r/adventofcode • il y a 5 a
daggerdragon MODO

-🎄- 2019 Day 22 Solutions -🎄-

SOLUTION MEGATHREAD

--- Day 22: Slam Shuffle ---

Post your full code solution using u/topaz2078's `paste` or other external repo.

- Please do NOT post your full code (unless it is *very* short)
- If you do, use old.reddit's four-spaces formatting, NOT new.reddit's triple backticks formatting.
- Include the language(s) you're using.

Lire la suite

Publication archivée. Impossible de voter et de publier de nouveaux commentaires.

30 168 Partager

Trier par : Nouveaux (par défaut)

sswain • -5 a

Part 2 was the least favorite of the challenges. Without an understanding of the underlying math you are lost. Cobbled together an answer after reading through a lot of comments here but still don't really understand how it works.

1

e_blake • -5 a

didn't understand the problem, but because m4 lacks 64-bit math. Coding up a 64-bit division/remainder macro on top of 32-bit math was not my idea of fun; and even power-of-2 math is difficult (m4 prefers to represent numbers as power-of-10 strings; and although eval() can produce power-of-2 results, you're back to the 32-bit limitations with no convenient way to split up larger numbers into 32-bit chunks). So instead, I did something totally different (and which I haven't seen mentioned anywhere else in this thread): I learned about Montgomery representations, and coded my solution using base-10000 bigint multiplication, addition, and subtraction; and in the few places where I used 32-bit division, the dividend in each of those is a power of 10 and I could just as easily use substr() to do string-chopping to get the same effects. Thus, never once does my solution divide by 10007 or 119315717514047.

Although my C solution computed a reverse shuffle using modular inverse, my m4 solution instead does forward shuffles for size - 1 - position, using the exponentiation-by-squaring algorithm on function compositions rather than directly performing modular exponentiation. Converting a bigint to binary for driving the function composition was my last stumping point, until I remembered that dividing by 2 is the same as multiplying by 5 then dividing by 10, putting me back in the realm of not needing to implement bigint division. Runtime was about 1.1s, but I'm quite pleased that my code runs under 'm4 -G' (and no GNU extension esyscmd calls like what I did for 64-bit math in my IntCode solution). Just for grins, I traced 58,692 eval(), 2,686 substr(), and 406 redc() calls in part 1; then 6,223 add(), 312,572 eval(), 72,281 substr(), and 538 redc() calls in part 2 (where redc is my Montgomery reduction macro).

m4 -Dfile=day22.input day22.m4

1 ···

WikiTextBot • -5 a

Montgomery modular multiplication

In modular arithmetic computation, Montgomery modular multiplication, more commonly referred to as Montgomery multiplication, is a method for performing fast modular multiplication. It was introduced in 1985 by the American mathematician Peter L. Montgomery.Given two integers a and b and modulus N, the classical modular multiplication algorithm computes the double-width product ab, and then performs a division, subtracting multiples of N to cancel out the unwanted high bits until the remainder is once again less than N. Montgomery reduction instead adds multiples of N to cancel out the low bits until the result is a multiple of a convenient (i.e. power of two) constant R > N. Then the low bits are discarded, producing a result less than 2N. One conditional final subtraction reduces this to less than N. This procedure has a better computational complexity than standard division algorithms, since it avoids the quotient digit estimation and correction that they need.

The result is the desired product divided by R, which is less inconvenient than it might appear.

[^PM | Exclude ^me | Exclude from ^subreddit | FAQ / ^Information | ^Source] Downvote to remove | v0.28

1 ···

nibarius • -5 a

My Kotlin solution which is a little math and no inverse solution based on this comment: https://www.reddit.com/r/adventofcode/comments/ee56wh/2019_day_22_part_2_so_whats_the_purpose_of_this/fbr0vjb/

NeilNjae • -5 a

Still plugging away! Another Haskell solution, described on my blog with code. I can't claim much credit for this, as it's basically a reimplementation of u/mstksg 's description of their solution, but I learnt a lot from it.

