - a, y_k + b)$ and the circle centred in $(0, 0)$ and with radius $R_k^2 = x_k^2 + y_k^2$, which passes by both points. Good job (especially since we can find this in a time of the order of $(\log b)$, given the modular arithmetic involved). And what's the *upper* bound? Well, we do have an upper bound for R given by the problem: $N = 10^5$. And we can express R_k in terms of k, of course. The solution is clear:

$$(bk + x_0)^2 + (ak + y_0)^2 \leq N^2 \Rightarrow \left(a^2 + b^2\right) k^2 + 2\,(bx_0 + ay_0)\,k + x_0^2 + y_0^2 - N^2 > 0.$$

Simple: we need to solve a second degree equation. There will be two solutions, and since x_0 and y_0 should be pretty small, one of them is negative,

meaning that we only need to worry about the other one. Which means that:

$$\begin{aligned} A &= a^2 + b^2, \\ B &= 2\left(bx_0 + ay_0\right), \\ C &= x_0^2 + y_0^2 - N^2, \\ k &\leq \left\lceil \frac{-B + \sqrt{B^2 - 4AC}}{2} \right\rceil. \end{aligned}$$

And that's it. We have a lower and an upper bound, and this is exactly what we need for this problem. Given a pair of odd coprime values (a, b) with $a > b$ (this is just to guarantee uniqueness), we have a range of valid k values and the values of x_k, y_k and R_k associated with them. We are still far from finishing, though: this is the more complicated mathematical analysis, but the actually difficult part comes now! Let's go back to the problem definition. We need to find compatible pairs of radii. Now, of course all the radii given by the same pair (a, b) are compatible between them, and we don't need to worry about lattice points thanks to the lattice calculations above: each lenticular hole will have two "bellies", one per circle involved, and the calculations are done to guarantee that these "bellies" don't have any lattice point. This looks good, but there is a problem: we need to remove duplicates. There are pairs of circles that appear in two or more of these sets. We have advanced a lot, but now we need to use a different formalisation. If we treat the procedure above as a black box that gives us a lot of radii given a and b, we can now define the set of such radii as $S_{a,b}$. These sets are finite and not so big (the biggest one, obviously for $a = b = 1$, has fewer than $N = 10^5$ elements, and the others get smaller pretty fast), but there are a few of them, and we need to calculate all the possible intersections. What's worse, we need to calculate the intersections of these intersections, and this is potentially exponential. Luckily, there aren't so many intersections, and there isn't any element which is inside more than 8 sets, so there isn't any *actual* exponential growth. There is still a lot of work to do, and the intersection of these sets is still a $O\left(mn\right)$ operation where m and n are the sizes (and these numbers are, sadly, too big to use something clever like BitSets). Another issue, that caught me a bit off-guard, is that the stop criterion (that is, which a or b is too big? When can I stop?) is not obvious. So, first things first: we ned to look for these bounds. Experimentally we find that:

- Given some b, the values of a that give the best chance of finding valid values are cases when $a \equiv \pm 1 \pmod{b}$. If we arrive at a case where $a \equiv -1$ (which will be found before the $+1$) and we don't find any valid

value of k, we can stop iterating over these values of a, and we move on to the next b.

- Now, for b, we can stop when we did get to such a value of a and we still didn't find any valid radius, either for that or for any other value of b.

With this, we generate a list of sets of radii, and we start intersecting them. We call the original set the first generation. The second generation is found by intersecting each set from this generation with other sets from the same generation. The third generation is found by intersecting all the sets from the second generation with the sets from the first generation (taking care to include only sets that weren't part of the original intersection, and also taking care to avoid duplicates). And so on. This is dynamic programming of sorts. And how do we calculate the result of the problem? That's easy. Given any set from the first generation, if the size is n, we can find $\frac{n\,(n+1)}{2}$ pairs (just basic combinatorics), so we add this value to the result. In the second generation, we do the same, but inclusion-exclusion tells us that now we must subtract the combination count, not add it. For the third generation we add it again, and so on. This will continue up to the eighth generation, which comprises just three sets with a single element each:

$$\{9520619225\}\,,\{8959743325\}\,,\{7691000005\}\,.$$

The original list of sets of radii is found in no time (1304 sets), but all this stuff with intersections is pretty heavy (the biggest generation is the second one, with 9642. The third one is smaller, already 5924 elements, and they get progressively smaller), and it takes about 50 seconds. But the result is correct!

In the problem thread there are some discussions about the best way to generate these intersections without much computational effort. There are some discussions about things like Stern-Brocot trees, which I didn't expect to find here, and the run times are crazy, below a second and so on. Kudos to the usual suspects for being crazy smart, I guess.

296. Angular Bisector and Tangent

Difficulty rating: 60 %.

Solution: 1137208419. *Solved: Wed, 8 Jun 2022, 02:46.*

Math knowledge used: angle bisector theorem, "first prime" Erathostenes Sieve.

Programming techniques used: none.

The geometric part of this problem requires some effort, but after that, the rest of the problem is pretty direct. Let D be the point where CE meets AB. Then, using some angle magic it can be shown that BED is the same angle than EDB, so the triangle is isosceles, that is: $BE = BD$. Now, the angle bisector theorem states that $\frac{AD}{AC} = \frac{BD}{BC}$, which means that $\frac{BE}{BC} = \frac{AD}{AC}$. With this and with the common angle at C (since the line k is a bisector), it can be shown that CEB and ADC are similar, and with this we can derive an expression of BE in terms of the triangle sides.

We start by noting that $\frac{AD}{BD} = \frac{AC}{BC}$, therefore $\frac{AD}{BD} + 1 = \frac{AC}{BC} + 1$. Now, $\frac{AD}{BD} + 1 = \frac{AD + BD}{BD} = \frac{AB}{BD} = \frac{AB}{BE}$, and so we have $\frac{AB}{BE} = \frac{AC}{BC} + 1 = \frac{AC + BC}{BC}$. So we finally arrive at $BE = \frac{AB \cdot BC}{AC + BC}$. All right.

Now we need to find a way to count triples $BC \leq AC \leq AB$ such that $\frac{AB \cdot BC}{AC + BC}$ is an integer number. Also, since this is a triangle, we must also ensure that $AB < BC + AC$. We can leave the bounds for now, and focus on the conditions under which BE is an integer. Let g be the gcd of AC and BC, so that $AC = ag$ and $BC = bg$ with a and b coprime (this includes the case $a = b = 1$). So we have

$$BE = \frac{AB \cdot BC}{AC + BC} = \frac{AB \cdot bg}{(a + b)\, g} = \frac{AB \cdot b}{a + b}.$$

Now, b and $a+b$ are coprime, and this means that in order to have an integer BE, the condition we need is that AB is a multiple of $a+b$. This is interesting: we can iterate over BC and AC, and for each pair of values we have enough info to enclose the valid values for AB. However, I wanted to avoid calculating something on the order of 10^{10} gcds, so I tried to find a way to determine the gcds without actually calculating them. What I used was something I called a "delta strip". So, let x be some integer, and $y = mx + n$ be another one. Clearly $\gcd(x, y) = n$, so maybe I could just generate the gcds for every possible remainder between 0 and x. This can be generated only once for a given x, using its prime decomposition to avoid actually calculating the gcds (for example, if x is even, every even value of n will have a gcd multiple of 2). With some care this can be generated in $O(x \log x)$. So I will end with a list of pairs (n, g) so that, for every integer m, the value $mx + n$ is such that $\gcd(x, mx + n)$ is g. Furthermore, this allows iterating in order (just increase m, and given a fixed m, increase n). Now, a characteristic of this problem is that we want every value where $g > 1$: if $g = 1$, the smallest value for AB

is $a + b = AC + BC$, which doesn't result in a valid triangle. And now this defines a complete algorithm. Let $L = 10^5$ be the limit; the algorithm is as follows:

- Initialise the result as 0.
- Iterate BC from 2 to $\left\lfloor \frac{L}{3} \right\rfloor$ (included).
- Now, given a value $BC = x$, generate the "delta strip" with all the values between 0 (included) and x (not included) whose gcd with x is greater than 1.
- Iterate over integer values of m (starting from $m = 1$) and values of n in the strip (starting from $n = 0$).
- We will have $AC = mx + n$. We can finish the iteration when $AC > \left\lfloor \frac{L - BC}{2} \right\rfloor$.
- We have a value of g associated to n in the delta strip, and we know that $\gcd(BC, AC) = g$. So we have $a = \frac{BC}{g}$ and $b = \frac{AC}{g}$, and we know that these are coprime.
- The valid range for AB is $[z_0, z_f)$, where $z_0 = AB$ and

$$z_f = \min\{BC + AC, L + 1 - (BC + AC)\}.$$

- We can now calculate the amount of valid values for AB, which is

$$n_{AB} = \left\lfloor \frac{z_f - 1}{a + b} \right\rfloor - \left\lfloor \frac{z_0 - 1}{a + b} \right\rfloor.$$

- Add this value, n_{AB}, to the result variable.
- Keep iterating AC using the delta strip until we get to the limit, and keep iterating BC in increments of one until we arrive to the maximum value.

Generating the "delta strip" is kind of slow, but the rest of the algorithm is pretty fast. In the end, the full algorithm is about $O\left(n^2 \log^2 n\right)$, but with a very small constant. The run time is about 3,9 seconds, out of which 2,8 are spent in the strip generation.

311. Biclinic Integral Quadrilaterals

Difficulty rating: 70 %.

Solution: 2466018557. *Solved: Tue, 12 Apr 2022, 02:33.*

Math knowledge used: cosine theorem, Pascal's triangle, Hermite-Serret algorithm, square sum enumeration, "first prime" Erathostenes Sieve.

Programming techniques used: memoisation.

It may not look like that at first, but this problem has a lot in common with 264. In both cases the heart of the problem resides in efficiently enumerating the ways to express numbers as the sum of two squares. I would say that this is hard by itself, but if you have already solved 264, at least you understand like 75 % of what you need for this problem and it's much easier.

Let's start with the most obvious insight: you can reduce the quadrilateral to a combination of four triangles, which in terms of the problem description diagram would be AOB, BOC, COD and DOA. And then, a series of additional, useful insights follow:

- Since AO = CO and BO = DO, two of the sides are the same for all four triangles.
- We can determine the length of the remaining side of the triangle (which is one side of the quadrilateral) using the cosine theorem.
- The angles in the center at AOB and DOA are supplementary. So are the ones at BOC and COD.
- Combining these last two pieces, we note that two supplementary angles α and β, verify $\cos\alpha = -\cos\beta$.

This looks promising. We now note that, as per the problem description, we are looking for cases where CO $\leq$ BO. Let's call x = AO = CO and y = BO = DO. Also, let's use $a < b < c < d$ for the four sides (so a and d have the supplementary angle relationship, and so do b and c). We start by looking separately at a and d, and get these relationships from the cosine theorem:

$$\begin{aligned} x^2 + y^2 - 2xy\cos\alpha &= a^2, \\ x^2 + y^2 - 2xy\cos\beta &= d^2. \end{aligned}$$

Here, α and β are the angles at O. Yes, those same angles that happen to be supplementary. Therefore we can rewrite the second equation as

$$x^2 + y^2 + 2xy\cos\alpha = d^2.$$

And adding the two equations we get a very nice one:

$$a^2 + d^2 = 2\left(x^2 + y^2\right).$$

Of course, the very same analysis is valid for the other pair of sides, yielding

$$b^2 + c^2 = 2\left(x^2 + y^2\right).$$

We can now start imposing some restrictions. First of all we note that the problem description says that BD is an integer, but it doesn't specify whether BO and DO (that is, y) are integers as well; they might be half integers. Fortunately, this can't happen: if that were the case, then $2\left(x^2 + y^2\right)$ would be a half integer as well, but that's not possible because both $a^2 + d^2$ and $b^2 + c^2$ are integers. Therefore we can assume that x and y are integers as well. Everything stays comfortably in integer arithmetic.

Let's now look at the decompositions. Let $A = x^2 + y^2$ be any number. We need to decompose A as a sum of squares (we know how to do that since 264, but we can add more finesse). Then, $a^2 + d^2 = b^2 + c^2 = 2A$, so we need to have at least two different decompositions of $2A$ as sum of squares (we can't use the same, because it won't satisfy $a < b < c < d$). It so happens that:

- $2A$ has exactly as many decompositions as a sum of squares as A does. In fact, if $A = m^2 + n^2$, then $2A = (m+n)^2 + (m-n)^2$.
- Provided that A has some decompositions as a sum of squares (well, we can filter the cases where it doesn't. It's easy to identify them beforehand), it has exactly one if and only if A is prime (a prime of the form $4k+1$, naturally). If A if the square of a prime, there are two different representations, but one is $p^2 + 0^2$, which we don't need.

These pieces of information are going to be useful when filtering, but now we can start defining the calculation scheme that we will use. We know that a number can't be expressed as a sum of squares if it has any prime factor of the form $4k+3$ raised to an odd power, and if that doesn't happen, we can still ignore factors of the form $(4k+3)^{2n}$ since they don't affect the decompositions (or lack thereof), except for a scaling factor. This suggests product combinations of primes of the form $4k+1$, and then calculating the amount of "rescaled" cases at the end.

And now it comes and interesting tidbit. There are infinite primes of the form $4k+1$. There are also infinite primes of the form $4k+3$. And aside from these, there is one single additional prime: 2. We know that 2 doesn't affect the amount of decompositions, so it can be part of the "rescaled" factors.

However, we need to be careful with it because it doesn't have the same effect as $4k+3$ primes. Let's illustrate the problem with an example:

- 65 is the product of the two smallest $4k+1$ primes. As such, it has two decompositions: $1+64$ and $16+49$.
- We can use $(4k+3)^{2n}$ prime powers as rescaling: $65 \cdot 9 = 585 = 9 + 576 = 144 + 441$.
- And we can also use 4 as rescaling: $65 \cdot 4 = 260 = 4 + 256 = 64 + 196$.
- However, a factor of 2 has a different effect: the amount of decompositions is the same, but these are not rescalings. $65 \cdot 2 = 130 = 9 + 121 = 49 + 81$.
- A factor of 8 can be seen as a rescaling over the decomposition of the "case with an added two": $65 \cdot 8 = 520 = 36 + 484 = 196 + 324$.

If we are going to apply the rescalings separately, we need to generate two different sets of numbers, and 2 appears in both of them:

- On one hand, we will have the "primitive numbers": product combinations of $4k+1$ primes and its powers, and possibly 2 (not as a power).
- On the other hand, we need the "rescaling factors": product combinations of $4k+3$ primes and 4, and their powers.

This is already starting to take shape. For each one of the "primitive numbers" we will need to calculate all the decompositions. We can do this in a relatively orderly way:

- For $4k+1$ primes we can use the Hermite-Serret algorithm, which I've used in several problems already.
- For powers of $4k+1$ I discovered a wonderful algorithm. Let $p = m^2+n^2$ be the primitive decomposition given by the Hermite-Serret algorithm. Then we have a single primitive $p^k = \mu^2 + \nu^2$, where

$$\begin{aligned} \mu &= \sum_{\substack{i=0 \\ i \text{ even}}}^{k} (-1)^{i/2} \binom{k}{i} a^i b^{k-i}, \\ \nu &= \sum_{\substack{i=1 \\ i \text{ odd}}}^{k} (-1)^{(i-1)/2} \binom{k}{i} a^i b^{k-i}. \end{aligned}$$

This is a direct consequence of expressing $p = (m + in)(m - in)$.

Additionally, we can rescale all the solutions (primitive or not) of p^{k-2}: given $p^{k-2} = m^2 + n^2$, then $p^k = (pm)^2 + (pn)^2$ is a valid solution as well. It's not a good idea to rely on rescaling because these separate solutions will also need to be combined with others, so giving it special treatment only makes the code needlessly complicated.

- Given the product of two coprime powers of $4k + 1$ primes, $p \cdot q$ with $p = m^2 + n^2$ and $q = r^2 + s^2$, we can combine the solutions:

$$pq = (mr + ns)^2 + (ms - nr)^2 .$$

  Note that using $p = n^2 + m^2$ (i.e. swapping the order of one of the decompositions) gives additional solutions that must be taken into acacount if we want to be exhaustive. We only need to do one swapping, i.e. either p or q. Swapping both, or swapping p and q separately, results in duplicate solutions.

  And yes, this is the same we needed to do in 264 and other similar problems.

- Finally, the factor 2: if $p = m^2 + n^2$, then $2p = (m + n)^2 + (m - n)^2$.

This is good enough to generate all the solutions of $A = a^2 + b^2$ for any A. We can also add the rescaling later. But this is not the whole story. First of all, remember how I had $x^2 + y^2 - 2xy \cos\alpha = a^2$. This means that not every value of a is valid: this cosine must be in the valid range, $[-1, 1]$ (in fact, by construction we don't get negative numbers, so we would get $[0, 1]$). In practice this means that $a > y - x$. Second, we need that sides are not equal. Therefore solutions where the cosine is 0 are also not valid. This happens when $a = b$. I lost a few hours because of this. Stupid me. Finally, we need to combine solutions intelligently to satisfy the constraints. Let's take a real case with $A = 5^3 \cdot 13 = 1625$, and in particular $x = 28$ and $y = 29$. There are three valid solutions of $2A = m^2 + n^2$: $m = 15$, $n = 55$; $m = 21$, $n = 53$; and $m = 35$, $n = 45$. If we want to assign these to a, b, c, d (i.e. one pair is (a, d) and the other one is (b, c)), then we can automatically satisfy $b < c$, but we need to make sure that $a < b$. This will also naturally satisfy $c < d$. So there are three valid assignments: $a = 15$, $b = 21$; $a = 15$, $b = 35$; and $a = 21$, $b = 35$. In general, if the amount of solutions we have for $2A = m^2 + n^2$ is some value t, then there are exactly $\dfrac{t\,(t-1)}{2}$ valid pair assignments. Basic combinatorics, really.

This is almost finished. We only need to take care of the limits. First we consider that $a^2+b^2+c^2+d^2 \leq N$. Since we are going to decompose numbers of the form x^2+y^2 so that $a^2+d^2 = b^2+c^2 = 2\left(x^2+y^2\right)$, this means that we can use $x^2+y^2 \leq \dfrac{N}{4}$ as a limit for the numbers to decompose. We don't need to use all the primes, since a product of two primes is necessary to have at least one valid solution. The smallest relevant prime is 5, so we only need primes up to $\dfrac{1}{5}\dfrac{N}{4} = \dfrac{N}{20}$. As per the scaling factors, we only need to consider values up to $\dfrac{N}{25}$, but since only squares are needed, there aren't that many numbers.

Now, a problem that I found is that there are too many "primitive" numbers and generating their decompositions takes time and also a lot of memory (there are around $2 \cdot 10^8$ numbers). So I removed numbers that I wasn't going to need. Long story short, if I was working with a prime p, beforehand I removed all the data for values greater than $\dfrac{N}{4p}$. With this, the memory never went above 3Gb even for the hungry JVM. The full calculation scheme I used is this:

- Get the primes up to $\dfrac{N}{20}$ with an Erathostenes sieve.
- Use the primes to generate all the products of even powers of 2 and $4k+3$ primes.
- Now the main iteration begins. Create a map of value to list of decompositions, and initialise a result variable with 0.
- Iterate every prime p of the form $4k+1$:
  - Remove elements from the map if the key is strictly greater than $\dfrac{N}{4p}$.
  - Create a new map of "new" elements.
  - Use the Hermite-Serret algorithm to find a decomposition of p as sum of squares.
  - Combine this decomposition with all the values in the preexisting map, adding them to the "new" one.
  - For every power of p, get all the decompositions and combine them as well into the new map. Stop when reaching $\dfrac{N}{4}$.
  - Now, iterate over all the entries of the new map:

- Let q be a number with a list of decompositions, $q = m_i^2 + n_i^2$.
- Initialise an accumulator to 0.
- Get the decompositions of $2q$ using the known scrambling method. Ignore the cases with $m_i = n_i$.
- Sort the decompositions of $2q$ using the smallest member as key.
- For every decomposition of $q = m_i^2 + n_i^2$, check how many decompositions of $2q$ have a smallest member whose value is strictly greater than $|n_i - m_i|$. If there are t such elements, increase the accumulator in $\dfrac{t\,(t-1)}{2}$.
- Once all decompositions have been iterated over, use a binary search with the set of scale factors to determine how many are below or equal $\dfrac{N}{4q}$. If the result is u, increase the result variable in u times the accumulator.
- If $q < \dfrac{N}{8}$, repeat the same steps for $2q$, taking advantage of the fact that we already have its decomposition.

- Add the elements of the new map to the preexisting one, so that the next prime can already use them.

- The value of the "result" variable after this loop is the solution to the problem.

It takes about 2 minutes on my PC. I don't think that the 70 % difficulty level is really warranted, but I guess that at the time this was indeed near the highest level of difficulty available. 264, which was very similar and also very doable, was rated 85 %, which seems also very high to me.

325. Stone Game II

Difficulty rating: 80 %.
Solution: 54672965. *Solved: Tue, 29 Sep 2020, 00:23.*
Math knowledge used: Beatty sequences, lower Wythoff sequence.
Programming techniques used: dynamic programming.

This is the hardest problem needed for the "Triangle trophy" achievement, with 253 being a second close. The problem is that the pattern is easy to intuit, but hard to pin out, because it has irregularities. The basic formula is this: given any fixed value of x, the values of y that we need to include in our sum are those which verify this condition: $x + 1 \leq y \leq \lfloor \varphi x \rfloor$. And here

comes some brief mathematical theory about $\lfloor \varphi x \rfloor$: these kind of sequences (the floor of the product of an irrational number with the succession of the natural numbers) are called Beatty sequences, and they are kind of regular but not really patterned (i.e. they are predictable, but there aren't cycles one could rely on). In particular, for the case of φ, this is called the lower Wythoff sequence, and pinning the whole sequence can be hard, possibly the hardest point of this problem. I don't remember where I got the formula, but I didn't find it myself. Here it is: suppose that instead of the succession $a_n = \lfloor \varphi n \rfloor$, we consider the sequence of differences, $b_n = \lfloor \varphi (n+1) \rfloor - \lfloor \varphi n \rfloor - 1$. This sequence looks like this:

$$0101101011011010110101101101011 01 \ldots$$

Looks random-ish, right? Sometimes there is a single one between two consecutive zeroes, sometimes there are two. Even if we consider the sequence of the amount of ones, it's still a random-looking sequence where it's not immediate to predict whether the next block will have one or two ones. But what about... this?

$$0 - 1 - 01 - 101 - 01101 - 10101101 - 0110110101101 \ldots$$

Each block has the length of the Fibonacci numbers. And each block save the first two is build by "adding" (concatenating, really) the previous two ones. Yes, this works. Yes, this works indefinitely. Boom.

I'm putting this here, at the start of the problem description, like it's nothing, but discovering this was a piece that was missing for me for a long, long time. So much that, even if the rest of the problem is laborious, once I knew this, I was able to solve this in about one or two evenings. For a long time this problem stood as one of the most difficult problems that I had managed to solve. There are a couple of 85 % problems that are in my opinion far easier than this one, if only because of the obscure pattern involved. The satisfying thing about this problem is that, despite this, the solution is blazing fast, less than 4 milliseconds even with BigIntegers, since it's calculated in logarithmic time.

Now I can start discussing the implementation of the solution, which is still not trivial or immediate, but it's about par for the course for middle-of-the-road project Euler problems; I've used it quite often for later problems. First off, given any x, I know that the valid values for y are between $x + 1$ and some value that can be calculated using the magic of the lower Wythoff sequence. Now, considering the regularity of the Wythoff sequences, it makes sense to divide the domain of x in what I called "Fibonacci-sized blocks".

Each block represents one of the subblocks (0, 1, 01, etc) built recursively from the previous ones (save for the first two ones). I visualise the blocks like two dimensional shapes composed of lines where each line has the same length as the previous ones, or maybe one additional element, so that in the end there is a fixed height and a variable width. I'm interested mostly in the total width increase. Aside from these blocks, I keep an object with the current state of the total sum, which contains the current summation index, the current width of the block and the current total sum. This doesn't represent the whole sum, though. The suggested scheme iterates over values of x and adds all the y in the interval $[x+1, \lfloor \varphi x \rfloor]$, but we need to limit it to $\{x, y\} \leq N$ for a given limit, $N = 10^{16}$. So we need to divide the space of x in two separate parts:

- The parts where, given x, the biggest y is below the limit and we need to sum them all. That is, $\lfloor \varphi x \rfloor \leq N$.
- The remaining parts, where $x \leq N$ by definition, but for which we don't want to sum all the y, just the ones below N.

So we calculate what I called the *Wythoff limit* of the problem, $L_W = \left\lceil \frac{x}{\varphi} \right\rceil$. We will use the block-based approach (which itself has two separate parts) for $x \leq L_W$, and a separate calculation (which happens to be very simple) for $x > L_W$.

First, the block approach. Each block, as described above, increases x in a certain amount. In fact, it's clearly always a Fibonacci number. However, it would be a happy coincidence if L_W happened to be a Fibonacci number. Indeed, it's not. So we need to add a "partial" block at the end, which is also built on partial building blocks. We are going to need three functions for these block calculations:

- First of all we need to create new Fibonacci blocks. That is, given the blocks n and $n+1$, create the $n+2$ one.
- We also need a function to update the current sum state with a full Fibonacci block, which is the main meat of the scheme.
- And finally we will need a function to update the current sum state with a partial Fibonacci block, building on the inner block structure and its two initial components.

Each one of these functions is reasonably simple, although the last one involves recursion. We can define the block as a list with the values of several

successions, $\{x_n\}$, for the same index n. The successions involved are these ones:

- l_n, the length of the block. This is, blatantly, always a Fibonacci number. This is the amount of increased x by each block.
- c_n, the added count of the block. These are in fact also Fibonacci numbers. Similarly, this is the amount of increased y by each block.
- t_n, the amount of total added terms. Things are starting to get complicated and there are multiplications involved in this calculation.
- x_n, the total sum of terms x added by this block.
- y_n, the total sum of terms y added by this block.

Now, we need to take into account the fact that at each step we actually sum two "blocks". One of them is rectangular and it depends mostly on the previous width (which is y in the problem) and the current height (x). The other one is irregular and it's the one we're updating with these five successions above. So, the successions are updated with the following formulas, building on the previous elements:

$$\begin{aligned}
l_n &= l_{n-1} + l_{n-2}; \\
c_n &= c_{n-1} + c_{n-2}; \\
t_n &= t_{n-1} + t_{n-2} + l_{n-1}c_{n-2}; \\
x_n &= x_{n-1} + x_{n-2} + t_{n-1}l_{n-2} + c_{n-2}S\left(l_{n-2}, l_n - 1\right); \\
y_n &= y_{n-1} + y_{n-2} + t_{n-1}\left(l_{n-2} + c_{n-2}\right) + \frac{\left(l_{n-2} + l_n + c_{n-2}\right)c_{n-2}l_{n-1}}{2}.
\end{aligned}$$

The first two formulas are simple enough, but the other kept needing an increasing amount of pen and paper. The function $S\left(a, b\right)$ represents just the sum of all the terms between a and b:

$$S\left(a, b\right) = \sum_{x=a}^{b} x = \frac{\left(a + b\right)\left(b + 1 - a\right)}{2}.$$

And yes, that argument is not a typo: it's $l_n - 1$, not l_{n-1}.

The second function we need is the one that calculates the current state of the sum. Again, we can consider it like a holder of three successions:

- I_n, the current index of x.
- W_n, the current width, which is in fact the maximum y reached.

- S_n, the current total sum of $x + y$.

At each step, we update these values using these formulas based on the immediately previous elements and the current Fibonacci block:

$$\begin{aligned} I_n &= I_{n-1} + l_n; \\ W_n &= W_{n-1} + c_n; \\ S_n &= S_{n-1} + W_{n-1} S\left(I_{n-1}, I_n - 1\right) + \left(t_n I_{n-1} + x_n\right) + \\ &\quad + \frac{\left(2I_{n-1} + l_n + W_{n-1}\right) W_{n-1}}{2} + \left(\left(I_{n-1} + W_{n-1}\right) t_n + y_n\right). \end{aligned}$$

The really relevant one is the last one, naturally. Aside from the previous sum S_{n-1}, there are four terms. The first one (with the consecutive sum) represents the addition of x in the rectangular block; the second one, which is the simplest one (the complicated term, x_n, is already calculated), represents the addition of x in the irregular shape of the Fibonacci block; the third one, with the fraction, is the addition of y in the rectangular block; and the last one, again relatively simple because the complicated formula for y_n was calculated elsewhere, is the addition of y in the irregular shape of the Fibonacci block.

The good news is that the complicated formulas end here. The bad news is that we haven't finished. With this we can calculate the sum of all the $x + y$ up to some $x = I_n$, where I_n is a Fibonacci number. We are missing the values that remain up to $x = L_W$. For this, we use partial functions. Let's say that we have the current state, $\{I, W, S\}$ and we need to update the values with the last block n, but only up to a partial amount of height, $L = L_W - I_n$. We can define a partial advance function f_n (associated to the Fibonacci block n), so that $\{I_{L_W}, W_{L_W}, S_{L_W}\} = f_n\left(I, W, S, L\right)$. We are going to need the "full advance" function, g_n, which is basically the application of the updates for I, W and S using the formulas above for a given block n. With this, the recursive expression of f takes the following form:

$$f_n\left(I, W, S, L\right) = \begin{cases} f_{n-1}\left(I, W, S, L\right) & \text{if } L < l_{n-2}, \\ g_{n-2}\left(I, W, S\right) & \text{if } L = l_{n-2}, \\ f_{n-1}\left(g_{n-2}\left(I, W, S\right), L - l_{n-2}\right) & \text{if } L > l_{n-2}. \end{cases}$$

In terms of code it's actually pretty simple, just a structure of recursive function calls that is guaranteed to end because eventually we get to blocks of length 1, and $0 < L \leq l_n$ by construction.

Anyway we haven't finished yet. We are missing the last part of the problem, which is fortunately quite easy. We have values of x in the interval $[L_W + 1, N]$, and for each one we will need values of y in the interval $[x + 1, N]$. It so happens that the sum of $x + y$ can be simplified greatly. Let $a = L_W + 1$ and $b = N$ be the limits of this function. So, we have $x = a$ a total of $b - a$ times; $x = a + 1$ a total of $b - a - 1$ times, and so on, up to $x = b - 1$ one single time; and $y = a + 1$ a total of 1 times, $y = a + 2$ two times, and so on, until $y = b$, which appears $b - a$ times. Very conveniently, each term in $[a, b]$ appears exactly $b - a$ times, so the sum of all the terms is $(b - a)\, S(a, b)$. And that, finally, finishes the computation.

The obscure part of this problem is finding the regularity of the Wythoff sequence. The laborious part is carefully finding all the formulas. The combination of the two make this a very hard problem, but fortunately I was able to solve it in the end. The full result is $O(N^3)$ before modding, so BigIntegers are very fast. The full result is this 48 digit number:

$$230327668541684176515052475525037682834286696168$$

326. Modulo Summations

Difficulty rating: 55 %.
Solution: 1966666166408794329. *Solved: Sat, 18 Jun 2022, 08:32.*
Math knowledge used: triangular numbers.
Programming techniques used: binary search.

This is a purely algorithmic problem, where the most difficult part is just finding a cycle, which seems to have fixed length depending on the starting parameter M. In particular, the cycle length is always $6M$. There are proofs of this in the problem thread. So, the cycle length is $6 \cdot 10^6$, which is manageable. So what I did was keeping an array of arrays with the positions of the partial sums. That is: if the first 10 elements from the problem are

$$\{1, 1, 0, 3, 0, 3, 5, 4, 1, 9\},$$

then the partial sums (modulo 10) are

$$\{1, 2, 2, 5, 5, 8, 3, 7, 4, 3\},$$

so I would keep these arrays:

$$\begin{aligned} 0 &\Rightarrow [\,]; \\ 1 &\Rightarrow [0]; \end{aligned}$$

$$
\begin{aligned}
2 &\Rightarrow [1, 2]\,; \\
3 &\Rightarrow [6, 9]\,; \\
4 &\Rightarrow [8]\,; \\
5 &\Rightarrow [3, 4]\,; \\
6 &\Rightarrow []\,; \\
7 &\Rightarrow [7]\,; \\
8 &\Rightarrow [5]\,; \\
9 &\Rightarrow []\,.
\end{aligned}
$$

Now, of course I only store indices up to the cycle length, $6 \cdot 10^6$; that's the reason why I want to determine the cycle length. These arrays are all I need to calculate the number I want, because:

- The array at position x can be used to calculate how many values p verify the condition $\sum_{i=0}^{p} a_i \equiv x \pmod{M}$ in the range $p \in [0, N]$.
- For each pair of values (p, q) such that $\sum_{i=0}^{p} a_i \equiv \sum_{i=0}^{q} a_i \equiv x \pmod{M}$, we have $\sum_{i=p+1}^{q} a_i \equiv 0 \pmod{M}$.

Let $L = 6M$ be the cycle length. We have $F = \left\lfloor \dfrac{N}{6M} \right\rfloor$ full cycles, and a remainder of $R = N \,\%\, (6M)$ elements. Therefore, for each one of these arrays, we do the following:

- If the array length is a and a binary search reports that the array has b numbers below R (not included), then there are $y = aL + b$ values such that $\sum_{i=0}^{p} a_i \equiv x \pmod{M}$.
- The case $x = 0$ is a special one because we need to count the start of the sequence as well, so we do $y \leftarrow y + 1$ in that case.
- That's it. Increase the result variable in $\dfrac{y\,(y-1)}{2}$, which is the amount of pairs with the same modulo sum (therefore the amount of sequences where the modulo sum is 0).

In some cases the array doesn't have any element and it can be discarded, saving all these calculations. Not a big deal since there aren't that many of them. The end result takes about 1,6 seconds to be calculated, and around 1,5 of them are spent doing all the calculations of the initial sequence (this is because, annoyingly, BigIntegers are needed).

327. Rooms of Doom

Difficulty rating: 40 %.
Solution: 34315549139516. *Solved: Sun, 7 Aug 2022, 20:06.*
Math knowledge used: none.
Programming techniques used: dynamic programming.

This is it. The last "easy" problem I had left. It had become a small goal of sorts to me, to leave this problem for some time when I wanted an easy problem to solve quickly. In the end I chose this problem as the 625th I solved, to mark my arrival to level 25, which is almost definitely the last level I will reach, considering that I'm not going to have much time to solve more problems. So long and thanks for all the fun, I guess? Although I expect to be able to solve a couple problems more. I don't think I'm going to get to 630, and I'm very definitely not getting to 640.

Anyway, let's talk about the problem. A few experiments reveal that for $C = 3$ you can use the formula

$$M(C, R) = \frac{3^{R-1} + 3}{2},$$

but things are not so easy when $C \geq 3$ (when $C \leq 2$ there can't be any solutions if $R \geq C$). The trick is that solutions can be built recursively on top of each other. Let's say that you have found the value $M(C, R-1) = x$. This means that there is a solution for C cards and $R-1$ rooms that uses exactly x cards. Now, our goal is to "prepare" the first room of the case with R rooms, so that it's functionally equivalent to the infinite dispenser. We know that we are only going to need x cards for the remaining $R-1$ rooms, so our goal is to find ourselves in a situation when the first room has y cards and we have n cards at our hand, so that $y + n = x$. Naturally $0 \leq n \leq C$, but experimentally it can be seen that the bests results are found when $y = x + 1 - C$, so that at the last visit to the infinite dispenser we get one last batch of C cards; we spend one entering the first room, and now we have $y = x + 1 - C$ cards in the first room, plus $C - 1$ cards in our hand, and $y + n = x$, so we can now reproduce the exact same set of steps from the $R - 1$ solution. This is always optimal.

Hooray? Well, we haven't finished yet, although, well, almost. So we need to build a situation where there are exactly $y = x + 1 - C$ cards in the first room. Now, at each visit we can spend C cards to put $C - 2$ cards, and visits can certainly not be a fractional number, so we are going to need

$v = \left\lceil \frac{y}{C-2} \right\rceil$ visits, adding to $2v$ additional elements. So, the decomposition of the solution is something like this:

- $y = x + 1 - C$ cards (where x is the previous solution) that will go in the first room's box.
- $2 \left\lceil \frac{y}{C-2} \right\rceil$ additional cards that we will need during as many travels as needed, which we will need in order to put such y cards in the first room's box.
- One last batch of C cards which, along with the y cards in the first room's box, is enough to get to the end by reproducing the base solution for $R - 1$ rooms.

This allows a very simple recursive procedure that ends in a few milliseconds (although apparently that's because `LongMath.divide` is annoyingly slow). In the case for $C = 3$, the division is particularly easy because the quotient is always 1. This is why there exists a simple closed formula. For $C > 3$ I'm pretty sure that something can be done because the remainders of $\frac{y}{C-2}$ are likely periodic, but I haven't bothered since the formula is so fast (and the size of the problem is ridiculously small). The solutions are expected to grow exponentially, something like $M(C,R) \approx O\left(\left(\frac{C}{C-2}\right)^R\right)$, but that's not exact. What can be done, though, is work a bit with the recursive formula in order to get an iterative one, which is even faster (although, really, most of my speedup came from removing the call to `LongMath`). Here it is:

- If $R < C$, the result is $R + 1$, no recursive calculations needed.
- Otherwise, let's suppose that $R \geq C$. We start with a temporary result, $r = 4$.
- Now, let's iterate for $i = C + 1$ up to R (included):
- Let $q = \left\lceil \frac{r}{c-2} \right\rceil$.
- Do $r = r + 2q + 1$ and go to the next i.
- After finishing this small loop, the value of $M(C,R)$ is $r + C - 1$.

It can be seen that this is equivalent to the recursive computation. It's not much faster, though; it's ridiculously fast either way. It could be done by hand in a couple minutes, probably. It takes about 14 microseconds on my machine. Basically everyone is doing the same calculation on the problem thread.

Aaaaaand now there aren't any more problems with more than 1000 solvers which I haven't solved. And I don't expect to solve many more problems. So long and thanks for all the fun?

330. Euler's Number

Difficulty rating: 70 %.

Solution: 15955822. *Solved: Wed, 10 Nov 2021, 07:26.*

Math knowledge used: exponential series, recurrence equations, combinatorial numbers, Chinese remainder theorem.

Programming techniques used: combinatorial numbers cache.

Yes! Chinese remainder theorem. That's the final insight needed for this problem, and one that I took a bit to realise. As expected, the solution uses just integers and e is not found anywhere other than in the very first stages of the analysis. If the usual stages of a solution are formulation, refinement and coding, this problem is almost 100 % refinement. The formulation is already given in the definition, and the coding is simple.

Clearly, the first thing to do for the analysis is to find a formula for $a(n)$. Possibly recursive, ideally direct. Spoiler: it will be recursive. We obviously need to consider a very well known result from the exponential series:

$$\sum_{n=0}^{\infty} \frac{1}{n!} = e.$$

So it's clear that $a(n)$ has some infinite series that sum up to e, minus some terms at the start which are replaced with the already calculated values of $a(n)$ for $a \geq 0$. The analysis starts separating the "base" values from the rest, like this:

$$a(n) = \sum_{i=1}^{\infty} \frac{a(n-i)}{i!} = \sum_{i=1}^{n} \frac{a(n-i)}{i!} + \sum_{i=n+1}^{\infty} \frac{1}{i!}.$$

The second series looks a lot like the exponential series, but with a few terms missing. So we do this:

$$a(n) = \sum_{i=1}^{n} \frac{a(n-i)}{i!} + \left(\sum_{i=0}^{\infty} \frac{1}{i!} - \sum_{i=0}^{n} \frac{1}{i!} \right) = \sum_{i=1}^{n} \frac{a(n-i)}{i!} + \left(\sum_{i=0}^{\infty} \frac{1}{i!} - 1 - \sum_{i=1}^{n} \frac{1}{i!} \right).$$

Inside the parenthesis we see that the first term is equal to e, the second is, well, 1, and the third one is a summatory over the same indices than the recursive part. So we get:

$$a(n) = \sum_{i=1}^{n} \frac{a(n-i) - 1}{i!} + e - 1.$$

This formula is valid for $n \geq 0$, and yes, in the particular case $n = 0$ there are no terms in the summatory and we get $a(0) = e - 1$, which is indeed the correct value.

Hooray, the first step is done. Now we are interested in removing the denominators. After all, the series we're interested in don't have a denominator. So we define a new succession, $b(n) = n! \cdot a(n)$, and we have

$$A(n) \cdot e + B(n) = b(n) = n! \cdot a(n) \Rightarrow a(n) = \frac{b(n)}{n!}.$$

We can get a recursive formula for $b(n)$ very easily:

$$b(n) = n! \left(\sum_{i=1}^{n} \frac{a(n-i) - 1}{i!} + e - 1 \right) = n! \left(\sum_{i=1}^{n} \left(\frac{b(n-i)}{i!\,(n-i)!} - \frac{1}{i!} \right) + e - 1 \right);$$

and we can see the combinatorial numbers appearing from the product and quotient of factorials:

$$b(n) = \left(\sum_{i=1}^{n} \binom{n}{i} b(n-i) \right) - \left(\sum_{i=1}^{n} \frac{n!}{i!} \right) + n!\,(e-1).$$

We can now replace $b(n)$ with $A(n) \cdot e + B(n)$, getting

$$A(n) \cdot e + B(n) = \left(\sum_{i=1}^{n} \binom{n}{i} A(n-i) \right) e + \left(\sum_{i=1}^{n} \binom{n}{i} B(n-i) \right) - \left(\sum_{i=1}^{n} \frac{n!}{i!} \right) + n!\,(e-1),$$

and since everything is an integer save from the appearances of e, we can cleanly separate $A(n)$ and $B(n)$:

$$\begin{aligned} A(n) &= \sum_{i=1}^{n} \binom{n}{i} A(n-i) + n!, \\ B(n) &= \sum_{i=1}^{n} \binom{n}{i} B(n-i) - n! - \sum_{i=1}^{n} \frac{n!}{i!}. \end{aligned}$$

The base cases, for $n = 0$, are $A(0) = 1$ and $B(0) = -1$. It's clear that $A(n)$ will always be positive, $B(n)$ will always be negative, and the sum of both will be negative as well.

Now, we could analyse both formulas separately ($A(n)$ appears in OEIS! But $B(n)$ does not), but since we are interested in $A(N) + B(N)$ for certain N, we can combine the two successions into one, $C(n) = A(n) + B(n)$. This works very cleanly because the recursive coefficients are the same in both successions. So we get a deceptively simple formula:

$$C(n) = \sum_{i=1}^{n-1} \binom{n}{i} C(n-i) - \sum_{i=1}^{n} \frac{n!}{i!}.$$

Note that we have removed the element for n in the summatory. This is because $C(0) = 0$. It's not very relevant, though. We can also define

$$D(n) = \sum_{i=1}^{n} \frac{n!}{i!},$$

so that

$$C(n) = \sum_{i=1}^{n-1} \binom{n}{i} C(n-i) - D(n).$$

We do this because we are going to analyse $C(n)$ and $D(n)$ separately. Now, there are some more or less obvious quadratic algorithms to calculate $C(n)$ and $D(n)$, but that's not good enough. We need something different, and here it is: the piece of insight you need to solve this problem. We don't need to calculate $C(N)$, but $C(N) \bmod M$, for some modulus M. Now, in later problems it became very common to use a large prime number slightly above (and, occassionally, below) 10^9 as the modulus, so it's very remarkable that in this problem we are given $M = 77777777$. This is because a value like this is necessary to solve this problem adequately. Yes, we are going

to exploit the periodicity of $D(n)$ and $C(n)$, getting small solutions and eventually combining them using the Chinese remainder theorem.

So, what's the periodicity of $C(n)$ and $D(n)$ modulo some prime p? Interestingly, it's not the same. In fact, $D(n)$ is not purely periodical (although surprisingly, $C(n)$ is, despite that). The periodiocity is like this:

- $D(n)$ has a period of length p, but there is a prefix of p different values. For example, for $p = 7$ the first p terms are $\{0, 1, 3, 3, 6, 3, 5\}$ (indices starting at 0: this means $D(0) = 0$, $D(1) = 1$ and so on), but these are followed by an infinite list repeating the terms $\{1, 2, 5, 2, 2, 4, 4\}$ indefinitely.

- $C(n)$ has a pure period, whose length is $p(p-1) = \varphi(p^2)$. This means that, for any $n \geq p(p-1)$ we have $C(n) \equiv C(n - p(p-1)) \pmod{p}$. Someone in the problem thread posted a very ugly proof, which I haven't tried following closely.

Anyway, these periodicities are the keys to solve the problem. Clearly we have

$$77777777 = 7 \cdot 11 \cdot 73 \cdot 101 \cdot 137,$$

and knowing that the period is $p(p-1)$ for each prime p, the obvious next thing to do is very fast. With $N = 10^9$ being the goal of the problem, for each one of these five primes p, we do this:

- Compute $D(n)$ up to $2p$ directly, using factorials. I actually used BigInteger for this, although I guess that there are finer ways of calculating these values. I only need to calculate up to 237!, which has less than 500 decimal digits, so it's not like I'm going to strain the processor or the memory.

- With these values of $D(n)$, and being aware of this periodicity, I can calculate $C(n)$ up to $p(p-1)$ as needed. In particular, I only need to calculate $-C(N \,\%\, (p(p-1)))$ (note the negative sign; I do this, paradoxically, to work with positive numbers only and avoid sign issues with the modulus) for each p. This is calculated using a cache of combinatorial numbers modulo p, and it can be computed using ints and longs (i.e. no BigIntegers) in a time of the order of n^2 where n is the highest value I need to calculate $C(n)$ for.

Now, we have some values C_7, C_{11} and so on, and we know that the solution of the problem is some value $R = C(N) \,\%M$ such that, for each one of the five primes involved, $-R \equiv C_p \pmod{p}$. Using the Chinese remainder

theorem I get a value x that is congruent with $-R \pmod{N}$, so I just return $M - x$ as the final result of the problem. Once I realised that the Chinese remainder theorem was the way to go, I got the right solution on my first try, with a run time well below 1 second. The order of the solution is $O(p^4)$ where $p = 137$ is the biggest prime that divides M, and BigIntegers are only modestly involved, which explains the speed.

334. Spilling the beans

Difficulty rating: 65 %.
Solution: 150320021261690835. *Solved: Wed, 15 Jun 2022, 07:08.*
Math knowledge used: triangular numbers.
Programming techniques used: none.

I might as well have added "none" as the math knowledge used. This problem looks like there is some kind of hidden mathematical result that you need to know in order to analyse the sequence, but in fact it's actually a purely algorithmic problem where the biggest problem is just studying the behaviour of the sequence and extracting a simple set of rules.

The first thing to note is that the amount of necessary movements is completely independent of the order in which you choose to do them. That is: there is no particularly optimal solution. You can just split the leftmost (or rightmost) bowl with more than one bean at each step (I'm using the word "split" to refer to each simple step of removing two beans from a bowl and moving them into the adjacent ones). This is a big relief, because the most potentially problematic issue (looking for the sequence that results in the lowest amount of steps) is just not there. I did indeed follow the systematic approach of always splitting the leftmost available bowl until I finished.

The next insight is that you are going to be left with a string of ones, with maybe some occasional zeroes, to the left of the current number you're working on; meanwhile, the bowls to the right of the current one are left largely untouched, save for the one immediately to the right, which grows slowly as you split the current one.

The final piece of the puzzle is realising that, each time you split the current bowl, the bean to the left is going to "propagate", altering the sequence of ones but ultimately leaving a slightly different sequence, while the current bowl slowly diminishes and the one immediately to its right slowly grows, and it's just a matter of analysing the behaviour of this string at each step. Most importantly, we can count how many steps does it take to stabilise the string after the current bowl is split. This analysis is actually not that com-

plicated, since there is just one primitive "move" and basically every other one is derived from this one.

First of all, it can be seen that there is only at most one zero in the string of ones (save for the infinite string at the left border, which we don't care about). Therefore we can abstract this whole string using just two variables: the length of the sequence (including the zero) and the position of this zero, which is most convenient to count from the right. Therefore we would abstract a sequence like 1011 as having a length of 4, with a zero in position 3 (from the right). I used -1 to indicate the cases where there is no 0, which is a bit of awkward in terms of the resulting code, and maybe not very efficient, but is easier to follow. Now let's look at the basic move:

- If the current state is $\{\{L, Z\}, N, M\}$, indicating that we have a sequence of ones of length L with a zero in position Z, with N and M being the amount of beans in the current and next bowl (where $N \geq 2$), then splitting the bowl with N beans results in $\{\{L, Z-1\}, N-1, M+1\}$ after exactly Z steps.

After splitting the bowl with N beans, only $N-2$ remain, but to its left we now have 2 beans, so after splitting this bowl one of the beans goes back to the original bowl, which is left with $N-1$. Note that the total amount of beans is preserved: $L-1+N+M$.

There are two special cases, related to the "beginning" and "end" of the movement of the zero along the sequence of strings:

- First, the case where there is no zero: we can treat the sequence of ones $\{L, -1\}$ as if it was $\{L+1, L+1\}$, that is: a sequence of length $L+1$ where the leftmost element is a zero. This is possible because of the infinite strings of zeroes at the left end of the relevant set of bowls. Therefore we can move from the state $\{\{L, -1\}, N, M\}$ into a new state, $\{\{L+1, L\}, N-1, M+1\}$, in $L+1$ steps.

- The other special case happens when $Z = 1$. In this case, after splitting the current bowl, the sequence is "filled", and there aren't any "twos", meaning that no bean comes back to the current bowl. As such, we move from $\{\{L, 1\}, N, M\}$ into $\{\{L, -1\}, N-2, M+1\}$ in 1 step.

- There is also a particular, unique case at the very start of the algorithm: the sequence $\{\{0, -1\}, N, M\}$ moves into $\{\{1, -1\}, N-2, M+1\}$ in a single step. I actually chose to use this state as the initial one, so as to not bother with an additional conditional sequence which would only run once at the start.

We can now focus a bit on N and M. We are using $N \geq 2$ as a precondition, therefore all of these steps are valid. The sequence ends when $N < 2$, that is, $N = 1$ or $N = 0$. After doing this, we can change the perspective a little, by accepting M as the new "current" bowl. The new "next" bowl comes from the initial sequence of 1500 bowls, but if we have finished it, we need to continue by retrieving a zero from the infinite set of empty bowls at the right end of the sequence. Let M' be this amount of beans (either the ones from the next bowl, or 0 if there is none). Then we consider these perspective changes:

- If the current state is $\{\{L, Z\}, 1, M\}$, with $Z \neq -1$, we add the one to the left sequence, resulting in $\{\{L + 1, Z + 1\}, M, M'\}$.
- If Z happened to be -1, we are still left with a sequence with no zeroes, so we move from $\{\{L, -1\}, 1, M\}$ to $\{\{L + 1, -1\}, M, M'\}$.
- We might also have $N = 0$ after "finishing" the current string, if we come from a case where $Z = 1$ and $N = 2$. In this case, we go from $\{\{L, -1\}, 0, M\}$ to $\{\{L + 1, 1\}, M, M'\}$. Note that it's not possible to arrive at $N = 0$ unless $Z = -1$, so there is a guarantee that there won't be more than one zero in the sequence of ones.

Note that these "perspective changes" don't include any bowl splitting, and so the total amount of steps is unaffected.

Now, this is enough to simulate the steps by decrementing N in one element (or occasionally two) at each time. However, this can be very slow, since M increases as well. Roughly, this means that if we have X beans in total, using these steps as atomic elements of the simulation would roughly take $O(X^2)$ operations. Since $X \approx 1{,}5e6$, this is kind of doable but too slow. Fortunately the formulas are way too simple and the steps are very easily combined into bigger sets. In fact, we can have just four kind of grouped sequences, and the total amount of steps can be easily counted by abusing this known formula:

$$A + (A + 1) + (A + 2) + \ldots + B = \frac{(A + B)(B + 1 - A)}{2}.$$

This is all the actual math kwnowledge I used for this problem; the rest is just algorithmics and counting the steps. Now, as I mentioned, there are four different kind of moves:

- If we are in a state like $\{\{L, Z\}, N, M\}$, with $Z \neq -1$ and $N \leq Z$, we can do $N - 1$ moves at a time. After them, the resulting sequence

is $\{\{L, Z-(N-1)\}, 1, M+N-1\}$. The amount of total steps taken will be $\frac{(2Z-N+1)(N-2)}{2}$.

- Now, if $Z \neq -1$ but $N > Z$, we can do Z steps in order to fill the sequence of ones. The resulting state is $\{\{L, -1\}, N-(Z+1), M+Z\}$, and this takes $\frac{(Z+1)Z}{2}$ steps. It's possible to have $N = 0$ at the end of this operation, and that's fine.

- We can now check the case $\{\{L, -1\}, N, M\}$. If $N \geq L+2$, we can do a "full" iteration into a whole new complete sequence of length $L+1$ with still no zeroes. This happens a lot, but only at the start of the algorithm; in the later stages L is too big and N is too small for this to happen. Anyway, we can move into $\{\{L+1, -1\}, N-(L+2), M+L+1\}$, and this operation takes $\frac{(L+1)(L+2)}{2}$ steps. I could have twisted this a bit further, counting how many "ones" can I add to the sequence in a single step, but this would have required a quadratic equation, and as I said, this doesn't really happen much after the very beginning, so I didn't bother. It's very possible that the end result would have been *less* performant, since I would have needed double operations for the square root, and in most cases it would result in no additional gain because the basic loop is going to have either zero or maybe one iteration in most of the cases. By the way, this can also result in $N = 0$.

- The last case comes when we have $\{\{L, -1\}, N, M\}$, but $N < L+2$. We can only do $N-1$ steps, and the resulting sequence will have a zero. In particular, the end result is $\{\{L+1, L-N+2\}, 1, M+N-1\}$. This takes exactly $\frac{(2L-N+4)N}{2}$ steps.

Whenever we get $N \leq 1$ after each of these steps, we need a "change of perspective", as explained above: increase the length of the sequence (updating Z if needed), do $N \leftarrow M$ and retrieve the next bowl's value (possibly zero) to assign it to M. If the new N is also ≤ 1, and we have finished the actual set of bowls, then the algorithm finishes. The total amount of steps is the result of the problem, of course. This has about $O(X)$ steps, and it's therefore very fast, about 10 milliseconds. I lost some time because of very stupid overflow issues, but aside from that, the code is simple and not too long.

I wouldn't have given this problem a 65 % rating (although it's true that older problem often have higher rating than current ones, at the same level of difficulty), although it's definitely harder than 335, which also has 65 %.

In the problem thread there are all kind of interesting variations, like closed formulas, studies determining the position of the zero using mass center distributions, and so on.

335. Gathering the beans

Difficulty rating: 65 %.
Solution: 5032316. *Solved: Sun, 12 Jun 2022, 09:41.*
Math knowledge used: recurrence equations.
Programming techniques used: binary exponentiation.

Like 666, this is a problem that looks intimidating, but then you start working on it and... what? That's it? Is it really that simple? Indeed, this problem is even easier than 666. Like that one, I'm assuming that the high problem rating must come from the reticence to tackle it. I know that the first time I met this problem was back in the day, during my first run, and upon seeing this problem I thought that this was basically impossible for someone like me. I never really thought again about this problem, until this morning, when I saw it and thought... well, maybe it's not so impossible? An analytical solution looks impossible, but maybe there is an easy pattern?

So I ran some simulations and... didn't find a clear pattern at all. At least, not for the general value. Although I did confirm the two sample values, confirming that the simulation was right. But I noticed some weird common patterns:

- $M(x)$ is a strictly increasing function. This sounds reasonable, but it's not at all a given. I wouldn't have been very surprised if this hadn't been the case.
- The increases are mostly smallish, but from time to time, we can see a value of $M(x)$ that is more than twice the value of $M(x-1)$.
- In fact, save for the case $M(2) = 2$, $M(3) = 5$, this happens exactly when $x = 2^n + 2$ for some n. In fact, most surprisingly, $x = 2^n + 1$ is the only case where $M(x)$ seems to be odd; it's even in any other case.
- Aside from that, increments are kind of erratic. There is some semblance of a pattern, but I didn't really study it.

Well, it so happens that these odd values at $x = 2^n + 1$ are precisely the ones being asked by the problem. Maybe they follow a simpler pattern? Let's define $f(k) = M\left(2^k + 1\right)$ and plot some values.

$$f(0) = M(2) = 2;$$

$$\begin{aligned} f(1) = M(3) &= 5; \\ f(2) = M(5) &= 15; \\ f(3) = M(9) &= 53; \\ f(4) = M(17) &= 207\ldots \end{aligned}$$

Nothing very clear, but it seems that each $f(k)$ is approximately 4 times $f(k-1)$. Good? So I tried looking at the value $f(k) - 4(k-1)$, and I found a succession that looked like this:

$$\{-3, -5, -7, -5, 17, 115, 473, \ldots\}$$

I didn't know what to expect, but just in case, I referred to OEIS, which helpfully pointed out that this was the sequence A214091, which corresponds to... $3^n - 2^{n+2}$. That's it, not even some combinatorial weirdness or some magical succession from an obscure theorem. Just that difference between exponents.

And this basically solved the problem. Because now we know that

$$f(k) = 4f(k-1) + 3^k - 2^{k+2}.$$

This immediately implies that

$$f(k) = A \cdot 4^k + B \cdot 3^k + C \cdot 2^k + D;$$

I wasn't sure about the term for D, but I included it anyway just in case. Now, using the first four values of f we can immediately see that

$$\begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 4 & 3 & 2 & 1 \\ 16 & 9 & 4 & 1 \\ 64 & 27 & 8 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 5 \\ 15 \\ 53 \end{pmatrix},$$

which gives $A = 1$, $B = -1$, $C = 2$, $D = 0$. Therefore,

$$f(k) = 4^k - 3^k + 2^{k+1}.$$

Now of course I don't actually want this value. What I would really like is having a formula for

$$g(k) = \sum_{i=0}^{k} f(i).$$

Basic recurrence theory says that, again, we have

$$g(k) = A \cdot 4^k + B \cdot 3^k + C \cdot 2^k + D$$

for some other unique solution to a linear system. In particular, we now have

$$\begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 4 & 3 & 2 & 1 \\ 16 & 9 & 4 & 1 \\ 64 & 27 & 8 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 7 \\ 22 \\ 75 \end{pmatrix},$$

a system whose solution is $A = \frac{4}{3}$, $B = -\frac{3}{2}$, $C = 4$, $D = -\frac{11}{6}$. So here it is, an analytical expression for g:

$$g(k) = \frac{8 \cdot 4^k - 9 \cdot 3^k + 24 \cdot 2^k - 11}{6}.$$

With binary exponentiation, this can be calculated in $O(\log k)$ time. I also need the value of 6^{-1} modulo 7^9, which is as direct as it gets. So there it is, an apparently insurmountable and abstruse problem whose solution I could find in about half an hour from beginning to end, and with a run time of 6 milliseconds.

356. Largest roots of cubic polynomials

Difficulty rating: 60 %.

Solution: 28010159. *Solved: Wed, 30 Dec 2020, 03:38.*

Math knowledge used: eigenvalue theory, characteristic polynomials, Cardano-Vieta formulas.

Programming techniques used: binary exponentiation.

Like with many other problems, the first time you see the description of this one, you might think that they're pulling your leg. But there is a very fast way to solve this problem, and it's done detouring from real analysis to linear algebra. The solution of this one is $O(n \log k)$ where n is the amount

of solutions to calculate, and k is the exponent. In other words, ridiculously fast since there are only 30 equations to solve.

The first hint to the problem (as pointed out by pipilupe) is that the problem talks about the "largest" root. Where are the others? Do they matter? Are they even real? The answers to both of these last two questions is yes. For the other one, we'll see a full explanation shortly. And then there is the second, very important hint, which is the usage of the $\lfloor \cdot \rfloor$ operator. This suggests staying in the domain of integer numbers, and it's indeed what we need to do.

Now, let's actually solve the problem. First of all, we need to give an approximate solution to the main question, which is: where are all the roots of the polynomial? And after that, we'll see how to get the power of a real solution without calculating the solution itself. In order to locate the roots, we can evaluate the polynomial at the following values: $\{-1, 0, 1, 2^n - 1, 2^n\}$. We get the following values: $g(-1) = n - 1 - 2^n$; $g(0) = n$; $g(1) = n + 1 - 2^n$; $g(2^n) = n$; and for $2^n - 1$ we can use Newton's binomial theorem: $x^3 = 2^{3n} - 3 \cdot 2^{2n} + 3 \cdot 2^n - 1$, and $x^2 = 2^{2n} - 2^{n+1} + 1$; therefore,

$$g(2^n - 1) = 2^{3n} - 3 \cdot 2^{2n} + 3 \cdot 2^n - 1 - \left(2^{3n} - 2^{2n+1} + 2^n\right) + n = -2^{2n} + 2^{n+1} + n - 1.$$

With these values, we can easily see that for $n > 1$ there are sign changes of $g(x)$ in the intervals $(-1, 0)$, $(0, 1)$ and $(2^n - 1, 2^n)$. For the special case $n = 1$, which can be solved separately by any math program from the last 30 years, there is one solution in the $(-1, 0)$ range and another in the $(2^n - 1, 2^n) = (1, 2)$ range. The other solution is exactly at $x = 1$.

And now comes the magic, in the form of eigenvalue theory. Without loss of generality, let's call a_n to the root we need, b_n to the one in the $(0, 1]$ range, and c_n to the one in the $(-1, 0)$ range. The trick is the following: if we have a matrix whose eigenvalues are $\{a_n, b_n, c_n\}$, we can calculate the N-th power (for $N = 987654321$, the exponent of the problem) of that matrix, and its eigenvalues will be $\left\{a_n^N, b_n^N, c_n^N\right\}$. Of course, the tool to convert the polynomial into a matrix, and vice versa, is the characteristic polynomial of a matrix. But wait, this doesn't solve the problem. These matrices are huge, how do we extract specifically a_n^N from them? And now here comes another very handy bit of eigenvalue theory: the trace of a matrix always equals the sum of its eigenvalues. In our case, with N being an odd number, we will also have $b_n^N \in (0, 1]$ and $c_n^N \in (-1, 0)$ (if N were an even number, c_n^N would be positive). Also, since $a_n < 2^n$, and the sum of the three solutions will equal to the coefficient of x^2 negated (as per Cardano-Vieta), i.e. 2^n, this means that $b_n + c_n > 0$, which means $b_n > |c_n|$. This implies that $b_n^N > \left|c_n^N\right|$, therefore $b_n^N + c_n^N > 0$. In fact, since b_n^N and c_n^N will be ridiculously minuscule

numbers, it's very safe to say that $b_n^N + c_n^N \in (0, 1)$. So, if we have some value $S = a_n^N + b_n^N + c_n^N$, then $\lfloor a_n^N \rfloor = S - 1$. Boom, away with real numbers (since the trace is always going to be an integer number). Welcome back, integers.

We now have enough information to write the algorithm. The first thing we need to do is to write the polynomial in a matrix form, in such a way that the resulting matrix M has the roots of $g(x)$ as its eigenvalues. This is straightforward and well known: given that the coefficients of $(1, x, x^2)$ are $(n, 0, -2^n)$ respectively, a matrix we can use is

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -n & 0 & 2^n \end{pmatrix}. \tag{1}$$

We will just need to calculate M^N with the appropriate mod, which can be done in $O(\log N)$ using binary exponentiation, sum its main diagonal terms to obtain the trace, and subtract one: the result will be $\lfloor a_n^N \rfloor$. The run time is a satisfying 3ms.

360. Scary Sphere

Difficulty rating: 50 %.

Solution: 878825614395267072. *Solved: Fri, 17 Jun 2022, 04:09.*

Math knowledge used: Hermite-Serret algorithm, square sum enumeration, "first prime" Erathostenes Sieve.

Programming techniques used: memoisation.

Oooh, scary. Yes. Like many others, this problem looks scary until you see the trick you needed. It's interesting, in that the math knowledge required, while somewhat obscure for a standard math graduate's level of knowledge, is more or less commonplace in project Euler. However, for this problem, if you try doing it directly, the run time is atrocious; I haven't calculated it properly, but it's likely to be on the level of several hours. My end run time, on the other hand, was about 6 seconds, because I saw the trick.

