Advent of Code 2015 — Explanation

Eugene Obrezkov · Follow
19 min read · Feb 15, 2016

Happy New Year! I've finished Advent of Code and want to share my solutions with you. Of course, some solutions can be improved, I don't claim these to be the best solutions possible.

When I've started play this game, I decided to write one paragraph with a solution and explanation to it. But you can see that I went a little further with that…

TL;DR: This isn't really an article, more like a collection of gists with explanations. All of the solutions are available on my GitHub if you just want the code.

Table of Contents

- Day 1 (Not Quite Lisp)
- Day 2 (I Was Told There Would Be No Math)
- Day 3 (Perfectly Spherical Houses in a Vacuum)
- Day 4 (The Ideal Stocking Stuffer)
- Day 5 (Doesn't He Have Intern-Elves For This?)
- Day 6 (Probably a Fire Hazard)
- Day 7 (Some Assembly Required)
- Day 8 (Matchsticks)
- Day 9 (All in a Single Night)

- Day 10 (Elves Look, Elves Say)
- Day 11 (Corporate Policy)
- Day 12 (JSAbacusFramework.io)
- Day 13 (Knights of the Dinner Table)
- Day 14 (Reindeer Olympics)
- Day 15 (Science for Hungry People)
- Day 16 (Aunt Sue)
- Day 17 (No Such Thing as Too Much)
- Day 18 (Like a GIF For Your Yard)
- Day 19 (Medicine for Rudolph)
- Day 20 (Infinite Elves and Infinite Houses)
- Day 21 (RPG Simulator 20XX)
- Day 22 (Wizard Simulator 20XX)
- Day 23 (Opening the Turing Lock)
- Day 24 (It Hangs in the Balance)
- Day 25 (Let It Snow)

Day 1 — Part 1 (Not Quite Lisp)

Problem definition: link

This one's easy. We just split the input to separate chars, getting the array. Afterwards, iterating over that array via *reduce*, we can calculate current floor and store it in accumulator.

Value from the accumulator will be our result.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('');
const result = INPUT.reduce((floor, direction) => direction === '(' ? ++floor : --floor, 0);

console.log(result);
```

aoc-1-1.js hosted with ❤ by GitHub view raw

Day 1 Part 1

Day 1 — Part 2 (Not Quite Lisp)

Problem definition: link

We need to find the position of the character that causes Santa to first enter the basement.

Quick solution is a mutable variable *floor*, which determines the current floor Santa is on. Split input to separate chars and *map* through it, replacing the char with current floor. That way we get the array with floors, where Santa was.

Afterwards, looking for *-1* floor and getting its index will be an our result.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('');

let floor = 0;
let result = INPUT.map(direction => direction === '(' ? ++floor : --floor).indexOf(-1) + 1;

console.log(result);
```

aoc-1-2.js hosted with ❤ by GitHub view raw

Day 1 Part 2

Day 2 — Part 1 (I Was Told There Would Be No Math)

Problem definition: link

Simple math. All formulas are provided in problem definition. We need to find out total square feet of wrapping paper.

Split the input by NL and iterate the resulting array via *reduce*. On each iteration we get string with sizes divided by *x*, so we split the sizes by *x* character and get *length*, *width* and *height*. Calculate result by formula and sum up with our accumulator.

Answer to this problem will be a value from that accumulator.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const result = INPUT.reduce((total, _lwh) => {
  const lwh = _lwh.split('x');
  const length = lwh[0];
  const width = lwh[1];
  const height = lwh[2];

  return total
    + (2 * length * width)
    + (2 * width * height)
    + (2 * height * length)
    + Math.min(length * width, width * height, height * length);
}, 0);

console.log(result);
```

aoc-2-1.js hosted with ❤ by GitHub view raw

Day 2 Part 1

Day 2 — Part 2 (I Was Told There Would Be No Math)

Problem definition: link

The same applies here, just with different formulas. We split the input by NL character and start reducing the resulting array. On each iteration, we need to split one line from input by *x* character and sort that array because:

> *The ribbon required to wrap a present is the shortest distance around its sides, or the smallest perimeter of any one face.*

We have the sorted array and calculating the smallest perimeter is not a big problem. All what we need to do is get the smallest sides, calculating the result and sum up with accumulator.

Answer to this problem is a value from accumulator.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const result = INPUT.reduce((total, _lwh) => {
  const lwh = _lwh.split('x').map(Number).sort((a, b) => a - b);

  return total
    + (lwh[0] + lwh[0] + lwh[1] + lwh[1])
    + (lwh[0] * lwh[1] * lwh[2])
}, 0);

console.log(result);
```

aoc-2-2.js hosted with ❤ by GitHub view raw

Day 2 Part 2

Day 3 — Part 1 (Perfectly Spherical Houses in a Vacuum)

Problem definition: link

We need to find out the number of houses which received at least one present. It can be easily achieved by using unique *Set* and getting its size after calculating.

Split the input by empty char so we get the array with directions where Santa should be at the next step. When we iterating through that array via *reduce*, accumulator is our current coordinates. Based on direction, we calculate new coordinates, add to the *Set* and return, so on next iteration we have current coordinates.

Size of our unique *Set* will be our houses count that receives at least one present.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('');

// Unique set of coordinates with the starting coordinates already added
const coordinates = new Set().add(`0x0`);

INPUT.reduce((curCoords, direction) => {
  let newCoords = {x: 0, y: 0};

  if (direction === '^') newCoords = {x: curCoords.x, y: curCoords.y + 1};
  if (direction === 'v') newCoords = {x: curCoords.x, y: curCoords.y - 1};
  if (direction === '>') newCoords = {x: curCoords.x + 1, y: curCoords.y};
  if (direction === '<') newCoords = {x: curCoords.x - 1, y: curCoords.y};

  coordinates.add(`${newCoords.x}x${newCoords.y}`);
  return newCoords;
}, {x: 0, y: 0});

console.log(coordinates.size);
```

aoc-3-1.js hosted with ❤ by GitHub view raw

Day 3 Part 1

Day 3 — Part 2 (Perfectly Spherical Houses in a Vacuum)

Problem definition: link

We got a robot now that helps us to send presents. The logic of traversing the path is the same — get all visited coordinates and push them to *Set*. With only one difference… we need to split Santa's directions and RoboSanta's directions.

Take input and split it by empty char. That's way we get all the directions that we can filter out with even and non-even items, which are Santa's and Robot's directions.

traverse function does the same as the code from part 1 — it takes directions and pushes all the visited coordinates to array.

Now, all that's left to do is get visited coordinates for Santa and Robot and concat them into unique *Set*.

Size of this *Set* will be our result.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('');
const santaDirections = INPUT.filter((item, index) => index % 2 === 0);
const roboSantaDirections = INPUT.filter((item, index) => index % 2 === 1);

// Get array of directions and return array of visited coordinates
const traverse = directions => {
  let visitedCoordinates = ['0x0'];
  let currentPosition = {x: 0, y: 0};

  directions.forEach(direction => {
    if (direction === '^') currentPosition.y++;
    if (direction === 'v') currentPosition.y--;
    if (direction === '>') currentPosition.x++;
    if (direction === '<') currentPosition.x--;

    visitedCoordinates.push(`${currentPosition.x}x${currentPosition.y}`);
  });

  return visitedCoordinates;
};

const result = new Set(traverse(santaDirections).concat(traverse(roboSantaDirections))).size;

console.log(result);
```

aoc-3-2.js hosted with ❤ by GitHub view raw

Day 3 Part 2

Day 4 — Part 1 (The Ideal Stocking Stuffer)

Problem definition: link

Increment the counter until our md5 hash starts with five zeros.

The first number that produces required hash is saved in *counter* and is the result.

```javascript
const crypto = require('crypto');
const INPUT = 'ckczppom';
const md5 = data => crypto.createHash('md5').update(data).digest('hex');
const isStartsWithFiveZeros = data => data.slice(0, 5) === '00000';

let counter = 0;
while (!isStartsWithFiveZeros(md5(`${INPUT}${counter}`))) counter++;

console.log(counter);
```

aoc-4-1.js hosted with ❤ by GitHub view raw

Day 4 Part 1

Day 4 — Part 2 (The Ideal Stocking Stuffer)

Problem definition: link

Same as in previous part, but hash must start with six zeros instead of five.

```
const crypto = require('crypto');
const INPUT = 'ckczppom';
const md5 = data => crypto.createHash('md5').update(data).digest('hex');
const isStartsWithSixZeros = data => data.slice(0, 6) === '000000';

let counter = 0;
while (!isStartsWithSixZeros(md5(`${INPUT}${counter}`))) counter++;

console.log(counter);
```

aoc-4-2.js hosted with ❤ by GitHub view raw

Day 4 Part 2

Day 5 — Part 1 (Doesn't He Have Intern-Elves For This?)

Problem definition: link

We need to find out strings that correspond to defined rules in problem definition. Our rules can be implemented as separate functions that accept string and check it against the rule.

Iterate input via *reduce* and if string is nice, increment the accumulator. In result, we get total count of nice strings, which is our answer.

Day 5 Part 1

Day 5 — Part 2 (Doesn’t He Have Intern-Elves For This?)

Problem definition: link

Pretty much the same as the last part, but let’s rewrite our rule functions to regular expressions.

Iterate input via reduce and increment accumulator if string is nice. Accumulator value is the result.

```js
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');

// Part 1 was written with pure functions but Part 2 I've decided to write with RegExp
const isContainPair = string => /([a-z][a-z]).*\1/.test(string);
const isContainRepeatLetter = string => /([a-z])[a-z]\1/.test(string);

const isNiceString = string => !!(isContainPair(string) && isContainRepeatLetter(string));

const result = INPUT.reduce((total, string) => isNiceString(string) ? ++total : total, 0);

console.log(result);
```

aoc-5-2.js hosted with ❤ by GitHub   view raw

Day 5 Part 2

Day 6 — Part 1 (Probably a Fire Hazard)

Problem definition: link

We have a list of commands that say which lights we need to switch. First step is to write regular expression that will parse that command and return object with parsed data.

Also we need to have an array of our lights where we can store current state — *Uint8Array* which is filled by zeros.

For each instruction from Santa, we parse it via regular expression and start switching the lights in defined region by setting 1 or 0 in *LIGHTS* array.

When all instructions are executed, iterate the *LIGHTS* array via *reduce* and sum up all the enabled lights to the accumulator.

Answer to this problem is the accumulator value.

Day 6 Part 1

Day 6 — Part 2 (Probably a Fire Hazard)

Problem definition: link

We found out that our *LIGHTS* array should store brightness of each light instead of the state of light (on\off).

Nothing really changes here except for writing to the *LIGHTS* array. We will increment\decrement values instead of writing 1\0.

