

「アルゴリズム×数学」が
基礎からしっかり身につく本

解答・解説



第 2 章 アルゴリズムのための数学の基本知識	3
節末問題 2.1 の解答 …… 4	節末問題 2.4 の解答 …… 13
節末問題 2.2 の解答 …… 6	節末問題 2.5 の解答 …… 16
節末問題 2.3 の解答 …… 9	
第 3 章 基本的なアルゴリズム	20
節末問題 3.1 の解答 …… 21	節末問題 3.5 の解答 …… 36
節末問題 3.2 の解答 …… 23	節末問題 3.6 の解答 …… 39
節末問題 3.3 の解答 …… 26	節末問題 3.7 の解答 …… 42
節末問題 3.4 の解答 …… 31	
第 4 章 発展的なアルゴリズム	47
節末問題 4.1 の解答 …… 48	節末問題 4.5 の解答 …… 70
節末問題 4.2 の解答 …… 54	節末問題 4.6 の解答 …… 81
節末問題 4.3 の解答 …… 59	節末問題 4.7 の解答 …… 85
節末問題 4.4 の解答 …… 63	
第 5 章 問題解決のための数学的考察	95
節末問題 5.2 の解答 …… 96	節末問題 5.7 の解答 …… 114
節末問題 5.3 の解答 …… 100	節末問題 5.8 の解答 …… 118
節末問題 5.4 の解答 …… 103	節末問題 5.9 の解答 …… 123
節末問題 5.5 の解答 …… 108	節末問題 5.10 の解答 …… 129
節末問題 5.6 の解答 …… 111	
第 6 章 最終確認問題	140
問題 01～15 …………… 141	問題 16～30 …………… 153

第2章

アルゴリズムのための
数学の基本知識

2.1

節末問題 2.1 の解答

問題 2.1.1

この問題は、数の分類（→2.1.1項）を理解しているかを問う問題です。解答は、

- 整数：**-100, -20, 0, 1, 70**
- 正の整数：**1, 70**

となります。整数は小数点が付かないような数、正の整数はそれらのうち 0 より大きい数のこと指します。

問題 2.1.2

この問題は、文字式とその書き方（→2.1.2項）を問う問題です。解答は、

- $A + B + C = 25 + 4 + 12 = 41$
- $ABC = 25 \times 4 \times 12 = 1200$

となります。ここで、 ABC は $A \times B \times C$ という意味であることに注意してください。

問題 2.1.3

この問題は、「3つの整数を入力し、この掛け算の値を出力してください」という意味です。したがって、次のようなプログラムを書くと正解が得られます。なお、数列のように番号を付した文字式に慣れていない人は、2.1.4 項・2.1.5 項に戻って復習しましょう。

```
#include <iostream>
using namespace std;

int main() {
    int A[4];
    cin >> A[1] >> A[2] >> A[3];
    cout << A[1] * A[2] * A[3] << endl;
    return 0;
}
```

※ Pythonなどのソースコードは GitHub の codes フォルダをご覧ください。

問題 2.1.4 (1)

この問題は、2進法を10進法に変換する方法（→2.1.7項）を問う問題です。以下のようにして計算すると、答えが9であることが分かります。

- 8の位の値は1
- 4の位の値は0
- 2の位の値は0
- 1の位の値は1
- 位とその桁の数の足し算は、 $(8 \times 1) + (4 \times 0) + (2 \times 0) + (1 \times 1) = 9$

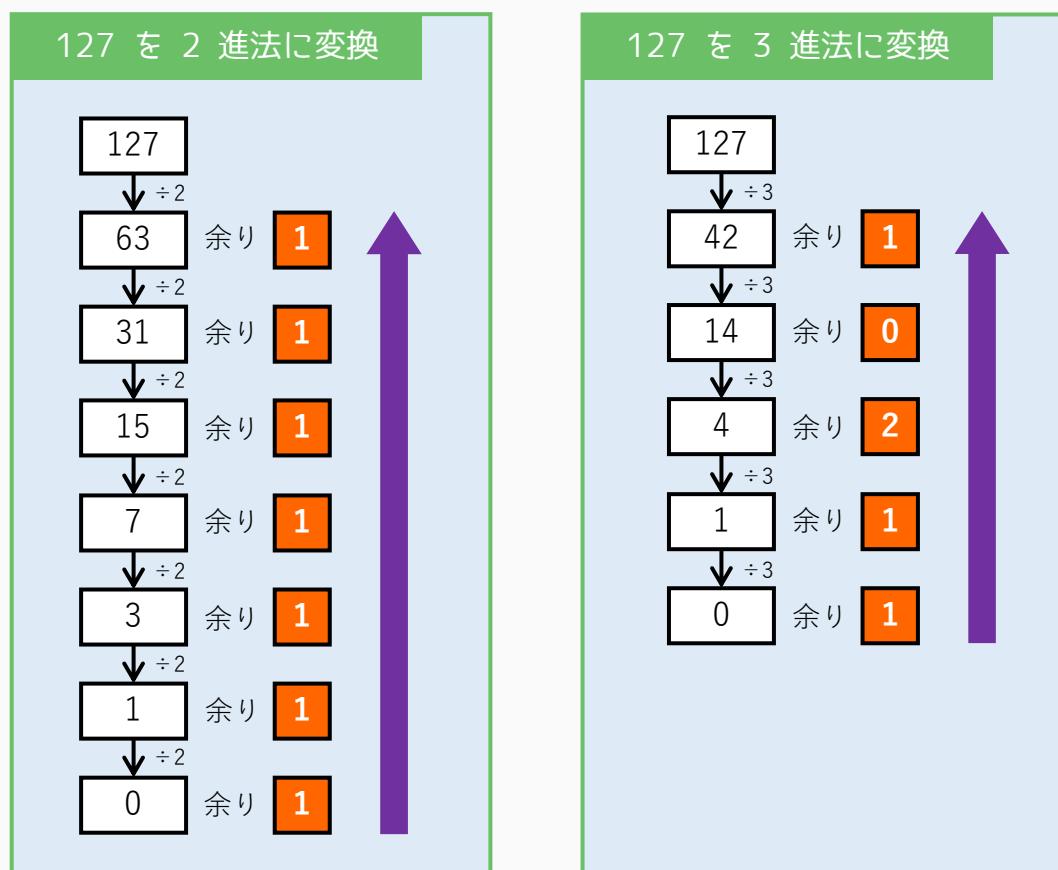
問題 2.1.4 (2)

この問題は、10進法を2進法などに変換する方法（→2.1.9項）を問う問題です。

一般に、10進法をK進法に変換するときは、値がゼロになるまでKで割り続け、余りを後ろから読めば良いです。したがって、以下のようにして計算すると、

- 127の2進法表記が1111111
- 127の3進法表記が11201

であることが分かります。イメージ図は以下の通りです。



2.2

節末問題 2.2 の解答

問題 2.2.1

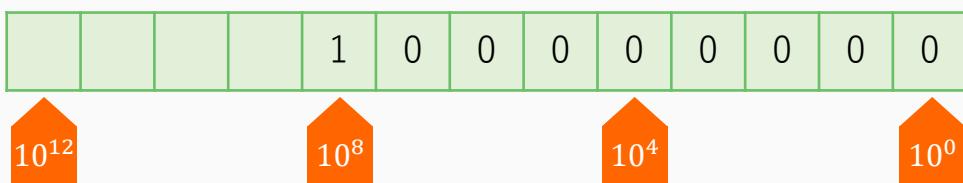
この問題は、累乗（→2.2.3項）を理解しているかどうかを問う問題です。早速ですが、答えは以下の通りです。

- 1万は **10 の 4 乗** (10^4)
- 1億は **10 の 8 乗** (10^8)
- 1兆は **10 の 12 乗** (10^{12})

これは、以下のように「 10 の n 乗を計算するとき、 n が 1 増えるとゼロの数が 1 増える」と考えると、簡単に答えが分かります。

- $10^1 = 10$ (ゼロが 1 個)
- $10^2 = 10 \times 10 = 100$ (ゼロが 2 個)
- $10^3 = 10 \times 10 \times 10 = 1000$ (ゼロが 3 個)
- $10^4 = 10 \times 10 \times 10 \times 10 = 10000$ (ゼロが 4 個)

1万は 10000 (ゼロが 4 個)、1億は 100000000 (ゼロが 8 個)、1兆は 1000000000000 (ゼロが 12 個) なので、それぞれ $10^4, 10^8, 10^{12}$ です。



問題 2.2.2

この問題は、累乗（→2.2.3項）、ルート（→2.2.4項）を理解しているかどうかを問う問題です。答えは以下のようになります。

- $29^2 = 841$ であり、 $\sqrt{841} = 29$
- $4^5 = 1024$ であり、 $\sqrt[5]{1024} = 4$

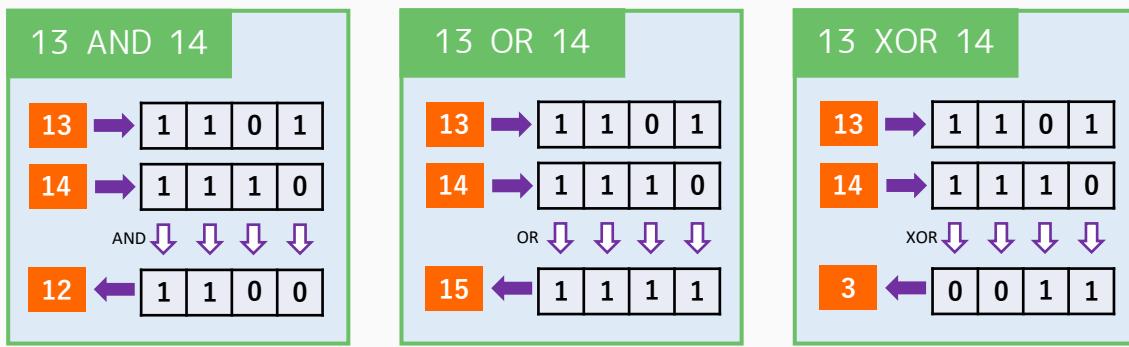
なお、 $a^b = x$ のとき $\sqrt[b]{x} = a$ となるという重要な性質が成り立ちます。

問題 2.2.3 (1)

この問題は、2つの数のビット演算（→2.2.7項～2.2.9項）を理解しているかどうかを問う問題です。以下のようない算により、

- $13 \text{ AND } 14 = 12$
- $13 \text{ OR } 14 = 15$
- $13 \text{ XOR } 14 = 3$

となります。分からぬ人は、論理演算の AND は「両方 1 ならば 1」、OR は「片方でも 1 ならば 1」、XOR は「片方だけ 1 ならば 1」であったことを思い出しましょう。ビット演算は、それぞれの数を2進法に変換し、それぞれの桁に対して論理演算を行うものです。



問題 2.2.3 (2)

この問題は、3つ以上の数のビット演算（→2.2.11項）を理解しているかどうかを問う問題です。以下のようない算により、答えが 15 であることが分かります。

$$\begin{aligned} & ((8 \text{ OR } 4) \text{ OR } 2) \text{ OR } 1 \\ &= (12 \text{ OR } 2) \text{ OR } 1 \\ &= 14 \text{ OR } 1 \\ &= 15 \end{aligned}$$

なお、8、4、2、1 を2進法に変換すると 1000、0100、0010、0001 となり、4つすべての桁について「1」が1個以上存在するため、答えは2進法で 1111 (10進法で 15) だと計算することも可能です。

問題 2.2.4

この問題は、剰余 (mod) の実装方法（→[2.2.1項](#)）を理解しているかどうかを問う問題です。C++ の場合、以下のようなプログラムを書くと正解が得られます。なお、C++ や Python などのプログラミング言語では、変数 `a` を `b` で割った余りは `a % b` で計算できます。

```
#include <iostream>
using namespace std;

int N, A[109];
int Answer = 0;

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) {
        cin >> A[i];
    }

    // 答えの計算
    for (int i = 1; i <= N; i++) {
        Answer += A[i];
    }

    // 出力
    cout << Answer % 100 << endl;
    return 0;
}
```

※ Python などのソースコードは chap2-2.md をご覧ください。

2.3

節末問題 2.3 の解答

問題 2.3.1

この問題は、 $f(x)$ といった関数の表記（→2.3.1項）、多項式関数（→2.3.7項）の理解を問う問題です。答えは以下の通りです。

- $f(1) = 1^3 = 1 \times 1 \times 1 = 1$
- $f(5) = 5^3 = 5 \times 5 \times 5 = 125$
- $f(10) = 10^3 = 10 \times 10 \times 10 = 1000$

なお、 $f(x) = ax^3 + bx^2 + cx + d$ の形で表される関数を **三次関数** といいます。この問題の $f(x) = x^3$ も三次関数の一種です。

問題 2.3.2 (1)

この問題は、対数関数（→2.3.10項）の理解を問う問題です。

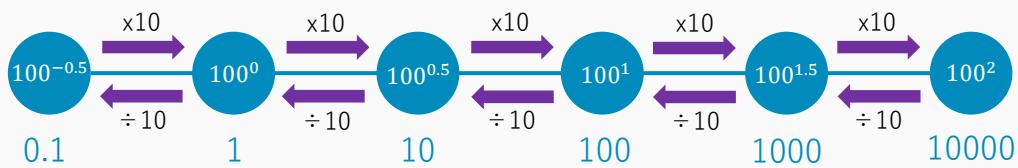
$2^3 = 8$ であるため、答えは $\log_2 8 = 3$ です。（36 ページの例を参照）

問題 2.3.2 (2)

この問題は、べき乗の拡張（→2.3.8項）の理解を問う問題です。

一般に、 $a^{\frac{n}{m}} = \sqrt[m]{a^n}$ が成り立つため、 $a = 100, n = 3, m = 2$ を代入することで、

答えが $100^{1.5} = \sqrt{100^3} = \sqrt{1000000} = 1000$ だと分かります。



問題 2.3.2 (3)