Basic operations: we know how to enumerate the ways to express a number as the sum of *two* squares, which takes more or less logarithmic time, or maybe $O\left(\log^2 n\right)$, not sure. This is assuming that we already know the prime decomposition of the number we're working with. This suggests a naive approach: iterate from $z = 0$ to $N = 10^{10}$, and for each number, decompose it in prime factors (not so easy: we're talking 10^{20} here), and call the algorithm repeated times for $x^2 + y^2 = N^2 - z^2$. The problem, as mentioned before, is

that this is a very slow bruteforcey algorithm. So we can analyse a bit this idea, to realise that, for every odd z, it so happens that $N^2 - z^2$ is of the form $4k + 3$ and therefore it can't be expressed as the sum of two squares. Very nice. We can use only even values of z. Ok, this helps us reduce the numbers; we might use a limit of $5 \cdot 10^9$, count all the numbers, and multiply the result times 2. Except that, well, $5 \cdot 10^9$ is still a multiple of 4, so we can apply the same reasoning. And 10^{10} is a multiple of 2^{10}, so we are going to be able to use this line of reasoning over and over a few times. In fact, after using it 9 times, we get to the number $M = 2 \cdot 5^{10}$, which is congruent with 2 modulo 4. This means that $M^2 - k^2$ is going to be congruent with either 0 or 1 modulo 4, so we now need to consider every number. Very neat. Trick found successfully. We can reduce the problem to three separate steps:

- Calculate the points on the disk $z = 0$. This is a standard check for square sums for M.
- Calculate the points on the poles. There are exactly two: $(0, 0, M)$ and $(0, 0, -M)$. Therefore this adds a total of $2M$ to the final result.
- Calculate all the remaining points by iterating for $z = 1$ up to $M - 1$. This must be added twice in order to consider the negative values of z as well.

We are going to decompose a lot of values of the form $M^2 - z^2$; considering that M is a bit below $2 \cdot 10^7$, this means numbers around $4 \cdot 10^{14}$. Fortunately, $M^2 - z^2 = (M + z)(M - z)$, so decomposing these numbers is a matter of decomposing $M - z$ and $M + z$ separately and combining the result. This can be done using a "first prime" sieve up to $2M$, which should take less than one second. So, for every z, we do the following:

- Decompose $M + z$ and $M - z$ into prime factors, which should be very fast with the help of the prime sieve.
- Return early if $M + z$ or $M - z$ contains a prime factor of the form $4k + 3$ with an odd exponent. Note that no prime factor of this form can be present in both $M + z$ and $M - z$. I used a more complicated approach in order to accommodate for the sample case 45, which is a multiple of 3, but it's not actually needed.
- Combine both decompositions in order to get the one for $M^2 - z^2$.
- Use the prime decomposition to get the square sum decomposition, using the same Hermite-Serret plus memoisation magic from several alike problems. Feel free to extract the members of the form $(4k + 3)^{2i}$ as a common factor to apply it at the end.

- Now, consider two variables, sum and count. Fill them iterating over the square pairs:
  - Let (x, y) be a pair. My code guarantees that $0 \leq x \leq y$, which makes this slightly easier.
  - If $x = 0$ or $x = y$, this pair appears 4 times. Otherwise, it appears 8 times. Let's call this number r.
  - Increase count in r, and sum in $r\,(x + y)$.
- If you extracted the $4k + 3$ factors, apply it now to the sum. The total amount of coordinates for the disk at z is the sum of the "sum" variable plus the product of z times the count.

We want to multiply this value times 2 to consider both positive and negative z. For $z = 0$, the problem is similar but even simpler, since $z = 0$ and since M^2 is, blatantly, a perfect square, so we can take some shortcuts.

I could reuse a lot of code from other problems, but even so, there was some logic to implement carefully, and I'm happy to say that I got it right at the very first try. Run time: about 6,3 seconds. Also, this was my 600th problem. Level 24! I never thought I could get this far. Not that I'm going to get much farther, unfortunately...

Most of the approaches from the problem thread are very similar. Some enumerate the different ways of expressing 10^{10} as sum of *four* squares, culling the cases where there isn't any zero and then removing the duplicates. This looks like overkill to me, but the run times reported are not as atrocious as one might fear.

368. A Kempner-like series

Difficulty rating: 45 %.

Solution: 253.6135092068. *Solved: Fri, 21 Jan 2022, 02:14.*

Math knowledge used: numerical series, Kempner series summation, combinatorial numbers, graph theory, automata.

Programming techniques used: dynamic programming, rail switching.

I lost a big amount of time on this problem because of an incredibly stupid bug. Otherwise, once you understand the logic under the Kempner summation series, helpfully laid out in the paper *Summing a Curious, Slowly Convergent Series* from T. Schmelzer and R. Baillie, the scheme used for the Kempner series can be very easily modified to sum the proposed series. The theory is not very complicated: use a graph to represent different statuses,

like in automaton theory, so that each combination of state and next digit results in a new state (possibly the same, although in this particular problem that doesn't happen). Then, for each state and number of digits, store some information related to partial sums of p-harmonic series up to certain power p, which is used to calculate the information on the next state.

The specifics don't get much complicated than that; there is a simple magic formula (actually an infinite series, but one that converges very fast) that can be used to iterate, and that's really all there is. We start by defining a set of statuses $\mathcal{S}$, and then we define the symbol $\psi_{i,k}^{j}$ as the sum of all the terms of the form $\dfrac{1}{n^k}$ for every n that belongs to the status j and which have exactly i digits. We will have, at each time, a set of values corresponding to all the statuses and powers we want and to a certain exact value of i, and we will iterate to calculate the values for $i+1$. The magic formula, whose derivation is on the paper mentioned above, and pretty easy to follow, is

$$\psi_{i,k}^{j} = \sum_{m=0}^{9} \sum_{l \in \mathcal{S}} f_{jlm} \sum_{w=0}^{\infty} a\left(k, w, m\right) \psi_{i-1,k+w}^{l},$$

where m represents the digit we are adding, l is the status we're transitioning from (so that $f_{jlm} = 1$ if status l transitions to j if it encounters the digit m, and 0 otherwise; in practice, this is calculated by iterating over l and adding terms to status j for each digit m), and the series in w is calculated as just the first terms, since it converges very fast. The terms $a\left(k, w, m\right)$, which are precalculated, follow this formula:

$$a\left(k, w, m\right) = 10^{-k-w} \left(-1\right)^{w} \binom{k+w-1}{w}.$$

The vast majority of $\psi_{i,k}^{j}$ is well below 1, so one can see why this series converges so fast.

The specifics of each Kempner-like series are encoded in the graph given by the statuses in $\mathcal{S}$. In this particular problem, we define a set of 20 states, $s_{m,r}$ for $m \in [0, 9]$ and $r \in [1, 2]$. The state $s_{m,r}$ includes numbers where the last digit is m and this last digit is repeated r times at the end of the number. So 399 belongs to $s_{9,2}$, for example. Determining the state a given number belongs to is quite trivial, although I was stupid enough as to have a bug here and lose about one full night of work which could have been much better spent fusing demons in Shin Megami Tensei V. The transitions are also quite evident:

- State $s_{m,r}$ transitions to $s_{n,1}$ when encountering a digit $n \neq m$.

- State $s_{m,1}$ transitions to $s_{m,2}$ when encountering a digit m.
- State $s_{m,2}$ discards the next number when encountering a digit m.

Although there are ways to speed up the algorithm even further (my program takes less than 0,1 seconds, so it's not like optimisations are super necessary), the algorithm I used is based on a constant maximum power P, and it also uses the initial power I and the maximum iteration N as arguments. First, iterate for all the numbers from 1 up to $10^{I-1}-1$ (numbers with less than I digits), adding the inverse to the result if the number is valid (i.e., it doesn't have three repeated digits and therefore it belongs to one of the twenty states above). Then, iterate from 10^{I-1} up to 10^I-1 (numbers with exactly I digits), adding again the inverses to the result, but also storing the sum of the inverses up to power P, calculating in practice the values of $\psi_{I,p}^j$ for every status $j \in \mathcal{S}$ and every power $p \in [1, P]$. After that, iterate for $n = I+1$ to N, using the magic formula above to calculate the values of $\psi_{i,p}^j$ for increasing values of i. The infinite series in w is estimated just by adding all the available terms, from p to P. Keep adding all the values of $\psi_{i,1}^j$, for all i and j, to the result, and that's it.

Using $P = 10$, $I = 4$ and $N = 5000$ is enough to get the correct result, up to the desired amount of digits, in less than one tenth of a second. I kind of regret putting this problem off for so much time, but in the end, it was very reasonably easy and I spent much less time reading and understanding the paper than I was fearing.

380. Amazing Mazes!

Difficulty rating: 60 %.

Solution: 6.3202e25093. *Solved: Sun, 6 Jun 2021, 16:40.*

Math knowledge used: graph theory, Laplacian matrices, Kirchhoff's theorem, block-tridiagonal matrix inversion, matrix determinant properties, Bareiss algorithm.

Programming techniques used: matrix binary exponentiation.

Yes, that's the same Kirchhoff. No, this is not about physics or circuits. Yes, the solution is a 25094 digit number. Yes, I generated it completely using BigIntegers. Also, yes, that's a lot of new things I learnt here, and I've listed them in the order I've applied them. This is a remarkably complicated problem because it has the annoying property of spawning a new, unexpected stopper issue at each single step, forcing you to introduce interesting tricks at each time. That 60 % feels actually low, especially for a kind of early problem, before the difficult 400s and 500s arrive.

If you think about the mazes as a rectangular set of connected squares, so that there is a single path from the start to each of the rest, it's clear that you can treat this as a graph of squares where you are finding spanning trees. So you need to find the total count of possible spanning trees. This is very different from Kruskal's algorithm (see also: problem 107), one of the classic spanning tree problem types. So after some search I found what I was looking for: Kirchhoff's theorem. This theorem says: create the Laplacian matrix of the graph, remove one row or column, calculate the determinant, that's the amount of spanning trees. Much easier said than done, since every step here is problematic (also, take that, Devlin! Determinants are useful after all). Let's go step by step.

First, the Laplacian matrix. Given a graph, assign an index to each node, and each element of the main diagonal will be set to the cardinality of the assigned node. The elements outside the main diagonal are equal to the adjacency matrix, negated. This is a graph and not a multigraph, therefore an element m_{ij} is set to -1 if i and j are connected, 0 if not. Note that this is before the spanning tree examination, meaning that every square is connected to its four neighbours. Ostensibly, each single row and colum amounts a total sum of 0, so the determinant of this matrix is always 0. The size of the matrix is the amount of total squares, so at the case the problem asks for, 100×500, we are looking at a 50000×50000 matrix. Of course it's a sparse matrix and it doesn't fit in memory, but that's just the first of our problems here.

Now, Kirchhoff's theorem. No current or voltage here, just graphs and Laplacian matrices. Kirchhoff's theorem says that if you remove one row and column from the Laplacian matrix (resulting in a regular matrix), and take its determinant, the result is the total count of spanning trees of the graph. Hooray, we can remove an element! So we only need to calculate the determinant of a 49999×49999 matrix! Yay?

We now know what to do but we don't know how to do it. How do you calculate the determinant of such a matrix? Gaussian inversion won't work, we don't have the time or the space. But the matrix has an interesting structure and we can take advantage of that. First of all, clearly the matrix is sparse. The amount of nodes is mn but the amount of adjacencies is smaller than $4\,(m + n)$ (and that's counting each adjacency twice, i.e. once in each direction). Now, most importantly, each row or column only has adjacencies towards the immediately previous or next row or column. This means that we can divide the row/column space of the matrix in blocks so that each block only has nonzero values in the row/column of the immediately previous or next block. This means that the matrix is tridiagonal per blocks. Now, we can divide the matrix in 100 blocks of size 500 or in 500 blocks of size 100. If there is an algorithm to calculate determinants using this structure, we

expect it to be cubic in the size of the block and linear in the amount of blocks, so it will be faster to have many smaller blocks. We opt, then, for the second alternative. In order to calculate the determinant, we refer to the appropriately named paper *Determinants of block tridiagonal matrices* from Luca Guido Molinari. This paper presents a general complicated case, and at the end it contains a reasonably simple formula that can be used by us. For the formula, we first define the main diagonal blocks, $\{A_n : n \in [1 \dots 500]\}$, and the subdiagonal blocks, $\{B_n : n \in [1 \dots 499]\}$ at the right/upper diagonal and $\{C_n : n \in [1 \dots 499]\}$ at the left/lower diagonal. The algorithm first asks us to calculate the transference matrix $T^{(0)}$ as

$$T^{(0)} = \begin{bmatrix} -A_n & -C_{n-1} \\ I_m & 0 \end{bmatrix} \begin{bmatrix} -B_{n-1}^{-1}A_{n-1} & -B_{n-1}^{-1}C_{n-2} \\ I_m & 0 \end{bmatrix} \dots \begin{bmatrix} -B_1 A_1 & -B_1^{-1} \\ I_m & 0 \end{bmatrix}.$$

This is a $(2m) \times (2m)$ matrix, so we take the upper left $m \times m$ corner, $T_{11}^{(0)}$, and then the determinant we are looking for is equal to

$$|M| = (-1)^{nm} \left| T_{11}^{(0)} \right| \cdot |B_1 \cdot \ldots \cdot B_{n-1}| .$$

Ok, this looks doable, especially considering that the sign can be ignored. But there is a problem, because this algorithm expects blocks from equal size, and we have removed one row and column to make the determinant not null. So we are going to need to do some matrix magic. The next step is defining the matrices $\{A_n\}$, $\{B_n\}$ and $\{C_n\}$ we will work with.

We are close to knowing exactly what we are going to calculate, but there are two problems: we have a $(mn-1) \times (mn-1)$ matrix and we need a $(mn) \times (mn)$ one; and we need that the B_n matrices are regular. Now, before defining the final matrices, let's look at the Laplacian matrix. The elements of the main diagonal are equal to 2 in each of the four corners, 3 in every element at each horizontal or vertical border except these corners, and 4 in the rest of the interior. Now, inside each "main" block, it so happens that the A_n matrices are also tridiagonal, since each square is adjacent only to the next and previous one, so the secondary diagonals have every value set to -1. And as for the secondary blocks, since each element is only related to the corresponding element of the previous/next row or column, the base B_n and C_n matrices are just the identity matrices, negated! This is very helpful because we can use that to avoid calculating inverse matrices. Now, with this in mind, we need to transform this structure into a slightly different one that we will be able to use. Let's say that, for example, we remove the last row and column of the 50000×50000 matrix, which means removing the last

row/column of the last A_n, the last row of the last C_n and the last column of the last B_n. This breaks the regularity of the structure and prevents us from using the block tridiagonal algorithm. But if we add a new row and column where all the elements are 0 except for the element in the main diagonal, which is 1, this is like adding an eigenvalue equal to 1 to the matrix, so that its determinant won't change. Great, but we are not quite there. The last row of C_{n-1} is all 0, and most importantly, so is the last column of B_{n-1}, meaning that it's not invertible and we can't use the algorithm. Argh.

Fortunately, we are not going to let this problem stop us in our tracks, now that we are so close to the answer. Matrix determinant properties to the rescue! We can add a row or column to another, multiplied by whichever number we want, and this doesn't change the value of the determinant. So we will do two steps:

- Subtract the last row from the last minus m, so that the last element from B_{n-1} is also -1, making it so that $B_{n-1} = -I$. This doesn't change any other value in the whole matrix and retains the determinant value.
- For symmetry, subtract the last column from the last minus m, so that the last element from C_{n-1} is also -1 and $C_{n-1} = -I$ as well. But by doing this we also change the last minus m element of the last minus m column, meaning that we need to update A_{n-1} so that we **add** 1 to its last element. This is not completely necessary, but it looks better and allows easy reasoning because the $\{B_n\}$ and $\{C_n\}$ matrices are, each single one of them, negated identities.

We can finally start building the matrices necessary for the final calculation. We are going to do a few matrix multiplications, actually not that many, one full determinant, and not a single inversion required thanks to all those submatrices which are actually identities. If we look at the formula of $|M|$, we can easily see that the sign is going to be 1, and more nicely, since each B_n has a determinant equal to $(-1)^m = 1$, we can just use $|M| = \left|T_{11}^{(0)}\right|$. We still need to generate $T^{(0)}$, but that's not going to be difficult. We know that $B_n = -I$ for every n, which also means that $B_n^{-1} = -I$. The simplified transference matrix looks like this:

$$T^{(0)} = \begin{bmatrix} -A_n & I_m \\ I_m & 0 \end{bmatrix} \begin{bmatrix} A_{n-1} & -I_m \\ I_m & 0 \end{bmatrix} \dots \begin{bmatrix} A_1 & I_m \\ I_m & 0 \end{bmatrix}.$$

Bulding the A_n matrices is easy. The first one, A_1, is tridiagonal, with its main elements being all 3, save for the first and last which are 2, and all the

elements from the secondary diagonals are -1. Then, all the matrices from A_2 up to A_{498} are exactly the same (which means: binary exponentiation!), with the same structure as A_1 but with 3 at the initial and final element of the main diagonal and 4 at the rest. Now, A_{499} would have been also the same, but since we have done the trick to regularise B_n and C_n, we will update the last element so that it's also 4 instead of 3. Finally, for A_{500}, the last one, we take the same structure as for A_1, but with changed values at the end row and column. The last element of the subdiagonals is not -1 but 0, and the last element of the main diagonal is not 2 but 1. So we end with the product of 500 matrices of size 200×200, and of those 500, there are 497 that are exactly the same, meaning that can use binary exponentiation and the operation ends being $O(m^3 \log n)$ instead of $O(m^3 n)$.

We just need to calculate that product of matrices, get the upper left quarter, calculate its determinant, and that's it. Are we finished yet? ...no, we haven't, not yet, unfortunately. The mathematical analysis is basically complete, but there are still specifics that are not trivial. The main one is the data type. What do we use? The result is going to be huge. Ints and longs are laughably unsuited for this. Modding is not necessary, nor expected, nor it would be useful anyway because we want the full number. Doubles would look like the obvious option, if not for the fact that we know that their limit of the order of 10^{308} is also way lower than what we need. Now, I guess that there are many ways of doing this, and there is a good chance that my choice is very suboptimal (or, looking from a different perspective, that Java is a particularly bad language for this and that I should have resorted to Mathematica), but I went for BigInteger. Ok? It will take some time, but it's just multiplying a bunch of matrices with (eventually) very high values, extracting the first block, and calculating its determinant. Can we do it yet?

No, no we can't, no. Multiplying, yes, that part is easy. But, determinants? A 100×100 determinant is not so huge, but the usual Gaussian elimination algorithm requiring divisions would force us to use a fraction type, and the denominator might grow too much; it might even not fit the memory. Aargh! Is there an alternative that uses only integer calculations? There is the standard $O(n!)$ algorithm, but, uh, with $n = 100$, I'd rather not. The solution is called Bareiss algorithm and it's, finally, the very last step needed for this. This magical algorithm contains divisions, but in such a way that they are guaranteed to be integer divisions. It's a surprisingly simple triple loop operation which slowly replaces most of the matrix with its minors, eventually calculating its determinant. To avoid copying, I used the 200×200, but with loops limited so that they would calculate the determinant of the 100×100 upper left matrix. And yes, this finally, eventually, after more than 9 minutes, it returns the correct value.

385. Ellipses inside triangles

Difficulty rating: 70 %.

Solution: 3776957309612153700. *Solved: Mon, 15 Nov 2021, 19:31.*

Math knowledge used: Marden's theorem, Cardano-Vieta formulas, binary quadratic forms, recursive successions, linear algebra.

Programming techniques used: none.

The only entry barrier for this problem is Marden's theorem, which I happened to already know about after someone had linked to it on Twitter or something as a curiosity. After that, the pattern is easy to spot and it becomes a matter of finding the solutions to a Pell-like equation and iterate over them.

Marden's theorem is a beautiful and surprising result, which in this case tells us that, if the ellipse has its foci in $\pm\sqrt{13}$, we can treat these points as roots (in the complex plane) of the polynomial $z^2 - 13$, and that by integrating this polynomial we will get another one whose roots are in the vertices of one of the triangles we're looking for. Now, of course any function has infinite antiderivatives, and iterating over the integration constant (we need Gaussian integers, not just integers) is not going to get us anywhere. So this is when we use Cardano-Vieta formulas. As per Marden's theorem, the polynomial is

$$\int \left(z^2 - 13\right)\, dz = \frac{z^3}{3} - 13z + C;$$

Cardano-Vieta suggests that we rewrite this as

$$z^3 - 39z + C$$

where C can be any complex value. A Gaussian integer in our case. But that's a red herring. Precisely because C can be anything, it's best not to focus on that, and use the results from the coefficients that we do know. Let's say that the roots we're looking for are $a+bi$, $c+di$ and $e+fi$. Then, Cardano-Vieta says:

$$\begin{aligned} (a+bi) + (c+di) + (e+fi) &= 0. \\ (a+bi)\,(c+di) + (a+bi)\,(e+fi) + (c+di)\,(e+fi) &= -39. \end{aligned}$$

The second equation looks more complicated, but the first one gives us some very interesting information:

$$e = -a - c,$$

$$f = -b - d.$$

If we replace these values in the second equation, simplify as possible (possibly with Matlab or something like that), and separate the real and imaginary results, we get two quadratic equations in a, b, c and d:

$$\begin{aligned} -a^2 + b^2 - c^2 + d^2 - ac + bd &= -39, \\ -2ab - bc - 2cd - da &= 0. \end{aligned}$$

This is much more manageable and at the very least it allows calculating the correct solution for $N = 1000$ using brute force (iterating over a, b and c, finding d with the second equation, and checking whether the first one holds) in a few seconds. This is enough to see the pattern clearly. I guess that there is room for being rigorous, but it's not so clear. Analysing the solution we see that there are only 34 triangles for $N = 1000$ (32, actually, since I get two colinear results that are not valid triangles), and a lot of them are variations of the same cases. In fact, the analysis reveals that there are only two kinds of solutions:

- Triangles of the form (a, b)-$(a, -b)$-$(-2a, 0)$ for some a, b. There are two variations of each of these, since the sign of a can vary. These are kind of obvious and the problem description even shows one of them.
- Triangles of the form $(5x, 3y)$-$(2x, -4y)$-$(-7x, y)$ for some x, y. These are less obvious. There are four variations of each, by playing with the signs of x and y.
- Surprisingly, there aren't any more.

If we plug these values (at least, the coordinates of the first two points) into the two equations in a, b, c and d, in both cases we get that the second equation is identically null, which is neat. The first equation becomes... a binary quadratic form! A different one for each case. In the first case we get

$$-3a^2 + b^2 + 39 = 0,$$

and in the second one,

$$-3x^2 + y^2 + 3 = 0.$$

Ok, they are different, but they only differ in the constant value. So the recursion is going to be the same. In fact, it's a very simple one (given by Alpertron):

$$a_{n+1} = 2a_n + b_n,$$

$$b_{n+1} \quad = \quad 3a_n + 2b_n.$$

This is enough to calculate all the solutions, but how do we calculate the area? There is an old matricial formula that helps us:

$$A = \frac{1}{2} \left\| \begin{vmatrix} 1 & a & b \\ 1 & c & d \\ 1 & e & f \end{vmatrix} \right\|.$$

Since $a + c + e = 0$ and $b + c + f = 0$, we can simplify a bit:

$$A = \frac{1}{2} \left\| \begin{vmatrix} 1 & a & b \\ 1 & c & d \\ 3 & 0 & 0 \end{vmatrix} \right\| = \frac{3}{2} \left\| \begin{vmatrix} a & b \\ c & d \end{vmatrix} \right\|.$$

We can simplify further, since we have more explicit expressions for the two cases we have identified. In the first one we have

$$A = \frac{3}{2} \left\| \begin{vmatrix} a & b \\ a & -b \end{vmatrix} \right\| = \frac{3}{2} \left| -ab - ab \right| = 3ab.$$

And similarly, in the second one:

$$A = \frac{3}{2} \left\| \begin{vmatrix} 5x & 3y \\ 2x & -4y \end{vmatrix} \right\| = \frac{3}{2} \left| -20xy - 6xy \right| = 39xy.$$

The only additional piece of information, again given by Alpertron, is the base solution. We have two for the symmetric case, $\{a = 4, b = 3\}$ and $\{a = 5, b = 6\}$, and one for the other one, $\{x = 2, y = 3\}$ (technically the base solution is $\{x = 1, y = 0\}$, but this is not valid; this is the colinear "triangle" I mentioned above).

The algorithm is very direct:

- Iterate for each one of the three base cases.
- For each one, calculate the area with the appropriate formula.
- Accumulate the result of the area, multiplied by the appropriate symmetry value (2 for the simple case, 4 for the other).

- Calculate the next triangle using the recursion.
- Finish if the triangle is too big (with $L = 10^9$, the condition is $2a > L$ or $b > L$ in the first case, and $7x > L$ or $4y > L$ in the second); otherwise keep iterating.

I took the opportunity to do some overengineering with abstract classes and so on, just because I wanted to, but really, the code is just too simple. Even with my excesses, the code takes about 80 lines include blanks. The algorithm is clearly logarithmic, so the run time is on the order of 40 milliseconds.

389. Platonic Dice

Difficulty rating: 30 %.
Solution: 2406376.3623. *Solved: Wed, 5 Apr 2017, 18:16.*
Math knowledge used: probability theory.
Programming techniques used: none.

I remember that the first time I confronted this problem, I was stumped for a while. This makes sense only because I didn't know a particular formula that I needed for this. In fact, the problem is very easy, and my issues came from the fact that I was trying brute force approaches. The correct way to do this is in a "reversed" approach where first we calculate the data for the later dice, and we work backwards up to the first one. There are only two formulas needed for this problem, and they are very simple.

- Let $X_1, X_2, \ldots, X_n$ be n independent random variables following respectively distributions with expected values $\overline{E}_1, \overline{E}_2, \ldots, \overline{E}_n$ and variances $\sigma_1^2, \sigma_2^2, \ldots, \sigma_n^2$. Then, the random variable $X = \sum_{i=1}^{n} X_i$ has expected value $\sum_{i=1}^{n} \overline{E}_i$ and variance $\sum_{i=1}^{n} \sigma_i^2$. In particular, if all the random variables follow the same distribution with expected value $\overline{E}$ and variance σ^2, then the sum of them all has expected value $n\overline{E}$ and variance $n\sigma^2$.
- Let $X_1, X_2, \ldots, X_n$ be n random variables with, again, expected values $\overline{E}_i$ and variances σ_i^2. Now, Let X be a random variable whose value is chosen from the random variable X_i with probability p_i (having $\sum_{i=1}^{n} p_i = 1$). Then, then expected value is $\overline{E} = \sum_{i=1}^{n} p_i \overline{E}_i$, but the variance follows

a more complicated formula, $\sigma^2 = \sum_{i=1}^{n} p_i \left(\sigma_i^2 + \overline{E}_i^2 - \overline{E}^2\right)$. By the way, this is called a *mixture distribution*.

This last formula is the tricky one I didn't know. It's actually easy to figure out, but without knowing it, I spent a lot of time searching elsewhere, until I looked for a formal, mathematical approach (as opposed to, say, a computational one) and I found these formulas. Now, with these concepts in mind, it's easy to work from the "later" dice and end with the final distribution of all of them. We can start with the distribution for the icosahedral die, which is just a discrete uniform distribution of the integer range $[1, 20]$. This distribution, I, has an expected value of $E_I = \frac{21}{2}$, and a variance of $\sigma_I^2 = \frac{20^2 - 1}{12} = \frac{133}{4}$.

From the icosahedral distribution we can calculate the distribution of the dodecahedron. That is, if we throw a dodecahedral die, and then we throw an icosahedral die as many times as the dodecahedral die indicated, which is the distribution of the resulting sum? Ok, we don't need to calculate the whole distribution: we only need the expected value and variance. So, if we were to throw n icosahedral dice, the resulting sum would have an expected value $I_n = n\overline{E}_I$ and a variance $n\sigma_I^2$ (as deduced from the first formula). But the amount of times we are going to throw the icosahedral die comes from a random variable: we will throw $1, 2, \ldots$ or 12 dice, with a probability of $\frac{1}{12}$ to choose each amount. If we use the second formula above, it's easy to determine the expected value and variance of the sum of the icosahedral dice for a sequence of dodecahedron→icosahedron. We can call this distribution D, with expected value $\overline{E}_D$ and variance σ_D^2. Actually, it is $\overline{E}_D = \frac{273}{4}$ and $\sigma_D^2 = \frac{24479}{16}$.

We can continue the procedure, and at each time we get a summarised distribution of the total icosahedral sum (just the expected value and variance, which is all we need). The procedure is immediate for the remaining three levels. Adding octahedral dice, we get $E_O = \frac{2457}{8}$ and $\sigma_O^2 = \frac{2005731}{64}$; the next is the hexahedral die, giving $E_H = \frac{17199}{16}$ and $\sigma_H^2 = \frac{98510139}{256}$. Finally, the tetrahedral die gives $E_T = \frac{85995}{32}$ and $\sigma_T^2 = \frac{2464129395}{1024} = 2406376{,}3623046875$, which is the solution to the problem. I used BigInteger fractions just in case, but standard longs are more than enough. In fact, even doubles would be good enough, since the numbers aren't big enough to cause

precision issues, and in fact, all of the fractions involved have small powers of two as denominators, meaning that they can be represented exactly as floating point numbers.

In the problem thread there are all kinds of wacky approaches, including dynamic programming (I wonder if what I did could be considered dynamic programming, actually?), Montecarlo simulations and even generating functions.

390. Triangles with non rational sides and integral area

Difficulty rating: 60 %.

Solution: 2919133642971. *Solved: Thu, 11 Nov 2021, 18:27.*

Math knowledge used: Heron's formula, binary quadratic forms (includes convergents, Tonelli-Shanks, Hensel's Lemma).

Programming techniques used: Alpertron (includes a prime sieve).

That's right, this has been the first problem where I used my implementation of Alpertron's QUAD solver. After having coded it a year ago, without daring to try 261 seriously, I found a problem where the solution seemed to be based on solving multiple quadratic diophantine equations with two variables. So I used it and, lo and behold, I got the right solution on my first try. Even better, the program was quite fast, getting an answer in a bit less than a minute despite needing to solve more than 70000 equations. As a bonus, I got a formula for the area of a triangle that I like more than Heron's formula, although it's a little more complex.

Now, the problem itself. We have triangles with sides $x = \sqrt{1 + b^2}$, $y = \sqrt{1 + c^2}$ and $z = \sqrt{b^2 + c^2}$, for any positive integers b and c. Two triangles are considered to be the same if their sides have the same values, no matter the order; so we add the additional condition $b \leq c$, which is not explicit in the problem definition but which must be taken into account. Later I will prove that, in fact, the working condition is $b < c$ because $b = c$ never results in an integer area. It's not a big deal, mostly because we won't iterate over b and c (we do something that is very close to iterating on b, though).

To start solving the problem, the first thing is to find a formula for the triangle's area. Since we have the sides, and not the base and the height, the ideal choice seems Heron's formula,

$$\begin{aligned} s &= \frac{x + y + z}{2}, \\ A &= \sqrt{s\,(s - x)\,(s - y)\,(s - z)}. \end{aligned}$$

If we unroll the value of s, we get a different formula that works directly with x, y and z, and which has some exploitable symmetries:

$$A = \sqrt{\frac{(x+y+z)\,(-x+y+z)\,(x-y+z)\,(x+y-z)}{16}}.$$

Unrolling this formula looks dreadful at the beginning, since there are $3^4 = 81$ terms, but fortunately we can exploit the symmetry, since there are only a few separate cases. If we group the possible combinations without considering the sign, like $xxyz$, we see that they repeat often, and we only need to check the possible signs of the product and add them all. Even better, combinations have variants, since we know that, for example, $zzxy$ will have the same coefficient as $xxyz$, and so on. These are all the cases:

- $xxxx$: there is one single combination, and three possible cases (the other two being $yyyy$ and $zzzz$). This only combination has negative sign, so the coefficient is -1.
- $xxxy$: there are 4 cases, and 6 possible combinations of sides. Out of these 4 cases, two are positive and two are negative, so the coefficient for this is 0. Hooray!
- $xxyy$: there are 6 cases, and 3 possible combinations of sides. 4 of these cases result in a positive sign, and the other two in a negative sign, therefore the coefficient for these forms is 2.
- $xxyz$: there are $\binom{4}{2,1,1} = 12$ cases, with 3 combinations of variables. The great news is that there are 6 positive and 6 negative terms, resulting in a final coefficient of 0.

Since $1{\cdot}3{+}4{\cdot}6{+}6{\cdot}3{+}12{\cdot}3 = 81$, this covers all the possible cases. Expanding the nonzero terms we get a different formula, which is, yes, totally equivalent to Heron's (therefore it's equal to the area of the triangle), and which I personally like better:

$$A = \sqrt{\frac{2\,(x^2y^2 + x^2z^2 + y^2z^2) - (x^4 + y^4 + z^4)}{16}}.$$

Conveniently, all the sides appear squared. This has a curious corollary, which is not really obvious when trigonometry is around: if you have a triangle whose sides are square roots of a rational number, then the area will always be the square root of a rational number. Ok, so we now need to find values of b and c that make this area an integer number. Let's focus on the numerator

of the formula, $N = 16A^2$. We can expand this numerator in terms of b and c:

$$\begin{aligned} N &= 2\left(\left(1+b^2+c^2+b^2c^2\right)+\left(b^2+c^2+b^4+b^2c^2\right)+\left(b^2+c^2+b^2c^2+c^4\right)\right)+ \\ &- \left(\left(1+2b^2+b^4\right)+\left(1+2c^2+c^4\right)+\left(b^4+2b^2c^2+c^4\right)\right). \end{aligned}$$

This is considerably more annoying to reroll than to unroll, but even so, it's still easy if we group by terms and add coefficients:

- The coefficient for 1 is $2-1-1=0$.
- The coefficient for b^2 is $2+2+2-2=4$, and, by symmetry, the coefficient for c^2 must also be 4.
- The coefficient for b^2c^2 is also $2+2+2-2=4$.
- Finally, the coefficient for b^4 is $2-1-1=0$, and therefore the coefficient for c^4 is 0 as well.

The dreadful formulas don't look so dreadful any more: we have proven that

$$N = 4\left(b^2+b^2c^2+c^2\right),$$

which means that

$$A = \sqrt{\frac{b^2+b^2c^2+c^2}{4}}.$$

This is an awesome simplification, and we can do even better, by removing the denominator. It's clear that the numerator will be a multiple of 4 if both b and c are even, but what happens in the other cases? If both are odd, then the numerator is the sum of three odd numbers and therefore also odd, so it can't be a multiple of 4. If one is even and the other is odd, then out of the three addends we have an odd one and two even ones, resulting again in an odd number that will never be a multiple of 4. So we can force b and c to be even, by doing $b = 2m$ and $c = 2n$ (and we can keep the condition $b \leq c$ as $m \leq n$). This results in

$$A = \sqrt{\frac{4m^2+4n^2+16m^2n^2}{4}} = \sqrt{m^2+n^2+4m^2n^2},$$

which is very close to our final equation. We can now observe that, if $m = n$, then the radical is

$$m^2+m^2+4m^4 = 2m^2+4m^2 = 2m^2\left(2m^2+1\right).$$

Since $m > 0$, this is the product of two positive consecutive numbers, which we know that can never be a perfect square. So there are no solutions where $m = n$.

Back to the problem, we can reformulate the equation as

$$m^2 + n^2 + 4m^2n^2 = x^2$$

for some real value x which, conveniently, is the value that cannot surpass the established limit of 10^{10}. Now, this is not an Alpertron-ready equation, but what if we treat m as a constant, M?

$$\left(4M^2 + 1\right) n^2 - x^2 + M^2 = 0.$$

This is a quadratic binary form in n and x, which we can solve with the Alpertron implementation. I tried looking for some more specific solutions iterating from the base (but invalid) case $n = 0$, $x = M$; but since there are additional solutions, the full Alpertron must be used in order not to miss possible cases. Ok, so we are going to need to solve an equation for each value of m. And where can we stop? To answer this we note that the lowest acceptable value of n for each m is m itself (or $m+1$), and it basically means that the smallest solution will always be bigger than $4m^4$. So we use the limit of

$$L_m = \sqrt[4]{\frac{L^2}{4}} = \sqrt{\frac{L}{2}},$$

which is a comfortable 70710 for the problem goal $L = 10^{10}$. The analysis stage finishes here and we can start coding a very simple algorithm:

- Iterate m from 1 to $L_m = \sqrt{\frac{L}{2}}$.
- For each value of m, call Alpertron to solve the equation $(4m^2 + 1)\, n^2 - x^2 + m^2 = 0$ in n and x.
- For each recursive solution, discard negative ones and those with $n < m$; finish if $x > L$. Otherwise, store the value of x.
- The solution of the problem is the sum of all the x values.

It can be argued that this problem has a very heavy amount of coding needed, but I had it already, so I finished it in a few hours, mostly dedicated to finding the final formula of the area. In total, there are just 1218 valid solutions, and the biggest value of m I actually need is 16256, needed for the case where $n = 259076$ and $x = 8423078916$. The biggest value of n is much higher, as

expected: 2403763488, needed for a case where $m = 1$ and $x = 5374978561$. Finally, the biggest area found is 9993949341, which is found for the case $m = 1077$ and $n = 4639716$.

The solution takes a bit less than one minute (it would have taken more in my old PC, though...), which is a big success. Now let's see if I can confront 261 or not (spoiler alert: I could).

394. Eating pie

Difficulty rating: 55 %.

Solution: 3.2370342194. *Solved: Sun, 31 Jul 2022, 07:40.*

Math knowledge used: integro-differential equations, Cauchy-Euler equations, basic statistics.

Programming techniques used: none.

Another one of those problems whose solution is a real number and not an integer, and which is, fortunately, easier than it looks. The obvious way to solve it is using integrals of progressively higher order (a double integral for terms for which there are two steps, a quadruple integral for terms indicating three steps, and so on), and add them all. There is, however, a much better way of doing it, using integral equations which result in a very simple closed formula, which could have been used for (say) $x = 10^{18}$, but the problem uses $x = 40$ so that the more bruteforcey approach is feasible.

Let's take a look at that integral equation. If we look at the pie at each point as a fraction of the unit circle, it's easy to see that for a given fraction of remaining pie we must have an expected value of the amount of steps required. And we can see that this expected value can be defined recursively using integrals. So, given a fixed value of $x = \dfrac{1}{F}$, and any amount of remaining pie, $r \in [0, 1]$, we can define the expected value, $e_x(r)$, like this:

$$e_x(r) = \frac{1}{r^2} \int_0^r \int_0^r (1 + e_x(\text{mín}\{r_1, r_2\}))\, \mathrm{d}r_1\, \mathrm{d}r_2.$$

r_1 and r_2 are, of course, the two points chosen at random. Note that the space has size r^2, hence the factor $\dfrac{1}{r^2}$ to normalise the probabilities. We have an initial condition, $e_x(r) = 1$ for $r \leq f$; and if we assume that this function is differentiable, which seems to make sense given its definition, we can also use $e'_x(r) = 0$ for $r \leq f$.

Well, okey. This is an integral equation and we can start solving it right now. We even have two boundary conditions. So, the first thing we do is

isolating the e_x inside the integral, like this:

$$r^2 e_x(r) = r^2 + \int_0^r \int_0^r e_x(\text{mín}\{r_1, r_2\})\, \mathrm{d}r_1\, \mathrm{d}r_2.$$

This "minimum" looks annoying, but now we can exploit symmetries. We have two separate cases: either $r_1 < r_2$ or $r_2 < r_1$. The case $r_1 = r_2$ has a probability density of 0 so it doesn't contribute to the expected value. So the integral equals the integral of one of the cases (I will choose $r_1 < r_2$, without loss of generality), multiplied times 2:

$$r^2 e_x(r) = r^2 + 2\int_0^r \int_{r_1}^r e_x(r_1)\, \mathrm{d}r_2\, \mathrm{d}r_1.$$

Very happily, the term related to e_x does not depend on the inner integrating variable, so we can, well, integrate normally. Which is dead easy, of course. So we have:

$$r^2 e_x(r) = r^2 + 2\int_0^r (r - r_1) \cdot e_x(r_1)\, \mathrm{d}r_1.$$

For the following steps, let's rewrite this expression a bit:

$$r^2 e_x(r) = r^2 + 2r\int_0^r e_x(r_1)\, \mathrm{d}r_1 - 2\int_0^r r_1 \cdot e_x(r_1)\, \mathrm{d}r_1.$$

The integral equation is almost ready to be solved. We are now going to differentiate twice to make it into a differential equation. First:

$$2re_x(r) + r^2 e_x'(r) = 2r + 2\int_0^r e_x(r_1)\, \mathrm{d}r_1 + 2re_x(r) - 2re_x(r).$$

Very conveniently the last two terms simplify into 0, so we are left with

$$2re_x(r) + r^2 e_x'(r) = 2r + 2\int_0^r e_x(r_1)\, \mathrm{d}r_1.$$

We differentiate again:

$$2e_x(r) + 2re_x'(r) + 2re_x'(r) + r^2 e_x''(r) = 2 + 2e_x(r).$$

So,

$$r^2 e_x''(r) + 4re_x'(r) = 2.$$

There are several methods to solve this, but what I did was differentiating again to get a homogeneous equation:

$$r^2 e_x'''(x) + 6re_x''(r) + 4e'(r) = 0.$$

Hooray! This is a Cauchy-Euler equation (we can multiply it times r to make it more clear, but it's not necessary), and as such it can be solved by doing $e_x(r) = r^m$. The first derivatives are

$$\begin{aligned} e_x'(r) &= mr^{m-1}, \\ e_x''(r) &= \left(m^2 - m\right) r^{m-2}, \\ e_x'''(r) &= \left(m^3 - 3m^2 + 2m\right) r^{m-3}. \end{aligned}$$

The equation is transformed into

$$\left(\left(m^3 - 3m^2 + 2m\right) + 6\left(m^2 - m\right) + 4m\right) r^{m-1} = 0 \Rightarrow m^3 + 3m^2 = 0.$$

We have one single solution $m = -3$ and a double solution $m = 0$. Therefore the solutions of the original differential equation are of the form $e_x(r) = a + b\log r + \dfrac{c}{r^3}$. We can plug this directly into the nonhomogeneous equation, getting

$$r^2\left(-\frac{b}{r^2} + \frac{12c}{r^5}\right) + 4r\left(\frac{b}{r} - \frac{3c}{r^5}\right) = 2 \Rightarrow -b + 3b = 2 \Rightarrow b = \frac{2}{3}.$$

For the other two coefficients, we can just use the definitions directly. We have $e\left(\dfrac{1}{x}\right) = 1$ and $e'\left(\dfrac{1}{x}\right) = 0$, that is:

$$\begin{aligned} a + \frac{2}{3}\log\frac{1}{x} + cx^3 &= 1, \\ \frac{2}{3}x - 3cx^4 &= 0. \end{aligned}$$

From the second equation we immediately see that $c = \dfrac{2}{9x^3}$, and substituting in the first one, we get

$$a + \frac{2}{3}\log\frac{1}{x} + \frac{2}{9} = 1 \Rightarrow a = \frac{7}{9} - \frac{2}{3}\log\frac{1}{x} = \frac{7}{9} + \frac{2}{3}\log x.$$

With these coefficients, we finally have the full solution of the equation:

$$e_x(r) = \frac{7}{9} + \frac{2}{3}\log x + \frac{2}{3}\log r + \frac{2}{9x^3r^3}.$$

The solution of the problem is $E(x) = e_x(1)$, the expected value at the start of the procedure when we have all the cake ($r = 1$). That is:

$$E(x) = \frac{7}{9} + \frac{2}{3}\log x + \frac{2}{9x^3}.$$

This simple function matches all the sample values, and plugging in $x = 40$ we get the solution to the problem.

397. Triangle on parabola

Difficulty rating: 70 %.
Solution: 141630459461893728. *Solved: Tue, 23 Aug 2022, 16:03.*
Math knowledge used: algebraic geometry, inclusion-exclusion.
Programming techniques used: none.

This is a curious problem. How do people come up with these? It doesn't require any obscure knowledge (and the "inclusion-exclusion" I mention is extremely simple), yet the problem is actually very hard. You need a lot of derivations and then a sensible iteration scheme (which I didn't really find). After having solved it, that 70 % sounds about right to me. In fact my solution is pretty crappy. I'm only writing this because this is likely to be my last problem solved.

So, let's start. There are triangles with one 45 degree angle, and triangles with two 45 degree angles. Any scheme that counts the first ones by batch will include the ones from the second twice, so we need to get the count of the angles with two 45 degree angles and subtract them from the total count. Kind of surprisingly, these "double" triangles are easier to count, mostly because they are so regular. By the way, this subtraction is the only inclusion-exclusion required. Everything else is basic geometry and a lot of derivations. And I really mean a lot.

We start with the "single" triangles (the ones with a single 45 degree angle). Without loss of genericity, let's say that B is the point where this angle lies. Algebraic geometry says that this is equivalent to saying that

$$\frac{AB \cdot BC}{|AB| \cdot |BC|} = \frac{1}{\sqrt{2}}.$$

The first problem is that these lengths are not going to be integers, so we need to square the result in order to get a purely integer equations; which is a pity, since it introduces a spurious solution (with angles of 135 degrees, which we don't want). We can just force the numerator to be positive. More on that later. For now we can plug the coordinates of points A, B and C as described by the problem (we can ignore the order of the points for now), and Mathematica or Matlab will give us this:

$$k = \pm\frac{c-a}{2} \pm \frac{\sqrt{c^2 - 6ac + a^2 - 4bc - 4ab - 4b^2}}{2}.$$

Kind of daunting. Fortunately, the thing inside the square root can be simplified:

$$c^2 - 6ac + a^2 - 4bc - 4ab - 4b^2 = (-a + 2b + 3c)^2 - 8\,(b + c)^2\,.$$

Because of this, from now on we will use these definitions:

$$\begin{aligned}
\Delta &= \sqrt{c^2 - 6ac + a^2 - 4bc - 4ab - 4b^2}; \\
x &= -a + 2b + 3c; \\
y &= b + c; \\
\delta &= c - a.
\end{aligned}$$

In other words, we have $x^2 - 8y^2 = \Delta^2$, or alternatively, $x^2 - \Delta^2 = 8y^2$. This is very interesting, because the left side is the product of two values. We introduce two new variables, p and q, so that $2y^2 = pq$. We use $2y^2$ instead of $8y^2$ because this allows the math to stay comfortably in the realm of integers, not rationals. We now have

$$\begin{aligned}
x &= p + q; \\
\Delta &= p - q.
\end{aligned}$$

This is interesting, but let's go back to the equation of k, which can be rewritten like this:

$$2k = \pm\delta \pm \Delta.$$

Not so bad, but these signs are uncomfortable. We can force both δ and Δ to be nonnegative (in fact, they can't be zero: $\delta = 0$ would imply $a = c$, which is not valid, and $\Delta = 0$ would imply $p = q$, making $2y^2$ a perfect square, which is impossible). This means $p > q$, by the way. We still have four different sign combinations, although we can't have both be negative because that would

imply $k < 0$, which is not covered by the problem description. So we have three different kind of solutions:

- Type A: $2k = \delta + \Delta = p - y$.
- Type B: $2k = \Delta - \delta = y - q$.
- Type C: $2K = \delta - \Delta = q - y$.

Interestingly, we also have, from the definitions of x, y and δ, that

$$\delta = x - 2y = p + q - 2y.$$

So, what this means is that given a number y and the decomposition $2y^2 = pq$, we have an indeterminate system of equations:

$$\begin{cases} c - a = \delta; \\ b + c = y. \end{cases}$$

We now use one of the problem restrictions, $a, b, c \in [-X, X]$, to count the amount of valid values. We have three restrictions, which we will express (without loss of genericity) in terms of valid values for c:

- We can start applying the obvious restriction: $c \in [-X, X]$.
- For a we have $a \in [-X, X]$, but since $a + \delta = c$, this gives us $c \in [-X + \delta, X + \delta]$.
- And for b, using $b = y - c$, we transform $b \in [-X, X]$ into $c \in [-X + y, X + y]$.

The valid set of values is the intersection of these three sets. Now, $\delta > 0$ by definition, which gives us the following set:

$$c \in [\text{máx}\left\{-X + y, -X + \delta\right\}, \text{mín}\left\{X, X + y\right\}].$$

Note that y can be negative, and it will be sometimes, so these must be taken into account. However, we haven't finished yet. Remember that thing about signs, and angles of 135 degrees? Here it is. For types A and C, the only valid values are these where either $b < a$ or $c < b$ (this seems to contradict the problem description. The thing here is that by assuming that the 45 degree angle was in B, we have slightly redefined the terms, and these conditions are valid. We do still need $a < c$ to avoid counting duplicates); for type B, we need the opposite, that is: $a < b < c$. This gives us a valid scheme for counting the values:

- Iterate for $y = 0$ to $2X - 1$ (this is the highest possible value, found when $b = X - 1$ and $c = X$).
- Decompose $2y^2 = pq$ in all its possible decompositions, with $q < p$. Yes, that means about $2e9$ decompositions, and all the divisors of each, although later on most cases yield 0 results. For the special case $y = 0$, we will have decompositions of the form $q = 0$, $p = k$ for $k \in [1, K]$.
- Operate separately for positive and negative values of y (although negative values stop yielding results when you hit $y = -K$). Except for the case $y = 0$. Don't count -0 like it was a valid value!
- For each sign of y, and each decomposition, calculate $\delta = p+q-2y$. Calculate also the minimum and maximum values for c, using the combined set of restrictions as noted above. Move on to the next decomposition if this set is empty.
- Now, for each one of the three types, calculate k using the simple formulas that rely on y, p and q. Return immediately if k is out of range. Otherwise, count only the cases where the range is valid, which is:
  - $\left(c > \dfrac{y+\delta}{2}\right) \vee \left(c < \dfrac{y}{2}\right)$ for types A and C.
  - $\left(c < \dfrac{y+\delta}{2}\right) \vee \left(c > \dfrac{y}{2}\right)$ for type B.
  - These are fastidious because of signs, because of strict inequalities and because of fractions. But they can be translated into separate closed sets. I did need to resort to `LongMath.divide` because the default division truncates to 0 instead of as a floor. It was a bit maddening.
- In the end, you have one (or two) sets of valid values for c. Just count the elements, using the old formula (maximum plus one minus minimum).
- The sum of all the counts is the total count of single triangles.

This is *horribly* inefficient and it took more than one hour even with a lot of threads. Sorry.

Let's go now for the "double" triangles, which are those where two angles have 45 degrees. Which means that, very interestingly, the other one has to have 90 degrees. Which means that two of the sides have some length l and the other one has length $\sqrt{2}l$. Also, no sign bullshit; we can't have 135 degree

angles here; there is no space. We have these equations (again, we assume that the "special" angle, in this case the straight one, lies in B):

$$\begin{aligned} (bk-ak)^2 + \left(b^2-a^2\right) &= (ck-bk)^2 + \left(c^2-b^2\right); \\ 2\left((bk-ak)^2 + \left(b^2-a^2\right)\right) &= (ck-ak)^2 + \left(c^2-a^2\right). \end{aligned}$$

Now, the condition of the straight angle means something additional about the segments that intersect in B: the horizontal length of the first one equals the vertical length of the second, and vice versa. That is:

$$\begin{aligned} |bk-ak| &= \left|c^2-b^2\right|; \\ |ck-bk| &= \left|b^2-a^2\right|. \end{aligned}$$

So...

$$\begin{aligned} k\,|b-a| &= |c-b| \cdot |c+b|; \\ k\,|c-b| &= |b-a| \cdot |b+a|. \end{aligned}$$

Very interesting. Lots of shared terms. If we multiply these equations and simplify (after all $c-b$ and $b-a$ can't be 0), we get

$$k^2 = |c+b| \cdot |b+a|.$$

Very interesting indeed. This means that we can iterate over values of k and decompose k^2 into factors. Let's say $k^2 = pq$. I realised that I could count only coprime cases and then rescale the solutions (this was not so easy in the single case!). So let's say that we have $k^2 = pq$ with p and q coprime. This also means that by construction we must have p and q be perfect squares. Now, remember those equations I included before the one about the straight angles? We can rewrite the first one as

$$(b-a)^2\left(k^2 + (b+a)^2\right) = (c-b)^2\left(k^2 + (c+b)^2\right),$$

which is very convenient now that we have the decomposition of $k^2 = pq$. We have, one again without loss of genericity:

$$(b-a)^2\left(k^2+p^2\right) = (c-b)^2\left(k^2+q^2\right).$$

However, since $q = \dfrac{k^2}{p}$, we have

$$(b-a)^2\left(k^2+p^2\right) = (c-b)^2\left(k^2+\frac{k^4}{p^2}\right),$$

which we can rewrite as

$$p^2\left(1+\frac{k^2}{p^2}\right)(b-a)^2 = k^2\left(1+\frac{k^2}{p^2}\right)(c-b)^2,$$

and we can remove that common term which we know that can't be zero:

$$p^2(b-a)^2 = k^2(c-b)^2.$$

Just one more simplification!

$$p|b-a| = k|c-b|.$$

Very neat, actually. We are going to need that either p or q (one, not both) is negative (the straight angle must always be in the centre, and either a or c must be negative. Think about how the triangle is positioned relative to the parabola, and you will see that either $a+b$ is negative, or $b+$ is), and without loss of genericity (how many times I've said that, just in this problem?), let's say that p is negative. Given k and p, we have two separate system of equations, depending on whether $b-a$ has the same sign as $c-b$ or not. These systems are linear and the solutions have fixed expressions. The first one is

$$\begin{cases} a+b = -p, \\ b+c = \dfrac{k^2}{p}, \\ p(b-a) = k(c-b), \end{cases}$$

with solution

$$\begin{aligned} a &= -\frac{p^3+k^3+2p^2k}{2p(k+p)}; \\ b &= \frac{k^3-p^3}{2p(k+p)}; \\ c &= \frac{p^3+k^3+2pk^2}{2p(k+p)}. \end{aligned}$$

The second system is very similar:

$$\begin{cases} a+b=-p, \\ b+c=\dfrac{k^2}{p}, \\ p\,(b-a)=k\,(b-c)\,; \end{cases}$$

note the sign change at the very end. The solution is

$$\begin{aligned} a &= \frac{p^3-k^3-2p^2k}{2p\,(k-p)}; \\ b &= \frac{k^3+p^3}{2p\,(k-p)}; \\ c &= \frac{k^3-p^3-2pk^2}{2p\,(k-p)}. \end{aligned}$$

And this gives us a possible iteration scheme:

- Iterate for values of k.
- Decompose $k^2=pq$ so that p and q are coprime. In other words, iterate over subsets of primes in k^2 (if there are n primes, there will be 2^n subcases. This is actually much less, and MUCH faster, than the "single" triangles).
- Solve both systems for each combination of k and p, except for the special case $k=1$, $p=1$, which doesn't have any solution. This gives us *rational* values for a, b and c. Multiply them by the least common multiple of the denominators (which will be a divisor of the common numerator, but beware: it's not always the same for the three numbers. You need to use irreducible fractions! I used a `Rational` type). Take note of this multiplier m. This means that the first k where this triangle is actually integer doesn't happen for the k we have decomposed, but rather, for mk.
- Let a, b and c be the rescaled, integer values.
- The amount of possible values including rescaling is this:

$$\text{mín}\left\{\left\lfloor\frac{K}{mk}\right\rfloor,\left\lfloor\frac{X}{|a|}\right\rfloor,\left\lfloor\frac{X}{|b|}\right\rfloor,\left\lfloor\frac{X}{|c|}\right\rfloor\right\}.$$

- Increase the total amount of double triangles in such minimum value, which might be 0. I had overflows somewhere here, so I had to resort to big integers, unfortunately. Anyway, this is still ridiculously faster than the "single" triangle count.

So, that's it, FINALLY. Count the single triangles, count the double triangles, subtract, endut, hoch hech.

Obviously in the problem thread there are smarter people using actually good parametrisations that get the result in under a minute. I'm still moderately happy with this problem because there was SO MUCH to do. You can see how that 70 % rating is appropriate.

And this is it. I probably don't have much time left and I don't think I'm going to be able to solve 289 in the days I have, so chances are that this will be my last problem. So long, Project Euler, and thanks for all the fun. I will make a nice donation before leaving.

403. Lattice points enclosed by parabola and line

Difficulty rating: 55 %.
Solution: 18224771. *Solved: Mon, 20 Jun 2022, 17:34.*
Math knowledge used: cake numbers, Faulhaber formulas.
Programming techniques used: none.

I liked this problem, because it has an interesting twist. There are many problems where the bulk of the job consists of running brute force cases until you identify an obscure sequence (possibly on OEIS), and then after that the result is jsut a summation or a single value of that sequence. Here, it's the other way around: the sequence is moderately easy to find, but then you need to be mindful about getting the correct result, with a careful algorithmic approach to get the exact sum.

We can start with the basics: when does the domain $D(a, b)$ actually exist, and when does it have a rational area? This is very basic stuff, a second degree equation:

$$x^2 = ax + b \Rightarrow x^2 - ax - b = 0 \Rightarrow x = \frac{a \pm \sqrt{a^2 + 4b}}{2}.$$

Although it's not relevant to the problem, we can note that the solutions of the equation are integers: $a^2 + 4b$ must have the same parity as a, and if it's a perfect square, so does its root. Therefore the numerator is the sum or difference of two numbers with the same parity, so it's even, and when we

divide it by 2 we get an integer value for x. And, given the formulas, y must be an integer as well.

This simple solution allows us to brute force some small values, where we can notice an interesting property: the values of $L(a,b)$ are not all over the place. Instead, they seem to come from a discrete set of values. For example, for $N = 10$ there are 76 valid values, but only 12 different values, from $L(a,b) = 1$, which is found in several cases like $L(0,0)$ or $L(6,-9)$, to $L(a,b) = 232$, found only for $L(-9,10)$ and $L(9,10)$. There also seems to be a symmetry regarding a, but not to b, that is: $L(a,b) = L(-a,b)$; but $L(a,b) \neq L(a,-b)$ except in the trivial case $b = 0$. In fact, it's not even very common that $L(a,b)$ and $L(a,-b)$ are both present (this is expected from the quadratic equation reasoning above), but we can see some cases like $L(-5,-6) = 2$ and $L(-5,6) = 64$ (actually, there are not "some cases". This is the only one for $N = 10$, save for its symmetric one for $a = 5$).

It's clear that $L(a,b)$ doesn't depend on a or b alone, but the discreteness of the values suggests that it might directly depend on another variable. OK, let's try then with the obvious one: the square root of the determinant, $k = \sqrt{a^2 + 4b}$, which after all determines the range of valid values for x and therefore could be related with the total amount of lattice points. Jackpot! There seems to be a fixed relationship between k and $L(a,b)$:

$$\begin{aligned} k = 0 &\Rightarrow L(a,b) = 1; \\ k = 1 &\Rightarrow L(a,b) = 2; \\ k = 2 &\Rightarrow L(a,b) = 4; \\ k = 3 &\Rightarrow L(a,b) = 8; \\ k = 4 &\Rightarrow L(a,b) = 15\ldots \end{aligned}$$

The first terms suggest an exponential formula, but after looking for a longer subset of the frequence on OEIS, it seems to be the sequence A000125, "cake numbers": *maximal number of pieces resulting from n planar cuts through a cube (or cake)*, whose formula is not exponential but cubic. Because, yes, there is a very simple formula:

$$c(k) = \binom{k+1}{3} + k + 1 = \frac{k^3 + 5k + 6}{6}.$$

Now, this is amazing, but it doesn't help us much, because the distribution of k is not so clear. There are some patterns, but ultimately we would need to iterate over either the possible values for a, b or k, and in every case the valid range is about the size of N. Not good when $N = 10^{12}$. And this is

where the problem starts requiring an algorithmic approach. On the three basic steps of a project euler problem (formulation, analysis and coding), we have been given a formulation, but what we have done for now is actually a small part of the full analysis (in fact, you could even say that it's just a reformulation, and the actual analysis is about to begin).

The obvious approach would be to count the amount of times each k appears in the total sum, which we will call $f(k)$, so we can do something like this:

$$S(n) = \sum_k f(k)\, c(k).$$

This sounds simple enough, if we can find a simple expression for $f(k)$, but we haven't even found the valid range for k. Let's try and analyse the distribution of values. If we look at each value of a, we can easily confirm that k always has the same parity than a, as mentioned a few paragraphs above. It also seems to take every possible value with the valid parity inside a range; for example, for $a = \pm 6$, the valid values are from $k = 0$ to $k = 8$ (five possible total values). It's obvious that the valid values of k for a given a come from this condition:

$$a^2 - 4N \leq k^2 \leq a^2 + 4N \Rightarrow \left\lfloor \sqrt{a^2 - 4N} \right\rfloor \leq k \leq \left\lfloor \sqrt{a^2 + 4N} \right\rfloor.$$

The problem is that these are not particularly easy to pinpoint, because ranges related to square roots and the floor operator are always going to be problematic. At this point, since we are going to find the distribution for k, it makes sense to change the formulas so that they are centred on k instead of on a: given a fixed a, we have the set of valid k values, but it would be nice to have the set of valid values for a given a fixed k. The distribution happens to look similar, because $k^2 - a^2 = b$ and the limits for b are symmetric (i.e., $[-N, N]$):

$$k^2 - 4N \leq a^2 \leq k^2 + 4N \Rightarrow \left\lfloor \sqrt{k^2 - 4N} \right\rfloor \leq a \leq \left\lfloor \sqrt{k^2 + 4N} \right\rfloor.$$

However, there is a "hidden" difference of sorts: the range for a is $[-N, N]$, but the range for k is $\mathbb{N}$ in principle. The sign is the major difference, but we are going to also an additional difference related to the range. The maximum value of k comes from $\sqrt{N^2 + 4N}$ and, since this value is right below $N + 2 = \sqrt{N^2 + 4N + 4}$, for the vast majority of N we will have a maximum value $k_{\text{máx}} = N + 1$. Good, but not very useful so far: at this point we can only get an $O(k_{\text{máx}}) = O(N)$ algorithm at best, which is not good. We need to continue analysing.

One obvious thing to notice is that we are only interested in the absolute value of a. As mentioned above, the results for a are exactly the same as for $-a$, and basically every single formula we've seen so far seems to confirm it. Great. So, let's consider only the positive range for a, and duplicate the results (save for the special case $a = 0$). Ok, and with that said, let's consider something interesting: the actual bounds for a can be improved. Let's consider two new functions: $a_0(k)$, the minimum absolute value of a that is valid for a given k, and $a_f(k)$, the maximum. Let's not even consider parity for now. It can be seen that the bounds for a are not actually the square roots displayed above; instead, we have other bounds, as given by the problem description:

$$\begin{aligned} a_0(k) &= \text{máx}\left(0, \left\lfloor \sqrt{k^2 - 4N} \right\rfloor\right); \\ a_f(k) &= \text{mín}\left(N, \left\lfloor \sqrt{k^2 + 4N} \right\rfloor\right). \end{aligned}$$

Interestingly, this means that there are discrete points at which the range $[a_0(k), a_f(k)]$ changes its behaviour. In particular, we can separate the valid space of values for k into three separate ranges:

- If $k \leq \sqrt{4N}$, then $a_0(k) = 0$. This is the "initial" range. This range includes all the special cases where $a = 0$ is involved.
- And if $k \geq \sqrt{N^2 - 4N}$, then $a_f(k) = N$. This is the "final" range.
- All the other terms, which are the vast majority, have the normal "square root" bounds both in $a_0(k)$ and $a_f(k)$. This is the "middle" range.

The "initial range" has size $O\left(\sqrt{N}\right)$ and can be iterated over in a reasonable amount of time. The "final" range, for a big enough N, has exactly 4 terms: $N - 2$, $N - 1$, N and $N + 1$. The problematic case comes from the "middle" range. It can be seen that the width of the $[a_0(k), a_f(k)]$ range rises slowly along the "initial" range (interestingly, it rises separately for the even and odd parts, so it might look alternating), then starts descending (first kind of abruptly, then very slowly; and parity doesn't seem to be a factor) over the middle range, and at the "final" range it has like 4 elements or fewer. For each value of k in the "initial" and "final" range, we can just calculate the amount of valid values: calculate $a_0(k)$ and $a_f(k)$, and let m be the amount of values in the range $[a_0(k), a_f(k)]$ with the same parity as k. The total amount of actual values is either $2m - 1$ if $a_0(k) = 0$ and k is even, or $2m$ in every other case. So in these cases we add $m \cdot c(k)$ to the

result. Now let's go for the most complicated part of the problem, which is the "middle" range.

We have this "middle" range of values for k, $\left[1 + \left\lfloor \sqrt{4N} \right\rfloor, \lfloor N^2 - 4N \rfloor \right]$. If we analyse the evolution of the ranges $[a_0(k), a_f(k)]$ for these k, we can see that the "alternating" pattern we saw in the "initial" range doesn't exist any more: the sequence is decreasing (non-strictly), and parity doesn't seem to have a role. Honestly, this stumped me, until I realised the exact trick we need. It's very clear that $a_0(k) < k < a_f(k)$, so let's define some additional functions, $\Delta_0(k) = k - a_0(k)$ and $\Delta_f(k) = a_f(k) - k$. Now, certain patterns are much clearer:

- Both of these functions are (non-strictly) decreasing, wich concurs with the observation of the ranges getting narrower.
- The amount of valid values depends on the parity of these functions: indeed, the range is reduced in one element when an increase in k causes a decrease in either $\Delta_0(k)$ or $\Delta_f(k)$, such that it goes from an even value into an odd value.
- In particular, the amount of valid values for a for a given k, including the duplication because of the sign, is exactly

$$2\left(\left\lfloor \frac{\Delta_0(k)}{2} \right\rfloor + \left\lfloor \frac{\Delta_f(k)}{2} \right\rfloor + 1\right),$$

  with that 1 corresponding to the case $a = k$. Remember that we are on the "middle" range, where we can be sure that a is neither 0 nor above N.
- This explains exactly why there aren't variations related to the parity of k: the parity of the ranges moves with that of k. So, for example, if we have $N = 10^4$ (which is what I used for my experiments), $k = 3393$ has a valid range of $[3388, 3398]$ for a, while $k = 3394$ has $[3389, 3399]$ and $k = 3395$ has $[3390, 3400]$.