When each of instructions are executed, we can calculate total brightness via *reduce* and accumulator. Answer is in accumulator.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const COMMANDS_REGEX = /(turn on|turn off|toggle) (\d+),(\d+) through (\d+),(\d+)/;

// Parse command from string and return object
const parseCommand = _command => {
  let command = _command.match(COMMANDS_REGEX);
  return {command: command[1], x1: +command[2], y1: +command[3], x2: +command[4], y2: +command[
};

// Map of our lights
let LIGHTS = new Uint8Array(1000 * 1000);

// Parse each command and change brightness of our lights
INPUT.forEach(_command => {
  let command = parseCommand(_command);

  for (let x = command.x1; x <= command.x2; x++) {
    for (let y = command.y1; y <= command.y2; y++) {
      let index = 1000 * x + y;

      switch (command.command) {
        case 'turn on':
          LIGHTS[index] += 1;
          break;
        case 'turn off':
          LIGHTS[index] = LIGHTS[index] === 0 ? 0 : LIGHTS[index] - 1;
          break;
        case 'toggle':
          LIGHTS[index] += 2;
          break;
      }
    }
  }
});

// Calculate brightness
const result = LIGHTS.reduce((brightness, light) => brightness + light, 0);

console.log(result);
```

aoc-6-2 is hosted with ❤ by GitHub　　view raw

Day 6 Part 2

Day 7 — Part 1 (Some Assembly Required)

Problem definition: link

Our input contains a list of wires and their values\instructions. We need to split this problem into separate steps:

- Parse instructions from input via regular expression;
- Fill a Map with parsed wires as keys and instructions as values;
- Recursively get values from a Map and if value is an instruction — execute it and return the result;

Let’s start with defining a *Map* which is called *WIRES*. We will store wire name as a key and parsed instruction as a value here.

Afterwards implement functions that will be our instructions from input. I’ve stored them in *BITWISE_METHODS* object.

Parsing is boring, nothing special. We have few regular expressions that return values from input. It returns object with *command, args* and *destination* fields. *command* is our bitwise instruction*, args* are our arguments for instruction and *destination* is a name of the wire which has this instruction.

We are able to fill our *WIRES* map with parsed instructions now. Iterate through *INPUT,* parse the instruction and store the result in our *WIRES* map.

Afterwards, when we have representation of our wires, we can calculate value of a specific wire. When we get value from *WIRES* and it’s a number, we return it. If it’s not a number but an object with instruction, we call our bitwise methods with arguments from object and store the result in *WIRES,* returning the result.

Step by step, our recursive function *calculateWire* will return value from a specific wire.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const COMMAND_REGEX = /[A-Z]+/g;
const ARGUMENTS_REGEX = /[a-z0-9]+/g;

// Our parsed wires in format {wire: value} or {wire: instruction}
const WIRES = new Map();

// Dictionary of our bitwise methods
const BITWISE_METHODS = {
  AND: (a, b) => a & b,
  OR: (a, b) => a | b,
  NOT: a => ~a,
  LSHIFT: (a, b) => a << b,
  RSHIFT: (a, b) => a >> b
};

// Parse instruction from input and return object with command, arguments and destination wire
const parseInstruction = instruction => {
  const command = instruction.match(COMMAND_REGEX);
  const args = instruction.match(ARGUMENTS_REGEX);
  const destination = args.pop();

  return {
    command: command && command[0],
    args: args.map(arg => isNaN(Number(arg)) ? arg : Number(arg)),
    destination: destination
  };
};

// Calculate value for one of the wires (recursively)
const calculateWire = wireName => {
  const wire = WIRES.get(wireName);

  // If wire already calculated, return value
  if (typeof wireName === 'number') return wireName;
  if (typeof wire === 'number') return wire;
  if (typeof wire === 'undefined') return undefined;

  if (!wire.command) {
    WIRES.set(wireName, calculateWire(wire.args[0]));
  } else {
    WIRES.set(wireName, BITWISE_METHODS[wire.command](calculateWire(wire.args[0]), calculateWir
  }

  return WIRES.get(wireName);
};

```

```
// Fill WIRES with parsed instructions and their future values
INPUT.forEach(instruction => {
  const parsedInstruction = parseInstruction(instruction);
  WIRES.set(parsedInstruction.destination, {command: parsedInstruction.command, args: parsedIns
});

console.log(calculateWire('a'));
```

Day 7 Part 1

Day 7 — Part 2 (Some Assembly Required)

Problem definition: link

We can use the same algorithm here to calculate the value of a wire. We just need to set up some initial value as problem definition says:

> *Now, take the signal you got on wire a, override wire b to that signal, and reset the other wires (including wire a).*

I take the result from previous part (which is 956 in my case) and set it implicitly to wire *b*.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const COMMAND_REGEX = /[A-Z]+/g;
const ARGUMENTS_REGEX = /[a-z0-9]+/g;

// Our parsed wires in format {wire: value} or {wire: instruction}
const WIRES = new Map();

// Dictionary of our bitwise methods
const BITWISE_METHODS = {
  AND: (a, b) => a & b,
  OR: (a, b) => a | b,
  NOT: a => ~a,
  LSHIFT: (a, b) => a << b,
  RSHIFT: (a, b) => a >> b
};

// Parse instruction from input and return object with command, arguments and destination wire
const parseInstruction = instruction => {
  const command = instruction.match(COMMAND_REGEX);
  const args = instruction.match(ARGUMENTS_REGEX);
  const destination = args.pop();

  return {
    command: command && command[0],
    args: args.map(arg => isNaN(Number(arg)) ? arg : Number(arg)),
    destination: destination
  };
};

// Calculate value for one of the wires (recursively)
const calculateWire = wireName => {
  const wire = WIRES.get(wireName);

  if (typeof wireName === 'number') return wireName;
  if (typeof wire === 'number') return wire;
  if (typeof wire === 'undefined') return undefined;

  if (!wire.command) {
    WIRES.set(wireName, calculateWire(wire.args[0]));
  } else {
    WIRES.set(wireName, BITWISE_METHODS[wire.command](calculateWire(wire.args[0]), calculateWir
  }

  return WIRES.get(wireName);
};

// Fill WIRES with parsed instructions and their future values
// Fill WIRES with parsed instructions and their future values
INPUT.forEach(instruction => {
  const parsedInstruction = parseInstruction(instruction);
  WIRES.set(parsedInstruction.destination, {command: parsedInstruction.command, args: parsedIns
});

// Set already known value to specific wire
WIRES.set('b', 956);

console.log(calculateWire('a'));
```

Day 7 Part 2

Day 8 — Part 1 (Matchsticks)

Problem definition: link

This one's interesting. At first, I wanted to use regular expressions and replace characters to get length of the string. Afterwards, I thought it can be achieved with simple *eval* of the string.

Reducing the input by calculating the difference between these two strings, we can find the result.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const result = INPUT.reduce((acc, line) => acc + (line.length - eval(line).length), 0);

console.log(result);
```

aoc-8-1.js hosted with ❤ by GitHub view raw

Day 8 Part 1

Day 8 — Part 2 (Matchsticks)

Problem definition: link

Same as the previous part, only in different order. We need to encode the string without evaluation. Two simple regular expressions and sum up with accumulator will be our result.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const result = INPUT.reduce((acc, line) => acc + (2 + line.replace(/\\/g, '\\\\').replace(/"/g,

console.log(result);
```

Day 8 Part 2

Day 9 — Part 1 (All in a Single Night)

Problem definition: link

Simple math again — combinatorics. When I see problems like finding the shortest distance, I bet, it's combinatorics (because I don't know graph theory, my bad). Here's how I split this problem:

- We need to build a map of all possible points pairs and distance between them;
- Build an unique set with all the places that we need to visit;
- Permute all possible places, getting all possible routes;
- Iterate through all the permutations and calculate the total distance for each route;

Our result will be a minimal value from the array, where total distances for each of routes are stored.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const DIRECTION_REGEX = /(\w+) to (\w+) = (\d+)/;

// Takes input and builds map of all possible distances between two points
const buildDistanceMap = input => {
  const map = new Map();

  input.forEach(direction => {
    const parsed = direction.match(DIRECTION_REGEX);
    map.set(`${parsed[1]} -> ${parsed[2]}`, +parsed[3]);
    map.set(`${parsed[2]} -> ${parsed[1]}`, +parsed[3]);
  });

  return map;
};

// Takes input and builds unique set of all places that need to be visited
const buildPlacesSet = input => {
  const places = new Set();

  input.forEach(direction => {
    const parsed = direction.match(DIRECTION_REGEX);
    places.add(parsed[1]).add(parsed[2]);
  });

  return places;
};

// Takes an array of items and builds an array with all possible permutations
const permute = input => {
  const array = Array.from(input);
  const permute = (res, item, key, arr) => {
    return res.concat(arr.length > 1 && arr.slice(0, key).concat(arr.slice(key + 1)).reduce(per
  };

  return array.reduce(permute, []);
};

const distances = buildDistanceMap(INPUT);
const places = buildPlacesSet(INPUT);
const allPossibleRoutes = permute(places);

// Builds an array with all possible distances
const allPossibleDistances = allPossibleRoutes.reduce((acc, route) => {
  let total = 0;

  for (let i = 0; i < route.length; i++) {
```

```
  for (let i = 0; i < route.length; i++) {
    if (route[i + 1] === undefined) break;

    total += distances.get(`${route[i]} -> ${route[i + 1]}`);
  }

  return acc.concat([total]);
}, []);

const result = Math.min.apply(Math, allPossibleDistances);

console.log(result);
```

Day 9 Part 1

Day 9 — Part 2 (All in a Single Night)

Problem definition: link

Problem definition says:

> *The next year, just to show off, Santa decides to take the route with the longest distance instead.*

No problem, just replace the *Math.min* with *Math.max* to calculate the maximum value of our total distances for each of routes.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const DIRECTION_REGEX = /(\w+) to (\w+) = (\d+)/;

// Takes input and builds map of all possible distances between two points
const buildDistanceMap = input => {
  const map = new Map();

  input.forEach(direction => {
    const parsed = direction.match(DIRECTION_REGEX);
    map.set(`${parsed[1]} -> ${parsed[2]}`, +parsed[3]);
    map.set(`${parsed[2]} -> ${parsed[1]}`, +parsed[3]);
  });

  return map;
};

// Takes input and builds unique set of all places that need to be visited
const buildPlacesSet = input => {
  const places = new Set();

  input.forEach(direction => {
    const parsed = direction.match(DIRECTION_REGEX);
    places.add(parsed[1]).add(parsed[2]);
  });

  return places;
};

// Takes array of items and builds array with all possible permutations
const permute = input => {
  const array = Array.from(input);
  const permute = (res, item, key, arr) => {
    return res.concat(arr.length > 1 && arr.slice(0, key).concat(arr.slice(key + 1)).reduce(per
  };

  return array.reduce(permute, []);
};

const distances = buildDistanceMap(INPUT);
const places = buildPlacesSet(INPUT);
const allPossibleRoutes = permute(places);

// All possible distances
const allPossibleDistances = allPossibleRoutes.reduce((acc, route) => {
  let total = 0;

  for (let i = 0; i < route.length; i++) {
```

```
  for (let i = 0; i < route.length; i++) {
    if (route[i + 1] === undefined) break;

    total += distances.get(`${route[i]} -> ${route[i + 1]}`);
  }

  return acc.concat([total]);
}, []);

// Just changed min() method to max()
const result = Math.max.apply(Math, allPossibleDistances);

console.log(result);
```

Day 9 Part 2

Day 10 — Part 1 (Elves Look, Elves Say)

Problem definition: link

We need to find repeating symbols in the string, get its length and replace these symbols with their length and symbol itself.

It's easily achieved with regular expression with global flag. Here, we are reducing all matches of regular expressions, counting their lengths and replacing them with length and symbol.

Our result will be a string after replacing its symbols 40 times.

```
const INPUT = '1113122113';
const FIND_REPETITIONS_REGEX = /(\d)\1*/g;
const lookAndSay = input => input.match(FIND_REPETITIONS_REGEX).reduce((acc, char) => acc + `${

let result = INPUT;
for (let i = 0; i < 40; i++) {
  result = lookAndSay(result);
}

console.log(result.length);
```

aoc-10-1.js hosted with ❤ by GitHub view raw

Day 10 Part 1

Day 10 — Part 2 (Elves Look, Elves Say)

Problem definition: link

Same as previous part, only replace symbols 50 times instead of 40.

```js
const INPUT = '1113122113';
const FIND_REPETITIONS_REGEX = /(\d)\1*/g;
const lookAndSay = input => input.match(FIND_REPETITIONS_REGEX).reduce((acc, char) => acc + `${

let result = INPUT;
for (let i = 0; i < 50; i++) {
  result = lookAndSay(result);
}

console.log(result.length);
```

aoc-10-2.js hosted with ❤ by GitHub view raw

Day 10 Part 2

Day 11 — Part 1 (Corporate Policy)

Problem definition: link

We need to find out the new password for Santa. Let's split the problem into separate steps:

- Implement functions that check string against rules;
- Implement functions for incrementing one char and the whole string;

It's simple with rules — few functions that check string against regular expression.

incrementChar accepts one character and checks if it's equal to "z". If so, return "a", otherwise, get ASCII code of this symbol, increment it by one and return symbol of the new ASCII code.

incrementString is a recursive function which accepts a string that needs to be incremented. We are incrementing the last character in string and if result is "a" then we need to increment character from the left recursively.

Having all these functions we are able to write loop — while our password is not valid — increment the password.