この問題は、床関数・天井関数（→2.3.11項）の理解を問う問題です。

[20.21] は 20.21 以下の最大の整数 20 です。

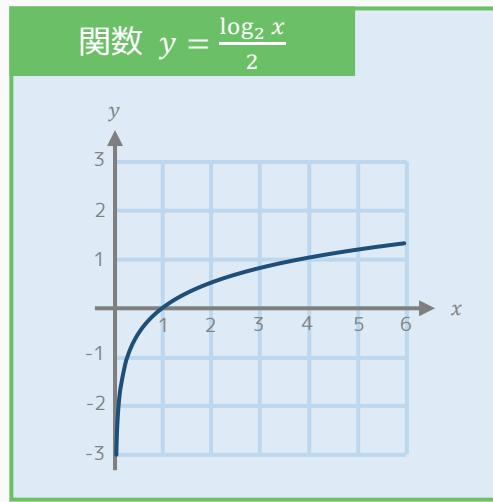
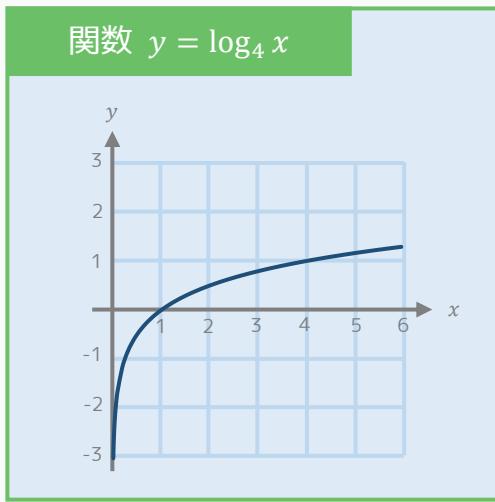
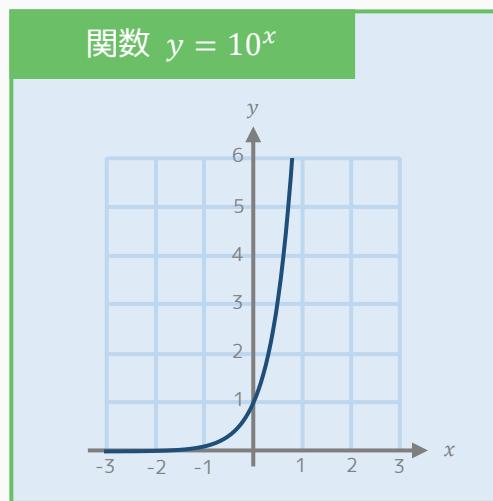
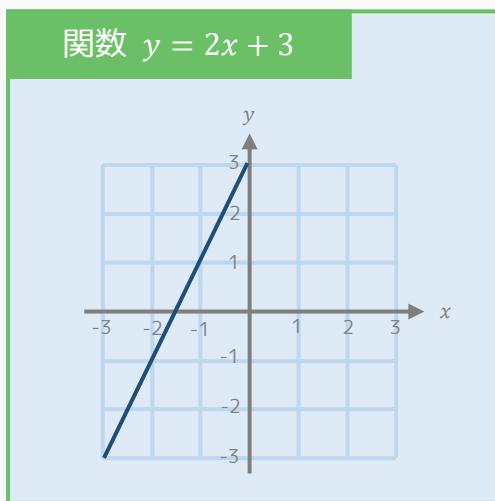
[20.21] は 20.21 以上の最小の整数 21 です。

問題 2.3.3

この問題は、関数のグラフ（→2.3.4項）の理解を問う問題です。答えは下図のようになります。以下の点に注意するとグラフが描きやすいです。

- 一次関数のグラフは直線である
- 指数関数は単調増加であり、増加ペースも急増する
- 対数関数は単調増加であるが、増加ペースは遅くなる

なお、3つ目のグラフと4つ目のグラフはまったく同一であることに注意してください。底の変換公式より、 $\log_4 x = \log_2 x \div \log_2 4 = (\log_2 x)/2$ が成り立ちます。



問題 2.3.4

この問題は、指数法則（→2.3.9項）の理解を問う問題です。答えは以下の通りです。

1. $f(x) = 2^x$ のとき $f(20) = 2^{20} = 1048576$
2. 指数法則より $2^{20} = 2^{10} \times 2^{10}$ である。 2^{10} がおよそ 1000 ということは、 2^{20} はおよそ $1000 \times 1000 = 1000000 (= 10^6)$ である。

問題 2.3.5 (1)

この問題は、対数関数（→**2.3.10項**）の理解を問う問題です。

$10^6 = 1000000$ より、 $g(1000000) = \log_{10} 1000000 = 6$ となります。

問題 2.3.5 (2)

この問題は、対数関数の公式（→**2.3.10項**）の理解を問う問題です。

$$\log_2 16N - \log_2 N$$

$$= \log_2 \left(\frac{16N}{N} \right)$$

$$= \log_2 16 = 4$$

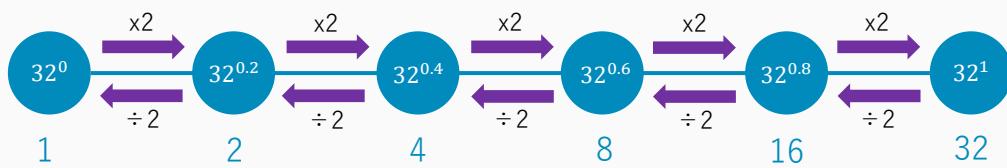
より、答えは **4** となります。なお、対数関数 $\log_a b$ には「真数 b が定数倍（2 倍など）されると値が一定だけ増える」という性質があります。

問題 2.3.6

この問題は、指数が小数の場合のべき乗の公式（→**2.3.8項**）、指数法則（→**2.3.9項**）を使いこなせるかどうかを問う問題です。それぞれの問題の答えは以下のようになります。

番号	マグニチュード	差が何倍か？	答え
1.	6.0 vs 5.0	$32^{6.0-5.0} = 32^{1.0}$	= 32 倍
2.	7.3 vs 5.3	$32^{7.3-5.3} = 32^{2.0}$	= 1024 倍
3.	9.0 vs 7.2	$32^{9.0-7.2} = 32^{1.8}$	= 512 倍

なお、 $32^{1.8}$ の値は、 $32^{1.0} \times 32^{0.8} = 32 \times 16 = 512$ と計算することができます。また、 $32^{0.8}$ などは 2.3.8 項の図を見れば分かります。



問題 2.3.7

この問題の答えは $y = \lfloor \log_2 x \rfloor + 1$ です。これは以下のようにして導出できます。

ステップ 1

ある整数 x が 2 進法で n 桁となるための条件は、 $2^{n-1} \leq x < 2^n$ を満たすことである。具体例は以下の通りである。

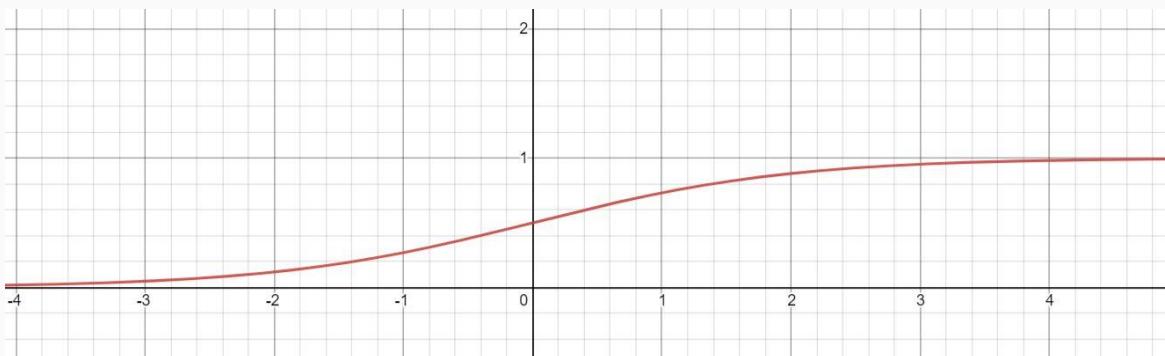
- 3 桁となる条件は、 $2^2 = 4$ 以上 $2^3 = 8$ 未満であること
- 4 桁となる条件は、 $2^3 = 8$ 以上 $2^4 = 16$ 未満であること
- 5 桁となる条件は、 $2^4 = 16$ 以上 $2^5 = 32$ 未満であること

ステップ 2

ステップ 1 を言い換えると、 $n - 1 \leq \log_2 x < n$ のとき、2 進法で n 桁となります。すなわち $\lfloor \log_2 x \rfloor = n - 1$ であるため、2 進法での桁数 n は $\lfloor \log_2 x \rfloor + 1$ 桁です。

問題 2.3.8

答えの例として、たとえば $f(x) = 1/(1 + 2^{-x})$ などが考えられます。なお、似たような関数として、機械学習などでよく使われる**シグモイド関数**があるので、興味のある人はぜひ調べてみましょう。



2.4

節末問題 2.4 の解答

問題 2.4.1

答えは以下の通りです。なお、 O 記法で表した値は、「最も重要な項を消した後、定数倍 ($7N^2$ の 7 の部分) を消す」という操作で求められます（→2.4.8項）。

1. $T_1(N) = O(N^3)$
2. $T_2(N) = O(N)$
3. $T_3(N) = O(2^N)$
4. $T_4(N) = O(N!)$

問題 2.4.2

このプログラムは二重ループを行っており、各変数は以下のような値をとります。

- 変数 $i : 1, 2, 3, \dots, N$ の N 通り
- 変数 $j : 1, 2, 3, \dots, 100N$ の $100N$ 通り

したがって、合計ループ回数は $N \times 100N = 100N^2$ 回であり、すなわち計算量は $O(N^2)$ となります。なお、ループ回数が掛け算で表される理由は、変数 i 、 j の取り方を長方形形状に並べると理解しやすいです（→2.4.5項）。



問題 2.4.3

$\log_2 N$ と $\log_{10} N$ が定数倍の差しかないことを確認するため、 $\log_2 N$ を $\log_{10} N$ で割ってみましょう。底の変換公式（→2.3.10項）より、次式が成り立ちます。

$$\frac{\log_2 N}{\log_{10} N} = \frac{\log_2 N}{\log_2 N \div \log_2 10} = \log_2 10 \approx 3.32$$

したがって、 $\log_2 N$ は $\log_{10} N$ の約 3.32 倍であることが分かります。このようなことが、対数を O 記法で表すときに $O(\log N)$ と底を省略した表記を使う理由の一つになっています。

問題 2.4.4

答えは以下のようになります。なお、 $N \log N$ は $N \times \log N$ と同じ意味です。

計算回数	$N \log N$	N^2	2^N
10^6 回以内	$N \leq 60000$	$N \leq 1000$	$N \leq 20$
10^7 回以内	$N \leq 500000$	$N \leq 3000$	$N \leq 23$
10^8 回以内	$N \leq 4000000$	$N \leq 10000$	$M \leq 26$
10^9 回以内	$N \leq 40000000$	$N \leq 30000$	$N \leq 30$

問題 2.4.5

N が 2 増えると実行時間がおよそ 9 倍になっているため、計算量は $O(3^N)$ だと考えるのが自然です。なお、 $O(N \times 3^N)$ や $O(10^{N/2})$ なども不自然ではないため、別解として扱います。

N	14	16	18	20
実行時間	0.049 秒	0.447 秒	4.025 秒	36.189 秒

9.12 倍 9.00 倍 8.99 倍

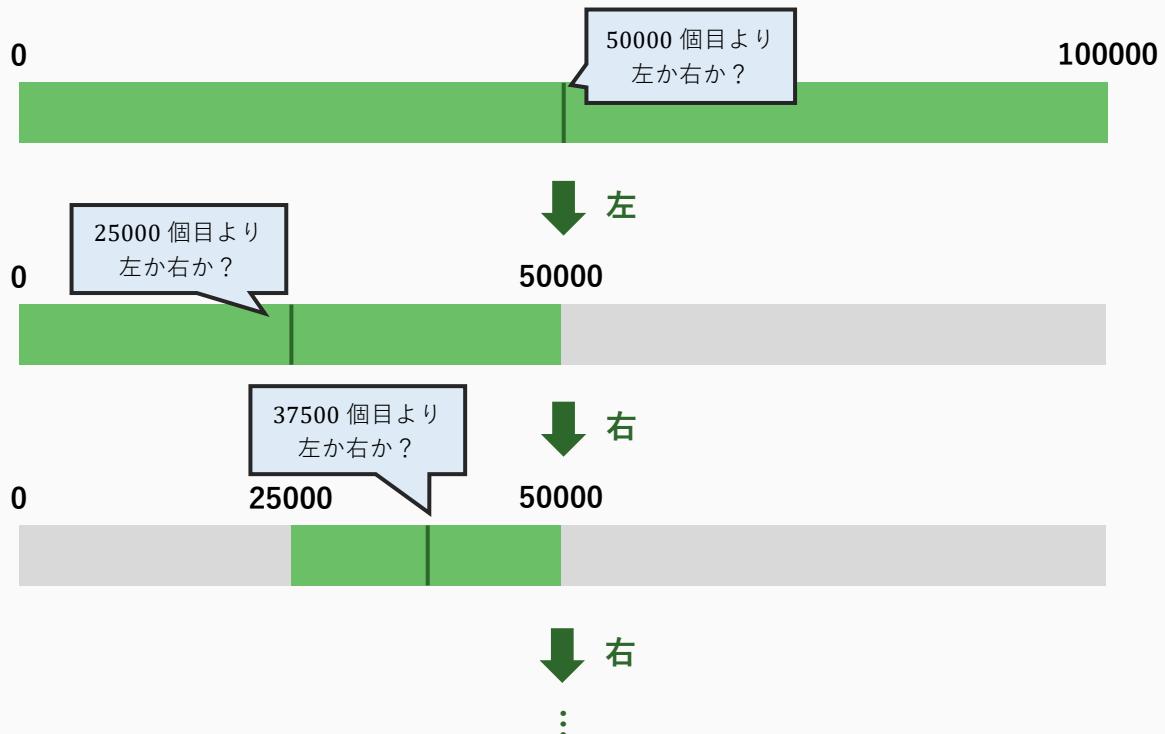
問題 2.4.6

直観的な方法として、“a” → “aardvark” → “aback” → “abalone” → “abandon” → ...といった感じで、前に載っている単語から 1 つずつ調べていくことが考えられます。しかし、単語数を N とするときのステップ数は最悪 N 回です。 $N = 100000$ もあれば、人間にはとても無理があります。そこで、たとえば以下の方法を使えば効率的です（→2.4.7項）。

「現時点を考えられる範囲の中央の単語を見て、それより前にあるか後ろにあるかを調べる」ことを繰り返す。下図は単語数が 100000 個の場合の手順のイメージを示している。

これは二分探索法と非常に似た手法であり、わずか $\lceil \log_2 N \rceil$ ステップで目的の単語を見つけることができます。

なお、実用上は「50000 個目の単語がどこにあるか」といったことを調べるのも面倒なので、たとえば最初の質問では、だいたい中央のページにある単語と比べれば良いです。皆さんも辞書で単語を調べるとき、ぜひ一度二分探索を使ってみましょう。



2.5

節末問題 2.5 の解答

問題 2.5.1

この問題は、シグマ記号（→2.5.9項）の理解を問う問題です。

答えは以下の通りになります。

$$\sum_{i=1}^{100} i = (1 + 2 + 3 + \dots + 100) = 5050$$
$$\sum_{i=1}^3 \sum_{j=1}^3 ij = (1 + 2 + 3 + 2 + 4 + 6 + 3 + 6 + 9) = 36$$

