CodeIQ

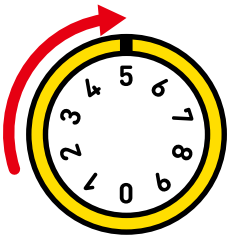
もっと

プログラマ脳を鍛える 数学パズル

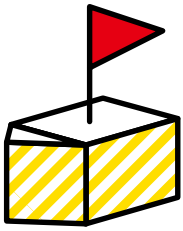
アルゴリズムが脳にしみ込む70問

増井敏克

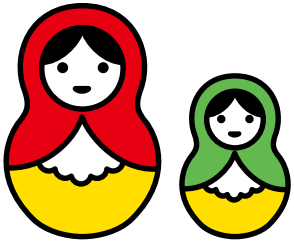
あなたのコーディングスキルで解けるか？



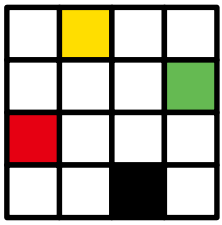
Q37 ダイヤルロックを解除せよ！



Q40 沈みゆく島で出会う船

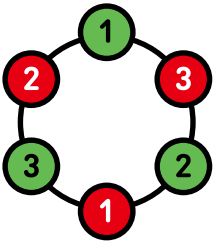


Q52 一列に並べたマトリョーシカ



Q65 *n*-Queenで反転

Q54 素数で作る天秤ばかり



Q43 隣り合えないカップル

お試し版

- Ruby／JavaScript完全対応
- 「パズル問題を解くコツ」も掲載

SE SHOEISHA

Q01 一発で決まる多数決

IQ: 70　目標時間: 20分

意見が対立したときに使われる多数決。単純でわかりやすいため、政治の世界だけでなく、学校や会社でも多く使われています。ここでは、じゃんけんで出す手を使った多数決を考えます。

それぞれが出せる手は「グー」「チョキ」「パー」のいずれかです。このとき、一番多くの人が出した手が勝つことになります。たとえば、6人で行う場合、表1のように1回で勝者がいずれかに決まる場合もありますが、表2のようにいずれにも決まらない場合もあります。

ある人数でじゃんけんをするとき、「一度で」勝つ手が決まるような人数の組み合わせが何通りあるかを求めます。たとえば、4人の場合は表3のようなパターンがあるので、全部で12通りです。

問題

100人の場合、一度で勝つ手が決まるような人数の組み合わせが何通りあるかを求めてください。

表1 決まる場合

グー	チョキ	パー	結果
3人	2人	1人	グーの勝ち
1人	4人	1人	チョキの勝ち

表2 決まらない場合

グー	チョキ	パー	結果
2人	2人	2人	すべて同数のため決まらず
3人	0人	3人	グーとパーが同数のため決まらず

表3 4人の場合は12通り

グー	チョキ	パー	結果
0人	0人	4人	パーの勝ち
0人	1人	3人	パーの勝ち
0人	2人	2人	決まらず
0人	3人	1人	チョキの勝ち
0人	4人	0人	チョキの勝ち
1人	0人	3人	パーの勝ち
1人	1人	2人	パーの勝ち
1人	2人	1人	チョキの勝ち
1人	3人	0人	チョキの勝ち
2人	0人	2人	決まらず
2人	1人	1人	グーの勝ち
2人	2人	0人	決まらず
3人	0人	1人	グーの勝ち
3人	1人	0人	グーの勝ち
4人	0人	0人	グーの勝ち