```
const INPUT = 'cqjxjnds';

// Rules for correct password
const isContainStraightIncreasingSymbols = string => string.split('').map(char => char.charCode
const isContainRestrictedSymbols = string => /i|o|l/.test(string);
const isContainPairs = string => /(\w)\1.*(\w)\2/.test(string);

// Increments one char
const incrementChar = char => char === 'z' ? 'a' : String.fromCharCode(char.charCodeAt(0) + 1);

// Increments the whole string by one char recursively
const incrementString = string => {
  const nextChar = incrementChar(string.slice(-1));
  return nextChar === 'a' ? incrementString(string.slice(0, -1)) + 'a' : string.slice(0, -1) +
};

// Checks if password is valid (based on rules above)
const isValidPassword = string => isContainStraightIncreasingSymbols(string) && !isContainRestr

let result = INPUT;
while (!isValidPassword(result)) result = incrementString(result);

console.log(result);
```

aoc-11-1.js hosted with ❤ by GitHub view raw

Day 11 Part 1

Day 11 — Part 2 (Corporate Policy)

Problem definition: link

We need to find the next password now. We had a valid password on previous part, so we can take it as an input for this part.

Algorithm remains the same with one difference… We need to increment our valid password immediately before loop, because our input is already valid.

```javascript
// I'm the laziest man in the world, I know :) Just changed INPUT to the password from part-1
const INPUT = 'cqjxxyzz';

// Rules for correct password
const isContainStraightIncreasingSymbols = string => string.split('').map(char => char.charCode
const isContainRestrictedSymbols = string => /i|o|l/.test(string);
const isContainPairs = string => /(\w)\1.*(\w)\2/.test(string);

// Increments one char
const incrementChar = char => char === 'z' ? 'a' : String.fromCharCode(char.charCodeAt(0) + 1);

// Increments the whole string by one char recursively
const incrementString = string => {
  const nextChar = incrementChar(string.slice(-1));
  return nextChar === 'a' ? incrementString(string.slice(0, -1)) + 'a' : string.slice(0, -1) +
};

// Checks if password is valid (based on rules above)
const isValidPassword = string => isContainStraightIncreasingSymbols(string) && !isContainRestr

// Our input is valid now, so increment it
let result = incrementString(INPUT);
while (!isValidPassword(result)) result = incrementString(result);

console.log(result);
```

aoc-11-2.js hosted with ❤ by GitHub   view raw

Day 11 Part 2

Day 12 — Part 1 (JSAbacusFramework.io)

Problem definition: link

We had been asked to find out the sum of all numbers in the document. It can be achieved via parsing JSON, iterating its result via *reduce* and…wait a minute.

> *What is the sum of all numbers in the document?*

What if we don't need to parse the JSON? Let's write regular expression which finds all the numbers in the document and sum them up.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8');
const NUMBER_REGEX = /-?\d+/g;
const result = INPUT.match(NUMBER_REGEX).reduce((total, number) => total + +number, 0);

console.log(result);
```

aoc-12-1.js hosted with ❤ by GitHub view raw

Day 12 Part 1

Day 12 — Part 2 (JSAbacusFramework.io)

Problem definition: link

A bigger problem now:

> *Ignore any object (and all of its children) which has any property with the value "red". Do this only for objects ({...}), not arrays ([...]).*

We definitely need to parse the JSON now. When parsing JSON we can provide *parse* method with additional function that accepts key and value from current iteration in parsing process.

We need to check if that value is not an array and contains "red". If so — return an empty object (ignoring all the children), otherwise return original value.

That way we can filter out JSON and then apply the same algorithm from the previous part to find out the sum of all numbers in the document.

```
const fs = require('fs');
const NUMBER_REGEX = /-?\d+/g;
const INPUT = JSON.parse(fs.readFileSync('./input.txt', 'utf-8'), (key, value) => {
  if (!Array.isArray(value)) return Object.keys(value).map(key => value[key]).indexOf('red') !=
  return value;
});

const result = JSON.stringify(INPUT).match(NUMBER_REGEX).reduce((total, number) => total + +num

console.log(result);
```

aoc-12-2.js hosted with ❤ by GitHub view raw

Day 12 Part 2

Day 13 — Part 1 (Knights of the Dinner Table)

Problem definition: link

Combinatorics again. I'm starting to love it. The problem is hard enough, so let's break it into smaller pieces:

- Build a map of attributes for each person. This map will contain happiness units for each person in pair with neighbor;
- Build a unique *Set* of all attendees;
- Build all possible permutations of attendees, getting the all possible seat arrangements;

After these steps, we can iterate through all possible permutations via *reduce* and calculate total happiness based on our map that we've built before.

Answer to this problem will be maximum total happiness that can be achieved by different seating arrangements.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const PERSON_ATTRIBUTES_REGEX = /(\w+) would (\w+) (\d+) happiness units by sitting next to (\w

// Generate all possible permutations for an array
const permute = input => {
  const array = Array.from(input);
  const permute = (res, item, key, arr) => {
    return res.concat(arr.length > 1 && arr.slice(0, key).concat(arr.slice(key + 1)).reduce(per
  };

  return array.reduce(permute, []);
};

// Parse input and return map with attributes of each person
const getPersonAttributes = input => {
  return input.reduce((map, person) => {
    const parsed = person.match(PERSON_ATTRIBUTES_REGEX);
    const name = parsed[1];
    const isLose = parsed[2] === 'lose';
    const count = +parsed[3];
    const neighbour = parsed[4];

    return map.set(`${name} -> ${neighbour}`, isLose ? -count : count);
  }, new Map());
};

// Get attendees list
const getAttendees = input => {
  return input.reduce((set, person) => {
    const parsed = person.match(PERSON_ATTRIBUTES_REGEX);
    return set.add(parsed[1]);
  }, new Set());
};

// Get all persons' attributes
const personAttributes = getPersonAttributes(INPUT);

// Get all possible permutations of the guests
const allPossiblePermutations = permute(getAttendees(INPUT));

const totalHappinnes = allPossiblePermutations.reduce((totalHappiness, permutation) => {
  const total = permutation.reduce((total, person, index, arr) => {
    const leftOne = arr[index - 1 < 0 ? arr.length - 1 : index - 1];
    const rightOne = arr[index + 1 > arr.length - 1 ? 0 : index + 1];

    total += personAttributes.get(`${person} -> ${leftOne}`);
    total += personAttributes.get(`${person} -> ${rightOne}`);
    total += personAttributes.get(`${person}->${rightOne}`);

    return total;
  }, 0);

  return total > totalHappiness ? total : totalHappiness;
}, 0);

console.log(totalHappinnes);
```

Day 13 Part 1

Day 13 — Part 2 (Knights of the Dinner Table)

Problem definition: link

Algorithm remains the same except we need to add yourself into attendees list:

> *So, add yourself to the list, and give all happiness relationships that involve you a score of 0.*

We don’t care with whom we are sitting, it simplifies solution a little bit.

When building person attributes map, add yourself as possible neighbor to that person with value of “0” and to the unique *Set* of attendees.

You’re ready to calculate new total happiness, based on our changes.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const PERSON_ATTRIBUTES_REGEX = /(\w+) would (\w+) (\d+) happiness units by sitting next to (\w

// Generate all possible permutations for an array
const permute = input => {
  const array = Array.from(input);
  const permute = (res, item, key, arr) => {
    return res.concat(arr.length > 1 && arr.slice(0, key).concat(arr.slice(key + 1)).reduce(per
  };

  return array.reduce(permute, []);
};

// Parse input and return map with attributes of each person
const getPersonAttributes = input => {
  return input.reduce((map, person) => {
    const parsed = person.match(PERSON_ATTRIBUTES_REGEX);
    const name = parsed[1];
    const isLose = parsed[2] === 'lose';
    const count = +parsed[3];
    const neighbour = parsed[4];

    map.set(`${name} -> ${neighbour}`, isLose ? -count : count);
    map.set(`ghaiklor -> ${name}`, 0);
    map.set(`${name} -> ghaiklor`, 0);

    return map;
  }, new Map());
};

// Get attendees list
const getAttendees = input => {
  return input.reduce((set, person) => {
    const parsed = person.match(PERSON_ATTRIBUTES_REGEX);
    return set.add(parsed[1]);
  }, new Set()).add('ghaiklor');
};

// Get all persons' attributes
const personAttributes = getPersonAttributes(INPUT);

// Get all possible permutations of the guests
const allPossiblePermutations = permute(getAttendees(INPUT));

const totalHappinnes = allPossiblePermutations.reduce((totalHappiness, permutation) => {
  const total = permutation.reduce((total, person, index, arr) => {
    const leftOne = arr[index - 1 < 0 ? arr.length - 1 : index - 1];
    const leftOne = arr[index - 1 < 0 ? arr.length - 1 : index - 1];
    const rightOne = arr[index + 1 > arr.length - 1 ? 0 : index + 1];

    total += personAttributes.get(`${person} -> ${leftOne}`);
    total += personAttributes.get(`${person} -> ${rightOne}`);

    return total;
  }, 0);

  return total > totalHappiness ? total : totalHappiness;
}, 0);

console.log(totalHappinnes);
```

Day 13 Part 2

Day 14 — Part 1 (Reindeer Olympics)

Problem definition: link

We need to find the maximum distance that reindeer can travel.

In this part it can be calculated via formula. As we know, we have three reindeer attributes: speed, active time and rest time. Traveled distance can be calculated at any point in time using these values by a simple formula.

Our result will be the maximum value of traveled distances.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const REINDEER_REGEX = /\d+/g;
const TIME = 2503;

// It can be calculated by formula without cycling it in some kind of loop
const getReindeerDistance = input => {
  const args = input.match(REINDEER_REGEX).map(Number);
  const speed = args[0];
  const time = args[1];
  const rest = args[2];

  return Math.ceil(TIME / (time + rest)) * (speed * time);
};

const result = INPUT.reduce((max, reindeer) => getReindeerDistance(reindeer) > max ? getReindee

console.log(result);
```

aoc-14-1.js hosted with ❤ by GitHub   view raw

Day 14 Part 1

Day 14 — Part 2 (Reindeer Olympics)

Problem definition: link

This part is harder than the first one. We need to know traveled distance by reindeer at each point in time and based on this, calculate points that reindeer achieved.

I wrote a generator that returns reindeer's distance at each point in time which is called *getReindeerDistanceIterator*. I'm stacking all traveled distances of each reindeer into the map and writing it into *allTraveledDistances* map.

What's left is to iterate through *allTraveledDistances* and set one point to the winner at the current point in time.

Maximum value of these points will be our result.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const REINDEER_NAME_REGEX = /^\w+/;
const REINDEER_ARGS_REGEX = /\d+/g;
const REINDEER_POINTS = new Map();
const TIME = 2503;

// Get reindeer name from input
const getReindeerName = input => input.match(REINDEER_NAME_REGEX)[0];

// Calculates distance for one of the reindeer from 0 to 2503 seconds
function* getReindeerDistanceIterator(input) {
  const args = input.match(REINDEER_ARGS_REGEX).map(Number);
  const speed = args[0];
  const time = args[1];
  const rest = args[2];

  let currentDistance = 0;

  for (let currentTime = 0; currentTime <= TIME; currentTime++) {
    let isMoving = (currentTime % (time + rest) <= time) && (currentTime % (time + rest) !== 0)
    yield isMoving ? currentDistance += speed : currentDistance;
  }
}

// Makes map of all distances for all reindeer
const allTraveledDistances = INPUT.reduce((map, reindeer) => map.set(getReindeerName(reindeer),

// Start gathering winners for each second
for (let currentTime = 0; currentTime <= TIME; currentTime++) {
  let winnerInTheRound = '';
  let max = 0;

  for (let reindeerName of allTraveledDistances.keys()) {
    let reindeerTraveled = allTraveledDistances.get(reindeerName)[currentTime];

    if (reindeerTraveled >= max) {
      winnerInTheRound = reindeerName;
      max = reindeerTraveled;
    }
  }

  REINDEER_POINTS.set(winnerInTheRound, (REINDEER_POINTS.get(winnerInTheRound) || 0) + 1);
}

// Calculate the winner and points
const result = Math.max.apply(Math, Array.from(REINDEER_POINTS.values()));
```

```
console.log(result);
```

aoc-14-2 is hosted with ❤ by GitHub view raw

Day 14 Part 2

Day 15 — Part 1 (Science for Hungry People)

Problem definition: link

We know the formula to calculate the score of the cookie. Our task is to find all scores of all cookies with different ingredients.