なお、1から100までの総和は1.1節にも記されていますが、和の公式（→2.5.10項）を用いて $100 \times 101 \div 2 = 5050$ と計算することもできます。

問題 2.5.2

集合の基本（→2.5.5項）の理解を問う問題です。答えは以下のようになります。

1. $|S| = 3, |T| = 4$
2. $S \cup T = \{2, 3, 4, 7, 8, 9\}$ (一方に含まれる部分)
3. $S \cap T = \{2\}$ (両方に含まれる部分)
4. 空でない部分集合は $\{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\}, \{2, 4, 7\}$ の7つ

分からぬ人は、58ページに戻って確認しましょう。

問題 2.5.3

階乗 $N! = 1 \times 2 \times 3 \times \dots \times N$ であるため、for文を用いて掛け算を行うプログラムを書けば良いです。なお、 $N = 20$ のとき $N! = 2.4 \times 10^{18}$ 程度となり、int型などの32ビット整数ではオーバーフローを起こすことに注意してください。（次のソースコードでは、代わりにlong long型を使っています）

```
#include <iostream>
using namespace std;
```

```

int main() {
    long long N;
    long long Answer = 1;
    cin >> N;
    for (int i = 2; i <= N; i++) Answer *= i; // Answer に i を掛ける
    cout << Answer << endl;
    return 0;
}

```

※ Python などのソースコードは chap2-5.md をご覧ください。

問題 2.5.4

以下のようなプログラムを書くと正解が得られます。なお、関数 `isprime(x)` は 2 以上の整数 x が素数であるかどうかを判定する関数であり、素数の場合 `true`、そうでない場合 `false` を返します。また、

- x は 2 で割り切れますか？
- x は 3 で割り切れますか？
- :
- x は $N - 1$ で割り切れますか？

といった感じで 1 つずつ調べていくことで、素数判定を行っています。

```

#include <iostream>
using namespace std;

bool isprime(int x) {
    for (int i = 2; i <= x - 1; i++) {
        // x を i で割った余りが 0 のとき、x は i で割り切れる
        if (x % i == 0) return false;
    }
    return true;
}

int main() {
    int N, Answer = 0;
    cin >> N;
    for (int i = 2; i <= N; i++) {
        if (isprime(i) == true) cout << i << endl;
    }
    return 0;
}

```

※ Python などのソースコードは chap2-5.md をご覧ください。

問題 2.5.5

この問題の答えは **1000** です。

最も単純な方法として、 $1 \leq a \leq 4, 1 \leq b \leq 4, 1 \leq c \leq 4$ に含まれる整数の組 (a, b, c) すべてについて計算する方法がありますが、これでは面倒です。

a = 1 のとき 合計 100			
1	2	3	4
2	4	6	8
3	6	9	12
4	8	12	16

a = 2 のとき 合計 200			
2	4	6	8
4	8	12	16
6	12	18	24
8	16	24	32

a = 3 のとき 合計 300			
3	6	9	12
6	12	18	24
9	18	27	36
12	24	36	48

a = 4 のとき 合計 400			
4	8	12	16
8	16	24	32
12	24	36	48
16	32	48	64

そこで、以下の二重シグマの値を考えましょう。合計は **100** です。

$$\sum_{b=1}^4 \sum_{c=1}^4 bc = 100$$

各 a における abc の合計を考えると、以下のようにになります。

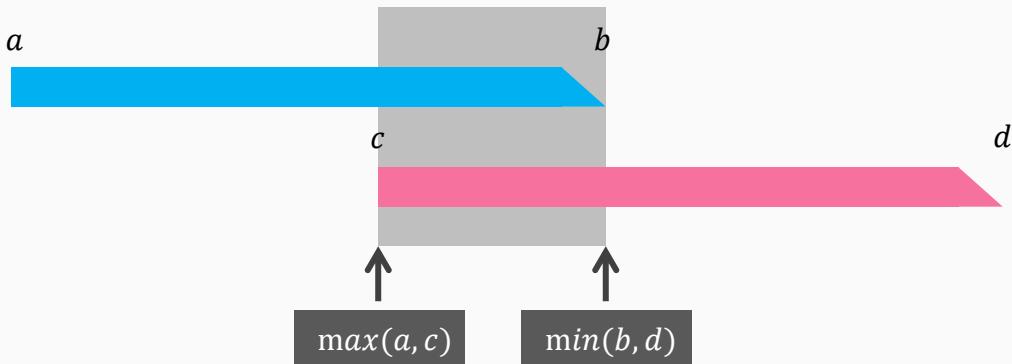
- $a = 1$ のときの $abc (= 1 \times bc)$ の合計 : $1 \times 100 = 100$
- $a = 2$ のときの $abc (= 2 \times bc)$ の合計 : $2 \times 100 = 200$
- $a = 3$ のときの $abc (= 3 \times bc)$ の合計 : $3 \times 100 = 300$
- $a = 4$ のときの $abc (= 4 \times bc)$ の合計 : $4 \times 100 = 400$

求める三重シグマは、上の 4 つをすべて足した値 **1000** となります。

問題 2.5.6

共通部分を持つことの必要十分条件は、 $\max(a, c) < \min(b, d)$ を満たすことです。

`max` 関数、`min` 関数が分からぬ人は、2.3.2 項に戻って確認しましょう。



問題 2.5.7

それぞれの `i` の値における「`cnt` が増える回数」は以下の通りです。

- `i=1` のとき : $2 \leq j \leq N$ なので $N - 1$ 回
- `i=2` のとき : $3 \leq j \leq N$ なので $N - 2$ 回
- ⋮
- `i=N-1` のとき : 1 回
- `i=N` のとき : 0 回

よって、実行終了時の `cnt` の値は、和の公式（→2.5.10項）より、

$$(N - 1) + (N - 2) + \cdots + 1 + 0 = \frac{(N - 1) \times N}{2} \left(= \frac{1}{2}N^2 - \frac{1}{2}N \right)$$

となります。`cnt` の値の中で最も重要な項は $(1/2) \times N^2$ であるため、このプログラムの計算量は $O(N^2)$ です。（→2.4.8項）

第3章

基本的なアルゴリズム

3.1

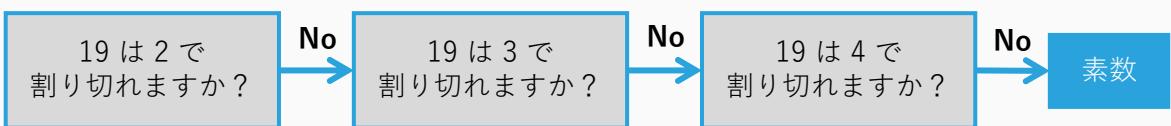
節末問題 3.1 の解答

問題 3.1.1

自分の年齢 N が 2 歳以上の場合、以下のようにして判定すれば良いです。

- 2 以上 \sqrt{N} 以下の整数で割り切れない場合：素数
- そうでない場合：合成数

たとえば 19 歳の場合は、 $\sqrt{19} = 4.358 \dots$ より 2, 3, 4 で割れば良いですが、いずれでも割り切れないため素数です。



問題 3.1.2

まず、自然数 N を素因数分解したときに、 \sqrt{N} を超えるものは高々 1 つしかありません。これは背理法（→3.1.3項）を用いて次のように証明することができます。

自然数 N を素因数分解したときに \sqrt{N} を超えるものが 2 つ以上あると仮定する。すなわち、 \sqrt{N} を超える整数 A, B を用いて、以下のように素因数分解されると仮定する。

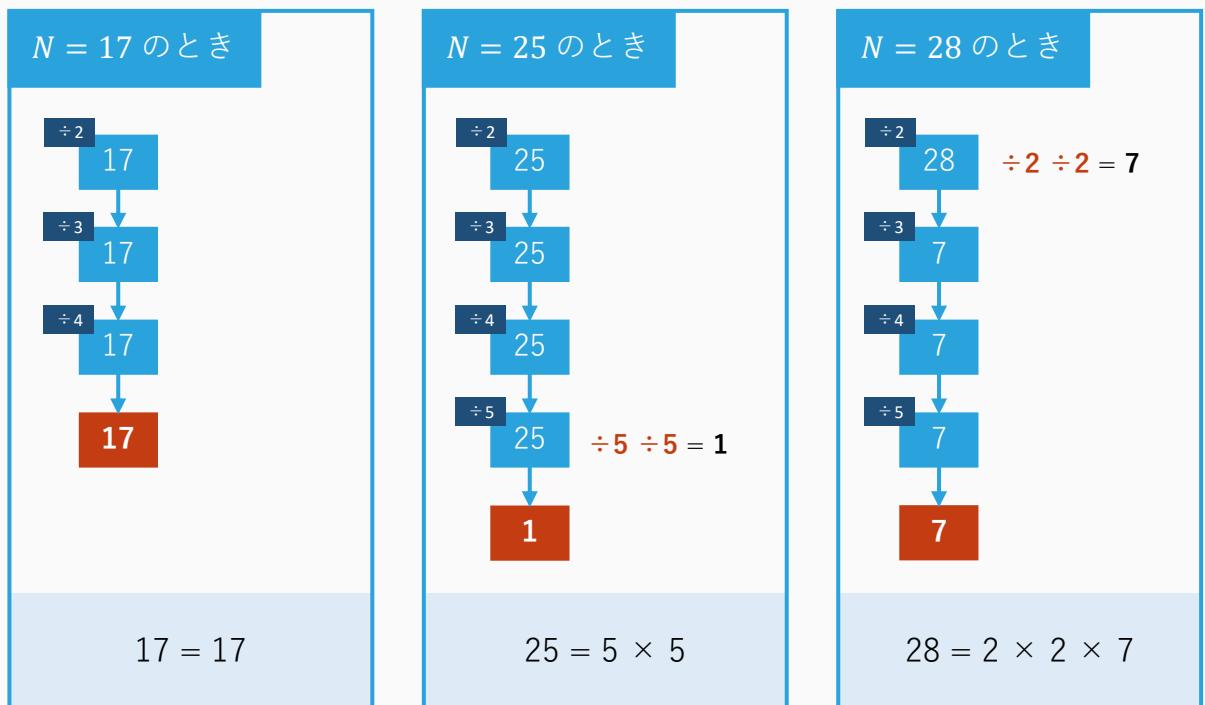
$$N = \bigcirc \times \bigcirc \times \cdots \times \bigcirc \times A \times B$$

しかし、 $A \times B > N$ であるため矛盾する。よって \sqrt{N} を超えるものは 1 つ。

したがって、次のようなアルゴリズムで素因数分解することができます。

- N を 2 で割れるだけ割る。
- N を 3 で割れるだけ割る。
- 同じような操作を、 $4, 5, \dots, \lfloor \sqrt{N} \rfloor$ についても行う。
- 最後に残った N が 1 以外の場合、それを素因数に加える。

たとえば、 $N = 17, 25, 28$ の場合、下図のようにアルゴリズムが動作します。同じ数を二度以上割るケースもあることに注意してください。



C++ での実装例は以下の通りです。

```
#include <iostream>
using namespace std;

int main() {
    // 入力
    long long N;
    cin >> N;

    // 素因数分解・出力
    for (long long i = 2; i * i <= N; i++) {
        while (N % i == 0) {
            N /= i;
            cout << i << endl;
        }
    }
    if (N >= 2) cout << N << endl;
    return 0;
}
```

※ Pythonなどのソースコードは chap3-1.md をご覧ください。

3.2

節末問題 3.2 の解答

問題 3.2.1

答えは以下のようになります。分からない人は、[3.2.2項](#)に戻って確認しましょう。

ステップ数	0	1	2	3	4	5	6
A の値	372	372	104	104	14	14	0
B の値	506	134	134	30	30	2	2

問題 3.2.2

整数 A_1, A_2, \dots, A_N の最大公約数は、以下のようにして計算できます。 ([→3.2.5項](#))

- まず、 A_1 と A_2 の最大公約数を計算する。
- 次に、前の計算結果と A_3 の最大公約数を計算する。
- :
- 次に、前の計算結果と A_N の最大公約数を計算する。

これを実装すると、以下のようになります。なお、関数 $\text{GCD}(A, B)$ は A と B の最大公約数を計算する関数です。また、変数 r は前の計算結果を表します。

```
#include <iostream>
#include <algorithm>
using namespace std;

long long GCD(long long A, long long B) {
    while (A >= 1 && B >= 1) {
        if (A < B) B = B % A; // A < B の場合、大きい方 B を書き換える
        else A = A % B; // A >= B の場合、大きい方 A を書き換える
    }
    if (A >= 1) return A;
    return B;
}

long long N;
long long A[100009];

int main() {
```

```

// 入力
cin >> N;
for (int i = 1; i <= N; i++) cin >> A[i];

// 答えを求める
long long R = GCD(A[1], A[2]);
for (int i = 3; i <= N; i++) {
    R = GCD(R, A[i]);
}

// 出力
cout << R << endl;
return 0;
}

```

※ Python などのソースコードは chap3-2.md をご覧ください。

問題 3.2.3

整数 A_1, A_2, \dots, A_N の最小公倍数は、以下のようにして計算できます。

- まず、 A_1 と A_2 の最小公倍数を計算する。
- 次に、前の計算結果と A_3 の最小公倍数を計算する。
- ⋮
- 次に、前の計算結果と A_N の最小公倍数を計算する。

また、2つの数 A, B には以下の性質が成り立ちます。 (→2.5.2 項)

$$A \times B = (\text{A と B の最大公約数}) \times (\text{A と B の最小公倍数})$$

$$\text{すなわち} \quad (\text{A と B の最小公倍数}) = \frac{A \times B}{(\text{A と B の最大公約数})}$$

したがって、以下のようなプログラムを書くと、正解が得られます。なお、関数 $\text{LCM}(A, B)$ は A と B の最小公倍数を計算する関数です。

```

#include <iostream>
#include <algorithm>
using namespace std;

long long GCD(long long A, long long B) {
    while (A >= 1 && B >= 1) {
        if (A < B) B = B % A; // A < B の場合、大きい方 B を書き換える
        else A = A % B; // A >= B の場合、大きい方 A を書き換える
    }
    return A;
}

long long LCM(long long A, long long B) {
    return (A * B) / GCD(A, B);
}

```

```
    }
    if (A >= 1) return A;
    return B;
}

long long LCM(long long A, long long B) {
    return (A / GCD(A, B)) * B;
}

long long N;
long long A[100009];

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 答えを求める
    long long R = LCM(A[1], A[2]);
    for (int i = 3; i <= N; i++) {
        R = LCM(R, A[i]);
    }

    // 出力
    cout << R << endl;
    return 0;
}
```

