JRE1 P 演習 難しい問題の解説

つまみ (@TrpFrog)

2020 年 3 月 12 日

個人的に詰まったやつとか、難しいやつだけ取りあげます。他には簡単だけど標準関数を駆使すれば一行で解ける系など。他にやってほしい問題があれば @TrpFrog にリプライを飛ばしてください。

問題の出典: https://optlab.org/JRE1/2019jre1p.html

目次

A4 - Digits

問題概要

10 進法で 3 桁以下の整数 n が与えられるので、百の位、十の位、一の位をスペースで分けて出力してください。

- $0 \leq n \leq 999$

解法 1: 数学的に処理する

3 回のループで $i = 2, 1, 0$ を回し (逆向きであることに注意)、$\lfloor n/10^i \rfloor$ と空白を文字列 s に代入して n に n を 10^i で割ったあまりを再代入する方法です。($\lfloor x \rfloor$ はガウス記号)

例えば $n = 123$ のとき、

- まず $\lfloor n/10^2 \rfloor = 1$ と空白を s に代入。(s: "1 ")
- n に $n \mod 10^2 = 23$ を代入。
- 次に $\lfloor n/10^1 \rfloor = 2$ と空白を s に追加。(s: "1 2 ")
- n に $n \mod 10^1 = 3$ を代入。
- 次に $\lfloor n/10^0 \rfloor = 3$ と空白を s に追加。(s: "1 2 3 ")
- n に $n \mod 10^0 = 0$ を代入。(ここはしなくても良い)
- 最後に s の末尾に残った邪魔な空白を関数 strip で除去して出力。

となります。

strip は文字列の前後に残った余計な改行や空白を取り除くための標準関数です。また、$i = 2, 1, 0$ と逆向きにループを回すには標準関数 downto を使えば良いです。

実装例 (解法 1)

```
def digits(n)
    s = ""
    2.downto(0) do |i|
        s += (n/10**(i)).to_s
        s += " "
        n %= 10**(i)
    end
    puts s.strip
end
```

解法 2: 数字を文字列と見立てて処理する

渡された数字を to_s で文字列に変換して解く方法です。Ruby には次のような便利な標準ライブラリ関数が存在します。

split　文字列を引数で指定した文字で区切って文字列の配列を作る。何も指定しないときは 1 文字ごとに区切って配列にする。

join　配列を引数に指定した文字でつなげた文字列にする。

例えば a = "TrpFrog".split("") とすると a には ["T","r","p","F","r","o","g"] が代入されます。これに a.join(" ") をすると文字列「T r p F r o g」になります。

これと同じことを受け取った数 n を n.to_s で文字列にしたものに対して行い、puts で出力すれば良いです。これなら n が最大 3 桁という制約がなくても解くことができます。

実装例 (解法 2)

```
1 def digits(n)
2     puts n.to_s.split("").join(" ")
3 end
```

(なんとワンライナーですわ！)

Tips

実は Ruby も C 言語のように文字列自体を文字の配列と捉えることができます。ですから、$n = 123$ のときは n.to_s[0] とすれば 1 が出ます。これを使って s = n.to_s; puts s[0]+" "+s[1]+" "+s[2] でも良さそうですが、$n = 1$ など 3 桁以外の場合は配列の範囲外アクセスとなってしまうので sprit("") を使った方が良いです。

join も文字列には使えないようです (今知った)。

B5 - Rev

問題概要

整数 n $(1 \leq n < 10^9)$ が渡されるので n を反転させた数を出力してください。
例えば $\mathrm{rev}(123) = 321$, $\mathrm{rev}(1200) = 21$ です。

解法 1: 数学的に解く

A4 の解法 1 を使います。ここで桁数を d として `d-1.downto(0)` とすれば良さそうですが、残念ながら桁数は渡されません。もちろん高校レベルの知識で数学的に求めることもできます[*1]が、`n.to_s.length` を使った方が頭を使わなくて済むのでそれで求めてください。

今回は A4 とは違い、逆順に出力することが目的なので要素数 d の配列に桁を入れておきましょう。$i = d-1, d-2, \ldots, 1, 0$ の d 回のループを終えて配列 `a` に逆順に全ての桁が入ったとします[*2]。例えば $n = 12300$ のとき `a = [0,0,3,2,1]` です。

配列の中身を出力しましょう。しかし、これを素直に出力してしまうと `00321` となり、$\mathrm{rev}(12300) = 321$ に反します。そこでフラグを用意しておき、0 でない数字が現れた瞬間から出力を開始すれば良いです。

実装例 (解法 1)

```
def rev(n)
    d = n.to_s.length
    a = Array.new(d,0)
    (d-1).downto(0) do |i|
        a[i] = (n/10**(i))
        n %= 10**(i)
    end
    flag = false
    res = 0
    a.each_index do |i|
        flag = true if a[i] != 0
        next if !flag
        res += a[i] * 10**(d-1-i)
    end
    return res
end
```

[*1] 面倒ですが計算で求めることもできます。$d(n)$ を n の桁数を求める関数とします。このとき、$d(n) = \lfloor \log_{10} n \rfloor + 1$ が成り立ちます。Ruby には `log10()` という常用対数を求める関数と、`floor()` というガウス記号で囲んだときの結果を返す関数があるのでそれを使ってください。