2 •••

wace001 • -5 a

Java

Finally posting my solution here. It is refactored quite heavily as most of it was rewritten several times of part 2. I did finally solve part 2, but it took me quite a while.

Pretty happy with the solution.

1 •••

Aidiakapi • -5 a

Rust

This was a really fun day. Solved part 1 the straightforward way, by actually shuffling the deck.

Part 2 really got me doing some mathmatics, and I'm happy with the fully general solution I came up with, and runs basically instantly.

1 •••

phil_g • -5 a

My belated solution in Common Lisp.

I did part 1 by composing individual functions for each instruction, which worked well enough for its scale. I actually misread the problem and calculated everything backwards based on the final position, rather than forward from an initial position. Rather than redo everything, I just searched through the final positions until I found the one that gave the right starting point.

I had to mull part 2 over for a few days. It took reading others discussing the calculation as a single linear function to point me in the right direction. I did break out my `modpower` function I wrote for Project Euler to do exponentiation under a modulus.

1 •••

loociano • -5 a

My solution to part one in **Python 3**, still figuring out part two. Feedback more than welcome.

1 •••

mcpower_ • -5 a • Modifié il y a -5 a

it without requiring knowledge of number theory, but I couldn't think of it.

A key observation to make is that every possible deck can be encoded as a pair of (first number of the deck, or `offset` AND difference between two adjacent numbers, or `increment`). **ALL** numbers here are modulo (cards in deck), or `MOD`.

Then, getting the `n` th number in the sequence can be done by calcuating `offset + increment * n`.

Starting off with `(offset, increment) = (0, 1)`, we can process techniques like this:

- **deal into new stack**: "reverses the list". If we go left to right, the numbers *increase* by `increment` every time. If we reverse the list, we instead go from right to left - so numbers should *decrease* by `increment`! Therefore, negate `increment`. However, we also need to change the first number, taking the new second number as the first number - so we increment `offset` by the new `increment`. In code, this would be:

```
increment *= -1
offset += increment
```

- **cut** `n` **cards**: "shifts the list". We need to move the `n` th card to the front, and the `n` th card is gotten by `offset + increment * n`. Therefore, this is equivalent to incrementing `offset` by `increment * n`. In code, this would be:

```
offset += increment * n
```

- **deal with increment** `n`: The first card - or `offset` - doesn't change... but how does `increment` change? We already know the first number in the new list (it's `offset`), but what is the second number in the new list? If we have both of them, we can calculate `offset`.
The `0` th card in our old list goes to the `0` th card in our new list, `1` st card in old goes to the `n` th card in new list (mod `MOD`), `2` nd card in old goes to the `2*n` th card in new list, and so on. So, the `i` th card in our old list goes to the `i*n` th card in the new list. When is `i*n = 1`? If we "divide" both sides by `n`, we get `i = n^(-1)`... so we calculate the modular inverse of `n` mod `MOD`. As all `MOD` s we're using (10007 and 119315717514047) are prime, we can calculate this by doing `n^(MOD - 2)` as `n^(MOD - 1) = 1` due to Fermat's little theorem.
To do exponentiation fast, we can use exponentiation by squaring. Thankfully, Python has this built in - `a^b mod m` can be calculated in Python using `pow(a, b, m)`.
Okay, so we know that the second card in the new list is `n^(-1)` in our old list. Therefore, the difference between that and the first card in the old list (and the new list) is `offset + increment * n^(-1) - offset = increment * n^(-1)`. Therefore, we should multiply `increment` by `n^(-1)`.
In (Python) code, this would be:

```
increment *= inv(n)
```

where `inv(n) = pow(n, MOD-2, MOD)`.

Okay, so we know how to do one pass of the shuffle. How do we repeat it a huge number of times?

- `increment` is always multiplied by some constant number (i.e. not `increment` or `offset`).
- `offset` is always incremented by some constant multiple of `increment` *at that point in the process*.

With the first observation, we know that doing a shuffle pass always multiplies `increment` by some constant. However, what about `offset`? It's incremented by a multiple of `increment`... but that `increment` can change during the process! Thankfully, we can use our first observation and notice that:

- all `increment`s during the process are some constant multiple of `increment` before the process, so
- `offset` is always incremented by some constant multiple of `increment` *before the process*.