I've split the problem into separate steps:

- Get a Map with attributes of each ingredient;
- Get a unique Set with all ingredients names;
- Implement a method that accepts ingredients list, their attributes and count of teaspoons of each ingredient. This method returns score of this cookie;

Simple for loop for each of ingredients can generate all possible permutations of how many teaspoons we need to use for cookie. Calling makeCookie method in that loop and stacking the result into an array can give us all possible scores.

Find out maximum score and it will be our result.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const INGREDIENT_ATTRIBUTES_REGEX = /(\w+): capacity (-?\d+), durability (-?\d+), flavor (-?\d+
const TEASPOONS_COUNT = 100;

// Parse input and get attributes for all of ingredients
const getIngredientAttributes = input => {
  return input.reduce((map, ingredient) => {
    const parsed = ingredient.match(INGREDIENT_ATTRIBUTES_REGEX);

    map.set(parsed[1], {
      capacity: +parsed[2],
      durability: +parsed[3],
      flavor: +parsed[4],
      texture: +parsed[5],
      calories: +parsed[6]
    });

    return map;
  }, new Map());
};

// Get list of all available ingredients
const getIngredientList = input => {
  return input.reduce((set, ingredient) => {
    const parsed = ingredient.match(INGREDIENT_ATTRIBUTES_REGEX);
    return set.add(parsed[1]);
  }, new Set());
};

// Make cookie from available ingredients and teaspoons count
const makeCookie = (ingredients, ingredientsAttributes, teaspoons) => {
  const map = new Map();

  ingredients.forEach(ingredient => {
    const attributes = ingredientsAttributes.get(ingredient);
    const teaspoonsCount = teaspoons[ingredient];

    map.set('capacity', (map.get('capacity') || 0) + attributes.capacity * teaspoonsCount);
    map.set('durability', (map.get('durability') || 0) + attributes.durability * teaspoonsCount
    map.set('flavor', (map.get('flavor') || 0) + attributes.flavor * teaspoonsCount);
    map.set('texture', (map.get('texture') || 0) + attributes.texture * teaspoonsCount);
  });

  if (Array.from(map.values()).some(item => item <= 0)) return 0;
  return map.get('capacity') * map.get('durability') * map.get('flavor') * map.get('texture');
};
```

```
const ingredients = getIngredientList(INPUT);
const ingredientsAttributes = getIngredientAttributes(INPUT);
const ALL_POSSIBLE_COOKIES = [];

for (let sprinkles = 0; sprinkles < 100; sprinkles++) {
  for (let butterscotch = 0; butterscotch < 100 - sprinkles; butterscotch++) {
    for (let chocolate = 0; chocolate < 100 - sprinkles - butterscotch; chocolate++) {
      let candy = 100 - sprinkles - butterscotch - chocolate;
      ALL_POSSIBLE_COOKIES.push(makeCookie(ingredients, ingredientsAttributes, {
        'Sprinkles': sprinkles,
        'Butterscotch': butterscotch,
        'Chocolate': chocolate,
        'Candy': candy
      }));
    }
  }
}

const result = ALL_POSSIBLE_COOKIES.sort((a, b) => b - a)[0];

console.log(result);
```

Day 15 Part 1

Day 15 — Part 2 (Science for Hungry People)

Problem definition: link

Calories is the important value now. We need to do the same — calculate score of each cookie and filter out cookies that don't equal to 500.

I've modified *makeCookie* method so it returns score of the cookie and its calories in the array.

Other than that, algorithm remains the same. We are stacking all possible cookies in the array that contains score and calories. Filtering out that array by cookie's calories is equal to 500 and finding maximum value is our result.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const INGREDIENT_ATTRIBUTES_REGEX = /(\w+): capacity (-?\d+), durability (-?\d+), flavor (-?\d+
const TEASPOONS_COUNT = 100;

// Parse input and get attributes for all of ingredients
const getIngredientAttributes = input => {
  return input.reduce((map, ingredient) => {
    const parsed = ingredient.match(INGREDIENT_ATTRIBUTES_REGEX);

    map.set(parsed[1], {
      capacity: +parsed[2],
      durability: +parsed[3],
      flavor: +parsed[4],
      texture: +parsed[5],
      calories: +parsed[6]
    });

    return map;
  }, new Map());
};

// Get list of all available ingredients
const getIngredientList = input => {
  return input.reduce((set, ingredient) => {
    const parsed = ingredient.match(INGREDIENT_ATTRIBUTES_REGEX);
    return set.add(parsed[1]);
  }, new Set());
};

// Make cookie from available ingredients and teaspoons count
const makeCookie = (ingredients, ingredientsAttributes, teaspoons) => {
  const map = new Map();

  ingredients.forEach(ingredient => {
    const attributes = ingredientsAttributes.get(ingredient);
    const teaspoonsCount = teaspoons[ingredient];

    map.set('capacity', (map.get('capacity') || 0) + attributes.capacity * teaspoonsCount);
    map.set('durability', (map.get('durability') || 0) + attributes.durability * teaspoonsCount
    map.set('flavor', (map.get('flavor') || 0) + attributes.flavor * teaspoonsCount);
    map.set('texture', (map.get('texture') || 0) + attributes.texture * teaspoonsCount);
    map.set('calories', (map.get('calories') || 0) + attributes.calories * teaspoonsCount);
  });

  if (Array.from(map.values()).some(item => item <= 0)) return [0, map.get('calories')];
  return [map.get('capacity') * map.get('durability') * map.get('flavor') * map.get('texture'),
};
});

const ingredients = getIngredientList(INPUT);
const ingredientsAttributes = getIngredientAttributes(INPUT);
const ALL_POSSIBLE_COOKIES = [];

for (let sprinkles = 0; sprinkles < 100; sprinkles++) {
  for (let butterscotch = 0; butterscotch < 100 - sprinkles; butterscotch++) {
    for (let chocolate = 0; chocolate < 100 - sprinkles - butterscotch; chocolate++) {
      let candy = 100 - sprinkles - butterscotch - chocolate;
      let cookie = makeCookie(ingredients, ingredientsAttributes, {
        'Sprinkles': sprinkles,
        'Butterscotch': butterscotch,
        'Chocolate': chocolate,
        'Candy': candy
      });

      ALL_POSSIBLE_COOKIES.push({score: cookie[0], calories: cookie[1]});
    }
  }
}

const result = ALL_POSSIBLE_COOKIES.filter(cookie => cookie.calories === 500).sort((a, b) => b.

console.log(result);
```

Day 15 Part 2

Day 16 — Part 1 (Aunt Sue)

Problem definition: link

We need to find out the number of Sue. Your input contains the list of all Sues' things that have been presented to you. Finding Sue, which has things from our signature will be our result.

Let's start with defining our signature from problem definition and regular expression that grabs things from your input.

Filtering our input by condition that Sue has exactly all the things from the list we can find the number of Sue and that is our result.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const AUNT_REGEX = /Sue (\d+): (\w+): (\d+), (\w+): (\d+), (\w+): (\d+)/;
const SIGNATURE = {
  children: 3,
  cats: 7,
  samoyeds: 2,
  pomeranians: 3,
  akitas: 0,
  vizslas: 0,
  goldfish: 5,
  trees: 3,
  cars: 2,
  perfumes: 1
};

const result = INPUT.filter(item => {
  const parsed = item.match(AUNT_REGEX);

  return (
    SIGNATURE[parsed[2]] == parsed[3] &&
    SIGNATURE[parsed[4]] == parsed[5] &&
    SIGNATURE[parsed[6]] == parsed[7]
  )
})[0].match(AUNT_REGEX)[1];

console.log(result);
```

aoc-16-1.js hosted with ❤ by GitHub view raw

Day 16 Part 1

Day 16 — Part 2 (Aunt Sue)

Problem definition: link

We don't have strict conditions in this part. Things' count can be greater or lesser now.

Replace values by functions that accept these values in our signature. These functions must return true or false, based on new conditions from problem definition.

Filtering our input we should call function from signature, providing the value from the input.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const AUNT_REGEX = /Sue (\d+): (\w+): (\d+), (\w+): (\d+), (\w+): (\d+)/;
const SIGNATURE = {
  children: value => value == 3,
  cats: value => value > 7,
  samoyeds: value => value == 2,
  pomeranians: value => value < 3,
  akitas: value => value == 0,
  vizslas: value => value == 0,
  goldfish: value => value < 5,
  trees: value => value > 3,
  cars: value => value == 2,
  perfumes: value => value == 1
};

const result = INPUT.filter(item => {
  const parsed = item.match(AUNT_REGEX);

  return (
    SIGNATURE[parsed[2]](parsed[3]) &&
    SIGNATURE[parsed[4]](parsed[5]) &&
    SIGNATURE[parsed[6]](parsed[7])
  )
})[0].match(AUNT_REGEX)[1];

console.log(result);
```

aoc-16-2.js hosted with ❤ by GitHub view raw

Day 16 Part 2

Day 17 — Part 1 (No Such Thing as Too Much)

Problem definition: link

Your current task says:

> *How many different **combinations** of containers can exactly fit all 150 liters of eggnog?*

Require *combinatorics* module and start iterating all possible combinations of defined containers. If sum of these containers is equal to 150 — increment the counter.

Result to our question is *total.*

```javascript
const Combinatorics = require('./combinatorics');
const CONTAINERS = [11, 30, 47, 31, 32, 36, 3, 1, 5, 3, 32, 36, 15, 11, 46, 26, 28, 1, 19, 3];

let total = 0;
for (let i = 1; i < CONTAINERS.length - 1; i++) {
  let combination = Combinatorics.combination(CONTAINERS, i);
  let c = [];

  while (c = combination.next()) {
    if (c.reduce((a, b) => a + b) === 150) total++;
  }
}

console.log(total);
```

Day 17 Part 1

Day 17 — Part 2 (No Such Thing as Too Much)

Problem definition: link

Sort the *CONTAINERS* array in descending order. Find out that you need at least 4 containers to get 150 liters.

Everything else with no changes. Iterate through all possible combinations with minimal length of 4 and accumulate total count.

```javascript
const Combinatorics = require('./combinatorics');
const CONTAINERS = [11, 30, 47, 31, 32, 36, 3, 1, 5, 3, 32, 36, 15, 11, 46, 26, 28, 1, 19, 3].s
const MIN_COUNT = 4;

let total = 0;
let combination = Combinatorics.combination(CONTAINERS, MIN_COUNT);
let c;

while (c = combination.next()) {
  if (c.reduce((a, b) => a + b) === 150) total++;
}

console.log(total);
```

aoc-17-2.js hosted with ❤ by GitHub view raw

Day 17 Part 2

Day 18 — Part 1 (Like a GIF For Your Yard)

Problem definition: link

When I'd read these lines at first, I've thought that this is Conway's Game of Life.