[*2] $i = d-1, d-2, \ldots, 1, 0$ で `a[i]` に桁を詰めていくと勝手に逆順になります。

解法 2: 標準関数を駆使する

これもワンライナーで解けます。次の関数を使います。

to_s　数字の文字列にする関数。
split　指定した文字で区切って配列にする関数。
reverse　配列の要素の順番を反転させる関数。
join　配列を繋げて文字列にする関数。
to_i　文字列を整数にする関数。先頭に並ぶ 0 は無視される。

実装例 (解法 2)

```
1 def rev(n)
2     n.to_s.split("").reverse.join.to_i
3 end
```

Ruby は実は return を使わなくても良いときがあります。return が無い時は最後に評価された数が返されるようです。しかし、返されるものが最後に評価された変数でなかったり、返すものが分かりにくいとき (可読性が悪いとき) は return 文を書いてください。

C1 - Missing Num

問題概要

0 から 9 までの 10 個の数の中の異なる 9 個の数から成る配列 arr が与えられるので、この配列に含まれていない 0 から 9 までの数を当ててください。

解法 1: 頑張る

頑張って「0 が含まれるかどうか？」「1 が含まれるかどうか？」と 10 回確認してください。

解法 2: 和を取る

$0 + 1 + 2 + \cdots + 9 = 45$ なので 45 から配列の中身の和を引いたものが答えです。
Ruby では `arr.inject(:+)` で配列の和を取ることができます。

実装例 (解法 2)

```ruby
def missing_num(arr)
    45 - arr.inject(:+)
end
```

Tips: 計算量

計算量の考え方からいくと解法 2 の方が良いです。この問題が「0 から $N-1$ までの N 個の数字で...」となったときに解法 1 では二重ループで最悪 N^2 回計算する必要があります。つまり $O(N^2)$ のアルゴリズムです。一方解法 2 は和だけが求まれば良いので N 回の計算で済みます。つまり $O(N)$ のアルゴリズムです。

この 2 つがどれだけ違うかというと $N = 10^8$(1 億) のとき、解法 2 は 1 秒で計算が終わりますが[*3]、解法 1 では 3 年強の計算時間が必要になってしまいます。$N = 10^{12}$(1 京) のときはもっと大変です。$O(N)$ 解法では 3 時間弱で計算が終わりますが、$O(N^2)$ 解法では 317 万世紀 (!) かかってしまいます。計算量の工夫の重要さが分かりますね！

[*3] 1 秒間で 10^8 回計算が出来ると仮定

C3 - Has Duplicates?

問題概要

n 個の整数でできた配列が与えられます。足して 2 となる 2 数が存在するかを判定してください。

- $1 \leq n \leq 100$
- 配列の要素 a は $-100 \leq a \leq 100$
- 配列の要素は重複することがある

解法 1: 二重ループ

二重にループを回して 1 つ 1 つ確認してください。このとき同じ要素同士を足さないことに注意してください。(例えば 0 が一つしかないのに $0 + 0 = 0$ をしてしまうなど)

解法 2: 標準関数を使ってワンライナー

Ruby は関数が多すぎるので組み合わせを列挙する関数が存在します。ヤバすぎる。

配列.combination(x)　配列から x 個選ぶ組み合わせが全て入った配列 (二次元配列) を返す。
配列.all?{ 処理 }　配列の各要素に対する処理の戻り値が全て true かどうかを判定する。

all?はちょっとよく分かりにくいですが実装例を見てもらえば理解できると思います。

実装例 (解法 2)

```
1 def has_duplicates?(arr)
2     !arr.combination(2).all?{|e| e[0]+e[1]!=0}
3 end
```

全ての二数の和が 0 ではないとき、all?は true を返します。したがって、!演算子でそれをひっくり返せば良いです。

C5 - ジャンケン

問題概要

2 人が n 回じゃんけんをしたとき，勝敗の集計結果を出力するメソッド `janken(n, arr1, arr2)` を作成してください。

- $1 \leq n \leq 10000$
- 配列 `arr1`, `arr2` はプレイヤー 1 の手、プレイヤー 2 の手を表す
- 各配列の各要素はジャンケンの手を表す 0,1,2 のどれか
  - 0 は グー
  - 1 は チョキ
  - 2 は パー

また、次のように出力してください。

`Total:` n`, Win:` a`, Loss:` b`, Draw:` c

- a はプレイヤー 1 が勝利した数
- b はプレイヤー 1 が敗北した数
- c は引き分け (あいこ) の数

解法 1: 頑張る

頑張って「プレイヤー 1 が 0 でプレイヤー 2 が 1 のときはプレイヤー 1 の勝ちで.....」というように if 文を何個も書いてください。

解法 2: あまりの活用

ちょっとトリッキーな方法です。これをするとコード量を大幅に削減することができます。

少し考察をしてみましょう。同じ手ならばあいこ、つまり 2 つの差が 0 ということに注目して二つの手の引き算 (プレイヤー **2** の手 − プレイヤー **1** の手) を考えてみます。

$$\begin{bmatrix} 0-0 & 0-1 & 0-2 \\ 1-0 & 1-1 & 1-2 \\ 2-0 & 2-1 & 2-2 \end{bmatrix} = \begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 分 & 負 & 勝 \\ 勝 & 分 & 負 \\ 負 & 勝 & 分 \end{bmatrix}$$