考え方

出せる手はグー、チョキ、パーの3通りなので、それぞれの手を出した人数を数えてみます。一度で決まるのは、その人数が最多となる手が1通りに決まる場合だといえます。

それぞれが3通りなので、100人だと3^{100}通りを調べるんですか？
そんなの計算できそうにないですね……

それぞれの手を出した人数だけを考えればいいんじゃないかしら？ いずれか2つの手を出した人数を決めると、残りの人数が決まるね。

それぞれの人数を変えながら、最大の人数になる手が1つになるものを探してみました。

q01_1.rb

```
N = 100

cnt = 0
0.upto(N) do |rock|                # グーの人数
  0.upto(N - rock) do |scissors|   # チョキの人数
    paper = N - rock - scissors    # パーの人数
    all = [rock, scissors, paper]
    cnt += 1 if all.count(all.max) == 1
  end
end
puts cnt
```

q01_1.js

```
N = 100;

var cnt = 0;
for (var rock = 0; rock <= N; rock++){ // グーの人数
  for (var scissors = 0; scissors <= N - rock; scissors++){
    // チョキの人数
    var paper = N - rock - scissors; // パーの人数
    if (rock > scissors){
      if (rock != paper)
        cnt++;
    } else if (rock < scissors) {
      if (scissors != paper)
        cnt++;
    } else {
      if (rock < paper)
```

```
            cnt++;
        }
    }
}
console.log(cnt);
```

Rubyでのif文に指定されている条件はどういう意味かな？

グー、チョキ、パーそれぞれの人数の最大値が「1つだけ」、つまり最大の人数になっている手が1つに決まることを表していマス。

JavaScriptのソースコードでは、最初にグーとチョキを比べて、その大きいほうとパーの数を比べているわ。こうすると、最大の人数になっている手がいくつあるかを求められるんだね。

処理速度は変わりませんが、以下のように区切る方法もありますよ。

Point

たとえば4人の場合、グー・チョキ・パーの間に区切り記号を入れることで、各パターンを作り出す、と考えることもできます。「○」を人間とし、1つ目の「｜」までがグー、2つ目の「｜」までがチョキ、それ以降はパーと考えると、「○」を4つ、「｜」を2つ並べるときの組み合わせ数を求める問題だと考えられます（図2）。

このような手法は、組み合わせを調べるときによく使われるので、考え方を学んでおきましょう。

○○○○｜｜	○○○｜○｜	○○○｜｜○	○○｜○○｜
○○｜○｜○	○○｜｜○○	○｜○○○｜	○｜○○｜○
○｜○｜○○	○｜｜○○○	｜○○○○｜	｜○○○｜○
｜○○｜○○	｜○｜○○○	｜｜○○○○	

図2 区切り記号でパターンを考える

ここでは、左の区切り記号を入れる場所を「l(left)」、右の区切り記号を入れる場所を「r(right)」で表現しています。

q01_2.rb

```ruby
N = 100

cnt = 0
0.upto(N) do |l|    # 左の区切り位置
  l.upto(N) do |r|  # 右の区切り位置
    all = [l, r - l, N - r] # グー、チョキ、パーそれぞれの人数
    cnt += 1 if all.count(all.max) == 1
  end
end
puts cnt
```

q01_2.js

```javascript
N = 100;

var cnt = 0;
for (l = 0; l <= N; l++){   // 左の区切り位置
  for (r = l; r <= N; r++){ // 右の区切り位置
    if (l > r - l){         // グーがチョキより多いとき
      if (l != N - r)       // グーがパーと異なるとき
        cnt++;
    } else if (l < r - l){  // チョキがグーより多いとき
      if (r - l != N - r)   // チョキがパーと異なるとき
        cnt++;
    } else {                // グーとチョキが同じとき
      if (l < N - r)        // パーが最大のとき
        cnt++;
    }
  }
}
console.log(cnt);
```

ソースコードはあまり変わらないような……

これは「重複組み合わせ」の考え方ですね。「n種類から重複を許してr個選ぶ方法」と「r個の○と$n-1$個の仕切りを一列に並べる方法」は1対1に対応する、というのは有名なので、覚えておきましょう。

解答 **5,100通り**

Q02 山手線でスタンプラリー

IQ: 70　目標時間: 15分

山手線のスタンプラリーを考えましょう。すべての駅にスタンプを設置し、利用者は最初の駅と最後の駅で必ずスタンプを押すものとします。スタンプは改札内に設置し、一度入場すれば改札を出ずにスタンプを集めることができます。