※ Python などのソースコードは chap3-2.md をご覧ください。

3.3

節末問題 3.3 の解答

問題 3.3.1

この問題は、場合の数の公式（→3.3.4項、3.3.5項）の理解を問う問題です。答えは、

$${}_2C_1 = \frac{2}{1} = 2$$

$${}_8C_5 = \frac{8 \times 7 \times 6 \times 5 \times 4}{5 \times 4 \times 3 \times 2 \times 1} = 56$$

$${}_7P_2 = 7 \times 6 = 42$$

$${}_{10}P_3 = 10 \times 9 \times 8 = 720$$

となります。なお、二項係数 nCr は nPr の $1/r!$ 倍であるため、

$$nCr = \frac{nPr}{r!} = \frac{n \times (n-1) \times (n-2) \times (n-r+1)}{r \times (r-1) \times (r-2) \times \cdots \times 1}$$

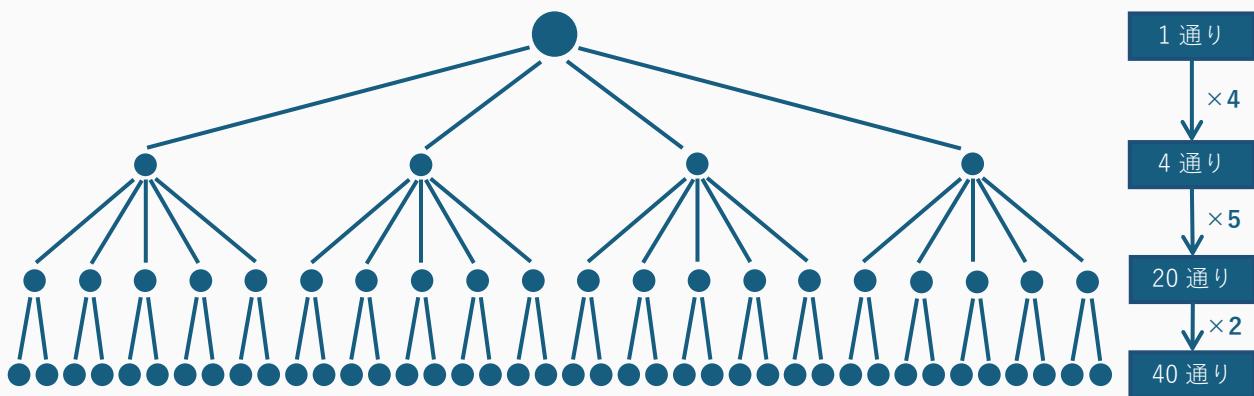
と計算することができます。

問題 3.3.2

この問題は、積の法則（→3.3.2項）を理解を問う問題です。

- 大きさの選び方： 4 通り
- トッピングの選び方： 5 通り
- ネームプレートの選び方： 2 通り

あるため、答えは $4 \times 5 \times 2 = 40$ 通り となります。以下はイメージ図となります。



問題 3.3.3

まず、以下の値を計算しましょう。（ $n!$ を計算する方法：→[節末問題 2.5.3](#)）

- Fact_n : $n!$ の値
- Fact_r : $r!$ の値
- Fact_nr : $(n - r)!$ の値

このとき、求める nCr の値は $\text{Fact_n} / (\text{Fact_r} * \text{Fact_nr})$ となるため、この値を出力するプログラムを書けば正解となります。C++ での実装例は次の通りです。

```
#include <iostream>
using namespace std;

long long n, r;
long long Fact_n = 1;
long long Fact_r = 1;
long long Fact_nr = 1;

int main() {
    // 入力
    cin >> n >> r;

    // 階乗の計算
    for (int i = 1; i <= n; i++) Fact_n *= i;
    for (int i = 1; i <= r; i++) Fact_r *= i;
    for (int i = 1; i <= n - r; i++) Fact_nr *= i;

    // 出力
    cout << Fact_n / (Fact_r * Fact_nr) << endl;
    return 0;
}
```

※ Python などのソースコードは chap3-3.md をご覧ください。

問題 3.3.4

書籍で解説された通りに実装すれば良いです。以下は C++ での解答例となります。なお、この問題は制約が $N \leq 200000$ と大きく、答えが 10^{10} 通りになる可能性があります。`int` 型などの 32 ビット整数ではオーバーフローを起こすことに注意してください。（Python の場合は関係ありません）

```
#include <iostream>
using namespace std;

long long N;
```

```

long long A[200009];
long long a = 0, b = 0, c = 0, d = 0; // オーバーフロー回避のため 64 ビット整数を使う

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // a, b, c, d の個数を数える
    for (int i = 1; i <= N; i++) {
        if (A[i] == 100) a += 1;
        if (A[i] == 200) b += 1;
        if (A[i] == 300) c += 1;
        if (A[i] == 400) d += 1;
    }

    // 出力 (答えは a * d + b * c)
    cout << a * d + b * c << endl;
    return 0;
}

```

※ Python などのソースコードは chap3-3.md をご覧ください。

問題 3.3.5

書籍で解説された通りに実装すれば良いです。以下は C++ での解答例となります。なお、この問題は制約が $N \leq 500000$ と大きく、答えが 10^{11} 通り以上になる可能性があります。

`int` 型などの 32 ビット整数ではオーバーフローを起こすため、`long long` 型などの 64 ビット整数を使うことが推奨されます。（Python の場合は関係ありません）

```

#include <iostream>
using namespace std;

long long N;
long long A[500009];
long long x = 0, y = 0, z = 0;

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // a, b, c, d の個数を数える
    for (int i = 1; i <= N; i++) {
        if (A[i] == 1) x += 1;
        if (A[i] == 2) y += 1;
    }
}

```

```

    if (A[i] == 3) z += 1;
}

// 出力
cout << x * (x - 1) / 2 + y * (y - 1) / 2 + z * (z - 1) / 2 << endl;
return 0;
}

```

※ Python などのソースコードは chap3-3.md をご覧ください。

問題 3.3.6

まず、 A_1, A_2, \dots, A_N の中に i が何個存在するかを $\text{cnt}[i]$ とするとき、 $\text{cnt}[1], \text{cnt}[2], \dots, \text{cnt}[99999]$ は以下のようにして数えることができます。

```

// 配列 cnt[i] を 0 に初期化する
for (int i = 1; i <= N; i++) cnt[i] = 0;

// A[i] が現れたら cnt[A[i]] に 1 を加算する
for (int i = 1; i <= N; i++) cnt[A[i]] += 1;

```

そこで、和が 100000 となる 2 枚のカードの選び方をリストアップすると、

- 1 と 99999 のカードを選ぶ ($\text{cnt}[1]*\text{cnt}[99999]$ 通り)
- 2 と 99998 のカードを選ぶ ($\text{cnt}[2]*\text{cnt}[99998]$ 通り)
- 3 と 99997 のカードを選ぶ ($\text{cnt}[3]*\text{cnt}[99997]$ 通り)
- :
- 49999 と 50001 のカードを選ぶ ($\text{cnt}[49999]*\text{cnt}[50001]$ 通り)
- 50000 のカードを 2 枚選ぶ ($\text{cnt}[50000]*(\text{cnt}[50000]-1)/2$ 通り)

となります。50000 のカードを 2 枚選んだとき、 $\text{cnt}[50000]*\text{cnt}[50000]$ 通りにならないことに注意してください。

したがって、赤く記された値の合計を出力するプログラムを書くと、正解が得られます。以下のプログラムは、C++ での解答例です。

```

#include <iostream>
using namespace std;

long long N, A[200009];
long long cnt[100009];
long long Answer = 0;

int main() {

```

```

// 入力
cin >> N;
for (int i = 1; i <= N; i++) cin >> A[i];

// cnt[1], cnt[2], ..., cnt[99999] を数える
for (int i = 1; i <= 99999; i++) cnt[i] = 0;
for (int i = 1; i <= N; i++) cnt[A[i]] += 1;

// 答えを求める
for (int i = 1; i <= 49999; i++) {
    Answer += cnt[i] * cnt[100000 - i];
}
Answer += cnt[50000] * (cnt[50000] - 1) / 2;

// 出力
cout << Answer << endl;
return 0;
}

```

※ Python などのソースコードは chap3-3.md をご覧ください。

問題 3.3.7

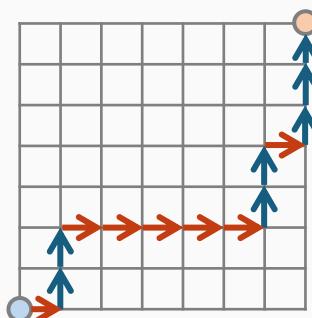
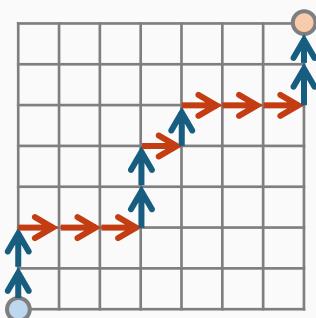
まず、スタートからゴールまで最短距離（14 手）で行くための必要十分条件（→**2.5.6項**）は以下のようになります。

「上方向に 1 つ移動すること」「右方向に 1 つ移動すること」を 7 回ずつ行う。それ以外の移動は行わない。

したがって、答えは 14 回のうち 7 回上方向を選ぶ場合の数、すなわち

$${}_{14}C_7 = \frac{14!}{7! \times 7!} = 3432 \text{ 通り}$$

となります。直接手計算で求めるのは面倒ですが、節末問題 3.3.3 のプログラムに $n = 14, r = 7$ を代入すると、簡単に答えが分かります。



上方向の移動 7 回
右方向の移動 7 回

3.4

節末問題 3.4 の解答

問題 3.4.1

21 個のパターンすべてが等確率で起こるとは限らないからです。実際、

- (1, 1) の目が出る確率は $1/36$
- (1, 2) の目が出る確率は $1/18$

と異なるため、「21 個中 15 個のパターンで出目の和が 8 以下となるから、求める確率は $15/21$ 」と単純に考えることはできません。

なお、★で示した部分は以下のようにして理解することができます。

サイコロを 2 個振ったとき、(1, 2) の目が出るパターンは

- 一回目に振ったものが 1 であり、二回目に振ったものが 2
- 二回目に振ったものが 2 であり、一回目に振ったものが 1

の 2 つあります。36 個中 2 個なので、確率は $2/36 = 1/18$ です。



一回目に振った
サイコロ



二回目に振った
サイコロ

問題 3.4.2

期待値の公式（→3.4.2項）に代入すると、答えは以下のようになります。

$$\begin{aligned} & \left(1000000 \times \frac{1}{10000}\right) + \left(100000 \times \frac{9}{10000}\right) + \left(10000 \times \frac{9}{1000}\right) + \left(1000 \times \frac{9}{100}\right) + \left(0 \times \frac{9}{10}\right) \\ & = 100 + 90 + 90 + 90 \\ & = 370 \end{aligned}$$

もらえる賞金の期待値が 370 円なので、参加費が 500 円の場合は損です。

問題 3.4.3

まず、期待値の線形性（→3.4.3項）より、次の関係が成り立ちます。

（合計勉強時間の期待値）

$$= (1 \text{ 日目の勉強時間の期待値}) + \cdots + (N \text{ 日目の勉強時間の期待値})$$

そこで、それぞれの日の勉強時間の期待値は、以下の通りです。

- 1 日目 : $(A_1 \times 1/3) + (B_1 \times 2/3)$
- 2 日目 : $(A_2 \times 1/3) + (B_2 \times 2/3)$
- 3 日目 : $(A_2 \times 1/3) + (B_2 \times 2/3)$
- :
- N 日目 : $(A_2 \times 1/3) + (B_2 \times 2/3)$

したがって、求める合計勉強時間の期待値は、以下のようになります。

$$\left(A_1 \times \frac{1}{3} + B_1 \times \frac{2}{3}\right) + \left(A_2 \times \frac{1}{3} + B_2 \times \frac{2}{3}\right) + \cdots + \left(A_N \times \frac{1}{3} + B_N \times \frac{2}{3}\right)$$

この値を出力するプログラムを書くと、正解となります。以下は C++ の解答例です。

```
#include <iostream>
using namespace std;

int N;
double A[109], B[109];
double Answer = 0.0;

int main() {
```

```

// 入力
cin >> N;
for (int i = 1; i <= N; i++) cin >> A[i] >> B[i];

// 期待値を求める
for (int i = 1; i <= N; i++) {
    double eval = A[i] * (1.0 / 3.0) + B[i] * (2.0 / 3.0);
    Answer += eval;
}

// 出力
printf("%.12lf\n", Answer);
return 0;
}

```

※ Python などのソースコードは chap3-4.md をご覧ください。

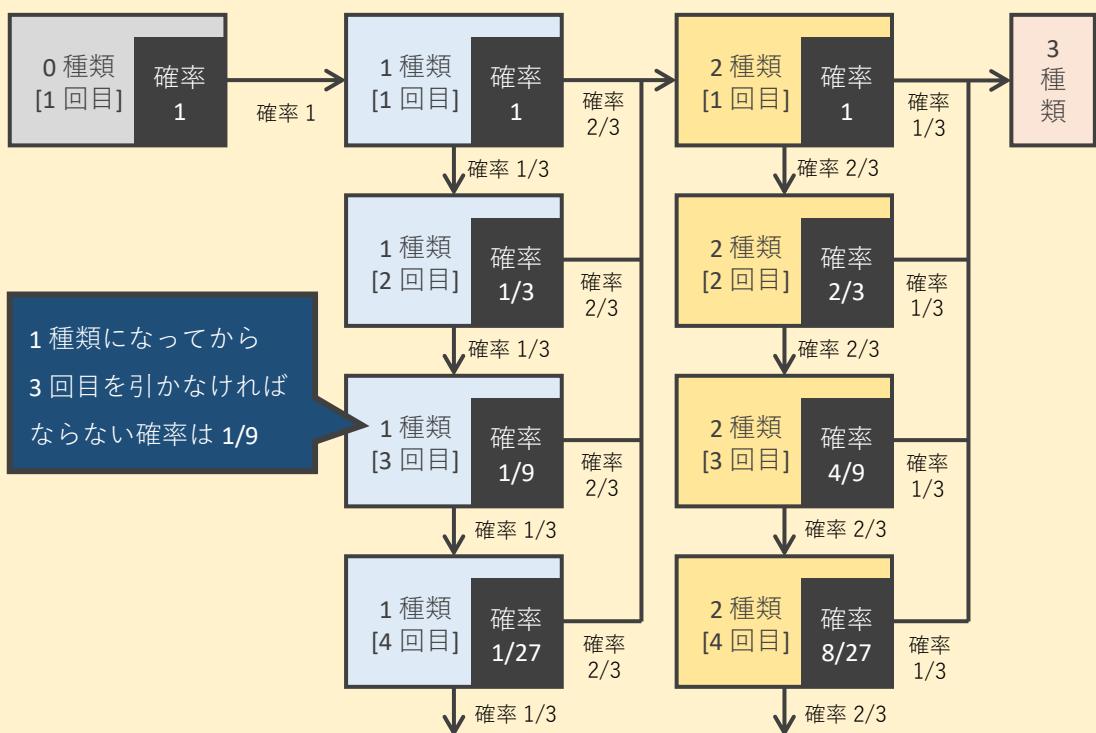
問題 3.4.4

この問題は設定が複雑で難しいので、まずは $N = 3$ の場合を考えてみましょう。

まず、全部のコインを集めるまでの過程を表した図は以下のようになります。

- 実線は、まだ集めていない種類のコインを得たこと
- 点線は、既に集めた種類のコインを引いてしまったこと

を示しています。



そこで、期待値の線形性より、全種類を集めるのにかかる回数の期待値は、

- 0 種類から 1 種類にするのにかかる回数の期待値 … (1)
- 1 種類から 2 種類にするのにかかる回数の期待値 … (2)
- 2 種類から 3 種類にするのにかかる回数の期待値 … (3)