分かりやすく行列にしてみました。このとき -1 が 2、-2 が 1 となってくれれば引き分け、勝ち、負けが $0, 1, 2$ に対応しそうです。そこであまりの出番です。$-1 \mod 3 = 2$, $-2 \mod 3 = 1$ ですから、全ての $\bmod 3$ を取れば引き分け、勝ち、負けが $0, 1, 2$ に対応します。

$$\text{勝敗を表す数字} = (\texttt{arr2[i]} - \texttt{arr1[i]}) \bmod 3$$

したがって、この計算式を使えば一発で勝敗の判定ができることが分かりました。

実装例 (解法 2)

```
def janken(n,arr1,arr2)
    w = 0; l = 0; d = 0
    arr1.each_index do |i|
        case (arr2[i] - arr1[i])%3
        when 0 then d += 1
        when 1 then w += 1
        when 2 then l += 1
        end
    end
    printf("Total: %d, Win: %d, Loss: %d, Draw: %d\n",n,w,l,d)
end
```

勝敗を表す数としての $0, 1, 2$ を配列の index に対応させるともっと短くなります。

```
def janken(n,arr1,arr2)
    r = [0,0,0]
    n.times {|i| r[(arr2[i]-arr1[i])%3]+=1}
    printf("Total: %d, Win: %d, Loss: %d, Draw: %d\n",n,r[1],r[2],r[0])
end
```

Tips

実は負の数のあまりの処理は言語によって異なります。Ruby ではあまりが必ず正の数になりますが全ての言語がそうであるとは限りません。例えば、Ruby の剰余演算子 % は

$$-5 \ \% \ 3 = 1$$

ですが、C 言語の場合は

$$-5 \ \% \ 3 = -2$$

となってしまいます。ですから、C 言語などの「負の数のあまりが負の数になる言語」で今回のようなジャンケンの判定を実装するときは「もし $a \ \% \ b$ が負なら b を足す」という処理を加えなければならないことに注意してください。

D5 - カエル跳びパズル

問題概要

カエル跳びパズルを作ってください。ルールは次の通り。

- カエルの目の前のマスが空いているとき、蛙はそこへ移動できる。
- カエルの目の前のマスが逆向きのカエルで占められているが、
  その奥のマスが空いているとき、カエルはその空いているマスへ移動できる。

カエルの数として整数 $n(1 \leq n \leq 20)$ が渡されるので盤面として要素数 $2n+1$ の配列を作成するメソッド `initialize_board(n)` を作成してください。盤面はグローバル変数としてください。盤面の状態としては、

- 0 は 何もいない
- 1 は 右向きのカエルがいる
- 2 は 左向きのカエルがいる

で、初期状態では左から n 個が右向きのカエル、右から n 個は左向きのカエルとなるようにしてください。

左から $i-1$ 番目のカエルをルールに従って移動させるメソッド `move(i)` も作成してください。$(0 \leq i < 2n+1)$ ただし、カエルが移動できないときは何もしないでください。

解法

`initialize_board` はやるだけなので省略。`move(i)` を作るにあたり、i 番目が盤面の上かどうかを返す関数 `onboard?(i)`、配列の要素を入れ替える `swap(i,j)` を作ると良いでしょう。ここではその 2 つの関数を作ったという前提で `move(i)` の解説を進めます。

次の手順に従って操作を行います。

1. カエルが移動する右向きの距離 m を決めます。
   右向きのカエルなら $m=1$、左向きのカエルなら $m=-1$ です。
2. カエルの 1 つ先のマスを見ます。つまり $i+m$ 番目のマスです。
3. `onboard(i+m)` が `false`、つまり範囲外であれば諦めて何もせず `return` します。
4. 進む先のマスの状態を確認します。
   同じ向きのカエル　ルールより移動できないので、諦めて何もせず `return` します。
   別の向きのカエル　もう 1 つ先の $i+2m$ 番目を見ます。$i+2m$ 番目でもダメならば諦めます。
   何もいない　次に進みます。
5. 進む先のマスを仮に g 番目のマスとします。g 番目に進むという行為は g 番目の値 $(=0)$ と i 番目の値を入れ替えることと同値ですから、`swap(i,g)` をします。
6. `return` します。

実装例

```
$board = []

def initialize_board(n)
    $board = []
    n.times {$board.push(1)}
    $board.push(0)
    n.times {$board.push(2)}
    return $board
end

def move(i)
    m = $board[i]
    m = -1 if m == 2
    2.times do |j|
        m *= j+1
        return $board unless onboard?(i+m)
        return $board if $board[i+m]==$board[i]
        swap(i,i+m) if $board[i+m] == 0
    end
    return $board
end

def onboard?(i)
    0<=i || i<$board.length
end

def swap(i,j)
    $board[i], $board[j] = $board[j], $board[i]
end
```