山手線は東京都内にある環状の鉄道路線で、全部で29個の駅があります。ここでは、山手線と共通の駅を持つほかの路線は使えないものとします。

さらに、ここでは山手線を一方通行で進むとします。利用者は片道の切符を購入し、ある駅から別の駅に向かいますが、一度通った駅を再度通ることはできません（途中で逆行すると違反になります）。

スタンプカードはすべての駅のスタンプを押すことができ、山手線の各駅には1～29の番号が順に付与されているとします。

問題

1番の駅から入場し、17番の駅から出場するとき、スタンプを押す順番として考えられるパターンが何通りあるかを求めてください（図3）。

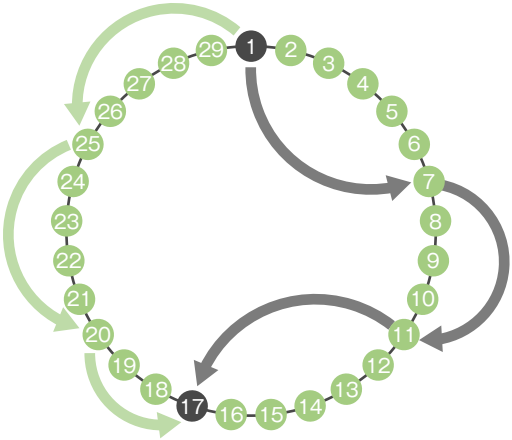


図3 問題のイメージ

途中の駅でスタンプを押さない場合は、内回りでも外回りでも押されるスタンプは同じ（2つのみ）ですね。

考え方

問題を簡単にするため、最初は循環している配置ではなく、一列に並んだ駅を考えてみます。逆行できないため、それぞれの駅で降りるかどうかを調べると、そのパターン数を求められます。

たとえば、1、2、3、4、5という5つの駅があった場合、図4の8通りがあります。

1→2→3→4→5	1→2→3→5	1→2→4→5	1→3→4→5
1→2→5	1→3→5	1→4→5	1→5

図4 駅が5つの場合

最初の駅と最後の駅は必ず止まるので、「2」に止まるかどうか、「3」に止まるかどうか、「4」に止まるかどうか、と考えられ、2×2×2で求められます。つまり、「間にある駅の数」をnとすると2^n通りとなります。

間にある駅の数は「出場した駅の番号」と「入場した駅の番号」を比べることで求められるわ。

一般的な場合を考えると、「入場した駅の番号」のほうが大きい可能性があるので、絶対値を使うと簡単に求められマス。

内回りと外回りはどう考えたらいいんだろう？

ヒントにもあるように、途中の駅でスタンプを押さない場合は、スタンプを押す順番が同じです。この1通りは重複しますので、除外しましょう。

q02.rb

```
N = 29

# 入場と出場の駅番号をセット
a, b = 1, 17

#「間にある駅の数」を求める
n = (a - b).abs

# 内回りと外回りを足して、重複を除外
puts (1 << (n - 1)) + (1 << (N - n - 1)) - 1
```

q02.js

```
N = 29;

var a = 1;
var b = 17;
var n = Math.abs(a - b);

console.log((1 << (n - 1)) + (1 << (N - n - 1)) - 1);
```


Point

ここでは2^nを求めるのに「<<」というシフト演算子を使用しています。「<<」は「左シフト」と呼ばれ、「1 << 3」であれば2進数の「1」を左に3ビットシフトします（図5）。

10進数	2進数
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100

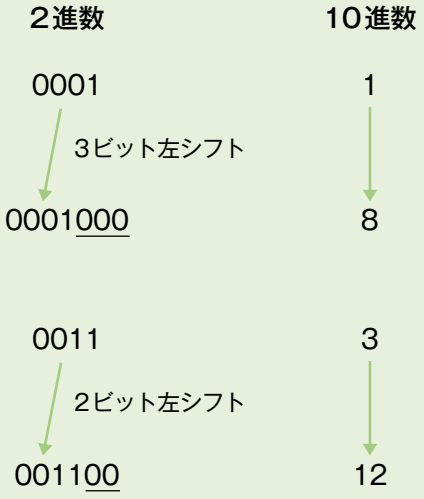


図5 左シフト

1ビット左にシフトするごとに2倍、右にシフトするごとに2分の1になるので、2のn乗を計算するには1をn回左シフトすると求められるわけです。

aのb乗を計算する場合、Rubyの場合は「a ** b」、JavaScriptの場合は「Math.pow(a, b)」と書くこともできますが、底が2、つまり2の累乗であればシフト演算を使うほうが高速に処理できます。