> *The state a light should have next is based on its current state (on or off) plus the number of neighbours that are on.*

These conditions are:

- A light which is on stays on when 2 or 3 neighbors are on, and turns off otherwise.
- A light which is off turns on if exactly 3 neighbors are on, and stays off otherwise.

It's definitely Conway's Game of Life.

I'm not going to explain how to implement Conway's Game of Life. You can find plenty of solutions on the internet.

The answer to this problem will be the count of enabled lights after 100 ticks.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split(/\n|/);
const WIDTH = 100;
const HEIGHT = 100;

class Grid {
  constructor(width, height, cells) {
    this.width = width;
    this.height = height;
    this.cells = cells;
    this.onSymbol = '#';
    this.offSymbol = '.';
  }

  getCell(x, y) {
    return this.cells[this.width * x + y];
  }

  setCell(x, y, value) {
    this.cells[this.width * x + y] = value;
    return this;
  }

  toggleCell(x, y) {
    return this.setCell(x, y, !this.isOn(x, y));
  }

  isOn(x, y) {
    return this.getCell(x, y) === this.onSymbol;
  }

  isOff(x, y) {
    return this.getCell(x, y) === this.offSymbol;
  }

  isInGrid(x, y) {
    return (x >= 0 && x < this.width && y >= 0 && y < this.height);
  }

  getNeighboursCount(x, y) {
    let count = this.isOn(x, y) ? -1 : 0;

    for (let yD = 0; yD < 3; yD++) {
      for (let xD = 0; xD < 3; xD++) {
        if (this.isInGrid(x + xD - 1, y + yD - 1) && this.isOn(x + xD - 1, y + yD - 1)) {
          count++;
        }
      }
    }

    return count;
  }

  tick() {
    let cells = new Array(this.width * this.height).fill(this.offSymbol);

    for (let y = 0; y < this.height; y++) {
      for (let x = 0; x < this.width; x++) {
        let onLightsCount = this.getNeighboursCount(x, y);

        if (this.isOn(x, y)) {
          if (onLightsCount === 2 || onLightsCount === 3) {
            cells[this.width * x + y] = this.onSymbol;
          }
        } else if (onLightsCount === 3) {
          cells[this.width * x + y] = this.onSymbol;
        }
      }
    }

    this.cells = cells;
  }

  render() {
    for (let y = 0; y < this.height; y++) {
      for (let x = 0; x < this.width; x++) {
        process.stdout.write(this.isOn(x, y) ? this.onSymbol : this.offSymbol);
      }

      process.stdout.write('\n');
    }
  }
}

let grid = new Grid(WIDTH, HEIGHT, INPUT);
for (let i = 0; i < 100; i++) grid.tick();

const result = grid.cells.reduce((total, cell) => cell === '#' ? ++total : total, 0);

console.log(result);
```

Day 18 Part 1

Day 18 — Part 2 (Like a GIF For Your Yard)

Problem definition: link

The same Conway’s Game of Life with fixed corners in your grid because:

> *Four lights, one in each corner, are stuck on and can’t be turned off.*

I didn’t think a lot and just hardcode state of each corner in *tick()* method and *constructor()*.

Result is the same — total count of enabled lights.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split(/\n|/);
const WIDTH = 100;
const HEIGHT = 100;

class Grid {
  constructor(width, height, cells) {
    this.width = width;
    this.height = height;
    this.cells = cells;
    this.onSymbol = '#';
    this.offSymbol = '.';

    this.cells[0] = '#';
    this.cells[this.width - 1] = '#';
    this.cells[this.width * this.height - this.width] = '#';
    this.cells[this.width * this.height - 1] = '#';
  }

  getCell(x, y) {
    return this.cells[this.width * y + x];
  }

  setCell(x, y, value) {
    this.cells[this.width * y + x] = value;
    return this;
  }

  toggleCell(x, y) {
    return this.setCell(x, y, !this.isOn(x, y));
  }

  isOn(x, y) {
    return this.getCell(x, y) === this.onSymbol;
  }

  isOff(x, y) {
    return this.getCell(x, y) === this.offSymbol;
  }

  isInGrid(x, y) {
    return (x >= 0 && x < this.width && y >= 0 && y < this.height);
  }

  getNeighboursCount(x, y) {
    let count = this.isOn(x, y) ? -1 : 0;

    for (let yD = 0; yD < 3; yD++) {
```

```
    for (let yD = 0; yD < 3; yD++) {
      for (let xD = 0; xD < 3; xD++) {
        if (this.isInGrid(x + xD - 1, y + yD - 1) && this.isOn(x + xD - 1, y + yD - 1)) {
          count++;
        }
      }
    }

    return count;
  }

  tick() {
    let cells = new Array(this.width * this.height).fill(this.offSymbol);

    cells[0] = '#';
    cells[this.width - 1] = '#';
    cells[this.width * this.height - this.width] = '#';
    cells[this.width * this.height - 1] = '#';

    for (let y = 0; y < this.height; y++) {
      for (let x = 0; x < this.width; x++) {
        let onLightsCount = this.getNeighboursCount(x, y);

        if (this.isOn(x, y)) {
          if (onLightsCount === 2 || onLightsCount === 3) {
            cells[this.width * y + x] = this.onSymbol;
          }
        } else if (onLightsCount === 3) {
          cells[this.width * y + x] = this.onSymbol;
        }
      }
    }

    this.cells = cells;
  }

  render() {
    for (let y = 0; y < this.height; y++) {
      for (let x = 0; x < this.width; x++) {
        process.stdout.write(this.isOn(x, y) ? this.onSymbol : this.offSymbol);
      }

      process.stdout.write('\n');
    }
  }
}

let grid = new Grid(WIDTH, HEIGHT, INPUT);
```

```
for (let i = 0; i < 100; i++) grid.tick();

const result = grid.cells.reduce((total, cell) => cell === '#' ? ++total : total, 0);

console.log(result);
```

Day 18 Part 2

Day 19 — Part 1 (Medicine for Rudolph)

Problem definition: link

This is a really tough one.

We have a list of all possible replacements for our molecule in *REPLACEMENTS* constant.

Our task is to replace one sub-molecule from *MOLECULE* with another one and add resulting molecule to *ALL_MOLECULES* set so we can count unique molecules after replacement.

Answer to this problem is the size of our unique set of molecules.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8');
const REPLACEMENTS = INPUT.split('\n\n')[0].split('\n');
const MOLECULE = INPUT.split('\n\n')[1];
const ALL_MOLECULES = new Set();

REPLACEMENTS.forEach(replacement => {
  const from = replacement.split(' => ')[0];
  const to = replacement.split(' => ')[1];
  const findRegex = new RegExp(from, 'g');
  const replaceRegex = new RegExp(from);

  let match;
  while (match = findRegex.exec(MOLECULE)) {
    ALL_MOLECULES.add(MOLECULE.slice(0, match.index) + MOLECULE.slice(match.index).replace(repl
  }
});

console.log(ALL_MOLECULES.size);
```

aoc-19-1.js hosted with ❤ by GitHub view raw

Day 19 Part 1

Day 19 — Part 2 (Medicine for Rudolph)

Problem definition: link

The process is the same as in previous part but in reverse.

We have a big molecule that we need to collapse to single character *e*. It must be done via possible replacements provided in our input.

For that, I've created *REPLACEMENTS* map which contains resulting molecule after replacement as a key and molecule that I can replace as a value. It's done that way because we need to do replacements in reverse order.

The last move is loop while our molecule is not *e*. This loop grabs random molecule from our replacements map and replace part of our molecule with random molecule, counting the counter alongside.

Result to this problem is our counter.

```javascript
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8');
const REPLACEMENTS = INPUT.split('\n\n')[0].split('\n').reduce((map, r) => map.set(r.split(' =>

let MOLECULE = INPUT.split('\n\n')[1];
let count = 0;

while (MOLECULE !== 'e') {
  const randomMolecule = Array.from(REPLACEMENTS.keys())[Math.round(Math.random() * REPLACEMENT

  MOLECULE = MOLECULE.replace(randomMolecule, match => {
    count++;
    return REPLACEMENTS.get(match);
  });

  console.log(`${MOLECULE} -> ${count}`);
}
```

Day 19 Part 2

Day 20 — Part 1 (Infinite Elves and Infinite Houses)

Problem definition: link

We need to find out which houses get at least as many presents as in your input to this problem, in my case — 34,000,000. A simple task to find maximum value.

As our input has a huge number, loop will be slow enough to wait for about few minutes. I decided to optimise some points:

- Using typed *Uint32Array* with predefined length against dynamic empty array;
- Each elf delivers *elf * 10* presents to a house so we can divide input by 10, decreasing iterations of our loop;

Afterwards, our loop just sums up elf's number to a value in *houses* array and if this value if greater that our input — we have an answer.

```javascript
var INPUT = 34000000 / 10;
var houses = new Uint32Array(INPUT);
var houseNumber = INPUT;

for (var i = 1; i < INPUT; i++) {
  for (var j = i; j < INPUT; j += i) {
    if ((houses[j] += i) >= INPUT && j < houseNumber) houseNumber = j;
  }
}

console.log(houseNumber);
```

aoc-20-1.js hosted with ❤ by GitHub view raw

Day 20 Part 1

Day 20 — Part 2 (Infinite Elves and Infinite Houses)

Problem definition: link

The same logic applies here but we need to calculate visits of our elves because:

> *Each Elf will stop after delivering presents to 50 houses.*

Try to not forget that our elves delivers 11 presents as well, multiplying the presents count by 11 and summing up in *houses* array.

The result as in previous part — our *houseNumber*.

```
var INPUT = 34000000 / 10;
var houses = new Uint32Array(INPUT);
var houseNumber = INPUT;

for (var i = 1; i < INPUT; i++) {
  var visits = 0;
  for (var j = i; j < INPUT; j += i) {
    if ((houses[j] = (houses[j] || 11) + i * 11) >= INPUT * 10 && j < houseNumber) houseNumber

    visits++;
    if (visits === 50) break;
  }
}

console.log(houseNumber);
```

aoc-20-2.js hosted with ❤ by GitHub view raw

Day 20 Part 2

Day 21 — Part 1 (RPG Simulator 20XX)

Problem definition: link

Let's split our task into steps:

- We have a store where we can buy weapons, armor and rings — simple constants;
- We need to calculate total stats that we have from our equipment and based on our stats calculate that damage per second;
- We need to find the best equipment with the lowest price — combinatorics;
- Play the game with all possible combinations of our equipment and find the lowest price of this.

The simplest part — declaration of our store which is a *Map*. Each of our *Map* has a name of item as key and an object with *cost*, *damage*, *armor* properties as value.

Having all these stuff we can write function that accepts these items and calculate total stats of your hero — *getTotalStats()*.

When you know your hero's stats you can calculate damage per round which is a simple substraction of your damage from boss armor. Dividing boss health points by your damage per round you can calculate how many rounds you need to play to win — *hitPerSecond()* and *makeMove()*.

We have all what we need to calculate state of the game. Now, we need to generate all possible equipment bundles which is simple to implement with generator — *possibleBundles()*. The logic there is simple — iterating through all store, yield total stats of current iteration.

The best part of this day — solution. Iterate your generator, calling the *makeMove* function and find the minimum price.