の総和となります。

また、前ページの図と和の公式（→**2.5.10 項の最後**）より、

- (1) の期待値は 1 回
- (2) の期待値は $1 + (1/3) + (1/9) + \dots = 3/2$ 回
- (3) の期待値は $1 + (2/3) + (4/9) + \dots = 3$ 回

となり、 $N = 3$ の場合の答えは $1 + 3/2 + 3 = \textcolor{red}{11/2}$ 回と分かります。

次に、一般の N の場合について考えましょう。期待値の線形性より、求める答えは以下の値をすべて足したものとなります。

- 0 種類から 1 種類にするのにかかる回数の期待値
- 1 種類から 2 種類にするのにかかる回数の期待値
- :
- $N - 1$ 種類から N 種類にするのにかかる回数の期待値

そこで、すでに r 種類のコインが集まっている状態から $r + 1$ 種類目のコインを集めると、既に集めているコインを取る確率は r/N ですから、

- 1 回以上かかる確率： 1
- 2 回以上かかる確率： $(r/N)^1$
- 3 回以上かかる確率： $(r/N)^2$
- 4 回以上かかる確率： $(r/N)^3$ [以下略]

となります。よって、回数の期待値は以下のようになります。

$$1 + \left(\frac{r}{N}\right)^1 + \left(\frac{r}{N}\right)^2 + \left(\frac{r}{N}\right)^3 + \dots = \frac{\textcolor{red}{N}}{\textcolor{red}{N-r}}$$

最後に、全体の回数の期待値は $r = 0, 1, 2, \dots, N - 1$ について赤い部分を足した

$$\frac{\textcolor{red}{N}}{N} + \frac{\textcolor{red}{N}}{N-1} + \frac{\textcolor{red}{N}}{N-2} + \dots + \frac{N}{2} + \frac{N}{1}$$

となります。

したがって、この値を出力する以下のようなプログラムを書くと、正解が得られます。

```
#include <iostream>
using namespace std;

int N;
double Answer = 0;

int main() {
    // 入力
    cin >> N;

    // 期待値を求める
    for (int i = N; i >= 1; i--) {
        Answer += 1.0 * N / i;
    }

    // 出力
    printf("%.12lf\n", Answer);
    return 0;
}
```

※ Python などのソースコードは chap3-4.md をご覧ください。

3.5

節末問題 3.5 の解答

問題 3.5.1 (1), (2)

表が出る確率が $p = 0.5$ 、試行回数が $n = 10000$ なので、3.5.6 項で述べた公式に代入すると、10000 回のうち表が出た割合の分布は、

- 平均 : $p = 0.5$
- 標準偏差 : $\sqrt{p(1-p)/n} = \sqrt{0.5 \times (1 - 0.5) \div 10000} = 0.005$

の正規分布に近似できます。回数に換算すると、平均 $\mu = 5000$ 回、標準偏差 $\sigma = 50$ 回となります。

そこで、 $\mu - 2\sigma = 4900, \mu + 2\sigma = 5100$ より、回数が 4900 回以上 5100 回以下となる確率は約 95% です（68 – 95 – 99.7 則 : →3.5.5項）。なお、平均と標準偏差は、以下の公式から直接計算することもできます。

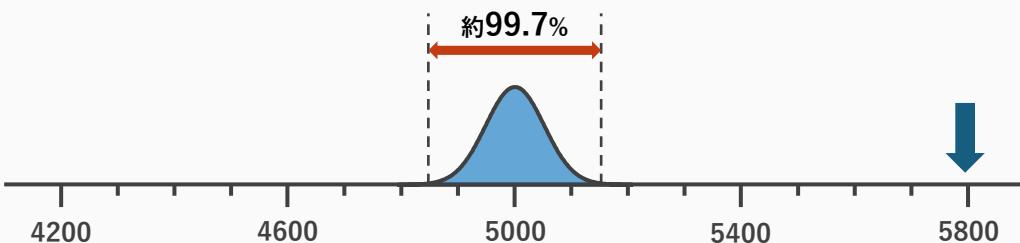
確率 p で成功する試行を n 回行ったとき、成功した回数の分布は、平均 $\mu = np$ 、標準偏差 $\sigma = \sqrt{np(1-p)}$ の正規分布に近似できます。

問題 3.5.1 (3)

(1), (2) の結果から、約 99.7% の確率で表が出た回数が

- $\mu - 3\sigma = 5000 - 150 = 4850$ 回以上
- $\mu + 3\sigma = 5000 + 150 = 5150$ 回以下

となります。5800 回はこの範囲を大幅に逸脱しているため、コインが出る確率は 50% ではない可能性が高いといえます。

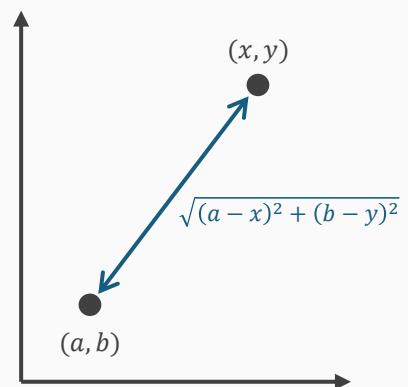


問題 3.5.2 (1)

以下のようなプログラムを書くと、ランダムに打った 100 万個の点のうち何個が 2 つの円のうち少なくとも一方に含まれたかを判定することができます。

たとえば著者環境では、このプログラムは **719653**※ と出力します。

なお、詳しくは 4.1 節で解説しますが、座標 (a, b) と座標 (x, y) の間の距離は $\sqrt{(a - x)^2 + (b - y)^2}$ となります。



```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    int N = 1000000;
    int M = 0;

    for (int i = 1; i <= N; i++) {
        double px = 6.0 * rand() / (double)RAND_MAX;
        double py = 9.0 * rand() / (double)RAND_MAX;

        // 点 (3, 3) との距離。この値が 3 以下であれば半径 3 の円に含まれる。
        double dist_33 = sqrt((px - 3.0) * (px - 3.0) + (py - 3.0) * (py - 3.0));

        // 点 (3, 7) との距離。この値が 2 以下であれば半径 2 の円に含まれる。
        double dist_37 = sqrt((px - 3.0) * (px - 3.0) + (py - 7.0) * (py - 7.0));

        // 条件分岐
        if (dist_33 <= 3.0 || dist_37 <= 2.0) M += 1;
    }

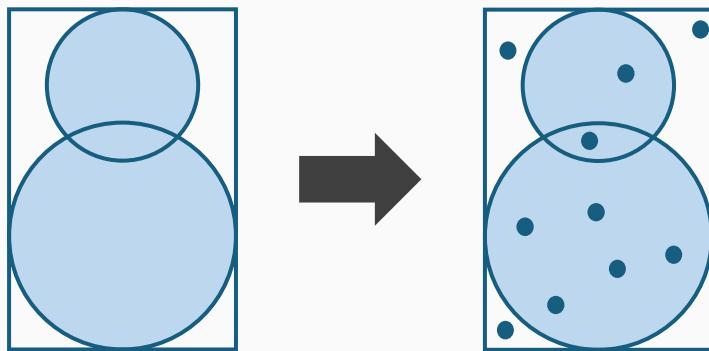
    // N 回中何回表に入ったかを出力
    cout << M << endl;
    return 0;
}
```

※ 著者環境では 719653 回ですが、718000～721000 回の範囲に入れば良いです。

※ Python などのソースコードは chap3-5.md をご覧ください。

問題 3.5.2 (2)

ランダムに点を打った領域 ($0 \leq x \leq 6, 0 \leq y \leq 9$) の面積は $6 \times 9 = 54$ であるため、下図の青色領域に入った点の割合を p とするとき、青色領域の面積は $54 \times p$ で近似することができます。



これを利用して面積を計算しましょう。たとえば著者環境での結果を使うと、 $54 \times 719653 \div 1000000 = 38.861262$ と計算されます。実際の値は約 $38.850912677 \dots$ であるため、0.02 以下の差におさまっています。

3.6

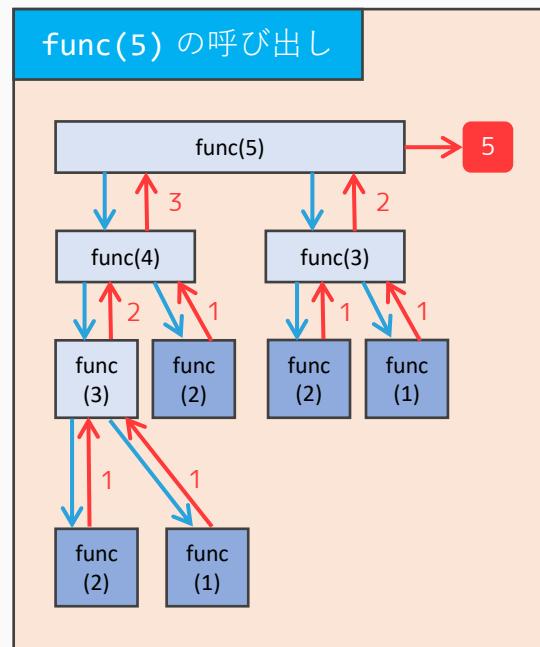
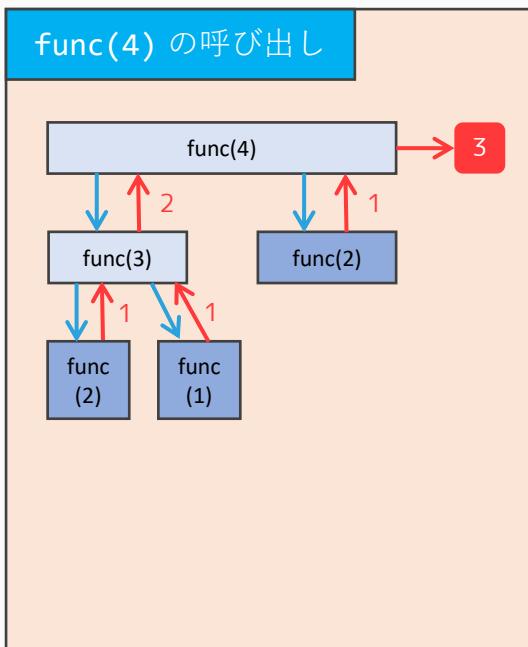
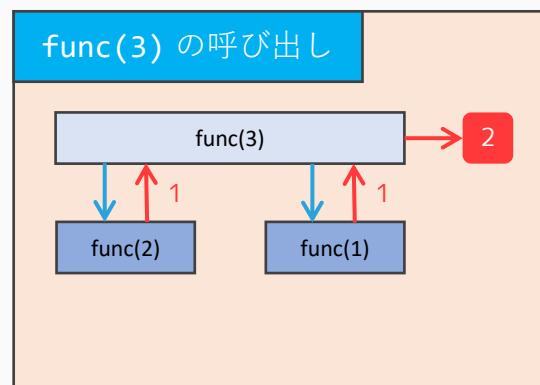
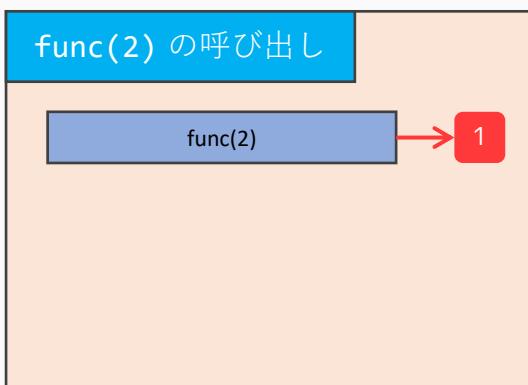
節末問題 3.6 の解答

問題 3.6.1

答えは以下の通りになります。特に `func(2)` では最初の条件分岐 ($N \leq 2$) で値が返されることに注意してください。

関数の呼び出し	<code>func(2)</code>	<code>func(3)</code>	<code>func(4)</code>	<code>func(5)</code>
答え	1	2	3	5

再帰呼び出しのイメージ図は以下のようになります。なお、関数 `func(N)` はフィボナッチ数（→3.7.2項）の第 N 項を返します。



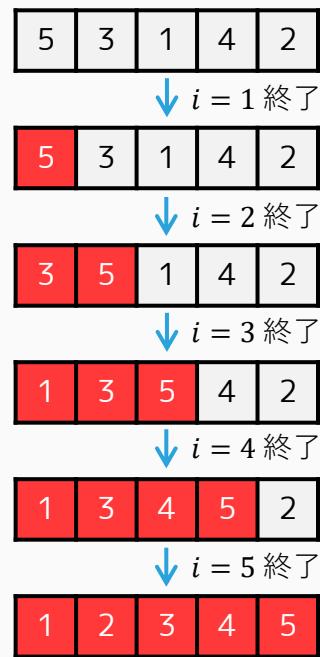
問題 3.6.2

このソートアルゴリズムでは、以下の事実 A が成り立ちます。右図は $A = [5, 3, 1, 4, 2]$ の場合の例を示しています。

$i = I$ のループが終わった時点で、
 $A[1] \leq A[2] \leq A[3] \leq \dots \leq A[I]$ が成り立つ。

これは次のように証明することができます。

なお、「 $i = I - 1$ で条件を満たすと仮定して、 $i = I$ で条件を満たすことを示す」という証明技法を **数学的帰納法** といいます。



証明したいこと（事実 B）

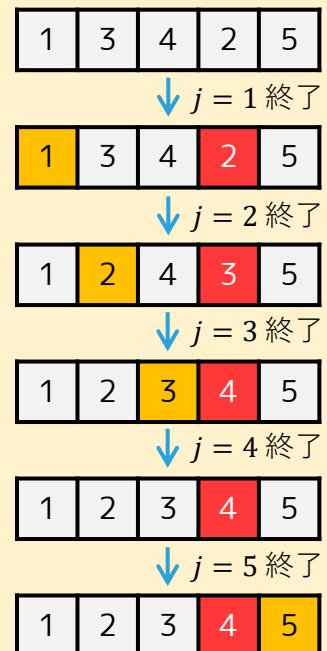
$i = I - 1$ のループが終わった時点で $A[1] \leq A[2] \leq \dots \leq A[I - 1]$ を満たす時、
 $i = I$ のループをすると $A[1] \leq A[2] \leq \dots \leq A[I]$ を満たすようになる。