Tips

カエルは英語で Frog。

E2 - マインスイーパ

AtCoder にほぼ同じ問題があるので気になる人はお好きな言語でレッツ・トライ！
https://atcoder.jp/contests/abc075/tasks/abc075_b

問題概要

マインスイーパの盤面を表すサイズ $m \times n$ の二次元配列 board が与えられます。0 は何もないところ、1 は爆弾を表す。盤面上の全てのマスについて、周囲 8 マスの爆弾の数を代入した盤面と同じサイズの二次元配列を返してください。ただし、爆弾が存在するマスには 0 を代入してください。

- $1 \leq m \leq 100$
- $1 \leq n \leq 100$

解法

まず 8 方向の探索の準備として探索用配列 dx,dy を作ります。
dx = [-1,0,1,1,1,0,-1,-1]; dy = [1,1,1,0,-1,-1,-1,0] です。例えば (x,y) の周囲 8 マスを探索したいときは (x+dx[i],y+dy[i]) で i=0,1,2,...,7 の 8 回ループをするだけで良いので便利です。これは二次元配列を操作するときのテクニックとしてよく使われる[要出典]ので覚えておきましょう。

次に結果を返すために盤面と同じ大きさの二次元配列を作り、それを result とします。board の各マスについて次の操作を行います。

1. 爆弾マスであれば next で次のマスにうつる
2. 変数 bomb = 0 を用意する
3. 8 方向探索を始める
4. 範囲外であれば次へ
5. board[x+dx[i]][y+dy[i]] が爆弾マスであれば bomb を 1 増やす
6. 全方位の探索が終わったら bomb を対応する result の座標に代入する
7. 次のマスへ進む

Ruby には配列の範囲外にアクセスするとエラーを出して止まるのではなく、代わりに指定した別の値を持ってくるという便利な 配列.fetch(index, value) という関数が存在します。しかし、Ruby において負の数の index は範囲外ではなく、逆順の index という扱いになることに注意してください。例えば arr = [0,1,2] のとき、arr[-1] は実行時エラーではなく 2 が返ってきます。

実装例

```
X=0; Y=1;
def sweep(board)
    dx = [-1,0,1,1,1,0,-1,-1]
    dy = [1,1,1,0,-1,-1,-1,0]
    result = []
    board.each do |e|
        result.push(Array.new(e.length,0))
    end
    h = board.length
    w = board[0].length
    board.each_index do |i|
        [*0..h-1].product([*0..w-1]).each do |v|
            next if board[v[X]][v[Y]]==1
            bombs = 0
            8.times do |j|
                next if v[X]+dx[j] < 0 || v[Y]+dy[j] < 0
                bombs += board.fetch(v[X]+dx[j],[]).fetch(v[Y]+dy[j],0)
            end
            result[v[X]][v[Y]] = bombs
        end
    end
    return result
end
```

可読性のために添字 $X = 0, Y = 1$ という定数が定義してあります。

Ruby には 2 つの配列 A, B の直積[*4] $A \times B$ の配列を返す関数 `A.product(B)` が存在します。これを使えば `[*0..h-1].product([*0..w-1]).each do |v|` と書くことで、`w.times do |x| h.times do |y| ..... end end` という二重ループの代わりをすることができます。`[*0..x]` は `[0,1,2,3,...,x]` という配列を表します。`v` は要素数 2 の配列で x 座標と y 座標は先ほど定義した定数 X, Y を使って、`v[X],v[Y]` と表されます。

[*4] 直積とは集合 A と集合 B から要素を一つずつ取ってきた時の全てのペアの集合のことです。例えば $\{1,2,3\} \times \{a,b\} = \{(1,a),(1,b),(2,a),(2,b),(3,a),(3,b)\}$ です。I 類の人はたぶん離散数学でやりましたね！

E4 - Transpose

問題概要

全ての要素が整数のサイズ $m \times n$ の二次元配列 mat が与えられるので、それを転置した二次元配列を返してください。

- $1 \leq m \leq 100$
- $1 \leq n \leq 100$

解法 1: 頑張る

サイズが $n \times m$ の二次元配列 result を用意し、二重ループで

```
result[j][i] = mat[i][j]
```

をすれば良いです。同じサイズではなく転置したサイズであることに気をつけてください。

実装例 (解法 1)

```ruby
def transpose(mat)
    result = Array.new(mat[0].length){Array.new(mat.length)}
    mat.length.times do |i|
        mat[i].length.times do |j|
            result[j][i] = mat[i][j]
        end
    end
    return result
end
```

解法 2: 怠ける

mat.transpose を return すれば良いです。なんで？

実装例 (解法 2)

```ruby
def transpose(mat)
    mat.transpose
end
```

E5 - オセロ

悪名高き P 演習最高難度の問題。ヤバいです。~~囲碁 (E5) なのにオセロ。~~

問題概要

オセロを模倣するプログラムを作ってください。まず initialize_othello で 8×8 の初期盤面として二次元配列$board を用意し次のように初期化してください。