```javascript
const WEAPONS = new Map([
  ['dagger', {cost: 8, damage: 4, armor: 0}],
  ['shortsword', {cost: 10, damage: 5, armor: 0}],
  ['warhammer', {cost: 25, damage: 6, armor: 0}],
  ['longsword', {cost: 40, damage: 7, armor: 0}],
  ['greataxe', {cost: 74, damage: 8, armor: 0}]
]);

const ARMOR = new Map([
  ['nothing', {cost: 0, damage: 0, armor: 0}],
  ['leather', {cost: 13, damage: 0, armor: 1}],
  ['chainmail', {cost: 31, damage: 0, armor: 2}],
  ['splintmail', {cost: 53, damage: 0, armor: 3}],
  ['bandedmail', {cost: 75, damage: 0, armor: 4}],
  ['platemail', {cost: 102, damage: 0, armor: 5}]
]);

const RINGS = new Map([
  ['nothing', {cost: 0, damage: 0, armor: 0}],
  ['damage+1', {cost: 25, damage: 1, armor: 0}],
  ['damage+2', {cost: 50, damage: 2, armor: 0}],
  ['damage+3', {cost: 100, damage: 3, armor: 0}],
  ['defense+1', {cost: 20, damage: 0, armor: 1}],
  ['defense+2', {cost: 40, damage: 0, armor: 2}],
  ['defense+3', {cost: 80, damage: 0, armor: 3}]
]);

const BOSS = new Map([
  ['damage', 8],
  ['armor', 2],
  ['health', 100]
]);

const PLAYER = new Map([
  ['damage', 0],
  ['armor', 0],
  ['health', 100]
]);

const getTotalStats = (weapon, armor, leftRing, rightRing) => {
  return {
    cost: weapon.cost + armor.cost + leftRing.cost + rightRing.cost,
    damage: weapon.damage + armor.damage + leftRing.damage + rightRing.damage,
    armor: weapon.armor + armor.armor + leftRing.armor + rightRing.armor
  };
};

const hitPerSecond = (defenderHealth, defenderArmor, attackerDmg) => Math.ceil(defenderHealth /
48  const hitPerSecond = (defenderHealth, defenderArmor, attackerDmg) => Math.ceil(defenderHealth /
49  const makeMove = (boss, player) => hitPerSecond(boss.get('health'), boss.get('armor'), player.g
50
51  function* possibleBundles() {
52    for (let weapon of WEAPONS.values()) {
53      for (let armor of ARMOR.values()) {
54        for (let leftRing of RINGS.values()) {
55          for (let rightRing of RINGS.values()) {
56            if (rightRing.cost !== leftRing.cost) yield getTotalStats(weapon, armor, leftRing, ri
57          }
58        }
59      }
60    }
61  }
62
63  let result = Infinity;
64  for (let bundle of possibleBundles()) {
65    PLAYER.set('damage', bundle.damage).set('armor', bundle.armor);
66    if (makeMove(BOSS, PLAYER)) result = Math.min(result, bundle.cost);
67  }
68
69  console.log(result);
```

Day 21 Part 1

Day 21 — Part 2 (RPG Simulator 20XX)

Problem definition: link

Whoah, all remains the same, except:

> *What is the most amount of gold you can spend and still lose the fight?*

Just update your code to find the maximum price when you lose (*Line 66*).

```javascript
const WEAPONS = new Map([
  ['dagger', {cost: 8, damage: 4, armor: 0}],
  ['shortsword', {cost: 10, damage: 5, armor: 0}],
  ['warhammer', {cost: 25, damage: 6, armor: 0}],
  ['longsword', {cost: 40, damage: 7, armor: 0}],
  ['greataxe', {cost: 74, damage: 8, armor: 0}]
]);

const ARMOR = new Map([
  ['nothing', {cost: 0, damage: 0, armor: 0}],
  ['leather', {cost: 13, damage: 0, armor: 1}],
  ['chainmail', {cost: 31, damage: 0, armor: 2}],
  ['splintmail', {cost: 53, damage: 0, armor: 3}],
  ['bandedmail', {cost: 75, damage: 0, armor: 4}],
  ['platemail', {cost: 102, damage: 0, armor: 5}]
]);

const RINGS = new Map([
  ['nothing', {cost: 0, damage: 0, armor: 0}],
  ['damage+1', {cost: 25, damage: 1, armor: 0}],
  ['damage+2', {cost: 50, damage: 2, armor: 0}],
  ['damage+3', {cost: 100, damage: 3, armor: 0}],
  ['defense+1', {cost: 20, damage: 0, armor: 1}],
  ['defense+2', {cost: 40, damage: 0, armor: 2}],
  ['defense+3', {cost: 80, damage: 0, armor: 3}]
]);

const BOSS = new Map([
  ['damage', 8],
  ['armor', 2],
  ['health', 100]
]);

const PLAYER = new Map([
  ['damage', 0],
  ['armor', 0],
  ['health', 100]
]);

const getTotalStats = (weapon, armor, leftRing, rightRing) => {
  return {
    cost: weapon.cost + armor.cost + leftRing.cost + rightRing.cost,
    damage: weapon.damage + armor.damage + leftRing.damage + rightRing.damage,
    armor: weapon.armor + armor.armor + leftRing.armor + rightRing.armor
  };
};

const hitPerSecond = (defenderHealth, defenderArmor, attackerDmg) => Math.ceil(defenderHealth /
const hitPerSecond = (defenderHealth, defenderArmor, attackerDmg) => Math.ceil(defenderHealth /
const makeMove = (boss, player) => hitPerSecond(boss.get('health'), boss.get('armor'), player.g

function* possibleBundles() {
  for (let weapon of WEAPONS.values()) {
    for (let armor of ARMOR.values()) {
      for (let leftRing of RINGS.values()) {
        for (let rightRing of RINGS.values()) {
          if (rightRing.cost !== leftRing.cost) yield getTotalStats(weapon, armor, leftRing, ri
        }
      }
    }
  }
}

let result = 0;
for (let bundle of possibleBundles()) {
  PLAYER.set('damage', bundle.damage).set('armor', bundle.armor);
  if (!makeMove(BOSS, PLAYER)) result = Math.max(result, bundle.cost);
}

console.log(result);
```

Day 21 Part 2

Day 22 — Part 1 (Wizard Simulator 20XX)

Problem definition: link

I was trying to find the solution around 2 days and still unsuccessful.

All what I can say here — is "Thanks" to some guy from reddit, who had posted solution there. I don't remember his nickname, but if you are reading this now — contact me and I'll mention your name here.

```
class Player {
  constructor(initial, isWizard) {
    this.history = [];
    this.initial = initial;
    this.isWizard = !!isWizard;

    if (this.isWizard) {
      this.spells = [{
        cost: 53,
        effect: (m, o) => o.damage(4)
      }, {
        cost: 73,
        effect: (m, o) => {
          o.damage(2);
          m.hp += 2;
        }
      }, {
        cost: 113,
        start: (m, o) => m.armor += 7,
        effect: (m, o) => {
        },
        end: (m, o) => m.armor -= 7,
        duration: 6
      }, {
        cost: 173,
        effect: (m, o) => o.damage(3),
        duration: 6
      }, {
        cost: 229,
        effect: (m, o) => m.mana += 101,
        duration: 5
      }];
    }

    this.start();
  }

  attack(opponent, spellIdx) {
    if (!this.isWizard) {
      opponent.damage(this.damageAmt);
    } else {
      let spell = this.spells[spellIdx];
      this.history.push(spellIdx);
      this.spent += spell.cost;
      this.mana -= spell.cost;

      if (spell.duration) {
        let newSpell = {idx: spellIdx, effect: spell.effect, duration: spell.duration};
        let newSpell = {idx: spell.idx, effect: spell.effect, duration: spell.duration};
        if (spell.start) spell.start(this, opponent);
        if (spell.end) newSpell.end = spell.end;
        this.activeSpells.push(newSpell);
      } else {
        spell.effect(this, opponent);
      }
    }
  }

  damage(n) {
    this.hp -= Math.max(1, n - this.armor);
  }

  duplicate() {
    let newPlayer = new Player(this.initial, this.isWizard);

    newPlayer.hp = this.hp;
    newPlayer.spent = this.spent;
    newPlayer.armor = this.armor;
    newPlayer.turn = this.turn;

    for (let i = 0; i < this.activeSpells.length; i++) newPlayer.activeSpells.push(Object.assi
    for (let i = 0; i < this.history.length; i++) newPlayer.history.push(this.history[i]);

    if (this.isWizard) {
      newPlayer.mana = this.mana;
    } else {
      newPlayer.damageAmt = this.damageAmt;
    }

    return newPlayer;
  }

  takeTurn(opponent) {
    this.turn++;

    for (let i = 0; i < this.activeSpells.length; i++) {
      let spell = this.activeSpells[i];

      if (spell.duration > 0) {
        spell.effect(this, opponent);
        spell.duration--;

        if (spell.duration === 0 && spell.end) spell.end(this, opponent);
      }
    }
  }

  start() {
    this.hp = this.initial.hp;
    this.spent = 0;
    this.armor = 0;
    this.turn = 0;
    this.activeSpells = [];
    if (this.isWizard) {
      this.mana = this.initial.mana;
    } else {
      this.damageAmt = this.initial.damageAmt;
    }
  }
}

let me = new Player({hp: 50, mana: 500}, true);
let boss = new Player({hp: 55, damageAmt: 8});
let cheapestSpent = Infinity;

function playAllGames(me, boss) {
  for (let i = 0; i < me.spells.length; i++) {
    let spellMatch = false;

    for (let j = 0; j < me.activeSpells.length; j++) {
      if (me.activeSpells[j].duration > 1 && i === me.activeSpells[j].idx) spellMatch = true;
    }

    if (spellMatch) continue;
    if (me.spells[i].cost > me.mana) continue;

    let newMe = me.duplicate();
    let newBoss = boss.duplicate();

    newMe.takeTurn(newBoss);
    newBoss.takeTurn(newMe);
    newMe.attack(newBoss, i);

    newMe.takeTurn(newBoss);
    newBoss.takeTurn(newMe);
    newBoss.attack(newMe);

    if (newBoss.hp <= 0) cheapestSpent = Math.min(cheapestSpent, newMe.spent);
    if (newMe.hp > 0 && newBoss.hp > 0 && newMe.spent < cheapestSpent) playAllGames(newMe, new
  }
}

playAllGames(me, boss);

```

```
console.log(cheapestSpent);
```

Day 22 Part 1

Day 22 — Part 2 (Wizard Simulator 20XX)

Problem definition: link

Problem remains the same with one difference:

> *At the start of each player turn (before any other effects apply), you lose 1 hit point.*

Just add decrementing the health points at each turn (*Line 129*).

```
class Player {
  constructor(initial, isWizard) {
    this.history = [];
    this.initial = initial;
    this.isWizard = !!isWizard;

    if (this.isWizard) {
      this.spells = [{
        cost: 53,
        effect: (m, o) => o.damage(4)
      }, {
        cost: 73,
        effect: (m, o) => {
          o.damage(2);
          m.hp += 2;
        }
      }, {
        cost: 113,
        start: (m, o) => m.armor += 7,
        effect: (m, o) => {
        },
        end: (m, o) => m.armor -= 7,
        duration: 6
      }, {
        cost: 173,
        effect: (m, o) => o.damage(3),
        duration: 6
      }, {
        cost: 229,
        effect: (m, o) => m.mana += 101,
        duration: 5
      }];
    }

    this.start();
  }

  attack(opponent, spellIdx) {
    if (!this.isWizard) {
      opponent.damage(this.damageAmt);
    } else {
      let spell = this.spells[spellIdx];
      this.history.push(spellIdx);
      this.spent += spell.cost;
      this.mana -= spell.cost;

      if (spell.duration) {
        let newSpell = {idx: spellIdx, effect: spell.effect, duration: spell.duration};
        let newSpell = {idx: spell.idx, effect: spell.effect, duration: spell.duration};
        if (spell.start) spell.start(this, opponent);
        if (spell.end) newSpell.end = spell.end;
        this.activeSpells.push(newSpell);
      } else {
        spell.effect(this, opponent);
      }
    }
  }

  damage(n) {
    this.hp -= Math.max(1, n - this.armor);
  }

  duplicate() {
    let newPlayer = new Player(this.initial, this.isWizard);

    newPlayer.hp = this.hp;
    newPlayer.spent = this.spent;
    newPlayer.armor = this.armor;
    newPlayer.turn = this.turn;

    for (let i = 0; i < this.activeSpells.length; i++) newPlayer.activeSpells.push(Object.assi
    for (let i = 0; i < this.history.length; i++) newPlayer.history.push(this.history[i]);