事実 B の証明

$i = I - 1$ の時点で、 $A[t - 1] \leq A[I] \leq A[t]$ のとき

- $j = 1, \dots, t - 1$ では swap されない。
- $j = t$ で初めて swap される。
- $j = t + 1, \dots, I - 1$ でも swap される。
- それ以降： $A[I]$ の値が増える可能性はあるが、減ることはないと

となるため、最終的には $A[1] \leq A[2] \leq \dots \leq A[I]$ が成り立ちます。右図はその一例です。



事実 B が証明できたら？

明らかに $i = 1$ のとき事実 A が成り立ちます。また、事実 B より、 $i = 2$ のとき事実 A が成り立ちます。事実 B を繰り返し適用させると、 $i = N$ のときも事実 A が成り立つ（操作終了時にソートされている）と分かります。

問題 3.6.3

まず、列 B' が空の場合、列 A' の最も左の要素 A[c1] を取り出すため、以下のようなプログラムを書くと良いです。ここで、要素を取り出した後、列 A' の最も左の位置 c1 が 1 増えることに注意してください。

```
else if (c2 == r) {
    // 列 B' が空の場合
    C[cnt] = A[c1]; c1++;
}
```

次に、列 A'・列 B' 両方が空でない場合、以下のように場合分けすると良いです。

- 列 A' の左端 A[c1] が列 B' の左端 A[c2] より小さい：A[c1] を取り出す
- 列 B' の左端 A[c2] が列 A' の左端 A[c1] より小さい：A[c2] を取り出す

これをプログラムにすると、以下のようになります。

```
else if {
    // 列 B' が空の場合
    if (A[c1] <= A[c2]) {
        C[cnt] = A[c1]; c1++;
    }
    else {
        C[cnt] = A[c2]; c2++;
    }
}
```

全体をまとめた C++ のプログラム、Python・JAVA・C での解答例については、chap3-6.md をご覧ください。

3.7

節末問題 3.7 の解答

問題 3.7.1

答えは以下のようになります。

分からぬ人は、3.7.1 項～3.7.3 項に戻って確認しましょう。

要素	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
値	1	1	1	3	5	9	17	31	57	105

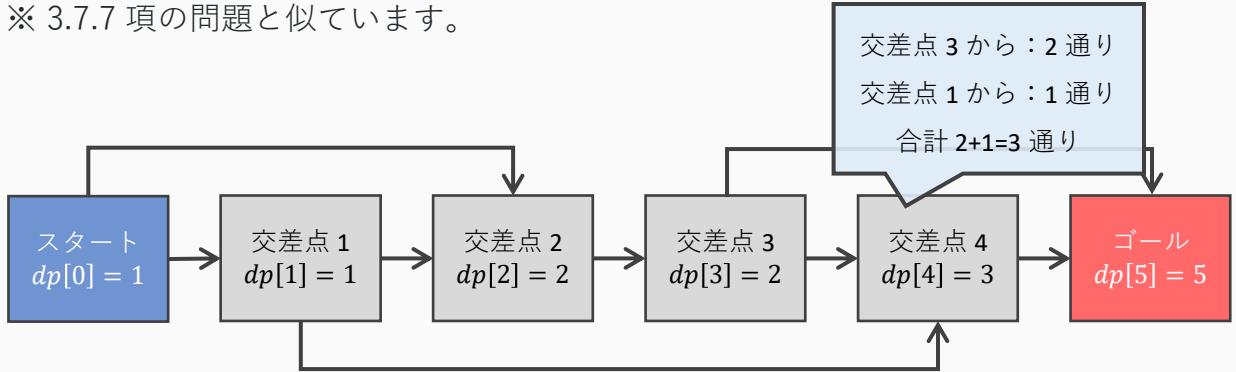
問題 3.7.2

答えは 5 通り となります。

$dp[i] = (\text{交差点 } i \text{ まで行く方法の数})$ として動的計画法をすると、答えが分かります。

ただし、スタートを交差点 0、ゴールを交差点 5 とします。

※ 3.7.7 項の問題と似ています。

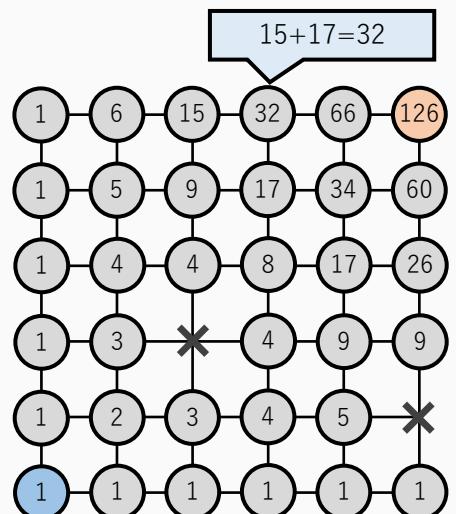


問題 3.7.3

答えは 126 通り となります。問題 3.7.2 と同様の方針で、動的計画法をすると良いです。

なお、スタートからゴールまで最短経路（10 手）で移動するには、上方向と右方向にしか移動することができないことに注意してください。

左から i 行目・下から j 列目のマス (i, j) に行くときの直前のマスは、 $(i - 1, j)$ または $(i, j - 1)$ です。



問題 3.7.4

部分和問題は、ナップザック問題（→3.7.8項）と似た、以下のような方法で解くことができます。

用意する配列（二次元配列）

$dp[i][j]$: 左から i 番目のカード（以下、カード i とする）までのなかから、和が j になる組合せが存在するならば `true`、そうでなければ `false`

動的計画法の遷移 ($i = 0$)

明らかに「何も選ばない」という方法しか存在しないので、

- $dp[0][j] = \text{true}$ ($j = 0$)
- $dp[0][j] = \text{false}$ ($j \neq 0$)

となります。

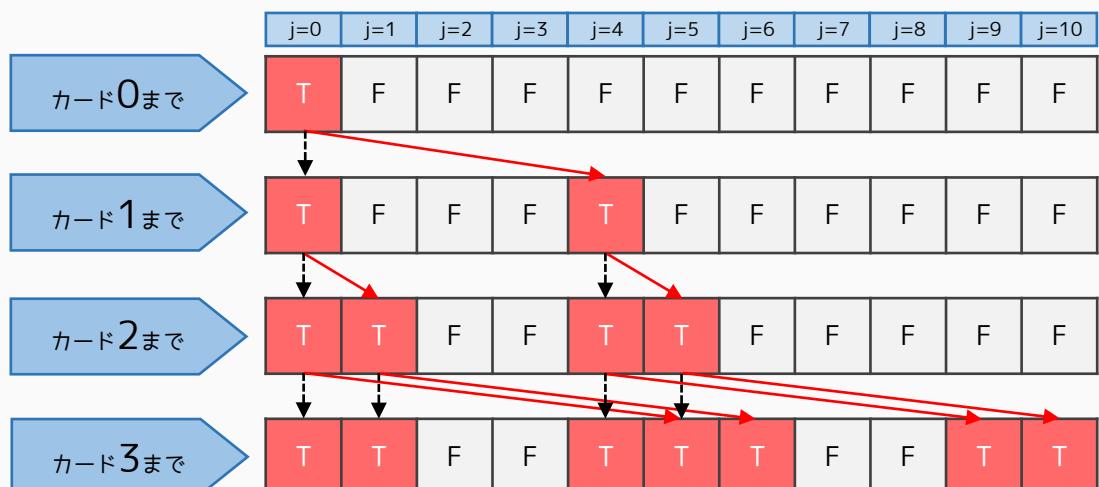
動的計画法の遷移 ($i = 1, 2, \dots, N$ の順に計算)

総和が j になるようにカード i までのなかから選ぶ方法は、以下の 2 つがあります。（最後の行動 [カード i を選ぶか] で場合分けします）

- カード $i - 1$ までの総和が $j - A_i$ であり、カード i を選ぶ
- カード $i - 1$ までの総和が j であり、カード i を選ばない

したがって、 $dp[i - 1][j - A_i]$, $dp[i - 1][j]$ のうち少なくとも一方が `true` の場合 $dp[i][j] = \text{true}$ 、そうでなければ `false` となります。

たとえば、 $N = 3, (A_1, A_2, A_3) = (4, 1, 5)$ の場合、配列 dp は次のようになります。ここで、 $dp[N][S] = \text{true}$ のとき、総和が S となるような選び方が存在します。



この解法を C++ で実装すると、以下のようになります。ナップザック問題のコード

3.7.3 とは異なり、配列 dp が bool 型であることに注意してください。

```
#include <iostream>
#include <algorithm>
using namespace std;

int N, S, A[69];
bool dp[69][10009];

int main() {
    // 入力
    cin >> N >> S;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 配列の初期化
    dp[0][0] = true;
    for (int i = 1; i <= S; i++) dp[0][i] = false;

    // 動的計画法
    for (int i = 1; i <= N; i++) {
        for (int j = 0; j <= S; j++) {
            // j < A[i] のとき、カード i は選べない
            if (j < A[i]) dp[i][j] = dp[i-1][j];
            // j >= A[i] のとき、選ぶ / 選ばない 両方の選択肢がある
            if (j >= A[i]) {
                if (dp[i-1][j] == true || dp[i-1][j-A[i]] == true) dp[i][j] = true;
                else dp[i][j] = false;
            }
        }
    }

    // 答えを出力
    if (dp[N][S] == true) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

※ Python などのソースコードは chap3-7.md をご覧ください。

問題 3.7.5

以下のようにすると、1.1.4 項の問題をナップザック問題に帰着することができます。

- 重さ：品物の値段
- 価値：品物のカロリー
- 重さの上限：500 円

問題 3.7.6

この問題は、以下のような方法で解くことができます。一次元配列を 2 個用意して、1 日目から順番に動的計画法の処理を行う方針です。

用意する配列（二次元配列）

$dp1[i]$ ： i 日目に勉強する場合の、これまでの実力アップの最大値

$dp2[i]$ ： i 日目に勉強しない場合の、これまでの実力アップの最大値

動的計画法の遷移 ($i = 0$)

1 日目から勉強できるので、 $dp1[0] = 0, dp2[0] = 0$ などの適切な値に設定しておけば良いです。

動的計画法の遷移 ($i = 1, 2, \dots, N$ の順に計算)

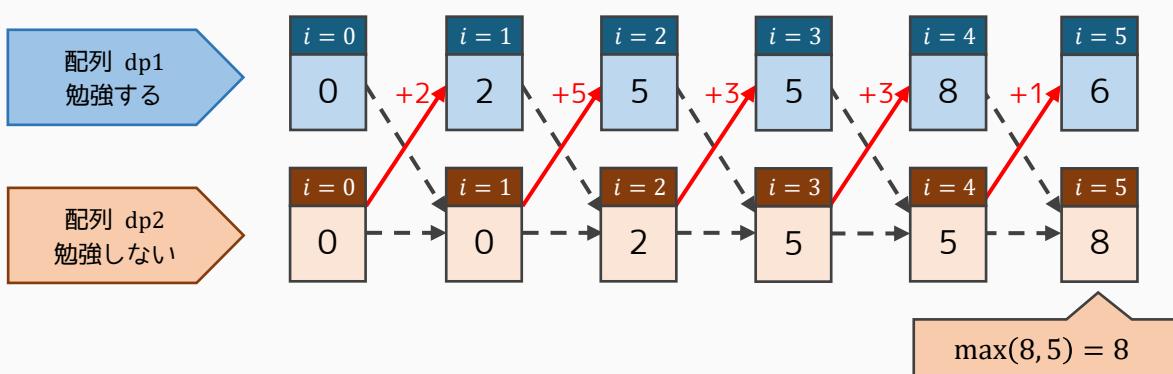
まず、 i 日目に勉強する方法は以下の 1 つしかなく、 i 日目に勉強すると実力が A_i 上がるため、 $dp1[i] = dp2[i - 1] + A_i$ となります。

- $i - 1$ 日目に勉強しない ($dp2[i - 1]$ に対応)

一方、 i 日目に勉強しない方法は以下の 2 つがあるため、 $dp2[i] = \max(dp1[i - 1], dp2[i - 1])$ となります。

- $i - 1$ 日目に勉強する ($dp1[i - 1]$ に対応)
- $i - 1$ 日目に勉強しない ($dp2[i - 1]$ に対応)

たとえば $N = 5, (A_1, A_2, A_3, A_4, A_5) = (2, 5, 3, 3, 1)$ の場合の配列 $dp1, dp2$ の遷移は以下のようになります。ここで、求める答え (N 日目を終えた後の実力アップの最大値) は $\max(dp1[N], dp2[N])$ なので、この例では答えが 8 となります。



この解法を C++ で実装すると、以下のようになります。なお、制約が $N \leq 500000$, $A_i \leq 10^9$ と大きく、答えが 10^{14} を超える可能性があります。

`int` 型などの 32 ビット整数ではオーバーフローを起こすため、`long long` 型などの 64 ビット整数を利用するすることが推奨されます。

```
#include <iostream>
#include <algorithm>
using namespace std;

long long N, A[500009];
long long dp1[500009], dp2[500009];

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 配列の初期化
    dp1[0] = 0;
    dp2[0] = 0;

    // 動的計画法
    for (int i = 1; i <= N; i++) {
        dp1[i] = dp2[i - 1] + A[i];
        dp2[i] = max(dp1[i - 1], dp2[i - 1]);
    }