※JavaScriptでもES2016よりa ** bという書き方ができるようになりました。

解答 **36,863 通り**

先生のコラム

実務にも使えるビット演算

この問題で登場したシフト演算以外にも、ANDやOR、XORなどのビット演算（論理演算）もよく使われます。ビット演算を使うと、ソースコードがシンプルになり高速に処理できるだけでなく、「1つのデータに複数の情報を入れられる」といったメリットがあります。

マニアックな印象を持たれやすいビット演算ですが、この特徴を活かして実務で使われることも珍しくありません。たとえば、Windowsのアプリケーションを開発する場合、ファイルやフォルダの属性情報の判定によく使われます。

保存したファイルには「読み取り専用」や「隠しファイル」など、様々な属性が付与されています。これを管理するのが「FileAttributes」という列挙体です。それぞれの属性に対し、1、2、4、8、16、…という2の累乗を使った定数を列挙することで定義しています。実際には表4のような属性があり、それぞれのビットに値が割り当てられています。

表4 FileAttributesの属性

メンバー名	説明	実際の値
ReadOnly	読み取り専用	1
Hidden	隠しファイル	2
System	システムファイル	4
Directory	ディレクトリ	16
Archive	アーカイブ	32
…	…	…

たとえば、読み取り専用のビットが立っているかを確認する場合は、以下のように「AND演算」を使って判定できます。

```
attr = File.GetAttributes("ファイル名")
if (attr & FileAttributes.ReadOnly) == FileAttributes.ReadOnly
```

Q03 ローマ数字の変換規則

IQ: 80　目標時間: 20分

時計の文字盤などで人気があるローマ数字。海外に行くと、歴史的建造物など、いたるところで目にします。しかし、その“変換規則”を知らないと、その数字がいくつを表しているのかわかりません。

そこでローマ数字について考えてみます。ローマ数字では表5にある記号が使われます。

表5 アラビア数字とローマ数字の対応表

アラビア数字	1	5	10	50	100	500	1,000
ローマ数字	I	V	X	L	C	D	M

この表にない数は、足し算して目的の数になるような表現の中から、できるだけ使う文字数が少ないものを選び、数字が大きい順に左から並べて書きます。たとえば、27であれば10＋10＋5＋1＋1で表現できるので、「XXVII」と書きます。

ただし、「同じ文字を4つ以上連続で並べることはできない」というルールもあります。例を挙げると、4 を「IIII」、9を「VIIII」とは書けないので、引き算を使って小さい数を大きい数の左に書きます。4なら「IV」、9なら「IX」と書きます。

なお、使える記号は上記のように「M（1,000）」までのため、ローマ数字では最大で3,999まで表現できます。

問題

ローマ数字の記号を12個並べたとき、ローマ数字として認識できる数が何通りあるかを求めてください。

たとえば、1個の記号で表現できるのは、I、V、X、L、C、D、Mの7通り、15個の記号で表現できるのは「MMMDCCCLXXXVIII」（3,888）の1通りです。

1から3,999までの数をそれぞれローマ数字で表現してみて、使う記号の数を数えれば求められます。

考え方

ヒントにあったように、それぞれの数をローマ数字に変換できれば、あとはその文字数を数えるだけです。そこで、アラビア数字からローマ数字にどのように変換するかを考えてみます。

まずは基本となる表（表5）から規則性を考えてみましょう。

規則性……。それぞれの最上位の桁が1、5、1、5、の繰り返しになっていますよね……それくらいしかわかりません。

1、5、1、5の繰り返しということは、桁が増えるタイミングが1、10、100、1,000の部分だけじゃない、ということね。

数字を読み取る場合、私たちは桁で考えます。一、十、百、千といった区切りだけでなく、日本では万を超えるとこれらを組み合わせて「千三百万」といった呼び方も使います。英語ではten thousandのように1,000の位で区切ります。