    if (this.isWizard) {
      newPlayer.mana = this.mana;
    } else {
      newPlayer.damageAmt = this.damageAmt;
    }

    return newPlayer;
  }

  takeTurn(opponent) {
    this.turn++;

    for (let i = 0; i < this.activeSpells.length; i++) {
      let spell = this.activeSpells[i];

      if (spell.duration > 0) {
        spell.effect(this, opponent);
        spell.duration--;

        if (spell.duration === 0 && spell.end) spell.end(this, opponent);
      }
    }
  }

  start() {
    this.hp = this.initial.hp;
    this.spent = 0;
    this.armor = 0;
    this.turn = 0;
    this.activeSpells = [];
    if (this.isWizard) {
      this.mana = this.initial.mana;
    } else {
      this.damageAmt = this.initial.damageAmt;
    }
  }
}

let me = new Player({hp: 50, mana: 500}, true);
let boss = new Player({hp: 55, damageAmt: 8});
let cheapestSpent = Infinity;

function playAllGames(me, boss) {
  for (let i = 0; i < me.spells.length; i++) {
    let spellMatch = false;

    for (let j = 0; j < me.activeSpells.length; j++) {
      if (me.activeSpells[j].duration > 1 && i === me.activeSpells[j].idx) spellMatch = true;
    }

    if (spellMatch) continue;
    if (me.spells[i].cost > me.mana) continue;

    let newMe = me.duplicate();
    let newBoss = boss.duplicate();

    newMe.hp--;
    newMe.takeTurn(newBoss);
    newBoss.takeTurn(newMe);
    newMe.attack(newBoss, i);

    newMe.takeTurn(newBoss);
    newBoss.takeTurn(newMe);
    newBoss.attack(newMe);

    if (newBoss.hp <= 0) cheapestSpent = Math.min(cheapestSpent, newMe.spent);
    if (newMe.hp > 1 && newBoss.hp > 0 && newMe.spent < cheapestSpent) playAllGames(newMe, new
  }
}

playAllGames(me, boss);
```

```

console.log(cheapestSpent);
```
Day 22 Part 2

Day 23 — Part 1 (Opening the Turing Lock)

Problem definition: link

Yeah! Low-level stuff, kind of. I was raised on Assembler and low-level stuff (thanks to my father).

We need to simulate a processor and macro assembler to determine what we need to do with input command.

Firstly, let's write simple regular expressions to parse input commands.

Secondly, a processor which has two registers (for our case).

Thirdly, a macro assembler which has an instruction like *hlf* or *inc* and assigned function to calculate this instruction.

We have all we need to simulate interpretator for our language. Having source code, which is your input, we can parse this source code from text and return object with *instruction*, *register* and *offset* properties.

The simplest part now is *while* loop. While our pointer points to existing instruction in our source code — we need to execute it. Parse this instruction and apply it to our macro assembler, storing the result in processor.

Answer to our problem will be value in register *b* from our processor.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const SIMPLE_INSTRUCTION = /(hlf|tpl|inc) (\w+)/;
const SIMPLE_JUMP_INSTRUCTION = /(jmp) ([+-]\d+)/;
const CONDITIONAL_JUMP_INSTRUCTION = /(jie|jio) (\w+), ([+-]\d+)/;

const PROCESSOR = new Map([
  ['a', 0],
  ['b', 0]
]);

const MACRO_ASSEMBLER = {
  hlf: (_, value) => value / 2,
  tpl: (_, value) => value * 3,
  inc: (_, value)=> value + 1,
  jmp: offset => +offset,
  jie: (offset, register) => register % 2 === 0 ? +offset : 1,
  jio: (offset, register) => register === 1 ? +offset : 1
};

const parseInstruction = instruction => {
  let parsed;

  if (SIMPLE_INSTRUCTION.test(instruction)) parsed = instruction.match(SIMPLE_INSTRUCTION);
  if (SIMPLE_JUMP_INSTRUCTION.test(instruction)) parsed = instruction.match(SIMPLE_JUMP_INSTRUC
  if (CONDITIONAL_JUMP_INSTRUCTION.test(instruction)) parsed = instruction.match(CONDITIONAL_JU

  return {
    instruction: parsed[1],
    register: isNaN(parseInt(parsed[2])) ? parsed[2] : null,
    offset: typeof parsed[3] === 'undefined' && parsed[1] === 'jmp' ? parsed[2] : parsed[3]
  }
};

let pointer = 0;
while (INPUT[pointer]) {
  const instruction = INPUT[pointer];
  const parsed = parseInstruction(instruction);

  if (['jmp', 'jie', 'jio'].indexOf(parsed.instruction) !== -1) {
    pointer += MACRO_ASSEMBLER[parsed.instruction](parsed.offset, PROCESSOR.get(parsed.register
  } else {
    PROCESSOR.set(parsed.register, MACRO_ASSEMBLER[parsed.instruction](parsed.offset, PROCESSOR
    pointer++;
  }
}

const result = PROCESSOR.get('b');
const result = PROCESSOR.get('b');

console.log(result);
```

aoc-23-1.js hosted with ❤ by GitHub view raw

Day 23 Part 1

Day 23 — Part 2 (Opening the Turing Lock)

Problem definition: link

The same task with different starting values in our processor.

> *What is the value in register b after the program is finished executing if register a starts as 1 instead?*

Update our processor declaration at line 8 and run the solution.

```
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n');
const SIMPLE_INSTRUCTION = /(hlf|tpl|inc) (\w+)/;
const SIMPLE_JUMP_INSTRUCTION = /(jmp) ([+-]\d+)/;
const CONDITIONAL_JUMP_INSTRUCTION = /(jie|jio) (\w+), ([+-]\d+)/;

const PROCESSOR = new Map([
  ['a', 1],
  ['b', 0]
]);

const MACRO_ASSEMBLER = {
  hlf: (_, value) => value / 2,
  tpl: (_, value) => value * 3,
  inc: (_, value)=> value + 1,
  jmp: offset => +offset,
  jie: (offset, register) => register % 2 === 0 ? +offset : 1,
  jio: (offset, register) => register === 1 ? +offset : 1
};

const parseInstruction = instruction => {
  let parsed;

  if (SIMPLE_INSTRUCTION.test(instruction)) parsed = instruction.match(SIMPLE_INSTRUCTION);
  if (SIMPLE_JUMP_INSTRUCTION.test(instruction)) parsed = instruction.match(SIMPLE_JUMP_INSTRUC
  if (CONDITIONAL_JUMP_INSTRUCTION.test(instruction)) parsed = instruction.match(CONDITIONAL_JU

  return {
    instruction: parsed[1],
    register: isNaN(parseInt(parsed[2])) ? parsed[2] : null,
    offset: typeof parsed[3] === 'undefined' && parsed[1] === 'jmp' ? parsed[2] : parsed[3]
  }
};

let pointer = 0;
while (INPUT[pointer]) {
  const instruction = INPUT[pointer];
  const parsed = parseInstruction(instruction);

  if (['jmp', 'jie', 'jio'].indexOf(parsed.instruction) !== -1) {
    pointer += MACRO_ASSEMBLER[parsed.instruction](parsed.offset, PROCESSOR.get(parsed.register
  } else {
    PROCESSOR.set(parsed.register, MACRO_ASSEMBLER[parsed.instruction](parsed.offset, PROCESSOR
    pointer++;
  }
}

const result = PROCESSOR.get('b');
const result = PROCESSOR.get('b');

console.log(result);
```

aoc-23-2 is hosted with ❤ by GitHub view raw

Day 23 Part 2

Day 24 — Part 1 (It Hangs in the Balance)

Problem definition: link

We have a few hints right in the problem definition:

> *The packages need to be split into three groups of exactly the same weight*
>
> *The one going in the passenger compartment — needs as few packages as possible so that Santa has some legroom left over.*

and

> *It doesn't matter how many packages are in either of the other two groups, so long as all of the groups weigh the same.*

Let's start with finding the weight (we have 3 groups in this case). Reduce the input array, finding the total sum and divide this sum by 3.

Afterwards, while we don't have the valid packages (with equal weight) iterate all possible combinations of packages and if weight of each package for current combination is equal — push to valid packages array.

Solution to this problem is quantum entanglement which can be calculated as following:

> *The quantum entanglement of a group of packages is the product of their weights, that is, the value you get when you multiply their weights together.*

Map the valid packages and calculate the quantum entanglement of this packages. Find the minimum quantum entanglement which is our result.

```javascript
const Combinatorics = require('./combinatorics');
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n').map(Number);
const TOTAL_SUM = INPUT.reduce((total, x) => total + x, 0);
const GROUP_WEIGHT = TOTAL_SUM / 3;
const VALID_PACKAGES = [];

for (var i = 1; VALID_PACKAGES.length === 0; i++) {
  var combination = Combinatorics.combination(INPUT, ++i);
  var cmb;

  while (cmb = combination.next()) {
    if (cmb.reduce((acc, x) => acc + x) === GROUP_WEIGHT) VALID_PACKAGES.push(cmb);
  }
}

const result = VALID_PACKAGES.map(pkg => pkg.reduce((acc, x) => acc * x)).sort((a, b) => a - b)

console.log(result);
```

Day 24 Part 1

Day 24 — Part 2 (It Hangs in the Balance)

Problem definition: link

Solution remains the same except the weight of each group because:

> *"Ho ho ho", Santa muses to himself. "I forgot the trunk".*

Divide total sum of each package weight by 4 and find the minimum quantum entanglement for this case.

```
const Combinatorics = require('./combinatorics');
const fs = require('fs');
const INPUT = fs.readFileSync('./input.txt', 'utf-8').split('\n').map(Number);
const TOTAL_SUM = INPUT.reduce((total, x) => total + x, 0);
const GROUP_WEIGHT = TOTAL_SUM / 4;
const VALID_PACKAGES = [];

for (var i = 1; VALID_PACKAGES.length === 0; i++) {
  var combination = Combinatorics.combination(INPUT, ++i);
  var cmb;

  while (cmb = combination.next()) {
    if (cmb.reduce((acc, x) => acc + x) === GROUP_WEIGHT) VALID_PACKAGES.push(cmb);
  }
}

const result = VALID_PACKAGES.map(pkg => pkg.reduce((acc, x) => acc * x)).sort((a, b) => a - b)

console.log(result);
```

Day 24 Part 2

Day 25 (Let It Snow)

Problem definition: link

Finally, we are on the top of Christmas tree. I afraid that the last problem will be mega-super hard to solve but it's not — just simple math.

Here some quotes from problem definition which are important:

> *The codes are printed on an infinite sheet of paper, starting in the top-left corner. The codes are filled in by diagonals: starting with the first row with an empty first box, the codes are filled in diagonally up and to the right.*
>
> *So, to find the second code (which ends up in row 2, column 1), start with the previous value, 20151125. Multiply it by 252533 to get 5088824049625. Then, divide that by 33554393, which leaves a remainder of 31916031. That remainder is the second code.*

Here, we have a simple formula to calculate the next code — (*previous code * 252533) % 33554393*.

All need to do is to determine index of our target [row, column] and iterate calculating the result by formula above.

```
const ROW = 3010;
const COLUMN = 3019;
const FIRST_CODE = 20151125;
const TARGET_INDEX = ((Math.pow(ROW + COLUMN - 1, 2) + ROW + COLUMN - 1) / 2) - ((ROW + COLUMN

let result = FIRST_CODE;
for (var i = 1; i < TARGET_INDEX; i++) {
  result = (result * 252533) % 33554393;
}

console.log(result);
```

aoc-25.js hosted with ❤ by GitHub view raw

Day 25

It was a new experience for me in solving these problems and I highly recommend to play this game on your own.

Thanks Konstantin Batura for helping me write this article.

Eugene Obrezkov, Developer Advocate at Onix-Systems, Kirovohrad, Ukraine.