- $board[3][4] と$board[4][3] には黒石として「1」を置く
- $board[3][3] と$board[4][4] には白石として「2」を置く
- 他は全部何もないものとして「0」を置く

次に石を置き、その後に挟んだ石を反転させる処理を行う move_othello(p,i,j) を作成してください。ここで引数は次の通り。

- p はプレイヤー番号 (1:先手, 黒石 2:後手, 白石)
- i, j は盤面の位置で$board[i][j] に石を置くことを意味する

なお、move_othello(p,i,j) で石が置けない場合もあるので、その場合はパスとみなし、配列を返してください。石が置ける場合は石をおいて挟んだ石を裏返す処理をした後、配列を返してください。

- $p = 1, 2$
- $0 \leq i, j \leq 7$

解法

initialize_othello はやるだけなので実装例を見てください。ここでは move_othello(p,i,j) の解説をします。

まず$board[i][j] が 0 でない、すなわち先に石が置かれているならば return します。次は二次元配列の探索の常套手段、配列 dx,dy[*5]の出番です。(i, j) の周囲 8 方向を確認します。周囲 8 方向について次の操作を行います。

1. そのマスに相手の石がなければ next で次のマスに進む。
2. 石を裏返す関数 reverse(p,i+dx[k],j+dy[k],dx[k],dy[k]) を実行する。
3. もし reverse で石を裏返したことを表すフラグ$flag が立っていれば、
   $board[i][j] = p で自分の石を$board[i][j] に置く。$flag = false でフラグを元に戻す。

小技として、相手の石の番号は $3 - p$ で計算できます。

[*5] 詳しくは E2 - Minesweeper の解説を参照

次に関数 reverse を作ります。reverse で石を裏返す操作としては次のことが考えられます。

- 隣のマスを見る。
- 相手の石ならばもう一つ隣を見る。
- ……
- 相手の石ならばもう一つ隣を見る。
- 自分の石が現れたら今までの相手の石を全て裏返す。
  現れなければ何もせずに return する。

これを愚直に実装すれば良いです。これは再帰呼び出しを使うと書きやすいです。

まずグローバル変数として、石を裏返していいことをお知らせするフラグ$flag を作ります。reverse では 5 つの数を受け取ります。

p　　プレイヤー番号
i, j　探索する座標
dx, dy　探索方向を示す dx,dy

呼び出される度に次の操作を行います。

1. まず探索する座標 (x, y) が盤面の範囲外であれば return する。
2. (i, j) の盤面の文字を確認する
   何もない (=0)　何もせず return
   自分の石 (=p)　探索終了。石の裏返し命令$flag を true にして return
   相手の石 (=3-p)　探索続行。reverse(p,i+dx,j+dy,dx,dy) で隣を探索。
3. return されるとここに帰ってくる。もし$flag が true、すなわち上流から石の裏返し命令が出ていたらここで$board[i][j] = p する。
   ここで (i, j) は move_othello の (i, j) ではなく、reverse の (i, j) であることに注意。
4. 操作を終了する (return)

ここでのポイントは一度$flag が true になると今まで見てきた分全ての裏返しが始まるところです。再帰関数ならではの動きですね。

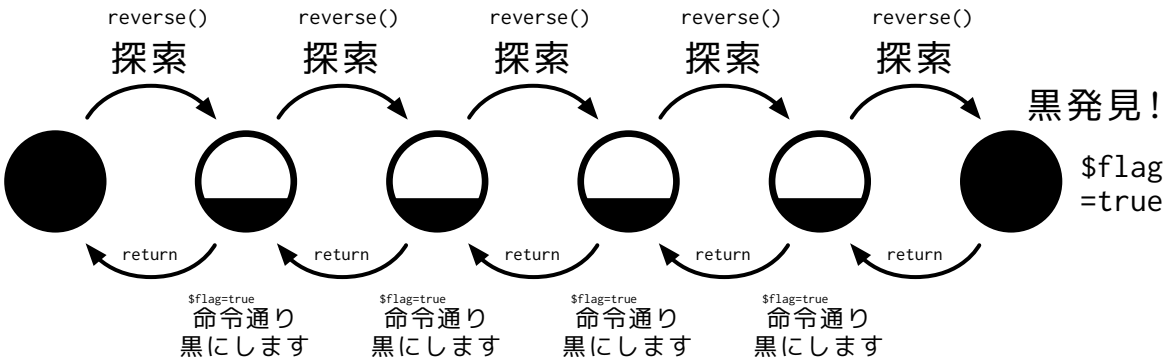

図 1　reverse() の再帰のイメージ

実装例

```
$board = []
def initialize_othello
    $board = Array.new(8){Array.new(8,0)}
    $board[3][4] = $board[4][3] = 1
    $board[3][3] = $board[4][4] = 2
    return $board
end

$flag = false
def move_othello(p,i,j)
    return $board unless $board[i][j]==0
    dx = [-1,0,1,1,1,0,-1,-1]
    dy = [1,1,1,0,-1,-1,-1,0]
    8.times do |k|
        next unless get(i+dx[k],j+dy[k])==(3-p)
        reverse(p,i+dx[k],j+dy[k],dx[k],dy[k])
        $board[i][j] = p if $flag
        $flag = false
    end
    return $board
end

def reverse(p,i,j,dx,dy)
    return unless 0<=i && i<=8 && 0 <= j && j <= 8
    case get(i,j)
    when 0 #empty
        return
    when p #your color
        $flag = true
        return
    when 3-p #enemy color
        reverse(p,i+dx,j+dy,dx,dy) #move
    end
    $board[i][j] = p if $flag
end

def get(x,y)
    $board.fetch(x,[]).fetch(y,0)
end
```