ローマ数字への変換の場合も、同じように桁で区切って考えてみます。まずは1、10、100、1,000という桁で区切るため、それぞれ割り算をして、その「余り」を求めます。この余りが4、9、40、90、…のようになる場合は無条件にローマ数字が決まります。

一方、余りがその他の場合は、さらに5、50、500で割った余りを考えてみます。表6のように整理すると、同じ色のところではローマ数字の増え方が同じようになっており、規則性があることがわかります。

表6 余りの規則性

アラビア数字	1	2	3	5	6	7	8
ローマ数字	I	II	III	V	VI	VII	VIII
アラビア数字	110	120	130	150	160	170	180
ローマ数字	CX	CXX	CXXX	CL	CLX	CLXX	CLXXX

この表をもとにプログラムを作ってみると、以下のように実装できます。

q03.rb

```
N = 12

# 1桁分の変換
```

```
def conv(n, i, v, x)
  result = ''
  if n == 9
    result += i + x
  elsif n == 4
    result += i + v
  else
    result += v * (n / 5)
    n = n % 5
    result += i * n
  end
  result
end

# ローマ数字への変換
def roman(n)
  m, n = n.divmod(1000)
  c, n = n.divmod(100)
  x, n = n.divmod(10)
  result = 'M' * m
  result += conv(c, 'C', 'D', 'M')
  result += conv(x, 'X', 'L', 'C')
  result += conv(n, 'I', 'V', 'X')
  result
end

cnt = Hash.new(0)
1.upto(3999){|n|
  cnt[roman(n).size] += 1
}
puts cnt[N]
```

q03.js

```
N = 12;

// 1桁分の変換
function conv(n, i, v, x){
  var result = '';
  if (n == 9)
    result += i + x;
  else if (n == 4)
    result += i + v;
  else {
    for (j = 0; j < Math.floor(n / 5); j++)
      result += v;
    n = n % 5;
    for (j = 0; j < n; j++)
      result += i;
  }
```

```
  return result;
}

// ローマ数字への変換
function roman(n){
  var m = Math.floor(n / 1000);
  n %= 1000;
  var c = Math.floor(n / 100);
  n %= 100;
  var x = Math.floor(n / 10);
  n %= 10;
  var result = 'M'.repeat(m);
  result += conv(c, 'C', 'D', 'M');
  result += conv(x, 'X', 'L', 'C');
  result += conv(n, 'I', 'V', 'X');
  return result;
}

var cnt = {};
for (i = 1; i < 4000; i++){
  var len = roman(i).length;
  if (cnt[len]){
    cnt[len] += 1;
  } else {
    cnt[len] = 1;
  }
}
console.log(cnt[N]);
```

10、100、1,000で割った商と余りを使って変換する処理を、共通処理として抜き出しているのね。これならシンプルになるわ。

共通の処理をまとめてもらえると、同じ処理を実行するだけなのでコンピュータは助かりマス。

結果の集計にハッシュ（連想配列）を使っているのもコツです。ただの配列で1～3,999まで確保してもよいですが、今回の場合は文字数だけわかれば十分ですね。

解答 **93通り**

Q04 点灯している量で考えるデジタル時計

IQ: 70　目標時間: 15分

図6のような7セグメントディスプレイを使ったデジタル時計があります。この時計では、時刻によって点灯している位置の数が決まります。たとえば、12:34:56（12時34分56秒）の場合は、右図のように27カ所が点灯しています。