JavaScript Nodejs Puzzle

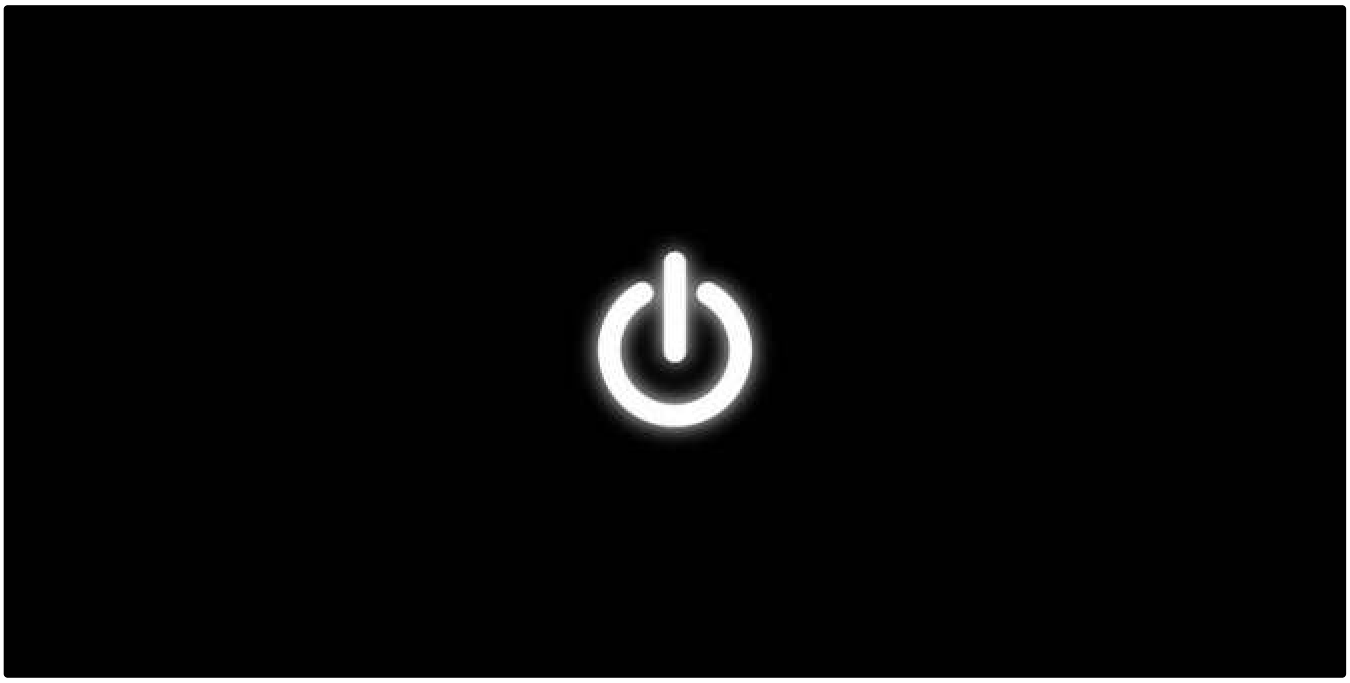
Follow

Written by Eugene Obrezkov

772 Followers

Software Engineer · elastic.io · JavaScript · DevOps · Developer Tools · SDKs · Compilers · Operating Systems

More from Eugene Obrezkov



Eugene Obrezkov in Eugene Obrezkov

How to implement your own "Hello, World!" boot loader

using Assembly language on bare-metal machine

5 min read · Oct 21, 2017

495 5

Eugene Obrezkov in Eugene Obrezkov

How NodeJS requires native shared objects

3 min read · Feb 8, 2017

22 1

Eugene Obrezkov in Eugene Obrezkov

Faster logs delivering from Fluentd

How to tune your fluentd configuration to send logs more frequently

3 min read · Jun 13, 2018

29

Eugene Obrezkov in Eugene Obrezkov

How do we build our platform from mono-repository on CircleCI

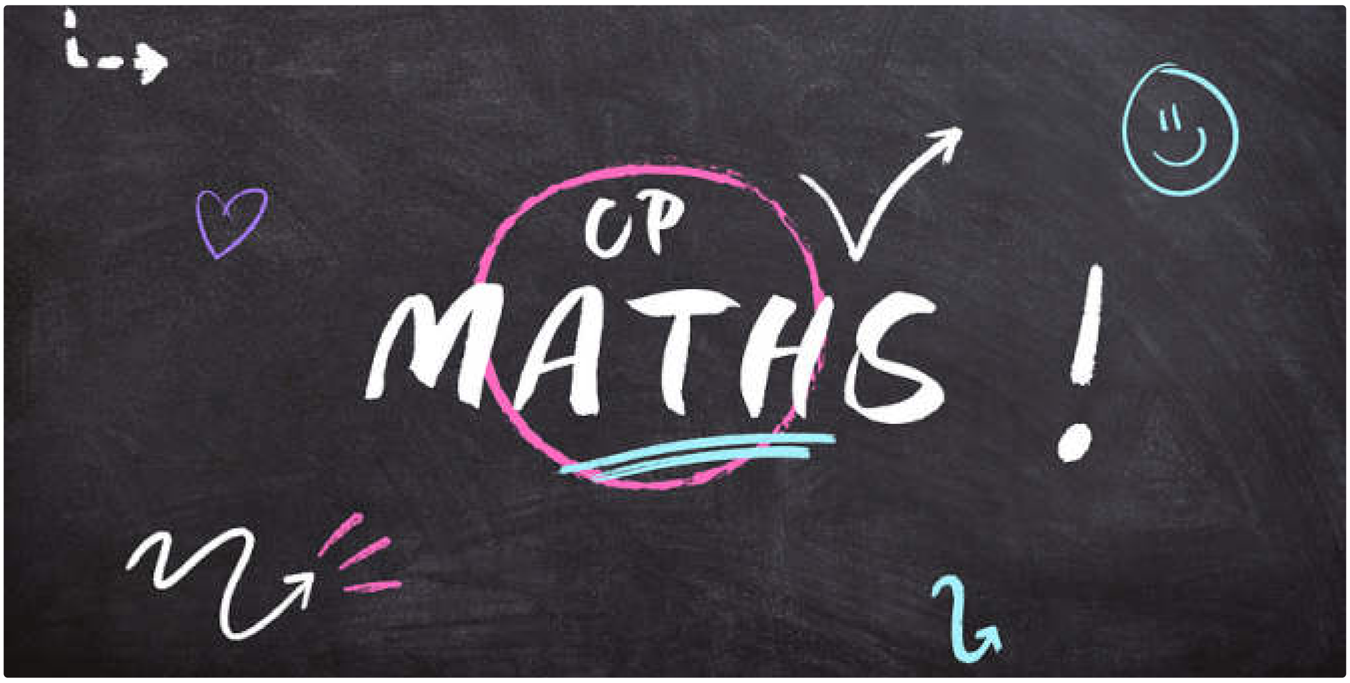
In this topic, I will show what exactly our build process looks like on CircleCI

5 min read · Aug 5, 2018

57 1

See all from Eugene Obrezkov

Recommended from Medium



Yakub

The Essential Mathematics for Competitive Coding

Competitive coding, also known as competitive programming, is a challenging and rewarding activity where programmers solve algorithmic and...

5 min read · Jul 21, 2023

60

Arslan Ahmad in Level Up Coding

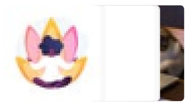
Don’t Just LeetCode; Follow the Coding Patterns Instead

What if you don’t like to practice 100s of coding questions before the interview?

5 min read · Sep 15, 2022

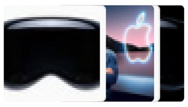
7.5K 44

Lists



Stories to Help You Grow as a Software Developer

19 stories · 699 saves



Apple's Vision Pro

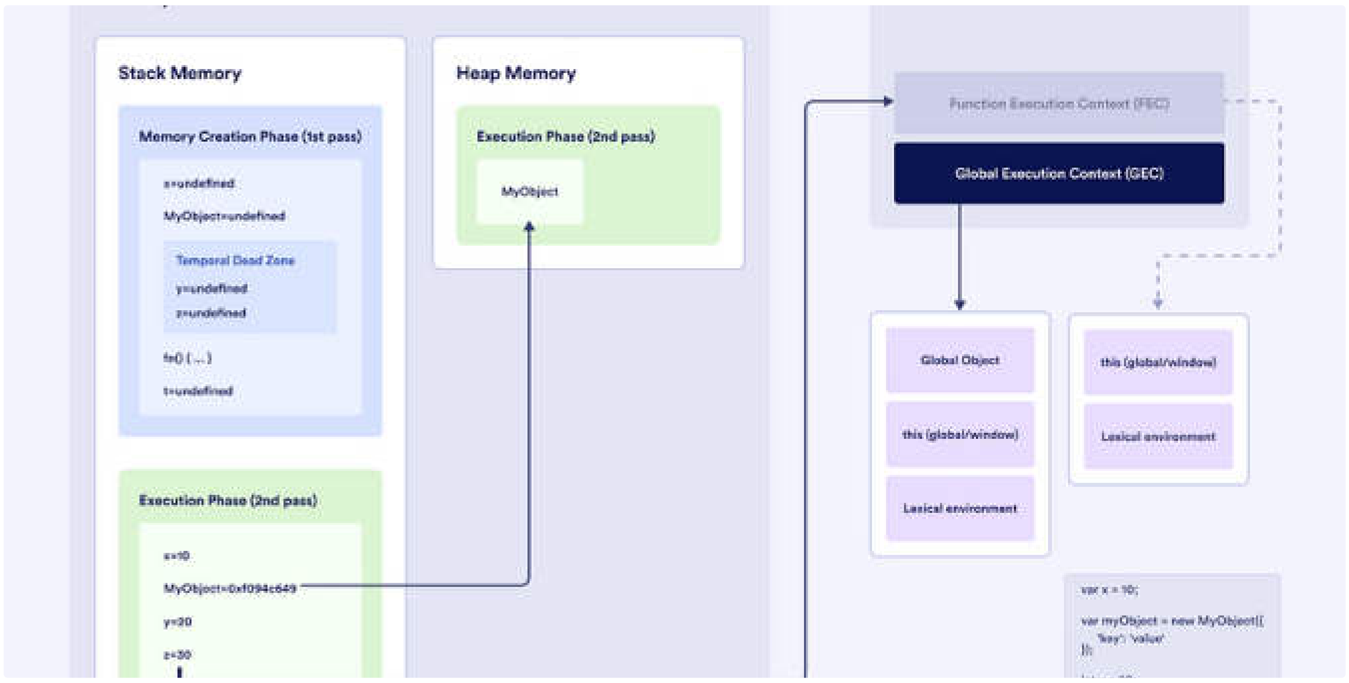
7 stories · 40 saves

General Coding Knowledge

20 stories · 773 saves

data science and AI

39 stories · 36 saves



```
var x = 10;
var myObject = new MyObject({
    'key': 'value'
});
```

Nalan Ozgedik in Jotform Tech

Unraveling the JavaScript execution pipeline: V8, event loop, and libuv

Unraveling the JavaScript execution pipeline: Understanding V8, event loop, and libuv for high-performance web experiences

11 min read · Aug 9, 2023

20

Rohit Verma

My Interview Experience at Google [L5 Offer]

Comprehensive Insights: A Deep Dive into the Journey from Preparation Through Interviews to Securing the Offer.

9 min read · Nov 25, 2023

5.5K 35

Benoit Ruiz in Better Programming

Advice From a Software Engineer With 8 Years of Experience

Practical tips for those who want to advance in their careers

22 min read · Mar 20, 2023

13.3K 248

Rabail Zaheer

IIFE Explained: Immediately Invoked Function Expressions

In the world of JavaScript, where functions are a fundamental building block, there exists a powerful concept known as IIFE or Immediately...

8 min read · Sep 27, 2023

4

See more recommendations