Let `(offset_diff, increment_mul)` be the values of `offset` and `increment` after one shuffle pass starting from `(0, 1)`. Then, *for any* `(offset, increment)`, applying a single shuffle pass is equivalent to:

```
offset += increment * offset_diff
increment *= increment_mul
```

That's not enough - we need to apply the shuffle pass a huge number of times. Using the above, how do we get the `n` th `(offset, increment)` starting at `(0, 1)` with `n=0`?

As `increment` only multiplies by `increment_mul` every time, we can calculate the `n` th `increment` by repeatedly multiplying it `n` times... also known as exponentiation. Therefore:

```
increment = pow(increment_mul, n, MOD)
```

What about `offset` though? It depends on `increment`, which changes on each shuffle pass. If we manually write out the formula for `offset` for a couple values of `n`:

```
n=0, offset = 0
n=1, offset = 0 + 1*offset_diff
n=2, offset = 0 + 1*offset_diff + increment_mul*offset_diff
n=3, offset = 0 + 1*offset_diff + increment_mul*offset_diff + (increment_mul**2)*off
```

we quickly see that

```
offset = offset_diff * (1 + increment_mul + increment_mul**2 + ... + increment_mul**
```

Hey, that thing in the parentheses looks familiar - it's a geometric series! Using the formula on the Wikipedia page (because I forgot it...), we can rewrite it as

With all of that, we can get the `increment` and `offset` after doing a huge number of shuffles, then get the `2020` th number. Whew!

After looking at the other comments, it seems like this question requires knowledge of modular inverses and exponentiation.

TBH I feel that this problem is unfair for most participants of Advent of Code, who are expected to have a background in intermediate programming (lists, dictionaries / hashmaps, for loops, functions). I wouldn't expect most AoC participants to have any deep experience in discrete mathematics like modular inverses / exponentiation - even if it is part of a typical undergraduate computer science course - as I'd assume that most programmers are self-taught and have never done a computer science course.

To me, Advent of Code is a series of programming puzzles that any intermediate programmer - with a bit of time - can work out by themselves. It feels like most people doing part 2 of this puzzle would need to look up the solution for it... while it arguably enhances the community aspect of AoC, it feels unfair for people doing AoC without external assistance.

On the other hand... there are many pathfinding puzzles in AoC which expect knowledge of BFS - which some (most?) programmers don't know about. Is AoC unfair to the people who don't know BFS? My gut says no. AFAIK BFS has never been explicitly mentioned in pathfinding puzzles - similarly, modular inverses etc. wasn't explicitly mentioned in today's problem. What happens to the people who encounter a pathfinding problem without knowledge of BFS? Probably the same as the people who encounter this problem without knowledge of modular inverses and exponentiation - either give up or look online for a solution.

I'm still not sure whether this problem is "unfair". My gut says yes, my brain says no.

25 ...

18 réponses supplémentaires

16 réponses supplémentaires

since that was the interesting part.

First, we notice that the problem is now asking us to go backwards. So let's first figure out which position a particular card comes from if we only carried out the shuffle process once.

```
D = 119315717514047  # deck size

def reverse_deal(i):
    return D-1-i

def reverse_cut(i, N):
    return (i+N+D) % D

def reverse_increment(i, N):
    return modinv(N, D) * i % D  # modinv is modular inverse
```

I grabbed modinv from here. Parse your input and call these in reverse order and we can now reverse the shuffle process once. Let f denote this reversing function.

Now, note how all three operations are linear. That means their composition f is also linear. Thus, there exists integer A and B such that f(i) = A*i+B (equality in modulo D).

To find such A and B, we just need two equations. In my case, I took X=2020 (my input), Y=f(2020), and Z=f(f(2020)). We have A*X+B = Y and A*Y+B = Z. Subtracting second from the first, we get A*(X-Y) = Y-Z and thus A = (Y-Z)/(X-Y). Once A is found, B is simply Y-A*X. In code form:

```
X = 2020
Y = f(X)
Z = f(Y)
A = (Y-Z) * modinv(X-Y+D, D) % D
B = (Y-A*X) % D
print(A, B)
```

`+D` is a hack to get around the fact that modinv I copied can't handle negative integers for some reason.