F1 - カタラン数

ここから C 言語です。

問題概要

整数 n $(0 \leq n \leq 19)$ が渡されるので、次の式で定義されるカタラン数 C_n を求めてください。

$$C_n = \begin{cases} 1 & (n = 0, 1) \\ \sum_{i=0}^{n-1} C_i C_{n-i-1} & (n \leq 2) \end{cases}$$

解法

やるだけ、といえばやるだけですが、再帰が絡むので少し難しいです。再帰関数は

- 再帰呼び出しをせず処理を終了する**ベースケース**
- 再起呼び出しをした結果を処理する**再帰ステップ**

の 2 つのパートから成り立ちます。これさえ意識できれば再帰関数も少しは簡単に書けるかもしれません。再帰関数についてはこちらのサイト (https://atcoder.jp/contests/apg4b/tasks/APG4b_v) に詳しく載っているので勉強したい人はどうぞ。(ここのサイトは C++ で解説していますが、まあだいたい C 言語と同じなので.....)

実装例

```c
#include <stdio.h>
#include <stdlib.h>

int catalan(int n) {
    if (n <= 1) return 1;
    int ans = 0;
    for (int i = 0; i <= n - 1; i++) {
       ans += catalan(i) * catalan(n - i - 1);
    }
    return ans;
}

int main(int argc, char *argv[]) {
    int ans = catalan(atoi(argv[1]));
    printf("%d\n", ans);
    return 0;
}
```

F2 - ウォリスの公式

問題概要

整数 k が与えられるので、円周率の近似値を計算する次の式で求まる値を出力してください。

$$2\prod_{i=1}^{k}\frac{4i^2}{4i^2-1}$$

- $1 \leq k \leq 2 \times 10^5$

解法

これもやるだけですが、注意点があります。int 型の最大値が $2^{31}-1 = 4,294,967,295 \approx 4.3\times10^9$ なのに対し、この問題で出現する i^2 としてあり得る最大値は $i^2 \leq k^2 \leq (2 \times 10^5)^2 = 4 \times 10^{10}$ でオーバーフローしてしまいます。そこで $2^{63}-1 = 9,223,372,036,854,775,807 \approx 9.2 \times 10^{18}$ まで扱える long long 型を使えば解決します。(long 型でチェッカーは通ったけど、32bit の PC だと int と同じになったりするらしいので注意[要出典])