This suggests a new approach: grouping values of k given their ranges of $[\Delta_0(k), \Delta_f(k)]$. The maximum value of $\Delta_0(k)$ (which is always greater or equal to $\Delta_f(k)$) is around (below, in fact) $2\sqrt{N}$, and the maximum value of $\Delta_f(k)$ is also below $\sqrt{N}$, so there are around $3\sqrt{N}$ places where there is a decrement, therefore the amount of ranges is on the order of $O\left(\sqrt{N}\right)$, which is finally good enough. Here is how we will, finally, proceed to calculate the sum of valid values in this problematic "middle" range:

- Start with $k_0 = \lfloor 4N \rfloor + 1$, the minimum value in the range.
- Calculate $x_0 = \Delta_0(k_0)$ and $x_f = \Delta_f(k_0)$.
- Now we are going to define yet two additional functions, indicating the values where there is a change in the delta functions. The change points for $\Delta_0(k)$ come from the function

$$l(x) = \left\lfloor \frac{4N + x^2}{2x} \right\rfloor,$$

and the change points for $\Delta_f(k)$ come from the function

$$u(x) = \left\lfloor \frac{4N - x^2}{2x} \right\rfloor.$$

These functions are such that, if $l(x) = k$, then $\Delta_0(k) \geq x$ and $\Delta_0(k+1) < x$, and the same goes for $u(x)$ and $\Delta_f(x)$. For convenience, we can precompute the values of these functions up to $l(x_0)$ and $u(x_f)$.

- Now we can start the actual iterations. The variables k_0, x_0 and x_f will persist between iterations.
  - Let $k_f = \min\{l(x_0), u(x_f)\}$. We must also make sure that $k_f \leq \lfloor \sqrt{N^2 - 4N} \rfloor$, although this is only relevant at the last iteration.
  - We know that, for every k in the range $[k_0, k_f]$, we have $\Delta_0(k) = x_0$ and $\Delta_f(k) = x_f$. Let m be the amount of valid values of a in this range, using the formula I included above. Now, calculate the sum of $c(k)$ for all the values in the given range:

$$S_{k_0,k_f} = m \sum_{k=k_0}^{k_f} c(k) = m\left(s(k_f) - s(k_0 - 1)\right).$$

The partial sums $s(k)$ have been defined like this (where $T(x)$ represents the triangular number $\frac{x(x+1)}{2}$), using a Faulhaber formula:

$$s(k) = \sum_{n=1}^{k} c(n) = \frac{T(n)^2 + 5T(n) + 6}{6}.$$

  - We can add S_{k_0,k_f} to the result (where we have accumulated the "initial" and "final" ranges).

- If $l(x_0) \leq k_f$, decrease x_0 until the condition no longer holds.
- If $u(x_f) \leq k_f$, decrease x_f until the condition no longer holds.
- if $k_f = \left\lfloor \sqrt{N^2 - 4N} \right\rfloor$, finish iterating. Otherwise, do $k_0 = 1 + k_f$ and move on to the next iteration.

Note that there is a lot of potential for overflows all over the place, since we need to square numbers of the order of 10^{12}. Aside from that, the result of the sum of the three ranges is the result of the problem, which for $N = 10^{12}$ is:

$$263720538952078758083690477915007981298118224771$$

So, the order of the solution is about N^4. This whole result, using BigInteger, takes around 0,8 seconds to run, so I'm guessing that with mods it would probably take less than 50ms. In the problem thread all the approaches seem to be very similar, although there are some small optimisations here and there.

404. Crisscross Ellipses

Difficulty rating: 60 %.
Solution: 1199215615081353. *Solved: Thu, 23 Jun 2022, 02:44.*
Math knowledge used: geometry, Pythagorean triples, sum of squares theorem.
Programming techniques used: none.

This is a considerably difficult problem with several steps to take. The hardest one is possibly the initial one, i.e. finding an integer formula for b and c. The trick is not to use the usual formula, $\frac{x^2}{4} + y^2 = 1$, nor the obvious parametrisation, $x = 2\cos\theta$, $y = \sin\theta$, but the polar form:

$$r(\theta) = \frac{a}{\sqrt{\cos^2\theta + 4\sin\theta}} = \frac{a}{\sqrt{1 + 3\sin\theta}}.$$

This form allows introducing rotations naturally by modifying θ. Anyway, since the lengths b and c are perpendicular, one can define θ_b as the angle of rotation that moves the vertical axis into the angle of b (and similarly with c), and then use perpendicularity to note that $\theta_b = \theta_c + \frac{\pi}{2}$. Therefore

$$b = \frac{2a}{\sqrt{1 + 3\sin^2\theta}},$$

$$c = \frac{2a}{\sqrt{1+3\cos^2\theta}}.$$

We can clear the trigonometric terms like this:

$$3\sin^2\theta = \frac{4a^2-b^2}{b^2},$$
$$3\cos^2\theta = \frac{4a^2-c^2}{c^2},$$

and we can remove the angle by summing these two expressions, giving us

$$\frac{4a^2-b^2}{b^2} + \frac{4a^2-c^2}{c^2} = 3 \Rightarrow 4a^2c^2 - b^2c^2 + 4a^2b^2 - b^2c^2 = 3b^2c^2.$$

We then have $4a^2b^2+4a^2c^2 = 5b^2c^2$, which can also be expressed as $\frac{4}{b^2}+\frac{4}{c^2} = \frac{5}{a^2}$ if you like it better. This finishes the initial formulation step, which is not so long but it's still tricky to get right.

Now we can start some brute force process for the smaller sample cases in order to see how do the solutions look like. Some experiments show that all the *primitive* solutions (that is, those that are not a multiple of another, smaller one) can be expressed in terms of three smaller terms, x, y and z, so that:

- The three numbers are coprime, and odd. There doesn't seem to be any particular restriction aside from that, although the smaller cases are always prime; not every prime appears, but I didn't find any glaring property separating the ones that appears from the ones that don't.
- $a = xy$, $b = xz$ and $c = 2yz$. Sometimes $b > c$ and sometimes is the other way around, but with this formula we catch everything regardless of the order.
- $b, c < 2a$ and $a < N$ give the conditions under which the solution is valid. The first one indicates whether the solution is valid, while the second one is just the problem bounds.

The most useful data points about these finds are the formulas for a, b and c. Plugging them into the formula we know, and doing some simplification, the end result is a simple formula:

$$x^2 + 4y^2 = 5z^2.$$

This is very simple, and we could even try to run it through Alpertron somehow, but what I did was moving it into a Pythagorean triple. So, I used an all-odd triple because it's what I observed, but it's more natural to just treat y as an even number (it's never going to be a multiple of 4, but that's not a problem. In fact we don't even care much about it being even, we just want a simplified formula). So $a = \frac{xy}{2}$, which we know to be always an (odd) integer, $c = yz$ and $x^2 + y^2 = 5z^2$. This is very convenient, because this formula looks even more like a Pythagorean triple. In fact, we can extract a Pythagorean triple from it! Let $p^2 + q^2 = r^2$ be a Pythagorean triple. We also know that $2^2 + 1^2 = 5$ is the only way to represent 5 as the sum of two squares (ignoring sign changes), so the sum of squares theorem tells us that:

- $(2p - q)^2 + (p + 2q)^2 = 5r^2$.
- $(2p + q)^2 + (p - 2q)^2 = 5r^2$.
- There is no additional way of expressing $5r^2$ as the sum of two squares (barring alternate forms of p and q, which in any case would be studied separately).

This hints at a way of generating solutions for this problem: generate Pythagorean triples, transform them, check whether the solution is valid, scale these primitive solutions as needed. The standard Pythagorean triple generation suggests to rely on two even simpler variables, m and n, using the standard iteration (iterate over values of m, then iterate over values of $n < m$ that are coprime with m and have different parity). Then we have:

- $p = m^2 - n^2$; $q = 2mn$; $r = m^2 + n^2$. Note that q is always a multiple of 4, while p and r are odd.
- Either $x = 2p - q$ and $y = p + 2q$ or $x = 2p + q$ and $y = p - 2q$; $z = r$ in any case. We don't care much about the distinction between x and y, but it can be seen that one of them ($p \pm 2q$) is always odd, while the other ($2p \pm q$) is always congruent with 2 modulo 4; as expected.
- Finally, $a = \frac{xy}{2}$, $b = xz$, $c = yz$.

This is workable, since it gives formulas for a as a function of m and m, but it's still kind of rough around the edges. It can be seen that $a \approx O(m^4)$ and the iterations are of size $O(m^2)$, so in the end we have $O\left(\sqrt{N}\right)$ calculations; this is good enough. We still need to take care of these conditions:

- We need to verify that $b, c < 2a$.

- In principle, $2p - q$ and $p - 2q$ might be negative. We need to discuss what happens in these cases.
- We also need to check whether x and y are coprime. It turns out that they aren't always so.
- Finally we will need to apply the condition $a < N$, i.e. the problem bounds.

The first conditions are very interesting: $c < 2a \Rightarrow yz < xy$, and $b < 2a \Rightarrow xz < xy$. Since all these numbers are positive, we can just say that $z < y$ and $z < x$. This is not super useful on its own, but now let's look at the subtractions. It can be easily shown that sometimes they do indeed result in negative numbers; for example, for the well known triple $(7, 24, 25)$ we have $2p - q = -10$ and $p - 2q = -41$. An obvious solution would be to use the absolute values; however, we know that these numbers must always be bigger than $z = r$. Let's look first at $2p - q$: we need to check whether $q - 2p$ can be bigger than r:

$$q - 2p > r \Rightarrow q > 2p + r \Rightarrow q^2 > 4p^2 + 4pr + r^2 \Rightarrow q^2 > q^2 + 5p^2 + 4pr.$$

Obviously this can't happen. Which is a bit of a problem because the equation is symmetric for p and q, meaning that $p - 2q > r$ does also never happen. Well, no problem: we will take $2q - p$ instead. Additionally we can check that the "sum" term is always going to be valid:

$$2p + q > r \Leftarrow p + q > r \Leftrightarrow p^2 + 2pq + q^2 > r^2 \Leftrightarrow r^2 + 2pq > r^2.$$

We now need to consider whether $2p - q > r$ and $2q - p > r$ hold. They not always do. We can check them separately using the expressions in terms of m and n. For the first case, we have:

$$2p - q > r \Leftrightarrow 2\left(m^2 - n^2\right) - 2mn > m^2 + n^2 \Leftrightarrow m^2 - 2mn - 3n^2 > 0.$$

We can consider this as an inequation, and check the valid ranges for m given some n. The solutions of the quadratic equation are

$$m = \frac{2n \pm \sqrt{4n^2 + 12n}}{2} = \frac{2n \pm 4n}{2} = -n \text{ or } 3n.$$

In particular, since both m and n must be positive, this inequation holds only when $m > 3n$.

We can now look at $2q - p > r$, again using the expressions in terms of m and n:

$$2 \cdot 2mn - \left(m^2 - n^2\right) > m^2 + n^2 \Leftrightarrow 4mn - m^2 + n^2 > m^2 + n^2 \Leftrightarrow 4mn > 2m^2.$$

Again, since m and n are positive, we can just say that the condition is valid when $m < 2n$. Interestingly, this set is disjoint with the previous one, which suggests iterating separately.

Let's now check coprimality. Looking at the formulas, it may looks like it's a given, but actually we sometimes get an additional factor. If we look at the gcds, we can see that $\gcd(2p - q, p + 2q) = \gcd(-5q, p + 2q)$. Now, $\gcd(q, p + 2q) = 1$ by construction, but this factor of 5 does appear. The same happens for the other pair. In fact, when one of these numbers is a multiple of 5, so is always the other. This can be exhaustively checked, with all the cases of m and n modulo 5. I'm assuming that there is a deep number theory reason behind that. Anyway, the good thing here is that we only need to check whether $2p - q$ (in the first case) or $2q - p$ (in the other) is a multiple of 5. If they are, then the solution is not coprime and we can discard it. If they aren't, this is a valid primitive solution.

Now, for the last condition, $a < N$. We can study the two cases separately:

- First case, $x = 2p - q$ and $y = p + 2q$. We have $xy = 2p^2 + 3pq - q^2$, and if we expand these in terms of m and n, we get

  $$xy = 2m^4 + 6m^3n - 12m^2n^2 - 6mn^3 + 2n^4.$$

  This means that

  $$a(m, n) = m^4 + 3m^3n - 6m^2n^2 - 3mn^3 + n^4.$$

  Now, for this case we need to be also sure that $m > 3n$, so the valid range of n is something like $\left[1, \frac{m}{3}\right]$. It can be seen that, for a fixed m, a as function of n may have a maximum near $\frac{m}{3}$, but just in case, let's use the fact that there is no local minimum, and therefore the global minimum is at one of the extremes. We have

  $$\begin{aligned} a(m, 1) &= m^4 + 3m^2 - 6m^2 - 3m + 1, \\ a\left(m, \frac{m}{3}\right) &= \frac{100}{81}m^4. \end{aligned}$$

  The most conservative action would be to take $a \approx m^4$, so that the maximum m we need to consider is $m_{\text{máx}} = \sqrt[4]{N}$.

- Second case: $x = 2p+q$, $y = 2q-p$. Now we have $xy = -2p^2+3pq+q^2$, which means that, using a similar expansion as before, we get

$$a\,(m,n) = -m^4 + 3m^3n + 6m^2n^2 - 3mn^3 - n^4.$$

The condition we need now is $m < 2n$, so that the range of n is approximately $\left[\frac{m}{2}, m\right]$. This function is mostly ascending in the given range, but just in case, we can check both limits:

$$\begin{aligned} a\left(m, \frac{m}{2}\right) &= \frac{25}{16}m^4, \\ a\,(m,m) &= 4m^4. \end{aligned}$$

So the minimum value for a given m is the first one, and the maximum m in this case is $m_{\text{máx}} = \sqrt[4]{\frac{16N}{25}}$.

This is, finally, everything that we need to calculate the solution of the problem. Using a generic Pythagorean triple generator doesn't work very well, so I iterated directly for m and n, with one separate loop for each case. The full algorithm is this:

- Init a result variable as 0.
- Calculate the limits for m, $M_1 = \sqrt[4]{N}$ and $M_2 = \sqrt[4]{\frac{16N}{25}}$.
- Now, iterate for $m = 1$ up to M_1 (which is the maximum of the two limits). For each one of those:
  - First case, to be checked every time (since $m < M_1$ will always happen):
  - For this first case, the limits of n are 1 and $\left\lfloor \frac{m}{3} \right\rfloor$. Iterate over values in this range, checking whether they are coprime with m and with different parity:
    - Calculate $m^2-mn-n^2 = \frac{2p-q}{2}$ (yes, we can remove a factor of 2 from the calculations). If this is a multiple of 5, move on to the next n. Otherwise, keep this n.
    - Calculate $a = m^4 + 3m^3n - 6m^2n^2 - 3mn^3 + n^4$ (feel free to precompute powers of m, which well be used aplenty!). This is a primitive solution, which can be rescaled up to $t = \left\lfloor \frac{N}{a} \right\rfloor$ times. Increase the result variable in this value t.

- Now, let's check the second case. We will only check this if $m < M_2$.
- In this second case, the range of n is $\left[1 + \left\lfloor \frac{m}{2} \right\rfloor, m - 1\right]$. Again, we must ensure coprimality and different parity. Iterate for n in this range:
  - Calculate $n^2 - m^2 + 4mn = q - 2p$. If this is a multiple of 5, go to the next n, but otherwise, this n is valid.
  - Calculate $a = -m^4 + 3m^3n + 6m^2n^2 - 3mn^3 - n^4$. Don't bother trying to reuse powers of n from the previous case, since there isn't any n for which both cases are valid. I guess you could cache powers of generic n, but the calculations are so fast that it's not worth it. Again, this a is a primitive solution, so rescale it, increasing the result variable in an amount $t = \left\lfloor \frac{N}{a} \right\rfloor$.
- Nothing else to do with this m.

- When this loop has finished, the value in the result variable is the solution to the problem.

The code is very simple, and it doesn't require any complicated library functions, nor the usage of any collection. I only had to call Euclid's algorithm to check coprimality. The end code has about 50 lines and runs in 1,5 seconds. Most of the solutions in the problem thread are very similar, although some use a ternary tree with the "magic" Pythagorean transform matrices, which I briefly considered as well (although I ended up going for the simpler approach of iterating through m and n).

409. Nim Extreme

Difficulty rating: 55 %.
Solution: 253223948. *Solved: Fri, 29 Jul 2022, 05:35.*
Math knowledge used: game theory, combinatorics.
Programming techniques used: dynamic programming.

Actually, the amount of game theory needed for this is almost zero. This is most eminently a combinatorics problem. The only amount of "game theory" needed for this is the (relatively well known, at least for Project Euler veterans) fact that Nim losing positions are exactly those where the XOR of all the piles equals 0. And so, in the typical three step fashion (formulation, analysis and coding), the formulation of this problem is this: *Given some*

N, find how many tuples of exactly N elements, all between 1 and $2^N - 1$ (both included), and without repeated elements, verify that the XOR of all its members is not 0. There you have. Everything else is a matter of combinatorics. Of course, the hard part is the analysis. It's in fact not that hard (55 %? Yeah, that seems more or less correct. Middle of the road, definitely not easy but not horribly difficult), but I lost almost a whole week because I kept chasing unfruitful paths, until I saw a relatively obvious one that finally worked.

It seems pretty obvious to compute the amount of piles whose XOR *is* zero, and then subtract this amount from the total amount of valid piles. There are many ways of approach this, so here are my approaches that didn't work:

- Inclusion-exclusion. You need to exclude pairs where two piles are equal, which means hell when you realise that you also need to count cases where four piles are equal, six piles are equal, and so on. The amount of cases to consider is, I believe, of the order of $\sum_{i=0}^{N/2} P(i)$, where P is the partition function, so this might be workable for $N = 100$, but forget about $N = 10^7$.

- Calculating $W(x)$ in terms of $W(x-1)$ (and as many $W(x-i)$ as needed), using a recursion, dynamic programming or whatever. Now, here is where I lost most of the time, because, since the amount of piles is related to the maximum amount of elements in a pile, maybe there is some relationship to find? So I kept trying to find schemes of "adding a new bit" to existing solutions, which quickly grow in amount of cases and become unmanageable.

The approach that worked for me is this: ignore the previous $W(x)$ values, and use piles of up to $2^N - 1$ elements from the beginning. Then, find a formula that relate the amount of winning and losing positions for n piles with the amounts for lesser n. Of course I had found a very similar formula by working with the $W(x)$ function, but now that I didn't need to "add bits", this was far easier to pin. Finally, here goes a full description of the algorithm.

First, given a maximum value e, which in our problem happens to be $2^n - 1$ (but which could have been any other number greater than n, although the relationship with powers of 2 is very convenient), and the amount of piles n, we define these three functions:

- $f(e, n)$ is the total amount of valid sets of piles.

- $w(e,n)$ is the amount of winning positions. Additionally we will consider a set $W_{e,n}$ containing all such piles, to aid our reasoning.
- Conversely, $l(e,n)$ is the amount of losing positions, and, why not, let's also define the set $L_{e,n}$ of such elements.

By definition, $w(e,n) = \#W_{e,n}$ and $l(e,n) = \#L_{e,n}$. It's also clear that $w(e,n) + l(e,n) = f(e,n)$. We can also see that $f(e,n)$ is just the amount of variations $e^{(n)} = e(e-1)(e-2)\ldots(e+1-n)$. Now let's try to build the set $L_{e,n}$ from the previous cases, which is the not so easy part. Given a winning solution from $W_{e,n-1}$, we know that the xor of all its piles is a number below e (we can only be sure of this because $e+1$ is a power of 2!); it's also nonzero, because otherwise it would be a losing position, so this means that we can be sure that the new pile has a value in the valid range. Unfortunately this isn't good enough, because what if it's a repeated value? This means that one of the other $n-1$ values is already a known one. Which means that if we remove both values (the new one and the repeated one), thanks to the properties of XOR, we get another losing position of size $n-2$. Very interesting! Let's count how many of these cases are there. Given an element from $L_{e,n-2}$, we can add new piles but we only have $e+2-n$ valid values remaining. One of the new piles goes at the very end, but for the other can be at any of the $n-1$ positions (at the start, at the end just before the "last" one we are also adding now, or in the middle of the existing $n-2$ piles). Which means that there are $(e+2-n)(n-1)l(e,n-2)$ such cases. And so, we arrive at the final formula we want:

$$l(e,n) = w(e,n-1) - (e+2-n)(n-1)l(e,n-2).$$

This is enough to build a very easy recursion. Start with $l(e,1) = l(e,2) = 0$ and $w(e,2) = f(e,2) = e(e-1)$. Although we also have $w(e,1) = f(e,1) = e$, we don't need it. Now, at each step, we can do:

$$\begin{aligned} f(e,n) &= (e+1-n)f(e,n-1); \\ l(e,n) &= w(e,n-1) - (e+2-n)(n-1)l(e,n-2); \\ w(e,n) &= f(e,n) - l(e,n). \end{aligned}$$

You can use arrays, but I just stored the variables I needed (at each instant, the latest w, the latest f and the two latest l). Modding is obviously needed at each step, since the final result is expected to have not much less than 10^{14} binary digits. The code is about 20 lines long and it runs in less than 80 milliseconds. Most people in the problem thread are doing this in almost

exactly this same way, unsurprisingly. Someone found a closed formula, which still needs $O(n)$ to be computed.

414. Kaprekar constant

Difficulty rating: 60 %.
Solution: 552506775824935461. *Solved: Sun, 13 Jun 2021, 11:36.*
Math knowledge used: combinatorics.
Programming techniques used: memoization.

Yes, the math knowledge used is combinatorics, and of such basic nature that I could have basically said "none" as the amount of math knowledge required. This is contrary to what I expected here: there is no magic algorithm that speeds up the computation; the trick is to find a way to group numbers in as condensed a way as possible. I find it very remarkable that this problem has the same level of difficulty than 380, which is vastly more difficult in my opinion.

The first times I approached this problem, I thought of a basic approach where each group could be defined as the sorted set $\{a_1, a_2, a_3, a_4\}$, indicating the differences between each number and the next one after sorting. This has the restriction that $\sum a_i <= b$, but it's still $O(b^4)$, not workable for cases as high as $b = 1803$. Then, after someone pinged me about the Kaprekar constant on twitter (thanks, THMP. Thanks, Bartual), I decided that I should give a try to this problem. After digging for a bit on the literature, some article mentioned a representation that was good enough for me. Let $\{A, B, C, D, E\}$ be the digits of a number. Instead of focusing on $\{B - A, C - B, D - C, E - D\}$ as I had initially planned, one could represent these groups by a certain symmetric difference using just two numbers: $\{x = D - B, y = E - A\}$. This reduces the problem to $O(b^2)$ and the sum of all the elements to $O(b^3)$, making it feasible for $b = 1803$.

Now, before continuing with the algorithm, let's analyse the Kaprekar constant itself. Given a base $b = 6k + 3$, with $k \geq 2$, there is indeed a paper showing that the Kaprekar constant exists, meaning that every five digit number in base b whose digits are not all the same will eventually converge to the same constant, and not enter any cycles. This is not trivial, and not every base, nor every amount of digits has this property; not even close. In this same paper we can see that the Kaprekar constant is $\left\{\frac{2b}{3}, \frac{b}{3} - 1, b - 1, \frac{2b}{3} - 1, \frac{b}{3}\right\}$. Sorting these digits we get $\left\{\frac{b}{3} - 1, \frac{b}{3}, \frac{2b}{3} - 1, \frac{2b}{3}, b - 1\right\}$, so for these constants we get $x = \frac{2b}{3} - \frac{b}{3} = \frac{b}{3}$ and $y = b - 1 - \left(\frac{b}{3} - 1\right) = \frac{2b}{3}$. Nice and kind of patterned.

To proceed with the final algorithm, we need three things: one, a way to evolve a pair (x, y) into the next one; two, a way to efficiently calculate and possibly store the amount of steps for each pair; and finally, a way to count how many elements are represented by the pair. The first step is not complicated, the second one is almost trivial, and the last one, which includes combinatorics, is the only one that requires additional care, although it's mostly because there are many cases to cover.

Let's start with the evolution. Suppose a number is represented as (x, y). That is, $D = B + x$ and $E = A + y$. This means that their sorted digits are $(A, B, C, B + x, A + y)$. For a step in the Kaprekar routine, we will subtract the digits of $(A, B, C, B + x, A + y)$ from $(A + y, B + x, C, B, A)$, and the result depends on whether $x = 0$ or not. The *unsorted* differences, i.e. the ones calculated directly, depend on whether $x = 0$ or not:

$$\begin{aligned}(y - 1, b - 1, b - 1, b - 1, b - y) &\quad \text{if} \quad x = 0,\\ (y, x - 1, b - 1, b - x - 1, b - y) &\quad \text{if} \quad x > 0.\end{aligned}$$

Now, we can start doing some case-by-case analysis, but why bother? We are using a computer. Create an array with these values, sort it, and from the result, $\{a_i : i \in [0, 4]\}$, the new values are $x' = a_3 - a_1$ and $y' = a_4 - a_0$. This might be slightly slower than the case-by-case analysis, but it's still constant time since the array size is fixed, and there is much less room for mistakes.

Now let's analyse how to store these values properly. Let's see. Which values can x and y have? Clearly y can be as high as $b - 1$ (think of $(0, 0, 0, 0, b - 1)$). Also, it must be the case that $x \leq y$; otherwise we would have $D = B + x > A + y = E$, which can't happen. Can it be the case that $x = y$? Yes, see for example $(0, 0, 0, n, n)$ for any $n < b$. Can it be that $x = 0$? Sure, why not: $(0, 0, 0, 0, n)$. And $y = 0$? Mmm, no. That means $A = E$, and this implies also $A = B = C = D = E$ by the sandwich theorem, and this is the special set that doesn't count for this problem. This gives us the following constraints of the space: $y \in [1, b - 1]$; $x \in [0, y]$. This suggests using a 2D array for the representation of the space, where y will be the first dimension and x will be the second. The array will not be "square" but triangular, since the size of each row will depend on which row it is. Anyway, the amount of memory is a very reasonable $O(b^2)$. As for how to fill the array, we can use a Collatz-like recursive iteration scheme:

- Initialise the Kaprekar pair, $x = \frac{b}{3}$ and $y = \frac{2b}{3}$, so that the count for this pair is set to 1. The rest of the pairs will be initialised to 0.

- Now, iterate over each pair (x, y). For each pair, we will call a recursive function that will ensure that the counter has been calculated for the given parameters.
- If the count for (x, y) is already in place, return immediately from the recursive function and step into the next (x, y). But if the count is 0, we need to continue.
- Let (x', y') be the next pair after a Kaprekar routine step, calculated as indicated above.
- Recursively call this same function so that the value for (x', y') is calculated.
- Assign to the pair (x, y) the value of the counter for (x', y'), plus 1.
- Having assigned the value of (x, y), the recursive function call ends, and the iterations can continue.

This will result in an infinite loop if the initial Kaprekar pair is not initialised before the main loop, or if the working base and set of digits does not have a Kaprekar constant. Since this is not our case, this algorithm ends and indeed it's very fast.

Now we need to calculate how many numbers belong to each group defined by the pair (x, y). This is not trivial, but it's also not very difficult. There are many cases to study separately. Let's start by considering that there are five digits A, B, C, D, E, and that they are sorted. There are four signs between them, and each one can be either $=$ or $<$. So there are 16 different cases. This is useful to guarantee that we don't miss any, but actually, instead of treating each group using this format, we will group depending on (x, y). We will have four cases, each one with subcases.

- $x = 0$, $y = 0$. This only has what I refer to as "case 0":
  - Case 0: $A = B = C = D = E$. There are exactly b elements, each corresponding to one digit in base b. But it doesn't matter, because this case is not counted in this problem.
  - So this case is happily ignored and we move on into more interesting ones.
- $x = 0$, $y > 0$. There are the following subcases:

- Case 1: $A = B = C = D < E$. There are $b - y$ ways to choose the value of A, and once chosen, every other number is fixed. There are $\binom{5}{4,1} = 5$ ways to sort these digits, so in total we have $5\,(b - y)$ cases.
- Case 2: $A < B = C = D = E$. Similar to the previous one, there are $b - y$ ways to choose A, which determines the rest, and 5 ways to sort the digits, therefore $5\,(b - y)$ cases again.
- Case 3: $A < B = C = D < E$. There are $b - y$ ways to choose A, which determines A, but not B, C or D. Now, we have $A < B < B + y$ so we have exactly $y - 1$ ways of choosing B, and this determines all the other numbers. And there are $\binom{5}{3,1,1} = 20$ ways to sort these digits, resulting in a total of $20\,(b - y)\,(y - 1)$ cases.
- These are all the cases where $x = 0$, and this includes a total of $(b - y)\,(10 + 20\,(y - 1))$ numbers.

▪ $0 < x < y$. This is the widest case, with 9 subcases:

- Case 4: $A = B = C < D < E$. Choosing A in any of the $b - y$ ways possible determines all the digits. We can sort these digits in $\binom{5}{3,1,1} = 20$ cases. The total is $20\,(b - y)$.
- Case 5: $A = B < C < D < E$. As always, $b - y$ cases for A. This determines B, $D = B + x$ and $E = A + y$, but not C. We have $B < C < B + x$ so there will be $x - 1$ possibilities for C. Each collection of digits has $\binom{5}{2,1,1,1} = 60$ arrangements and so the total is $60\,(b - y)\,(x - 1)$.
- Case 6: $A = B < C = D < E$. All numbers are determined by A, so we have $b - y$ collections of digits, each one of which can be arranged in $\binom{5}{2,2,1} = 30$ cases, for a total of $30\,(b - y)$.
- Case 7: $A < B = C < D < E$. Choosing A fixes E, and then we can choose B in such a way that $B > A$ and $D < E$, that is, $B + x < A + y$. So $A < B < A + y - x$, and there will be $y - x - 1$ ways to choose B. This fixes C and D. Sorted in any of the $\binom{5}{2,1,1,1} = 60$ possible ways, this case entails $60\,(b - y)\,(y - x - 1)$ numbers.
- Case 8: $A < B < C = D < E$. We can proceed in the exact same way than in the previous case, only that we make $C = D$ instead of $C = B$ at the end. So, $60\,(b - y)\,(y - x - 1)$ numbers again.
- Case 9: $A < B < C < D = E$. This is the same as case 5, so we will have $60\,(b - y)\,(x - 1)$ cases.

- Case 10: $A < B < C = D = E$. This works in the same way as case 4, so there will be $20\,(b-y)$ numbers again.
- Case 11: $A < B = C < D = E$. Same as case 6. The total amount will therefore be $30\,(b-y)$.
- Case 12: $A < B < C < D < E$. We stasrt by choosing A in any of the possible $b-y$ ways. Then, as we did in case 7, we have $y-x-1$ ways of choosing B. Finally, with $B < C < B+x = D$, there will be $x-1$ ways of choosing C. Finally, since every digit is different, we will have $5! = 120$ orderings, for a grand total of $120\,(b-y)\,(y-x-1)\,(x-1)$ numbers. This is the only case with three degrees of freedom.
- That's everything for the $0 < x < y$ case. The total amount of combinations is

$$(b-y)\,(100 + 120\,((y-x-1) + (x-1) + (y-x-1)\,(x-1)))\,.$$

- $0 < x = y$. This forces $A = B$ and $D = E$, but we can still choose C. This is the last "umbrella" case, with these subcases:
  - Case 13: $A = B = C < D = E$. Every number is fixed once we choose A, so we have $b-y$ collections of digits and $\binom{5}{3,2} = 10$ subcases, for a total of $10\,(b-y)$.
  - Case 14: $A = B < C = D = E$. Same as the previous one, $10\,(b-y)$ cases.
  - Case 15: $A = B < C < D = E$. After choosing any of the $b-y$ possible cases for A, we have room for $y-1$ different values of C. After taking into account the $\binom{5}{2,2,1} = 30$ arrangements, the total is $30\,(b-y)\,(y-1)$.
  - And that's it. Total: $(b-y)\,(20 + 30\,(y-1))$ cases.

Phew. We have found 16 different cases (15 plus the special "zero" case), so we know that the analysis is complete. Now, the good news is that the analysis is laborious, but the formulas are very reasonably simple. So, after having calculated all the Kaprekar counters for each pair of numbers, we can just iterate again for all the pairs, and for each one, add the value of the counter, multiplied times the amount of numbers in this case, to a final "result" variable, which we will have initialised to -1 to indicate that the Kaprekar constant itself belongs to the $\left(\frac{b}{3}, \frac{2b}{3}\right)$, but it's a special case and we will add 0 instead of -1. We can do this for all the bases proposed

in the problem, adding it all (the results for each base fit in a long; the whole results of the problem don't, so either use mod carefully or be like me and resort to BigIntegers). The end result is the very manageable total of 275552506775824935461. Removing the first three digits results in the 18 ones that we need to input as the problem solution.

420. 2x2 positive integer matrix

Difficulty rating: 60 %.

Solution: 145159332. *Solved: Fri, 24 Jun 2022, 04:18.*

Math knowledge used: "first prime" Erathostenes sieve, Chinese remainder theorem, matrix square roots.

Programming techniques used: none.

This was far easier than I suspected, although it still relies on the relatively obscure formula for matrix square roots. In the end I could go as far as to directly generate the results, with no conditional sentence of the kind of "if this condition doesn't hold, this number doesn't work, so keep iterating".

First, the theoretical basis. Let's say that we have a square matrix,

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

Its determinant is $\delta = AD - BC$ and its trace is $\tau = A + D$. First of all, the determinant must be nonnegative for the square root to exist: there must exist a real number s such that $s^2 = \delta$. This includes both positive and negative square roots. Now, for each possible s value, we are also going to require that $\tau + 2s$ is positive (0 was acceptable for s, but now it isn't), so that there exists a real number t such that $t^2 = \tau + 2s$. In principle we also consider positive and negative values, although for our algorithm we will see that only positive numbers are valid. Given these choices for s and t, the square root of M is this matrix:

$$R = \frac{1}{t} \begin{pmatrix} A + s & B \\ C & D + s \end{pmatrix}.$$

This is reasonably simple, and we can start working with this immediately. First of all, we are interested in matrices M such that every element of M and R is positive. Right off the bat, this means that t can't be negative,

because in that case the signs of B and $\frac{B}{t}$ are different, so not every member of M and R is strictly positive. This means that, out of the four square roots, we consider only two: s positive and t positive on one hand, s negative and t positive on the other. This also excludes the case $s = 0$ because there would only be one square root.

This, alone, adds a big restriction on the values of the coefficients. For starters, the determinant δ must be a perfect square. However, we won't add this as a condition to the coefficients; instead, we will just work with integer values of s: in all the full algorithm there isn't any square root calculation at all, in fact.

Ok, so we have this s such that $AD - BC = s^2$. Now, what about the t? This is where things start getting really interesting, in terms of finer restrictions. We are going to have two different values of t, one for each sign of s. We can, from now on, consider that s is a positive number, and then we can use $t_1 = \sqrt{\tau - 2s}$ and $t_2 = \sqrt{\tau + 2s}$. And, since the square root matrices must have integer numbers, this means that each coefficient must stay an integer after dividing by t. Which means that:

- $A - s$ and $D - s$ are multiples of t_1.
- B and C are multiples of t_1.
- $A + s$ and $D + s$ are multiples of t_2.
- B and C are multiples of t_2.

These conditions allow for huge prunes of the search space. We can immediately see that both B and C will need to be multiples of $\operatorname{lcm}(t_1, t_2)$; the conditions for A and D are not so obvious, but it looks like the Chinese remainder theorem will be useful.

We can now start having an idea of what is the algorithm going to look like. We can iterate over t_1 and then over t_2. For each pair of values, we have $s = \frac{t_2^2 - t_1^2}{4}$. Conveniently, this is an integer if and only if t_1 and t_2 have the same parity, so we can iterate over all natural numbers for t_1, and then for t_2 we can consider $t_1 + 2$, $t_1 + 4$ and so on, up until we get to a value such that the trace $\tau = t_1^2 + 2s = \frac{t_1^2 + t_2^2}{2}$ is greater than the problem limit, $N = 10^7$. But for each combination of t_1 and t_2 (with the additional, derived s) there can still be many matrices, so let's look for additional conditions.

For example, if we look at the trace and the main diagonal values, A and D, we see something interesting: the trace equals $A + D = t_1^2 + 2s$, and yet we are going to need that $A - s > 0$ and $D - s > 0$ so that the square root

values remain positive. This means that we can "abstract" A and D into two values, $x = A - s$ and $y = D - s$, which must be positive (in particular, nonzero) and such that $x + y = t_1^2$.

These values also have an interesting effect on the determinant. First of all let's call $l = \operatorname{lcm}(t_1, t_2)$, so that B and C must be multiples of l. We can also say that $b = \beta l$ and $c = \gamma l$, where β and γ are, again, some strictly positive numbers that we are going to use as abstractions for B and C. With these "abstract" numbers, we can see that the determinant $s^2 = AD - BC$ is equal to

$$(s + x)(s + y) - \beta\gamma l^2 = s^2 + (x + y)s + xy - \beta\gamma l^2 = s^2 + t_1^2 s + xy - \beta\gamma l^2.$$

Since this is equal to s^2, we get this useful result:

$$t_1^2 s + xy = \beta\gamma l^2.$$

Now, by definition we have that l is a multiple of t_1, and this means that xy must be a multiple of t_1^2; yet another restriction on these values. Now, the conditions $x + y = t_1^2$ and $xy \equiv 0 \pmod{t_1^2}$ imply that both x and y must be multiples of t_1: if one of them is not a multiple of t_1, then the other must also not be, since the sum of both is a multiple of t_1, and therefore their product can't be a multiple of t_1^2. This is anyway required so that the matrix coefficients stay as integers, but it's nice to see that everything matches.

We can also add restrictions based on t_2. We need that $x + 2s$ and $y + 2s$ are multiples of t_2 so that the second square root is also integer. This is as easy as applying the Chinese remainder theorem. In particular, we might only apply it to x, and the value for y will follow. This is because the reasoning we did for $x + y = t_1$ can also be done "symmetrically" for t_2, although with subtractions: $A + D = t_2^2 - 2s$, and with $p = A + s$, $q = D + s$, we have $p + q = t_2^2$. Using the same reasoning as before, we can deduce that pq must be a multiple of t_2^2, so that p is a multiple of t_2 if and only if q is as well. Clearly $p = x + 2s$ and $q = y + 2s$, so if we find a value x_0 so that $x_0 \equiv 0 \pmod{t_1}$ and $x_0 \equiv -2s \pmod{t_2}$ the rest of the conditions will follow.

What do we do with this x_0? Simple. We have already determined that we are looking for values of x such that $0 < x < t_1^2$, and such that the modular equations hold. In other words, values in that range so that $x \equiv x_0 \pmod{\operatorname{lcm}(t_1, t_2) = l}$. That is: x_0, $x_0 + l$, $x_0 + 2l \ldots$ until we exceed t_1^2. Note that if $x_0 = 0$, which is a solution that appears, this is not valid and we need to use l as the starting value.

We are almost finished, but we haven't taken proper care of B and C. We have "abstracted" them into some values β and γ, and right now we only

know that $st_1^2 + xy = \beta\gamma l^2$. However: we have built the values of x and y in such a way that we can guarantee that $st_1^2 + xy$ is a multiple of l^2 (since it's trivially a multiple of t_1^2, and since a similar reasoning with $p = A + s$ and $q = D + s$ guarantees that it's a multiple of t_2^2). So we have this integer value,

$$m = \beta\gamma = \frac{st_1^2 + xy}{l^2}.$$

Do we have any additional conditions to apply? No, actually we haven't. We just need to decompose m as a product of β and γ. This means that, if we got this far, we can just *count the amount of divisors in* m as the total valid solutions. This gives us, finally, a complete algorithm for the problem, which is like this:

- Initialise a result variable as 0.
- Create a "first prime sieve" up to N, which we will use to determine divisors of a number. In practice the biggest number will be smaller than N, but really, $N = 10^7$ is tiny and the sieve takes well below a second.
- Now, iterate for all values from t_1 up until $t_1^2 \geq N$. And, for each t_1, iterate for values of t_2 bigger than t_1 and with the same parity, up until such a value that $\frac{t_1^2 + t_2^2}{2}$ is equal or bigger than N. This condition ensures that the matrix trace is below the problem limit. Now, for each pair (t_1, t_2):
  - Calculate $s = \frac{t_2^2 - t_1^2}{4}$. This will always be an integer number, and by construction, this implies that the matrix determinant is a perfect square, $\delta = s^2$, although we don't need this value.
  - Calculate the gcd and lcm of t_1 and t_2: $g = \gcd(t_1, t_2)$ and $l = \frac{t_1 t_2}{g} = \operatorname{lcm}(t_1, t_2)$.
  - Use the Chinese remainder theorem to find x_0. I used my standard Chinese remainder theorem calculator for coprime values, so that x_c is the solution of $x \equiv 0 \pmod{\frac{t_1}{g}}$ and $x \equiv \frac{-2s}{g} \pmod{\frac{t_2}{g}}$. Iterate over positive (i.e. nonzero) values of x such that $x \equiv gx_c \pmod{l}$ in the range $(0, t_1^2)$. These are the values of x that guarantee that A and D will be integer values in the square root matrices, and the range makes them stay positive.

    - Calculate $y = t_1^2 - x$ and $m = \dfrac{st_1^2 + xy}{l^2}$. I think this is always a perfect square, but I haven't bothered trying to prove it.
    - Use the "first prime" sieve to get the prime decomposition of m, and use this prime decomposition to calculate the amount of divisors of m, d.
    - Increase the result variable in d.
  - Nothing else to do with these t_1 and t_2.

- The result variable after the loop contains the solution to the problem.

The run time takes 1,2 seconds. Most of the approaches in the problem thread follow a similar scheme, but some use different ways, such as the study of eigenvalues.

431. Square Space Silo

Difficulty rating: 40 %.
Solution: 23.386029052. *Solved: Sun, 24 Jul 2022, 17:19.*
Math knowledge used: trigonometry, integration by tubes.
Programming techniques used: binary search.

Lately I'm using Mathematica a lot. Or maybe I'm doing too many problems with integrals. Anyway, like about 75 % of the problems whose answer is a real number with decimal digits and not an integer, this one is far easier than it looks (additional note: yeah, the other 25 % is, usually, legitimately hard).

The key to this problem is, of course, the calculation of the "wastage" volume given the parameters of each case (the radius of the silo, R; the horizontal distance between the centre and the vertex of the cone-like figure, d; and the angle of repose, α). The problem has a kind of circular symmetry in the sense that all points at the same distance to the vertex have the same vertical "wastage", which allows us to group them and integrate accordingly. Unfortunately the border of the silo is not concentrical to these circumferences, so we can't integrate a whole "tube". But we can still use integration by tubes, just not with full circumferences. It's clear that for smaller radii we are going to be able to use small circumferences, and for greater radii only a portion of the circumference lies inside the silo. So the first thing we need is to establish the borders.

For simplicity, we can analyse the problem in two dimensions, ignoring the height for now. So the silo is a circumference in the plane, and we can assume without loss of generality that both the vertex and the silo centre

lie in the horizontal axis. In particular, we choose the origin $(0, 0)$ as the position of the vertex, and the silo centre will be located at some point to its left, $(-d, 0)$ with $d \in [0, R]$. This allows us to define the integration limits, separated into two parts.

- Since the silo centre is located at $(-d, 0)$, the intersection between the silo and the horizontal axis is the interval $[-R - d, R - d]$. This means that the valid radii we need fall into the interval $x \in [0, R + d]$.
- For any radius $x < R - d$, any circumference centred in the origin is fully inside the silo and we can use $2\pi r$ as the arc length.
- For $x \in [R - d, R + d]$, the circumference is only partially inside the silo. In particular, there are two intersections between the circumference of radius x and the silo border: these are (a, b) and $(a, -b)$ for some real values a and b, which we can calculate solving a system of equations. In fact, we will use only the value of a (we also have $b = \sqrt{x^2 - a^2}$, but we don't care about that). The equations are:

$$\begin{aligned} (a + d)^2 + b^2 &= R^2; \\ a^2 + b^2 &= x^2. \end{aligned}$$

Subtracting them, the squares go away and we are left with an easier equation:

$$2ad + d^2 = R^2 - x^2 \Rightarrow a = \frac{R^2 - x^2 - d^2}{2d}.$$

This value allows us to calculate the arc length, since $-\dfrac{a}{x}$ is the cosine of one half of the arc. In other words, the arc length is exactly

$$\theta(x) = 2 \arccos \frac{d^2 + x^2 - R^2}{2dx}.$$

As a sanity check, we can see that $x = R - d$ and $x = R + d$ give us the extremes. In particular:

$$2 \arccos \tfrac{d^2 + (R-d)^2 - R^2}{2d(R-d)} = 2 \arccos \tfrac{2d^2 - 2dR}{2d(R-d)} = 2 \arccos(-1) = 2\pi;$$
$$2 \arccos \tfrac{d^2 + (R+d)^2 - R^2}{2d(R+d)} = 2 \arccos \tfrac{2d^2 + 2dR}{2d(R+d)} = 2 \arccos 1 = 0.$$

With the borders having been determined, we can now start calculating volumes. The angle of repose α implies that at each radius x we have a

common "wastage" height, $h = x \tan \alpha$. So the integrand function at each point is the length of the arc, $x\theta\,(x)$ (or $2\pi x$ in the complete case) multiplied by this height. Therefore we can calculate the wastage as a function of d (with fixed R and α) like this:

$$w\,(d) = \int_0^{R-d} 2\pi \tan \alpha \cdot x^2 \,\mathrm{d}x + \int_{R-d}^{R+d} 2 \arccos \frac{d^2 + x^2 - R^2}{2dx} \tan \alpha \cdot x^2 \,\mathrm{d}x.$$

This looks difficult, but hey, we won't need to actually calculate the integral; that's what Mathematica is for. We can also see that this does not yet solve the problem, but most of the mathematics is already there and we only need to calculate the formula above a bunch of times. First of all we can observe that the minimum wastage happens when $d = 0$, and the maximum happens when $d = R$. I haven't tested it explicitly but I'm pretty sure that this is in fact an increasing function of d, at least in the interval $[0, R]$, which is what makes sense anyway. We only need perfect squares that fall within that range for the given parameters, $R = 6$ and $\alpha = \dfrac{40}{180}\pi$ (yes, radians, PLEASE), and it can be seen that there are only six of them: from 20^2 to 25^2. For each of these values, we can do a binary search in Mathematica in the range $d \in [0, R]$ (using numerical integration in order to take a non-infinite amount of time), which calculates the full result in about 2 seconds. Most of the people in the problem thread is doing basically the same, and in fact a lot of them use Mathematica. Although this case is so simple that I could also have done manual numerical integration using any trapezium method or whatever.

440. GCD and Tiling

Difficulty rating: 60 %.
Solution: 970746056. *Solved: Wed, 5 May 2021, 03:23.*
Math knowledge used: recurrence equations, GCD properties.
Programming techniques used: binary exponentiation.

This was a fun problem to unravel. My solution is $O\,(n^2 \log n)$ and I don't think you can go much farther than that. Some people in the forum have better solutions, but ultimately they are all of the same order, because even if they do some additional magic to precompute values, instead of repeated binary exponentiation, the initial exponent analysis is still $O\,(n^2 \log n)$.

The very first part of the problem consists on finding a recurrence formula for $T\,(n)$. This is very easy because the construction is very simple. With

$T(0) = 1$ and $T(1) = 10$, the recurrence is

$$T(n) = 10T(n-1) + T(n-2).$$

This allows quick calculations for $T(n)$ for reasonable values of n, using binary exponentiation. However, the values grow pretty fast ($T(n) \gtrsim 10^n$, so $T(2000^{2000})$ doesn't fit in memory) and calculating the GCDs with the full values is not feasible. Of course, using GCD with moduli is not correct and can't be done either. So we need to do some finer analysis on the properties of the succession. There are two key observations for this problem, one of which is related to the $T(n)$ succession and the other of which is related to the GCD operation.

The observation related to $T(n)$ is relatively simple and in fact it's also present on the Fibonacci numbers. It's the following relationship:

$$\gcd(T(i), T(j)) = T(\gcd(i+1, j+1) - 1). \tag{2}$$

This can be found by doing some quick experiments, but someone on the problem thread has an outline of the proof, which is not complicated. Anyway, this goes a long way towards a full solution, but it's not enough. We need to calculate arbitrary values of the form $T\left(\gcd\left(c^a, c^b\right)\right)$, so what the previous formula tells us is that we need to find $\gcd\left(c^a + 1, c^b + 1\right)$. In particular,

$$\gcd\left(T(c^a), T\left(c^b\right)\right) = T\left(\gcd\left(c^a + 1, c^b + 1\right) - 1\right). \tag{3}$$

And now comes the second observation, which is also moderately easy to prove, but which is more complex since it features several cases: the value of $\gcd\left(c^a + 1, c^b + 1\right)$ depends mostly on operations of a and b, and there are the following cases:

- If $a = b$, obviously $c^a + 1 = c^b + 1$ so their GCD is also $c^a + 1$.
- Otherwise, first we calculate $g = \gcd(a, b)$ and $a' = \frac{a}{g}$, $b' = \frac{b}{g}$.
- If both a' and b' are odd, then $\gcd\left(c^a + 1, c^b + 1\right) = \gcd(c^g + 1)$.
- But if any (or both) of $\{a', b'\}$ are even, then the GCD depends on c: it's 2 if c is odd, and 1 if c is even.

These formulas are already good enough to calculate everything we need, but we need a workable way to do them. The first thing to note is that, since the boundaries for a and b are common to all values of c, we can do the full

exponent analysis beforehand: for each $\{a, b\}$ pair, we compute their gcd and store the result of the calculations above, storing the amount of times each GCD is found and storing also a special "base" counter for the cases where a' or b' is even. Annoyingly, this can't be easily done with a sieve, because of the distinction between odd and even multiples, so the reasonable option is the $O(n^2)$ iteration of GCDs. I guess that sieving is possible, but it's harder to get right. After this analysis is done, we can just iterate over values of c; for each one of them, we first create an array of size $L+1$, so that the values are calculated using the following recurrence: start with the transition matrix of the recurrence, $M_0 = \begin{pmatrix} 10 & 1 \\ 1 & 0 \end{pmatrix}$; for each i in $[1, L]$, calculate the matrix $M_i = M_{i-1}^c$ using binary exponentiation, and store the upper left element, $M_i[1,1]$, in the i-th position of the array. Now it's a matter of iterating over the GCD "counts", multiplying the number found in position i of the array times the GCD counter for exponent i, and after that, adding either $T(1) = 10$ or $T(0) = 1$ (depending on whether c is, respectively, odd or even) multiplied by the amount found on the special "base" counter. And that's it.

441. The inverse summation of coprime couples

Difficulty rating: 65 %.

Solution: 5000088.8395. *Solved: Sat, 22 Jan 2022, 06:26.*

Math knowledge used: harmonic series, inclusion-exclusion, Möbius function, combinatorics, recurrence equations, Erathostenes sieve (for divisor generation).

Programming techniques used: none.

At first I wondered why this problem was rated 65 %. Then I reread it and noticed that it wasn't asking for $R(n)$ but for $S(n) = \sum_n R(n)$, and I realised that this wasn't going to be as easy as I thought.

The trivial calculation for $R(n)$ is about $O(n^2 \log n)$. With clever usage of some precalculation combined with inclusion-exclusion based on prime factors lattices, this can be reduced to $O(n \log n)$, but this means $O(n^2 \log n)$ for $S(n)$, which is still not good. However, it's pretty trivial to see that $R(n-1)$ and $R(n)$ are going to share a lot of terms, so it makes sense to try to calculate $\Delta R(n) = R(n) - R(n-1)$ in a reasonable time and then use this to calculate $R(n)$ and, in turn, $S(n)$. I found a way to calculate deltas in $O(\log n)$ time, yielding a $O(n \log n)$ algorithm for the whole calculation.

It's probably possible to reorder summations so that the algorithm is reduced even further to $O(n)$, but I haven't thought much about it, and the scattered sign changes probably make it not very easy, if it's even doable.

Now, for the algorithm itself. The key, as mentioned above, is calculating $\Delta R(n)$ in a reasonable time, possibly with the help of some precalculations. The first thing to do, obviously, is to find a systematic formula for $\Delta R(n)$, even if it's not directly workable. What I found is that I can separate two kind of terms, the ones added and the ones removed, so that we have:

$$\begin{aligned}
\Delta R^{+}(n) &= \sum_{\substack{d\in[1,n]\\ \gcd(n,d)=1}} \frac{1}{dn};\\
\Delta R^{-}(n) &= \sum_{\substack{1\le a<b<n-1\\ a+b=n-1\\ \gcd(a,b)=1}} \frac{1}{ab};\\
\Delta R(n) &= \Delta R^{+}(n) - \Delta R^{-}(n).
\end{aligned}$$

Calculating $\Delta R^{+}(n)$ looks easy enough with inclusion-exclusion. Let's call $h(n) = \sum_{i=1}^{n} \frac{1}{i}$, the prefix sums of the harmonic series. Clearly we can extract n so that the term becomes

$$\Delta R^{+}(n) = \frac{1}{n} \sum_{\substack{d\in[1,n]\\ \gcd(n,d)=1}} \frac{1}{d},$$

And this sum over coprime terms can be calculated with a lattice of prime factors, just like I used on problem 777. Let $p(n)$ be the list of primes that divide n, so that for example $p(n) = \{2,3\}$. From this list we generate a lattice of prime products (along with the value of their Möbius function), generating the set

$$l(n) = \left\{ \prod_{i\in S} : S \in \mathcal{P}(p(n)) \right\}.$$

The formula looks complex, what with the power set and all, but it's pretty simple. Continuing with the case $n = 12$, we generate the full set of products: $\{1,2,3,6\}$, and the Möbius function for these values can also be calculated on the fly: $\mu(1) = 1$, $\mu(2) = -1$, $\mu(3) = -1$, $\mu(6) = 1$. In practice, what we are doing is enumerating the distinct factors of n which are squarefree, and therefore their Möbius function is non zero. Now, given this set $l(n)$, we can

use this formula:

$$\Delta R^{+}(n) = \frac{1}{n} \sum_{d \in l(n)} \frac{1}{d} \mu(d)\, h\left(\frac{n}{d}\right).$$

Considering that $l(n)$ can be calculated in $O(\log n)$ with the help of a pre-calculated sieve of first primes, this solves the easy half of the problem.

Now let's go for $\Delta R^{-}(n)$, which isn't much more complicated, but it requires knowing about certain kind of obscure series, which I found on OEIS. Let's go step by step: the terms in $\Delta R^{-}(n)$ look like this for, say, $n = 8$:

$$\Delta R^{-}(8) = \frac{1}{1 \cdot 6} + \frac{1}{2 \cdot 5} + \frac{1}{3 \cdot 4}.$$

Looks easy enough (although the sum is not obvious), but when we look at $\Delta R^{-}(13)$, we can see many terms missing:

$$\Delta R^{-}(13) = \frac{1}{1 \cdot 11} + \frac{1}{5 \cdot 7}.$$

The good news is that the same scheme of inclusion-exclusion works. The bad news is that we need the sum of

$$f(n) = \sum_{\substack{1 \le a < b < n \\ a+b=n}} \frac{1}{ab}$$

for certain n, and this is where I hit a bit of a roadblock. The first thing I did was to thing in term of the "whole" series,

$$f'(n) = \sum_{\substack{1 \le a,b < n \\ a+b=n}} \frac{1}{ab}.$$

For odd numbers, $f'(n) = 2f(n)$. For even numbers there is an additional term, but it's going to be removed because of the inclusion-exclusion, since it's where $a = b = \dfrac{n}{2}$ and therefore $\gcd a, n \neq 1$ except for the case $n = 2$. Because of this, the subsequent formulas only work for $n \geq 3$, and we need to start the iteration from $n = 4$, adding manually the previous values. Not a big deal, but it must be taken into account.

Now, $f'(n)$ is more "regular", and it always includes $n - 1$ terms. For example, for $n = 5$ it looks like this:

$$f'(5) = \frac{1}{1 \cdot 4} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 2} + \frac{1}{4 \cdot 1}.$$

This is still not trivial to sum, but I suspected that this was probably a well known succession. What I did now was to multiply it times a factorial, to get a succession of integer values. I got this:

$$g(n) = n!f'(n) = \{0, 2, 6, 22, 100, 548, 3528, \ldots\}.$$

Still doesn't ring a bell. However, looking for that sequence on OEIS yielded the sequence A052517, *Number of ordered pairs of cycles over all n-permutations having two cycles*, so that $a(n) = g(n-1)$ (since this OEIS sequence has an additional 0 at the start of the sequence). This is the "combinatorics" part I added to the problem's list of required mathematics, by the way. Now, the description of the sequence doesn't say much, but OEIS helpfully included a valuable formula which, once accommodated to my definition of $g(n)$, looked like this:

$$g(n) = (2n-3)\,g(n-1) - (n-2)^2\,g(n-2).$$

Jackpot. This is the last piece I needed to calculate $\Delta R^-(n)$. With this we can find a formula for $f'(n)$:

$$f'(n) = \frac{2n-3}{n}f'(n-1) - \frac{(n-2)^2}{n(n-1)}f'(n-2).$$

The initial terms of this series are $f'(1) = 0$ and $f'(2) = 1$. Now, the exact half of this series would be

$$f^*(n) = \frac{2n-3}{n}f^*(n-1) - \frac{(n-2)^2}{n(n-1)}f^*(n-2),$$

with initial terms $f^*(1) = 0$, $f^*(2) = \frac{1}{2}$. This series matches the one we want, $f(n)$, for every value except for $n = 2$ (where $f^*(2) = \frac{1}{2}$, but $f(2) = 0$).

A bit or reordering and Möbius magic tells us that

$$\Delta R^-(n) \sum_{d \in l(n-1)} \frac{1}{d^2}\mu(d)\,f^*\left(\frac{n-1}{d}\right).$$

Note the usage of $l(n-1)$ instead of $l(n)$. This makes sense since these are terms that were present in $R(n-1)$ but not in $R(n)$. Note again that this formula only works for $n > 3$ because of the anomaly in $f^*(2)$.

Anyway, this gives us a workable formula. Start with $R(3) = 1$ and $S(3) = 1{,}5$, and for $i \in [4, N]$, just do

$$\begin{aligned} R(i) &= R(i-1) + \Delta R^{+}(i) - \Delta R^{-}(i); \\ S(i) &= S(i-1) + R(i). \end{aligned}$$

Using doubles, and with precalculations of $h(n)$, $f^{*}(n)$ and $l(n)$, this calculates $S(10^7)$ in about 6 seconds, but unfortunately that yielded the wrong result because of precision issues. I had to resort to BigDecimals with 30 digits of precision, which finally give the correct result in about 75 seconds.

In the problem thread there are a lot of different approaches based on the Möbius formula, but I haven't seen anyone doing exactly what I did, which is surprising to me.

442. Eleven-free integers

Difficulty rating: 60 %.
Solution: 1295552661530920149. *Solved: Sun, 5 Jun 2022, 15:07.*
Math knowledge used: inclusion-exclusion.
Programming techniques used: none.

An unusual case, for two different reasons. The first one is that this is a very lightweight problem in the mathematical sense; it's almost purely algorithmic, with just some inclusion-exclusion. The second one is that my algorithm is wrong somewhere, yet I'm getting the right answer for this problem. For the base cases I'm not getting a correct value. I got lucky, I guess.

First of all: no number theory or discrete math. No prime numbers or magic related to the number 11. Just pure algorithmic and digit matching, and relatively basic inclusion-exclusion. There aren't any complicated data structures to be used, either. I did use a lot of recursion at two different levels (the kind of recursion where a call spawns an indeterminate amount of calls, like a big tree).

How to proceed? We need to consider the powers of 11 up to 11^{17} (11^{18} is too far away to be useful). Some of them can be actually ignored (in particular, 11^{11} contains the substring "11"; twice, although having it once is enough to make it redundant; also, 11^{13} contains the substring "121"). Now we need to consider ways to "hook" these powers, because we need to consider cases like "11331" and so on. Now, my first attempt consisted on trying to generate all these from the beginning, and I apparently used a very bad data structure, which made the code very slow (30 seconds, and that's just before

starting the count itself), so I restarted the problem with a completely new approach, based on "hooking" dynamically. Now, everything is in terms of digits, so I used a `DigitPattern` class which is basically a glorified array of digits with some utility functions. One of these functions is used to find "hooks" between two patterns. For example, 11 can be "hooked" with 121 because the end of the first pattern overlaps with the beginning of the second. There is even a case where a power ends in 161, so it can be "hooked" with $11^5 = 161051$ (a favourite number of mine for historical reasons!!) in two different ways.

Now, this list of "hooks" can be precalculated, and that's indeed very useful because we are going to use this information a lot. The algorithm is more or less like this:

- Calculate the amount of "eleven-free" numbers in an interval.
- If we haven't reached the expected amount, move to the next interval.
- Otherwise, if we have exceeded the limit, discard this calculation and do it again, but refined (with an additional digit of precision).
- If the amount of remaining "eleven-free" numbers is below a certain threshold, just start looking manually, number by number.

So, let's say that we start with the interval $(0, 10^{18})$. Obviously there are fewer than 10^{18} eleven-free numbers in that interval. So we would move to the next interval, $(10^{18}, 2 \cdot 10^{18})$. But now we exceed the expected number, so let's use subintervals like $(10^{18}, 11 \cdot 10^{17})$, $(12 \cdot 10^{17}, 13 \cdot 10^{17})$ (note that we skip the set that starts with 11, of course) and so on. This is the first level of recursion, and it's the simplest one. Now, with this flow in place, the next part of the problem consists of calculating the amount of "eleven-free" numbers in an interval. This is harder than the first part, because of all the patterns that get mixed.

The recursion I set for this second part needs to take into account the fact that patterns might be counted several times. Let's say that we are counting six-digit numbers. So we count numbers like $x11xxx$, $xx121x$ and $x1121x$ separately. Of course we are going to need inclusion-exclusion. Now, the good news is that this is the simplest case of inclusion-exclusion, where repeated elements don't get in the way (like in, say, problem 768), nor we have tricky concepts like the Möbius function, so it's a very simple calculation: if we have included an *odd* amount of patterns, we subtract; if we have included an *even* amount (including the base case 0), we add. We still need some finesse regarding the "hooks", and for that, I create a recursive function with four parameters:

- The first element is the last pattern used. This is useful so that we know which "hooks" we can use. The latest pattern might be an empty string (this is the case at the very beginning, and never again), or the latest power of 11 used, but it could also be the prefix we are using (for example, in the interval $(12 \cdot 10^{17}, 13 \cdot 10^{17})$, the prefix is 12).

- The second element is the amount of digfits remaining. In the previous case, we would have 17, but this gets reduced as we get deep into the recursion.

- The third element is what I call the "extra" digits. These are digits that don't count for the recursion, but that we need to use to calculate how many numbers are there at every set. These are digits that remain to the left of the latest pattern used. For example, let's say that we initially have five digits of leeway. Now, we can introduce an 11 at the very start, resulting in $11xxx$, so three digits would remain. But we can also introduce at 11 after the first digit, resulting in $x11xx$. In this second case we only have two remaining digits, but there is an extra digit: there are 10^3 numbers in this set, not 10^2.

- The last element is a boolean indicating whether we need to add or subtract.

Now, the algorithm gets clearer. We start with a given prefix and a given amount of digits. Initially the "extra" set has zero digits. And the boolean starts being true. Now, we can either "hook" a pattern to the latest pattern (i.e. the prefix), or add a pattern at any position. But before that, we calculate the amount of elements in the current set, which is 10^p where p is the sum of the remaining and the extra digits. We will change the sign of this element if the boolean so indicates. At each recursive call we use the latest pattern, and both digit counts get updated accordingly depending on which pattern we have used, and which position it's at. The boolean always gets toggled at every level.

Now this is the kind of recursion where the depth is not very high, but the tree is very wide and there are a lot of subcases. So there are a lot of calls, but not very deep ones. The result is acceptably fast, at about 2,1 seconds of run time. Since the recursion is so complex, it's hard to calculate the order of the algorithm, but I don't think it's more than $O(\log^a n)$ for some unknown a. In the problem thread there are creative solutions using automata and things like that, but nothing too outlandish. This problem is more algorithmic than anything else, and more tedious than difficult.

456. Triangles containing the origin II

Difficulty rating: 55 %.
Solution: 333333208685971546. *Solved: Fri, 2 Apr 2021, 14:52.*
Math knowledge used: basic trigonometry.
Programming techniques used: none.