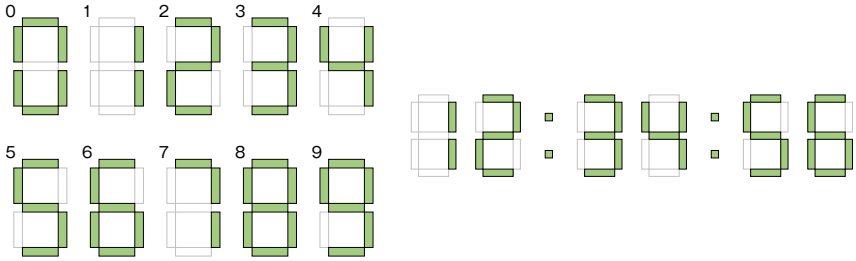


図6 7セグメントディスプレイの点灯位置

ここでは逆に考えて、点灯している箇所の数から時刻を調べてみましょう。ただし、時刻としてありえない数字の並びは対象にはなりません（「53:61:24」も27カ所が点灯しますが対象外）。

なお、このデジタル時計は24時間表示で、23時59分59秒まで表示します。また、1桁になる場合は、時・分・秒のいずれも0埋めして表示されるものとします。

問題

30カ所が点灯するような時刻が何通りあるかを求めてください。

27カ所が点灯するような時刻は、「12:34:56」を含めて8,800通りあります。

点灯している位置をどんなデータ構造で表現すればいいんだろう？

この問題を解くうえで必要なのは、点灯している位置ではなく個数だね。

考え方

デジタル時計において、点灯している箇所を数えるにはいくつかの方法が考えられます。点灯可能な箇所は7カ所×6文字あるので、全部で42通りです。その中から30カ所が点灯しているものを調べようとすると、42個から30個を選ぶ組み合わせになってしまい、膨大な数になってしまいます。

しかし、表示されるのは時刻になる並びだけですので、調べるパターンをすべての時刻だけに絞り込み、その点灯箇所を数えれば求められます。

点灯しているかどうかをすべての位置について調べるのではなくて、時刻を前提としてプログラムを作ればいいのか！……でもどうやるんだろ。

時刻の組み合わせは24×60×60。つまり86,400通りを調べれば済むね。

それぞれの数字で点灯される量は事前に求めておくと楽ですね。

時刻として可能なものを調べると、以下のようなプログラムを作成できます。

q04_1.rb

```
N = 30

# 2桁の数字に対する点灯数を返す
def check(num)
  light = [6, 2, 5, 5, 4, 5, 6, 3, 7, 6]
  light[num / 10] + light[num % 10]
end

cnt = 0
24.times do |h|
  60.times do |m|
    60.times do |s|
      cnt += 1 if check(h) + check(m) + check(s) == N
    end
  end
end
puts cnt
```

q04_1.js

```
N = 30;
```

```
// 2桁の数字に対する点灯数を返す
function check(num){
  var light = [6, 2, 5, 5, 4, 5, 6, 3, 7, 6];
  return light[Math.floor(num / 10)] + light[num % 10];
}

var cnt = 0;
for (var h = 0; h < 24; h++){
  for (var m = 0; m < 60; m++){
    for (var s = 0; s < 60; s++){
      if (check(h) + check(m) + check(s) == N)
        cnt++;
    }
  }
}
console.log(cnt);
```

なるほど。事前に、数字に対応する点灯数を配列にセットしているんですね。

10の位と1の位に分けて、その合計を求めているよ。

もう少し調べる量を減らせませんか？ たとえば、「12」という数字は時・分・秒のいずれでも調べていますし、それをループの中で何度も調べていますね？

同じ数字を何度も調べることを防ぐためには、一度だけ調べて記憶させておく方法が有効です。ここでは、0 ～ 59までの数字について、事前にこの考え方を使って配列に格納しています。

q04_2.rb

```
N = 30

# 2桁の数字に対する点灯数を返す
def check(num)
  light = [6, 2, 5, 5, 4, 5, 6, 3, 7, 6]

  light[num / 10] + light[num % 10]
end

lights = Array.new(60)
60.times do |i|
```

```
  lights[i] = check(i)
end

cnt = 0
24.times do |h|
  60.times do |m|
    60.times do |s|
      cnt += 1 if lights[h] + lights[m] + lights[s] == N
    end
  end
end
puts cnt
```