実装例

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int k = atoi(argv[1]);
    double ans = 2;
    for(long i=1; i<=k; i++){
        ans *= (4*i*i)/((double)4*i*i-1);
    }
    printf("%.3f\n",ans);
    return 0;
}
```

F5 - ボウリングの得点計算

問題概要

入力としてボウリング 1 ゲームにおける一人のプレイヤーの投球記録 (その投球により倒したピンの総数) が与えられるので、次のルールと計算方法に従って得点を計算してください。

ルール

- 1 ゲームは 10 個のフレームで構成される
- 各フレームの開始時には 10 本のピンが立てられている
- 第 1 フレームから第 9 フレームでは最大 2 回投球が行える
- 第 1 投で 10 本すべてのピンを倒すことをストライクと呼ぶ
- 第 1 投がストライクであった場合、第 2 投は行わない
- 第 2 投では倒れずに残されたピンを倒すことを目的とする
- 第 1 投と第 2 投で 10 本すべてのピンを倒すことをスペアと呼ぶ
- 第 10 フレームでは、最大 3 回投球が行える
- 第 10 フレームにおいて、第 1 投でストライクを出せば第 2 投の前に
  新たに 10 本のピンが立てられる
- 更に第 2 投でストライクを出せば第 3 投の前に新たに 10 本のピンが立てられる
- 第 10 フレームにおいて、第 1 投と第 2 投で 10 本すべてのピンを倒すことができれば
  スペアとなる
- そのとき、第 3 投の前に新たに 10 本のピンが立てられるそれを全て倒せばストライクとなる
- 第 10 フレームにおいて、第 1 投と第 2 投で 10 本すべてのピンを倒すことができなければ
  第 3 投を行うことはできない

計算方法

- 各フレームにおいて倒したピンの総数がそのまま得点に加算される
- 第 1〜9 フレームにおける各スペアに対して、
  その直後の投球で倒したピンの数がボーナス得点として加算される
- 第 1〜9 フレームにおける各ストライクに対して、
  その直後 2 つの投球で倒したピンの総数がボーナス得点として加算される

入出力

入力は総投球数 n、各投球で倒れたピンの数 $p_1, p_2, \ldots, p_n$ がコマンドライン引数で与えられます。得点を出力してください。

- $11 \leq n \leq 21$
- $0 \leq p_i \leq 10$ $(1 \leq i \leq n)$

解法

この問題において重要なのは第 1〜9 フレームのゲームの進め方と得点の計算方法です。第 10 フレームのめんどくさい処理は全く追う必要がありません。なぜならば、得点は「倒したピンの和 + 第 1〜9 フレームにおいて生じるボーナス点」によって決まるため、第 10 フレームのゲームの進め方は得点計算に全く関係しないからです。

以上を踏まえてもう一度ルールと得点の計算方法を読み直すと次の手順に沿って計算を進めれば良いことがわかります。

1. 倒したピンの和 $\sum_{i=1}^{n} p_i$ を計算しスコアとする
2. 各投球ごとの倒したピンをループで回す
3. もし i 回目でストライクをしていれば $p_{i+1} + p_{i+2}$ をスコアに加算
4. もし i 回目でスペアをしていれば p_{i+1} をスコアに加算
5. スコアを出力する

ストライクとスペアの判定は次のように行います。

1. フレーム数を管理する変数 `int frame = 1` を作成する
2. 各フレームでの投球数を管理する変数 `int pitch = 1` を作成する
3. $i = 1, \ldots, n$ のループに入る
4. もし `pitch` $= 1$ かつ $p_i = 10$ であればストライクとする
5. もし `pitch` $= 2$ かつ $p_{i-1} + p_i = 10$ であればスペアとする
6. `pitch` $= 1$ でストライクでなければ `pitch` $= 2$ とし i を 1 増やして `cointinue` する
7. `pitch` を 1 に戻し、`frame` を 1 増やす
8. これを `frame` ≤ 9 である限り繰り返す

これらの処理をそのまま実装すれば良いです。

`continue` とは C 言語における Ruby の `next` で、一旦ループの今やっている回の処理をやめてループの頭に戻るという指示です。このとき単に頭に戻るのではなく、`for(int i=0; i<n; i++)` の `i++` のような継続処理の式が実行されることに気をつけてください。

Tips

P 演習のサイトだと「ボウリング」ではなく「ボーリング」になっている。調査するぞ！

実装例

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool is_strike(int pitch, int pins){
    return (pitch == 1 && pins == 10);
}

bool is_spare(int pitch, int prevpins, int pins){
    return (pitch == 2 && prevpins + pins == 10);
}

int main(int argc, char *argv[]){
    int n = atoi(argv[1]);
    int totalscore = 0;
    int pins[n];
    for(int i=0; i<n; i++){
        pins[i] = atoi(argv[i + 2]);
        totalscore += pins[i];
    }

    //judge strike/spare
    int frame = 1, pitch = 1;
    for(int i=0; frame<10; i++){
        if(is_strike(pitch,pins[i])){
            totalscore += pins[i+1]+pins[i+2];
        }else if(is_spare(pitch,pins[i-1],pins[i])){
            totalscore += pins[i+1];
        }else if(pitch == 1){
            pitch++;
            continue;
        }
        pitch = 1;
        frame++;
    }

    printf("%d\n",totalscore);
    return 0;
}
```

G4 - シーザー暗号

問題概要

平文として大文字アルファベットの文字列 S と鍵として整数 n が与えられるので、各アルファベットを n 後ろにずらした暗号文 E を出力してください。ここで後ろにずらすとは A → B → C →... → X → Y → Z → A → B →... ということを指します。
例えば S =ABCXYZ, $n = 3$ のとき E =DEFABC です。

- $0 \leq n \leq 25$
- $1 \leq |S| \leq 100$ (文字数)
- S は全て大文字のアルファベット

解法

文字の”実体”は数字です。例えば'A' の実体は 65 です。これを char に入れることで初めて'A' として扱われます。つまり文字と数字は対応しています。また、通常文字コードはアルファベットで連続しています。例えば、'A'+1 を char 型に入れると'B' となります。同様に'A'+2 は'C'、'A'+3 は'D'、'D'+1 は'E' です。この特徴を使うとこの問題を解くことができます。

さて、このことから「文字を後ろにずらす操作」は S[i]+n で出来ると予想できそうです。'Z' をこの方法で 1 つ後ろにずらすことを考えてみます。'Z'+1 はどうなるでしょうか？残念ながら答えは'A' ではなく'[' です。当然文字コードには大文字アルファベット以外も割り当てているのでこのようなことがおきてしまいます。従って'A' = 65 から'Z' = 90 におさまるように「アルファベットをずらす」計算をする必要があります。

でもやっぱり S[i]+n で「文字を後ろにずらす操作」をした方がとても都合が良いので、この方法を出来るだけ崩さないようにするにはどうすれば良いのかを考えます。そういえば 26[*6]で割ったあまりの世界では、演算結果は必ず 0 から 25 の間になります。例えば $23 + 7 \equiv 4 \pmod{26}$ です。これを使いましょう。