At first this seems impossible, short of checking all the n^3 triangles, since the points are basically random. However, this can be done using a main loop with *linear* time (although before that, some sorting is required, making the whole algorithm $O(n \log n)$ in the end). Of course, there is a lot of work before getting to that level of finesse.

First of all: points themselves are only needed for the initialisation. The main loop is based on indices and nothing more. Also, the points are discarded almost immediately: we actually need just the angles (as in $\alpha = \arctan \frac{y}{x}$), thanks to an interesting property: we can determine whether a triangle contains the origin using these, and the actual distance from the points to the origin isn't necessary for anything. In fact, given two points with angles α and β, a triangle whose third points is at angle γ will contain the origin if and only if $\gamma \in (\alpha + \pi, \beta + \pi)$. This suggests a three dimensional iteration scheme, which is the base for my algorithm although the end result is just a single loop. The scheme is as follows:

- Use angles to represent points. To avoid precision issues, use rational values to represent them. I used a class with members x and y, and an additional "quadrant" variable which was helpful for comparing two angles quickly (which is used for the sort).

- All the angles where there is at least one point will be inserted into a sorted array. Let N be the total amount of different angles (which is slightly smaller than the amount of points, $2 \cdot 10^6$, because there are angles with repeated points, most commonly the horizontal and vertical axes). In the end we will have an array where, given an index i, the angle it represents is $\alpha(i)$. The first angle is the horizontal positive axis, and the last one is slightly below that same axis, after having circled over all the angle space.

- Let $f(i)$ be the amount of points whose angle is $\alpha(i)$, and let $F(i) = \sum_{n=0}^{i} f(n)$. We also define $k(i)$ as: the last index before $\alpha(i) + \pi$ if $\alpha(i) < \pi$, or $X - 1$ otherwise. For each index i in the array, we store $f(i)$, $F(i)$, $k(i)$, and a boolean flag indicating whether $\alpha(1 + k(i))$ is exactly $\alpha(i) + \pi$ or not. With this flag, we define $k'(i)$, the first index

after $\alpha(i)+\pi$, as $k(i)+1$ if the flag is set to FALSE, or $k(i)+2$ if it's set to TRUE.

- The solution of the problem comes from the following formula:

$$S=\sum_{i=0}^{X}\sum_{j=i+1}^{k(i)}\sum_{k=k'(i)}^{k(j)} f(i)\cdot f(j)\cdot f(k).$$

  Note that, thanks to having limited $k(i)$ to the end of the circle, we are counting each triangle exactly once.

Now we can start working on this formula to get a $O(n)$ iteration (again: generating the array is $O(n\log n)$ because it's sorted and because finding $k(i)$ is a $O(\log n)$ operation, meaning that finding it for every index is also $O(n\log n)$; therefore the whole algorithm is $O(n\log n)$ and not $O(n)$). The first operation is, obviously, extract common factors:

$$S=\sum_{i=0}^{X} f(i)\cdot\sum_{j=i+1}^{k(i)} f(j)\cdot\sum_{k=k'(i)}^{k(j)} f(k).$$

Unfortunately, these sums are not yet separable. However, since we have been accumulating the values of f, we can remove the last summatory:

$$S=\sum_{i=0}^{X} f(i)\cdot\sum_{j=i+1}^{k(i)} f(j)\cdot\left(F(k(j))-F(k'(i)-1)\right).$$

And just with that we move the sum from $O(n^3)$ to $O(n^2)$. Note that $k'(i)-1$ is $k(i)$ if the boolean flag was set to FALSE, and $k(i)+1$ if it was set to TRUE.

The next goal is to separate the summations of i and j. In fact, what we want to do exactly is to have a summation in j that doesn't include i at all. First, we separate both terms of F:

$$S=\sum_{i=0}^{X} f(i)\cdot\left(\sum_{j=i+1}^{k(i)} f(j)\cdot F(k(j))-\sum_{j=i+1}^{k(i)} f(j)\cdot F(k'(i)-1)\right).$$

Note however that the second F term doesn't depend on j, and we can extract it from the summatory:

$$S = \sum_{i=0}^{X} f(i) \cdot \left(\sum_{j=i+1}^{k(i)} f(j) \cdot F(k(j)) - F(k'(i)-1) \cdot \sum_{j=i+1}^{k(i)} f(j) \right).$$

Now, that last summatory can be expressed in terms of the values of F we have been collecting:

$$S = \sum_{i=0}^{X} f(i) \cdot \left(\left(\sum_{j=i+1}^{k(i)} f(j) \cdot F(k(j)) \right) - F(k'(i)-1) \cdot (F(k(i)) - F(i)) \right).$$

We still have an $O(n^2)$ summation, but we're very close to a linear summation because we can easily notice that the internal summatory is mostly common for each pair of consecutive values, $i-1$ and i. Let's define now:

$$\begin{aligned} g_1(i) &= \sum_{j=i+1}^{k(i)} f(j) \cdot F(k(j)); \\ g_2(i) &= F(k'(i)-1) \cdot (F(k(i)) - F(i)); \\ g(i) &= g_1(i) - g_2(i). \end{aligned}$$

With these definitions, we have

$$S = \sum_{i=0}^{X} f(i) \cdot g(i).$$

And here comes the trick. First, we calculate $g(0)$ normally (I'm using programming terms, so the first element of the array is 0). This is an $O(n)$ operation. But now, we can calculate g using differentials:

$$\begin{aligned} \Delta g_1(i) &= g_1(i) - g_1(i-1); \\ \Delta g_2(i) &= g_2(i) - g_2(i-1); \\ \Delta g(i) &= g(i) - g(i-1) = \Delta g_1(i) - \Delta g_2(i). \end{aligned}$$

Calculating $g_2(i)$ is already a constant time operation, so Δg_2 is also constant time. And regarding $g_1(i)$, we have:

$$\Delta g_1(i) = \sum_{j=i+1}^{k(i)} f(j) \cdot F(k(j)) - \sum_{j=i}^{k(i-1)} f(j) \cdot F(k(j)) =$$
$$= \left(\sum_{j=1+k(i-1)}^{k(i)} f(j) \cdot F(k(j)) \right) - f(i) \cdot F(k(i)).$$

The trick is that this summation has very few terms (we rarely expect it to have more than 2 or 3 terms, since $k(i-1)$ and $k(i)$ will be very close), and in the vast majority or cases there are either 0 or 1 terms. So this is, de facto, also constant time. Now, the outline of the algorithm is much more clear:

- Generate all the points, inserting them into a sorted map, counting the amount of times each angle appears.
- Process the sorted map and generate the array of objects described above (with $f(i)$, $F(i)$, $k(i)$ and the boolean flag).
- Calculate $g(0)$, and initialise the sum as $f(0) \cdot g(0)$.
- For each i, calculate $g(i)$ as $g(i-1)$ (already calculated) and $\Delta g(i)$, which is a constant time operation. Therefore, adding $f(i) \cdot g(i)$ to the end result is also a constant time operation. The end result is the sum we're looking for. We can abort the iteration as soon as we hit $g(i) = 0$, which should happen around $\alpha(i) = \pi$, i.e. at the middle of the array, give or take.

The first two steps (the initialisation) are $O(n \log n)$, and the other two are $O(n)$. At first I thought that Fenwick trees could have been necessary, but no, just storing the accumulated values is good enough. By the way, all these algorithmic orders I've been mentioning are related to time complexity, but the space complexity is also $O(n)$.

The runtime is about 3,7 seconds. Without the summation differential analysis, the algorithm is $O(n^2)$ and the run time is almost two hours and a half! Which is almost acceptable, in the sense that I could have done it if I hadn't found a faster way, but why do it the slow way if we can have fun coding the fast algorithm?

Finally, some trivia about this problem. Given three random points, the probability that the triangle they form contains the origin is $\frac{1}{4}$. So, given 2000000 points, the amount of expected triangles including the origin

would be $\frac{1}{4}\binom{2000000}{3} = 333332833333500000$. The result of this program is slightly above that: 333333208685971546.

481. Chef Showdown

Difficulty rating: 70 %.

Solution: 729.12106947. *Solved: Sat, 5 Jun 2021, 14:43.*

Math knowledge used: basic probability, Markov chains, block matrix inversion.

Programming techniques used: dynamic programming.

That 70 % has very good reason to be there. This problem is HARD. I feel like this is close to my limit, in terms of difficulty, but anyway I managed to solve it, and I got the really complicated part right at the first try, which is a triumph. That "block matrix inversion" mentioned was difficult to nail, but once I got the theory right, the code is very fast. Not bad, for a $10^5 \times 10^5$ matrix inversion.

Now, let's discuss the solution. The "count the amount of steps" format of the problem makes it clear that the way to go here is using Markov chains, just as in other similar problems like 227. Therefore the problem has two separate parts: one, generate the transition matrix. Two, get the solution from Markov chains theory's magic. Initially I thought that the first part was the hard one. I was very wrong.

Ok. Markov chains is it. The first thing to define is the problem states. We have every possible combinations from 2 to 14 players, and for each one of them, the player holding the current turn must also be part of the state. This means that for each combination of N players we will have N states, one per player. And this means that the total amount of states is (drum roll):

$$\sum_{n=2}^{14} n \cdot \binom{14}{n} = \sum_{n=0}^{14} n \cdot \binom{14}{n} - \left(1 \cdot \binom{14}{1} + 0 \cdot \binom{14}{0}\right) = 14 \cdot 2^{14-1} - 14 = 114674.$$

Oops. A matrix of that size doesn't look very easily invertible. Whatever, that's a problem for the future. For now, we can concentrate on the first part of the problem, which is calculating the transition probabilities. The problem description throws a red herring here, because it shows the probabilities of each chef winning like it matters, but this is actually not needed for the solution, although it can be calculated as part of one of the loops of the code (tecnically, only the probabilities for states with 13 or fewer players need to be calculated, but depending on how you structure the loops, you

might end up calculating them for the states with exactly 14 players as well). Nevertheless, the provided sample for $N = 7$ is useful to make sure that the calculations so far are correct.

But I digress. What we need for the first part of the problem is, as I said, not the probabilities of each state, but rather, the chef that the turn winner chooses to eliminate if it wins: each player has a constant probability $S(k)$ of winning its turn, and at each turn, there is a $S(k)$ probability that the next state will have one fewer player than the current set, and a $1 - S(k)$ probability that the next state will have the same set of players as the current one. In any case, the next turn owner will be different (and this needs to be taken into account). Unfortunately, to determine which player is chosen to be eliminated, the probabilities are needed. In order to do this, we use dynamic programming: we start with the base cases with 2 players, and at each step we use the probabilities for N players to build both the probabilities and the chosen losers for $N+1$ players. For a given set of players of size N, and having calculated all the cases for $N-1$ players, we first determine who is chosen by each player to be removed if they win their turn, and then the probabilities of each player winning are calculated (this can be technically skipped for the very last case for $N = 14$, as I mentioned before). The eliminated players are chosen using a simple loop to look for a maximum, while the probabilities are calculated by solving an equation system with N^2 unknowns.

The full procedure for this first part is as follows. We start by calculating the probabilities of each chef to win their turns, which is straightforward. Then, all the states are generated and grouped by amount of players, then by set of current players; inside these sets, we index each state per current turn owner. So we have a three level table where the indices are ints. As I advanced, we iterate first by amount of players, so we start at $N = 2$ players. For each combination of 2 players, say P_1 and P_2 with probabilities S_1 and S_2, it's pretty clear that each player chooses to eliminate the other, which gives themself a probability of winning equal to 1, of course. But we need to calculate the probability of each player to win, both when it's their turn and the other player's. Note that this is the *final* probability of winning, that is, the total probability that each player has to, eventually, win. If we call P_{ij} the probability of player j winning when the turn owner is i, the probabilities can be constructed with a system of four equations. When it's the turn of P_i, player i wins with probability S_i, and the turn passes to j with probability $1 - S_i$. The unknowns are P_{11}, P_{12}, P_{21} and P_{22}, and the equations are:

$$\begin{aligned} P_{11} &= S_1 + (1 - S_1) P_{21}, \\ P_{12} &= (1 - S_1) P_{22}, \end{aligned}$$

$$\begin{aligned} P_{21} &= (1 - S_2)\, P_{11}, \\ P_{22} &= S_2 + (1 - S_2)\, P_{22}. \end{aligned}$$

This system has an easy solution (especially since it can be separated into two smallers systems which two equations each, since the probabilities of each player don't affect the other's):

$$\begin{aligned} P_{11} &= \frac{S_1}{S_1 + S_2 - S_1 S_2}, \\ P_{12} &= \frac{S_2 - S_1 S_2}{S_1 + S_2 - S_1 S_2}, \\ P_{11} &= \frac{S_2}{S_1 + S_2 - S_1 S_2}, \\ P_{11} &= \frac{S_1 - S_1 S_2}{S_1 + S_2 - S_1 S_2}. \end{aligned}$$

Ok, this was the base case. Now we need to use these probabilities, in a dynamic programming of sorts, to escalate on the amount of players. So, for the next step, we assume that the probabilities have been calculated for all the states of size $N - 1$, and we need to work on states of size N. The first thing to do is use the known probabilities of smaller states to determine who is chosen to be eliminated at each turn. Given a set of players (and all its associated states, each corresponding to a turn owner), $\{P_i\}$, each player proceeds by trying to remove each player $\{P_j : j \neq i\}$. If a player j is removed, the transition goes to a state where there is one fewer player and the turn owner has moved as well. Therefore for each removed player j there is a state X_j that follows. We determine the probability of winning for P_i in each of these states, and we choose the state which gives the maximum probability (and, according to the problem rules, if two probabilities match, the closest turn owner is chosen. This benefits P_2 disproportionately over P_1, by the way. This is very apparent in the final probabilities). Let X_i be the new state, and X_{ij} will be the probability of player j winning when the player i has eliminated some player and the turn has moved to state X_i (which will be 0 if j happens to be the eliminated player). We have these probabilities from the previous step (dynamic programming, yay), and thus we can build all of the equations with the same structure:

$$P_{ij} = S_i X_{ij} + (1 - S_i)\, P_{i+1,j}.$$

Here, the index $i+1$ doesn't indicate player $i+1$ (which might not be present in the current state), but rather, the next player after i. Similarly, in the state

X_i the turn owner is not i, but another player (which could be, following the same pattern, $i+2$ if the next player happens to be the eliminated one, or $i+1$ otherwise).

This process ends when all the states up to $N = 14$ have been iterated over. Recapitulating: at each step we have a set of N players, and all the probabilities for all the cases with $N-1$ players are already known. Using these, we calculate (and store) the following pieces of data:

- The probabilities that each player has to win, for each turn owner.
- The player chosen to be eliminated by each player.
- The state (and turn owner) of the next state, after said elimination.

Now, with this information we need to build the transition matrix. We are not going to build it completely (it's huge, although it only has at most two elements per row, so it could fit in memory; it's just... not needed), and for that we need *only* the information about the next states. Let X_j be a state with N players, $\{P_i\}$, and the turn owner being some player P_j. If this player wins, they chooses some unfortunate loser P_k and the state moves to some state Y_j, which we have already determined during the previous calculations. But if P_j doesn't win its turn, the current state preserves the current amount of players and moves to the next turn owner in this set, X_{j+1}. So there are just two possible transitions: X_j to Y_j with probability S_j, and X_j to X_j to X_{j+1} with probability $1-S_j$. Easy. Once again, notice how the final probabilities of winning are not used here; they were useful only to determine the transition target.

Ok, so we have a (theoretical, for now) transition matrix M, and if we remove the absorbing states, we have a submatrix Q of more than 10^5 states (*note: some of the possible combinations never happen. This eliminates less than* 10 % *of the total, so the remaining set is still huge*), and Markov chain theory states that we need to do the following steps:

- Calculate the matrix $M = (I - Q)^{-1}$.
- Given M, sum all the elements of the row corresponding to the initial state (first row?). This is the solution of the problem.

The second step sure sounds easy. The first one is absolutely, 100 %, not directly workable. Even if I wanted to spend the time to do something of the order of 10^{15} operations, the required size is about 10^{10} and I don't think I can fit that much in memory (note that these are doubles. And no, the inverse of a sparse matrix is not necessarily sparse as well).

I mentioned that I (wrongly) believed the first part to be the difficult one, but here we are, needing to invert a huge matrix. We can't do that, at least not directly, so we need to find a way to calculate all the elements from the row we need without doing a full inversion. We need to use as much information as possible, and this requires some ingenuity. Let's see:

- We only need the first row of the inverted matrix, yes. Unfortunately, getting the first row of a matrix inverse is still an $O(n^3)$ operation, not $O(n^2)$ as someone could naively think. However, this suggests using the transpose (the inverse of the transpose is the transpose of the inverse!) and doing some kind of row elimination with one full matrix and just one additional vector. We can call this vector v.

- The matrix is very sparse (the inverse will NOT be sparse, though). This suggests that whichever clever scheme we could concot should iterate over some small set of values, not the whole matrix (nor a whole submatrix. At least, not always). This is indeed what we will do.

- The matrix is not tridiagonal, or upper triangular, or any shit like that. But it has some structure, since the transition only happens from certain states to other "close" ones. This is the key of this problem.

Yes. Structure. Use the matrix structure. But which structure is this? First of all, let's see. States with N players can only transition to states with N or $N-1$ players. This creates a "blocky" structure of sorts, where the state space is divided in blocks with $14, 13, 12, \ldots, 2$ players. The only blocks that have nonzero data (and remember that we are using the transpose matrix, as per the first bullet point above) are either in the "main diagonal" blocks corresponding to both rows and columns of i players, or in the "lower diagonal" blocks where the row is for i players but the column is for $i-1$ players. This is starting to get promising, but it's not enough. One other conclusion of the first bullet point is that we are going to use just a vector v as the placeholder for the first row of the inverse, right? And we will use elimination at the same time as in the main matrix, so that it's updated properly. Hmmm. By the way: as per the usual inverting method, this corresponds to the first column of the transpose inverse, therefore it starts with all its elements equal to 0, save for the very first one, which is equal to 1. So, let's start from the first, obvious case: the first initial set of states, the 14 states with $N = 14$, can be inverted, and multiplying the inverse times the first 14 elements of the "transpose inverse", we are de facto applying the row elimination we want, so that we will automatically get the correct values for these first 14 elements of v. Great! Now, uh, 114660 to go. Ok, but now that

we have "inverted" the first 14 rows of the matrix, we have an identity, and we can easily "drop" these values to the block of 13 players. Let's say that the matrix we are inverting, $R = I - Q^T$, has an element $R_{ij} = \alpha$, where the row i is at the $N = 13$ block, and the column j is at the $N = 14$ block. Gaussian elimination: we subtract the row j multiplied times α to the row i, and this includes the vector we're working on... and after doing this for all the values in this $(N = 13) \times (N = 14)$ subblock, the rest of the rows corresponding to $N = 13$ are empty save for the main block... so we can invert this block, and multiply the inverse times the working vector for values $N = 13$! And we can cascade this way of operating to lower values of N, until we get to $N = 2$! Yes, this looks like a huge improvement, but it's far from enough. Some of the sub-matrices are still huge. For example, we have one of size $7 \cdot \binom{14}{7} = 24024$. Oops. We have advanced a lot, but this is not enough.

Let's continue digging. Structure, let's look at the structure. For each set of N players, the transitions go to either one of the other N states with the exact same subset of N players, or to a state with $N - 1$ players. There are no other interactions between states with the same amount of players. So, let's use a much more fine tuned algorithm! We can use chunks of just N players! And since N doesn't grow bigger than 14, these are very easily manageable! Yes! We finally have a workable scheme. This is the part that I got, miraculously, right at the first try. There are many small steps, but the end result is not so complex, actually!

- We start by assuming that we have all the "chosen losers" calculated. This means that we can calculate the probabilities of each transition.
- Now, we build a structure that is close to the one we had, but is kind of reversed. We will have one state per set of players (no specific turn substates here), and these states will be indexed first by amount of players and then by set of players. For each one of these states, we build a set of transitions for which the current state is the *target*. Each transition will indicate the source state, the source player (i.e. the owner of the previous turn) and the target player (i.e. the owner of the current turn). Each one of these states will hold a vector of size N, i.e. the amount of players in this set. This structure can be built easily by iterating on the data collected during the initial phase of the problem.
- Let $v^{(X)}$ be the portion of the vector corresponding to each state X. Before building it, we first need to build the submatrix corresponding

to R for the main diagonal of this state, which looks like this:

$$R_X = \begin{pmatrix} 1 & 0 & 0 & \dots & S_n - 1 \\ S_1 - 1 & 1 & 0 & \dots & 0 \\ 0 & S_2 - 1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}.$$

This matrix is very small (14×14 at most) so inverting it is quick. Now, we need to take care of the transitions. By the way, Matlab implies that the inverted matrix looks like this:

$$R_X^{-1} = \frac{1}{1 + \prod_{i=1}^{n} (S_i - 1)} \begin{pmatrix} 1 & \prod_{i=2}^{n}(1-S_i) & \prod_{i=3}^{n}(1-S_i) & \dots & 1-S_n \\ 1-S_1 & 1 & (1-S_1)\prod_{i=3}^{n}(1-S_i) & \dots & (1-S_1)(1-S_n) \\ (1-S_1)(1-S_2) & 1-S_2 & 1 & \dots & (1-S_1)(1-S_2)(1-S_n) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \prod_{i=1}^{n-1}(1-S_i) & \prod_{i=2}^{n-1}(1-S_i) & \prod_{i=3}^{n-1}(1-S_i) & \dots & 1 \end{pmatrix}.$$

I guess I could generate this structure, but doing that is still a $O(n^3)$ operation, so I won't bother. Creating the inverse explicitly with a function call is clearer and the speed difference, although present, is tiny.

- We are going to use the inverted submatrix R_X^{-1}, but first we will use a "base" vector w representing the base transitions. In the special case for the very first state, we will use a vector with all zeros except the first element, which is 1 (this is because we are doing elimination on the first column of an identity matrix. Just as in the usual matrix inversion procedure). But for the other states, we will proceed as follows. Initialise w as a vector of all zeros. Now, we had a some transitions from higher states, right? Each transition has some probability S_i (since it comes from a turn someone has won), but this needs to be negated... but we will need to *subtract* this negated value! So let's save on a double negation and store the probability directly, and add it as it is. So, if the transition came from state Y, player i, to this state, player j, we update our j-th element by adding the probability S_i multiplied times the i-th element of state Y's vector: $w_j = w_j + S_i v_i^{(Y)}$. It might sound a bit confusing, but this is just Gaussian elimination. For the sake of

completion we can define w in a slightly more rigorous way:

$$w_j = \sum_{\{Y\}} \sum_{\left\{i_{Y,j}\right\}} S_i v^{(Y)}_{i_{Y,j}},$$

with $\{Y\}$ being the set of states that transition into the current state X, and $i_{Y,j}$ being the set of players from Y that transition into state X and turn owner j. These sets are overwhelmingly empty, of course, and some of them will have size 1. Theoretically some could also have size 2. I have not verified whether this happens.

- These calculations allow getting some vector w which is not what we need. We now make use of the inverted submatrix of the state, so that $v^{(X)} = R_X^{-1} w$. And now, $v^{(X)}$ is the final, stored vector for this state X.

- The final result of the problem comes from the addition of all the values of $v^{(X)}$ from all the states of all sizes. This can be done just as we iterate over all the states. We will also need to store $v^{(X)}$ so that it can be used by the next states (does this count as dynamic programming also?).

And that's it! That sum of all values is the number we want. The code is considerably complicated (about 300 lines of Java code, wow. Usually the number is something at the smaller end of the $[100, 200]$ range, or even smaller), but the loops have few iterations, so the run time is about 1,6 seconds. Not bad.

The first part of the problem has an approximate order of $O\left(2^N N^7\right)$ (inverting matrices of size N^2 to solve the equation systems) and the second part has $O\left(2^N N^4\right)$ (inverting matrices of size N). I guess that the first part can be optimised so that everything is $O\left(2^N N^4\right)$.

483. Repeated permutation

Difficulty rating: 100 %.
Solution: 4.993401567e22. *Sun, 6 Mar 2022, 07:50.*
Math knowledge used: group theory, combinatorics.
Programming techniques used: dynamic programming.

Don't be fooled. The amount of group theory needed for this problem is actually pretty small. The meat of this problem is in the permutation generation (I came up with a very useful trick that helps immensely, and which I think I'm going to be able to use in other problems where I want to generate permutations with dynamic programming without repetitions),

and in the data structures. I'm pretty sure that my structure was suboptimal (some people in the problem thread skip some of my levels of indexing, which results in a probably much more manageable memory usage and run time). My run time is not that bad, though, so I'm still happy with the results.

Ok. Let's go for the main mathematical part of the problem. Each permutation of order n is composed of many primitive subpermutations of order $l_1, l_2, \ldots, l_k$, where each $l_i \geq 1$ and $\sum l_i = n$. For such a partition, the order of the permutation is $f(P) = \operatorname{lcm}\{l_1, l_2, \ldots, l_k\}$. Another useful piece of information related to this is that, given a segment of n elements, there are exactly $(n-1)!$ primitive permutations of order n (there might be other non-primitive ones with the same order, but we don't want to count those). And this paragraph contains the totality of the group theory knowledge required for this problem. The rest is algorithmics.

Trying to enumerate partitions is a no-go, since the amount of partitions of order n is, according to Mathematica, 279363328483702152. I'm not going to enumerate these, naturally. I need a way to group permutations, possibly by value. Another problem is that the amount of possible values is still very high (although it does fit in memory), and trying to count partitions given its value is also much more difficult than it sounds. So the best bet is to try to build each new pack of solutions from the already existing ones.

Let's stop for a bit and try to calculate the contribution of each kind of permutation to the final sum. Let's say that we want to count permutations of order n, so that the subpermutations have orders $l_1, l_2, \ldots, l_k$. We can group the lengths, and say that we have n_1 permutations of length l_1, n_2 of length l_2, and so on. These numbers will verify that

$$\sum_i l_i n_i = n.$$

The "value" of these permutations is $\operatorname{lcm}\{l_i\}$. And we also want to calculate in how many we can distribute n elements in such amount of combinations. This can be done using combinatorics, and the result is this:

$$\frac{n!}{\prod_i (l_i!)^{n_i} n_i!}.$$

This $n!$ is very interesting, because we know that we will be able to cancel it with the final denominator of the average calculation. Also, since each cycle of length l_i has $(l_i - 1)!$ cases, the full formula for $g(n)$ is:

$$g(n) = \frac{1}{n!} \sum_{P_i} \left(\operatorname{lcm}\{l_i\}\right)^2 \cdot \frac{n!}{\prod_i (l_i!)^{n_i} n_i!} \cdot \prod_i \left((l_i - 1)!\right)^{n_i}.$$

We can do some simplifications and arrive at a final, manageable formula:

$$g(n) = \sum_{P_i} \frac{(\text{lcm}\{l_i\})^2}{\prod_i l_i^{n_i} n_i!}.$$

Now, before defining a structure to fill the data we need, here comes the interesting trick, which somehow I never realised until now despite having solved many problems about permutations where this problematic arises. In order to generate these groupings of permutations, I can generate each permutation once and only once if I iterate over all the existing permutations, but only add new elements when the element I'm going to add has a size equal or greater to the biggest element already present in the permutation. This is incredibly useful and I believe that it's the real key to solving this problem.

So, this suggests using a key equal to the maximum element l_i present in the permutation. Now, how do I update the value? Let's say that I have a permutation stored somewhere in the structure, and since the lcm is a key, I'm storing only the denominator,

$$\prod \frac{1}{l_i^{n_i} n_i!}.$$

I mentioned that I'm only adding elements that are equal or greater than the biggest element. Well, this is a bit of a problem. If the new element l is greater than all the existing ones, I just need to add a factor $\frac{1}{1!l} = \frac{1}{l}$. However, if it's equal, I need to add a factor $\frac{1}{l(n+1)}$ to transform $n_i!$ into $(n_i+1)!$. This means that I need yet another key, with the exact amount of times the maximum element appears. The good news is that this is the last key we need to add to our structure, and all the permutations that share the same set of keys can be lumped into a same sum.

So first we are going to have an array with indices from 1 to N. Inside each of these arrays, say of index n, we are going to have another array with index from 1 to n, indicating the value of the biggest element in the permutation. And inside these arrays, at each element m, we will have arrays indexed from 1 to $\left\lfloor \frac{n}{m} \right\rfloor$. At each one of these elements we will have a map from long to double, where the key is the lcm of the elements of the permutations, and the value is

$$\sum_{P_i} \left(\prod_i \frac{1}{l_i^{n_i} n_i!} \right)$$

for all the permutation types that fill the description. The update process follows this algorithm:

- Iterate over all $n \leq N$. For each n, iterate over all $m \leq n$.
- Let $\Delta = n - m$ be the element we are going to add to the existing permutations. Iterate over permutations of order m where the maximum element is $e \leq \Delta$.
- For each permutation with lcm equal to l and value equal to v, the new lcm will be $l' = \operatorname{lcm}(l, \Delta)$, and the new value will be calculated as $v' = \dfrac{v}{\Delta}$ if $e < \Delta$, or as $v' = \dfrac{v}{\Delta\,(k+1)}$ if $e = \Delta$ and the amount of times e appears is k.
- The resulting pair $\{l', v'\}$ will be stored in the appropriate map: the total permutation size is n, the maximum element is Δ, and the amount of repeated times is 1 if $e < \Delta$ or $k + 1$ if $e = \Delta$. If the map already has a key with this mapping, we can add v' to the existing value.

This looks good and it does give the correct result for smaller values. There are, however, two issues.

- The memory usage is huge. Even 60 gigabytes might not be enough (LongDoubleMaps are not very memory efficient, unfortunately). Solution: don't generate values we are not going to need. If the maximum value is N, for values higher than $\frac{N}{2}$ we only need to generate the permutations where the maximum element e is below or equal $N - n$.
- There are overflows! Some of the lcms don't fit in a long. Fortunately, thanks to the big prune from the above paragraph, this only happens for $N = 350$, and not before. In this case, it so happens that these values are not going to be used as argument in the lcm calculations, so we can just use doubles. So we can calculate all the values on the go! As a bonus, we are not storing the temporary values, just adding them to a big counter, so the result is found faster.

And now we have the final algorithm that solves the problem:

- For $n = 1$ to 349, update the counters using the formulas above. At each n we only need the cases where the maximum length of subpermutations is $350 - n$, which means that for bigger numbers the calculations will be faster. The slowest calculations happen somewhere between $n = 270$ and $n = 290$.

- For $n = 350$, we will calculate the result directly. For each $m \in [1, 349]$, and for each maximum element $\Delta \in [1, 350 - m]$, gather all the relevant permutations.
- For each permutation, calculate the lcm l and the sum of values v, considering already the addition of Δ. Add $l^2 v$ to the total.

The sum of all the $l^2 v$ is the result of the problem. It takes about 166 seconds of run time and more than 50 gigabytes of memory, but nobody said that 100 % problems would be easy! Speaking of which, I found this a bit harder than 484, but still not as hard as other beasts like 769 or 261. Even so, we can go a couple problems back and I would say that 481 was harder than this. It's subjective, I guess. Then again, I think that this problem would have been impossible with the resources of my previous PC, so I guess that much more additional finesse would be needed to do this in an older computer (the problem was released in 2014, after all).

I now have two 100 % problems, but not any 95 % problem yet. I'm not sure if I'm ready to tackle those yet.

484. Arithmetic Derivative

Difficulty rating: 100 %.
Solution: 8907904768686152599. *Solved: Fri, 4 Mar 2022, 07:51.*
Math knowledge used: multiplicative functions, inclusion-exclusion.
Programming techniques used: Erathostenes sieve.

I managed to solve a 100 % problem and I still can't believe it. The run time is horribly slow, though (about 1111 seconds on my last successful run), and I strongly believe that this is caused by the abundance of TreeSet operations. Further experiments with LongSet reduced the time to almost exactly one fourth of that (278 seconds), which is much better; I could achieve better times by removing certain recalculations, but that would require much more memory and the gains wouldn't be so great.

To be honest I've solved problems that felt much harder, but not even 261 or 769 were rated 100 %. The inclusion-exclusion for this problem is maybe trickier than usual, but honestly it's not that complex. The problem is hard anyway and I'm happy to have been able to solve it. Another curious part is that this problem has little pen and paper. I could manage everything in my head, because there wasn't any long chain of derivations, nor any kind of symbol pushing of any kind, really. Once you see the formula, it's just a matter of devising a scheme to avoid recounting, which is tricky but doable.

I'm not sure if this kind of scheme can be used in other similar problems, but so far none comes to mind.

Now let's talk about the problem itself, and its solution. It's easy to brute force the first values, or directly search OEIS, and the sequence A003415 will appear. On the related sequences one can find A085731, which is the function we're looking for: $f(n) = \gcd(n, n')$. And this sequence comes with the formula we're going to work with. $f(n)$ is multiplicative, which means that we can calculate its value given the prime decomposition of n, and for each prime power p^e we have:

$$f(p^e) = \begin{cases} p^e & \text{if } e \equiv 0 \pmod{p}, \\ p^{e-1} & \text{otherwise} \end{cases}$$

This is very interesting, but it's irregular enough to be annoying. The multiplicative function summation trick from problem 639 is not really going to help, because there isn't an easy expression for the sum of $f(n)$ in any interval. A Lucy Hedgehog sieve isn't going to help either, because the function is not completely multiplicative (and we would also need an expression for the sum in a given interval, which is not really feasible since the expression is not Faulhaber formula friendly). But there is a wonderful property that is the real key to solving this problem in a reasonable amount of time: every single squarefree number n verifies $f(n) = 1$ (save for $n = 1$, a special case that we don't need to cover), no matter whether it has one, two or a million prime factors. This means that we can use only powerful numbers and its multiples.

So, first experiment: how many powerful numbers are below the limit? The limit happens to be very high ($5 \cdot 10^{15}$), which is part of why this problem is so intimidating, but a quick experiment reveals that the amount of powerful numbers is just above $1{,}5 \cdot 10^8$, which is very reasonable. Furthermore, they can be generated in relatively little time. So let's think about how to count effective values. Let's say that we have a powerful number like $n = 72 = 2^3 \cdot 3^2$. The formula says that $f(n) = 2^2 \cdot 3 = 12$. Ok. This means that there is a family of numbers of the form $m = k \cdot 72$ (where k is squarefree, and also not a multiple of 2 or 3), so that $f(m) = 12$ for all of them. Neat, but the problem is that counting values in this family is not so straightforward. My method of summation has two parts:

- Calculate the succession of values n such that its prime decomposition includes 2^3 (but not 2^k for any other power k) AND 3^2 (but not 3^k for any other power k).

- Subtract values that take part in the aforementioned succession, but for which $f(n) \neq f(72)$, such as $1800 = 72 \cdot 25$ and its multiples.

Thus we have two kinds of inclusion-exclusion. First we want to "isolate" powers of primes, and then we want so subtract "misfit" numbers with additional prime powers. It turns out that this is not so difficult to do, it just needs some care in the operation order.

I used two simple data structures. The first one is called `PrimePowerData` and it's just a tuple with three values:

- The number itself, which is always a prime power p^e where $e \geq 2$. It's used as a source for the succession $\{p^e, 2p^e, 3p^e \ldots\}$, most (but not all) whose elements share a common value of $f(n)$.

- The next power, p^{e+1}. This is useful because it signals values that we need to *subtract*, since it refers to the succession $\{p^{e+1}, 2p^{e+1}, 3p^{e+1} \ldots\}$, which signals one predictable (and linearly spaced) set of values for which $f(n)$ does not equal the same values as in the previous case. If p^{e+1} exceeds the limit $5 \cdot 10^{15}$, I just store a -1 to indicate that I don't need to subtract anything.

- I also store the value of $f(p^e)$, calculated quickly with the known formula.

The second data structure, `FullNumberData`, is built on the previous one. It represents any powerful number, and it has two fields:

- First it has an array with all the `PrimePowerData` values corresponding to its prime decomposition. So the object representing 1800 would have three objects: one for 2^3, another one for 3^2 and a last one for 5^2.

- It also has a *mutable* counter indicating the amount of numbers already counted. It's simple: when we calculate the value for 1800, we calculate the amount of elements for which $f(n) = f(1800)$, and we add this amount to the counters of all its divisors, which are the other 7 combinations of 2^3, 3^2 and 5^2 (this includes the empty case $n = 1$!). This suggests iterating in descending order, so that when we arrive at a number n, all its multiples have been passed over already.

The scheme becomes clearer now. The full algorithm is as follows.

- First of all, generate all the primes below $\sqrt{N}$ where $N = 5 \cdot 10^{15}$ is the problem limit.

- Now, use these primes to generate `PrimePowerData` objects. For each prime p, we want to generate an object for each power p^e between $e = 2$ and $e = \lfloor \log_p N \rfloor$. There are about four million of these IIRC, and they can be generated quickly enough.

- Now, generate all the powerful numbers, as `FullNumberData` objects. Include an element for $n = 1$. This can be done with a not so deep recursive algorithm (no more than 8 primes are ever needed, I believe), and takes a bit of time, about one minute or maybe a bit less.

- We will need to access the `FullNumberData` directly by index, but we also want to iterate over them in decreasing order. It seems like a `TreeSet` is a good solution, but it's far too slow. A much better idea is to store the objects in a `LongObjMap`, and then use the keys of this map to generate an array and then sort it. I've experimentally found that sorting longs is far faster than sorting objects even if they are sorted by a long value (on the order of 9 or 10 times faster!), so I store just the longs, and I use the map for queries.

- Now, the meat of the algorithm. We are going to iterate over the *descending* order (easy enough with the array of longs). For each number n we will do the following:

  - Retrieve the associated `FullNumberData` with all the relevant prime powers.
  - The prime powers data can be used as a source of inclusion-exclusion in order to count the amount of elements. Following the example of 72, we will need to add multiples of 72, but will also subtract multiples of 144 and multiples of 216. And then we will need to add multiples of 432, which we have subtracted twice! This is just a typical inclusion-exclusion lattice, but easier since there aren't "squares", nor any combinatoric magic, so each product is either added or subtracted, no zeroes or multiples. If n is the product of k elements, there are at most 2^k terms, and since $k \leq 8$ (and it's much lower, being just 3 or less in the vast majority of cases), this is much faster than sounds. Again with the example of 72, we calculate the amount of elements as $x = \left\lfloor \frac{N}{72} \right\rfloor - \left\lfloor \frac{N}{144} \right\rfloor - \left\lfloor \frac{N}{216} \right\rfloor + \left\lfloor \frac{N}{432} \right\rfloor$.
  - However, chances are that we have already counted some values that are multiples of n. Let's say that the counter indicates a value

of y. Then, the actual amount of terms m for which $f(m) = f(n)$ is exactly $z = x - y$.

- We can calculate $f(n)$ easily with the `FullNumberData` information. So we add $z \cdot f(n)$ to the final result.
- And now we calculate all the powerful divisors of n. Again, if n is the product of k prime powers, there will be 2^k elements, including 1 and n itself. For each one of these values, update the "already counted" counter, adding z to it.

- That's it. The sum of all the $z \cdot f(n)$ gives the result. We only need to subtract 1 since we mustn't count $f(1)$ (also, it's technically wrong since the arithmetic derivative of 1 is not 1 but 0). But aside from that, the sum is the final result.

And that's everything. It's complicated, because it wouldn't be a 100 % problem otherwise, but still, it's probably the easiest 100 % problem out there. And I'm still proud of having been able to solve it in a reasonable amount of time (both in terms of thinking on the problem, which took me about three days, and getting the result, which takes a bit more than four minutes and a half).

492. Exploding sequence

Difficulty rating: 60 %.
Solution: 242586962923928. *Solved: Sun, 26 Dec 2021, 19:58.*
Math knowledge used: quadratic maps, cyclic groups.
Programming techniques used: dynamic programming.

This problem is hard and it includes a lot of comparatively advanced math. In particular, the part about quadratic maps is kind of obscure (I wish I had studied this in my subject about chaos theory, but no, it's not covered). First of all: looking for cycles will not work in a reasonable time. It's in practice an $O(mn)$ operation where there are $m \approx 10^6$ numbers and cycles can be as long as $n \approx 10^9$. As expected, we need to be smart.

So the first thing we need to do is look for a non-recursive formula that gets the value of a_n for any given n. Not every quadratic sequence allows this, but this one is, of course, carefully created so that it does. Now, given the recursive formula

$$a_{n+1} = 6a_n^2 + 10a_n + 3,$$

we want to transform it into a simpler one, which we will do by defining $b_n = 6a_n + 5$ (note the square completion of sorts), which after some simplification

takes us to

$$b_{n+1} = b_n^2 - 2.$$

This is awfully convenient because the only quadratic maps of the form $x_{n+1} = x_n + a$ which have an explicit solution are the cases $a = 0$ and $a = -2$ (with $x_0 > 2$), and we are in the latter one. And in this fortunate case, the solution is

$$x_n = A^{(2^n)} + B^{(2^n)},$$

where $A = \dfrac{x_0 + \sqrt{x^2 - 4}}{2}$ and $B = \dfrac{x_0 - \sqrt{x^2 - 4}}{2}$ are the solutions of the equation $z^2 - x_0 z + 1 = 0$. We still have several problems now. The first, stupid one, is that we need indices to start in 0 instead of 1. Ok, sure. We define $b_n = 6a_{n+1} + 5$, so that $b_0 = 11$ is the first term. Now, the second one is that these numbers are not integers. In particular, we have $A = \dfrac{11 + 3\sqrt{13}}{2}$ and $B = \dfrac{11 - 3\sqrt{13}}{2}$. Now, we know that the final result will be integers, but we can't work with these irrational values so easily. We solve this by using group theory, working on the $\mathbb{Z}\left[\sqrt{13}\right]$ group. Let (a, b) be a member of this group, representing the value $a + b\sqrt{13}$. We can calculate a product using

$$(a_1, b_1) \cdot (a_2, b_2) = (a_1 a_2 + 13 b_1 b_2, a_1 b_2 + a_2 b_1).$$

So far so good: we can define $A' = (11, 3)$ and $B' = (11, -3)$, so that each number can be calculated as

$$b_n = \frac{A'^{2^n} + B'^{2^n}}{2^{2^n}}.$$

We know that the numerator, although in $\mathbb{Z}\left[\sqrt{13}\right]$, will have its irrational part equal to 0. Here comes the next problem: the obvious approach is using binary exponentiation, but when the exponent is 2^n, this is linear in n. Which is not good when $n = 10^{15}$ (well, $10^{15} - 1$, actually. Hooray?). And now we need to do some group theory magic. We need to work in the realm of numbers modulo P for some prime P around 10^9. For the numerator we know that 2^{2^n} will be congruent with $2^{2^n \bmod P-1}$, so the binary exponentiation is reduced to order $O(\log P)$, but what do we do for the numerators? Well, binary exponentiation will work as well, but now we are dealing with a group with P^2 elements, so excluding the zero we get cycles of length $P^2 - 1$. Then, for the numerators we will use that, for every $x \in \mathbb{Z}\left[\sqrt{13}\right]$ and every $n \in \mathbb{Z}$, x^{2^n} will be congruent with $x^{2^n \bmod P^2-1}$, and so we will only need to do binary exponentiation of the order $O(\log P^2) = O(\log P)$ as well. Now, a

very important detail: since $P \approx 10^9$, we need a modulo $P^2 - 1 \approx 10^{18}$, and we will need BigIntegers to calculate it. Fortunately, this is everything we need to calculate the number we are being asked for. The algorithm would be:

- Let $N = 10^{15} - 1$.
- Iterate over each prime p in the set $[10^9, 10^9 + 10^7]$, as asked by the problem. For each prime:
- Calculate the exponents $e_1 = 2^N \bmod P^2 - 1$ and $e_2 = 2^N \bmod P - 1$ with binary exponentiation, using BigInteger for the first one.
- Use binary exponentiation to calculate $n_1 = A'^{e_1}$, $n_2 = B'^{e_1}$ and $n_3 = (2^{-1})^{e_2}$, always modulo P. Conveniently, for any prime $P > 2$ we have $2^{-1} = \frac{P+1}{2}$.
- Get the value b_N as $(n_1 + n_2) \cdot n_3$. Guaranteed to be an integer; discard the $\sqrt{13}$ part, which will be zero anyway.
- Now, get $a_{10^{15}}$ as $6^{-1} \cdot (b_{10^{15}-1} - 5)$. Again, always modulo P.
- Finally sum all the values for every required prime.

The result takes less than 8 seconds on my machine. I'm assuming that slower machines will still take less than a minute.

539. Odd elimination

Difficulty rating: 35 %.
Solution: 426334056. *Solved: Fri, 19 Nov 2021, 05:52.*
Math knowledge used: none.
Programming techniques used: dynamic programming.

Almost all there is to this problem is the analysis of the brute force results, and the careful creation of a recursive function to calculate the sum using blocks, just like many other problems like 325 or 769. The value of $P(n)$ can be found for each n in a time $O(\log n)$, so for smaller values of n we can just brute force through $S(n)$. I guess that there are patterns for $S(n)$, but I went for the pattern in $P(n)$ and found a way to get $S(n)$ from that. The final algorithm, as expected, is logarithmic. It's funny, how in these problems there aren't workable middle points: either you go for the fully linear or log-linear brute force (which is not a good idea for $N = 10^{18}$), or you use a logarithmic algorithm and get the result in milliseconds.

First of all, let's see how to calculate $P(n)$. To find it, we use an iterative algorithm where, at each step, we have m values remaining, which are of the form $ak + b$ for $k \in \{1 \dots m\}$. We also store a boolean flag indicating whether we must reduce from the left or from the right. We start with $a = 1$, $b = 0$ and $m = n$ (so we have the succession of natural numbers), and the reduction flag set to "left". Then, for each step, we calculate a new boolean flag, "removeOdds", which is set to true if the reduction flag is set to "left" *or* if m is an odd number. Then we update the flags like this:

$$\begin{aligned} a_{i+1} &= 2a_i, \\ b_{i+1} &= \text{if (removeOdds) then } b \text{ else } b - a, \\ m_{i+1} &= \left\lfloor \frac{m_i}{2} \right\rfloor, \\ \text{reduceLeft}_{i+1} &= \text{NOT reduceLeft}_i. \end{aligned}$$

We end this process when $m = 1$, and the result is $P(n) = a_i + b_i$ (that is: the first, and only, term of the succesion $\{a_i k + b_i\}$).

Now we want to analyse the pattern. This is where the bulk of the problem goes. Some people in the problem thread have found different patterns, but this is the one I found.

- There is a special case, $P(1) = 1$. This is, trivially, the only odd value in all the succession.
- Besides this special case, we consider ranges defined by powers of 2. Namely, we consider ranges of the form $[2^{2n+1}, 2^{2n+3})$, starting from $n = 0$, so each range has $3 \cdot 2^{2n+1}$ elements.
- Inside each of these ranges, we have two parts. First we have four groups, each of length 2^{2n}, which have the exact same values. That is, the first 2^{2n+2} terms of the range consist of a succession of the same 2^{2n} elements, repeated four times. We denote this sequence G_n.
- After that, the remaining terms, which I called the "transition group", are again two equal sequences of length 2^{2n}. These sequences are different to the previous ones. We denote each sequence T_n. So, the values in the range $[2^{2n+1}, 2^{2n+3})$ come from a concatenation of these sequences: $\{G_n; G_n; G_n; G_n; T_n; T_n\}$.
- For $n = 0$, the main group has $2^{2 \cdot 0} = 1$ element, which is 2. So we say that the main group is $G_0 = \{2\}$. This is the starting point of the recursion.

- Given a sequence G_n, we calculate T_n by adding 2^{2n+1} to each element of G_n: $T_n = G_n \oplus 2^{2n+1}$. So $T_n = \{4\}$.
- Also, given a sequence G_n, we create the sequence G_{n+1} by concatenating G_n four times and adding values based on the powers of 2. The formula is:

  $$G_{n+1} = \left\{G_n \oplus 2 \cdot 2^{2n+1}; G_n \oplus 2 \cdot 2^{2n+1}; G_n \oplus 3 \cdot 2^{2n+1}; G_n \oplus 3 \cdot 2^{2n+1}\right\}.$$

  For example, $G_1 = \{6, 6, 8, 8\}$. This G_{n+1} sequence is repeated four times inside the range for $n + 1$, just like G_n was in the case n.

This is enough information to build a recursive, block-based scheme that can calculate $S(N)$ using $\lceil \log_4 2N \rceil$ blocks. We will have blocks for $n = 0, 1 \ldots$, and for each one we will store:

- The range of the "normal" blocks, $[2^{2n+1}, 3 \cdot 2^{2n+1})$.
- The range of the "transition" blocks, $[3 \cdot 2^{2n+1}, 2^{2n+3})$.
- The size of G_n, which is $A_n = 2^{2n}$.
- The sum of all values in G_n, $H_n = \sum G_n$.
- The total sum of all values in the range $[2^{2n+1}, 2^{2n+3})$, J_n.

We start with $H_0 = 2$, since $G_0 = \{2\}$. We can also calculate immediately that $J_0 = 4 \cdot 2 + 2 \cdot 4 = 16$. The values of H_n and J_n can be calculated recursively like this:

$$\begin{aligned} H_n &= 4H_{n-1} + 10 \cdot 2^{2n+1}; \\ J_n &= 6H_n + 2^{2n+2} A_n. \end{aligned}$$

And with these blocks in place we can calculate $S(N)$. First of all, we can calculate the sum of the "full" blocks, that is, those for which the range falls below N, just adding all the values of J_n. We must add 1 as well to account for the special case $P(1) = 1$, which is not covered by the blocks. To get the remaining result, we first calculate a diff value, $D = N - 2^{2n+1}$, with the subtrahend being the low end of the current range (this means that there are $D + 1$, not D, elements to sum!). Then, we will use four functions that call themselves recursively to calculate the sum of $P(n)$ in the range not covered by $\sum T_n$. These functions are: the sum inside the transition block, $t_n(x)$; the sum inside a group, $g_n(x)$; the sum inside a four-groups block, $b_n(x)$, and the total sum inside $[2^{2n+1}, 2^{2n+3})$, $s_n(x)$. The formulas vary in complexity.

- $t_n(x)$ can be calculated as $t_n(x) = 2^{2n+1}(x+1) + b_n(x)$, since it's just a shift over the "main" groups. This is the simplest of them all.

- For $g_n(x)$ we will use a recursion based on the functions for $n-1$. First of all, we calculate a shift; for this, we divide $x+1$ in two parts: if $x > 2^{2n-1}$, we call $a_2 = 2^{2n-1}$, $a_3 = x + 1 - a_2$. Otherwise, $a_2 = x+1$, $a_3 = 0$. The shift is $s = 2^{2n-1}(2a_2 + 3a_3)$.

  We then divide x by the *previous* group length, A_{n-1}, and we store the quotient q and the remainder r. We define the previous sum p as: if $r = A_{n-1} + 1$, then $p = (q+1)H_{n-1}$. Otherwise we need a recursive call, $p = qH_{n-1} + g_{n-1}(r)$. Note that at some point we have $A_n = 1$ for $n = 0$ so the recursion will end eventually. In any case, the result is $g_n(x) = s + p$.

- The calculation of b_n looks a bit like g_n, but simpler. First, we divide x by the *current* group length, A_n, and we note the quotient q and the remainder r. Then, if $r = A_n - 1$, we use $b_n(x) = (q+1)H_n$; otherwise, $b_n(x) = qH_n + g_n(r)$.

- Finally, the total sum inside of this block depends on the value of x: if $x \geq 4A_n$, then $s_n(x) = 4H_n + t_n(x - 4A_n)$; otherwise, $s_n = b_n(x)$.

With all these functions in place, we call $s_n(D)$ and add the result to the sum of the T_n values prior to the relevant block. This is the solution of the problem. The full value for $N = 10^{18}$ is

$$271051139927967385167988795568221935,$$

which is, as expected, a number of $2 \cdot 18 = 36$ digits. The result is found in about 2 milliseconds.

541. Divisibility of Harmonic Number Denominators

Difficulty rating: 90 %.

Solution: 4580726482872451. *Solved: Sat, 8 Jan 2022, 00:26.*

Math knowledge used: p-adic numbers, *A p-adic Study of the Partial Sums of the Harmonic Series* (David W. Boyd), linear algebra.

Programming techniques used: none.

What a rush. I didn't expect to ever be able to solve a 90 % problem, but here I am. Also, I would have expected to solve 289 first, but that one

remains in the drawer for now, since I don't believe that my current approach is feasible.

I learned a lot about p-adic numbers with this problem (and with Boyd's paper); first and foremost, that p-adic numbers are not the same thing as "representation in base p", since in the former case p must be prime and the unusual definition of the norm results in a lot of unexpected behaviour.

I'm not sure about what to say that isn't explained on the paper. I wrote a lot of code just for the p-adic number library, which I might not ever use again, but at the very least it worked, and I got to solve a very difficult problem. Even if this is one of these "*oh, there's a paper for that*" problems, implementing it was challenging and I learned a lot, so I appreciate this even if I'm not sure if this is the kind of problems I like. The algorithm itself is not super-complicated, and the only really difficult part (the one that I couldn't have arrived at by myself) is the recursion to calculate H_{pn} from H_n. The rest, I could have managed even if I would have needed some brute force to find the approximate pattern. There is a good chance that there are small bugs here and there, since the algorithm that returns the correct result for $p = 137$ does not work for $p = 7$, at least with the parameters I used.

Long story short, there are three insights to be made. The first one is that working on a p-adic system instead of on rationals or reals is the way to go. Even if I didn't know about p-adic numbers, I would have found them eventually, probably. This problem is basically asking for them. The second insight, which is one I could have observed by brute force, is that the value we are looking form is of the form $(j+1)\,p-1$, where j is the last element of the set

$$J_p = \{n \in \mathbb{N} : v_p\,(H_n) > 0\}\,,$$

with H_n being the nth harmonic number and $v_p\,(x)$ being the smallest non-zero number on the p-adic representation of x, meaning that $v_p\,(H_n) > 0$ implies that the numerator of the real harmonic number is a multiple of p. Anyway, the set J_p can be built in a recursive way, because if $n \in J_p$, then $\left\lfloor \frac{n}{p} \right\rfloor \in J_p$ as well, so we can start from 0, and for each number n in the current generation, try all the values $pn + [0, p-1]$ and see which ones fulfill the condition.

The third insight, which is the part where advanced math comes in, is the way of calculating H_{pn} given H_n. For this, we need a theorem saying that there exist p-adic coefficients $\{c_k\}$ such that

$$H_{pn} = \frac{1}{p} H_n + \sum_{k=1}^{\infty} c_k p^{2k} n^{2k},$$

and these $\{c_k\}$ not only are independent from n (so we can calculate them from the first values, and then reuse for every computation afterwards) are such that $v_p\left(c_k p^{2k}\right)$ is increasing, therefore we can use just the only ones up to certain precision.

The algorithm is deceptively simple. First, decide a number N of terms for c_k and a precision $s > 2N$ for the p-adic computations. $N = 10$ and $s = 50$, which is actually pretty small, gave me the right result for $p = 137$ in 50 milliseconds. Using the chosen precision, calculate the first pN harmonic numbers. Then, build an equation system (which comes from a Vandermonde matrix, so it's regular. And yes, I implemented Gaussian elimination in p-adic numbers), and solve it to get $\left\{c_k p^{2k} : k \in [1, N]\right\}$. After that comes the iterative part. Start with a set $G_1 = \{n \in [1, p-1] : v_p(H_n) > 0\}$, and iterate. At each step, start with every value $n \in G_m$, and calculate H_{pn} using the equation with the coefficients $\{c_k\}$. Then, calculate all the numbers $\{H_{pn+k} : k \in [1, p-1]\}$ (just add the inverses normally, starting from H_{pn}), and store in the set G_{m+1} those ones for which $v_p(H_x) > 0$. When you find an empty G_{m+1}, take the highest value from G_m, n, and the solution to the problem is the number $(n+1)\,p - 1$. Otherwise, if G_{m+1} is not empty, just keep iterating. Any computation related to H_n or c_k must be done, of course, in p-adic representation.

570. Snowflakes

Difficulty rating: 55 %.

Solution: 271197444. *Solved: Wed, 5 Jan 2022, 17:40.*

Math knowledge used: recurrence equations, GCD properties, Chinese remainder theorem.

Programming techniques used: binary exponentiation.

I spent some time on this problem back in the day, and I more or less solved half of it. Then, four years and some months later, I solved almost all the remaining part, with a bruteforcish attempt, and the problem thread gave me the last detail to finally have a good solution.

Here's the first part. We start with the formalisation of the problem: the portion with an even amount of layers always has some weird approximately hexagonal shape, but the portions with an odd amount of layers is composed of triangles (all of the exact same size at each iteration). We can separate the triangles by the amount of layers of the triangle itself and the amount of layers of the area immediately next to each of its three sides. If we sort these three values, to normalise identifiers, we end with a set of successions of the form $\{t_{l;a,b,c}(n)\}$, where each of the $\{a, b, c\}$ are either $l+1$ or $l-1$.

Initially, for $n = 1$, we have $t_{1;0,0,0}(1) = 1$, and for all the others successions, we have $t_{l;a,b,c}(1) = 0$. Then, we can move *forward* to see which triangles are generated by each triangle, which can be done by studying the distribution. Each triangle at step n will generate 6 different triangles at step $n + 1$, following this pattern:

- One triangle $(1; 0, 0, 0)$ generates six triangles $(1; 0, 0, 2)$.
- One triangle $(1; 0, 0, 2)$ generates three $(1; 0, 0, 2)$, two $(1; 0, 2, 2)$ and one $(3; 2, 2, 2)$.
- One triangle $(1; 0, 2, 2)$ generates one $(1; 0, 0, 2)$, two $(1; 0, 2, 2)$, one $(1; 2, 2, 2)$ and two $(3; 2, 2, 2)$.
- One triangle $(1; 2, 2, 2)$ generates three $(1; 2, 2, 2)$ and three $(3; 2, 2, 2)$.
- One triangle $(3; 2, 2, 2)$ generates six $(3; 2, 2, 4)$.
- One triangle $(3; 2, 2, 4)$ generates three $(3; 2, 2, 4)$, two $(3; 2, 4, 4)$ and one $(5; 4, 4, 4)$.
- One triangle $(3; 2, 4, 4)$ generates one $(3; 2, 2, 4)$, two $(3; 2, 4, 4)$, one $(3; 4, 4, 4)$ and two $(5; 4, 4, 4)$.
- One triangle $(3; 4, 4, 4)$ generates three $(3; 4, 4, 4)$ and three $(5; 4, 4, 4)$.

Now, we can condense this information into eight actual succession formulas. In fact, we can skip the succession $t_{1;0,0,0}(n)$, which is equal to 1 for $n = 1$, and to 0 in every other case. So we get

$$\begin{aligned}
t_{1;0,0,2}(n+1) &= 6t_{1;0,0,0}(n) + 3t_{1;0,0,2}(n) + t_{1;0,2,2}(n); \\
t_{1;0,2,2}(n+1) &= 2t_{1;0,0,2}(n) + 2t_{1;0,2,2}(n); \\
t_{1;2,2,2}(n+1) &= t_{1;0,2,2}(n) + 3t_{1;2,2,2}(n); \\
t_{3;2,2,2}(n+1) &= t_{1;0,0,2}(n) + 2t_{1;0,2,2}(n) + 3t_{1;2,2,2}; \\
t_{3;2,2,4}(n+1) &= 6t_{3;2,2,2}(n) + 3t_{3;2,2,4}(n) + t_{3;2,4,4}(n); \\
t_{3;2,4,4}(n+1) &= 2t_{3;2,2,4}(n) + 2t_{3;2,4,4}(n); \\
t_{3;4,4,4}(n+1) &= t_{3;2,4,4}(n) + 3t_{3;4,4,4}(n).
\end{aligned}$$

And the successions presented in the problem are

$$A(n) = \sum_{a,b,c \in \{0,2\}} t_{1;a,b,c}(n);$$

$$B(n) = \sum_{a,b,c\in\{0,2\}} t_{3;a,b,c}(n).$$

After some work, which I don't quite remember because it was more than five years ago, but which could have been done using Mathematica, I arrived at the following formulas, valid at least for $n \geq 3$:

$$\begin{aligned} A(n) &= 6\left(2\cdot 4^{n-2} - 3^{n-2}\right); \\ B(n) &= 6\left((3n-23)\,4^{n-2} + (2n+13)\,3^{n-2}\right). \end{aligned}$$

Well, at the very least we know that every gcd is a multiple of 6. This is where I stopped during my first attempt, probably because my books for the fourth year of my maths degree arrived and these took priority (also because I realised that $A(n)$ had some periodic properties that $B(n)$ lacked). Fast forward until january 2022, and I retake this problem, retracing my steps and realising that I had some code which basically spelled these formulas above. I immediately realised that $G(n) = 6$ for most n, and the trick was to multiply times some primes here and there. This was not trivial, though, but the run time was somewhere between 30 and 40 minutes, which is decent for a brute force. This was not optimal, but anyway, here's how it went: I worked prime by prime, and for each prime p, I noticed that the values of n for which $A(n) \equiv 0 \pmod{p}$ were either non-existant or of the form $n = a + bk$ for every integer k, and for $b = \dfrac{p-1}{x}$ for some small x; in fact, $x = 1$ in the vast majority of cases. You know, usual modular arithmetic. Now, $B(n)$ was a much more irregular succession, and there are usually several subsuccessions that verify $B(n) \equiv 0 \pmod{p}$, so I started from $A(n)$: we are looking for values of n for which both successions vanish, so why not start by $A(n)$? If $A(n) = 0$, this means that $2\cdot 4^{n-2} - 3^{n-2} \equiv 0 \pmod{p}$, so if we define $x = 4^{n-2} \pmod{p}$, then $3^{n-2} \equiv 2x \pmod{p}$. Plugging these values on the $B(n)$ formula yields something very interesting:

$$B(n) = (3n-23)\,x + (2n+13)\cdot 2x = (7n+3)\,x.$$

Now, since $x \equiv 4^{n-2} \pmod{p}$, it's not possible that $x \equiv 0 \pmod{p}$. And, since p is prime, 0 has no divisors modulo p, therefore $B(n) \equiv 0$ is equivalent to $(7n+3) \equiv 0$. Now this is very useful, because once we have the succession of zeroes of $A(n)$ modulo p, $\{a + bk : k \in \mathbb{N}\}$, we can define the set of values where both $A(n)$ and $B(n)$ vanish by solving this diophantine equation system:

$$n \in \{a + bk : k \in \mathbb{N}\} \quad\Rightarrow\quad n \equiv a \pmod{b};$$

$$7n + 3 \equiv 0 \quad \Rightarrow \quad n \equiv 7^{-1} \cdot (-3) \pmod{p}.$$

The usual Chinese remainder theorem calculations can give us the proper solution, and since b divides $p - 1$, we know that there is a single solution of the form $n = n_0 + bpk$ for $k \in \mathbb{N}$. Also, 7 is one of the cases where $A(n) \equiv 0$ has no solution, so there is no degenerate or special solution. The problem with this scheme is that solving $A(n) \equiv 0$ is a discrete logarithm and I had no other means to solve than iterating until I found zeros or cycles, so this gave me a workable algorithm, but of order $O(N^2)$, not good when $N = 10^7$ even if the constant is very small (I insist, the result took well less than 40 minutes). I realised that I could prune the primes, and the last one is a bit below 1350000, so I used that as an upper bound for the primes and reduced the total run time by a bit. Of course, this is basically cheating since I could only do that because I had found the answer beforehand, but in any case, this was my best solution. There is the assumption that squared primes don't appear, which is reasonable, and it happens to be correct (they would appear at cycles of length bp as divisors of $A(n)$, and this is the cycle of $B(n)$, meaning that either the square never divides $B(n)$ or it always does for a certain prime. I bet on never dividing, albeit being aware of the possibility in order to do a a deeper study if the result wasn't right at first. But it was).

Now, enter the problem thread. It turns out that I had a much better solution almost at my hands! We can try using some gcd logic to calculate $\gcd(A(n), B(n))$ from its formulas. First, extract the trivial 6, and we have

$$G(n) = 6 \gcd\left(2 \cdot 4^{n-2} - 3^{n-2}, (3n - 23)\, 4^{n-2} + (2n + 13)\, 3^{n-2}\right).$$

We can notice that both values are odd, so the gcd remains unchanged if we multiply the second one times 2:

$$G(n) = 6 \gcd\left(2 \cdot 4^{n-2} - 3^{n-2}, (6n - 46)\, 4^{n-2} + (4n + 26)\, 3^{n-2}\right).$$

Now we subtract the first value times $3n - 23$ from the second:

$$G(n) = 6 \gcd\left(2 \cdot 4^{n-2} - 3^{n-2}, (7n + 3)\, 3^{n-2}\right).$$

Yes, that's the same $7n+3$ I arrived at above. Now, since 4 and 3 are coprime, so are $2 \cdot 4^{n-2}$ and 3^{n-2}, and in particular, so are $2 \cdot 4^{n-2} - 3^{n-2}$ and 3^{n-2}. And with this we can remove the 3^{n-2} factor from the second argument, leaving:

$$G(n) = 6 \gcd\left(2 \cdot 4^{n-2} - 3^{n-2}, 7n + 3\right).$$

And this is workable! No need to work prime by prime, no need to fear the possibility of squared primes, just calculate the bona fide gcd at each step. It's simple: iterate for $n \in [3, 10^7]$, and for each one, use binary exponentiation to calculate the result of the first argument modulo the second. If the result is some value a, now calculate the gcd of a and $7n + 3$, and add that value to the total. Multiply times 6 at the very end, and in 2,2 seconds the answer arrives.