    // 答えを出力
    cout << max(dp1[N], dp2[N]) << endl;
    return 0;
}
```

※ Python などのソースコードは chap3-7.md をご覧ください。

第 4 章

発展的なアルゴリズム

4.1

節末問題 4.1 の解答

問題 4.1.1

(1) $\vec{A} + \vec{B} = (2 + 3, 4 - 9) = (5, -5)$ なので、答えは以下のようになります。

- $|\vec{A}| = \sqrt{2^2 + 4^2} = \sqrt{20} = 2\sqrt{5}$ ($\sqrt{5}$ の 2 倍)
- $|\vec{B}| = \sqrt{3^2 + (-9)^2} = \sqrt{90} = 3\sqrt{10}$ ($\sqrt{10}$ の 3 倍)
- $|\vec{A} + \vec{B}| = \sqrt{5^2 + (-5)^2} = \sqrt{50} = 5\sqrt{2}$ ($\sqrt{5}$ の 2 倍)

(2) 内積の公式 (\rightarrow 4.1.4項) にしたがって計算すると、

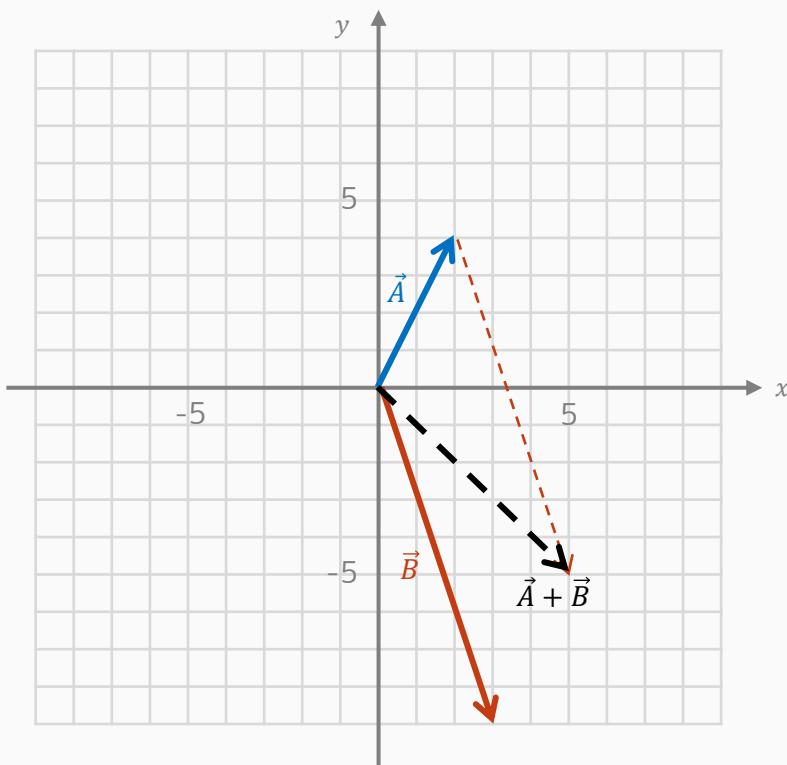
$$\vec{A} \cdot \vec{B} = 2 \times 3 + 4 \times (-9) = -30 \text{ となります。}$$

(3) 下図を見れば一瞬で分かるのですが、あえて内積を利用して求めましょう。

(2) の答えより内積が負であるため、なす角は 90 度を超えます。

(4) 外積の公式 (\rightarrow 4.1.5項) にしたがって計算すると、

$$|\vec{A} \times \vec{B}| = |2 \times (-9) - 3 \times 4| = 30 \text{ となります。}$$



問題 4.1.2

点 (x_i, y_i) と点 (x_j, y_j) の間の距離は以下の式で表されます：

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

したがって、以下のようにすべての点の組 (i, j) を全探索するプログラムを書くと、正解が得られます。なお、ルートは `sqrt` 関数を用いて計算することができます。

```
#include <iostream>
#include <cmath>
using namespace std;

int N;
double x[2009], y[2009];
double Answer = 1000000000.0; // 非常に大きい値に初期化

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> x[i] >> y[i];

    // 全探索
    for (int i = 1; i <= N; i++) {
        for (int j = i + 1; j <= N; j++) {
            // dist は i 番目の点と j 番目の点の距離
            double dist = sqrt((x[i]-x[j]) * (x[i]-x[j]) + (y[i]-y[j]) * (y[i]-y[j]));
            Answer = min(Answer, dist);
        }
    }

    // 答えの出力
    printf("%.12lf\n", Answer);
    return 0;
}
```

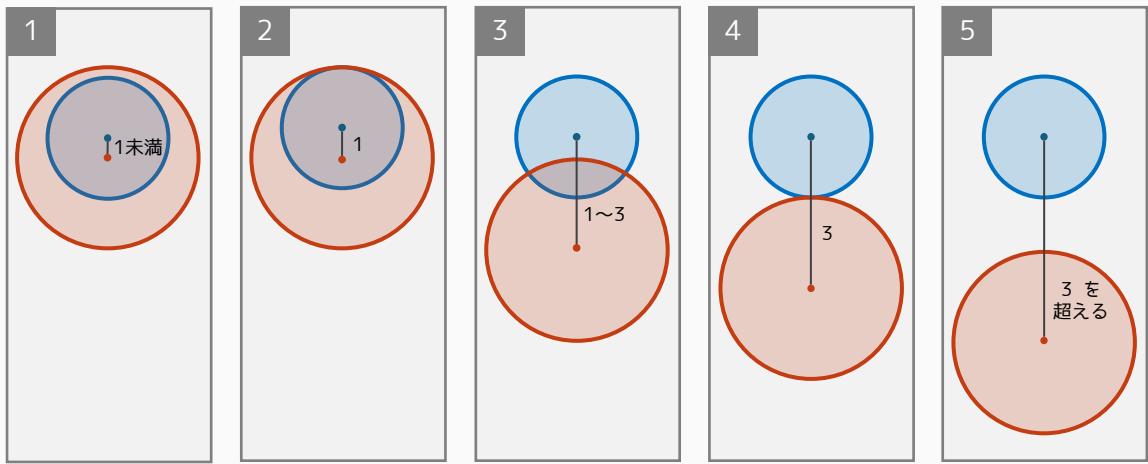
※ Python などのソースコードは chap4-1.md をご覧ください。

問題 4.1.3

2 つの円の中心間距離を d とするとき、円の重なり具合は右の表のようになります。 (パターンの番号は書籍の問題文を参照してください)

中心間距離 d	重なり 具合
$d < r_1 - r_2 $	パターン [1]
$d = r_1 - r_2 $	パターン [2]
$ r_1 - r_2 < d < r_1 + r_2$	パターン [3]
$d = r_1 + r_2$	パターン [4]
$r_1 + r_2 < d$	パターン [5]

たとえば、半径 2 の円と半径 3 の円を少しずつ離していくと下図のようになります。前ページの表の通り、距離が 1 のときに内側で接し（**内接**し）、距離が 5 のときに外側で接する（**外接**する）ことが分かります。



したがって、以下のように円の中心間距離 d を求めるプログラムを書くと、正解が得られます。2 点間距離を求める方法は、節末問題 4.1.2 で扱った通りです。

```
#include <iostream>
#include <cmath>
using namespace std;

double X1, Y1, R1;
double X2, Y2, R2;

int main() {
    // 入力
    cin >> X1 >> Y1 >> R1;
    cin >> X2 >> Y2 >> R2;

    // 円の中心間距離を求める
    double d = sqrt((X1 - X2) * (X1 - X2) + (Y1 - Y2) * (Y1 - Y2));

    // 答えを出力
    if (d < abs(R1 - R2)) cout << "1" << endl;
    else if (d == abs(R1 - R2)) cout << "2" << endl;
    else if (d < R1 + R2) cout << "3" << endl;
    else if (d == R1 + R2) cout << "4" << endl;
    else cout << "5" << endl;
    return 0;
}
```

※ Python などのソースコードは chap4-1.md をご覧ください。

問題 4.1.4

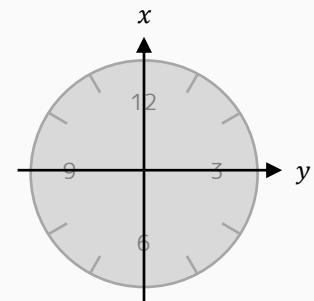
この問題は三角関数（→コラム4）を利用すると解くことができます。まず、12 時の方向を 0° とするとき、 H 時 M 分における角度は次のようにになります。

- ・時針の角度： $30H + 0.5M^\circ$
- ・分針の角度： $6M^\circ$

したがって、時計の中心を座標 $(0, 0)$ とするとき、各針の座標は次のようにになります。ただし、12 時の方向が x 軸になっていることに注意してください。

- ・時針の先端： $(A \cos(30H + 0.5)^\circ, A \sin(30H + 0.5)^\circ)$
- ・分針の先端： $(B \cos 6H^\circ, B \sin 6H^\circ)$

この 2 点間の距離を求めるプログラムを作成すると、正解となります。なお、余弦定理（本書の範囲外）を使って解く方法もあります。



```
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159265358979;

int main() {
    // 入力
    double A, B, H, M;
    cin >> A >> B >> H >> M;

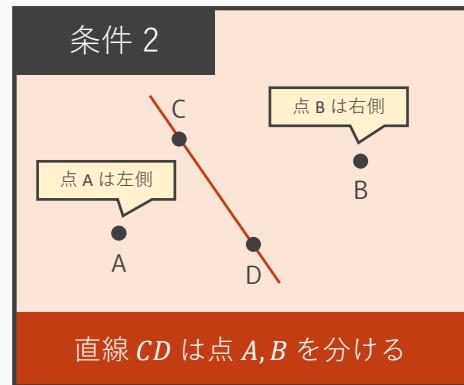
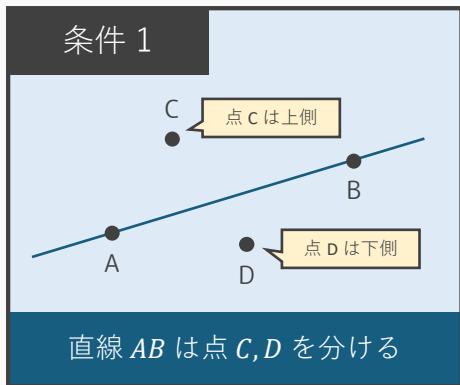
    // 座標を求める
    double AngleH = 30.0 * H + 0.5 * M;
    double AngleM = 6.0 * M;
    double Hx = A * cos(AngleH * PI / 180.0), Hy = A * sin(AngleH * PI / 180.0);
    double Mx = B * cos(AngleM * PI / 180.0), My = B * sin(AngleM * PI / 180.0);

    // 距離を求める → 出力
    double d = sqrt((Hx - Mx) * (Hx - Mx) + (Hy - My) * (Hy - My));
    printf("%.12lf\n", d);
    return 0;
}
```

※ Python などのソースコードは chap4-1.md をご覧ください。

問題 4.1.5

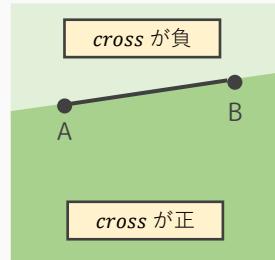
1つ目の線分の端点を A, B とし、2つ目の線分の端点を C, D とするとき、2つの線分が交差する（共通する点を持つ）ための必要十分条件は、基本的には以下の2つの条件両方を満たすことです。



そこで、線分 AB が点 C,D を分けるかどうかは、

- $\text{cross}(\overrightarrow{AB}, \overrightarrow{AC})$ の符号（正負）
- $\text{cross}(\overrightarrow{AB}, \overrightarrow{AD})$ の符号（正負）

が異なるかどうかで判定することができます。なお、
cross 関数は **4.1.5 項**に戻って確認しましょう。



したがって、以下のように実装すると答えを求めることができます。なお、点 A, B, C, D が一直線上に並んでいる場合、 $\text{cross}(\overrightarrow{AB}, \overrightarrow{AC}) = 0$ の場合など、特殊なケース（コーナーケースといいます）では場合分けをする必要があることに注意してください。

```
#include <iostream>
using namespace std;

long long cross(long long ax, long long ay, long long bx, long long by) {
    // ベクトル (ax, ay) と (bx, by) の外積の大きさ
    return ax * by - ay * bx;
}

int main() {
    // 入力
    long long X1, Y1, X2, Y2, X3, Y3, X4, Y4;
    cin >> X1 >> Y1; // 点 A の座標を入力
    cin >> X2 >> Y2; // 点 B の座標を入力
    cin >> X3 >> Y3; // 点 C の座標を入力
    cin >> X4 >> Y4; // 点 D の座標を入力
```

```

// cross の値を計算
long long ans1 = cross(X2-X1, Y2-Y1, X3-X1, Y3-Y1);
long long ans2 = cross(X2-X1, Y2-Y1, X4-X1, Y4-Y1);
long long ans3 = cross(X4-X3, Y4-Y3, X1-X3, Y1-Y3);
long long ans4 = cross(X4-X3, Y4-Y3, X2-X3, Y2-Y3);

// すべて一直線上に並んでいる場合（コーナーケース）
if (ans1 == 0 && ans2 == 0 && ans3 == 0 && ans4 == 0) {
    // A, B, C, D を数値（正確には pair 型）とみなす
    // 適切に swap することで A<B, C<D を仮定できる
    // そうすると、区間が重なるかの判定（節末問題 2.5.6）に帰着できる
    pair<long long, long long> A = make_pair(X1, Y1);
    pair<long long, long long> B = make_pair(X2, Y2);
    pair<long long, long long> C = make_pair(X3, Y3);
    pair<long long, long long> D = make_pair(X4, Y4);
    if (A > B) swap(A, B);
    if (C > D) swap(C, D);
    if (max(A, C) <= min(B, D)) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}

// そうでない場合
// IsAB: 線分 AB が点 C, D を分けるか？
// IsCD: 線分 CD が点 A, B を分けるか？
bool IsAB = false, IsCD = false;
if (ans1 >= 0 && ans2 <= 0) IsAB = true;
if (ans1 <= 0 && ans2 >= 0) IsAB = true;
if (ans3 >= 0 && ans4 <= 0) IsCD = true;
if (ans3 <= 0 && ans4 >= 0) IsCD = true;