答えが'A' から'Z' に収まれば良いので次の式を考えます。

$$\texttt{E[i]} = (\texttt{S[i]} - \texttt{'A'} + n) \% \ 26 + \texttt{'A'}$$

$0 \leq$S[i] − 'A'< 26 ですからこれに n を足してずらして、26 以上になって溢れた分は剰余演算子で 0 から 25 の範囲に納めてあげれば良いです。最後に最初に引いた'A' の文を足して戻してあげれば、'A'≤E[i]≤'A'+25 = 'Z' となります。自分でもよく分かんなくなってきた、伝われ (雑)

[*6] アルファベットの数

実装例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

int main(int argc, char *argv[]) {
    char *str;
    str = argv[1];
    int n = atoi(argv[2]);
    for(int i=0; str[i]!='\0'; i++){
        printf("%c",(char)(((int)str[i]-(int)'A'+n)%26+(int)('A')));
    }
    printf("\n");
    return 0;
}
```

char *str = argv[1] というようにポインタ型で宣言した char に argv を突っ込むことで文字列を代入することができます。(でもこれ P 演の時間中手探りで見つけたやつなので C 言語つよつよの人で「もっといい方法あるよ！」って人がいたら教えてください)

G5 - 足し算

問題概要

整数 a, b が渡されるので $a + b$ を出力してください。ただし、a と b の大きさに注意してください。

- $0 \leq a, b < 10^{40}$

解法

足し算だから簡単だよ！と言いたいところですが、a, b それぞれ最大 $10^{40} - 1$ という頭が悪い制約のせいでとても計算が大変になっています。F2 の解説でも書いた long long 型を使っても高々 10^{18} 程度なので到底使えそうにありません。そこで、数字だけが入った文字列同士を計算させることを考えます。

まず「数字の文字」同士の演算に便利な関数を作ります。”数字の文字”`char a` と `char b` を受け取り、$a + b$ の計算結果を int で返す `char_addition(char a, char b)`、”数字の文字”`char c` を受け取り、それを int にして返す `ctoi(char c)` の 2 つを作ります。G4 が理解できていれば、この関数も作れると思います。

次に `main` 関数の処理を考えます。簡単な流れとして次が考えられます。

1. a, b のうち桁数が多い方を m、もう一方を n とする
2. 答えを入れる要素数 41 の int 配列 `ans[41]` を作る
   (最大 40 桁同士の足し算なので答えが 41 桁になり得る)
3. 1 の位から数えて 1 から n の最大桁数までは `char_addition` で n と m の各桁同士の和を計算し、その一の位を `ans[i]` に加算、十の位を `ans[i+1]` に加算
4. n の最大桁数より先は m の最大桁数まで `ctoi` でその桁を int に直したものを `ans[i]` に加算し続ける
5. 最後に配列 `ans` の頭から出力すれば良い
   このとき先頭の 0 は解に関係ないので出力しないことに注意する[*7]

要するにこれは足し算の筆算をプログラムにしただけです。計算量は $O(\log \max(a, b))$ で十分高速です。

[*7] 詳しくは B5 - Rev の解説を参照

実装例

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int char_addition(char a, char b) {
    return (int)(a + b - 2 * '0');
}

int ctoi(char c){
    return (int)(c-'0');
}

int main(int argc, char *argv[]) {
    char *n, *m;
    if (strlen(argv[1]) < strlen(argv[2])){
        n = argv[1];
        m = argv[2];
    }else{
        m = argv[1];
        n = argv[2];
    }
    int n_len = (int)strlen(n), m_len = (int)strlen(m);
    int ans[41];
    ans[0] = 0;
    int i;
    for (i = 0; i<m_len; i++) {
        if(i < n_len){
            ans[i] += char_addition(n[n_len-1-i], m[m_len-1-i]);
        }else{
            ans[i] += ctoi(m[m_len-1-i]);
        }
        ans[i + 1] = ans[i] / 10;
        ans[i] %= 10;
    }
    if(ans[i] != 0) i++;
    while(--i >=0){
        printf("%d",ans[i]);
    }
    printf("\n");
    return 0;
}
```

- `ans[0]` だけ 0 で初期化しているのは `ans[1]` 以降はあとで「代入」するため今 0 で初期化する必要がないからです。でもバグ防止のために 0 で初期化するのはアリだと思います。
- `n_len-1-i` は配列の後ろから何番目？をやっています
- `ans[i + 1] = ans[i] / 10` あたりの処理で困ったら A4 の解説を読んでください。
- `if(ans[i] != 0) i++` は $\max(a, b)$ の最高桁で繰り上がりが発生したかどうかを判定しています。繰り上がるようなら i をインクリメントします。
- `while(--i >=0)` で今まで増やしてきた `i` を一気に降下させて `ans` の中身を出力をします。これをすることで、どこまで先頭に 0 が詰まっているかを気にする必要がなくなります。
- これ可読性がカスなので読めなかったらごめんなさい。カッコつけすぎてる (これでも一応手直ししました、オリジナルはもっとゴミだったのでコード読みながら泣いてしまいました)

Tips: 多倍長整数

このように非常に大きな整数同士の演算をサポートするライブラリが言語によっては存在します。このようにデータ型にとらわれず任意に数値の精度を伸ばすことが出来る整数を多倍長整数と言ったりします。(Wikipedia 情報) `https://ja.wikipedia.org/wiki/`任意精度演算