Finally we are ready to repeadly apply f many times to get the final answer. Notice the pattern when you apply f multiple times.

```
f(f(f(x))) = A*(A*(A*x+B)+B)+B
           = A^3*x + A^2*B + A*B + B
```

In general:

```
f^n(x) = A^n*x + A^(n-1)*B + A^(n-2)*B + ... + B
```

In code form:

```
n = 101741582076661
print((pow(A, n, D)*X + (pow(A, n, D)-1) * modinv(A-1, D) * B) % D)
```

which yields our answer.

20 ...

2 réponses supplémentaires

DFreiberg • -5 a • Modifié il y a -5 a

By far my biggest-ever Part 2 improvement; combined with solving days 18 and 20 earlier today, and part 1 of 21, and I'm almost caught up! My solution is the same as everybody else's, but Mathematica has spoiled me, since rather than doing a modulus, I can work directly with the *TRUE* base of the exponent:

```
75236682861560785441110002665869023282022157985463035196513882016168959503106612715
```

Overall, this is the happiest I've been with a problem in quite a while, even if it just reminds me of how far behind on Project Euler I've gotten...

[POEM]: Scrambled

To mix one hundred trillion cards
One-hundred-trillion-fold
Cannot be done by mortal hands
And shouldn't be, all told.

The cards make razors look like bricks;
An atom, side to side.
And even so, the deck itself,
Is fourteen km wide.

The kind of hands you'd need to have,
To pick out every third,
From cards that thin and decks that wide?
It's, plain to say, absurd!

And then, a hundred trillion times?
The time brings me to tears!
One second each per shuffle, say:
Three point one million years!

Card games are fun, but this attempt?
Old age will kill you dead.
You still have an arcade in here...
How 'bout Breakout instead?

15 ···

⊕ 4 réponses supplémentaires

gengkev • -5 a • Modifié il y a -5 a most interesting part was figuring out how to repeat the computation f(x) = ax + b a large number of times, which it seems like most people used the formula for geometric series to do.

I used 2x2 matrix multiplication instead — the main observation is that you can rewrite f(x) = ax + b as a matrix operation y = Ax via something like this: https://imgur.com/a/jyBkXMx

Then repeating this computation N number of times only requires exponentiating this matrix to the power of N, which can be done in logarithmic time with fast matrix exponentiation (with a modulus).

Interestingly: the closed form of {{a, b}, {0, 1}}^n is just {{a^n, (a^n-1)b/(a-1)}, {0, 1}} which is the same formula as the other solution! (computing the closed form can be done by diagonalization, or by just asking WolframAlpha)

This is similar to a problem from 2017 in USACO, COWBASIC (solution), which involves simulating a very simple "program" (which can only perform linear operations) for a very large number of steps! The solution for that also happens to discuss using the geometric sum formula vs. the matrix exponentiation approach, for the case of only one variable.

11 ···

5 réponses supplémentaires

SlimChanceOfSloth • -5 a

Part 2: C++

I used 2x2 matrix operations to represent the operations on the index. After obtaining the matrix *m* for the input, I used fast exponentiation to get the matrix *m*^101741582076661 and multiply this with the vertical vector (2020, 1)

To avoid using big integers, i used the gcc type __int128_t

11 ···

tckmn • -5 a

explanation:

```
nc = 119315717514047
a,b = 1,0
read.lines.reverse.each do |line|
    case line.chomp
    when /new/ then a,b = -a, -b + nc-1
    when /cut (.*)/ then b += $1.to_i
    when /increment (.*)/ then a *= m=invmod $1.to_i, nc; b *= m
    end
end
p a%nc,b%nc
```

i then stuck the constants into mathematica (because i had the foresight to prepare a modular inverse in ruby, but somehow not a powermod) and solved the problem with this expression:

```
Mod[2020 PowerMod[a, n, nc] + b (1 - PowerMod[a, n, nc]) ModularInverse[1 - a, nc],
```

the basic idea is that all the shuffles can be described as operations modulo the card count.