589. Quintinomial coefficients

Difficulty rating: 40 %.

Solution: 11651930052. *Solved: Sat, 27 Nov 2021, 12:32.*

Math knowledge used: *Odd Entries in Pascal's Trinomial Triangle* (Finch et al., 2008).

Programming techniques used: none.

Another one of these "there's a paper for that" problems. There is no way I could have derived this on my own, although the theory looks interesting and not so difficult to follow. It's still a moderately dense paper with lots of definitions. Anyway, after a million tries and some careful recurrence analysis that could have been fruitful but still suboptimal, I took the direct path and used the matrix multiplication formula, which is fast enough and doesn't require caching, recursive calls and so on. First we define the vectors e and w, and the matrices D_0 and D_1:

$$e = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad w = \begin{pmatrix} 1 \\ 3 \\ 2 \\ 4 \end{pmatrix}, \quad D_0 = \begin{pmatrix} 1 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \quad D_1 = \begin{pmatrix} 0 & 1 & 2 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

Now, let $n_i n_{i-1} n_{i-1} \dots n_2 n_1 n_0$ be the binary representation of some number N. That is, each n_i is a binary digit. We then define, for each index j,

$$M_j = \begin{cases} D_0 & \text{if } n_j = 0, \\ D_1 & \text{if } n_j = 1; \end{cases}$$

and, lo and behold, the function defined in the problem can be calculated as

$$Q(N) = w^T M_i M_{i-1} M_{i-2} \cdots M_2 M_1 M_0 e.$$

And that's it. The order is logarithmic on N, the matrices involved are not very big, and there are only 18 numbers to calculate, so the run time is well below one millisecond.

There is an alternate formulation in terms of recursive functions, based on $Q(2n) = Q(n)$ and on derivations of $Q(16n + i)$ (for odd values of i from 1 to 15) as a summation of already known values, which can be exactly determined by the matrix formulation; but the direct matrix operation is just faster. Even if there are some more calculations and many unnecessary multiplications times 0, the calculations are too direct and almost branchless, and the code is shorter as well, so the matrix formulation wins, in my opinion.

594. Rhombus Tilings

Difficulty rating: 85 %.
Solution: 47067598. *Solved: Sun, 12 Jun 2022, 03:02.*
Math knowledge used: combinatorics, Bareiss algorithm.
Programming techniques used: none.

Curious problem. I wonder if I could have solved it by brute force, without knowing the paper that spells the solution. I expected the result to be some 15 digit monster, but it's actually small enough that a brute force count or something with dynamic programming could reasonably solve it. Anyway, the paper is called *A formula for the number of tilings of an octagon by rhombi*, from N. Destainville, R. Mosseri and F. Bailly. The algorithm described is actually a bit more general than the one asked by the problem, with room for 4 different sides instead of only 2. The implementation is simple enough, but I lost a stupid amount because I got a stupid index comparison wrong, and it kept messing with the numerical values. This is what happens when you try to shoehorn an algorithm using 1-based indices into a language with 0-based indices, I guess. From the mathematical standpoint I guess that 1-based indices might make more sense, but from the programming sense, 0 is the standard and the way to go.

The algorithm has two steps. First we need to generate two sets of "families", which are merely 2D successions of integer values, one called X and another called Y. Then, for each combination of an X family and an Y family, we calculate a series of matrices and the product of their determinant results on a value. The solution of the problem is the sum of all such values.

Let a, b, c and d be the length of four consecutive sides. In our case, $a = c = 4$ and $b = d = 2$ (we could also take $a = c = 2$ and $b = d = 4$, and indeed the result is the same, but it's slower because there are many more terms). Each family is a 2D array with $b \times d$ terms, and the values

are between 0 and a in the case of X, or between 0 and c in the case of Y (in our problem these are the same and we could take some shortcuts to reuse the solution, but I went for the complete implementation, so I generate them separately). Then, the family X must be generated in such a way that elements with a bigger horizontal or vertical index have also equal or bigger values; that is: $k \leq k' \wedge l \leq l' \Rightarrow x_{k,l} \leq x_{k',l'}$. The family Y is similar, but while it still must increase horizontally, vertically it must decrease instead: $k \geq k' \wedge l \leq l' \Rightarrow y_{k,l} \leq y_{k',l'}$. These sequences can be generated recursively and, for the values we need, there aren't too many of them. I believed this to be the hard part (even if the recursive algorithm was relatively simple), but I spent a lot of time chasing a mistake in the other part.

Now, let's say that we have already created all our families. Then, for each combination of an X family and an Y family, we will generate $d + 1$ matrices, $M^{(u)}$, of size $b \times b$, plus other $b+1$ matrices, $P^{(v)}$, of size $d \times d$. Each element of these matrices is a combinatorial term of moderate complexity, and here is where I botched an index bound calculation which lost me a few hours. The product of all the $d + b + 2$ determinants is the term we must add; I wanted to stay in integer arithmetics, so I reused the Bareiss algorithm from problem 380 (this time, using longs instead of BigIntegers). Although, well, the Bareiss algorithm is there just because I wanted the algorithm to be generic. In practice, all I need for this problem are 2×2 determinants, which could have been calculated with the old, simple $a_{11}a_{22} - a_{12}a_{21}$ formula.

Choosing $a = c = 4$ and $b = d = 2$, there are 105 families of type X and another 105 of type Y (which are the exact same, swapped vertically; but, again, my algorithm is general and is valid for cases where $a \neq c$, so I didn't make use of this). So there are 11025 terms, each consisting of the product of 6 matrices of size 2×2, which in turn have combinatorial terms where the maximum upper term is 8. Of course I used my trusty old combinatorial number cache for this. The end result is very fast, about 40 milliseconds. If I go with the other route, $a = c = 2$ and $b = d = 4$, there are 1764 families of each type, and each one of the 3111696 terms needs to calculate 10 determinants of size 4×4, which takes about four seconds and a half. At least the result is the same, so I guess that my Bareiss implementation must be correct.

Most of the people from the problem thread are using this paper (or, sometimes, a different one!), but some brave ones went for the hand-made dynamic programming algorithm from scratch. All in all, I guess that the 85 % rating is kind of high, but maybe not totally undeserved.

By the way, this marks the first time my count of Project Euler problems solved exceeds my count of times having finished KOF 98. $596 > 595$, and fittingly, this is problem 594.

600. Integer sided equiangular hexagons

Difficulty rating: 35 %.
Solution: 2668608479740672. *Solved: Sun, 10 Apr 2022, 00:41.*
Math knowledge used: combinatorics.
Programming techniques used: none.

This is not exactly a challenging problem, but it is a long problem. 35 % sounds about right, especially since there isn't absolutely any complicated mathematics involved. I actually had a lot of fun with this, and filled a few sheets of paper with some derivations and simplifications. This is certainly not at the level of 777, but it's not that far. And the reason is similar: the problem is not particularly difficult, but there are a few cases to explore.

"Math knowledge used: combinatorics". There really isn't anything else to this problem. It's about finding all the cases and just counting carefully. Getting it right (and finding formulas with only as much as two levels of loop nesting) is not straightforward, and that's where my several hours filling sheets of paper went. Some cursory analysis can quickly determine the following:

- We can represent different hexagons by the lengths of their consecutive sides (I guess that more sophisticated representations are possible, but this is what I used. These representations don't appear in my final problem, but I used them a lot in my intermediate code and in my formulas).

- The only restriction for a given representation $abcdef$ to be valid is that these three conditions are fulfilled:

$$\begin{aligned} a+b &= d+e, \\ b+c &= e+f, \\ c+d &= f+a. \end{aligned}$$

  And actually, the third one can be deduced from the first two, so this can only reduce as much as 2 orders of magnitude (this means that the solution to the problem will always be approximately $O\left(n^4\right)$).

- Each representation can appear "rotated" (i.e. sides $abcdef$ might appear as $bcdefa$, $cdefab$ and so on) and reflected. This means that each hexagon might have as much as twelve different representations, and the problem description specifies that the "same" hexagon must not be counted twice.

- While 12 is the maximum amount of representations a single hexagon may have, it can also have less than that. The most extreme case is the regular hexagon, with a single representation, $aaaaaa$ for some fixed a.
- There are a few different categories of side distribution such as $aaaaaa$, $ababab$ and so on, and each has its fixed amount of different representations.

All this suggests a simple (well... at least, simple at a high level description) way of solving this problem: separate all the different cases, and count them separately, making sure that there isn't any overlap. This is a good idea in principle, but the case where all the six sides can be different (which is asymptotically the most numerous one) is a bitch to count correctly. So what I did was slightly different:

- Count separately the cases where the amount of representations is less than 12.
- Count also *all* the representations without worrying about repetitions.
- Combine these two results to get the total count of cases where the amount of representations is exactly 12.
- Sum all the cases.

It turns out that this is much easier, despite the second step, which is also a bitch to get right in $O(n^2)$ time, but not nearly as much as the completely irregular case since we don't need to remove all the repeated cases. So let's start with all the subcases and their formulas. There are a total of 9 subcases, although 3 of them have 12 representations, so I only needed to do the calculations for the other 6 (plus the big one). At each case there was, usually, a simple formula where the complications come from avoiding cases where two sides are the same (therefore corresponding to another, simpler case that must be counted separately because its amount of representations is probably smaller) and to counting only up to the given limit, $N = 55106$. I gave a name to each case, because yes, of course I used enums for this. The cases are:

- "Regular": obviously the most simple case. A regular hexagon where all the sides are the same (the only representation is $aaaaaa$ for some a). There are exactly $\left\lfloor \frac{N}{6} \right\rfloor$ cases, and each one has 1 single representation.

- "Elongated": Imagine that you take a regular hexagon with horizontal bases, and pull it from the side corners so that these bases grow. Or you can compress it as well. So the result is a pattern like *aabaab* where $a \neq b$. The amount of cases is

$$\left\lfloor \frac{N}{4} \right\rfloor \left(\left\lfloor \frac{N}{2} \right\rfloor - \left\lfloor \frac{N}{4} \right\rfloor - 1 \right) - \left\lfloor \frac{N}{6} \right\rfloor,$$

  and each case has 3 representations.

- "Superman regular": initially called just "Superman" until I discovered a similar, slightly more complicated, pattern. This one is *ababab* and it does look kind of like the Superman emblem. For reference, the first hexagon in the problem description is one of these. This is the last representation that has a simple formula, since for all the rest there are at least three side lengths and loops are mostly unavoidable. The formula is

$$\left\lfloor \frac{1}{2} \left(\left\lfloor \frac{\left\lfloor \frac{N}{3} \right\rfloor^2 - \left\lfloor \frac{N}{3} \right\rfloor}{2} \right\rfloor - \left\lfloor \frac{N}{6} \right\rfloor \right) \right\rfloor,$$

  which is fortunately much simpler in terms of code than in mathematical notation because of the way integer division works, and each hexagon has exactly 2 representations.

- "Prism". I used this name because this hexagon looks like the profile of a prism with three different side lengths, when looked at from an approximately isometric perspective. The side lengths are *abcabc*; simple enough, but this is the first case where there isn't a direct formula to work with (the previous cases had it even if additional care was needed because of the repetitions). After some symbol pushing, I arrived at the following formula which loops over the side a:

$$\sum_{a=1}^{\lfloor N/6 \rfloor} \left(\left\lfloor \frac{N}{2} \right\rfloor - a \right) \left(\left\lfloor \frac{1}{2} \left(\left\lfloor \frac{N}{2} \right\rfloor - a \right) \right\rfloor - a \right) - 2 \left(T\left(\left\lfloor \frac{1}{2} \left(\left\lfloor \frac{N}{2} \right\rfloor - a \right) \right\rfloor \right) - T(a) \right).$$

  Here, $T(x)$ is the triangular number $\dfrac{x(x+1)}{2}$. The formula looks kind of daunting but it's about six lines of code. In this category, hexagons can be represented in 6 different ways.

- "Trapezium": I was about to call this "potato" (this is the one in the rigth lower corner in the problem sample), but since three of its sides are similar, there is actually more regularity than it might look at a glance. This might be closer to the actual shape of the Superman emblem than the cases I actually called "Superman". Anyway, the sides are *aaabcb*, and it's actually a kind of elegant shape if you use the c and the second a as horizontal bases. This is also the first case where the additional restrictions are actually needed and not trivially fulfilled by the representation. Namely, we need to ensure that $a + a = b + c$. The asymmetry between b and c suggests calculating separately the cases $b < a < c$ and $c < a < b$, and the final formulas are curiously different. In both cases we iterate over a, but even the iteration bounds are different. For the case $b < a < c$ we get

$$\sum_{a=2}^{\lfloor N/5 \rfloor} \text{mín}\left\{n - 5a, a - 1\right\},$$

  and for the case $c < a < b$ we have

$$\sum_{a=2}^{\lfloor N/6 \rfloor} \left(a - \text{máx}\left\{7a - n, 1\right\}\right).$$

  We need to add the results of both formulas separately, but aside from that they can be treated as the same kind of shape. Each hexagon has 6 different representations.

- "Superman irregular". I initially missed this because my program to enumerate cases had a bug, and because the smallest of these cases has length 13 and therefore didn't appear in the problem description. I found it when I was bruteforcing the case for $N = 100$. The formula is also quite complicated, since there are four different sides: *abcbad*. The only useful restriction we have is $b + c = a + d$, and therefore this case has three degrees of freedom. I managed to find a workable formula by iterating over a and over $\Delta = b - a = d - c$, and counting valid values of c. The full scheme is this (no formula; it would be too much):

  - Iterate a from 1 to $\left\lfloor \frac{N}{4} \right\rfloor$, both included.
  - For each a, iterate Δ from 1 to $\left\lfloor \frac{n - 2 - 4a}{3} \right\rfloor$, included.

- Given a and Δ, c can take values from 1 to

$$x = \left\lfloor \frac{n - 4a - 3\Delta}{2} \right\rfloor,$$

but we need to take into account the restriction that side lengths can't be repeated. Therefore, we will use an additional variable r, initialised to 0.

- If $x \geq a$, there is a collision where $a = c$ (also $b = d$). Increase $r \leftarrow r + 1$.
- If $x \geq a + \Delta$, there is a collision $b = c$. Increase $r \leftarrow r + 1$.
- Finally, if $1 \leq a - \Delta \leq c$, there is a collision $a = d$, so again we need to increase $r \leftarrow r + 1$.
- Add $x - r$ to the result and move onto the next Δ.

This is one of the two $O(n)^2$ operations we need, the other being the full set. By the way, each of these hexagons has 6 different representations.

- "Four sides". Not a very imaginative name. Also yes, I named this before I found the "Superman irregular" pattern which also has four sides. The side length pattern is *aabbcd*, and here we have two different useful restrictions: $a + a = b + c$ and $a + b = c + d$. Although it's not difficult to iterate over this (four variables minus two degrees of freedom means two actual variables, meaning one single loop with a chunk count), I didn't need it in the end because each one of these hexagons has all full 12 representations, so I can put these into the generic box.

- "Five sides", that is, *aabcde*. Not a very imaginative name, once again. We have two restrictions, $a+a = c+d$ and $a+b = d+e$, and I actually spent some time deriving a quadratic scheme for these, but I didn't use it in the final program because it so happens that these hexagons also have the full set of 12 representations.

- "Irregular": *abcdef*. I didn't bother with a formula for this. Counting the different cases is not so bad, but taking care of the repetitions sounds like hell. Obviously in this case we also have 12 representations, so no formula needed.

Aside from all these formulas, I needed a formula for all the representations, and arriving at a quadratic formula was a matter of operating carefully. In the end this is the case *abcdef*, but without caring about repetitions. The

restrictions are $a+b=d+e$ and $b+c=e+f$. So the first option is to iterate over a, b and e in that order. For each of these, we can set $d=a+b-e$. We also have $c-f=e-b$, and $c+f\leq N-(a+b+c+d)$. This means that there are exactly

$$\left\lfloor \frac{N-(a+b+c+d)-|e-b|}{2} \right\rfloor$$

cases. Which is a good start, but the combination of the floor and absolute value operations makes things difficult if we want to use summation formulas to condense one of the iteration levels into a single expression or sequence of code, to have $O\left(n^2\right)$ instead of $O\left(n^3\right)$. To take care of the absolute value, I separated the cases $b<e$ and $b\geq e$, although both end up with the same treatment. The counts for a given tuple of (a,b,d) are respectively

$$\left\lfloor \frac{n-2\left(a-b\right)-e+b}{2} \right\rfloor, \quad \left\lfloor \frac{n-2\left(a-b\right)-b+e}{2} \right\rfloor.$$

The next thing to note is that values are going to be "almost" consecutive, and summing chunks of these formulas might be feasible. The details to take into account are that values appear twice, and that at the very start and the very end they might appear only once. The obvious approach is to count the values that appear twice, and include the singleton ones if needed. We note that the upper limit for a is $\left\lfloor \frac{N}{2} \right\rfloor - 1$, and the upper limit for b is $\left\lfloor \frac{N}{2} \right\rfloor - (a+1)$. Given a pair (a,b), we calculate the following for $b\geq e$:

- First, the bounds for e, $e_0=\text{máx}\left\{1,3b+2a+2-n\right\}$ and $e_f=b$.
- Let $f\left(x\right)=n-2a-3b+x$. Calculate $f_0=f\left(e_0\right)$ and $f_f=f\left(e_f\right)$. This is the numerator of the fraction we need to calculate the floor of.
- If $f_0=f_f$, just accumulate $\left\lfloor \frac{f_0}{2} \right\rfloor$ and move on to the next (a,b) pair.
- Otherwise we need to actually sum the sequence. We will have $f_0<f_f$, and therefore the non-repeated values happen when f_0 is odd (in whose case, accumulate $\left\lfloor \frac{f_0}{2} \right\rfloor$ and increase f_0 in 1) and when f_f is even (in whose case, accumulate $\left\lfloor \frac{f_f}{2} \right\rfloor$ and decrease f_f in 1).

- Now, accumulate the whole sequence, whose sum is

$$\left(\left\lfloor\frac{f_f}{2}\right\rfloor+1-\left\lfloor\frac{f_0}{2}\right\rfloor\right)\left(\left\lfloor\frac{f_f}{2}\right\rfloor+\left\lfloor\frac{f_0}{2}\right\rfloor\right).$$

For $b < e$ the operation is similar, but not identical.

- The bounds are $e_0 = b + 1$ and $e_f = \min\{n - 2a - b - 2, a + b - 1\}$.
- The function is also different, $f(x) = n - 2a - b - x$. Also, now we will have $f_0 \geq f_f$.
- Since now f_0 is the higher value, the corrections will now be to make sure that f_0 is odd (add the half and decrease if it's even) and f_f is even (add the half and increase if it's odd).
- For the same reason, the final formula is

$$\left(\left\lfloor\frac{f_0}{2}\right\rfloor+1-\left\lfloor\frac{f_f}{2}\right\rfloor\right)\left(\left\lfloor\frac{f_f}{2}\right\rfloor+\left\lfloor\frac{f_0}{2}\right\rfloor\right).$$

These formulas allow us to iterate over just a and b, calculating a single value for each pair and adding it to the total. This calculates the full set of hexagons (with repeated representations being counted multiple times) in $O(n^2)$ time. And with this, all that looooong series of formulas is mostly over. The final set of operations is this:

- For all the six hexagon cases where the amount of representations was less than 12, calculate the amount of unique representations, so that there are A regular hexagons, B elongated ones, C regular "Superman" cases, D prism cases, E trapezium cases and F irregular "Superman" cases.
- Calculate the full amount of cases without caring about representations with the scheme described above. Let R be the result.
- The total amount of "four sides", "five sides" and "irregular" cases, combined, can be calculated as

$$G = \frac{1}{12}\left(R - A - 3B - 2C - 6\left(D + E + F\right)\right).$$

- The problem result is the sum of $A + B + C + D + E + F + G$.

The analysis took me a lot of time, but the run time is just above 2 seconds. The problem thread includes lots of very interesting formulas, including an extremely simple polynomial one that actually works.

611. Hallway of square steps

Difficulty rating: 60 %.

Solution: 49283233900. *Solved: Mon, 27 Jun 2022, 09:13.*

Math knowledge used: "first prime" Erathostenes sieve, sum of squares theorem, prime counting function.

Programming techniques used: binary search, Lucy Hedgehog sieve.

This problem was pretty hard. Out of the three usual steps (formulation, analysis and coding), I got most of the problems in the last one, because there are so much opportunities for small errors to appear. The basic formulation of the problem could be stated as this: find how many numbers n in the range $[0, 10^{12}]$ are such that they can be expressed as the sum of two different squares, $x^2 + y^2 = n$ with $0 < x < y$, in an odd amount of ways. This could be the initial formulation, which is not really useful. I was hoping that there would be a relatively simple way to characterise these numbers, but it's not; this makes the analysis a bit complicated.

The obvious tool for the analysis, at least for the first part, is the sum of squares theorem. For the purpose of this problem, what this theorem tells us is that:

- The amount of available sum of squares depends mostly on the prime factors of the form $4k + 1$. A factor of 2 is irrelevant (yes, even for the purposes of counting only cases with $0 < x < y$, since adding a factor of 2 transforms a useless solution like $(0, x)$ into another useless one like (x, x), and vice versa), and factors of the form $4k + 3$ may appear as long as they are squared.

- Considering only the powers of prime factors of the form $4k + 1$, two different numbers with the same prime exponents will have the same amount of ways to be expressed. The order of the powers is, of course, also irrelevant.

This suggests a clear scheme which can be described at a high level like this: enumerate sets of powers, determine whether they result on an odd amount of powers, and then, for each valid set, enumerate the numbers. This is still very coarse, since enumerating the powers is not trivial, and enumerating the numbers is even more complicated.

We can start with the powers. First of all, we can see that we won't need more than 8 exponents, since the product of the first 9 primes of the form $4k + 1$ already exceeds the limit, 10^{12}. With this limit, we can enumerate powers in a controlled way recursiely, updating at each point the biggest exponent (or potentially starting the next index) in such a way that powers are always decreasing. This analysis reveals a total of 241 power distributions, out of which only 53 are valid. These range from [1] (that is, single primes, which can be expressed in exactly one way) up to [17] (9 ways), including more complicated distributions like $[5, 4, 2]$ (45 ways) or $[4, 2, 2, 2, 1]$ (135 ways). These sets can be generated pretty fast, and there are few of them, which is good. Of course, this is the easy part.

Now, let's enumerate all the numbers for a given set of powers. I'm not going to say that enumerating these is obvious, but it's not especially hard to write a basic recursive algorithm that does this, with one level of depth per indices in the array with the distribution powers, iterating a single prime at each level. Here are, however, some of the possible gotchas:

- Remember that factors of the form $2^a \prod_i (4k_i + 3)^{2b_i}$ must be taken into account. I precalculated all such factors in the given range (there are about $4 \cdot 10^5$), which is one of the most costly operations in the whole problem. Then, I iterated per factor (that is: these factors are $1, 2, 4, 8, 9 \ldots$, so I do the full calculation for N, $\frac{N}{2}$, $\frac{N}{4}$, $\frac{N}{8}$, $\frac{N}{9}$ and so on). This is not as slow as it seems, but still, there are a lot of operations.

- Simple cases like [1] don't seem to be complicated, but N is far too big and a prime sieve doesn't go well. I had to use a prime counting function. A prime counting function for only $4k + 1$ primes. Which, fortunately, I had developed during my experiments with Lucy Hedgehog sieves for problem 715 (although IIRC I ended up not using it). It's kind of slow because it uses objects instead of longs (since it works with pairs of $(4k + 1, 4k + 3)$ primes in a given range), so this is about the most time consuming operation, more or less on par with the factor generation. The rest is super direct since it's either querying the Lucy Hedgehog sieve or doing binary searches.

- Accounting for repeated primes is a bit of a bitch. With schemes with repeated powers, like $[2, 2, 1]$, we want to iterate over indices in such a way that, for a same exponent, we "keep" the index: that is, if the first exponent of 2 is used for a prime like 17, we want the next iteration to continue at the next $4k + 1$ prime, which is 29. Otherwise we will generate repeated numbers. The last element is particularly problematic because instead of iterating we do a direct check, either with the Lucy

Hedgehog sieve or with a binary search in the list of primes. We have to be very careful in order to get the count we really want.

- There is also the matter of already used primes. When iterating over a case like $[2, 1]$, the exponents are different so we would "reset" the prime index, but still, we wouldn't want to use the same prime for both exponents. A set of "already used" primes has to be kept, which must be taken into account both during normal iterations and during the last exponent, with the prime counting and so on. The combination of this and the previous condition, plus a few off-by-one errors related to binary searches, was where I lost the biggest amount of time.

- Finally, in order not to iterate over unneeded values, it would be nice to end the iterations as soon as possible. I used a very conservative approach, which looked more or less like this: consider the current exponent, and note how many times it appears. Also sum the rest of the exponents, and assume that the prime will be at least 5. So for a set of remaining exponents like $[4, 4, 2, 1]$ we would have a "current" exponent of $4 + 4 = 8$, and an "added" exponent of $2 + 1 = 3$. Therefore, divide the current limit by 5^3, and then take the 8th root of the result; that's the maximum prime you need. Fortunately, this means that this "maximum prime" gets small very, very fast; especially so for bigger exponents.

Well, that actually takes care of the description of the algorithm, mostly. Generate the sets of valid powers, generate the sets of extra factors (for 2 and the $4k+3$ primes), and then, for every combination of one of each, count how many combinations of $4k + 1$ primes are there. Keep in mind all the gotchas above. I also used a cache for powers and roots (in fact: in order to calculate roots, I used binary searches over the powers. I trust that much more than floating point math, considering that infuriating issue where Java believes that the cube root of 125 is not quite 5, and calling floor with that result returns 4).

There are a lot of iterations, but they are actually pretty fast. Most of the calculations, as I said above, are spent in the factor generation and in the Lucy Hedgehog sieve. I got a run time of around 100 seconds, which is above par. There is a bit of room for improvement, but this is good enough for me.

635. Subset sums

Difficulty rating: 40 %.

Solution: 689294705. *Solved: Sat, 11 Jun 2022, 03:26.*

Math knowledge used: combinatorics, Erathostenes sieve.

Programming techniques used: Montgomery modular inversion.

I'm guessing that getting the whole formula from scratch is not super complicated, but I just found it on OEIS after bruteforcing smaller values. It's quite simple, really. For any odd prime p, we have:

$$\begin{aligned} A_2(p) &= \frac{1}{p}\left(\binom{2p}{p} + 2(p-1)\right); \\ A_3(p) &= \frac{1}{p}\left(\binom{3p}{p} + 3(p-1)\right). \end{aligned}$$

And yes, the formula generalises nicely for any A_x for whatever integer x that you like. As usual, calculating big binomials coefficients is not really possible, but why would you do that when you can just reuse previous results. After all,

$$\begin{aligned} \binom{2p}{p} &= \frac{2p(2p-1)}{p^2}\binom{2(p-1)}{p-1}; \\ \binom{3p}{p} &= \frac{3p(3p-1)(3p-2)}{2p(2p-1)\cdot p}\binom{3(p-1)}{p-1}. \end{aligned}$$

It's a matter of applying these formulas over and over, as needed. These formulas don't work for $p = 2$, so the result is calculated differently for that prime. For the rest, we can apply a very simple algorithm. Therefore, a full algorithm would be like this:

- Initialise a "result" variable as $A_2(2) + A_3(2) = 2 + 6 = 8$.
- Also initialise two variables, $b_2 = \binom{4}{2} = 6$ and $b_3 = \binom{6}{2} = 15$.
- We also need a "last prime" variable which will be initialised as 2 since that's the value reflected in these combinatorial variables.
- Now, calculate all the primes up to the given limit, $L = 10^8$.
- Let P be the biggest prime found in the previous operation. So, use Montgomery inversion (or whichever method you like to generate inverses in a range) to calculate inverses of all values modulo $M = 10^9+9$ in the range $[1, 2P]$. You can generate them on the fly if you like, but run time will suffer a bit.

- Now, iterate over every odd prime p in the problem range.
  - First we need to update b_2 and b_3, so, for every value q in the range [last prime $+ 1, p$], calculate:

    $$\begin{aligned} b_2 &\leftarrow b_2 \cdot (2q-1) \cdot 2q \cdot \left(q^{-1}\right)^2; \\ b_3 &\leftarrow b_3 \cdot (3q-2) \cdot (3q-1) \cdot 3q \cdot q^{-1} \cdot (2q-1)^{-1} \cdot (2q)^{-1}. \end{aligned}$$

    Every single of these multiplications must be modulo M, so this is what takes the vast majority of the run time, since divisions are costly.
  - Now that b_2 and b_3 are updated and correspond respectively to $\binom{2p}{p}$ and $\binom{3p}{p}$ modulo M, calculate

    $$a = A_2(p) + A_3(p) = (b_2 + b_3 + 5(p-1))\, p^{-1}$$

    where, again, the inverse is modulo M. The product should also be done modulo M.
  - Increase result $\leftarrow$ result $+ a$. You can mod if you want, but since every value is smaller than 2^{30} and there are much less than 10^8 of them, no overflows will happen if the result variable is a long. So I only modded at the end.
  - Remember to update the "last prime" variable, assigning the value p before going to the next iteration.
- After this loop, mod the result with M to get the solution to the problem.

That's it. This is one of these cases where the difficulty resides in finding the pattern, since the problem has been studied by smarter people and OEIS, as so often happens, has the answer. The implementation is really simple despite the long series of mods. And if you use a fast inversion method, the run time is quite low (3,3 seconds on my PC).

646. Bounded Divisors

Difficulty rating: 40 %.
Solution: 845218467. *Solved: Sun, 2 Jan 2022, 03:37.*
Math knowledge used: prime decomposition.
Programming techniques used: meet-in-the-middle.

I don't know how to describe what I used to study the factorial divisors, so I'm just slapping "prime decomposition" and I'll leave that label. What I did is the following: let $N = 70$ be the factorial operand (the other inputs for the problem are the lower bound L, the upper bound U, and the mod M). So I enumerate all the primes below N, and for each one of them, calculate how many times does it appear in $N!$. There is a simple algorithm that calculates this for any prime p in $O\left(\log_p N\right)$ steps:

1. Start with the variable result, r, set to 0.
2. Let $x = N$.
3. And now we start a `do` loop. I don't use these so often.
   a) Let $x = \left\lfloor \frac{x}{p} \right\rfloor$.
   b) Update r by adding the current value of x.
   c) Continue if $x \geq p$, end otherwise.
4. The value of r after the loop is the highest power of p that divides $N!$.

Intuitively, the first step of the loop adds the multiples of p, the second adds the multiples of p^2, and so on. I've used this algorithm elsewhere, but it's not as common as one could think.

We will use this algorithm to find the biggest exponent of each prime. In fact, this is finding the prime decomposition of 70!:

$$70! = 2^{e_2} \cdot 3^{e_3} \cdot 5^{e_5} \cdot \ldots \cdot 61^{e_{61}} \cdot 67^{e_{67}}.$$

The total amount of divisors of 70! is $\prod (e_p + 1)$ for all the primes below 70. This is more than $3{,}5 \cdot 10^{12}$, so enumerating all of them is not workable, but we can use a meet-in-the-middle approach. I even managed to do it without BigIntegers, although the run time doesn't change that much (about one fifth less, from 5,8 to 4,6 seconds). The first thing we are going to do is divide these divisors in two sets of a moderately similar size. To do this, I iterated over the primes in order, and kept calculating the amount of divisors with the first N primes, until this first half exceeded the rest. The result is that there are 3204432 factors that come from the first 5 primes, and 1105920 that come from the rest. The general idea is to iterate over this smaller set, adding partial sums extracted from the bigger one.

The first thing to do, and the only step that is common to both halves, is to generate a sorted list of divisors. Now, I don't want the full number, so

I can get away with a structure that is very nice in its simplicity: a sorted map containing all of the divisors from this half. For each number x, the key will be $\log x$ (I can use natural logarithms, since I'm only doing this so that big values can be sorted using a small data type; I don't care about the actual value beyond using it for sorting) and the value will be $x \cdot \lambda(x) \bmod M$ (always positive, though). This can be calculated pretty quickly iterating by prime factors.

Now, let's focus on the first set. Given this sorted map, we are going to create a different structure: a map of partial sums. It doesn't have much complication: iterate over the factors map, add the value to a current sum (mod M, of course), store in the new map using the same factor logarithm as key.

Now we iterate over the second, slightly smaller set. This set happens to be sorted because I reused the same function to create it, but it doesn't actually need to be sorted. The steps are simple: given an entry with key l (this is the logarithm of the factor) and value n (this is the number itself, but modded) look in the map of partial sums for the highest entry below $\log L - l$ and the highest entry below $\log U - l$. Subtract the value of the lower entry (or 0 if there is none) from the value of the higher entry (or 0, again, if it doesn't exist. In whose case we can actually return early). Multiply the difference times the modded number n and add it to an accumulator variable, modding where necessary. The final value of this accumulator is the solution to the problem.

657. Incomplete words

Difficulty rating: 30 %.

Solution: 219493139. *Solved: Sat, 27 Nov 2021, 06:35.*

Math knowledge used: combinatorial numbers, inclusion-exclusion, modular inverses.

Programming techniques used: Montgomery modular inversion, binary exponentiation.

I can't believe I let this problem slide so far. This is what happens when a problem is published in the middle of an exams period or right after, during my decompression, I guess. The formula is so simple that 30 % actually sounds too high. Maybe people are having problem with the chained modular inverses?

Anyway, here's the thing. Let α be the amount of symbols in the alphabet, and N be the maximum length of the string. The full result of the problem

can be computed with this formula:

$$R = \sum_{i=0}^{\alpha-1} (-1)^{\alpha-1-i} \binom{\alpha}{i} S(i, N),$$

with $S(i, N)$ being the total amount of strings of length N that can be constructed with an alphabet of i symbols. Here is where one of the slight complications of the algorithm appears, since $S(i, N)$ has a piecewise formula:

$$\begin{aligned} \text{If } i = 0, \quad \text{then} \quad & S(i, N) = 1; \\ \text{If } i = 1, \quad \text{then} \quad & S(i, N) = N + 1; \\ \text{otherwise}, \qquad\quad & S(i, N) = \frac{i^{N+1} - 1}{i - 1}. \end{aligned}$$

The second slight complication comes from the combinatorial numbers. Calculating them using additions would be an $O(\alpha^2)$, not doable when $\alpha = 10^7$, so we use an iterative process where we build the numbers recursively. I thought that it would be natural for this problem to iterate downwards (this way the first sign of the sum is always positive), from $i = \alpha - 1$. So we keep a value $C_i = \binom{\alpha}{i}$, which at the start of the operation is defined as $C_\alpha = \binom{\alpha}{\alpha} = 1$. At each iteration, we use that

$$\frac{C_i}{C_{i+1}} = \frac{\dfrac{\alpha!}{i!\,(\alpha - i)!}}{\dfrac{\alpha!}{(i+1)!\,(\alpha - i - 1)!}} = \frac{(i+1)!\,(\alpha - i - 1)!}{i!\,(\alpha - i)!} = \frac{i+1}{\alpha - i}.$$

So we need to multiply C_{i+1} times $\dfrac{i+1}{\alpha - i}$ to get C_i. We are guaranteed that this is an integer number, but anyway we are not going to work with rationals or reals; we need a modular result, so we will actually multiply by the product of $i + 1$ and the modular inverse of $\alpha - i$.

And here it comes, the final detail of the problem: since we are going to need a ton of modular inverses, and they happen to be the inverses of 1 to $\alpha - 2$ modulo the same value, we can use some magic in the form of Montgomery inversion (I actually used a slightly different formula, which is faster and which I got from some problem thread ages ago) to calculate the full array of inverses on the go, reusing previous results. By the way, we need these inverses for the division in $S(i, N)$ as well. The numerator is calculated using binary exponentiation, naturally.

So the process is just this: first calculate the modular inverses array, and then, for each i in the appropriate range, calculate the addend and either add it or subtract it, depending on the parity of $\alpha - 1 - i$. It's as simple as it sounds, and the run time is very fast ($O(\alpha)$ in space and $O(\alpha \log N)$ in time, although with a big constant because of all the modular calculations. The result comes in about 1,25 seconds.

And with this, Nov 2021 becomes oficially my most prolific non-holidays Project Euler month, with 14 problems solved (plus maybe some in the next three days), some of them being really hard ones I had been putting away for a long time, like 261 or 275.

660. Pandigital Triangles

Difficulty rating: 40 %.

Solution: 474766783. *Solved: Tue, 5 Oct 2021, 17:30.*

Math knowledge used: numerical bases, triangular numbers, Eisenstein triples.

Programming techniques used: none.

This is kind of similar to the early problem 9, in the sense that it's about generating triangles and rescaling them into proper values. This is more complex than that, but actually, not that much. I've added a relatively complicated sequence of objects with the goal of avoiding certain recalculations and to use smaller loops in the main code, but most of the complexity is kind of accidental and I've added it mostly because I like overengineering my code (I also believe that thet end code is more efficient because most of the branching is relegated to the start of the code).

Anyway. The base for the problem is that the triangles can be generated using Eisenstein triples. That is: for every ordered pair of coprime numbers, (m, n), with $m > n$ and $m - n$ not a multiple of 3, we are going to get a single primitive triangle (this iterative scheme is guaranteed to hit every primitive triangle exactly once), and these triangles can be rescaled into pandigital ones. We can stop when the primitive triangles get too high. By the way, given these m and n, the triangle has these sides:

$$\begin{aligned} a &= m^2 + mn + n^2, \\ b &= 2mn + n^2, \\ c &= m^2 - n^2. \end{aligned}$$

Now, how to rescale? Should we try every multiple until they get too high? Of course not. We can sieve. A lot, in fact. This is where the code

needs to be the most careful for this problem. So we have a triangle, (a, b, c), and we want to apply a rescaling parameter k so that the rescaled triangle, (ka, kb, kc), is pandigital in some base B. I'm assuming that smarter people will be able to see more restrictions for the resulting triangle, but I can see two ones:

- Obviously, the resulting triangle has to have the correct amount of digits. Exactly B digits. So, if k is too small, we get less than B digits and the triangle can't be pandigital. If k is too big, we get more than B digits and, again, it can't be pandigital (since every digit must happen exactly once).

- Additionally, the sum of all the digits must be equal to $\sum_{n=0}^{B-1} n = \frac{B^2 - B}{2}$. This is actually a big restriction and helps a lot to trim the set of possible values.

I'm going to start with the second kind of restriction, since it helps to understand the exact architecture. The way we are going to apply this restriction is by comparing it with the old and trusty rule of divisibility by 9: sum all the digits, and repeat until we get a single digit. If it's 0 or 9, the number is a multiple of 9. The usual mathematical term for this operation is *digital root*. So we are going to get the digital root of the triangle, which I will call s since it's basically the modulus of the sum $a + b + c$ after dividing it by $B - 1$. Clearly, the product ks must be congruent with $\frac{B^2 - B}{2}$ (modulo $B - 1$), since it must be include all the digits from 0 to $B - 1$ exactly once, and there is an interesting casuistry here because this has a different behaviour depending on the parity of B.

- If B is even, then $B-1$ is odd. So $\frac{B}{2}$ is an integer number, so $\frac{B^2 - B}{2}$ is congruent with 0 (modulo $B-1$). Now, let's see. We have a digital root s, and we want to find the set of numbers K such that $k \in K \Leftrightarrow ks \equiv 0 \pmod{B - 1}$. This is, fortunately, easy to translate to arithmetical terms. Let g be the gcd of s and $B - 1$. Then, k is the set of multiples of $\frac{B-1}{g}$. That's it.

- If B is odd, things are not so simple. $B - 1$ is even, and $\frac{B^2 - B}{2}$ is an odd multiple of $\frac{B-1}{2}$. Which means that $\frac{B^2 - B}{2} \equiv \frac{B-1}{2}$ (modulo

$B-1$). We need to operate more carefully, and there are cases where a triangle can't be rescaled into a pandigital one (namely, if $s \equiv 0 \pmod{B-1}$, there is no way that we are going to rescale it into something that verifies $ks \not\equiv 0 \pmod{B-1}$). What we will do is separate the powers of 2 from the rest. So let $s = 2^\alpha \beta$ for some odd β, and let also $B - 1 = 2^\gamma \delta$ for some odd δ. If $\alpha \geq \gamma$, then there is no solution. Otherwise the solution must be an odd multiple of $p = 2^{\gamma-\alpha-1}$. On the other hand, let $g = \gcd\{\beta, \delta\}$, and $q = \dfrac{\delta}{g}$ (yes, this is the same thing we did in the simple case for even B). And k must be an odd multiple of pq.

The nice thing is that these conditions can be precalculated. For each relevant base B we can keep an "arithmetic progression" object that encapsulates the behaviour of k that we need, and these progressions can be restricted to the range that results in a proper amount of digits when we apply the first restriction.

So now we want to apply the first restriction. We will keep an array with all the powers of B up to a certain amount (I used $\left\lceil \dfrac{B+1}{2} \right\rceil$, which I believe is overkill). This is very convenient from the start, because a simple binary search is enough to determine the amount of digits of a number: let's say we have a number x and we want to know the amount of digits it has in base B. Using this array and Java's `Arrays.binarySearch` we can know it in logarithmic time without any floating point operation. For example, if the binary search determines that x is somewhere between B^4 and B^5, we know for sure that it has 5 digits. And if x happens to be exactly B^N for some N, then the number will have exactly $N + 1$ digits. But this array can be used to determine the range of valid values of the rescaling parameter k. First, we determine the total amount of digits of a, b and c combined using the above procedure. Let n be such amount of digits. If $n \geq B$ we can terminate immediately. Otherwise, the scaling parameter must add exactly $d = B - n$ digits. So what we are going to do is the following: for each $x \in \{a, b, c\}$, and for all powers of B that are greater than x, list of the values of $\left\lceil B^N \right\rceil x$. Each one of these values represents a value of k that increases the amount of digits of one of $\{ka, kb, kc\}$, therefore it marks an amount of digits in the total set. We can then combine these lists into a single one and sort it. If we need to add d digits, the valid range of values for k is those between the positions $d - 1$ and d in the sorted array (if $d = 0$, the lower limit is 1). Note that this set is closed at the left and open at the right. It's also possible that the bounds are the same number, because it increases two of the values at the same time. Well, this just means that at certain value of k we simultaneously

increase the amount of digits of two of the three sides of the triangle, and in such a way that it skips the amount of digits we were aiming for. Tough luck! I don't know if this actually happens, but the code can manage it naturally.

Now, with all that theoretical stuff out of the way, we can start designing OBJECTS! With lots of factories and such. Overengineerer and proud of it, baby!

At the lowest level we have an `Enumerator` class which is just a stupidly simple iterator-like object that returns the proper values of k, incrementing them by a fixed amount until the upper limit has been surpassed. These `Enumerator`s are generated by objects of an `ArithmeticSequence`, which are immutable and precalculated, but are not restricted to a range. There are three implementations: one for the simple "even B" case, another one for the "no solutions" case, and a final one for the "odd multiples" case. Finally, we have a `PandigitalChecker` which has a precalculated set of `ArithmeticSequence`s. The operation is as follows:

- First, create the `PandigitalChecker` objects. There will be exactly one per base. Each one of this checkers will also have a set of $B - 1$ `ArithmeticSequence`s, corresponding respectively to remainders 0, 1, $2 \dots B - 2$, modulo $B - 1$. This means that all the branching for the different cases happens beforehand, and we effectively retrieve the appropriate function pointer for each triangle, depending on the exact value of $s = a + b + c$.

- Now, the main iteration. We use increasing values of m, and for each one of them, we iterate over values of n, skipping those where the difference is a multiple of 3: $m - 1$, $m - 2$, $m - 4$ and so on. If m and n are coprime, we calculate the primitive triangle (a, b, c) along with its perimeter s. Each checker will first verify whether the amount of digits is below the limit, and in that case, we apply the range restriction to get a working range, $[k_1, k_2)$. Then, using the remainder of s modulo $B - 1$, we retrieve the proper `ArithmeticSequence` object and we pass the aforementioned range in order to create the correct `Enumerator`. For all the values given by this enumerator, we rescale the triangle to get some triple (ka, kb, kc) with exactly B digits, so we extract their digits into a bitset that keeps the tally. If each digit is found exactly once, then we have found a pandigital triangle and we add its longest side (which is always ka) to the result.

- After each value of m from the main iteration we check whether the values are still below the limit. If every single triangle was above the limit for some checker, we can remove it from the list of active checkers,

so that the main iteration ends when there aren't any active checkers left.

I had a lot of fun with this code because the maths are actually simple and I could concentrate on making all that tangled mess of interconnected objects and optimising for precalculations and so on. My brand new PC runs the code in about 7,5 seconds, and I guess I could reduce this if I had been less conservative about when to finally remove an active checker (unlike Pythagorean triples, here we have the side $c = m^2 - n^2$, meaning that there is no clear line to stop a checker inside a given value of m. Infuriating).

666. Polymorphic Bacteria

Difficulty rating: 45 %.
Solution: 0.48023168. *Solved: Fri, 10 Jun 2022, 05:49.*
Math knowledge used: Newton-Raphson method.
Programming techniques used: none.
HELL AND FIRE WAS SPAWNED TO BE RELEASED!

Now, in all seriousness: this problem was very, very underwhelming. I did indeed postpone it for a long time because the long description looked daunting, but it's actually incredibly easy. "Could have done it while at second year of university, maybe even first one" level of easy. It also gave me an opportunity to write overengineered code with enums and interfaces and all that shit. Still, it took me like an hour to write it, two at most. It's that direct. Although I did reuse a matrix class with inverse calculation and what not. Once again, Gaussian elimination in Java is just so much faster than in Matlab or Mathematica that it's clearly the best solution.

There are 500 equations, which is not a big number at all, so I used Newton-Raphson. The most time-consuming operation at each iteration is an $O(n^3)$ matrix inversion, and 500^3 is well within the amount of operations that a processor can do in a second, after all. So I kept an array with the current solution (initialised with everything at 0, really). To model the calculation of the function and its Jacobian, I used a `Behaviour` interface with methods to calculate the probability of dying and its derivative, based on the current value of the probability, p, and then I implemented classes like `Die` or `Mutate` to model each particular case. For each "class" of bacteria I have an array of such behaviours, initialised with the pseudo-random sequence as indicated by the problem description. These behaviours define equations of the form $p_i = \sum_j f_{i,j}(p)$, which in turn define a series of functions whose roots I want to find: $f_i = \sum_j f_{i,j}(p) - p_i$. In other words, I have a vector function $f(p)$

with 500 elements, and I want to find the (hopefully unique) vanishing point within $(0,1)^{500}$. In fact I actually used $m \cdot f(p)$, which blatantly shares the same zeros, to avoid divisions. So, using the Newton-Raphson algorithm, I solve the problem using these steps:

- Use the pseudorandom sequence to generate the behaviours and the $f_{i,j}$ functions.
- Initialise p. I used $p_i = 0$ for every i, and it was good enough.
- Now, the iterative algorithm:
  - Calculate $f(p)$ and $Jf(p)$ by iterating over the behaviours with the current values of p.
  - Calculate the decrement as $\Delta = (Jf(p))^{-1} \cdot f(p)$.
  - Update p by subtracting the vector Δ.
  - Calculate the modulus of the vector Δ, and finish iterating if it's below certain tolerance. I used 10^{-9}.
- After the iterations have finished, the first element of p is the solution of the problem.

That's it. Defining the enum and the small behaviour classes takes about 80 % of the code for this problem. The remaining is just a set of vector and matrix operations which I already had for many other problems. The run time is about 70 milliseconds, and it takes just about 5 iterations of the Newton-Raphson algorithm. I was fearing that the algorithm wouldn't converge unless a good base solution was provided, but this is definitely not the case; I was also fearing that the convergence could happen but be very slow, requiring thousands of iterations (which is not good when each iteration has about 10^8 operations), but this was, laughably, also not the case.

This problem is clearly designed to be intimidating, but it's really one of the easiest problems out there. That 45 % level of difficulty can only come from people not even attempting it, which was also my case for an embarrassingly long amount of time.

669. The King's Banquet

Difficulty rating: 45 %.
Solution: 56342087360542122. *Solved: Wed, 8 Jun 2022, 07:48.*
Math knowledge used: none.

Programming techniques used: memoisation.

Interesting problem. Like many more, it looks more or less impossible at first, since the sequences look more or less random; but then you find the pattern, which is pretty simple. But apparently I'm doing this in a very different way than most people! Most people derive a relatively explicit formula for the value at position n of a series with F_k elements. What I do is building a solution from the two previous ones.

First, let's describe how a standard solution is derived term by term (this is not workable when there are nearly 10^{17} terms, but it does the job for smaller series): we can construct the sequence for the k-th Fibonacci number, F_k in the direction of the *right* of the king (i.e. the opposite of the one the problem wants) by doing this:

- The first element is F_k. In fact, the next ones are F_{k-1}, F_{k-2} and F_{k-3}, as long as there are enough elements.
- The sum of two consecutive elements is always one of these values: F_{k-1}, F_k or F_{k+1}. Note that, for the first four elements, we have: $F_k + F_{k-1} = F_{k+1}$; $F_{k-1} + F_{k-2} = F_k$; $F_{k-2} + F_{k-3} = F_{k-1}$.
- If the sum of elements in positions n and $n+1$ was F_{k-1} or F_{k+1}, then the sum of elements in positions $n+1$ and $n+2$ is always F_k. Therefore the element at position $n+2$ can be uniquely determined.
- If the sum of elements in positions n and $n+1$ was F_k, then the sum of elements in positions $n+1$ and $n+2$ might be either F_{k-1} or F_{k+1}. In some particular cases (like $F_k = 55$) there are two solutions; to guarantee that the solution corresponds to the "normal" pattern, proceed like this to choose the element at $n+2$: first, try to fulfill a sum of F_{k+1}. If the element you would need has already been used, then fulfill a sum of F_{k-1}.
- The two rules above allow to choose an element at every position of the sequence, so just iterate until you get to the end od the sequence.

Now, a first approach would be trying to predict the cases where the next sum after F_k is F_{k+1} or F_{k-1}, and use it to find the valid value at any arbitrary position. This does look feasible, but I found another way. To see it, let's look at the first series:

$F_k = 3$	3	2	1													
$F_k = 5$	5	3	2	1	4											
$F_k = 8$	8	5	3	2	6	7	1	4								
$F_k = 13$	13	8	5	3	10	11	2	6	7	1	12	9	4			
$F_k = 21$	21	13	8	5	16	18	3	10	11	2	19	15	6	7	14	...
$F_k = 34$	34	21	13	8	26	29	5	16	18	3	31	24	10	11	23	...

Now, if we look at the vertical strips, the first four obviously are series of Fibonacci numbers. However, if we look at sequences like $\{4, 6, 10, 16, 26, \ldots\}$ or $\{7, 11, 18, 29, \ldots\}$, one can see that they also verify the "Fibonacci" recurrence of having $a_{n+1} = a_n + a_{n-1}$. This gives us an interesting idea about generating a sequence from the previous ones, but there is a huge problem: with this scheme we can only generate values that fall within the indices of the series that we have initially, which are at most going to have something like 10^8 elements. So, the next question is: which values should we use to "fill" the small sequences, so that the result can be used to calculate the larger sequence completely? So I proceeded backwards, filling the table above with the values that should be there so that the summation holds. And a beautiful, simple pattern appears:

$F_k = 3$	3	2	1	**1**	**2**	**3**	**0**	**2**	**1**	**1**	**2**	**3**	**0**	**2**	**1**	
$F_k = 5$	5	3	2	1	4	**4**	**1**	**2**	**3**	**0**	**5**	**3**	**2**	**1**	**4**	
$F_k = 8$	8	5	3	2	6	7	1	4	**4**	**1**	**7**	**6**	**2**	**3**	**5**	
$F_k = 13$	13	8	5	3	10	11	2	6	7	1	12	9	4	**4**	**9**	
$F_k = 21$	21	13	8	5	16	18	3	10	11	2	19	15	6	7	14	...
$F_k = 34$	34	21	13	8	26	29	5	16	18	3	31	24	10	11	23	...

Well, that's interesting. So here are the basic observations:

- We can "extend" the sequence by reversing it, and then we can reverse it once again if needed.
- For the series of F_k, sometimes we need to replace the value of F_k with a 0. In fact: when k is even (so $F_k = 3$, or 8), we can reverse the sequence fully once, and then the first element of the third subsequence (i.e. the "normal" sequence after reversing twice) is 0. But when k is odd ($F_k = 5$, 13 and so on), the zero appears at the last element of the second subsequence (i.e. the first reversed sequence). This is not fully

apparent in the table above, but I verified it with (using Excel!) for values up to $F_k = 610$.

- The pattern eventually breaks down. Here are the first terms for the sequence of $F_k = 3$:

  $$3, 2, 1; 1, 2, 3; 0, 2, 1; 1, 2, 3; 0, 2, 1; 4, -1, 3; \ldots$$

  There is still some regularity of sorts in the "expanded" sequence, but I didn't bother formalising it. Still, the pattern definitely holds for at least thrice the length of the original sequence (i.e. exactly one 0 needed). And, since $\frac{F_{k+2}}{F_k} \approx \varphi^2 = \varphi + 1 < 3$, this is enough to generate the $k+2$-th sequence from the k-th and the $k+1$-th.

The nice thing is that I only need one element from the latest F_k, so an obvious recursive approach can work; and with memoisation to avoid recalculating values, this is actually very fast (a bit above 50 milliseconds). This also gave me the opportunity to go to town with my typical object oriented astronaut architectures, with polymorphism and so on. I believe that my code is still slower than that of most people commenting in the problem thread, but it's still well below a second so I'm not going to complain about it. I mean, my solution is very original compared to most people's, and it's still very fast. That's good.

By the way, since all of this is for the sequence from the *right* of the king, in order to use the index relative to the left one has to invert the index. With a zero-based index, the index I actually use is $99194853094755497 - 10^{16} = 89194853094755497$.

670. Colouring a Strip

Difficulty rating: 40 %.
Solution: 551055065. *Solved: Thu, 7 Oct 2021, 06:00.*
Math knowledge used: Recurrence equations.
Programming techniques used: binary exponentiation.

This looks like a standard dynamic programming problem, and in fact it is, but there is some difficulty in the fact that the states need to be chosen very carefully in order to find every possible arrangement, but at the same time some extra care is needed to avoid counting states twice. Fortunately, with some ordering this is doable; but this would be much more difficult to get right if there were more than two rows. Also, related to the colours,

you need to be super careful about the special case where the blocks are positioned at the very start of the strip and there is an additional colour available.

There are multiple ways to define the state space, but what I did was consider seven "base" states, each associated to a position x (the number of blocks behind the current state):

- A: the last block is a vertical one. The special case $A(0)$ is the initial state, which has special properties.
- B: the last block is "split", because there were horizontal blocks in the previous one. This means that the bound is vertical, but the only block you can add is a vertical one in order to transition to A.
- C: there is an extra horizontal block in the first row.
- D: there is an extra horizontal block in the second row.
- E: there are two extra horizontal blocks in the first row.
- F: there are two extra horizontal blocks in the second row.
- G: there are three extra horizontal blocks in the first row.

No, no need for state H. This is because of the way we are going to build the strip. In order to avoid duplicates, we will only add blocks to the bottom row when there are strictly fewer blocks than in the top one. This means that going from A to H by adding a three-block to the bottom is forbidden.

The set of available transitions is sizeable:

- From state $A(x)$, we can either add another vertical block (transitioning to $A(x+1)$) or a horizontal block on the top row. Depending on the size of this block, we will transition to either $C(x)$, $E(x)$ or $G(x)$. These are, by the way, the only transitions into a state with the same "index". This is not super problematic, but we need to take it into account when building the transition matrix.
- From state $B(x)$, the only allowed move is to add a vertical block to transition to $A(x+1)$.
- From state $C(x)$ we can only add blocks into the bottom row. Blocks with size 1, 2 and 3 transition respectively to $B(x+1)$, $D(x+1)$ and $F(x+1)$.

- Similarly, from state $D(x)$ we can add blocks into the top row, which will transition into states $B(x+1)$, $C(x+1)$ or $E(x+1)$.
- From state $E(x)$ we must add the new blocks into the bottom row. The transitions go to states $C(x+1)$, $B(x+2)$ and $D(x+2)$.
- State $F(x)$ transitions into $D(x+1)$, $B(x+2)$ and $C(x+2)$.
- And finally, from $G(x)$ you can transition into $E(x+1)$, $C(x+2)$ or $B(x+3)$.

Now we can revert these transitions and calculate the formulas for the amount of states. We need to take into account the different colours, which is why there are multipliers representing the amount of colours available. If the new block is in contact with a single additional block, there are three colours available, but if it's in contact with two blocks, we can only choose between two colours. There are special cases, marked with an asterisk, indicating that if we are in contact with the origin, we must add one colour (i.e., 2 is replaced with 3 and 3 is replaced with 4). The formulas are:

$$\begin{aligned}
A(x) &= 3^*A(x-1) + 2B(x-1), \\
B(x) &= 2^*C(x-1) + 2D(x-1) + 2^*E(x-2) + 2F(x-2) + 2^*G(x-3), \\
C(x) &= 3^*A(x) + 2D(x-1) + 2^*E(x-1) + 2F(x-2) + 2^*G(x-2), \\
D(x) &= 2^*C(x-1) + 2^*E(x-2) + 2F(x-1), \\
E(x) &= 3^*A(x) + 2D(x-1) + 2^*G(x-1), \\
F(x) &= 2^*C(x-1), \\
G(x) &= 3^*A(x).
\end{aligned}$$

With this we can define the space state. It's pretty clear that we are going to need all the values from $A(x)$ to $G(x)$, but we also need $E(x-1)$, $F(x-1)$, $G(x-1)$ and $G(x-2)$. That's a total of 11 states. In order to fill the transition matrix, we need to note that C, E and G depend on the *current* state A, not the previous one, so we start filling their rows with the value of the row for A, multiplied by the corresponding constant A. Also, considering that the last asterisk appears multiplying a $G(x-3)$, the initial vector must start with $x = 3$ if we want to have all the special cases covered.

So, first of all we init $A(0) = 1$, and we use it to init $C(0)$, $E(0)$ and $G(0)$ (all of them are equal to $3^* \cdot A(0) = 4 \cdot 1 = 4$). With these values we can carefully initialise all the states for $x = 1$, which are in turn used to calculate the values for $x = 2$, and finally for $x = 3$. The remaining values of x don't rely on the special case $x = 0$, so we can use a common transition for all of

them. We define the transition matrix as

$$M = \begin{pmatrix} 3 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 2 & 0 & 0 & 0 & 2 & 2 & 0 & 2 \\ 9 & 6 & 0 & 2 & 2 & 0 & 0 & 0 & 2 & 2 & 0 \\ 0 & 0 & 2 & 0 & 0 & 2 & 0 & 2 & 0 & 0 & 0 \\ 9 & 6 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

We also define the initial vector as

$$V_3 = [A(3), B(3), C(3), D(3), E(3), F(3), G(3), E(2), F(2), G(2), G(1)]^\top,$$

And the final state of the problem can be calculated as

$$V_n = M^{n-3} V_3.$$

Since the solution comes from states A and B, we can just add the two first values of V_n to get $A(n) + B(n)$, which is the solution of the problem.

The problem wants us to calculate the result for $n = 10^{16}$. With binary exponentiation, and applying modulus where needed, this can be calculated in a time of the order of $O(\log n)$, which is obviously ridiculously fast (less than 5 milliseconds).

672. One more one

Difficulty rating: 50 %.
Solution: 91627537. *Solved: Fri, 31 Dec 2021, 18:09.*
Math knowledge used: numerical bases, periodic sequences.
Programming techniques used: dynamic programming, binary exponentiation.

I'm fond of this problem, which was the last I finished on 2021. It's a problem based around number 7, whose index, $672 = 7 \cdot 96$, contains a 7. It was the 67th problem I solved on 2021, and after it, my total amount of problems solved reached $553 = 7 \cdot 79$. The solution contains two 7 digits and my run time is around 0,7 milliseconds.

Ok, now for the solution itself. The value of $g(n)$ is clearly related to the representation of n in base 7, and after some quick experiments, a general recursive formula for all positive n can be easily found:

- First, if n is a multiple of 7, divide at many times so that it becomes a non-multiple of 7. So 98 becomes 2, 119 becomes 17 and so on.
- If the result is 1, then $g(n) = 0$.
- Otherwise, let $d_a d_{a-1} d_{a-2} \dots d_1 d_0$ be the representation of the remaining number, $\frac{n}{7^k}$. Then, $g(n) = 1 + \sum_{i=0}^{a} (6 - a_i)$.

This is simple enough, but it's kind of problematic. I spent some time working around this formula, but in the end, I worked directly with the following properties, which come directly from the problem definition:

- $g(1) = 0$.
- $g(7n) = g(n)$.
- $g(7n + d) = 7 - d + g(n + 1)$ for $d \in [1 \dots 6]$.

For the real trick, we want to be able to sum values in bulk. So, what about this? We have $S(x)$, the sum of $g(n)$ for $n \in [1 \dots x]$, and we want to calculate $S(7x + d)$ for some digit d. It turns out that this is easy to do, and we only need three variables for each state: the value of the last number, x; the value of $g(x + 1)$; and the sum, $S(x) = \sum_{n=1}^{x} g(n)$. At each step, we have this state and an additional input digit, d. We can treat the different values as successions $\{x_n, g_n, s_n\}$, so that for each of them a_{n+1} represents the state for one additional digit. Then, for the value of x we can obviously do $x_{n+1} = 7x_n + d$; for the value of g (remember, $g_n = g(x_n + 1)$) we can do $g_{n+1} = g_n + 7 - d$, and for the sum we have $s_{n+1} = 21x_n + dg_n + 7s_n + T(d) - 6$, where $T(d)$ is a "triangular" function of sorts so that $T(0) = 0$, $T(1) = 7 - 1 = 6$, $T(2) = T(1) + 7 - 2 = 11$, and so on, until $T(6) = 21$. Of course, all of this is done modulo $P = 1117117717$. This is linear in the amount of digits, which is... wait. Which are the digits? We need to calculate $S\left(\frac{7^{10^9} - 1}{11}\right)$. As per Fermat's little theorem we know that $7^{10} \equiv 1 \pmod{}$

11), and therefore for every n multiple of 10 we will have $7^n \equiv 1 \pmod{11}$, so the number is guaranteed to be an integer since 10^9 is ostensibly a multiple of 10. The good news is that the quotient is going to have repeated digits. In particular, if we divide $7^{10} - 1$ by 11, and express the number in base 7, the result is the string of digits $S = 0431162355$, and the trailing 0 is important. This means that, given a multiple of 10, $n = 10m$, the result of $\dfrac{7^n - 1}{11}$ will be a repetition of this string of digits in base 7, m times. This can be easily proven. We start with

$$7^n - 1 = 7^{10m} - 1 = \left(7^{10} - 1\right)\left(1 + 7^{10} + 7^{20} + \ldots + 7^{10(m-1)}\right),$$

and dividing by 11 we get

$$\frac{7^n - 1}{11} = \frac{7^{10} - 1}{11}\left(1 + 7^{10} + 7^{20} + \ldots + 7^{10(m-1)}\right) = S\left(1 + 7^{10} + 7^{20} + \ldots + 7^{10(m-1)}\right).$$

Since the trailing zero makes S have exactly ten digits, we can see that the result will be a concatenation of m times the string.

Ok, we have the digits. We can apply the iteration directly, in a linear way, and get the result in $O(K)$ time, but we can do even better thanks to binary exponentiation, since the digits are obviously periodic. Since the recursion is not "pure", because there are constants, we will use a state of four elements, with the fourth one representing a constant one. Just like when doing generic transforms in 3D graphics. We can use matrices to represent the transitions, with left multiplications for composition (that is, the composed transformation $a \oplus b$, consisting of applying first b and then a, comes from the matrix product $M_a M_b$). We define the state again as a tuple $(x, g(x+1), S(x), 1)$, and for each digit d we define the following transition matrix:

$$M_d = \begin{pmatrix} 7 & 0 & 0 & d \\ 0 & 1 & 0 & 6 - d \\ 21 & d & 7 & T(d) - 6 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The initial state comes from the first digit of S, which happens to be a 4. We start with the state $s_4 = (4, g(5), S(4), 1)$ and we apply two transformations:

- First we apply the "remaining" transformation, for the first 10 digits. This is $M_5 M_5 M_3 M_2 M_6 M_1 M_1 M_3$.

- Now we get the "full" transformation. We start with the 7^{10} transform, $M = M_5 M_5 M_3 M_2 M_6 M_1 M_1 M_3 M_4 M_0$. The transform we want to apply is $M^{\frac{K-10}{10}}$, where $K = 10^9$ is the problem input. This gets us the transformation matrix in $O(\log K)$ time.