q04_2.js

```
N = 30;

// 2桁の数字に対する点灯数を返す
function check(num){
  var light = [6, 2, 5, 5, 4, 5, 6, 3, 7, 6];
  return light[Math.floor(num / 10)] + light[num % 10];
}

var lights = new Array(60);
for (var i = 0; i < 60; i++){
  lights[i] = check(i);
}

var cnt = 0;
for (var h = 0; h < 24; h++){
  for (var m = 0; m < 60; m++){
    for (var s = 0; s < 60; s++){
      if (lights[h] + lights[m] + lights[s] == N)
        cnt++;
    }
  }
}
console.log(cnt);
```

Point

関数呼び出しだけでなく、データベースの取得やファイルの読み込みなど、処理に時間がかかる処理を事前に実行しておく考え方は、実務の現場においてもよく利用されます。

解答 **8,360通り**

Q05 枚数で考えるパスカルの三角形

IQ: 70　目標時間: 20分

規則性を学ぶときによく登場する「パスカルの三角形」。各行の左右の端を「1」として、それ以外の箇所は左上と右上の数の和を書くことで作成できます。

今回は、それぞれの値を「金額（日本円）」だと考えることにします。たとえば、「1」は1円、「2」は2円、「10」は10円です。このとき、n段目のそれぞれの値について、紙幣や硬貨での最小の枚数を考え、その枚数の和を求めることにします。

たとえば、$n=4$のとき、1、4、6、4、1の並びでは、1円＝1枚、4円＝4枚、6円＝2枚（5円玉＋1円玉）なので、すべて足すと12枚になります。同様に、$n=9$のとき、すべて足すと48枚になります（図7）。

なお、使える紙幣や硬貨は1円玉、5円玉、10円玉、50円玉、100円玉、500円玉、千円札、2千円札、5千円札、1万円札です（2千円札を忘れないように）。

n=0									1											1枚
n=1								1		1										2枚
n=2							1		2		1									4枚
n=3						1		3		3		1								8枚
n=4					1		4		6		4		1							12枚
n=5				1		5		10		10		5		1						6枚
n=6			1		6		15		20		15		6		1					12枚
n=7		1		7		21		35		35		21		7		1				22枚
n=8	1		8		28		56		70		56		28		8		1			31枚
n=9	1	9		36		84		126		126		84		36		9		1		48枚

図7 パスカルの三角形と計算例

問題

$n=45$のとき、紙幣・硬貨の枚数の和を求めてください。

金額から紙幣や硬貨の枚数を求める際、大きな金額から順番に使っていくと最小の枚数を求められます。

考え方

一気に実装するのではなく、大きく以下の3つのステップに分けて考えてみます。1つ目は、パスカルの三角形を生成するステップです。2つ目は、パスカルの三角形の該当行について、それぞれの値から最小の枚数を算出するステップ。3つ目は、その最小の枚数の和を求めるステップです。

パスカルの三角形は左上と右上の数の和ですよね。どうやって表現すればいいんだろう？

行単位で配列に入れておくと、次の行はその値を足し算すれば求められそう。

順に処理することを考えると、1つの配列でも処理できますね。

図8のように配列を用意して、右側から順に計算していけば、直前の行のデータから、次の行のデータを順にセットできます。

1	0	0	0	0	0	0
1	1	0	0	0	0	0
1	2	1	0	0	0	0
1	3	3	1	0	0	0
1	4	6	4	1	0	0
1	5	10	10	5	1	0
				③	②	①
1	6	15	20	15	6	1

図8 配列を右側から順に計算する

1行目から順に加算して該当の行まで作成したうえで、その行について最小の枚数を算出します。枚数を計算するためには、紙幣や硬貨の金額が大きなほうから順に割り算をすると、その商が枚数になります。

たとえば、178円の場合、100で割ると商が1なので100円が1枚、余りの78を50で割ると商が1なので50円が1枚、余りの28を10で割ると商が2なので10円が2枚、…というように求められます。