- the reversal maps position *p* to *-p + nc-1* (where *nc* is the number of cards)
- the cut by *n* maps position *p* to *p - n* (which generalizes to negative *n*)
- the deal by *n* maps position *p* to *pn*

therefore, we can fully describe the entire shuffle transformation as *ap+b* for some constants *a* and *b*. specifically, they start out as *1* and *0* respectively, and each line in the shuffle procedure modifies them as follows:

- reversal: *(a, b)* → *(-a, -b + nc-1)*
- cut: *(a, b)* → *(a, b-n)*
- deal: *(a, b)* → *(an, bn)*

we actually want to invert these, though, because we're asked for the card that ends up at 2020, not where 2020 ends up. the inverses are simple, because reversal is its own inverse, cut by *n* just becomes cut by *-n*, and for the deal we take the modular inverse (the number of cards is prime). we also need to make sure to apply the steps in reverse order.

after that (which is what the ruby code does), we have to apply the function *x* → *ax+b* a large number of times. taking a look at the expanded version of the first few applications gives a clue:

x

$ax + b$

$a^2x + ab + b$

$a^3x + a^2b + ab + b$

$a^4x + a^3b + a^1b + ab + b$

which can be factored into

a^n x + b(1-a^n)/(1-a)

the mathematics does this all mod number of cards, which gives the solution

twattanawaroon • -5 a • Modifié il y a -5 a

Python 3, 31/5, with annotations and refactoring after the fact, of course. I tried to keep the explanation somewhat concise.

It seems like everyone write out the repetition using a polynomial-looking thing, but I like a matrix representation for this kind of job. Discussion of this in the code.

My best ranking so far! Playing with operations modulo n definitely has a (ACM/ICPC-style) programming-competition-taste to it, and probably why this was my best performance.

... and actually I used an online tool to find the multiplicative inverse before I actually put it in code lol.

5

p_tseng • -5 a • Modifié il y a -5 a

r/adventofcode • il y a 2 m.

-❄- 2024 Day 22 Solutions -❄-

20 upvotes · 450 commentaires

r/adventofcode • il y a 2 m.

-❄- 2024 Day 23 Solutions -❄-

23 upvotes · 504 commentaires

r/adventofcode • il y a 2 m.

-❄- 2024 Day 19 Solutions -❄-

24 upvotes · 586 commentaires

r/adventofcode • il y a 2 m.

-❄- 2024 Day 21 Solutions -❄-

23 upvotes · 399 commentaires

r/adventofcode • il y a 2 m.

-❄- 2024 Day 17 Solutions -❄-

37 upvotes · 548 commentaires

r/adventofcode • il y a 2 m.

-❄- 2024 Day 18 Solutions -❄-

23 upvotes · 536 commentaires

r/adventofcode • il y a 2 m.

-❄- 2024 Day 25 Solutions -❄-

41 upvotes · 349 commentaires

r/adventofcode • il y a 2 m.

- Let's try it for 10007 instead of 119315717514047. Does the card at position 2020 repeat after a certain time? Well, if it did, it's a long time. Even if it did, how am I supposed to shuffle 119315717514047 cards to find that repeat anyway?
- I obviously can't compute a permutation matrix since it's too big.
- I wrote code to compute "card at position i is at position j at time t - 1" (undo all shuffle steps). I used modular inverse implementation from 2016 day 15 for this. But I can't apply this inverse function 101741582076661 times. Even a simple loop that does *nothing at all* `101741582076661.times { }` does not finish in reasonable time on my computer. So how do you even do this?