Applying both transforms one after another (I multiplied the matrices into a single one, and then I multiplied that matrix times the initial state vector) we get the final state. The solution, $H(K)$, is the third element of the tuple. Thanks to the modulus we can stay in integer operations, so it's ridiculously fast; below a millisecond.

676. Matching Digit Sums

Difficulty rating: 50 %.
Solution: 3562668074339584. *Solved: Tue, 4 Jan 2022, 16:34.*
Math knowledge used: numerical bases.
Programming techniques used: dynamic programming.

There is room for speeding up my solution to this problem (not that 0,14 seconds is slow), but it requires writing a different algorithm for two separate cases (one where k is a multiple of l, and the general one), and the general algorithm is good enough, so I didn't bother. Both algorithms use iterations of the order $O(\log N)$, but one requires much fewer operations and can be done almost instantly, taking advantage of the fact that the only possible solutions are combinations of ones and zeroes, while the other requires an initialisation which is linear and takes more time. Since I needed the general one for the cases where k is not a multiple of n, I just used that one for all cases. There is also room for pruning, but again, I didn't bother because the current algorithm is quick already.

For each instance of the algorithm, the inputs are k, l (both taking several values) and N (fixed at 10^{16} for all the cases). The first thing we do is calculate a common base p, calculated as the least common multiple of k and l: $p = \dfrac{kl}{\gcd\{k, l\}}$. If k is a multiple of l, then $p = k$. Otherwise, it's bigger. The biggest case is $k = 6$, $l = 4$, resulting in $p = 12$. The order of the algorithm in the worst case is about $O\left(\text{máx}\left\{p2^p, \left(\log_p N\right)^2\right\}\right)$ for each case, which is on the order of about thousands of operations, although there are also multiple logarithmic operations. I used BigInteger because I need the result modulo 10^{16}, so working with longs is not easy.

From k, l and p we calculate the actual bases we will be working on: $K = 2^k$, $L = 2^L$ and $P = 2^p$. The trick is that P is the smallest base such

that one single digit of a number expressed on it corresponds exactly to an integer amount of digits in both base K and base L. Therefore P is the smallest base such that we can treat each digit separately, and then use an iterative approach (not sure if it could be classified as dynamic programming. Probably a stretch) to add numbers to the final result. The first thing we will do is to calculate the differences added by each digit. We start by creating an array from 0 to $P-1$ where each element $A[i]$ represents the difference between the sum of digits of the representation of i on bases K and L. This is calculated as the difference of two arrays, one per base, since we can calculate the sum of digits very quickly with an iterative process:

- Let B be the smaller base (either K or L), and P the bigger one.
- Initialise an array of size P, $A[]$. Use a sane programming language, one of those where the indices start at 0, since 0 is a valid digit.
- Set $A[0] = 0$.
- Iterate from $i = 0$ to $\frac{P}{B}$.
    - For each i, iterate from $j = 0$ to B.
        - For each pair (i, j), set $A[B \cdot i + j] = A[i] + j$.

As an added bonus, these arrays can be cached (using B and P, or b and p, as keys), because some of them will be reused at some point.

Now, from these arrays representing the "added difference" of each digit in base P, which I will treat as a function called $d(x)$ on the subsequent description, we will create a structure holding the cumulative sums. We need a basic data structure to hold both a count of numbers and their sum, which I named just `CountAndSum`, and then a `CountAndSumDistribution` object which is basically a wrapper over a map from integers (representing the digit difference) to `CountAndSum`s. So, from the array of differences I create a cumulative array of these `CountAndSumDistribution`s, so that at index i we have the summary of all the differences from 0 to i, included. We will also need to keep the initial diffs array for the final algorithm. Each of these big arrays needs about $O(P \log P)$ space, which is tiny compared to most project euler problems.

The real trick is that these `CountAndSumDistribution` objects can be convoluted with each other. Let $f(x)$ be the cumulative count and sum distribution of all the values up to x (note that $f(x)$ is a map from integer differences to count and sum of values). $f(P-1)$ is of particular interest, since convoluting it with itself yields $f(P^2-1)$ (although a factor of P must

be added during the convolution so that the sum is exact); convoluting this one with $f(P-1)$ data again will yield $f(P^3-1)$, and so on. In general, convolution between $f(k-1)$ and $f(P^n-1)$, adding a factor P^n to the first element, will yield $f(kP^n-1)$, and we will use this to our advantage. We can now define a full algorithm that receives N and all the count and sum distribution objects, and returns the expected result. The algorithm assumes $N \geq P$, so that N has at least two digits in base P.

- Given k and l, calculate p and $P = 2^p$.
- Calculate the digit differences and the cumulative count and sum distributions, as explained above.
- Get the representation of N in base P, $n_a n_{a-1} n_{a-2} \dots n_1 n_0$. Note that this assumes a proper representation with no trailing zeroes, so in particular $n_a \geq 1$.
- Initialise the algorithm by working on the first digit:
  - Use the method described above to calculate $f((n_a-1)P^a-1)$. Given this map, take the `CountAndSum` object associated with a difference of 0, and add it to the total.
  - Initialise a variable *diff* as the difference of digits $d(n_a)$.
  - Initialise also a *prefix*, equal to n_a.
- Now comes the main loop: iterate on descending order, from $i = a-1$ to $i = 1$:
  - If $n_i > 0$, use the convolution to calculate $f((n_i-1)P^i-1)$. Take the `CountAndSum` object associated with the diff equal to $-diff$ (keep in mind that this object might not exist). Now, from this object, we need a *shifted sum*: if the count is c and the sum is s, we need to add $s + prefix \cdot P^{i+1} \cdot c$ to the total. All this step should be skipped if $n_i = 0$, but the other updates for $diff$ and $prefix$ must still be done.
  - Set $diff \leftarrow diff + d(n_i)$ and $prefix \leftarrow prefix \cdot P + n_i$.
- The last step is basically the same as the main loop ones, but $diff$ and $prefix$ don't need to be updated any more, and we will use $f(n_0)$ (note that this is not (n_0-1), which is what we would use if we reused the loop's logic).

We need to call this algorithm 10 times for the different combinations of k and l, but it's fast even with BigIntegers because the longest operations are the initial difference array creations, which are of order $pP = p2^p$. Everything else is logarithmic.

A bug in the project euler page gave me the *Big Game Hunter* award after solving this, but it didn't actually register the award. Also, after solving this problem I no longer have any blank "square" on my board of solved problems. Fitting for a square numbered problem.

678. Fermat-like Equations

Difficulty rating: 55 %.

Solution: 1986065. *Solved: Thu, 13 Jan 2022, 02:26.*

Math knowledge used: sum of squares characterisation formula, sum of cubes characterisation formula, "first prime" Erathostenes sieve (for divisor generation).

Programming techniques used: none.

This problem can be solved easily by operating differently with each of the left hand exponents. There are three cases: the sum of squares, the sum of cubes, and the rest. It's worth noticing that there are only 10^6 cubes, $10^{4,5}$ fourth powers, $10^{3,6}$ fifth powers, and so on, so the total amount of powerful numbers to consider is a bit above 10^6. This means that iterating over all the powers is feasible, but operating over all the pairs of powers is too much. We can, however, iterate over pairs of powers of the same exponents for $e \geq 4$, since this gives us an $O\left(\sqrt{N}\right)$ scheme which is feasible for $N = 10^{18}$. For squares and cubes, we proceed by finding decompositions of each power.

For the sum of squares, we iterate over all the powers $m^e = n$ for $e \geq 3$. This means that the range of m is $\left[1, \sqrt[3]{N}\right]$. We can use a standard prime sieve to get the decomposition of m, and then iterate over all the powers required, i.e., in the range $[3, \lfloor \log_m N \rfloor]$. If m has any prime factor of the form $4k + 3$ with an odd exponent, we need to consider only even values in that range; otherwise, all values must be considered. In any of the valid cases, we can use the typical formula: if $e_1, e_2, \ldots, e_k$ are the exponents of the $4k + 1$ prime factors of m, then we can express $n = m^e$ as a sum of squares in this many ways:

$$\left\lfloor \frac{1}{2} \prod_{i=1}^{k} (e_i \cdot e + 1) \right\rfloor.$$

Some numbers will appear twice or more during this step, and that's OK, we need to count them as many times as they can be expressed in terms of a power of a natural number with exponent greater than 2. Theoretically we should apply a slightly different formula depending on the exponent of 2 in the prime decomposition of m (and n), but since neither $a^e + 0^e$ nor $a^e + a^e$ are valid decompositions, the formula above counts exactly all the cases we want. Thus it becomes a matter of iterating over values of m and exponents, and applying this formula.

For the sum of cubes, there is an interesting paper called *Characterizing the Sum of Two Cubes*, from Kevin A. Broughan, that gives a simple and workable formula, which is actually surprisingly easy to derive. Given a number n, let k be a divisor in the range $\left(\sqrt[3]{n}, \sqrt[3]{4n}\right)$ (we want an open range to prevent, again, the cases $a^e + 0^e$ and $a^e + a^e$). Then calculate $l = \dfrac{1}{3}\left(k^2 - \dfrac{n}{k}\right)$. If l is an integer, and if $k^2 - 4l$ is a perfect square, then we can express n as $x_1^3 + x_2^3$, where x_1 and x_2 are the solutions of the equation $x^2 - kx - l = 0$, which are integer since the discriminant $k^2 - 4l$ is a perfect square. This gives us an iterating scheme that looks a bit like the one for squares, but a bit different. We want to iterate over values of m in the range $\left[1, \sqrt[4]{N}\right]$ (no need to use cubes, since Fermat's last theorem guarantees that they don't exist. Although I did iterate over all the range of m, simply because I reused the same loop than the one for squares). So, we first create the prime decomposition of m, which we already have because of the squares step. Then, for every exponent e in the range $[4, \lfloor \log_m N \rfloor]$, we create the decomposition of m^e by multiplying all the exponents times e. This decomposition can be used to find all the divisors of $m^e = n$ in the range $\left(\sqrt[3]{n}, \sqrt[3]{4n}\right)$. We can iterate over each of them, calculate l, and check whether it verifies the conditions. If so, add one to the count of valid decompositions.

Finally, for all the other powers, we just use brute force. First of all, create a set with all the possible target powers. As explained above, this set will have somewhere above 10^6 elements, so it fits comfortably in memory. Then, for each exponent e in the range $[4, \lfloor \log_3 N \rfloor]$, check all the pairs of powers a^e and b^e with $a < b$, increasing a counter when the sum of the powers happen to be present in the set of all powers. We need to take into account the fact that some numbers can be expressed as powers in more than one way. This happens for just 7 numbers:

$$33^5 + 66^5 = 1291467969 = 33^6 = 1089^2;$$
$$289^4 + 578^4 = 118587876497 = 17^9 = 4913^3;$$
$$244^5 + 732^5 = 211027453382656 = 244^6 = 59536^3;$$

$$550^5 + 825^5 = 432510009765625 = 275^6 = 75625^3;$$
$$6724^4 + 20172^4 = 167619550409708032 = 82^9 = 551368^3;$$
$$18818^4 + 28227^4 = 760231058654565217 = 97^9 = 912673^3;$$
$$129^7 + 258^7 = 76686282021340161 = 129^8 = 16641^4.$$

Note that $66 = 2 \cdot 33$, $578 = 2 \cdot 289 = 2 \cdot 17^2$, $732 = 3 \cdot 244$, $550 = 2 \cdot 275$, $825 = 3 \cdot 275$, $20172 = 3 \cdot 6724 = 3 \cdot 82^2$, $18818 = 2 \cdot 97^2$, $28227 = 3 \cdot 97^2$ and $258 = 2 \cdot 129$. Also, the last 10 digits of 275^6 are the same last ten digits of 5^{10}. Yes, this includes trailing zeroes.

Decomposing the first one million natural numbers is kind of fast, but the subsequent brute force operation for powers greater than 3 is a bit slower. The run time is about 8 seconds; not ridiculous or $O(\log N)$ or anything like that, but still fast. In the problem thread there are people trying things for the fourth powers, but there is no magic formula like for the cubes; they mostly iterate over sums of squares and checks which ones happen to be fourth powers.

681. Maximal Area

Difficulty rating: 50 %.

Solution: 2611227421428. *Solved: Sun, 2 Jan 2022, 01:12.*

Math knowledge used: Cyclic quadrilaterals, Brahmagupta's formula, "first prime" Erathostenes sieve (for divisor generation).

Programming techniques used: none.

This problem is kind of tricky. I haven't found a way of making it faster, which is a bit infuriating because it seems like a different iteration scheme is just out there. But anything I try end being slower than my current solution. Which is not so slow (35 seconds), truth be told.

Let's start with the facts: a quadrilateral of lengths (a, b, c, d) has maximal area if and only if it's cyclic, that is, if their vertices lie on a circumference. This is kind of expected, although not so obvious to prove. Anyway, we are treating it like a fact, and the idea of the code is based on it. The next fact we are going to use is Brahmagupta's formula, which is a quadrilateral's homologous to Heron's formula. This formula says that, for a cyclic quadrilateral with sides (a, b, c, d), and with semiperimeter $s = \dfrac{a+b+c+d}{2}$, the area of the quadrilateral is

$$A = \sqrt{(s-a)(s-b)(s-c)(s-d)}.$$

Given that, the least inefficient scheme I've found consists of this: first, iterate over the area A. Now, find all the factors of A^2 (the fastest way to do this is decompose A using a prime sieve, duplicating all the prime exponents of such decomposition, and use this result to find all the divisors), and sort them (this is useful so that we can exit loops early when values get too big). We will iterate over factors e, $f \geq e$ and $g \geq f$, so that $d = s - e$, $c = s - f$ and $b = s - g$. We will then find the last value, $h = \dfrac{A^2}{efg}$, so that $a = s - h$. Of course, we can interrupt the iteration early if either ef or efg doesn't divide A^2. There are two additional conditions that must be satisfied for this iteration to be correct. First of all, we must have $h < e + f + g$. Otherwise we get $a < 0$, which is not possible. We also need that $e + f + g + h$ is even, because otherwise the semiperimeter is not an integer, so the side lengths aren't either. Now, given this result, we need to get the perimeter, $a+b+c+d$. However, note that

$$e + f + g + h = 2s \Rightarrow a + b + c + d = (s - h) + (s - g) + (s - f) + (s - e) =$$
$$= 4s - (e + f + g + h) = 4s - 2s = 2s.$$

So we can store the sum of $e + f + g + h$ and not use a, b, c or d explicitly at any point. To summarise, here's the full algorithm (I'm using a single nested loop for all the divisors, because latex is choking on the excessive enumeration level. The code actually has a for loop for e, another for f and a last one for g, aside from the one for the area):

1. Use an Erathostenes sieve to get, for each integer $n \in [2, L]$, a prime that divides n.
2. Init the result as 0.
3. Now, iterate over all the integer areas, $n \in [1, L]$.
    a) Get the prime decomposition of n using the sieved data. This should be very fast.
    b) From the prime decomposition of n, square the exponents to get the prime decomposition of n^2.
    c) From the prime decomposition of n^2, get a full list with all the divisors of n^2.
    d) Sort the list of divisors of n^2. We will iterate over this list.
        1) Let e be a divisor of n^2.

2) If $e > (n^2)^{1/4} = \sqrt{n}$, end the loop.
3) Let $F = \sqrt[3]{\frac{n^2}{e}}$, maximum value allowed for f.
4) And now the next nested loop, where we will iterate over divisors of n^2 that are greater or equal than e.
5) Let f be a divisor of n^2 so that $f \geq e$.
6) If $f > F$, end the loop.
7) If ef doesn't divide n^2, discard this value of f and move on to the next.
8) We can also discard f if $e + f$ is even and $\frac{n^2}{ef}$ is a multiple of 2 but not of 4. This is because this forces g and h to be one odd and one even, so the perimeter will always be odd. This saves about one second of run time.
9) Let $G = \sqrt{\frac{n^2}{ef}}$, maximum value allowed for g.
10) We finally start the innermost nested loop, for factors of n^2 greater than f.
11) Let g be a divisor of n^2 so that $g \geq f$.
12) If $g > G$, end the loop.
13) If efg doesn't divide n^2, discard this g and move on to the next.
14) Otherwise, let $h = \frac{n^2}{efg}$. By construction, $h \geq g$ (otherwise we would have $g > G$).
15) If $h \geq e + f + g$, discard this g and move on to the next.
16) Calculate the perimeter $s = e + f + g + h$.
17) If the perimeter is even, increase the result in s.

4. The value of the result variable is the solution to the problem.

I tried a smarter (in principle) approach that iterated over e, f, g, and then tried to iterate for values of h that always result in valid values. The idea is not bad, but at each step I needed to keep a count of "unmatched" primes that had to be present in h; this can be done with sets, but intersecting them is far too slow, and the combinations of values of e, f and g already exceed $2 \cdot 10^9$, so I was getting times around three minutes even before doing any calculation for h. No point following that trail, sadly. The best I got is what I described above, which is just a more or less dumb iteration with as much pruning as I could muster.

703. Circular Logic II

Difficulty rating: 45 %.
Solution: 843437991. *Solved: Thu, 1 Jul 2021, 03:26.*
Math knowledge used: Graph theory.
Programming techniques used: dynamic programming, tree traversal.

This problem is not so difficult, but you need to be able to shift the view of the problem. If you look at problem 209 (the one that was called just "Circular Logic"), there is a tiny difference in the definition, since the last bit uses $b_1 \oplus (b_2 \wedge b_3)$ instead of this problem's $b_1 \wedge (b_2 \oplus b_3)$. It turns out that this changes the structure of the problem, since in 209 you only get "pure" cycles, but here the cycles have "protrusions" with several entry points, requiring a more nuanced approach. We can still do a separation of the 2^N nodes into disjoint sets and evaluate them separately.

The gist of the approach is: consider the problem space as a directed graph with 2^N nodes. A node x is connected to another, y, if $f(x) = y$. Now, we can "colour" (i.e. label, but colours seem to be the classic approach for this kind of graph problems) each node with either *true* (1) or *false* (0), and we need to find all the colourings where there isn't any pair of nodes (x, y) where $f(x) = y$ and both nodes are labeled *true*. The graph is complicated, but there is only one edge per node, so the size stays manageable. About the shape of the graph, some preliminary analysis reveals that there are 17 disjoint subgraphs, and that each one contains a small cycle, with one or more entry points. The biggest element by far is a very ramified subgraph with 1048250 elements, which ends in a single element cycle with $x = 0$. Other subgraphs have a cycle of 5, 10 or 20 with a varying amount of entry points, and there is a two element cycle without entry points. The way to proceed is: calculate the base patterns for these cycles, and move upwards from the graph calculating all the possible values.

The solution can be implemented using three parts. First, we build the graph and extract the cycles. We can save the structure using a list of "entry points" so that from each of them there is a tree hanging, constructed with f^{-1}. That is, if a node x has a child y that's because $f(y) = x$. This means that any node has either 0, 1 or 2 children. This looks a bit daunting but it's actually simple. First, detect the cycles and the disjoint connected subgraphs. Create a tree for each node in the graph, and from each cycle, create an array of "entry points" and fill each i element with the tree of the i-th element of the cycle. Then, for every node x which is not part of a cycle, calculate $f(x) = y$ and then add the tree x as a child of the tree y.

Second, build the set of basic patterns for each cycle. Given a cycle of length N, we want to find all the strings of N bits so that no two adjacent

bits are both set to 1, taking into account that the first and the last bit must also verify this condition. We can build this using dynamic programming: for size 1, we have the strings 0 and 1. For size n, take the solutions from size $n-1$, and for each one of them, if the last bit was 0, the next one can be either 0 or 1, whereas if the last bit was 1, the next one must be 0. This gets the "open" cycles for size N. Remove those where both the first and last bit are 1, and the result is the set of "closed" cycles for size N. There are about φ^N and there appears to be a relationship with Lucas numbers. For $N = 20$, which is also the longest cycle length, this means 15127 patterns. Note that rotated patterns are considered different, i.e. 00101 is not the same as 01010, since the entry points are ordered. This is already well managed by this pattern finding algorithm.

And finally, we can evaluate the trees. We can evaluate them recursively, using a very simple approach. Each node will have a 0-value and a 1-value, defined as the amount of valid (i.e. no consecutive ones) colourings from the subtree stemming from this node where this node is coloured respectively as 0 or 1. We also define the full value as the sum of the 0-value and the 1-value. These values can be calculated like this:

- If a node doesn't have any child, then both the 0-value and the 1-value are 1.
- If a node has children, the 0-value is the product of the full values of all its children.
- If a node has children, the 1-value is the product of the 0-values of all its children.

In a single recursive pass we can calculate all the 0-values and 1-values of all the nodes. The trees are very ramified and not so deep, so a stack overflow is not likely to happen despite there being so many elements. And now, with all the values in place, we just need the values of the cycles. After the first stage we have a set of cycles where each cycle element has a tree, and we have already calculated the patterns, so we can proceed like this:

- Iterate over the list of subgraphs (each represented by the cycle with entry points), and for each of them, iterate over all the patterns with the cycle size.
- Given a pattern, calculate the product of all the bit-values for the cycle. That is, if a cycle has 3 elements and the pattern is 001, get the 0-value of the first element, the 0-value of the second element, and the 1-value of the third one. This results in the pattern-value of this pattern for this subgraph.

- Add all the pattern-values for each subgraph in order to get the full subgraph-value.
- Finally, multiply all the subgraph-values to get the full graph value, i.e. the solution of the problem.

This finishes in a bit more than 1 second, using longs with modding. The full BigInteger result has 224541 digits, and has 745905 bits. To my surprise, the run time for BigIntegers is just about 2,5 seconds.

707. Lights Out

Difficulty rating: 55 %.
Solution: 652907799. *Solved: Wed, 9 Jun 2021, 06:16.*
Math knowledge used: numerical base 2, block-tridiagonal matrices.
Programming techniques used: cycle finding, binary exponentiation.

This is one of the problems that made me think "this is not outrageously difficult, but it's still out of my league". Of course, at the time of the problem publication I had very little time of my own because of my Physics degree, but that finished many months ago, and now it was a matter of putting the time. I started looking at this problem again because, when looking for information on some other one (481 or 380 if I remember correctly), someone mentioned the structure of the Lights Out "adjacency" matrix (tridiagonal per blocks, where each "side" block is the identity matrix, and each "main" matrix is a tridiagonal matrix where all values from all the three diagonals are equal to 1). After giving some thought to it, I realised that the amount of valid combinations had to be 2^R where R is the rank of the matrix, but I wasn't getting right values; for example, for the 4×4 case, the rank of the matrix is 14, but the real result was 12. I then realised that the normal rank calculation wouldn't be enough, because in this game, $1+1 = 0$. So I resorted to calculations in $\mathbb{Z}_2$, which fortunately are very easy to do with BitSets, and I started getting the right results. Calculating the rank of a $(mn) \times (mn)$ was doable but only for relatively small values of m and n, so I created a structure to manage block-tridiagonal matrices and calculate their rank with operations of $O(mn^3)$ instead of $O(m^3n^3)$. This allows calculation for much bigger numbers, and in fact, having a fixed m, the calculation for $n+1$ can be computed as as differential from the calculation for n, at a cost of $O(mn^2)$. I didn't take advantage of this because I realised it too late, but it can speed up the calculations.

Once I have all that in place, I started experimenting with a fixed $m = 199$. After some analysis, it seemed that there was a cycle length of 120,

so that, calling $f(n) = F(199, n)$, I could do $f(n) = \left\lfloor \frac{n}{120} \right\rfloor f(120) + f(n\,\%120)$. This was true up to $n = 1000$ in my initial calculations, and it was enough to get an initial result. Too bad that it's wrong. True up to 1000 indeed. The first value for which it doesn't work is $n = 1024$. Does that make you think of powers of two? Think again. The problematic values are of the form $n = 1025x - 1$ for some x. For $x = 1$ the result happens to be a power of two, which will confuse you. Fortunately, after some analysis I found the complete pattern. First, we define:

$$g(x) = \begin{cases} 20 & \text{if } x \text{ is odd,} \\ 40 & \text{if } x \text{ is even but not a multiple of 4,} \\ 80 & \text{if } x \text{ is a multiple of 4 but not of 8,} \\ 160 & \text{if } x \text{ is a multiple of 8.} \end{cases}$$

No, the recurrence doesn't continue, it's still 160 if it's a multiple of any higher power of 2. Now, with that in mind, we just need to detect the cases where n is of the form $1025x - 1$, and in that case, subtract $g(x)$ to the value of f calculated with the recursive formula above. The full algorithm is as follows:

- Calculate the rank of the $\mathbb{Z}_2$ matrices of size $199 \times n$, $R(n)$.
- Calculate the fibonacci numbers required for this problem. Some are huge (more than 40 digits), but I still used BigInteger to avoid issues with modding.
- For each one of the Fibonacci numbers, f_i, calculate $q_i = \left\lfloor \frac{f_i}{120} \right\rfloor$ and $r_i = f_i\,\%120$.
- Calculate $R(f_i) = q_i R(120) + R(r_i)$.
- If f_i is of the form $f_i = 1025x - 1$, note the value of $g(x)$ and subtract it from $R(f_i)$. This results in the correct value for the matrix rank.
- After subtracting (if needed), calculate $2^{R(f_i)}\,\%M$. Use binary exponentiation. Also, use $R(f_i)\,\%\varphi(M)$, where $M = 10^9 + 7$ is the modulus, to reduce the amount of calculations.
- Add everything, do $\%M$ at the end.

The run time is about 6,3 seconds, which is a bit too much because I didn't use the optimised version of the tridiagonal block calculations, but it's still a decent time.

724. Drone Delivery

Difficulty rating: 30 %.

Solution: 18128250110. *Solved: Thu, 21 Jul 2022, 06:02.*

Math knowledge used: combinatorics, harmonic numbers, inclusion-exclusion.

Programming techniques used: none.

I find this problem kind of infuriating. I spent a ridiculous amount of time doing experiments until by chance I got a working formula, which is $O(n)$ and gives me the result in 0,1 seconds (it's kind of slow because real numbers are involved and I have to use doubles, not longs). Of course, I found in the problem thread that this is a known problem and the formula is indeed widely available if you know what to look for.

Here's what I did. Let's start working with a fixed N. The probability that at the instant x I only have touched elements from a subset of $\{1, N\}$ with exactly n elements (this includes the possibility that *not every element from that set has been touched*) is $\left(\frac{n}{N}\right)^x$. With a bit of inclusion-exclusion we can get the probability that at instant x we have touched every element in the subset, which is

$$Q_N(n, x) = \sum_{i=1}^{n} (-1)^{n-i} \binom{n}{i} \left(\frac{i}{N}\right)^x.$$

And with this we can calculate the probability that we touch the last of the N elements of the full set precisely at time x, which is:

$$P_N(x) = \sum_{i=1}^{N-1} (-1)^{N-i-1} \binom{N-1}{i} \left(\frac{i}{N}\right)^x.$$

Now, it's pretty straightforward to see that the total distance is always the triangular number $T(x)$, so the expected value can be calculated as

$$E(N) = E\left[\frac{x^2+x}{2N}\right] = \sum_{x=N}^{\infty} \sum_{i=1}^{N-1} (-1)^{N-i-1} \binom{N-1}{i} \left(\frac{i}{N}\right)^x \frac{x^2+x}{2N}.$$

With the usual technique of reordering the summations, and with a few lines of Mathematica code, we can calculate a "simple" formula (I mean, it fits in a single line), which appears to be $O(n)$, for $E(n)$:

$$\sum_{i=1}^{N-1} (-1)^{N-i-1} \binom{N-1}{i} \left(\frac{i}{N}\right)^N \frac{N^5 - 2iN^4 + N^4 + i^2N^3 - i^2N^2 + 2iN^2}{2(N-i)^3 i}.$$

Unfortunately, there is no fucking way in hell that this will work in a reasonable time, which is why this problem is a 30 % one instead of a 10 %. The reason is that, although the result stabilises into something smallish (one or two orders of magnitude above N), the summation has huge terms which cancel each other because of the signs. And they don't cancel each other exactly, in the sense that for some values i and j the terms are equal except in sign, or shit like that. We really would need the full calculation. So we would need to calculate N terms, and we need all the digits (which is around a low multiple of N itself) in order to exactly calculate the small final value. This works up to around $N = 1000$, but beyond that, the run time becomes infeasible pretty quickly.

Now, cue days and DAYS of experiments with Mathematica using summations, and unfruitful attempts to use relatively advanced techniques like binomial inversion, until by chance I noticed (thanks to the numerator 137 appearing for $N = 5$) that

$$E\left[\frac{x}{N}\right] = H_N = \sum_{i=1}^{N} \frac{1}{i}.$$

Harmonic numbers? I didn't expect them, but hey, this looks like a promising clue. Cue two more days of experiments, and FINALLY, I notice that the succession

$$a_N = \frac{(N!)^2}{N} \left(E\left[\frac{x^2}{N^2}\right] - E\left[\frac{x}{N}\right]^2\right)$$

is a series of integers and it appears in OEIS as A060944, without any other additional description than the formula

$$(n!)^2 \sum_{k=1}^{n} \sum_{j=1}^{k} \frac{1}{j^2}.$$

Jackpot. FINALLY. The terms are displaced, though (OEIS A060944 for a value n matches the term a_{n+1} of my succession, not a_n). This finally got me

a feasible formula:

$$E(n) = \frac{1}{2}\left(\sum_{i=1}^{n-1}\sum_{j=1}^{i}\frac{1}{j^2} + n\left(\sum_{i=1}^{n}\frac{1}{i}\right)^2 + \sum_{i=1}^{n}\frac{1}{i}\right).$$

At each iteration I calculate the harmonic numbers $H(n,1)$ and $H(n,2)$, and add $H(n-1,2)$ to the result. Then at the end I add $nH(n,1)^2 + H(n,1)$, divide the result by two, and get the nearest integer. 0,1 fucking seconds. Then I read the problem thread and I learn that the numerators and denominators of $E(n)$ are actually in OEIS as well (how did I miss that???), and this is a well known problem called "coupon collectors problem". Oh well.

756. Approximating a Sum

Difficulty rating: 30 %.
Solution: 607238.610661. *Solved: Sun, 2 May 2021, 10:15.*
Math knowledge used: combinatorics, totient number.
Programming techniques used: none.

This is one of these problems where the theoretical analysis is not very complicated, but you need to be very careful about which data types to use and how to order the operations. The programming part is more difficult to get right than the mathematical part.

In any case, the mathematical analysis comes first. There are several ways to tackle this problem, but the one I chose is: the proposed statistical scheme divides the space of the sum in M segments (actually, $M+1$, with a possibly empty $(X_m, N]$ interval at the end), but these segments are finite and there aren't that many of them, so we can calculate the contribution of each segment multiplied by the proportion of total divisions in which this segment is present. This concept might sound convoluted, but it's very successful at reducing the problem to a simple, manageable formula. The whole statistical space consists of choosing M points beteween the N possible ones, and there are two kinds of segments:

- Segments of the form $(0, i]$. These segments are characterised by having a point exactly at i, and then having the other $M-1$ in the $[i+1, N]$ interval. Therefore, this segment appears in exactly $\binom{N-i}{M-1}$ cases. They contribute a total of $f(i)\cdot i$ to the sum.
- Segments of the form $(i-d, i]$ for some $i \geq 2$ and $d < i$. Se we have one point in $i-d$, other one in i, and the remaining $M-2$ are in the

set $[1, i-d-1] \cup [i+1, N]$. This means that this segment appears in $\binom{N-1-d}{M-2}$ cases, and its contribution to the sum is $f(i) \cdot d$.

We also need to consider the total amount of distributions of points, which is naturally $\binom{N}{M}$. All of this means that the average value of the statistical sum can be computed using the following formula:

$$\overline{S^*} = \binom{N}{M}^{-1} \sum_{i=1}^{N} \left(f(i) \cdot i \cdot \binom{N-i}{M-1} + \sum_{d=1}^{i-1} f(i) \cdot d \cdot \binom{N-1-d}{M-2} \right).$$

The real trick comes here: the binomial numbers are simply huge and there is no way to manage them, not even with BigInteger. But we expect that they will cancel each other, so, instead of keeping the total count as a common factor, we introduce it in the summations:

$$\overline{S^*} = \sum_{i=1}^{N} \left(f(i) \cdot i \cdot \binom{N}{M}^{-1} \binom{N-i}{M-1} + \sum_{d=1}^{i-1} f(i) \cdot d \cdot \binom{N}{M}^{-1} \binom{N-1-d}{M-2} \right).$$

This seems like overkill, but it makes the calculations much more numerically stable, because the enormousness of each combinatorial factor is offset by the other one. We can reduce their complexity, but before that, we are going to do some analysis on the summation bounds. The first thing to do is to separate the result into two separate summation, $\overline{S^*} = A + B$, where:

$$\begin{aligned} A &= \sum_{i=1}^{N} f(i) \cdot i \cdot \binom{N}{M}^{-1} \binom{N-i}{M-1}, \\ B &= \sum_{i=1}^{N} \sum_{d=1}^{i-1} f(i) \cdot d \cdot \binom{N}{M}^{-1} \binom{N-1-d}{M-2}. \end{aligned}$$

And now we will do one of the most common operations in project Euler: reverse the summation order, which makes a lot of sense because the combinatorial term depends on d and not on i. Therefore,

$$B = \sum_{d=1}^{N-1} \sum_{i=d+1}^{n} f(i) \cdot d \cdot \binom{N}{M}^{-1} \binom{N-1-d}{M-2}.$$

This looks good, but it can look even better, by isolating the $f(i)$ term:

$$B = \sum_{d=1}^{N-1} d \cdot \binom{N}{M}^{-1} \binom{N-1-d}{M-2} \sum_{i=d+1}^{N} f(i).$$

We can immediately see a way to reduce the complexity from $O(n^2)$ to $O(n)$: by storing an array with the accumulated sums, $F(i) = \sum_{j=0}^{i} f(i)$, we reduce B to

$$B = \sum_{d=1}^{N-1} d \cdot \binom{N}{M}^{-1} \binom{N-1-d}{M-2} (F(N) - F(d))$$

This is starting to look really good, and before the last simplification, we are going to reduce these summation indices a bit. In the case of A, we notice that $\binom{N-1}{M-1}$ only makes sense when $N-i \geq M-1$; otherwise it will always be 0. This condition can be rearranged into $i \leq N-M+1$, so we can reduce the summation limit to this value. In the case of B, we have a similar observation: we only need the cases where $N-1-d \leq M-2 \Rightarrow d \leq N-M+1$. So, very conveniently, both upper bounds are the same! We now have

$$\begin{aligned} A &= \sum_{i=1}^{N-M+1} f(i) \cdot i \cdot \binom{N}{M}^{-1} \binom{N-i}{M-1}, \\ B &= \sum_{d=1}^{N-M+1} d \cdot \binom{N}{M}^{-1} \binom{N-1-d}{M-2} (F(N) - F(d)); \end{aligned}$$

and now it's finally time to work on these combinatorial numbers. First, we can do this:

$$\frac{\binom{N-i}{M-1}}{\binom{N}{M}} = \frac{\dfrac{(N-i)!}{(M-1)!\,(N-M+i-1)!}}{\dfrac{N!}{M!\,(N-M)!}} = \frac{(N-i)!}{N!} \frac{M!}{(M-1)!} \frac{(N-M)!}{(N-M+i-1)!}.$$

And then we do this:

$$\frac{\binom{N-1-d}{M-2}}{\binom{N}{M}} = \frac{\dfrac{(N-1-d)!}{(M-2)!\,(N-M+d-1)!}}{\dfrac{N!}{M!\,(N-M)!}} = \frac{(N-1-d)!}{N!} \frac{M!}{(M-2)!} \frac{(N-M)!}{(N-M+d-1)!}.$$

These formulas are very convenient (even having terms that can be reduced to constants), because each term can be calculated from the previous one just by adding a small term that depends on i or d. We finally have a workable scheme. First of all, we are going to rename the summation index of B as i instead of d, so that both summation indices are homogeneous:

$$\begin{aligned} A &= \sum_{i=1}^{N-M+1} f(i) \cdot i \cdot \binom{N}{M}^{-1} \binom{N-i}{M-1}, \\ B &= \sum_{i=1}^{N-M+1} i \cdot \binom{N}{M}^{-1} \binom{N-1-i}{M-2} (F(N) - F(i)); \end{aligned}$$

now, we define two recurrences, $C_A(i)$ and $C_B(i)$, in this way:

$$\begin{aligned} C_A(1) &= \frac{M}{N}, \\ C_A(i) &= \frac{N-M-i+2}{N-i+1} \cdot C_A(i-1); \\ C_B(1) &= \frac{M^2-M}{N^2-N}, \\ C_B(i) &= \frac{N-M-i+2}{N-i} \cdot C_B(i-1). \end{aligned}$$

So calculating $C_A(i)$ and $C_B(i)$ is a constant time operation, provided that we already have the values for $i-1$. And then,

$$\begin{aligned} A &= \sum_{i=1}^{N-M+1} f(i) \cdot i \cdot C_A(i), \\ B &= \sum_{i=1}^{N-M+1} i \cdot (F(N) - F(i)) \cdot C_B(i), \\ \overline{S^*} &= A + B. \end{aligned}$$

For extra efficiency, we can calculate A and B using the same loop, and even better, extracting the common factor i and multiplying it after having added the rest of the terms:

$$\overline{S^*} = \sum_{i=1}^{N-M+1} i \cdot [f(i) C_A(i) + (F(N) - F(i)) C_B(i)].$$

With a precalculated array of $f(i)$ (and $F(i)$ as well), and being able to calculate $C_A(i)$ and $C_B(i)$ with just a multiplication and a division applied to the respective $i-1$ term, this is a perfectly workable $O(n)$ formula. However, this still doesn't tell the whole story: there are non-integer divisions here (very much, since the result is obviously not an integer!). Rationals are not usable here, because they will grow enormously, so the first option is to use doubles. However, this won't work. They don't have enough precision for this problem. So, in order to perform the calculations for this problem, **the whole calculation must be done using BigDecimal**. Fortunately, no complex or weird operations are needed; the most difficult thing we are going to do to this numbers is divide quantities. Special care must be taken to ensure that there is enough precision. 30 digits of precision are not enough, and I had to resort to 50. However, with this data type, calculations are finally good enough, and I got the right result.

As a final note, take into account that the complicated calculation is done for $\overline{S^*}$, but the complete result of the problem is $F(N)-\overline{S^*}$. Of course, $F(N)$ has already been calculated during the previous stage.

For something with so many operations (ok, on the order of 10^7 terms, so not *that* many, maybe), and such a hyper-slow numeric data type, the run time is not bad at all: a bit below 7,2 seconds.

757. Stealthy Numbers

Difficulty rating: 10 %.
Solution: 75737353. *Solved: Sun, 16 May 2021, 10:24.*
Math knowledge used: basic algebra.
Programming techniques used: none.

This is one of these problems where the math involved is really simple, and all the work is done by exploiting a pattern that is simple but it needs to be discovered. Well, at least that's how I did it. In the problem thread there are alternate approaches that rely on the fact (which I didn't know when I solved the problem) that so-called *stealthy numbers* are always of the form $N = x(x+1)\,y(y+1)$. These are called bi-pronic numbers, because the product of two consecutive numbers is called a pronic number.

My approach is different than the most common one. Instead of following this formula, I transformed the original tuple $\{a, b, c, d\}$ into a different set of numbers, more appropriate for my calculations:

- From a and b I derive their half-sum and half-difference: $\alpha = \dfrac{a+b}{2}$, $\beta = \dfrac{a-b}{2}$. This is so that $\alpha^2 - \beta^2 = ab = N$.

- Similarly, from c and d I derive $\gamma = \dfrac{c+d}{2}$ and $\delta = \dfrac{c-d}{2}$. Therefore, $\gamma^2 - \delta^2 = cd = N$.

Since $a+b$ and $c+d$ are consecutive numbers, it's clear that one of them must be odd and the other one must be even. Therefore, we will have two pairs, $\{\alpha, \beta\}$ and $\{\gamma, \delta\}$, so that both elements are either integers or half integers, and one of the pairs will be of each type. Now, after analysing the pattern of the first results for the test case for 10^6, some general properties emerge:

- It's always the case that $\alpha - \gamma = 0{,}5$. This removes one degree of freedom.

- Similarly, it's always the case that $\beta > \delta$. However, the difference is not constant. In fact, it can be seen that, for every β, we will find values of δ ranging from $\beta - 1{,}5$ down to 0 in decrements of 1. In fact, if β is a half integer, which means that δ is an integer, the case $\delta = 0$ must be explicitly treated, just like the rest of the range.

- There isn't a clear pattern about whether each pair is composed of integers or half integers. Both cases ($\{\alpha, \beta\}$ integers and $\{\gamma, \delta\}$ half integers, and vice versa) happen.

These conditions are enough to generate a two-dimensional iteration scheme. Considering that $\gamma = \alpha - 0{,}5$, the condition $\alpha^2 - \beta^2 = N = \gamma^2 - \delta^2$ can be transformed into

$$\alpha^2 - \beta^2 = \left(\alpha - \frac{1}{2}\right)^2 - \delta^2 \Rightarrow \alpha = \beta^2 - \delta^2 + \frac{1}{4}.$$

Note the consistency: if β is a half integer and δ is an integer, the right hand of the formula yields a half integer; conversely, if β is an integer and δ is a half integer, the result is an integer. So $\alpha \pm \beta$ is an integer.

This formula makes it also quite clear that, with any fixed β, smaller values of δ yield higher values of α, and therefore higher values of $N = \alpha^2 - \beta^2$. Then, what makes the most sense is to iterate first over increasing values of β, then over decreasing values of δ, so that the inner loop ends prematurely if we find that the resulting N is greater than the limit.

There are still two issues to be taken into account. The first one is that, if we iterate in this way, we will find repeated values of N. No problem, we can just use a set. Some people in the problem thread complain about sets being too slow (albeit their algorithm is different to mine), and I suspect that the cause is a poorly chosen structure, i.e., either a tree (like C++'s default `set`) or a hash table where the amount of buckets hasn't taken into account the fact that we expect to have an amount of elements of the order of tens of millions. The second problem is that, if we want to keep the code clean and based on integers, we will need two separate loops: one for the case where $\{\alpha, \beta\}$ are integers and another one for the case where they are half integers. This is kind of ugly, but it avoids all the pitfalls of floating point numbers. I considered using a custom type supporting both integers and half integers, but I thought that a structure where the branching paths are always different from one execution to the next would not be very optimisable by the processor's internal Tomasulo magic. So, two separate loops it is. The outline of the algorithm is this:

- Initialise the set of "found numbers" as an empty set. If possible, give indications that the size will be somewhere between 10^7 and 10^8.
- First loop: integers $\{\alpha, \beta\}$ and half integers $\{\gamma, \delta\}$. Iterate over $\beta = 2$, increasing in values of 1.
- In this first loop we will use a variable $\delta' = \delta - 0{,}5$, whose initial values is $\beta - 2$ (i.e. $\delta_0 = \beta - 1{,}5$) and decreasing.
- $\alpha = \beta^2 - \delta^2 + \dfrac{1}{4} = \beta^2 - \delta^2 - \delta$, and $N = \alpha^2 - \beta^2$.
- If N is still inside the bounds, add it to the set of "found numbers". Otherwise, end the inner loop.
- If the inner loop did not yield any valid value of N, end the outer loop. Otherwise, increase β and continue.
- The second loop is similar to the first one. We will consider $\beta' = \beta - 0{,}5$ as the iterating variable.
- Now, the inner loop iterates from $\delta = \beta - 1{,}5 = \beta' - 1$ downwards to 0.
- $\alpha = \beta^2 - \delta^2 + \dfrac{1}{4}$. We will consider an integer value, $\alpha' = \alpha - 0{,}5$, so that $\alpha' = \beta'^2 + \beta - \delta^2$. With this value, we have $N = \alpha'^2 + \alpha' - (\beta'^2 + \beta)$.

- Again, if N is still inside the bounds we add it to the "found numbers" and otherwise we terminate the inner loop.
- The outer loop finishes when an execution of the inner loop did not found any valid value.
- After both loops have finished, the solution of the problem is the amount of elements present in the set.

The run time is reasonably low, about 7,1 seconds. People in the problem thread use a different approach (generating the bi-pronic numbers), usually with higher run times.

I like that the solution is a funny number with repeated digits, the first ones being precisely 757, the index of the problem.

758. Buckets of Water

Difficulty rating: 50 %.

Solution: 331196954. *Solved: Sun, 30 May 2021, 16:14.*

Math knowledge used: Bézout's Lemma, Euclid algorithm, modular inverses.

Programming techniques used: binary exponentiation.

Very remarkable problem. I'm kind of proud of having solved it in the day it got out, since it's one of the complicated ones and I expect it to be rated as something between 50 and 60 %. Surprising, considering that the problem deals with such well known concepts as Euclid's algorithm and the "three buckets" problem. At first, I didn't even expect to be able to solve it.

This problem has two parts, and both of them include a mix of pattern finding and actual math reasoning. The first part is finding a general expression for $P(a, b)$, and the second part is getting this expression to work with numbers of the form $2^x - 1$ where x nears 10^{15} (BigIntegers won't work since these numbers don't fit in memory). I found the second part more difficult, because taking modulus removes some important information that you need, and you need to find a way to recover it.

First part: the pattern. There is a very helpful diagram somewhere in Wolfram Alpha that shows the search space as a skewed parallelogram where the possible states are points in the boundary, and each "pouring operation" is a movement through its interior following certain set of lines. After examining this image, it's clear that there is certain periodicity in the numbers you get (so reaching 1 is a matter of time if S and M are coprime), and that there are two routes but one is always shorter. With some mathematical intuition

here, and looking at the cycles, you can guess that, with Bezout's Lemma at hand, having $aS + bM = 1$, or at least a variant like $aS = bM + 1$ or $aM = bS + 1$ with both a and b positive (this is unique!), the solution is directly related to a and b. Indeed it is: the search space has size $2\,(S + M)$, so it's easy to bruteforce small values (yes, that includes the sample case $P\,(1234, 4321)$); and it's easy to see that the result is indeed

$$P\,(S, M) = 2\,(a + b - 1)\,. \tag{4}$$

Second part: actually running the pattern through very big numbers. It's pretty clear that modular arithmetic is going to be involved from the start, but it's not easy seeing how. Bézout's Lemma's values a and b can be found using the extended Euclid algorithm, but the extended Euclid algorithm only works if the numbers fit in memory, which definitely do not in this problem. Fortunately, every number involved follows the formula $2^x - 1$ for some x which is moderately big, but which will still fit in a long. This suggests running a variant of Euclid's algorithm using exponents, but since we want to get Bézout's lemma's coefficients, the standard algorithm won't work. Reverse engineering time! Let's analyse the inner workings of the extended Euclid algorithm so that we can modify it.

The extended Euclid algorithm works with a simple premise. Given the inputs a and b, there are successions r_n, s_n and t_n, so that at every iteration it holds that $r_n = as_n + bt_n$. We perform the typical Euclid's algorithm modulus operations on r_n until we arrive at $r_n = 1$ for some n; since we have been updating s_n and t_n accordingly, their values at the end of the algorithm are the coefficients we want. The update is actually very simple, since at each time we have divide r_{n-2} by r_{n-1}, so that $r_n = r_{n-2} - qr_{n-1}$ for some quotient q, and the updates to s_n and t_n follow the same structure:

$$\begin{aligned} s_n &= s_{n-2} - qs_{n-1}, \\ t_n &= t_{n-2} - qt_{n-1}. \end{aligned}$$

Now, let's suppose that we want to work with exponents. It's tricky, because for r_n we want to use exponents, but for s_n and t_n we need the actual coefficients. The good news is that for r_n we can use the same formula: we divide the last two elements, so that r_n is the remainder, and then we store the quotient to update s_n and t_n. But this update is the trickier one. First of all, what are we achieving with these modular operations on the $\{r_n\}$ succession? Well: when we store the exponent r_n, what we actually mean is that we have the number $2^{r_n} - 1$, which happens to be a streak of r_n ones. So it's

easy to see how we "cut" this streak in pieces each time we do a division. But the actual, numerical update to the base values can't be described just by the usual operation with q. Instead, we need to pay attention to what we are doing. First of all, assume that the inputs verify $a > b$. This guarantees that the quotient will always be at least one, and it saves some silly special case handling later. Now, what we are doing is: remove q appearances of r_{n-1} from r_{n-2}, leaving the reminder r_n. Let's expand on this:

$$r_{n-2} = qr_{n-1} + r_n \Rightarrow r_{n-2} = r_{n-1} + r_{n-1} + \ldots + r_{n-1} + r_n.$$

In the standard division, each r_{n-1} block (i.e. each time we subtract r_{n-1} from r_n) has the same meaning, but here, each "block" is at a different position in the streak of ones, so we have:

$$\begin{aligned} 2^{r_{n-2}} - 1 = \left(2^{r_{n-2}} - 2^{r_{n-2}-r_{n-1}}\right) + \left(2^{r_{n-2}-r_{n-1}} - 2^{r_{n-2}-2r_{n-1}}\right) + \ldots + \\ \ldots + \left(2^{r_{n-2}-(q-1)r_{n-1}} - 2^{r_{n-2}-qr_{n-1}}\right) + \left(2^{r_{n-2}-qr_{n-1}} - 1\right). \end{aligned}$$

Each one of the blocks is a separate streak of ones with width r_{n-1}, save for the last one. And of course, we have $r_{n-2} - qr_{n-1} = r_n$. The blocks are equally spaced, so we can use this formula, which is the base of the modified Euclid algorithm we need for this problem (note how we use the relationship between the r_n succession values to express the exponents in terms of r_{n-1} and r_n in the right side, discarding r_{n-2}):

$$2^{r_{n-2}} - 1 = 2^{r_n} - 1 + \sum_{i=0}^{q-1} \left(2^{r_n+(i+1)r_{n-1}} - 2^{r_n+ir_{n-1}}\right).$$

Inside the summatory, we have

$$2^{r_n+(i+1)r_{n-1}} - 2^{r_n+ir_{n-1}} = 2^{r_n}\left(2^{(i+1)r_{n-1}} - 2^{ir_{n-1}}\right) = 2^{r_n}\left(2^{r_{n-1}} - 1\right)2^{ir_{n-1}},$$

so we can express the original relation as

$$2^{r_{n-2}} - 1 = 2^{r_n} - 1 + 2^{r_n}\left(2^{r_{n-1}} - 1\right)\sum_{i=0}^{q-1} 2^{ir_{n-1}} = 2^{r_n} - 1 + 2^{r_n}\left(2^{r_{n-1}} - 1\right)\frac{2^{qr_{n-1}} - 1}{2^{r_{n-1}} - 1}.$$

This formula is, finally, enough to expand the Euclid algorithm. If we define, at each iteration,

$$Q = 2^{r_n}\frac{2^{qr_{n-1}} - 1}{2^{r_{n-1}} - 1},$$

then we can say that

$$2^{r_n} - 1 = (2^{r_{n-2}} - 1) - Q\,(2^{r_{n-1}} - 1)\,,$$

so we finally have a way to use the standard Euclid algorithm update operation when r_n is treated as an exponent, but s_n and t_n are treated "normally". The new, modified algorithm is this:

- Operate with the r_n succession normally, getting the quotient q as usual at each time.
- Calculate Q from q and from the known values of $\{r_n\}$. Of course, this is calculated using modular arithmetic. Some implementation details: binary exponentiation is suggested for the powers of two; the fraction can be ignored if $q = 1$ (special care should be needed if $q = 0$, but as we've seen, this won't happen if the inputs are such that $a > b$); if $q > 1$ and the division is needed, what we do is multiplying times the modular inverse of the denominator; finally, caching is suggested for the powers and the inverses, although it saves very little time and the algorithm is super fast anyway.
- We can update: $s_n = s_{n-2} - Qs_{n-1}$ and $t_n = t_{n-2} - Qt_{n-1}$.
- The rest of the algorithm is the same as the standard Euclid algorithm. We know that in this problem the inputs will always be coprime, so finish when $r_n = 1$.

The problem is almost finished now, but there is something missing. Ok, we have the coefficients s_n and t_n so that $s_nM+t_nS = 1$, but now what? With small numbers, we would rewrite as either $s_nM = -t_nS+1$ or $t_nS = -s_nM+1$ (resp. if t_n is negative, or if s_n is), so the result is either $2\,(s_n - t_n - 1)$ or $2\,(t_n - s_n - 1)$, but since we have been doing modular arithmetic, these are random-looking numbers and signs aren't there. Fortunately, we can examine the original Euclid algorithm for some more insight. It's pretty clear that the signs are alternating: save for the initialisation, when $s_1 = 1$, $s_2 = 0$, $t_1 = 0$, $t_2 = 1$, in any subsequent iteration (with these indices, the first iteration would calculate r_3, s_3 and t_3), the set of $\{s_n, t_n\}$ will have exactly one positive and one negative number. In fact, it can be easily seen that in odd iterations we will have a positive s_n and a negative t_n, while in even iterations s_n will be negative, and t_n will be positive. This is enough to calculate the sign we need. Finally, an algorithm for solving the whole problem can be outlined:

- Generate all the primes smaller than 1000, and store their fifth powers in an array.

- Iterate over pairs of indices in the array (i.e. a nested loop).
- For each pair of values p_i^5 and p_j^5, call the modified Euclid algorithm to get s_n, t_n and the sign. Remember respecting the condition that $a > b$ when calling this method.
- Calculate either $s_n - t_n$ or $t_n - s_n$ (depending on the amount of iterations); subtract one, multiply times two, add to a counter.
- Perform a last modulus operation to ensure that the counter remains in the appropriate values. ENDUT! HOCH HECH!!

There isn't much room for optimisation, but I believe that there isn't also much room for doing something wacky and completely different that happens to work. This is, in fact, why the problem is kind of difficult: there aren't many approaches that can work. Of course, this being Project Euler, I expect that someone might do some wild impossible magic arriving at the solution in a ridiculous different way, but in any case I don't think you can get much better than $O(n^2 \log n)$, which is the order of my solution, and which is fast since $n = 1000$ is tiny. Run time: 0,37 seconds. Not much room to improve.

759. A squared recurrence relation

Difficulty rating: 25 %.
Solution: 282771304. *Solved: Mon, 14 Jun 2021, 05:45.*
Math knowledge used: induction.
Programming techniques used: dynamic programming.

This problem reminds me of 325, in that the solution comes from dynamic programming after dividing the space in blocks, in such a way that the final solution comes from accumulating some "whole" blocks and then a "partial", smaller one, and both kinds of blocks are built using dynamic programming. In 325, the blocks were "Fibonacci"-sized, but here their sizes are powers of two.

Let's start with the obvious: find a pattern about $f(n)$. It can be seen that $f(n)$ is equal to n multiplied times the amount of set bits in the binary expression of n. Let's call $B(n)$ to this amount of set bits. So we want to prove that $f(n) = nB(n)$ for all n. This can be proven by induction:

- Base case: $f(1) = 1$. It holds because $B(1) = 1$, so $f(1) = 1 \cdot B(1) = 1$.
- For $n > 1$, assume that the proposition has been proven for every value smaller than n. So, if n is even, we have $f(n) = f(2k) = 2f(k)$ for some

k. Now, since the proposition has been already proven for k, we can say that $f(2k) = 2kB(k)$. Since the binary expansion of n is the same one as for k but with an extra 0 at the end, this means that $B(n) = B(k)$. Therefore $f(n) = f(2k) = 2kB(k) = 2kB(n) = nB(n)$.

- Now, for odd numbers, we have $f(n) = f(2k+1) = 2k+1+2f(k)+\frac{1}{k}f(k)$. We also know that the amount of set bits in the binary expansion of n is the same one as in k plus one, since the string of bits is the same but with an 1 appended at the end. In other words, $B(n) = B(k)+1$. Now, operating with the expression of $f(n)$, assuming that the proposition has been proven for k, we have: $f(n) = f(2k+1) = 2k+1+2f(k)+\frac{1}{k}f(k) = n+2kB(k)+\frac{1}{k}\cdot kB(k) = n+(2k+1)B(k) = n+nB(k) = n(B(k)+1) = nB(n)$.

Ok. I'm convinced. But how does this help us calculate the solution? The answer is that the binary expansions the problem asks follow a simple recursive pattern when grouped in intervals of the form $[2^n, 2^{n+1}-1]$. Let's assume that we have already calculated the values of $B(k)$ for every number in this interval, and we want to calculate the values of $B(k)$ in the next interval, $[2^{n+1}, 2^{n+2}-1]$. The pattern is this: for the first half of the interval, $[2^{n+1}, 2^{n+1}+2^n-1]$, the binary expansions are the same, but replacing the very first digit, 1, with the string 10. Therefore the binary expansion is the same. For the second half of the interval, $[2^{n+1}+2^n, 2^{n+2}-1]$, we are replacing said 1 with 11, i.e. adding another bit. Mathematically, we can say that these formulas hold:

$$\begin{aligned} B(2^n+k) &= B(k) \qquad \text{when} \quad k \in [2^n, 2^{n+1}-1]; \\ B(2^{n+1}+k) &= B(k)+1 \quad \text{when} \quad k \in [2^n, 2^{n+1}-1]. \end{aligned}$$

Now, how does this help find the solution? To understand the trick, let's go back to the initial formula: $S(n) = \sum_{i=1}^{n} f(i)^2$. We are going to expand this formula into something much easier to work with. First we replace $f(i)$ with the formula we had:

$$S(n) = \sum_{i=1}^{n} f(i) = \sum_{i=1}^{n} (B(i)\cdot i)^2.$$

And now we are going to put the focus on $B(i)$ instead of on i. We will transform this formula into one that is more complicated, but which makes

it easier to follow a recurrence. The first thing we will do is to define the set of numbers in any given interval such that their binary expansion has a certain fixed amount of ones:

$$G_{[a,b]}(x) = \{i : i \in [a,b] \wedge B(i) = x\}.$$

Note that the $G_{[a,b]}(x)$ is a set, not a number. This definition can be used to turn the expression of $S(n)$ into something more complicated-looking:

$$S(n) = \sum_{x=1}^{\infty} x^2 \cdot \sum_{i \in G_{[1,n]}(x)} i^2.$$

That infinity symbol is kind of a placeholder indicating that we aren't going to bother with an explicitly calculated limit, at least not for now. In fact, the highest value of x we will need to use scales logarithmically with n, since the smallest component of $G_{\mathbb{N}}(x)$ is $2^x - 1$, meaning that for a given n we won't encounter values of x bigger than $\log_2 n$.

The key, now, is to group the values of $G_{[1,n]}(x)$ in such a way that we don't need to store the whole set. We only need to calculate the sum of the squares of its elements. For this, we are now going to define three functions that operate on sets, so that for each relevant interval we only need to store the values of these functions. Given any set Q, these functions are:

$$\begin{aligned} a(Q) &= \sum_{i \in Q} 1; \\ b(Q) &= \sum_{i \in Q} i; \\ c(Q) &= \sum_{i \in Q} i^2. \end{aligned}$$

That is, $a(Q)$ is just the cardinality of Q; $b(Q)$ is the sum of its elements, and $c(Q)$ is the sum of its squares. This means that

$$S(n) = \sum_{x=1}^{\infty} x^2 \cdot c\left(G_{[1,n]}(x)\right).$$

We are now very close to the solution, because thanks to the easy recurrence of $B(i)$, it's easy to build the values of $c\left(G_{[a,b]}(x)\right)$ for all the relevant values of x and any given interval. First of all, we note that given two disjoint sets

A and B, we have

$$\begin{aligned} a\left(A \cup B\right) &= a\left(A\right) + a\left(B\right); \\ b\left(A \cup B\right) &= b\left(A\right) + b\left(B\right); \\ c\left(A \cup B\right) &= c\left(A\right) + c\left(B\right). \end{aligned}$$

Now, much more interestingly, given any set A we can define the set $A^{\oplus n} = \{i + n : i \in A\}$, and note that

$$\begin{aligned} a\left(A^{\oplus n}\right) &= a\left(A\right); \\ b\left(A^{\oplus n}\right) &= b\left(A\right) + na\left(A\right); \\ c\left(A^{\oplus n}\right) &= c\left(A\right) + 2nb\left(A\right) + n^2 a\left(A\right). \end{aligned}$$

These formulas can be easily checked, given their definition, and they allow us to operate on sets at a very small cost. What we are going to do is this:

- We will divide the working interval of the problem, $[1, 10^{16}]$, into smaller blocks.
- For each one of these blocks, representing a set A, we are going to keep an array of values, so that the index of the array represents the value of x, and each element of the array contains the values of $a\left(A\right)$, $b\left(A\right)$ and $c\left(A\right)$.
- We will create each of these blocks from previous ones, and at the end we will combine them into a single one.
- Finally we will have a single array representing the whole working interval, where we can very easily evaluate the solution to the problem by evaluating the expression of $S\left(n\right)$ in terms of the function c.

Let's work on the intervals. First we start with the smallest interval, $\{1\}$. Obviously we only need to keep an element for the case $G_{\{1\}}\left(1\right)$, since $x = 1$ it's the only value for which $G_{\{1\}}\left(x\right)$ is not empty. In this set, it's clear that $a\left(G_{\{1\}}\left(1\right)\right) = 1$, $b\left(G_{\{1\}}\left(1\right)\right) = 1$ and $c\left(G_{\{1\}}\left(1\right)\right) = 1$.

The first piece of magic comes now thanks to the recursive nature of the pattern of $B\left(n\right)$, which allow us to deduce that:

$$G_{[2^{n+1}, 2^{n+2}-1]}\left(x\right) = \left(G_{[2^n, 2^{n+1}-1]}\left(x\right)\right)^{\oplus 2^n} \cup \left(G_{[2^n, 2^{n+1}-1]}\left(x-1\right)\right)^{\oplus 2^{n+1}}.$$

Furthermore, these two sets are disjoint and we can use the "summation" formulas for such pairs of sets. With these formulas, the calculation for in-

tervals up to any power of two is immediate and very fast. Too bad that 10^{16} is not a power of 2. So we need a final touch, and having calculated all the sets and the needed sums for sets of the form $[2^n, 2^{n+1} - 1]$, we can use some recursive formulas, only slighttly more complicated that the one above. Now we are interested on sets of the form $[2^n, 2^n + N]$ where $N < 2^n - 1$. For these sets, we have:

$$G_{[2^{n+1},2^{n+1}+N]}(x) = \begin{cases} \left(G_{[2^n,2^{n+1}-1]}(x)\right)^{\oplus 2^n} & \text{if} \quad N = 2^n - 1; \\ \left(G_{[2^n,2^n+N]}(x)\right)^{\oplus 2^n} & \text{if} \quad N < 2^n - 1; \\ \left(G_{[2^n,2^{n+1}-1]}(x)\right)^{\oplus 2^n} \cup \left(G_{[2^n,N]}(x-1)\right)^{\oplus 2^{n+1}} & \text{if} \quad N > 2^n - 1. \end{cases}$$

Note the interval $[2^n, N] = [2^n, 2^n + (N - 2^n)]$. In the first case, the recursion ends immediately, while in the other two, we base the results on other non-power of 2 based sets, requiring recursive calls. As usual, these sets are disjoint and the summation formulas can be used. This looks complicated because there are many definitions, but take into account that for each relevant set A and its associated subsets G_A we only store the values of $a(G_A(x))$ in an array. This makes the calculations actually really simple, and even better, also very fast since the run time is $O\left(\log^2 n\right)$.

The summarised algorithm is this:

- Initialise the values of a, b and c for the set $G_{\{1\}}(1)$.
- Generate a, b and c for as many valyes of x as necessary (one more at each step) for sets of the form $G_{[2^n,2^{n+1}-1]}(x)$.
- For the final, "remainder" set, use recursive formulas in order to generate the whole sets of values for $G_{[2^n,10^{16}]}(x)$ for the last n (i.e., $n = \lfloor \log_2 10^{16} \rfloor = 53$).
- Considering that all the subsets are disjoint, use the sum formulas to unify all the sets into the values of a, b and c for $G_{[1,10^{16}]}(x)$.
- Calculate the value of $S(10^{16}) = \sum_{x=1}^{\infty} x^2 \cdot c\left(G_{[1,10^{16}]}(x)\right)$.

Ideally we could just store the modded values of the functions a, b and c, but since it's easy to see that the whole result is going to be of the order of $O(n^3)$, i.e. about 10^{50} in this case, we can just use BigIntegers. The final result, generated in 25 milliseconds, is this:

$$244966793232759468702280778963970515261259050549248.$$

760. Sum over Bitwise Operators

Difficulty rating: 35 %.
Solution: 172747503. *Solved: Mon, 28 Jun 2021, 06:05.*
Math knowledge used: induction.
Programming techniques used: dynamic programming.

This problem is a lot like the previous one, in the sense that it can also be solved using formulas built on top of smaller results, meaning that the "block" approach works very well with even the same basic block structure as in the previous problem (that is, blocks representing intervals of the form $[2^n, 2^{n+1} - 1]$ for some given n) but, although this solution lacks the "sub-blocks" from the previous one, here we have an additional level of complexity given by the fact that each block is built not on the previous one, but on all the previous ones. This also means that the recursive calls for partial blocks have a potential to explode, and this must be taken into account (caching is enough, since most often the affected blocks are small ones at the bottom of the recursions).