これを実装すると、以下のようなプログラムを作成できます。

q05.rb

```ruby
N = 45

def count(n)
  coin = [10000, 5000, 2000, 1000, 500, 100, 50, 10, 5, 1]
  result = 0
  coin.each do |c|
    # 大きな金額から順に、商と余りを計算
    cnt, n = n.divmod(c)
    result += cnt
  end
  result
end

row = [0] * (N + 1);
row[0] = 1;
N.times do |i|
  # 各行に対して右からセット
  (i + 1).downto(1) do|j|
    # 前の行の値に、その左隣の値を足す
    row[j] += row[j - 1]
  end
end

# 枚数の和を計算
puts row.map{|i| count(i)}.inject(:+)
```

q05.js

```javascript
N = 45;

function count(n){
  var coin = [10000, 5000, 2000, 1000, 500, 100, 50, 10, 5, 1];
  var result = 0;
  for (var i = 0; i < coin.length; i++){
    // 大きな金額から順に、商と余りを計算
    var cnt = Math.floor(n / coin[i]);
    n = n % coin[i];
    result += cnt;
  }
  return result;
}

row = new Array(N + 1);
row[0] = 1;
for (var i = 1; i < N + 1; i++){
  row[i] = 0;
}
for (var i = 0; i < N; i++){
  // 各行に対して右からセット
```

```
  for (var j = i + 1; j > 0; j--)
    // 前の行の値に、その左隣の値を足す
    row[j] += row[j - 1];
}

// 枚数の和を計算
var total = 0;
for (var i = 0; i < N + 1; i++){
  total += count(row[i]);
}
console.log(total);
```

Rubyだと、商と余りを同時に計算できるんですね！

枚数の和を計算する最後の処理でも、Rubyでは配列の合計を1行で求めているわ。もちろん、JavaScriptのようにループで処理することもできるね。

解答 **3,518,437,540枚**

数学 うんちく

貪欲法

複数のパターンから最適な選択肢を選ぶ場合、全探索ですべてのパターンを調べることもできるけれど、問題に合わせて作成した基準を使って、簡単に最もよいものを求める方法があるの。このような方法は、「貪欲法」または「グリーディ算法」と呼ばれているわ。

この問題で登場した、「金額から紙幣や硬貨の枚数を求める方法」は貪欲法の典型例。本来なら、最小の枚数を調べるには、すべての組み合わせを洗い出す必要があるよね。でも、紙幣や硬貨の枚数を求めたいのなら、大きな金額から順に使えば求められることは直感でわかるはず。

問題によっては厳密解が得られない場合もあるけれど、単純なプログラムで高速に解を求める方法としてよく使われているよ。

続きは本書にて
お楽しみください

Amazonで予約

他書店で予約

著者プロフィール

増井敏克（ますい としかつ）

増井技術士事務所代表。技術士（情報工学部門）。情報処理技術者試験にも多数合格。IT エンジニアのための実務スキル評価サービス「CodeIQ」にて、アルゴリズムや情報セキュリティに関する問題を多数出題。また、ビジネス数学検定 1 級に合格し、公益財団法人日本数学検定協会認定トレーナーとしても活動。「ビジネス」×「数学」×「IT」を組み合わせ、コンピュータを「正しく」「効率よく」使うためのスキルアップ支援や、各種ソフトウェアの開発、データ分析などを行っている。著書に『おうちで学べるセキュリティのきほん』、『プログラマ脳を鍛える数学パズル』、『エンジニアが生き残るためのテクノロジーの授業』（以上、翔泳社）、『シゴトに役立つデータ分析・統計のトリセツ』、『プログラミング言語図鑑』（以上、ソシム）がある。

装丁・デザイン　植竹 裕
DTP　株式会社 シンクス

もっとプログラマ脳を鍛える数学パズル
アルゴリズムが脳にしみ込む 70 問

2018年2月19日　初版第1刷発行

著者　増井 敏克
発行人　佐々木 幹夫
発行所　株式会社 翔泳社（http://www.shoeisha.co.jp）
印刷・製本　株式会社 ワコープラネット

ISBN978-4-7981-5361-2　Printed in Japan