I started playing around with the last example given in part 1. I tried seeing what happens if the shuffle is applied twice. The result was... `6 5 4 3 2 1 0 9 8 7` **Then the light bulb turned on**. That made me see that you can simplify sequences of transformations. I played around to see how to properly simplify the transformations. For example, it's obvious that adjacent cuts can simply be added. Adjacent deal with increment just multiply together (the example has a handy pair of 9 and 3 together to help verify that it's the same as if you'd dealt with increment 7). But to really simplify it into a manageable state, I have to figure out how to transpose any two different transforms so that I could get the same ones next to each other, so had to play around with the examples some more. I used the example and my answer on part 1 to help ensure that my simplified transformations were still the same as the original. When the simplification process is complete, the input contains only one of each type of technique. Once it looked like simplification was working 100%, then I used exponentiation to construct the correct transforms for 101741582076661, simplified that, ran it in reverse (so that "undo shuffle" I wrote wasn't a waste after all!!!), and crossed my fingers hoping that the answer produced was correct. And let out a huge sigh of relief when it was.

I think I was not mathematically strong enough to see this faster like some people in this thread apparently did, so it looks like I still have some work to do...

Ruby: 22_slam_shuffle.rb

Haskell: 22_slam_shuffle.hs

5 ...

2 réponses supplémentaires

zniperr • -5 a

Part 1 and 2 in python here. The nice thing about python is that all ints are bigints :)

Figured out the modular multiplicative index thing myself but needed to read someone's hint that all the functions are linear. The rest is just browsing on wikipedia/wolframalpha on how to efficiently compose linear functions. Imo this was way too heavy on math.

5 ...

Afficher plus de commentaires

r/adventofcode • il y a 2 m.
-❄️- 2024 Day 15 Solutions -❄️-
22 upvotes · 405 commentaires

r/adventofcode • il y a 2 m.
-❄️- 2024 Day 24 Solutions -❄️-
32 upvotes · 339 commentaires

r/adventofcode • il y a 2 m.
-❄️- 2024 Day 14 Solutions -❄️-
24 upvotes · 743 commentaires

r/adventofcode • il y a 2 m.
-❄️- 2024 Day 13 Solutions -❄️-
28 upvotes · 771 commentaires

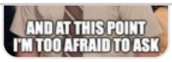
r/adventofcode • il y a 2 m.
-❄️- 2024 Day 11 Solutions -❄️-
19 upvotes · 963 commentaires

r/adventofcode • il y a 2 m.
-❄️- 2024 Day 12 Solutions -❄️-
36 upvotes · 696 commentaires

r/adventofcode • il y a 3 m.
2024 Day 7 Solutions -❄️-
38 upvotes · 1,1 k commentaires

r/adventofcode • il y a 2 m.
[2024 Day 14 (Part 2)] Hear me out: the problem would have been less frustrating for some if it was even more vague
147 upvotes · 69 commentaires

r/adventofcode • il y a 2 m.
[20
54 upvotes · 14 commentaires



234 upvotes · 37 commentaires

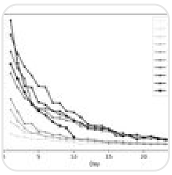
r/adventofcode • il y a 2 m.

[2024 day 11] I made a part 3 to this day

27 upvotes · 13 commentaires

r/adventofcode • il y a 2 m.

it's that time again - see my comment for details



54 upvotes · 34 commentaires

r/adventofcode • il y a 2 m.

[Unpopular opinion] Day 14 part 2's problem was great

529 upvotes · 151 commentaires

r/adventofcode • il y a 3 m.

-❄- 2024 Day 3 Solutions -❄-

57 upvotes · 1,7 k commentaires

r/adventofcode • il y a 2 m.

[2024 Day 11] I was certain that this was going to matter in Part 2, the last bit was even bolded!

522 upvotes · 45 commentaires

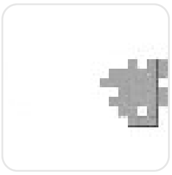
r/adventofcode • il y a 3 m.

-❄- 2024 Day 2 Solutions -❄-

52 upvotes · 1,4 k commentaires

r/adventofcode • il y a 2 m.

[2024 Day 12 (Part 2)] - Visualisation of my first thoughts for Part 2



193 upvotes · 28 commentaires

PUBLICATIONS LES PLUS TENDANCE

Reddit

reReddit : meilleures publications de décembre 2019

Reddit

reReddit : meilleures publications de 2019