Surprisingly, reversing the summation order doesn't do much in this problem. So we can start by defining $f(n)$ as the internal summation, that is:

$$f(n) = \sum_{k=0}^{n} g(k, n-k) = \sum_{k=0}^{n} ((k \oplus (n-k)) + (k \vee (n-k)) + (k \wedge (n-k))).$$

This means that the goal of the problem is to calculate

$$G(N) = \sum_{n=0}^{N} f(n).$$

The lower limit of this summation might as well be a 1 since $f(0) = 0$ quite clearly.

As for how to generate the "blocks", well, I spent a lot of time deriving precise methods, but the binary nature of the problem made it natural to use blocks with powers of 2, where each block has one bit more than the rest. The method is still a bit different to what I had imagined initially, because the presence of subtraction throws off most obvious approaches. In fact, contrary to what I would have initially imagined, this method is not based on adding bits to the top of existing values, but rather, to the *bottom*. In fact, I don't just add single bits, but strings of them. I'm sure that there are better ways of doing this (although most approaches will behave naturally in a logarithmic way, as the calculations will be done digit by digit), but this is what I came up with.

The blocks are defined by an exponent n, so that each block encompasses the interval $[2^n, 2^{n+1} - 1]$, which coincidentally includes all the numbers with exactly $n+1$ bits. The first block, for $n = 0$, includes just the singleton set $\{1\}$. Each block is built on top of all the other previous ones (this is important: not just the immediately previous one, like in problems 325 or 759), using a curious approach. Let's say that we have already calculated all the blocks for values of $0, 1, \ldots, n-1$, and we are now going to calculate the values for the block n. Then:

- We add a 1 at the end of the block $n - 1$ (this gets the odd numbers).
- We add a 10 at the end of the block $n - 2$.
- We add a 100 at the end of the block $n - 3$.
- We add a 1000 at the end of the block $n - 4$.
- And so on, until we append $100\ldots000$ (with $n - 1$ zeroes) at the end of the block 0.
- Finally we calculate the value for $f(2^n)$ separately.

This means that we have $2^{n-1} + 2^{n-2} + \ldots + 2^3 + 2^2 + 2^1 + 2^0$ elements in total, plus one for the power of two, making a total of 2^n elements, which is indeed the length of the interval. The values are different by construction, and all of them have the same amount of bits, meaning that this scheme will correctly calculate all the values from the $[2^n, 2^{n+1} - 1]$ interval.

As for how to calculate the value of $f(n)$, well, this is interesting. Let's say that we have a value x and we are going to append the string $100\ldots000$, with exactly s zeroes (for some $s \geq 0$) at the end. The resulting number is $x' = 2^{s+1}x + 2^s$. Now, it's clear that each $f(x)$ is the sum of exactly $x + 1$ addends, from $k = 0$ to $k = x$, included. So we first do a tensor multiplication of sorts, so that we create a cross product of each addend in $f(x)$ and each addend of $f(2^s)$. For example, if $x = 11$ and $s = 1$, we have the cross product of pairs $\{(11, 00), (10, 01), (01, 10), (00, 11)\}$ (as a prefix) with $\{(10, 00), (01, 01), (00, 10)\}$ (as a suffix). This is what I call the "normal" construction. Unfortunately, this is not complete: these are $4 \cdot 3 = 12$ elements, but the number we're working on is 1110, which is 14 in base ten, meaning that there should be 15 terms. The other three come from what I call the "special" construction. In this construction we use $f(x - 1)$ instead of $f(x)$, and for the suffix, which still has $s + 1$ bits, the leftmost one is always a 1, and the rest come from a subset of the case for 2^s, including only the $2^s - 1$ non-extreme cases, i.e., ignoring the pairs $(100\ldots00, 000\ldots00)$

and $(000\ldots00, 100\ldots00)$. This means, in the previous case, that we get a cross product of $\{(10, 00), (01, 01), (00, 10)\}$ (the cases for $x - 1 = 10$) with just $\{(1, 1)\}$. So in total we have $(x + 1)(2^s + 1) + x(2^s - 1)$ elements, for a grand total of $2^{s+1}x + 2^s + 1$, which is exactly the amount of terms we need. This "special" construction is necessary because of the subtractions, which are problematic, but fortunately this is enough.

With this construction method, the full formula for $f(x' = 2^{s+1}x + 2^s)$ can be derived. It's divided in four terms:

- The "normal" prefixes, coming from the values of $f(x)$ shifted and replicated: $f(x) \cdot 2^{s+1} \cdot (2^s + 1)$.
- The "normal" suffixes, coming from the values of $f(2^s)$ replicated: $(x + 1) \cdot f(2^s)$.
- The "special" prefixes, coming from the values of $f(x - 1)$ shifted and replicated: $f(x - 1) \cdot 2^{s+1} \cdot (2^s - 1)$.
- The "special" suffixes, coming from the restricted values of $f(2^s)$ (and the extra 1s) replicated: $x \cdot (f(2^s) + 2^{s+1} \cdot (2^s - 3))$.

These formulas can be proven with some pen and paper. Adding these four results yields the exact value for $f(x')$ given $f(x)$ and s (OK, and $f(x - 1)$ as well), but this scheme can't be used for the calculation of $f(2^s)$ for any s. For these, we use the following formula, which can be proven using induction although I arrived at it using some inspection and solving a recurrence equation based on some (then unproved) assumptions:

$$f(2^s) = 2^{2s+1} + 2^{s+1} - s \cdot 2^s.$$

This is interesting to derive single values of $f(x)$ (which we will need to do), and conveniently, the multipliers only depend on s, meaning that we can calculate them once and store them for repeated use. Now, if we want to act on blocks, we will need the sums of the factors based on x, which means that for each block of the form $[a, b]$ we will store all this information:

- The start of the interval (always a power of 2).
- The amount of values in the interval (a power of 2 in the full blocks, not so in the partial ones that we will also need).
- The sum of all the values of x in the interval.
- The sum of all the values of $f(x)$ in the interval.

- The value of $f\,(a-1)$ and of $f\,(b)$, which we need for the block sum of $f\,(x-1)$.

We will have "full" blocks representing sets of the form $[2^s, 2^{s+1}-1]$, which will be calculated at the start of the algorithm, but we will also need a way to calculate partial blocks of the form $[2^s, x]$ for some $x \in [2^s, 2^{s+1}-1]$. Aside from these blocks of data, we will also have very useful "multipliers", one for each value of $s \geq 0$ needed, storing the values of $f\,(2^s)$ and the factors involved in the "normal" and "special" prefixes and suffixes.

Now, the full algorithm. First, generate all the "multipliers" with those juicy, useful factors that we will use over and over. If the limit is some N, we will need all the multipliers from $s = 0$ to $s = \lfloor \log_2 N \rfloor$. Then we can use these to generate the "full" blocks up to, again, $n = \lfloor \log_2 N \rfloor$. But the latest block will be a "partial" one. To do this, we use a recursive algorithm. Given a problematic interval of the form $[2^s, 2^s + x]$ (this format, using x as a difference, is convenient to quickly determine the amount of elements from each smaller blocks that we need), we first add the value of $f\,(2^s)$ which we already know. Then, we proceed like this: let $x_1 = \left\lfloor \frac{x}{2} \right\rfloor$, and $n_1 = x_1 + (x\,\%2)$. So we will append an 1 to the partial block $[2^{s-1}, 2^{s-1} + n_1 - 1]$, which we can calculate recursively. Then, let $x_2 = \left\lfloor \frac{x_1}{2} \right\rfloor$ and $n_2 = x_2 + (x_1\,\%2)$, and we will append an 10 to the partial block $[2^{s-2} + 2^{s-2} + n_2 - 1]$, again calculated recursively. And so on. We can finish early when some $x_i = 0$. Each one of these blocks might be recursive as well, but if $n_i = 0$, we only need to use the (already calculated) power of 2 and end the recursion immediately. In any case, the value of $f\,(a-1)$ for this block can be conveniently extracted from the "full" blocks, but unfortunately the value of $f\,(b)$ will need to be calculated manually. This recursion is not linear and there is potential for an exponential growth, but fortunately most of the values are reused, so in addition to all this, we will keep a cache for the values of $f\,(x)$ and for the partial blocks we might need. This trims the exponential growth so nicely that the end result, even with BigIntegers, can be calculated in 0,12 seconds. By the way, this full result is:

$$566654255793637336012332947078208682645672458325196800.$$

And with this, I finally achieved Ten Out of Ten, possibly the highest Project Euler award I will ever get.

764. Asymmetric Diophantine Equation

Difficulty rating: 40 %.

Solution: 255228881. *Solved: Sun, 12 Sep 2021, 01:15.*

Math knowledge used: Pythagorean triples.

Programming techniques used: none.

There are two basic ways of solving this problem. Both of them seem kind of obvious to me and, although at first I thought that one of them would be more complicated, in fact both have about the same amount of code. I prefer the one with Pythagorean triples, but the other one, based on factorisations, is only a bit slower.

Let's see. $16x^2 + y^4 = z^2$. This is clearly a special case of a Pythagorean triple, with $a = 4x$, $b = y^2$ and $z = c$. Now let's see. The standard way of generate all the primitive Pythagorean triples is with $a = m^2 - n^2$, $b = 2mn$ and $c = m^2 + n^2$. Of course we can exchange a and b as we like. And a's formula is especially interesting because y^2 must, blatantly, be a square. So let's start with $a = y^2$ and $b = 4x$. So, $m^2 - n^2 = y^2$. Which is the same as saying $y^2 + n^2 = m^2$. Which is also a Pythagorean triple! Awesome. So we can iterate over primitive Pythagorean triples using the standard pairs of coprime (m, n) with different parity, and then we do this:

$$\begin{aligned} n' &= 2mn, \\ m' &= m^2 + n^2, \\ y &= m^2 - n^2. \end{aligned}$$

This gets us

$$(2mn)^2 + \left(m^2 - n^2\right)^2 = \left(m^2 + n^2\right)^2,$$

and now we can use $4x = 2m'n'$, so that $x = mn\left(m^2 + n^2\right)$, and $z = m'^2 + n'^2$. If we use $b = 2mn$ and $c = m^2 + n^2$, we can simplify as $x = mnc$ and $z = b^2 + c^2$. We accept the triple if z is below the limit, and we end the iteration when the first z for a given m is already above the limit. Easy!

Not so fast. We haven't finished yet. This always results in odd values of y, but the problem shows an example where $y = 4$. Clearly we are missing solutions here. How to proceed? Let's think in a different way. We need that y is a perfect square, so we can now start with the other assignment of a and b to $4x$ and y^2, meaning $4x = m^2 - n^2$ and $y^2 = 2mn$, and look for products that result in a square. Since m and n must be coprime, no common factors between them, we can get the cases where one of them is an odd square and the other one is the even half of an even square. The really good part about this is that you can just iterate over m and n normally, and then create

squares from them. For even m, we do:

$$\begin{aligned} m' &= \frac{m^2}{2}, \\ n' &= n^2; \end{aligned}$$

for odd m, instead we will do

$$\begin{aligned} m' &= m^2, \\ n' &= \frac{n^2}{2}. \end{aligned}$$

This ensures that $y = 2m'n'$ is a perfect square. Unfortunately $m'^2 - n'^2$ is odd, so we need to multiply the whole triple times 4, so that we get:

$$\begin{aligned} 4x &= 4\left(m'^2 - n'^2\right), \\ y^2 &= 8m'n' = 2mn, \\ z &= 4\left(m'^2 + n'^2\right). \end{aligned}$$

As long as m and n are coprime, this results in a primitive triple satisfying the conditions. Also, y will always be even, ensuring that there is no overlap with the previous set of solutions. Like in the other case, we accept the triple if z is below 10^{16}, and when the first z for a given m is above that limit, we can stop.

That's it. This covers all the solutions. This is very fast, just a bit above 1,5 seconds to sum them all and calculate the modulus when needed.

Just for fun, let's think of the alternate approach. We start with the formula

$$(4x)^2 + y^4 = z^2,$$

which we transform into

$$y^4 = z^2 - (4x)^2 = (z + 4x)(z - 4x).$$

How can we guarantee that this product will be a fourth power? Let's just think the other way around. We start with y, getting all its factors, and we can decompose the factors of y^4 into coprime products where the difference between both factors is a multiple of 8. And now, something beautiful and very useful: for every odd number n, $n^4 \equiv 1 \bmod 8$. This means that for odd y we can just use the coprime decompositions. For even y we need to pay more attention. If 2 appears exactly once in the decomposition of y, then it appears 4 times in the decomposition of y^4: we need to split it so that each

factor has two of them. If 2 appears more than once, it will appear $4n > 6$ times in y^4. We will give 2^3 to one factor, and 2^{4n-3} to the other. Aside from that, it's a matter of trying all the combinations of factors. This is kind of exponential, although in practice there aren't more than 4 or 5 different factors. After each decomposition we will have $y^4 = ab = (z + 4x)(z - 4x)$ and we can guarantee that $z = \dfrac{a+b}{4}$ and $x = \dfrac{a-b}{8}$ will be integer. However sometimes this results in cases where both x and z are even (I suspect where this might come from, but I just haven't bothered to look further into it. I just add the triple only if either x or z is odd). By the way, we need to iterate up to $y = \sqrt{10^{16}} = 10^8$.

The description is simpler, but it involves more code than the Pythagorean approach. There is the typical mass decomposition using a prime sieve, and then we do some iterative magic to create the nigh-exponential list of coprime decompositions. The code is also somewhat slower, although I used BigIntegers instead of long (in hopes that this could be used for bigger limits), so the times can't really be compared.

769. Binary Quadratic Form II

Difficulty rating: 90 %.

Solution: 14246712611506. *Solved: Sat, 26 Feb 2022, 18:46.*

Math knowledge used: homogeneous ternary quadratic forms, Chinese remainder theorem.

Programming techniques used: none.

This is a hard problem and I'm quite proud of having solved it. I solved it in that narrow space when the problem has not yet been finalised (94 public solvers at this time), but enough time has passed so that the difficulty rating has been set. Obviously this space only exists for problems hard enough to not reach 100 public solvers in 10 weeks. Anyway this is about the second or third hardest problem I've been able to solve and I'm quite happy. I wouldn't have been able to solve it if not for the fantastic reference book that the user *philiplu* mentioned in the comments for another, more recent problem (785, I believe). Also, this is the first time I remember seeing a problem change its difficulty rating. It was at 85 % when I solved it, but now it displays 90 %.

Now, let's discuss the problem itself. As the header says, here we need **ternary** quadratic forms, not binary ones, so Alpertron doesn't help us. It turns out that they work in a very different way, and if they have a solution, they have an infinite amount of them. Then again, this is also about *homogeneous* ternary quadratic forms, meaning that all terms have degree 2. There

are only 6 terms to worry about, and in our case, there are only 4 since the xz and yz terms are null. Obviously we are going to use the equation

$$x^2 + 5xy + 3y^2 - z^2 = 0,$$

and we need to find how many primitive solutions are with $0 \leq x, y, z \leq 10^{14}$. The first thing we are going to do is to invoke the ternary forms magic, which basically tells us that we can choose any primitive solution (we will use $x = 1$, $y = 0$ and $z = 1$; it doesn't count for the problem, but it's a solution to the equation, and we can use it as a starting point), and from it we derive a parametrisation of the solutions:

$$\begin{aligned} x &= 1 \cdot r, \\ y &= 0 \cdot r + p, \\ z &= 1 \cdot r + q. \end{aligned}$$

Plugging these values into the equation we get

$$r\left(5p - 2q\right) = q^2 - 3p^2,$$

and some additional symbol pushing gets us

$$r \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -3 & 0 & 1 \\ 5 & -2 & 0 \\ -3 & 5 & -1 \end{pmatrix} \begin{pmatrix} p^2 \\ pq \\ q^2 \end{pmatrix}.$$

Note that the determinant of this matrix is 13, which is a number that will be featured prominently in the solution because there are important implications. Long story short, we are going to use the equations

$$\begin{aligned} x &= -3p^2 + q^2, \\ y &= 5p^2 - 2pq, \\ z &= -3p^2 + 5pq - q^2. \end{aligned}$$

Sometimes, all these numbers will be a multiple of 13, and sometimes they will not. There could be as many cases as divisors of 13, but fortunately, in this case there are only two. Now, with all the ternary forms magic properly set up, we just need to play with these expressions to ensure that the problem conditions (all values within bounds, and only primitive solutions) remain. First, we need cases where x and y are both positive. If we restrict the

parametrisation to positive values of p and q, which makes sense at first glance (and it happens to be good enough, but it might not have been), then x's formula gives us $q \leq \lceil \sqrt{3}p \rceil$, and $y = p\,(5p - 2q)$ gives us $5p > 2q$. This suggests iterating values of p, and for each one of them, use values of q such that $\lceil \sqrt{3}p \rceil \leq q < \left\lceil \frac{5p}{2} \right\rceil$. Note that the first symbol is $\leq$ but the second one is $<$. Of course, this is not enough. It's pretty clear that $z > x$ and $z > y$, so for the upper bound we only need to consider whether z exceeds it or not. Before going into that, we note that some quick experiments reveal very useful information about this parametrisation, namely:

- No repeated solutions appear: different pairs of $\{p, q\}$ yield different $\{x, y, z\}$ tuples.
- Non-primitive solutions appear when p and q are not coprime: if certain pair $\{p_0, q_0\}$ gives us the solution $\{x_0, y_0, z_0\}$, then the pair $\{np_0, nq_0\}$ gives us the nonprimitive solution $\{n^2x_0, n^2y_0, n^2z_0\}$.
- If we ignore these non-coprime pairs, we do get all the solutions, but sometimes all $\{x, y, z\}$ appear multiplied times 13. This is the only problematic point.

Ok, so we need to iterate over coprime pairs of p and q, where q satisfies the aforementioned bounds, but we also need to take into account the fact that sometimes the solution will be multiplied times 13. Ok, but when does this happen? If we can predict it beforehand, everything becomes easier. So we can look for values where x and y are multiple of 13 (without both p and q being it as well, since we are looking for coprime cases), and z will follow, by construction. This gives us two equations modulo 13:

$$\begin{aligned} q^2 - 3p^2 &\equiv 0, \\ p\,(5p - 2q) &\equiv 0. \end{aligned}$$

The second equation is easier. Since 13 is prime, either p is a multiple of 13, or $5p - 2q$ is. If p is a multiple of 13, q must be as well for the first equation to hold. This means that the solution wouldn't be coprime and we can discard it. We can then look at the other case, $5p - 2q \equiv 0 \pmod{13}$. Since $2^{-1} = 7 \pmod{13}$, we can multiply the equation times 7 to get

$$35p - 14q \equiv 0 \Rightarrow 9p \equiv q.$$

This equation also satisfies the first one, so this is exactly the condition we wanted. In terms of solving the problem, we are going to use this to

determine whether z is below 10^{14} or not (note that we are not interested in the values themselves; we only want to count them, and we already know that we are going to get all the solutions with the bounds of q and the coprimality condition, but we now know that, if $q \equiv 9p \pmod{13}$, we need that $z = -p^2 + 5pq - 3q^2$ is below $13 \cdot 10^{14}$, whereas if $q \not\equiv 9p$, the bound is just 10^{14}. Note that these values of z are always going to be positive if we are between the correct bounds of p and q. We can use a standard bound, $C \in \{10^{14}, 13 \cdot 10^{14}\}$, and for each one of the two values the analysis is similar. Treating $p = P$ as a constant, we want to find the set of valid values of q:

$$-3P^2 + 5Pq - q^2 < C \Rightarrow q^2 - 5Pq + \left(3P^2 + C\right) > 0.$$

Very simple quadratic equation that we can solve for every given value of p. Now, this solution might have two real solutions q_1 and q_2, in whose case we want the set of values *outside* the interval $[q_1, q_2]$, or two complex solutions, in whose case every value of q satisfies the solution (this sounds surprising, but consider that for very high values we will get negative values of z albeit with a high absolute value; these also happen to be discarded by the previous set of bounds between p and q). We need to consider $\Delta = (-5P)^2 - 4\left(3P^2 + C\right) = 13P^2 - 4C$. The center between the two solutions happens to be $-\dfrac{5P}{2}$, the upper bound we already had. This suggests that the proper way of proceeding is replacing this bound with the smallest solution of the equation, if it exists. Now, if p is high enough, we can guarantee that $\Delta > 0$ and there are two solutions, so the upper bound for P can be determined by comparing the smallest solution against the lower bound for q. The first condition, $\left\lceil \sqrt{3}p \right\rceil \leq q$, is set in stone, and the second one depends on Δ, but if p is high enough, this is always positive, so we can use this upper bound for q:

$$q < \left\lceil \frac{5p - \sqrt{13p^2 - 4C}}{2} \right\rceil.$$

The upper bound for p comes from the case where the valid set of values of q is empty:

$$\left\lceil \sqrt{3}p \right\rceil \geq \left\lceil \frac{5p - \sqrt{13p^2 - 4C}}{2} \right\rceil.$$

Although it's not completely exact, we can ignore the $\lceil \cdot \rceil$ operator (we are only interested in integer values anyway) and proceed with just

$$\sqrt{3}p \geq \frac{5p - \sqrt{13p^2 - 4C}}{2}.$$

After some manipulation, this becomes $p \leq \left\lceil \sqrt{\frac{C}{5\sqrt{3}-6}} \right\rceil$. We have almost finished now, but we still have two separate problems:

- Given some value of p, count the amount of $q \in [q_0, q_1)$ such that p and q are coprime and $q \not\equiv 9p \pmod{13}$.
- Given some value of p, count the amount of $q \in [q_0, q_1)$ such that p and q are coprime and $q \equiv 9p \pmod{13}$. Obviously the bounds are different than in the previous case.

It turns out that this can be done using inclusion-exclusion with the prime factors of p (yes, I'm putting my "prime factors lattice generator" to good use lately), and with the Chinese remainder theorem. For each prime factor of p, we reduce this to one or two applications of the following function: given some bounds q_1 and q_2, count how many values of q verify $q_1 \leq q < q_2$ and $q = ak + b$ for some given a and b. This can be computed in constant time. So it's a matter of iterating carefully and counting separately the cases for multiples and non-multiples of 13. The final code is about 70 lines, although the prime lattice generation and the Chinese remainder theorem are not counted in these. And the run time is, funnily, around 13 seconds.

770. Delphi Flip

Difficulty rating: 30 %.
Solution: 127311223. *Solved: Sun, 7 Nov 2021, 09:04.*
Math knowledge used: Game theory, Catalan numbers.
Programming techniques used: dynamic programming.

Although I did use game theory, Catalan numbers and dynamic programming to arrive to the solution, in fact the code itself is *ridiculously* simple. So much that even the code for problem 1 is more complex. This it not at 218's levels of ridiculousness, but almost. The nice part is that there are "proper" ways to solve the problem, but there is an approximation that happens to be good enough. An approximation thanks to which the *full* set of operations used to solve this problem consists of: a single subtraction; a single multiplication (a square, in fact); two divisions; and a single ceiling operation. Getting there takes more work, of course. Let's see how it is.

We start with the problem definition. We are looking for a function $g(X)$ which, in usual Project Euler fashion, is not calculated directly. In fact, we are more interested in the inverse function: if $g(X) = n$, then we are looking for an n so that $f(n) = X$ for some function f. In particular, given the

value of n, we want to calculate the expected earnings, $f(n) = X$, and we will say that $g(X) = n$ the first time we get $f(n) = x \geq X$. So, how do we calculate $f(n)$? Game theory comes to the rescue. Let's start with a different function: let's say that we currently have 1 unit (we will always work this way, with ratios; it makes everything simpler), and we know that there are i *gives* and j *takes* pending (which are chosen by the other player, B). So we define the function $h(i, j)$ as the expected earnings at the end of the game, and we can calculate this value using dynamic programming. Furthermore, $f(n) = h(n, n)$ by definition, which is why this function is useful. The values can be calculated recursively, starting by some base cases:

- If $i = 0$, meaning that all the pending operations are *takes*, it doesn't make sense to bet any positive amount, since we know that we will lose what we bet. So the bet is always 0 for all these values. We say that $h(0, j) = 1$ for every value of j.
- On the other hand, if every remaining operation is a *give*, what makes sense is to bet as much as possible, since B is obligated to return our bet. So we will bet everything, doubling our earnings. This means that $h(i, 0) = 2^i$ for every i.
- Finally, let's analyse the complicated case, $h(i, j)$ for $i, j > 0$. We start with the value 1, and we bet an amount $a \in [0, 1]$. If B decides to *give*, then our value is now $1+a$, but then we move to a status where we have one less *give* remaining. And if B decides to *take*, our value is $1-a$, but at least there is one less *take* left. If we call $T = h(i, j-1)$ (the result after a *take*) and $G = h(i-1, j)$ (the result after a *give*), then the final solutions are respectively $G \cdot (1+a)$ and $T \cdot (1-a)$. Now, we assume that B is adversarial, and we are looking for guarantees, so we will take the minimum of these amounts: $h(i, j) = \text{mín}\{G \cdot (1+a), T \cdot (1-a)\}$.

  This is a classic example of game theory. The equilibrium is found when these two quantities are the same, and this gives us an equation for the optimal value of a:

$$G \cdot (1+a) = T \cdot (1-a) \Rightarrow a = \frac{T-G}{T+G}.$$

  It's expected that T will be bigger than G, since the total amount of turns is the same but we have one more *give* remaining in T. So $a \in [0, 1]$ (in fact, $a \in (0, 1)$) and this amount is always valid. Some quick calculations indicate that the value of the turn is $h(i, j) = \dfrac{2GT}{G+T}$, which is the harmonic mean of G and T.

With these formulas we can easily calculate $f(n) = h(n, n)$ in quadratic time, and we can get some interesting results.

n	0	1	2	3	4	5	6
$f(n)$	1	$\frac{4}{3}$	$\frac{16}{11}$	$\frac{32}{21}$	$\frac{256}{163}$	$\frac{512}{319}$	$\frac{2048}{1255}$

As expected, the numerator is always a power of two. The denominators are not so clear. And why do the exponents of the numerator grow so irregularly? Well, this is in fact a byproduct of canonisation. Let's force the numerators to be 4^n, and we get:

n	0	1	2	3	4	5	6
$f(n)$	$\frac{1}{1}$	$\frac{4}{3}$	$\frac{16}{11}$	$\frac{64}{42}$	$\frac{256}{163}$	$\frac{1024}{638}$	$\frac{4096}{2510}$

Now we have a sequence of denominators that grows more steadily. It will be much easier to work with this sequence of denominators, in fact. So, let's define the sequence of denominators, $d_n = \dfrac{4^n}{f(n)}$:

$$d_n = \{1, 3, 11, 42, 163, 638, 2510 \ldots\}.$$

Ok, it seems like d_n grows more or less exponentially, with each d_n being about four times as big as the previous one. In fact...

n	0	1	2	3	4	5
$4d_n - d_{n+1}$	1	1	2	5	14	42

... these are the Catalan numbers!! So, $d_{n+1} = 4d_n - C(n)$. Here's the thing, though. This sequence d_n, which is so closely related to the Catalan numbers, happens to be in OEIS. In fact, it is OEIS A032443. OEIS tells us several interesting things about this succession, like the formula

$$d_n = \sum_{i=0}^{n} \binom{2n}{i}.$$

This could be useful for a recursive calculation of d_n, but a bit below we see something even better: an asymptotic formula.

$$d_n \approx 2^{2n-1}\left(1+\frac{1}{\sqrt{\pi n}}\right).$$

Well, this is *amazing*, since we expect the value of n to be considerably big, and therefore d_n should be very close to its asymptotic value. And we are going to need a ceiling operation anyway, since we don't expect $f(n)$ to be exactly equal to 1,9999 for a given n, so the difference with the real value might be within this margin. This is very promising. So let's call this asymptotic succession d'_n, and then we use $f'(n) = \frac{4^n}{d'_n}$ as an approximation of $f(n)$, and $g'(n)$ as an approximation of $g(n)$ which is based on $f'(n)$ instead of on $f(n)$. Let's put all this in mathematical terms:

$$g'(1{,}9999) = n \Rightarrow f'(n) \geq 1{,}9999 \Rightarrow \frac{4^n}{d'_n} \geq 1{,}9999.$$

Expanding the formula for d'_n we get

$$1{,}9999 \leq \frac{4^n}{2^{2n-1}\left(1+\frac{1}{\sqrt{\pi n}}\right)} = \frac{2}{1+\frac{1}{\sqrt{\pi n}}} \Rightarrow 1+\frac{1}{\sqrt{\pi n}} \leq \frac{2}{1{,}9999}. \Rightarrow$$
$$\Rightarrow \frac{1}{\sqrt{\pi n}} \leq \frac{0{,}0001}{1{,}9999} \Rightarrow \pi n \geq 19999^2 \Rightarrow n \geq \frac{19999^2}{\pi} \Rightarrow n = \left\lceil \frac{19999^2}{\pi} \right\rceil.$$

Lo and behold, it works! In general, the approximation for a given value $X \in (1,2)$ is

$$g'(X) = \frac{1}{\pi}\left(\frac{X}{2-X}\right)^2,$$

and we would expect it to be a very good approximation, especially when X is close to 2. In fact, for 1,9999, it so happens that $g'(X) = g(X)$! Which means that this ridiculously simple formula gives the correct solution. Incredible! It's a pity that it doesn't return the correct solution for the trial case, $X = 1{,}7$ (it returns 11 instead of the correct 10), but this is because the formula doesn't work as well for smaller values.

The run time is on the order of 0,1 milliseconds, and most of it is spent loading the Math class and calling ceil. Without it, the result is on the order of *tenths of microseconds*. Close to 218, indeed.

Just to be a bit fair, I can also explain how to solve this problem in a more "normal" way, perhaps more exact as well. Most of the reasoning is the

same, but we use the correct formula for d_n instead of an approximation. We start with

$$d_n = \sum_{i=0}^{n} \binom{2n}{i};$$

from which we can derive a relatively easy explicit formula, since we know that $\binom{2n}{i} = \binom{2n}{2n-i}$:

$$2d_n = \binom{2n}{n} + \sum_{i=0}^{2n} \binom{2n}{i} = \binom{2n}{n} + 2^{2n} \Rightarrow d_n = \frac{1}{2}\binom{2n}{n} + 2^{2n-1}.$$

This gives us a more interesting formula for $f(n)$:

$$f(n) = \frac{4^n}{\frac{1}{2}\binom{2n}{n} + 2^{2n-1}} = \frac{2}{1 + \frac{1}{4^n}\binom{2n}{n}}.$$

This formula is nice because n is contained into a single expression that can be easily iterated over (i.e., the calculation for $n+1$ follows easily from the value for n; no needo to recalculate from scratch). We are looking for a value of n such that $f(n) \geq X$, so we can do

$$\frac{2}{1 + \frac{1}{4^n}\binom{2n}{n}} \geq X \Rightarrow \frac{1}{4^n}\binom{2n}{n} \leq \frac{2}{X} - 1.$$

And yes, the left term decreases with n, and it's in fact more or less proportional to $n^{-\frac{1}{2}}$. Now, the quotient between two consecutive values of the left side is

$$\frac{\frac{1}{4^n}\frac{(2n)!}{(n!)^2}}{\frac{1}{4^{n-1}}\frac{(2n-2)!}{((n-1)!)^2}} = \frac{(2n)(2n-1)}{4 \cdot n^2} = \frac{2n-1}{2n}.$$

So we have a workable solution which is $O(N)$ in terms of the solution, which doesn't feel like cheating and which is structured in a way that actually looks like an actual Project Euler solution. First, calculate the right term, $\frac{2}{X} - 1$; then, calculate the left term for $n = 1$, which happens to be $\frac{1}{4}\binom{2}{1} = \frac{1}{2}$. Then, iterate starting from $n = 2$. If L_n is the left term $\frac{1}{4^n}\binom{2n}{n}$, the formula is just $L_n = \frac{2n-1}{2n} L_{n-1} = \left(1 - \frac{1}{2n}\right) L_{n-1}$. Stop when you find an L_n which

is smaller than the expected right term. It finishes in 0,3 seconds, so it's still a really good solution.

772. Balanceable k-bounded partitions

Difficulty rating: 20 %.
Solution: 83985379. *Solved: Tue, 16 Nov 2021, 05:38.*
Math knowledge used: none.
Programming techniques used: prime sieve.

I didn't quite prove it, but I had a suspicion that happened to be true. The answer for any N is twice the gcd of all the numbers from 1 to N (I theorised this after doing some brute force and seeing that $f(5) = f(6)$). There are many ways of doing this, but what I did was:

- Initialise a "result" variable with the value 2.
- Iterate over all the primes up to N.
- For each prime, multiply "result" times the highest power of the prime that doesn't exceed N. Apply mod as needed.
- That's it.

I'm not going to say that this is at 770 levels of difficulty, but it's not very far.

775. Saving Paper

Difficulty rating: 40 %.
Solution: 946791106. *Solved: Sun, 16 Jan 2022, 12:14.*
Math knowledge used: none.
Programming techniques used: dynamic programming.

The pattern needed to solve this problem was a big surprise to me. I expected the optimal decomposition to be always cuboids or very close representations (such as cuboids with a single added or removed cube), so that any number of the form *abc* would be most compactly represented as such with a surface of exactly six rectangles, but there is a pattern based on filling cubes (I mean, in the sense of third powers) which doesn't depend on decompositions. In some cases the best solution happens to verify this (for example, for $n = 24 = 2 \cdot 3 \cdot 4$ the best solution, with $g = 92$, *might* be a $2 \times 3 \times 4$ cuboid, but the pattern provides a way of building a solution which

is not a cuboid), but the general pattern can be found without decomposing the numbers. I should have suspected it when I saw that the limit was 10^{16}. Anyway, the solution is $O\left(\sqrt[3]{N}\right)$, so it takes one third of a second even with BigIntegers.

The pattern, in a geometrical sense, is this: we start with a best solution in the form of a literal exact cube, n^3. Then we add a "wall" of n^2 cubes to form a $(n+1)\,n^2$ cuboid. Next, we add another, perpendicular, "wall" of $n\,(n+1)$ cubes, resulting in an $(n+1)^2\,n$ cuboid. Finally we add a last "wall" (ceiling?) of $(n+1)^2$ cubes in the remaining dimension, finishing the $(n+1)^3$ cube, and we can iterate again.

And how do we fill these "walls"? There is again a simple pattern, which is basically the same but in 2D. First we put a cube in a corner of the last cuboid. Then we proceed in a similar iterative way: assume that we have filled a square portion with k^2 cubes. Then, fill one more line with k cubes, reaching a rectangle $k\,(k+1)$; this might be the end of the "wall" in some configurations, but if it's not, fill a perpendicular line with $k+1$ cubes, resulting in a $(k+1)^2$ square again. Rinse and repeat.

I was very surprised to learn that this approach was optimal, but numbers don't lie, and that's what I found after an experiment for $N \leq 2000$. Of course, this results in a regular pattern, which is especially exploitable since even the way we build a rectangle is exactly the same way used for the next one, save for a new line. We first notice that there are only three kind of additions:

- The first time we add a block to a corner of the cuboid to start a new wall, there is a single contact point between the cuboid and the cube. This means *two* sides hidden, therefore $g\,(n+1) = g\,(n) + 2$.
- When we are in the middle of building a wall and we start a new row or column, there are two contact points between the cuboid and the new cube, so $g\,(n+1) = g\,(n) + 4$.
- In any other case there are three contact points between the cuboid and the new cube, and $g\,(n+1) = g\,(n) + 6$.

This means that, for *every single wall*, there is an exact same sequence of increases in $g\,(x)$, which in the long term includes mostly 6s. In fact, to go from a $(n-1)^3$ cube to a n^3 one, there are exactly three cases where the difference is 2, whereas the amount of times the difference is 4 grows linearly with n, and the amount of times the difference is 6 grows with n^2. This means that, in the long term, $g\,(n) \lessapprox 6n$, and $G\,(n) \lessapprox 3n^2$.

Now, let's build the iterative scheme to calculate $G(n)$ in $O\left(n^{\frac{1}{3}}\right)$. The algorithm has two steps, one to fill "walls" and a last one to fill the last, unfinished one (needed since 10^{16} is not n^3, $n^2(n+1)$ nor $n(n+1)^2$ for any n). We start defining the sequence $d(n)$, meaning the value added to $g(x)$ when adding the nth element of a wall (note that this is not $g(n+1)-g(n)$, but rather, $g(x+n+1)-g(x+n)$ for certain values of x). This sequence can be built periodically:

$$d(1\ldots) = \{2,4,4,6,4,6,4,6,6,4,6,6,4,6,6,6,\ldots\}.$$

We start with a single 2 for the corner, then we add a 4 to fill the 1×2 rectangle; we continue with two $\{4,6\}$ pairs, one to get to 2×2 and another one to get to 2×3, and the pattern follows in the same way: at each time we add two tuples of increasing size (two tuples of length 3, then two tuples of length 4, and so on), where the first element is a 4 and all the next ones are 6. The pattern continues indefinitely.

From $b(n)$ we define two additional successions, $s(n)$ and $t(n)$. We start with

$$s(n) = \sum_{k=1}^{n} d(k),$$

the prefix sum of b. We only need $s(n)$ for certain values, and we can calculate them recursively.

Then we define $t(n)$, the prefix sum of $s(n)$:

$$t(n) = \sum_{k=1}^{n} s(k) = \sum_{k=1}^{n} (n+1-k)\,d(k).$$

This can also be calculated recursively for certain values, since the pattern is so simple.

Finding the general value of $s(n)$ and $t(n)$ has some nuance, but first we are interested in cases where we have $s(x)$ and $t(x)$ for certain x that happens to be the end of a row (i.e. a value of the form k^2 or $k(k+1)$ for some $k\geq 1$) and we add a whole new row of length l (meaning a single 4 and $l-1$ 6s). In these cases we can show that

$$\begin{aligned} s(x+l) &= s(x)+6l-2; \\ t(x+l) &= t(x)+l(s(x)-2)+3l(l+1). \end{aligned}$$

How do we proceed in the case where we want to add an incomplete row of length $l'<l$? Well, assuming that $l'\geq 1$, we can use the exact same formula,

since it's just an incomplete row of 6s. It's just that these resulting values will not represent the end of a row and thus can not be used to keep iterating, but the formula works just as well. We will need this formula later, for the final stage of the problem. For now we can think about how to calculate $G(n)$ for some n.

The procedure to calculate $G(n)$ for some n that happens to be k^3, $(k+1)k^2$ or $(k+1)^2 k$ is simple: we keep adding "walls", and we keep calculating $s(n)$ and $t(n)$ recursively. One can see that, if x is in one of the above cases, and we want to add a complete "wall" with a total of n cubes, we can use this formula:

$$G(x+n) = G(x) + ng(x) + t(n).$$

This is nice, but it means that we need to keep up with values of $g(x)$. Which can also be computed with the addition of these "walls", since in these cases we will have

$$g(x+n) = g(x) + s(n).$$

Which is very convenient, since we needed to calculate $s(n)$ anyway as a stepping stone for $t(n)$.

We now have all the formulas we need, and we can define an algorithm that solves the problem in $O\left(\sqrt[3]{N}\right)$ time.

- We will have "blocks" representing the state of the algorithm at certain values n, either squares or the other two cuboid cases. For each block we want to store the values of n, $g(n)$ and $G(n)$, as well as the last wall length used, l, along with $s(l)$ and $t(l)$.
- We will need two operations: one that "repeats" the same wall, resulting in a block for $n+l$, and another one that adds a length k to the walls, resulting in the information for a new wall of length $l+k$ and data for $n+l+k$. These operations update the values of the block using the formulas described above.
- We start iterating from some value of n^3 (I used 2^3, but 1^3 should work, I believe), including a wall length of n^2.
- Iterate for $i = n+1\ldots$:
  - First, repeat the last wall (of length $(i-1)^2$) to go from $(i-1)^3$ to $i(i-1)^2$.
  - Then increase the length of the wall in $i-1$ units, creating a wall of length $i(i-1)$ and moving to $i^2(i-1)$.

- Finally increase the wall in i units, so that the wall has length i^2 and we get the end result for i^3.

- We will end when at some of these steps we reach a value that is greater or equal to the problem input N. For the sample cases in the problem (18 and 10^6) this corresponds exactly to the end of one block, and we can return $G(N)$ immediately. Otherwise, we need an additional step:

  - Let x be the last complete cuboid, so that we need an incomplete wall of length $N-x$. In this case, calculate $t(N-x)$ iteratively (I recalculated it from scratch, which is $O\left((N-x)^{\frac{1}{3}}\right)$ as well, but I could have saved all the temporary values and calculated it from the last relevant value), and use

$$G(N) = G(x) + (N-x)\, g(x) + t(N-x).$$

That's it. There are several sequences on OEIS that are related to this problem, namely A193416 (minimum surface area for polycubes with volume n) and A007818 (maximal number of bonds joining n nodes in simple cubic lattice; this corresponds to $\frac{1}{2}g(n)$), which I used to find the initial pattern. Since $N^{\frac{1}{3}}$ is a bit above $2 \cdot 10^5$, the full result can be found in one third of a second (I assume that using longs with mod there is a bit of time to be gained, but probably not that much). The full result is

$$299998329026942697823463812134580,$$

which as expected is just a bit below $3 \cdot 10^{32}$.

776. Digit Sum Division

Difficulty rating: 25 %.
Solution: 9.627509725002e33. *Solved: Sun, 12 Dec 2021, 18:22.*
Math knowledge used: none.
Programming techniques used: dynamic programming.

Very easy problem. I'm assuming that it will be categorised as 10 or 15 %. The trick is to group numbers by the common factor $d(n)$, so that the sum becomes

$$\sum_{d=1}^{M} \frac{1}{d} \sum_{\{n:d(n)=d\}} n,$$

where M is the maximum digit sum allowed, typically reached cramming as many 9s as possible. It can be computed beforehand, or we can just use maps where the amount of digits is the key, so that any new value is accepted. Any number with N digits can never have a digit sum greater than $9N$, and for this problem, the number fits on a long, so the maximum is about 160. Now, the fact that the limit is not a power of 10 requires some additional tweaking over the basic dynamic programming. I used several different data structures, progressively more complex but never too big, to represent the data.

- The basic data point is a data struct called `CountAndSum` with just, well, the count of total numbers and its sum. It stores BigIntegers. Each combination of number of digits and total digit sum will be associated to one of these. If we have a count c and a sum s, associated with n digits whose sum is d, then we can add a digit a, so that the result will be associated to $n+1$ digits with sum $d+a$, with the new count being $c' = c$ and the new sum being $s' = 10s + ac$. Note that this is only a component of all the set of values with $n+1$ digits which add to $d+a$. There will be other sources (such as evolving the pair $(n, d-1)$ with the digit $a+1$, if $a < 9$), and all of them will need to be combined at the end.

- With this class we build another one, `Counters`, which has three separate `CountAndSum` objects, indicating whether the last digit is below, equal to or above the limit. This is needed because we operate digit by digit, so if the number is 9876, at the third digit (starting from the left) we want to account for the numbers that are below, equal to or above 7. So, when adding a digit, the "below" and "above" get accumulated to their homologous in the new object, but the "equal to" will be accumulated to one of the three sets, depending on how the current digit compares to the limit.

- The next class is called `GenerationalCounters` and it represents all the numbers with a fixed set of digits. It contains a map where the keys are the possible sums of digits, and the values are `Counters` objects representing the total set of objects with such sum. There will be one of these per amount of digits, and this is where the dynamic programming happens.

- Now we have a `FinalCounters` class, perhaps not very well named, which accumulates all the values where the sum of digits has certain value, regardless of the amount of digits. It stores counters for every amount of digits (with special care for the ones where the amount of

digits matches the limit of the problem, since in this case we need to take the limit into account), and when everything is finished, it condenses them into a single `Counters` object.

- Finally we have a `FullData` object with all the state of the problem. This object will use dynamic programming digit by digit to generate the `GenerationalCounters` object, filling the `FinalCounters` along the way. Finally it will be able to generate a summary object consisting of a map where the key is the amount of digit and the value is a `Counters` object (although it could as well be the final `BigInteger` with the sum), from which we can easily calculate the answer of the problem with the slightly tweaked formula.

I used `BigDecimal` just to be safe, but the amount of digits asked for is well below the size of the resulting number, so `BigInteger`s could have worked as well. The run time is about one tenth of a second.

777. Lissajous Curves

Difficulty rating: 75 %.

Solution: 2.533018434e23. *Solved: Mon, 10 Jan 2022, 06:07.*

Math knowledge used: trigonometric sum formulas, Möbius function, inclusion-exclusion, Erathostenes sieve (for divisor generation), Chinese remainder theorem, triangular numbers.

Programming techniques used: none.

All lucky 7s! This was also my seventh problem solved on this year, and I got to write the seventh post on the problem thread. Also, uh, I got a bug in the "solved problem" screen which was answered on page $35 = 5 \cdot 7$ of a forum thread. Oh, also this is $3 \cdot 7 \cdot 37$.

This problem is a curious one. It looks intimidating, but it's not really difficult. It does take a LOT of work, though. Right after seeing it I had a hunch about how to solve it, which turned out to be close to the truth even if I got some details wrong. My hunch was that:

- The intersection points were mostly periodical and relatively easy to pinpoint (correct).
- The intersection points would always belong to values where $t = \dfrac{n\pi}{ab}$ or maybe $t = \dfrac{n\pi}{2ab}$ or something like that (close but ultimately wrong).

- The intersection points would be arranged in such a way that $\sum x^2$ and $\sum y^2$ would correspond to linear formulas of the form $k_1 a + k_2 b + k_3$, probably with several cases depending on the prime factors of a and b (also wrong but close; there is an ab term. Right about the several formulas part: there is a special case when ab is a multiple of 10).

- $\sum x^2$ and $\sum y^2$ would either be needed to be summed separately because of completely different formulas, or complicated but in such a way that $\sum x^2 + y^2$ can be simplified (kind of wrong as well: all the formulas for $\sum x^2$ and $\sum y^2$ are pretty simple, and the easier thing to do is to sum them together just because the formula is still very simple. This part of the problem was actually much easier than I expected).

It's a bit surprising that more than three weeks after the problem publication there aren't 100 solvers yet (seriously, at this point it seems like the problem will be rated about 60 % or 70 %; baffling considering that there is no complex underlying math. **Update**: rated 75 %. Two months and a half after the problem publication, the amount of solvers sits at 84. I truly don't understand this). The problem can be solved in three steps, and all of them are manageable. The problem does look intimidating, but the presence of cosines hints at periodic and predictable locations. The steps are:

- Find the intersection points. This is not straightforward, but it's still doable with a bit of work. I did rely on a page somewhere that told the exact conditions forr the intersection (although said page has a mistake in one formula!).

- Use trigonometry to sum x and y, yielding a simple formula for $d\,(a, b)$. Prepare some pen and paper: there a lot of cases to consider although the formulas are simple.

- Sum $d\,(a, b)$ for all the pairs involved.

Each step has some complications, but none of them involves obscure mathematics. Even if you don't know the squared cosines summation formulas by heart, you can do like me and check some cases in Matlab or something to get the gist of it. And the formulas are indeed easy. The summation part is also easy, or at least it's nothing we haven't done in many other problems.

First step: finding the intersection points. I'm guessing that this is the part that prevents this problem from having too many solvers at this point, although it's more intimidating than difficult. Although there is an easily found page with the formulas, one can always separate the space $[0, 2\pi]$ into

ab equal parts and, for each pair of them, look for matches in x and y making use of the fact that

$$\cos x = \cos y \Leftrightarrow (x + y = 2K\pi) \wedge (x - y = 2K\pi)$$

One can try all the combinations of a and b multiples of 2, 5, or both, to account for the effects of that $\frac{\pi}{10}$ in the formula for y, but again, this is nothing that can't be easily automated with some exploratory code.

Now, assuming that a and b are coprime, the intersection points are found at exactly these values of t:

- If ab is not a multiple of 10 (common case), then there are two sets of intersections. Each one of these points is hit twice in the interval $t \in [0, 2\pi]$:
  - $t = \frac{n\pi}{ab}$ for integer $n \in [0, 2ab - 1]$ and $n \not\equiv 0 \pmod{b}$.
  - $t = \frac{(n + \alpha)\,\pi}{ab}$ where $\alpha = \frac{ab}{10}$, for integer $n \in [0, 2ab - 1]$ and $n \not\equiv 0 \pmod{a}$. These ones are probably the most difficult to find points, because of the added α term. They can still be found by doing some simple inspecting, though.
- If ab is a multiple of 10, the curve does something interesting: it has "corners", and it goes back to itself, so that there are two half cycles over the same curve, one moving forward and the other backwards. The "half cycles" don't start exactly at $t = 0$ and $t = \pi$, but we still have an exact full cycle in $[0, 2\pi]$. This means that we will pass *four* times, not two, over each point. There is, however, a single set of intersections, whose formula is similar to the previous one, but a bit more complicated to sum. The values of t are: $t = \frac{n\pi}{ab}$ for integer $n \in [0, 2ab - 1]$ and $n \not\equiv 0 \pmod{b}$ and $n \not\equiv \alpha \pmod{a}$ (note that α is now an integer). This is where lissajous.it has a mistake, since it wrongly says that the congruence is also to 0 modulo a. I lost a bit of time with this, being baffled as to why I was getting irrational values of $d\,(a, b)$, until I examined all the points and saw some weird behaviour. But again, this can be found with some tinkering.

Now, with the first part of the problem out, let's move on to the second one, which is finding formulas for $\sum (x^2 + y^2)$ for all these values of t. This is laborious and requires some inclusion-exclusion, but all the formulas are

essentially the same. There is one single magic formula that we are going to use, in several variations:

$$\forall N \in \mathbb{N}, \delta \in \mathbb{R}, N \geq 2 : \sum_{K=0}^{N-1} \cos^2 \frac{(K+\delta)\,\pi}{N} = \frac{N}{2}.$$

There are two useful corollaries for this. The first one is:

$$\forall N, M \in \mathbb{N}, \delta \in \mathbb{R}, N \geq 2 : \sum_{K=0}^{MN-1} \cos^2 \frac{(K+\delta)\,\pi}{N} = \frac{MN}{2}.$$

The second one is even more general; we introduce a coprime factor in the numerator.

$$\forall N, M, P \in \mathbb{N}, \delta \in \mathbb{R}, N \geq 2, \gcd(N,P) = 1 : \sum_{K=0}^{MN-1} \cos^2 \frac{(PK+\delta)\,\pi}{N} = \frac{MN}{2}.$$

The only special case is when the denominator is $N = 1$. In this case, if $\delta = 0$ all the cosines are $(-1)^n$ for some integer n, and the sum ends being $MN = M$ instead of $\frac{MN}{2}$. If $n = 1$ and $\delta \neq 0$ it gets complicated, but that case doesn't appear in this problem.

It's easy to see how all the summation formulas end belonging to these cases. Basically, as long as we are summing over a range $[0, n_1 - 1]$ and the denominator is some n_2 that divides n_1, the sum will be equal to $\frac{n_1}{2}$, save for the special case. And yes, this means that the additional shifts are not going to be a problem. We can now start summing all the cases; there are a lot but all of them follow the same pattern.

We start from the standard case, $ab \not\equiv 0 \pmod{10}$. There are two sets of points, and for each one of them there are some points that we need to subtract because they verify the modulo. We start defining these sets:

$$\begin{aligned}
t_{1+} &= \left\{ \frac{n\pi}{ab} : n \in \mathbb{Z}, n \in [0, 2ab-1] \right\}; \\
t_{1-} &= \left\{ \frac{n\pi}{ab} : n \in \mathbb{Z}, n \in [0, 2ab-1], n \equiv 0 \pmod{b} \right\}; \\
t_{2+} &= \left\{ \frac{(n+\alpha)\,\pi}{ab} : n \in \mathbb{Z}, n \in [0, 2ab-1] \right\}; \\
t_{2-} &= \left\{ \frac{(n+\alpha)\,\pi}{ab} : n \in \mathbb{Z}, n \in [0, 2ab-1], n \equiv 0 \pmod{a} \right\}.
\end{aligned}$$

For each one of these cases we also define

$$\begin{aligned} x_* &= \{\cos at : t \in t_*\}; \\ y_* &= \left\{\cos b\left(t - \frac{\pi}{10}\right) : t \in t_*\right\}. \end{aligned}$$

With these definitions, we have (for a and b coprime with $ab \not\equiv 0 \pmod{10}$) that

$$2d(a,b) = \sum_{x\in x_{1+}} x^2 - \sum_{x\in x_{1-}} x^2 + \sum_{x\in x_{2+}} x^2 - \sum_{x\in x_{2+}} x^2 + \sum_{y\in y_{1+}} y^2 - \sum_{y\in y_{1-}} y^2 + \sum_{y\in y_{2+}} y^2 - \sum_{y\in y_{2-}} y^2.$$

Note that we must divide this result times two to get the actual $d(a,b)$ because every point will be counted twice. At any rate, all of these summatories are of the same kind and can be very easily added. First of all note that we can redefine t_{1-} and t_{2-} as

$$\begin{aligned} t_{1-} &= \left\{\frac{n\pi}{a} : n \in \mathbb{Z}, n \in [0, 2a-1]\right\}; \\ t_{2-} &= \left\{\frac{n\pi}{b} + \frac{\pi}{10} : n \in \mathbb{Z}, n \in [0, 2b-1]\right\}. \end{aligned}$$

No need to worry about these values getting out of the range $[0, 2\pi]$ because any cycle will result on the exact same cosine values. So, with these transformations, all of the summatories can be calculated easily with the squared cosines formula:

$$\begin{aligned} \sum_{x\in x_{1+}} x^2 = \sum_{n=0}^{2ab-1} \cos^2\frac{n\pi}{b} &= ab; \\ \sum_{x\in x_{1-}} x^2 = \sum_{n=0}^{2a-1} \cos^2 n\pi &= 2a; \\ \sum_{x\in x_{2+}} x^2 = \sum_{n=0}^{2ab-1} \cos^2\left(\frac{n\pi}{b} + \frac{\alpha\pi}{b}\right) &= ab; \\ \sum_{x\in x_{2-}} x^2 = \sum_{n=0}^{2b-1} \cos^2\left(\frac{an\pi}{b} + \frac{a\pi}{10}\right) &= b; \\ \sum_{y\in y_{1+}} y^2 = \sum_{n=0}^{2ab-1} \cos^2\left(\frac{n\pi}{a} - \frac{b\pi}{10}\right) &= ab; \end{aligned}$$

$$\begin{aligned}\sum_{y\in y_{1-}} y^2 = \sum_{n=0}^{2a-1} \cos^2\left(\frac{bn\pi}{a} - \frac{b\pi}{10}\right) &= a;\\ \sum_{y\in y_{2+}} y^2 = \sum_{n=0}^{2ab-1} \cos^2\frac{n\pi}{a} &= ab;\\ \sum_{y\in y_{2-}} y^2 = \sum_{n=0}^{2b-1} \cos^2 n\pi &= 2b.\end{aligned}$$

Thus we end with a delightfully simple formula:

$$ab \not\equiv 0 \pmod{10} \Rightarrow d(a,b) = 2ab - \frac{3(a+b)}{2}.$$

The case where $ab \equiv 0 \pmod{10}$ is slightly more complicated despite there being only one set of intersections. This is because there are two conditions in this set, so that we need to use inclusion-exclusion and get 4 terms. Again we are going to end with eight different terms for the calculation of $d(a,b)$, and this time we need to take care of the fact that the divisor is 4 and not 2. Let's start by defining some sets of values for t, like before; and let's use blood type names, since they are surprisingly fitting:

$$\begin{aligned}t_0 &= \left\{\frac{n\pi}{ab} : n\in\mathbb{Z}, n\in[0,2ab-1]\right\};\\ t_B &= \left\{\frac{n\pi}{ab} : n\in\mathbb{Z}, n\in[0,2ab-1], n\equiv 0 \pmod{b}\right\};\\ t_A &= \left\{\frac{n\pi}{ab} : n\in\mathbb{Z}, n\in[0,2ab-1], n\equiv \alpha \pmod{a}\right\};\\ t_{AB} &= t_A \cap t_B.\end{aligned}$$

Of course we also define again the values of x_* and y_* for all the sets t_*. Some of these sets are the same as in the previous case: t_0 is the same as our previous t_{1+}; t_B is the same as t_{1-}; and t_A is the same as t_{2-} (this might not be obvious, but it's true thanks to the cyclic properties of the sets we're working with). For t_{AB} the result is different, but quite simple; and we can even define it by extension since it has exactly two elements. Let m be the solution of the following diophantine equation system in the range $[0, ab]$ (remember that a and b are coprime by definition!):

$$\begin{aligned}m &\equiv 0 \pmod{b},\\ m &\equiv \alpha \pmod{a}.\end{aligned}$$

This value of m doesn't have a surefire method of calculation besides solving the equation by the usual means (for example, if $a = 5$ and $b = 4$, the solution is $m = 12$. In other cases there is an obvious solution $m = b$, but this doesn't always work). So we will have

$$t_{AB} = \left\{ \frac{m\pi}{ab}, \frac{m\pi}{ab} + \pi \right\}.$$

These points are the "corners" of the curve, where it "bounces" back to repeat itself in the opposite direction. Note that these points are hit only twice at each cycle, not four, which is expected because there aren't points where the curve crosses itself: these are just points that we need to take into account during inclusion-exclusion, because they are subtracted twice (once because of t_B and once again because of t_A) and we need to add them again. Now, we have on one side that

$$x = \cos a \frac{m\pi}{ab} = \cos \frac{m\pi}{b};$$

however, since $m \equiv 0 \pmod{b}$, this is the cosine of an integer multiplied by π (and the same is true for $m + ab$, clearly), so $x^2 = 1$ in both cases. For y we have that

$$y = \cos b \left(\frac{m\pi}{ab} - \frac{\pi}{10} \right) = \cos \frac{10m - ab}{10a} \pi,$$

and since $m \equiv \alpha = \dfrac{ab}{10} \pmod{a}$, we have also $10m \equiv ab \pmod{10a}$, so the argument of the cosine is, again, an integer multiple of π. Therefore we also have $y^2 = 1$ at these points! This has been a more formal proof, but this is obvious if you look at the curve, because these points are the only places where the curve is not smooth, and they are always located at two of the four corners of the $[-1, 1] \times [-1, 1]$ square.

Now, let's go back to the definition of $d(a, b)$. In a similar way than the standard case, we have

$$4d(a,b) = \sum_{x \in x_0} x^2 - \sum_{x \in x_B} x^2 - \sum_{x \in x_A} x^2 + \sum_{x \in x_{AB}} x^2 + \sum_{y \in y_0} y^2 - \sum_{y \in y_B} y^2 - \sum_{y \in y_A} y^2 + \sum_{y \in y_{AB}} y^2.$$

Except for the particular case of t_{AB}, the summations are similar than in the standard case:

$$\sum_{x \in x_0} x^2 = \sum_{x \in x_{1+}} x^2 \quad = \quad ab;$$

$$\begin{aligned}
\sum_{x\in x_B} x^2 = \sum_{x\in x_{1-}} x^2 &= 2a;\\
\sum_{x\in x_A} x^2 = \sum_{x\in x_{2-}} x^2 &= b;\\
\sum_{x\in x_{AB}} x^2 &= 2;\\
\sum_{y\in x_0} y^2 = \sum_{y\in y_{1+}} y^2 &= ab;\\
\sum_{y\in x_B} y^2 = \sum_{y\in y_{1-}} y^2 &= a;\\
\sum_{y\in x_A} y^2 = \sum_{y\in y_{2-}} y^2 &= 2b;\\
\sum_{y\in y_{AB}} y^2 &= 2.
\end{aligned}$$

And here we have it, the final formula we were looking for, similar to the other one we had found:

$$ab \equiv 0 \pmod{10} \Rightarrow d(a,b) = \frac{2ab - 3(a+b) + 4}{4}.$$

And with this, we have the complete formula for $d(a,b)$, which is

$$d(a,b) = \begin{cases} \dfrac{2ab - 3(a+b) + 4}{4} & \text{if} \quad ab \equiv 0 \pmod{10};\\ 2ab - \dfrac{3(a+b)}{2} & \text{if} \quad ab \not\equiv 0 \pmod{10}. \end{cases}$$

And that's the most complicated part of the problem. I'm not going to say that this was simple or fast, but it's more about being laborious than anything else. This formula assumes that a and b are coprime, though; the general case could be a bit more complicated.

We are almost at the end of the problem, but we haven't finished yet. We need to calculate the sum of all $d(a,b)$ for coprime pairs of (a,b) in the range $2 \leq a, b \leq N$ for $N = 10^6$. Ostensibly, the amount of terms is of the order of N^2, so calculating all of them is not a good solution (it should be doable in about 6 hours of my machine, but that would still be very suboptimal). Instead, let's try iterating for every value of a, and for each one of them, let's calculate the sum of $d(a,b)$. This is a bit annoying, because we can't apply Faulhaber's formulas blindly or anything like that. Since we need to exclude the cases where a and b are not coprime (note the rewording, wink wink

nudge nudge), we can use inclusion-exclusion with a sorta Möbius function approach. We also need to consider the fact that the formula is different if ab is a multiple of 10. So what we are going to do is defining two functions that operate with fixed multiples of b, so that for a fixed a and certain b we get the sum of $d(a,b) + d(a,2b) + d(a,3b) + \ldots$. We start with the two expressions for $d(a,b)$:

$$\begin{aligned} f(a,b) &= 2ab - \frac{3(a+b)}{2}; \\ g(a,b) &= \frac{2ab - 3(a+b) + 4}{4}. \end{aligned}$$

From these, we define:

$$\begin{aligned} F(a,m,N) &= \sum_{i=1}^{k} f(a,im); \\ G(a,m,N) &= \sum_{i=1}^{k} g(a,im). \end{aligned}$$

Here, $k = \left\lfloor \frac{N}{M} \right\rfloor$ is the amount of terms in the sum. These functions have simple expressions in terms of a and m. We can use the fact that the sum of $m + 2m + \ldots + km$ is m multiplied by the triangular number $T_k = \frac{k(k+1)}{2}$. Thus we have

$$\begin{aligned} F(a,m,N) &= 2amT_k - \frac{3}{2}(ak + mT_k); \\ G(a,m,N) &= \frac{2amT_k - 3(ak + mT_k) + 4k}{4}. \end{aligned}$$

With these building blocks we can calculate a function that, for each value of a, gives us the value

$$s_a(N) = \sum_{\substack{2 \le b \le N \\ \gcd(a,b)=1}} d(a,b).$$

The first thing we are going to do is to find the prime factors of a, and from these, we can generate a lattice of their products (no need to consider repeated powers. That's what Möbius function tells us!). And then we are going to use several different formulas, depending on which primes are shared bet-

ween 10 and a. So, with S_a being the set of numbers that divide a and whose Möbius function is ± 1, we have these different cases. Remember to subtract $f(a,1)$ from each case, since this inclusion-exclusion approach includes all the values starting from $b=1$, but the problem asks for cases where $b \geq 2$:

- If $a \equiv 0 \pmod{10}$, we only need to consider g. So we have

$$s_a(N) = -f(a,1) + \sum_{m \in S_a} G(a,m,N)\,\mu(m).$$

  For every other value, we will first calculate the sum of f, then subtract the terms we have added incorrectly because ab is a multiple of 10, then add the sum of g. This means that the other three cases follow the same pattern.

- If $a \not\equiv 0 \pmod{10}$, but $a \equiv 0 \pmod{5}$, then

$$s_a(N) = -f(a,1) + \sum_{m \in S_a} \left(F(a,m,N) - F(a,2m,N) + G(a,2m,N)\right)\mu(m).$$

- Similarly, when $a \not\equiv 0 \pmod{10}$, but $a \equiv 0 \pmod{2}$, then

$$s_a(N) = -f(a,1) + \sum_{m \in S_a} \left(F(a,m,N) - F(a,5m,N) + G(a,5m,N)\right)\mu(m).$$

- Finally, if $a \not\equiv 0 \pmod{5}$ and $a \not\equiv 0 \pmod{2}$, then

$$s_a(N) = -f(a,1) + \sum_{m \in S_a} \left(F(a,m,N) - F(a,10m,N) + G(a,10m,N)\right)\mu(m).$$

That's it. After so much work and so many formulas, we have a simple algorithm that calculates the value asked for in about $O(N \log N)$ time, very fast for $N = 10^6$:

- Create a prime sieve for fast divisor calculation.
- Initialise the problem result as 0.
- Iterate for values of $a = 2$ to N.
- For each value of a, get the primes that divide a and use them to build the set S_a along with the Möbius function values.

- Use the corresponding formula to calculate $s_a(N)$ and add it to the problem result.

And that's it. This was a very complete problem requiring considerable pen and paper work, but man was it fun. At first I got a wrong solution, because "10 significant digits" referred to 9 decimal digits, not 10, but otherwise I got the right solution at the first try.

778. Freshman's Product

Difficulty rating: 30 %.
Solution: 146133880. *Solved: Sun, 26 Dec 2021, 12:52.*
Math knowledge used: none.
Programming techniques used: binary exponentiation.

Extremely simple problem. You can work independently on each digit thanks to the absence of carries, and the problem has only 6 digits. The run time is about $O(\log M)$ thanks to the binary exponentiation. We only need two data structures, working only in terms of mods (BigIntegers can be used as well, but the numbers get huge very fast):

- A `SingleDigitDistribution`, representing the distribution of each one of the measly 6 digits we need to consider. Each one of these objects has an array of ten longs, with each one indicating how many numbers have the index (i.e. $n \in [0, 9]$) as its digit in the order represented by this object. This object has a product function so that distributions can be multiplied. This product is associative, which is important because it allows binary exponentiation.

- And a `CompleteDistribution` which has 6 objects of the previous kind, one per digit, representing a full set of numbers. This object is initialised with a method that partitions the input number (i.e., 234567), and it also allows an associative product, which is simply the product of the single digit distributions, digit by digit.

We will proceed as follows: first, we partition the number $R = 234567$ into a complete distribution. Now, the product of these distributions represents the $\boxtimes$ operator of every pair of numbers. So, if we use this "product" with a binary exponentiation with exponent $M = 765432$, we can get the result in $O(\log M)$ time, as stated above. Then the final result can be obtained easily from the resulting object. It takes less than 50 milliseconds.

779. Prime Factor and Exponent

Difficulty rating: 25 %.

Solution: 0.547326103833. *Solved: Sat, 1 Jan 2022, 22:13.*

Math knowledge used: geometric series, arithmetic-geometric series, Erathostenes' sieve.

Programming techniques used: none.

I lost a lot of time on this very easy problem because of a very stupid mistake (I added an extra factor in a function, and I kept getting the wrong result). In fact the result can be found quickly with a very simple iterative process.

We start with the basic formula asked in the problem,

$$S = \sum_{K=1}^{\infty} f_k = \sum_{K=1}^{\infty} \lim_{N\to\infty} \frac{1}{N} \sum_{n=2}^{N} \frac{\alpha(n) - 1}{(p(n))^K}.$$

This looks unmanageable, but we need a different approach. Let's group by values of $p(n)$ and $\alpha(n)$. We can easily see that, for a given pair (p, α), the numbers n such that $p(n) = p$ and $\alpha(n) = \alpha$ follow a set of arithmetic progressions. For example, for $p = 2$ and $\alpha = 2$, the succession is $\{4, 12, 20, \ldots\}$, that is: $\{8n + 4\}$ for every n. For bigger primes the successions are a bit more complex. For example, for $p = 5$ and $\alpha = 1$ the numbers are: $\{5, 25, 35, 55, \ldots\}$. These are two interleaved successions, $\{30n + 5\}$ and $\{30n + 25\}$. Now, this means that exactly one in 8 numbers verifies $p(n) = 2$ and $\alpha(n) = 2$, and two in 30 numbers verify $p(n) = 5$ and $\alpha(n) = 1$. We can compute these ratios with reasonable ease. Let's define $f(p, \alpha)$ as the ratio of numbers $n \in \mathbb{N}$ for which $p(n) = p$ and $\alpha(n) = \alpha$, so that $f(2, 2) = \frac{1}{8}$ and $f(5, 1) = \frac{2}{30}$. So, the problem becomes

$$S = \sum_{K=1}^{\infty} \sum_{p\in\mathbb{P}} \sum_{\alpha=1}^{\infty} \frac{(\alpha - 1) f(p, \alpha)}{p^K},$$

where $\mathbb{P}$ is the set of all primes. Ok, we now have a three-dimension summatory instead of a two-dimension one, but the limit is gone and the formula inside the summatory doesn't look too bad. We need a proper expression for the function f, but before that, let's define some related functions.

- First we define $g(p, \alpha)$ as the ratio of numbers such that $f(n) = p$ and $\alpha(n) \geq \alpha$ (note the $\geq$ symbol instead of $=$).

- We are also going to need the function $h(p)$, defined for every prime, and representing the ratio of all the numbers which are not a multiple of any prime lower than p.

We can easily see that $h(2) = 1$. For every other prime p_i, we use $h(p_i) = h(p_{i-1})\left(1 - \frac{1}{p_{i-1}}\right)$. We can see that, for every $\alpha \geq 1$, $g(p, \alpha) = \frac{1}{p^\alpha} h(p)$. Finally, $f(p, \alpha) = \frac{p-1}{p} g(p, \alpha)$. Therefore, $f(p, \alpha) = \frac{p-1}{p^{\alpha+1}} h(p)$. Let's plug this into the formula for the problem solution S:

$$S = \sum_{K=1}^{\infty} \sum_{p \in \mathbb{P}} \sum_{\alpha=1}^{\infty} \frac{(\alpha - 1) \frac{p-1}{p^{\alpha+1}} h(p)}{p^K}.$$

Ok, this is interesting. Let's separate the parts where α appears, so that they can be summed easily:

$$S = \sum_{K=1}^{\infty} \sum_{p \in \mathbb{P}} \frac{(p-1) h(p)}{p^K} \sum_{\alpha=1}^{\infty} \frac{\alpha - 1}{p^{\alpha+1}}.$$

If we focus on the inner summatory and do a small shift, we get

$$\sum_{\alpha=1}^{\infty} \frac{\alpha - 1}{p^{\alpha+1}} = \sum_{\alpha=0}^{\infty} \frac{\alpha}{p^{\alpha+2}} = \frac{1}{p^2} \sum_{\alpha=0}^{\infty} \frac{\alpha}{p^\alpha}.$$

What we have left inside the summation is an arithmetic-geometric series with $r = \frac{1}{p}$, which converges because $\frac{1}{p} < 1$, and whose sum is famously $\frac{r}{(1-r)^2}$. Therefore we can simplify this as

$$\sum_{\alpha=1}^{\infty} \frac{\alpha - 1}{p^{\alpha+1}} = \frac{1}{p^2} \frac{\frac{1}{p}}{\left(1 - \frac{1}{p}\right)^2} = \frac{1}{p(p-1)^2}.$$

We are now left with "only" a two-dimensional summation:

$$S = \sum_{K=1}^{\infty} \sum_{p \in \mathbb{P}} \frac{(p-1) h(p)}{p^K} \cdot \frac{1}{p(p-1)^2} = \sum_{K=1}^{\infty} \sum_{p \in \mathbb{P}} \frac{h(p)}{p^{K+1}(p-1)}.$$

That p^K in the denominator is asking for a simplification, so we now make use of the happy fact that the summation limits are unrelated and we can just switch the order:

$$S = \sum_{p\in\mathbb{P}} \sum_{K=1}^{\infty} \frac{h\left(p\right)}{p^{K+1}\left(p-1\right)} = \sum_{p\in\mathbb{P}} \frac{h\left(p\right)}{\left(p-1\right)} \sum_{K=1}^{\infty} \frac{1}{p^{K+1}}.$$

The inside of the inner summatory is, blatantly, an arithmetic series with, again, $r = \frac{1}{p}$. Once again we have $r < 1$ and therefore the series converge. The sum of this series is $\frac{a_0}{1-r}$, where a_0 is the first term of the summatory, $\frac{1}{p^2}$. That is, the sum is

$$\sum_{K=1}^{\infty} \frac{1}{p^{K+1}} = \frac{\frac{1}{p^2}}{1-\frac{1}{p}} = \frac{1}{p\left(p-1\right)}.$$

This leaves us with

$$S = \sum_{p\in\mathbb{P}} \frac{h\left(p\right)}{\left(p-1\right)} \cdot \frac{1}{p\left(p-1\right)} = \sum_{p\in\mathbb{P}} \frac{h\left(p\right)}{p\left(p-1\right)^2},$$

and this is enough to start iterating. The full procedure is just: use the Erathostenes' sieve to get all the primes up to some limit L, and for each prime, calculate $h\left(p\right)$, calculate the rest of the term inside the summatory, and keep adding. The result converges far faster than one might think: using $L = 10^5$ is enough to get 12 digits correctly. The run time is a bit above 20 milliseconds, including the sieve generation. And yes, doubles are enough, no need for BigDecimal and those things.

By the way, using the formula $\sum_{p\in\mathbb{P}} \frac{h\left(p\right)}{p^2\left(p-1\right)}$ instead of $\sum_{p\in\mathbb{P}} \frac{h\left(p\right)}{p\left(p-1\right)^2}$ results in f_1, the sample value given by the problem definition. I think it converges even faster.

784. Reciprocal Pairs

Difficulty rating: 30 %.
Solution: 58333030125764292 31. *Solved: Sun, 6 Feb 2022, 06:02.*

Math knowledge used: binary quadratic forms.

Programming techniques used: Alpertron (includes a prime sieve).

I'm quite proud of this, because I got position 17, which is the highest I've ever got, but my first solution was actually pretty bad.

Let p, q, r be numbers such that r is both the modular inverse of $p \pmod q$ and the modular inverse of $q \pmod p$. Then, there are some integers k_1 and k_2 such that

$$\begin{aligned} rp &= 1 + qk_1, \\ rq &= 1 + pk_2. \end{aligned}$$

Subtracting these equations, we get

$$r\left(p - q\right) = qk_1 - pk_2,$$

which we can rewrite as

$$p\left(r + k_2\right) = q\left(r + k_1\right).$$

Since the modular inverses exist, p and q must be coprime. This means that the equation above can only be satisfied when

$$\begin{aligned} p &= r + k_1, \\ q &= r + k_2. \end{aligned}$$

(This is not strictly true. There might be additional solutions, if $r + k_2 = nq$ and $r + k_1 = np$. These don't intervene in this problem because $r < p$).

We aren't actually going to use the second equation; the first one is enough. From that one, we note that $k_1 = p - r$, and from the first of the initial equations we get

$$rp = 1 + q\left(p - r\right).$$

If we treat p as a constant, P, we have a binary quadratic form in q and r:

$$qr - Pq + Pr - 1 = 0.$$

My first solution was simple: iterate for values of p, call Alpertron to solve this equation, accept all the results if $q > p$. It's pretty suboptimal, though; almost two minutes of run time. So, considering that all of these equations are of the same type (simple hyperbolic equations, with $A = C = 0$; and $B = 1$ for added insult), I rewrote the code for this subtype of binary forms, tailored

to this problem. It turns out that the solution is really simple. Starting with any value of p, we get all the factors of $p^2 - 1$ (this is one of the main simplifications: I can use the fact that $p^2 - 1 = (p-1)(p+1)$ and avoid decomposing thirteen digit numbers; combining the decompositions of two smaller numbers is much faster since I can use a prime sieve for that). If d is a factor of $p^2 - 1$, then $q = \dfrac{d-p}{B}$ and $r = \dfrac{\frac{p^2-1}{d} + p}{B}$. Well, in this problem it so happens that $B = 1$, so no need to check whether these divisions are exact or not: they always are. The solution doesn't make use of r, so I just enumerate the divisors of $p^2 - 1$, subtract p to each divisor to get a potential value of q, and add $p + q$ to the solution if $p < q$. Between the faster decompositions, the usage of longs instead of BigIntegers and the lack of divisions, the run time is like 9 times less (13 seconds).

785. Symmetric Diophantine equation

Difficulty rating: 55 %.

Solution: 29526986315080920. *Solved: Sun, 20 Feb 2022, 08:15.*

Math knowledge used: binary quadratic forms, convergents, Tonelli-Shanks, Hensel lifting, Chinese remainder theorem.

Programming techniques used: Alpertron (includes a prime sieve).

I'm not especially proud of my solution (75 minutes of run time), but at least I got to solve this. It's pretty hard (one week after publication, the amount of solvers hadn't got yet to 100) so I expect a final rating of about 60 or 65 %, and I invested a lot of time to find a solution which ultimately relies too much on brute force, even if the calculations include some decently clever shortcuts and sieves. The solution is more or less $O(n)$ (maybe even $O(n \log n)$, probably), but with a very high constant, so that $n = 10^9$ is still quite slow.

After some experiments it becomes clear that, for each solution of the proposed equation, there exists some parameter q that verifies the following equalities:

$$\begin{aligned} x + y + z &= 8q, \\ xy + xz + yz &= 15q^2, \\ x^2 + y^2 + z^2 &= 34q^2. \end{aligned}$$

The same experiments provide very interesting data used to avoid recalculations: $z > 5q$, therefore we only need to use values of q below $\dfrac{N}{5} = 2 \cdot 10^8$.

Also, q is always odd, and all its prime factors are either 3 (which doesn't appear squared) or primes of the form $6n+1$. There are still a lot of values where these conditions hold, but it's still a sizable improvement over the 10^9 initial range.

Now, back to these equations. The second and third ones are directly related to the original one, but for the subsequent calculations we are less interested in the second, and more in the first one. The gist of the procedure is something like this: iterate over values of q, and for each of them, find suitable values of x, and find y and z from that. Alpertron and convergents will be involved.

Let's say that q is a constant. And let's say that we have found a valid x, so that it's also a constant. Then we have

$$\begin{aligned} y+z &= 8q-x, \\ y^2+z^2 &= 34q^2-x^2. \end{aligned}$$

From the first equation we have

$$y^2+2yz+z^2 = 64q^2-16qx+x^2,$$

so that subtracting the second equation and dividing by two we get

$$yz = 15q^2-8qx+x^2.$$

So we have the sum and product of y and z as formulas of q and x. Hmmm. We know then that y and z will be the solutions of this quadratic equation in w:

$$w^2-(y+z)\,w+yz=0 \Rightarrow w^2-(8q-x)\,w+\left(15q^2-8qx+x^2\right)=0.$$

We need the determinant of the equation to be a perfect square. This determinant is equal to

$$(8q-x)^2-4\left(15q^2-8qx+x^2\right)=4q^2+16qx-3x^2;$$

also, by the way, this determinant shares parity with x, and so does $8q-x$, therefore the only condition we need for the equation in w to have integer solutions is that the determinant is indeed a square. The solutions will be

$$\{y,z\}=\frac{8q-x\pm\sqrt{4q^2+16qx-3x^2}}{2}.$$

Now we want to find valid values of x. Let's treat q as a constant. We need to find values such that the expression $4q^2 + 16qx - 3x^2$ is a perfect square. Let's say $4q^2 + 16qx - 3x^2 = \Delta^2$ for some Δ. You know where this is going. Let's rewrite the equation with $\Delta = a$ and $x = b$, for clarity (initially I used $x = a$ and $\Delta = b$. It turns out that my solver is such that having the first coefficient be equal to 1 simplifies things a lot, so I resorted to this other denomination):

$$a^2 + 3b^2 - 16qa - 4q^2 = 0.$$

Note that q is a constant. So, yes, this is a binary quadratic form. So, yes, Alpertron. Unfortunately, the general Alpertron is far too slow because it's too general, so I had to reverse engineer my own code and extract the relevant bits with some tweaks for speed. The first relevant transformation that Alpertron does is a Lagrange transform that, long story short, translates this equation into

$$a'^2 + 3b'^2 = 57 \cdot 64q^2.$$

We can recover the solutions we want by doing $\Delta = -\dfrac{a'}{12}$ and $x = \dfrac{32q - b'}{12}$, provided that these values are integers. There is a solution with changed signs, but we are only interested in these cases because of the condition $y < z$, which implies a positive Δ (there is an additional check down the line so that a' is *negative*. As confusing as all this might be, it's still much better than the alternative).

So the equation above is first solved by doing $T^2 + 3 \equiv 0 \pmod{57 \cdot 64q^2}$, which can be solved for any generic value of q by a combination of Tonelli-Shanks, Chinese remainder theorem and Hensel lifting. Long story short, we need to decompose $57 \cdot 64q^2$ as dr for some integer values d and r, so knowing that this number is in fact $57 \cdot 64 \cdot q^2$ is very helpful. The standard Alpertron doesn't have a simple way of incorporating this piece of knowledge, but the deconstructed one I'm implementing does. Also, here goes another fruity piece of knowledge that the original Alpertron can't incorporate: in order for this to yield a valid solution, we need that $d \in \{4, 8\}$ if q is not a multiple of 3, or $d \in \{12, 24\}$ if it is. This is also a huge improvement over the original Alpertron, which checks all the decompositions and then forces us to discard the majority of the solutions because they are not primitive.

Ok, let's say that we have some d and r, and we find an n such that $n^2 \equiv -3 \pmod{r}$ using the classic combination of Tonelli-Shanks, Hensel lifting and Chinese remainder theorem. We are just starting. Now it's the time for CONVERGENTS! This is the kind of magic internal to Alpertron that I needed to replicate. First we define $P = \dfrac{n^2 + 3}{r}$, an integer by definition,

and then $Q = 2n$ and, uh, $R = r$ (this is because the very convenient main coefficient equal to 1. This is not the only advantage; it also avoids the need for unimodular transforms). We need that P, Q and R are coprime (not pairwise, just the three of them. It's very common that two of them are even, for example). If they are, we find the (thankfully finite, because this is an elliptic equation) convergents of the fraction $\frac{Q}{2P}$. We need to find a convergent $\frac{p}{q}$ such that $Pp^2 - Qpq + Rq^2 = 1$. Then the solutions os the equation are $a' = d\,(np - rq)$, $b' = dp$. Some pen and paper reveals that we also need $a' < 0$ and $3b' + a' > 0$ to guarantee that $x < y < z$. We can finally do $x = \frac{32q - b'}{12}$ and $\Delta = -\frac{a'}{12}$, provided that they are indeed integers, and finally, $y = \frac{8q - x - \Delta}{2}$, $z = \frac{8q - x + \Delta}{2}$. Accept the solution if $z < 10^9$ and if $\{x, y, z\}$ are coprime, in whose case we can add $x + y + z = 8q$ to the solution.

I spent most of the time reverse engineering my Alpertron, but also chasing overflows (in the end I changed most of the code to BigIntegers, which is probably mostly unnecessary, but my overflow was happening on the one and only operation with 32 bit integers I was inadvertently doing) and, well, running the code (more than one hour. It's still better than a code that doesn't run or doesn't return the valid answer). I know that my solution is not a very good one, but it's not far off to the ones proposed in the problem thread, which use also approaches based on binary quadratic forms, although usually more clever and resulting in actual parameter successions instead of millions of equations.

787. Bézout's Game

Difficulty rating: 40 %.
Solution: 202642367520564145. *Solved: Fri, 15 Apr 2022, 20:23.*
Math knowledge used: Erathostenes' sieve, inclusion-exclusion.
Programming techniques used: none.

I have to admit that I'm a bit pissed off about this problem. This looked an awfully lot like it would have much in common with 433. So I started working on that one, hoping that in the comments I would find something that could help me with this one. It turns out that getting a good solution for that one is fucking hard, and everyone and their mothers are bruteforcing it. So did I (5 hours of run time. That's after separating the workload in 17 fucking threads. My PC's cooling system certainly earned its pay). By

the way, the first *good* solution in that problem thread earned 52 kudos, which I believe to be the highest I've ever seen in a problem after the first 100 (*that* comment from Lucy Hedhehog came in problem 10, so I'm not counting it). The solution relied on a fucking obscure paper that I haven't been able to locate. If brute force hadn't been feasible (seriosly, people in the comments talk about clusters and using MapReduce), that one would have easily been a 100 % one. No Fibonaccis or Zeckendorf magic (although continuous fractions are relevant); only postdoc weird shit. And now let's go back to 787. It has been solved by quite some people, indicating that it's far easier, therefore probably not related to 433. So I start experimenting and I found the following:

(a, b) *is a winning position if and only if* $\min\{a, b\}$ *is odd.*

ARE YOU FUCKING KIDDING ME? No wonder this has been solved by so many people. It's far easier than it looks. There is still a bit of actual math involved, but not much, and certainly not something I haven't needed for any other problem. By the way: although the problem insists that order matters, it's pretty clear that it doesn't (it was clear before this realisation, in fact. The problem is very symmetrical). For each (a, b) pair there are exactly two (c, d) pairs, one for which $ad - bc$ equals 1 and other one for which it equals -1. They also verify $c_1 + c_{-1} = a$ and $d_1 + d_{-1} = b$.

Now let's define the counting scheme. We only need to count coprime pairs, which is about the only difficulty in this problem, and that can be done with the coprime counter I used for 754. Let's say that we want to count numbers that are coprime with N in an interval $[N_0, N_f]$ (we can call this function $S(N, N_0, N_f)$). Then we enumerate the prime factors of N, we create a prime lattice and for each squarefree factor x that divides N, including 1, we add $\mu(d)\left(\left\lfloor \frac{N_f}{x} \right\rfloor - \left\lfloor \frac{N_0 - 1}{x} \right\rfloor\right)$ to a counter. The prime factors of N, and therefore the prime lattice, can be generated with a "first prime" sieve in logarithmic time. And with this efficient function, the problem result is

$$1 + 2 \sum_{\substack{i=1 \\ i \text{ odd}}}^{N/2} S(i, i+1, N-i).$$

Since the limit $N = 10^9$ is high, this takes around 30 seconds to run. In the problem thread people are using clever tricks to avoid using the explicit prime factorisation, resulting in more complex code that runs about one order of magnitude faster than mine.

788. Dominating Numbers

Difficulty rating: 10 %.
Solution: 471745499. *Solved: Sat, 5 Mar 2022, 17:34.*
Math knowledge used: combinatorics.
Programming techniques used: none.

Outrageously simple problem. We can start by defining $f(n,i,d,j)$ as the amount of numbers with exactly n digits such that the leading one is i, there are exactly j non-leading digits equal to d (which might, or might not, be equal to i). From basic combinatorics we get

$$f(n,i,d,j) = \binom{n-1}{j} 9^{n-1-j}.$$

We now need to see which combinations of i d and j we are interested in. Clearly $j \leq n-1$. And d is the dominant digit if one of the following conditions holds:

- $d = i$ and $j \geq \left\lceil \frac{n-1}{2} \right\rceil$.
- $d \neq i$ and $j \geq \left\lceil \frac{n+1}{2} \right\rceil$.

Obviously there are 9 combinations of $\{d,i\}$ that verify the first condition, and 81 that verify the second. Therefore we have

$$D(N) = \sum_{n=1}^{N} \left(9 \sum_{j=\left\lceil \frac{n-1}{2} \right\rceil}^{n-1} \binom{n-1}{j} 9^{n-1-j} + 81 \sum_{j=\left\lceil \frac{n+1}{2} \right\rceil}^{n-1} \binom{n-1}{j} 9^{n-1-j} \right).$$

A little reordering yields the formula I actually used, which is:

$$D(N) = 9 \sum_{n=1}^{N} \left(\binom{n-1}{\left\lceil \frac{n-1}{2} \right\rceil} 9^{n-1-\left\lceil \frac{n-1}{2} \right\rceil} + 10 \sum_{j=\left\lceil \frac{n+1}{2} \right\rceil}^{n-1} \binom{n-1}{j} 9^{n-1-j} \right).$$

This is approximately $O(N^2)$ with a combinatorial number cache. You can be even more clever and combine powers of 9 using the hockeystick identity, but this was good enough for me.

I assume that this will get rated 5 %. Maybe 10 % at most.

791. Average and Variance

Difficulty rating: *unrated as of today, since it's a recent problem.*
Solution: 404890862. *Solved: Sun, 27 Mar 2022, 12:29.*
Math knowledge used: Faulhaber's formulas.
Programming techniques used: none.

I have to confess that I'm a bit baffled by this problem. It's not super-easy, but it's definitely not one of the hard ones, yet here we are, 17 hours and a half after being published and it still hasn't even reached 50 solvers. Just to be clear, we only need Faulhaber's formula to sum a sequence of squares (classic induction problem with a very well known result), and then there is an extremely basic statistics formula, and a second degree equation to solve. Aside from that, it's all symbol pushing. You do need to have a certain insight that makes the problem workable, but it's not a big one. I'm also surprised that no one in the problem mentions the same scheme I used, and run times are higher than mine. I actually expected to see some magical formula involving the average and variance and solving the problem instantly, but so far, the best method in the comments seems to be the one I used!

So let's go to the problem. We have four integer numbers, $1 \le A \le B \le C \le D \le L$ where $L = 10^8$ is the problem limit. Let μ be the mean. Now we use one of the stellar super duper obscure formulas we need for the problem! And that is... the alternate formula for the variance, $\sigma^2 = \frac{1}{N}\sum x_i^2 - \mu^2$. Which, as far as I remember, is still taught in high school. For this particular problem, we have $\sigma^2 = \frac{1}{4}(A^2 + B^2 + C^2 + D^2) - \mu^2$. The problem description says that we need $\sigma^2 = \frac{1}{2}\mu$, and lo and behold,

$$A^2 + B^2 + C^2 + D^2 = 4\mu^2 + 2\mu.$$

Of course we also have the equation of the average itself, which is

$$A + B + C + D = 4\mu.$$

Now, it so happens that the equations are more convenient if we express them in terms of twice the mean, $x = 2\mu$. The new equations are

$$\begin{aligned} A + B + C + D &= 2x, \\ A^2 + B^2 + C^2 + D^2 &= x\,(x + 1)\,. \end{aligned}$$

Still complicated, but now comes the second insight, which is the happy idea that actually got me through the problem. Instead of the numbers A, B, C

and D, let's use their deviations from the mean:

$$\begin{aligned}
\alpha = A - \mu &\Rightarrow A = \frac{x}{2} + \alpha, \\
\beta = B - \mu &\Rightarrow B = \frac{x}{2} + \beta, \\
\gamma = C - \mu &\Rightarrow C = \frac{x}{2} + \gamma, \\
\delta = D - \mu &\Rightarrow D = \frac{x}{2} + \delta.
\end{aligned}$$

The ordering is preserved: $\alpha \leq \beta \leq \gamma \leq \delta$. And the first equation adopts a predictably simple format:

$$\left(\frac{x}{2} + \alpha\right) + \left(\frac{x}{2} + \beta\right) + \left(\frac{x}{2} + \gamma\right) + \left(\frac{x}{2} + \delta\right) = 2x \Rightarrow \alpha + \beta + \gamma + \delta = 0.$$

But where this approximation really shines is in the second equation, which just needs some additional symbol pushing. We start with

$$\left(\frac{x}{2} + \alpha\right)^2 + \left(\frac{x}{2} + \beta\right)^2 + \left(\frac{x}{2} + \gamma\right)^2 + \left(\frac{x}{2} + \delta\right)^2 = x\left(x + 1\right),$$

and expanding the binomials we get

$$\frac{x^2}{4} + \alpha x + \alpha^2 + \frac{x^2}{4} + \beta x + \beta^2 + \frac{x^2}{4} + \gamma x + \gamma^2 + \frac{x^2}{4} + \delta x + \delta^2 = x^2 + x.$$

Grouping the coefficients from the left hand we see that the coefficient for x^2 is $4\frac{1}{4} = 1$, which can be cancelled with the x^2 from the right term, and the coefficient for x is $\alpha + \beta + \gamma + \delta$, which we know from the other equation to be equal to 0. The independent terms are just $\alpha^2 + \beta^2 + \gamma^2 + \delta^2$. So we end with

$$\alpha^2 + \beta^2 + \gamma^2 + \delta^2 = x,$$

which is elegant and will be very useful.

Now it's a good time to look at the values of these variables and try to look at their format. While the equation $A + B + C + D = 2x$ suggests the possibility of having x be a half integer, this can't happen because in that case we would have a non-integer value for $A^2 + B^2 + C^2 + D^2 = x^2 + x$. So x is always an integer and the sum of all the numbers is always even. Still, the expression of the new variables in terms of the old ones still allow the possibility that they are half integers, and indeed they can be. We can also determine a very interesting piece of information from the last equation:

- Let's assume that at least one of the variables is a half integer. This means that its square is of the form $n + \frac{1}{4}$ where n is some integer. However, the sum of all the four must be an integer. Therefore, in this case, all four variables must be half integers so that the four lingering fourths sum up to one. This happens when the average μ is a half integer, that is, when $A + B + C + D$ is even but not a multiple of 4.

- Obviously we must also allow the case where the variables are integers. This happens when μ is an integer, i.e. when the sum of the original numbers is a multiple of four.

- There aren't more cases! Either all the variables are half integers, or none is.

At this point I define new variables in order to work with integers as much as possible:

$$\alpha' = 2\alpha, \quad \beta' = 2\beta, \quad \gamma' = 2\gamma, \quad \delta' = 2\delta.$$

The homogeneous equation remains the same, and the nonhomogeneous one will look like this:

$$\alpha'^2 + \beta'^2 + \gamma'^2 + \delta'^2 = 4x.$$

I realise that it gets confusing, but it does make the remaining operations clearer. Plus it's much better for a computer to iterate on. Because in fact we can now start devising a workable scheme. We know from the first equation that

$$\delta' = -\left(\alpha' + \beta' + \gamma'\right),$$

so we can iterate over the three remaining variables. This is how I solved the problem the first time, in fact. It takes a full half an hour, more if modding is present. So not a very good idea. We can do better, but we need to be very careful about the limits of these variables. Let's say that we iterate from α', then from β', and then from γ' only if we need it. The key to the remaining parts of the problem is in calculating the bounds of these iterations.

We start from the bounds for α', which are the more general ones since they won't be directly related to other variables. We can start from $A + B + C + D = 2x$, and notice that $D \leq L$ means that the most disadvantageous case comes when all the variables have the same value. Of course this doesn't happen because in that case the variance would be 0, but it's just an upper bound. We conclude that $\frac{x}{2} \leq L$, or better, $x \leq 2L$. Now, let's go back to the later stages of the equation. Since the sum of the four variables equals 0, given some fixed α' we find the most disadvantageous case when all the

other three variables have the same value, $\beta' = \gamma' = \delta' = -\frac{\alpha'}{3}$. In this case, the sum of the squares is $\frac{4\alpha'^2}{3}$. Yes, that very sum of squares that equals $4x$. Therefore we can be sure that

$$\frac{4}{3}\alpha'^2 \leq 4x \leq 8L \Rightarrow |\alpha'| \leq \sqrt{6L}.$$

Of course, α' will always be negative, since otherwise the ordering would cause the sum of the four variables to be nonzero (a similar argument proves that δ' must always be positive. β' and γ' adopt six different combinations of signs, including being zero. Even both at the same time. We don't need that fact in my solution). So we will iterate in descending order for α, starting at -1! I mean, we could do increasing order as well, but in my opinion this feels more natural.

Ok, now let's see what happens to the next variable, β'. By definition, $\beta' \geq \alpha'$, so the lowest value for β' would be, indeed, the current value of α'. As for the upper bound, we can see that again the most disadvantageous condition (i.e. the biggest β' that we could potentially have) happens when $\beta' = \gamma' = \delta'$. So, the homogeneous equation tells us that $\beta' \leq -\frac{\alpha'}{3}$. Remember that α' is negative, therefore this upper bound is positive.

Now let's look at γ'. We have an obvious lower bound, $\gamma' \geq \beta'$, and following the same reasoning as with β', we have $\gamma' \leq -\frac{\alpha' + \beta'}{2}$. By the way, since the intermediate variables were all half integers or all integers, the new ones with the apostrophe are either all odd or all even, meaning that this bound is always an integer. Not that it changes much. Also, it's always positive since $|\alpha'| > |\beta'|$. But things are going to be interesting from now on, because we have an additional set of bounds for γ' that comes from the equations.

Let's say that the sum of our fixed variables is $S = \alpha' + \beta'$. The sum of their squares is also some constant $K = \alpha'^2 + \beta'^2$. Clearly we have

$$S + \gamma' + \delta' = 0,$$

which we will use as a substitution. The equation we are going to use now, however, is not the other second equation (at least not directly), but rather, the initial problem condition $D \leq L$. We have $D = \frac{x + \delta'}{2}$, therefore $x + \delta' \leq$

$2L$. And now we will use that second equation, because we can replace x:

$$\frac{K^2+\gamma'^2+\delta'^2}{4}+\delta' \leq 2L \Rightarrow K+\gamma'^2+\delta'^2+4\delta' \leq 8L.$$

This is very interesting, but it still has two variables. No problem, since we have $\delta' = -(S+\gamma')$. We therefore have

$$K+\gamma'^2+(S+\gamma')^2-4(S+\gamma') \leq 8L,$$

which we can rewrite as a quadratic equation in γ':

$$2\gamma'^2+(2S-4)\gamma'+\left(S^2-4S+K-8L\right) \leq 0.$$

The signs of the inequation and of the first coefficient mean that:

- If the associated equation (i.e. having $=$ instead of $\leq$) doesn't have solutions, then the inequation doesn't have any solutions either.
- If the associated equation has two solutions, γ_1 and γ_2, then the valid interval for γ' is $[\gamma_1, \gamma_2]$.

Now of course this is a relatively simple quadratic equations, and it's not difficult to solve. The determinant is

$$\Delta = 16L+4+4S-S^2-2K,$$

so if $\Delta < 0$ we don't have any solution, and if $\Delta \leq 0$, we have two solutions,

$$\gamma_{1,2} = 1-\frac{S}{2} \pm \frac{1}{2}\sqrt{\Delta}.$$

We don't care about the relationship between this set of bounds for γ' and the other, simpler one we had. We only care that we have two sets of bounds, whose intersection defines a (possibly empty) final valid interval for γ', whose minimum is the maximum of the two minima, and whose maximum is the minimum of the two maxima. Sounds confusing when put in text, but in code it's stupidly simple. So we have this final set of bounds, $\gamma' \in [\gamma_{\text{mín}}, \gamma_{\text{máx}}]$, which we will interpret as a set of values γ_0, γ_0+2, γ_0+4, …, $\gamma_0+2(N-1)$ (remember that γ_0 must have the same parity as α' and β'), where γ_0 is the lowest integer above $\gamma_{\text{mín}}$ with the correct parity, and N is the amount of terms, $N = 1+\left\lfloor \frac{1}{2}(\gamma_{\text{máx}}-\gamma_0) \right\rfloor$.

Ok, what do we do now? Uh, just calculate the result, i.e., the sum of all $A+B+C+D$. This sum equals $\frac{x}{2}$, therefore we can use this sum all over

the valid values of γ':

$$\sum_{\gamma'} \frac{K + \gamma'^2 + \delta'^2}{2} = \sum_{\gamma'} \left(\frac{K + S^2}{2} + S\gamma' + \gamma'^2 \right).$$

It's clear now why I defined γ_0 and N. We can iterate for integers $i \in [0, N)$, and for each one we have $\gamma' = \gamma_0 + 2i$ and $\delta' = -(S + \gamma_0 + 2i)$. The terms are easy to sum, and the only complication comes from Faulhaber's formula for the sum of squares. We will have a constant term,

$$N \frac{K + S^2}{2};$$

another that comes from the middle γ' term, which ends being

$$SN(\gamma_0 + N - 1);$$

and the final one, where Faulhaber's formula for the sum of squares is needed,

$$N\gamma_0^2 + 2N(N-1)\gamma_0 + \frac{2}{3}\left(2(N-1)^3 + 3(N-1)^2 + (N-1)\right).$$

Aaaaand that's it. The full algorithm goes like this:

- Calculate the lower bound for α', $\sqrt{6L}$.
- Iterate starting from $\alpha' = -1$, decreasing up until the bound. Now, for each α', iterate β' from α' to the upper bound, $-\frac{\alpha'}{3}$. Use only values of β' with the correct parity.
- Given α' and β', calculate S and K.
- Solve the quadratic equation. If it doesn't have any solutions, go to the next β'.
- Calculate the bounds for γ'. With them, calculate γ_0 and N.
- Now that you have S, K, N and γ_0, calculate the sum of the three terms above and add it to the solution. Mod where needed.

Are we finished? No! There is a small detail. The proposed scheme is fast and exhaustive, but it does include three spurious solutions that we don't want,

because they feature terms equal to zero. These solutions are

$$\begin{array}{llllllllll}\alpha'=-1, & \beta=-1, & \gamma=1, & \delta=1 & \Rightarrow & A=0, & B=0, & C=1, & D=1;\\ \alpha'=-2, & \beta=0, & \gamma=1, & \delta=2 & \Rightarrow & A=0, & B=1, & C=1, & D=2;\\ \alpha'=-3, & \beta=1, & \gamma=1, & \delta=1 & \Rightarrow & A=0, & B=2, & C=2, & D=2;\end{array}$$

Instead of bothering with special conditions for these, it's just better to subtract 12 from the result. Even better, initialise the result as -12 and just keep adding the values normally. And now we have finished for real.

There is a small room for improvement, since we can predict which values of β' result in a negative Δ and filter them. The result is a quadratic equation, but unlike the one we use for γ', here the result is an interval that we need to subtract to the "normal" one, potentially resulting in a disjoint pair of intervals, making the result a bit ugly. There is not that much run time to be gained by this, either.

792. Too Many Twos

Difficulty rating: 85 %.
Solution: 2500500025183626. *Solved: Wed, 6 Apr 2022, 08:11.*
Math knowledge used: combinatorics, binary representations.
Programming techniques used: none.

I was about to add "Lucas' theorem" and "Kummer's theorem" to the needed math knowledge list, but I didn't actually use them in the end. They are kind of floating around this problem, never actually touching it. I had them in mind constantly, waiting for an opportunity to use them, but none arose. The problem is in fact almost 100 % managed with combinatorics, and by this I mean relatively basic combinatorial numbers identities. Plus a lot of pen and paper and many not so obvious derivations. I did use OEIS though.

I started with a brute force, where I noticed that $u(n) \geq n+2$, and in fact $u(n) = n+2$ for $n = 2^k - 1$. This suggested defining the functions

$$f(n) = \frac{3S(n)+4}{2^{n+2}}, \quad g(n) = \nu_2(f(n)),$$

so that $u(n) = n + 2 + g(n)$. The challenge is to calculate $f(n)$ in a way that allows calculating $g(n)$ easily as well. So the first thing is to enumerate the first values of $f(n)$:

$$-1, 4, -13, 46, -166, 610, -2269, \ldots$$

OEIS says: the absolute value of this sequence corresponds to A026641, *Number of nodes of even outdegree (including leaves) in all ordered trees with n edges*. Ok? We don't care about the sign, so this is good. But how can I calculate it? It corresponds to the left border of a triangle sequence, A158815, which is itself the matrix product of two other sequences (seriously, it doesn't have any meaningful definition in OEIS, it's just defined as that product). Fortunately, this A158815 sequence has an explicit formula:

$$T(n,k) = \sum_{j=0}^{n} (-1)^{j+k} \binom{j}{k} \binom{2n-j}{n}.$$

What OEIS tells us is that $|f(n)| = T(n,0)$. Very conveniently, we have $(-1)^{j+0} = (-1)^j$ and $\binom{j}{0} = 1$. As such we have

$$|f(n)| = \sum_{j=0}^{n} (-1)^{j} \binom{2n-j}{n}.$$

At this point the derivation starts getting interesting. I start by defining an auxiliary function,

$$h(a,n) = \sum_{j=0}^{2n-a} (-1)^{j} \binom{2n-j}{a}.$$

We are going to work on this function in the interval $a \in [n, 2n]$. Clearly the function we are interested in is $h(n,n) = |f(n)|$. The interesting part is that we are going to find a recursive expression in terms of combinatorics function. To see more clearly how does this function look, we can unroll it, and we get

$$h(a,n) = \binom{2n}{a} - \binom{2n-1}{a} + \binom{2n-2}{a} - \ldots \pm \binom{a}{a}.$$

Now, it may make sense to separate positive and negative values, but what we are going to do is separating this function into these two parts: $h(a,n) = h_+(a,n) - 2h_-(a,n)$, with

$$\begin{aligned} h_+(a,n) &= \binom{2n}{a} + \binom{2n-1}{a} + \binom{2n-2}{a} + \ldots + \binom{a}{a}, \\ h_-(a,n) &= \binom{2n-1}{a} + \binom{2n-3}{a} + \binom{2n-5}{a} + \ldots + \binom{x}{a}. \end{aligned}$$

Here, $x = a+1$ or a depending on the parity of a. We don't care *much* about this, because there will be two separate cases but both result in the same end formula. For now, we can notice that the first function corresponds to the so-called hockey stick identity, which means that

$$h_+(a,n) = \binom{2n+1}{a+1}.$$

This is very convenient and it's the main reason why we are using this separation instead of the obvious one, with positive numbers in one side and negative ones in the other. For h_- the story is more complicated. We start from the simplest of all combinatorial identities, $\binom{m}{n} + \binom{m}{n+1} = \binom{m+1}{n+1}$, which we rewrite as $\binom{m}{n} = \binom{m+1}{n+1} - \binom{m}{n+1}$. If we apply it to every single term in $h_-(a,n)$, we get

$$h_-(a,n) = \binom{2n}{a+1} - \binom{2n-1}{a+1} + \binom{2n-2}{a+1} - \binom{2n-3}{a+1} + \ldots + \binom{x+1}{a+1} - \binom{x}{a+1}.$$

We can refine it further by noticing that there are two separate cases:

- If $x = a+1$, we leave the formula as it is, with $-\binom{a+1}{a+1}$ being the last term.
- But if $x = a$, we notice that the last term is $-\binom{a}{a+1} = 0$ and remove it, and so, the last term is $+\binom{a+1}{a+1}$.

In other words:

$$h_-(a,n) = \binom{2n}{a+1} - \binom{2n-1}{a+1} + \binom{2n-2}{a+1} - \ldots \pm \binom{a+1}{a+1} = h(a+1,n).$$

And so we arrive to the following recursive formula:

$$h(a,n) = \binom{2n+1}{a+1} - 2h(a+1,n).$$

It's easy to see that this formula gets us to

$$h(a,n) = \binom{2n+1}{a+1} - 2\binom{2n+1}{a+2} + 4\binom{2n+1}{a+3} - \ldots,$$

and so we arrive at the formula we were looking for:

$$|f'(n)| = \sum_{j=0}^{\infty} (-2)^j \binom{2n+1}{n+1+j}.$$

For $j \geq n$ and onwards the binomial term is going to be 0, so we can just say that

$$|f'(n)| = \sum_{j=0}^{n-1} (-2)^j \binom{2n+1}{n+1+j},$$

and this is the formula we are actually going to use. We won't need all the terms; far from it. We are only interested in this formula as long as the lower end bits are 0, and it so happens that this formula above is particularly well suited to find the first nonzero bit, since once we find it we can stop looking at the higher terms. And so we arrive at an algorithm that calculates $u(n)$ for some arbitrary n. First we need to define a maximum amount of bits we will look for, say B. Having defined this, this is the algorithm:

- Initialise a variable $r \leftarrow n+2$.
- Initialise also a variable $v \leftarrow 0$.
- Now, iterate starting from $a = n+1$:
  - Calculate $x = \binom{2n+1}{a}$ modulo 2^B, and update $v \leftarrow v + x$.
  - If v is odd, break the loop. The result is $u(n) = r$.
  - But if v is even, update $r \leftarrow r+1$, $a \leftarrow a+1$, $v = -\frac{v}{2}$ and continue looping.

Not so complicated, right? The real difficult part is in calculating the combinatorial numbers. It's easy to see that

$$\binom{2n+1}{n+a+1} = \frac{n-a+1}{n+a+1}\binom{2n+1}{n+a},$$

which we can use to update the successive binomial terms for a fixed $u(n)$. We are doing calculations modulo 2^B, so we can use 2-adic numbers, which in practice can be achieved with a normal number with B bits and an additional term for the power of 2. The real problem comes when calculating the initial $\binom{2n+1}{n+1}$ modulo 2^B. I'm not very happy to say that I bruteforced it, using the formula

$$\binom{2n+1}{n+1} = \frac{2(2n+1)}{n+1}\binom{2n-1}{n},$$

but being smart about modular inverses so that I only called the Euclid algorithm once per value of $u(n)$, i.e., only 10^4 times instead of 10^{12}. So I got the result in half an hour plus-minus 20 seconds, which is acceptable but suboptimal.

Another problem I ran into is that I used too low a value for B, namely 31. You need at least 32, which is a bit dangerous when doing arithmetic with 64-bit numbers, although it could have worked. So initially I got 2500500025183625 instead of the correct result, 2500500025183626. I saw the results and noted that I only got the theoretical maximum, $u\left(n^3\right) = 33$, for $n = 4373$. I suspected that this value wouldn't be too far from the actual result, so I tried increasing it by one, and there, it worked.

I'm now looking at ways to calculate $\binom{2n+1}{n+1}$ in a reasonable time. The comments can help me, but I'll need to read one or two papers. I had some, but I got unconvincing formulas. But maybe I can still do this in less than one minute?

Here's an update, ten weeks later. The problem is rated 85 %, making it the hardest problem I've solved in the same week it was posted. Currently only 67 people have (publicly) solved it.

799. Pentagonal Puzzle

Difficulty rating: *unrated as of today, since it's a recent problem.*

Solution: 1096910149053902. *Solved: Mon, 23 May 2022, 06:21.*

Math knowledge used: Hermite-Serret algorithm, sum of two squares theorem, "first prime" Erathostenes sieve, binary quadratic forms.

Programming techniques used: memoisation.

Interesting problem. At first it looks hard, then it looks doable, then it looks hard again, and you finally get the hang of it. The base idea seems to be simple: find cases of pentagonal numbers such that $P_a+P_b = P_c$, and then find a particular c such that P_c has at least 100 such decompositions. The problem is that finding pairs of pentagonal numbers whose sum is also pentagonal is not nearly as easy to do as when you're dealing with squares, so the problem has a strong brute force component. Iterating blindly is $O\left(n^2 \log n\right)$ for n equal to the index of the pentagonal number we're looking for, and that won't work when this number happens to be around $2 \cdot 10^7$ (I had a pretty good guess about this on my first attempt, which used 10^7 as a limit, although I didn't let it finish because it was horribly slow).

So I eschewed brute force and I started a serious attempt: pentagonal numbers are ultimately quadratic forms, so we have a second degree dipohantine equation with three variables. If we fix one of them, we get a second degree diophantine equation with two variables, that is, a perfectly Alpertronable formula. In particular, $P_a + P_b = P_c$ can be transformed into

$$\frac{3a^2 - a}{2} + \frac{3b^2 - b}{2} = \frac{3c^2 - c}{2} \Rightarrow 3a^2 - a + 3b^2 - b = 3c^2 - c = 2P_c.$$

Now of course I could iterate blindly using my implementation of Alpertron, but that would still take many hours, because I wouldn't be able to take advantage of the common structure of the equation. So I analysed the behaviour of Alpertron (I actually used the original version, which spells things more clearly than mine), which told me two things I knew and another one which I only half-knew, by which I mean that I knew that there would be a formula, but I hadn't worked it out. The interesting bits of information are these:

- The equation can be reduced to one of the form $3\alpha^2 + 3\beta^2 = R$ for some unknown right side R depending exclusively on c. This I knew.
- The equation's determinant is -36 and therefore it's an elliptical equation with a finite amount of solutions. This I also knew.
- The relationships between the first equation's variable and the second's are as follows: $-36a = \alpha - 6$; $-36b = \beta - 6$; $R = 216 + 2592P_c$. This was what I was looking for.

Of course the first thing we are going to do is divide by 3 in order to get a simpler equation:

$$\alpha^2 + \beta^2 = 72 + 864P_c.$$

And now the problem is, again, starting to look easy.

Thanks to the sum of two squares theorem, I know very well what I need to do with that $72 + 864P_c$: decompose it into primes of the form $4k + 1$ and then recompose their basic component solutions. I already did it for problem 311 and several others. So let's start with a better expression for this value:

$$72 + 864P_c = 72 + 432\left(3c^2 - c\right) = 72\left(18c^2 - 6c + 1\right).$$

Having a separate factor of $72 = 2^3 \cdot 3^2$ is very nice because it will not affect the solution. Meaning that for any way of expressing any n as the sum of two squares, there is exactly one way to express $72n$, and vice versa. Bijection, bitches. So we have this basic scheme:

- Decompose $18c^2 - 6c + 1$ as the sum of two squares (is it always possible?), $\alpha'^2 + \beta'^2$. Ensure that $0 < \alpha' \leq \beta'$ for uniqueness.
- Now, the solutions for our equation in α and β will be these: $\alpha = 6\left(\beta' - \alpha'\right)$; $\beta = 6\left(\beta' + \alpha'\right)$. Signs can be changed if needed.
- Ensure that α and β are such that $a = \dfrac{\alpha - 6}{-36}$ and $b = \dfrac{\beta - 6}{-36}$ are positive integer numbers.

For best results, the following analysis goes from the last point to the first one, instead of the other way around. First of all, it's nice that we can change the signs of α and β, because blatantly we are going to need them to be negative numbers so that a and b are positive. Ok? Still, it's more comfortable to work with their absolute values, so that they stay positive. This is good because the basic code to find solutions I've used (in this problem, but also in several other ones... and there's no need to rewrite) has a base set of positive solutions. So we have

$$a = \frac{-|\alpha| - 6}{-36} = \frac{|\alpha| + 6}{36},$$

so I can eschew the signs and simplify by saying that $\alpha+6$ must be a multiple of 36. Similarly, $\beta + 6 \equiv 0 \pmod{36}$.

Now, if we use the formulas for α and β in terms of α' and β', we can see that

$$\begin{aligned} 6(\beta' - \alpha') + 6 \equiv 0 \pmod{36} &\Rightarrow \beta' - \alpha' \equiv 5 \pmod{6}; \\ 6(\beta' + \alpha') + 6 \equiv 0 \pmod{36} &\Rightarrow \beta' + \alpha' \equiv 5 \pmod{6}. \end{aligned}$$

And with this trick we remove the 72 term, while also simplifying the congruence we need to satisfy. We can divide this into two separate cases of congruences of α' and β' (either $\alpha' \equiv 3$ and $\beta \equiv 2$, or $\alpha' \equiv 0$ and $\beta \equiv 5$, all modulo 6), but that is not super helpful. I still used the basic congruences.

Now back to the first step, which is going to be the most problematic by far: decomposing $18c^2-6c+1$ into prime factors and using that decomposition to decompose it again, but now as a sum of two squares. First issue: is it always possible? Well, yes! At first I was puzzled about this, but then I realised that $18c^2 - 6c + 1 = (3c-1)^2 + (3c)^2$, therefore it can always be expressed as the sum of two squares in at least one way, which means that every possibly present prime of the form $4k + 3$ will appear with an even power, and will be a divisor of both $3c - 1$ and $3c$. Now, since no prime number can divide two consecutive numbers, such a prime can't exist. Therefore $18c^2-6c+1$ must always be the product of primes of the form $4k+1$ (the reasoning above doesn't preclude 2 from being a factor, but $18c^2-6c+1$ is blatantly odd for any integer c, so it doesn't happen either). This is good, but we still have a big problem: we expect c to get big, so we are not going to be able to use the nice and fast "first prime" decomposition. Oops. This could take hours to do correctly.

So let's put a bit of additional effort. We are looking for cases where $18c^2 - 6c + 1$ has *at least* 100 decompositions as the sum of two squares

(note that we can't just count, we really need to check them one by one and some will be discarded). But in order for that to happen, this number must have a big amount of divisors (roughly, the more divisors it has, the more decompositions, although the formula is not super clear-cut. It depends on the exponents of the primes involved). So we can cull very aggressively in order to sieve as many numbers as possible. In particular, I did this:

- Force $18c^2 - 6c + 1$ to be a multiple of the first four $4k + 1$ primes: $5 \cdot 13 \cdot 17 \cdot 29 = 32045$. I got lucky here: if I had added another prime (the next one is my old friend 37), I wouldn't have found the solution. This can be kind of generated directly, instead of filtering (for example, this value is a multiple of 5 if and only if $c \equiv 3$ or 4, modulo 5; there are similar recurrences for the other primes. But I didn't bother because the code was so fast anyway).

- Also don't bother with primes of the form $4k + 3$, which we know that will never divide our number.

- Ignore the cases where there is a prime greater than a million.

The first culling made the algorithm go from several hours to 5 minutes. The second one, as expected, divided it by almost exactly half. The third one was the killer move that made it go below a second. Hooray!

And that's really it. I used a small system of caches to avoid calling the Hermite-Serret algorithm continuously, although I did recalculate the combinations. Most of the code related to finding the actual decompositions as sum of squares is recycled from older problems. The full sequence of calculations, at a high level, is this:

- Get a list of the primes below one million. Yes, this is enough, and in fact I could have been much more aggressive. This can be used as a "first prime" factor generation, but also as a source for a list of primes of the form $4k + 1$.

- Now, iterate for $c = 1$ to, well, infinity. Or until we find the solution, whichever happens first (guess who wins?). Now, for every c:

  - Calculate $K = 18c^2 - 6c + 1$.
  - If K is not a multiple of 32045, finish immediately and go to the next c.
  - Find the decomposition of K as a product of primes, which we know will be all of the form $4k + 1$.

- If there is a prime greater than a million, pass and go to the next c.
- Let e_1, e_2, e_3, …, e_n be the exponents of the prime number. An estimate for the number of ways to express the number as sum of two squares is $2^{n-1}\prod_i \left(1+\left\lfloor\frac{e_i}{2}\right\rfloor\right)$. If this number is smaller or equal to 100, move on to the next c. This heuristic is safe, in the sense that it's always greater or equal to the actual number.
- If we have still passed all these culls, it's time to actually find these decompositions, with the Hermite-Serret algorithm and all the associated combination methods.
- Given a decomposition $K = \alpha'^2+\beta'^2$, count it if $\beta'-\alpha' \equiv \beta'+\alpha' \equiv 5 \pmod 6$.
- If the total count of valid solutions is greater than 100, we have found our solution.

The end solution is found when $c = 27042068$, for a total of 108 solutions. For this number we have

$$K = 13162921788646825 = 5^2 \cdot 13^2 \cdot 17 \cdot 29^2 \cdot 41 \cdot 97 \cdot 157 \cdot 349.$$

So yes, I could have used a much more aggressive, with 1000 instead of 1000000. But meh, it's very fast anyway, no reason to complain. These 108 solutions are sometimes close, like

$$P_{19078421} + P_{19164741} = P_{27042068},$$

and sometimes they are far away, like

$$P_{12738} + P_{27042065} = P_{27042068}.$$

800. Hybrid Integers

Difficulty rating: *unrated as of today, since it's a recent problem.*
Solution: 1412403576. *Solved: Sun, 29 May 2022, 06:17.*
Math knowledge used: Erathostenes sieve.
Programming techniques used: binary search.

I suspected that the problem would be kind of easy for a "multiple of 100" problem, like 700 was, but this was far, far easier than I imagined. It reached exactly 49 solvers within an hour of publication time, not counting "hidden" ones (this means definitely at least 50 since I'm one of the "hidden"

ones). I took 17 minutes because I'm not a fast enough programmer, and I got 24th place.

So, here it is. We are looking for numbers of the form $p^q q^p$ such that p and q are two different primes and the number is below 800800^{800800}. Now, enumerating all of them would be a bit too slow since the total count exceeds 10^{10}, so we can be a bit clever and count in a batch. We use the fact that, given any p, the function $f_p(q) = p^q q^p$ is monotonally increasing for integer values of q. So we can do a binary search. And this is all we need after enumerating all the primes, really. This is the full algorithm:

- Let $L = \log(800800^{800800}) = 800800 \log 800800$.
- Get a list of all the primes below certain limit. I used 10^8.
- Calculate also the logarithms of all the primes.
- Initialise the result as 0.
- Now, iterate for every *index*, i. Let p_i be the i-th prime and l_i its logarithm.
- For each i, use a binary search to get the biggest index j such that $l_i p_j + l_j p_i \leq L$.
- if $j > i$, increase the result in $j - i$. Otherwise, stop iterating.
- The value of the result at the end of the iteration is the solution of the problem.

It can be seen that the biggest necessary prime is 15704473. The next prime after that one, $p = 15704509$, already verifies $p^2 2^p > 800800^{800800}$. I used 10^8 because it was enough and I didn't bother fine-tuning the limit. I guess that the run time can be reduced by using a stricter prime limit like $1{,}6 \cdot 10^7$ (from half a second to 88 milliseconds), but really, the code is just fast enough. Other people keep the last index at each iteration and go down in decrements of 1, which makes the code slightly longer but make the algorithm $O(n)$ instead of $O(n \log n)$ (here, n is the length of the prime list). Although it's so fast that the difference is really negligible.

801. $x^y \equiv y^x$

Difficulty rating: *unrated as of today, since it's a recent problem.*
Solution: 638129754. *Solved: Tue, 7 Jun 2022, 04:40.*

Math knowledge used: Erathostenes sieve, Miller-Rabin algorithm, Jordan functions.

Programming techniques used: none.

This was a really fun problem to solve. The kind of problem that looks completely unsurmountable unless you know some very obscure piece of math, until you start studying the results and see the pattern. It's not like the pattern was super easy to spot, but OEIS was there to help me.

The first thing I did was running the brute force algorithm for some small values of p; the proposed base case limit, 10^2, was good enough, and the result was easy to reproduce. Now I started examining the results to see the pattern. Some observations:

- $f(p)$ grows more or less like p^3, but there isn't any clear formula.
- f seems to be mostly an increasing function, but this is not always the case: $f(79) > f(83)$. And $f(47)$ is only a tiny bit above $f(43)$. This got me stumped for a while, and I treated this as a confirmation that there wasn't a simple formula for $f(p)$ as a function of p.

Not very useful, but now let's define $g(p, x)$ as the amount of values $0 < y \leq p^2 - p$ such that $x^y \equiv y^x \pmod{p}$. And the pattern becomes clearer:

- $g(p, x)$ is always greater than 0 for any x in the proposed interval.
- In fact, for most of the values of x, $g(p, x) = p - 1$.
- Also, $g(p, x)$ is always a multiple of $p - 1$.
- In particular, $g\left(p, (p-1)^2\right) = (p-1)^2$. This is the only value of x for which we get this result, and this seems to be the upper limit.
- Otherwise there isn't any particular obvious pattern. I'm guessing that the pattern exists, I just didn't spot it. There is some semblance of regularity but I didn't really look into it.

So, the next obvious idea was to group the different values of g for a given p, trying to count them so that they could be easily summed. And this is where I found the pattern.

- The possible values of g are of the form $(p-1) \cdot d$, where d is any divisor of $p - 1$. In particular, the minimum value is $p - 1$ and the maximum value is $(p-1)^2$.
- The function is more or less decreasing, i.e.: if $g_2 > g_1$, then the count of values equal to g_2 tends to be smaller than that of g_1. This is not always true, though.

- Also, all the counts seem to have very small prime decompositions, save for the minimum value, where we find rarities such as $g(11, 10) = 82$ or $g(23, 22) = 382$. This suggests that there exists a reasonably compact calculation for all the divisors, and then the "leftovers" go to the base value. By the way, there are exactly $p^2 - p$ cases, since that's the valid space of values for x, so this amount of "leftovers" can be calculated easily.

- The next obvious thing to try is checking whether the is a relationship between the value of $g = (p-1)\,d$ and the count. There seemed to be a regularity, in that there was only a single value for $(p-1)^2$, as mentioned above, and the second least frequent value was always that of $\frac{(p-1)^2}{2}$, appearing exactly 3 times. The third most frequent value already presented some variation: sometimes it's 8, other times it's 12, but sometimes it's 24 or a bigger number. In particular, the case for $p = 83$ was particularly noticeable, with the third most frequent case being $164 = 82 \cdot 2$, appearing 1680 times, which I immediately recognised as $41^2 - 1$. And 82 is a multiple of 41 ...

- So I thought that the pattern could be related to a different divisor: after all, all these values of g correspond to a value of the form $\frac{(p-1)^2}{d}$ for some divisor d. Jackpot! The value is consistent: except for the final "leftover" case, for every other value there is a fixed relationship between this d and the count of values. This count of values was helpfully recognised by OEIS as the Jordan function $J_2(n)$, which is a multiplicative function where $J_2(p^e) = (p^2-1) \cdot (p^2)^{e-1}$.

This was all I needed. The pattern is not obvious but it's there and it's reasonably simple; and a bit of work can uncover it. As expected, the amount of solvers was small at first, but it quickly rose. I don't think this problem is going to have more than 50 % of difficulty or so (not that I expect to be there and see it, since it won't be "finished" until after this summer). The full algorithm is this:

- Iterate in the given range, using Miller-Rabin to find the primes. There are less than 30000.

- For every prime p in the given interval, decompose $p-1$ in prime factors. This doesn't take *that* much even if I used a brute force iteration using all the primes (I wonder whether Pollard's rho would have been faster?), but it's nevertheless what takes most of the time, about 12 out of the 18 seconds of run time.

- Now, with the decomposition in hand, iterate over the divisors to calculate these temporary results:

$$\begin{aligned} r_p &= \sum_{\substack{d|p \\ d\neq 1}} J_2(d) \cdot \frac{p-1}{d}, \\ s_p &= \sum_{\substack{d|p \\ d\neq 1}} J_2(d). \end{aligned}$$

  Here, r_p is a cumulative sum of results (save for a factor of $p-1$ that we will include at the end), and s_p is the count of values we have found, so that $p^2 - p - s_p$ is the amount of remaining values for which $g(p, x) = p - 1$.

- The value of $f(p)$ is exactly $(p-1) \cdot (r_p + p^2 - p - s_p)$.

- And the solution of the problem is the sum of all the $f(p)$ for the given primes.

The full result is a 53 digit number, which is reasonable since it's a sum of around 30000 values that are somewhat above $(10^{16})^3 = 10^{48}$. This is the full number:

$$44728530971547698409637317057763477817334488235803868.$$

I knew that the size would be reasonable, so I didn't bother with modding until the end. The run time is around 18 seconds and the BigInteger-heavy segment takes less than 2 seconds (although Miller-Rabin also uses them, probably unavoidably so), so it's not like there is a lot to gain if I stay with longs and use mods aggresively to avoid overflows.

In the problem thread there are some moderately convincing mathematical arguments explaining why this pattern with the Jordan function appears, but my experience with number theory doesn't seem to be good enough to accurately follow and understand it completely.

802. Iterated Composition

Difficulty rating: *unrated as of today, since it's a recent problem.*
Solution: 973873727. *Solved: Sun, 12 Jun 2022, 13:14.*
Math knowledge used: Cardano-Vieta formulas, Erathostenes sieve.
Programming techniques used: none.

After a first look at this problem, the most likely reaction might be something like "are you fucking kidding me?", which is indeed a common reaction to certain kind of problems. Then you realise something in the problem description: *interestingly,* $P(n)$ *is always an integer*. Interestingly, indeed. So this is like 356, in the sense that we don't need to calculate a real solution to an equation; we only need to study its properties. And we can stay in the realm of integers, no need for this complicated and inaccurate floating point.

Looking again at the problem formulas and the description, one can notice a basic property, which is obvious if you think a bit about it: if (x, y) is a pair with cycle n, then it will be a solution of the equation $f^{(m)}(x, y) = (x, y)$ for any m which is a multiple of n. So, for each n, we would be interested in counting only the "primitive" solutions for each n. If we had a method to calculate all the solutions for a given n, we could just subtract the non-primitive values (i.e. the results for every proper divisor of n, including 1) and get the value we want. Great, but how do we calculate such values? And this is a question whose complete answer I don't know, but I had a hunch.

So, for $n = 1$ we have an obvious expression for $f(x, y) = (x, y)$, but for $n = 2$ the answer might not be so obvious. If we unroll the expression of $f^{(2)}(x, y)$, the first equation looks something like this:

$$x^4 - 2x^3 + g_2(x, y) = x,$$

where $g_2(x, y)$ is a complicated expression where the highest degree of x is 2. Similarly, for $n = 3$ we get an equation like

$$x^8 - 4x^7 + g_3(x, y) = x,$$

and for $n = 4$ we have

$$x^{16} - 8x^{15} + g_4(x, y) = x,$$

This was a bit of a shot in the dark, but I assumed that I could use Cardano-Vieta and assume that the solutions for any given $n > 1$ would have a sum exactly equal to 2^{n-1}. This is not so obvious because complex solutions exist, but I only did some experiments with Matlab, which has a lousy solver based on standard floating point numbers. Anyway, worth a try. Let's assume that this is in fact true. If this was the case, an algorithm like this would work:

- Store a 2 as the solution for $n = 1$, which is a special case.
- Also initialise the total result as 2.

- We will also need a "power of 2" counter, p, which can be initialised as 1.
- Now, iterate over every integer n from 2 to the limit, $L = 10^7$ (you might as well skip 2, since there are no solutions for that case, but it's not like the run time is going to soar if you leave it; initialise p as 2 instead of 1 if you do this).
    - Update the power of 2, by doing $p \leftarrow 2p \bmod M$.
    - Initialise the result for n as the current value of p.
    - Get the prime factors of n, and use them to get a list with all its divisors.
    - For every divisor d of n, except n itself, subtract the result for d from the result for n.
    - Store the result for n, and update the total result with it. Mod as needed, remembering that after all the subtractions we might get a negative value.
- That's it. If you have been updating the result as you iterate over n, the result at the end of the loop is the solution to the problem.

Lo and behold, it works! And with a run time of 8,3 seconds (there are faster ways to do it if you are clever with the Möbius and/or Mertens functions). So the assumption was correct! And, from what I can gather from the problem thread, it would be correct even if that scary π wasn't there in the second member of the function. It's just a red herring.

I got position 16, and maybe I could have got a better one if I had started with the problem immediately, but oh well, too late.