// 答えの出力
if (IsAB == true && IsCD == true) {
    cout << "Yes" << endl;
}
else {
    cout << "No" << endl;
}
return 0;
}

```

※ Python などのソースコードは chap4-1.md をご覧ください。

4.2

節末問題 4.2 の解答

問題 4.2.1

この問題は、各経由地間の距離を単純なやり方で計算量 $O(N)$ かけて求めると、全体の計算量が $O(NM)$ となり、本問題の実行時間制限には間に合いませんが、累積和（→4.2.1項）のアイデアを使うとアルゴリズムを改善することができます。

まず、 $X < Y$ の場合は以下の性質が成り立ちます。

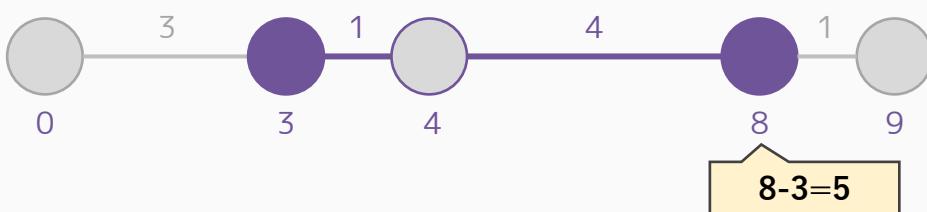
(駅 X から駅 Y までの距離)

$$= (\text{駅 1 から駅 } Y \text{ までの距離 } S_Y) - (\text{駅 1 から駅 } X \text{ までの距離 } S_X)$$

下図は、 $N = 5, (A_1, A_2, A_3, A_4) = (3, 1, 4, 1)$ の場合の具体例を示しています。駅 2 から駅 4 までの距離は $1 + 4 = 5$ と計算できますが、その代わりに

- 駅 1 から駅 4 までの距離： 8
- 駅 1 から駅 2 までの距離： 3

であることを使って、 $8 - 3 = 5$ と求めることも可能です。 $X > Y$ の場合は、 X と Y を逆にして考えれば良いです。



そこで、駅 1 から駅 i までの距離は $S_i = A_1 + \dots + A_{i-1}$ であるため、列 $[A_1, A_2, \dots, A_N]$ に累積和をとると列 $[S_1, S_2, \dots, S_N]$ が得られます。したがって、以下のように実装すると、正しい答えが求められます。計算量は $O(N + M)$ です。

```
#include <iostream>
using namespace std;

int N;
int A[200009], B[200009]; // 駅間距離、累積和
```

```

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N - 1; i++) cin >> A[i];
    cin >> M;
    for (int i = 1; i <= M; i++) cin >> B[i];

    // 累積和をとる
    S[1] = 0;
    for (int i = 2; i <= N; i++) S[i] = S[i - 1] + A[i - 1];

    // 答えを求める
    long long Answer = 0;
    for (int i = 1; i <= M - 1; i++) {
        if (B[i] < B[i + 1]) {
            Answer += (S[B[i + 1]] - S[B[i]]);
        }
        else {
            Answer += (S[B[i]] - S[B[i + 1]]);
        }
    }

    // 出力
    cout << Answer << endl;
    return 0;
}

```

※ Python などのソースコードは chap4-2.md をご覧ください。

問題 4.2.2

この問題は、以下のような手順で解くことができます。

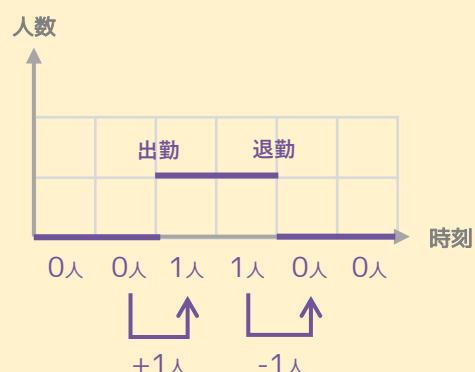
- 時刻 $t - 0.5$ と時刻 $t + 0.5$ の従業員数の差 B_t を計算する。
- 列 $[B_1, B_2, \dots, B_T]$ に累積和をとると、列 $[A_1, A_2, \dots, A_T]$ となる。

ここで、差 B_i については次のようにして計算すれば良いです。

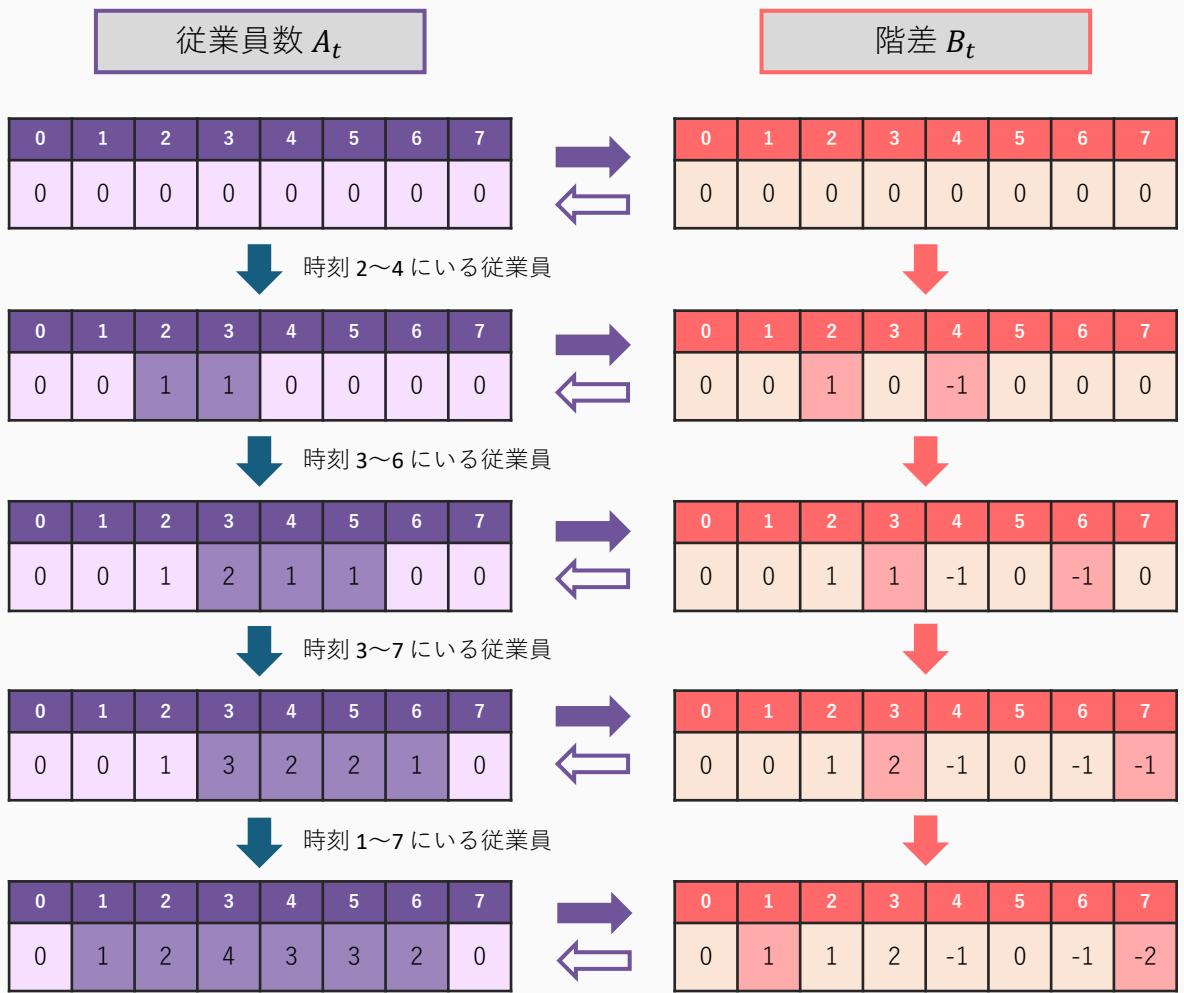
時刻 L に出勤し、時刻 R に退勤する従業員については、以下の操作を行う。

- B_L に 1 を足す。
- B_{R+1} に 1 を引く。

なお、操作を行う前は $B_i = 0$ に初期化しておく。



たとえば、 $T = 8$, $(L_i, R_i) = (2, 4), (3, 6), (3, 7), (1, 7)$ の場合における、差 B_t と従業員数 A_t の変化は以下のようになります。



したがって、各 i ($1 \leq i \leq N$) について $B[L_i]$ に $+1$ して $B[R_i]$ に -1 した後、配列 B の累積和を出力する以下のようなプログラムを書くと、正解が得られます。計算量は $O(N + T)$ です。

```
#include <iostream>
using namespace std;

int N, T;
int L[500009], R[500009];
int A[500009], B[500009];

int main() {
    // 入力
    cin >> T >> N;
    for (int i = 1; i <= N; i++) cin >> L[i] >> R[i];

    // 階差 B[i] を計算する
    for (int i = 0; i <= T; i++) B[i] = 0;
```

```

for (int i = 1; i <= N; i++) {
    B[L[i]] += 1;
    B[R[i]] -= 1;
}

// 累積和 A[i] を計算する
A[0] = B[0];
for (int i = 1; i <= T; i++) {
    A[i] = A[i - 1] + B[i];
}

// 答えを出力する
for (int i = 0; i < T; i++) cout << A[i] << endl;
return 0;
}

```

※ Python などのソースコードは chap4-2.md をご覧ください。

問題 4.2.3

まず、以下の 2 つのことを証明できれば、 $f(x) = ax^2 + bx + c$ の形で表される時に限り `true` を返すことが分かります。

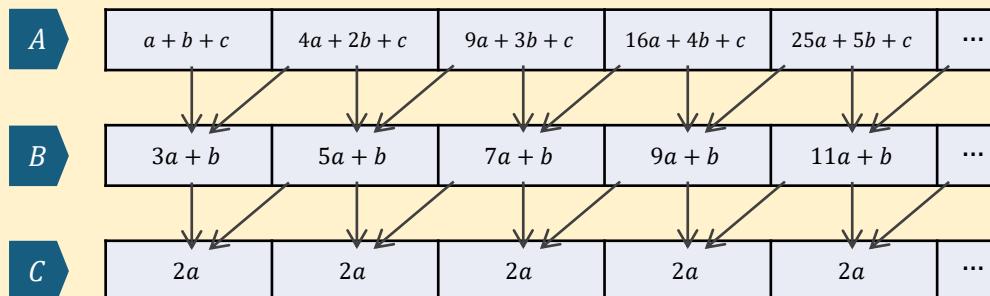
1. $f(x) = ax^2 + bx + c$ の形であれば `true` を返す
2. `true` を返した場合は $f(x) = ax^2 + bx + c$ の形で表される

それについて、証明していきましょう。

1. の証明

$f(x) = ax^2 + bx + c$ のとき、 $A[1] = a + b + c$ 、 $A[2] = 4a + 2b + c$ 、 $A[3] = 9a + 3b + c$ 、… と続きます。

そこで $B[1] = A[2] - A[1]$ なので、 $B[1] = 3a + b$ となります。それ以降についても計算していくと、以下の表の通りになります。



$C = [2a, 2a, \dots, 2a]$ になっているため、`true` を返します。

2. の証明

関数 `func` が `true` を返す、すなわち $C[1] = C[2] = \dots = C[N] = p$ となる場合を考えましょう。

階差は累積和の逆ですので、 B は C の累積和、 A は B の累積和となります。

したがって、 $A[1] = r, B[1] = q$ とすると、たとえば $B[2] = p + q$ となります。それ以降についても計算していくと、以下の表の通りになります。

A	<table border="1" style="width: 100%; border-collapse: collapse;"><tr><td style="padding: 5px; width: 15%;">r</td><td style="padding: 5px;">q + r</td><td style="padding: 5px;">p + 2q + r</td><td style="padding: 5px;">3p + 3q + r</td><td style="padding: 5px;">6p + 4q + r</td><td style="padding: 5px;">...</td></tr></table>	r	q + r	p + 2q + r	3p + 3q + r	6p + 4q + r	...
r	q + r	p + 2q + r	3p + 3q + r	6p + 4q + r	...		
B	<table border="1" style="width: 100%; border-collapse: collapse;"><tr><td style="padding: 5px; width: 15%;">q</td><td style="padding: 5px;">p + q</td><td style="padding: 5px;">2p + q</td><td style="padding: 5px;">3p + q</td><td style="padding: 5px;">4p + q</td><td style="padding: 5px;">...</td></tr></table>	q	p + q	2p + q	3p + q	4p + q	...
q	p + q	2p + q	3p + q	4p + q	...		
C	<table border="1" style="width: 100%; border-collapse: collapse;"><tr><td style="padding: 5px; width: 15%;">p</td><td style="padding: 5px;">p</td><td style="padding: 5px;">p</td><td style="padding: 5px;">p</td><td style="padding: 5px;">p</td><td style="padding: 5px;">...</td></tr></table>	p	p	p	p	p	...
p	p	p	p	p	...		

そこで、 $[A[1], A[2], \dots, A[N]] = [r, q + r, p + 2q + r, 3p + 3q + r, \dots]$ のとき、

$$f(x) = A[x] = \left(\frac{1}{2}x^2 - \frac{3}{2}x + 1 \right)p + (x-1)q + r$$

と表すことができます。これは二次関数 ($f(x) = ax^2 + bx + c$ の形) です
ので、2. が証明できました。

証明は以上です。なお、 $f(x)$ が K 次関数のとき、1 回階差をとると $K-1$ 次関数になることが知られているため、 K 回階差をとると全ての要素が同じになります。（本問題は $K=2$ の場合です）

4.3

節末問題 4.3 の解答

問題 4.3.1

この問題は、多項式関数の微分（→4.3.3項）の理解を問う問題です。答えは以下のようになります。

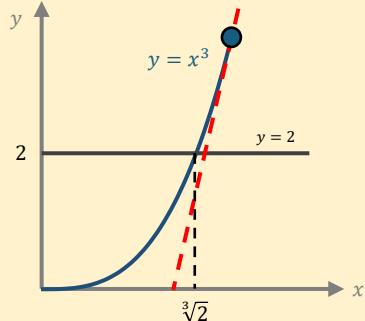
1. $f'(x) = 7$
2. $f'(x) = 2x + 4$
3. $f'(x) = 5x^4 + 4x^3 + 3x^2 + 2x + 1$

問題 4.3.2

$\sqrt[3]{2}$ の値は、以下のような方針で求めることができます。（→4.3.6項）

- $f(x) = x^3$ とする。ここで $f'(x) = 3x^2$ 。
- 最初、適当な初期値 a を設定する。
- その後、 a の値を以下に更新し続ける。

点 $(a, f(a))$ における接線と直線
 $y = 2$ の交点の x 座標



このため、以下のようなプログラムを書けば良いです。

```
#include <iostream>
using namespace std;

int main() {
    double r = 2.0; // √2 を求めたいから
    double a = 2.0; // 初期値を適当に 2.0 にセットする

    for (int i = 1; i <= 5; i++) {
        // 点 (a, f(a)) の x 座標と y 座標を求める
        double zahyou_x = a;
        double zahyou_y = a * a * a; ← コード 4.3.1 からの変更部分

        // 接線の傾きを求める [y = (sessen_a)x + sessen_b とする]
    }
}
```

```

double sessen_a = 3.0 * zahyou_x * zahyou_x; ← コード 4.3.1 からの変更部分
double sessen_b = zahyou_y - sessen_a * zahyou_x;

// 次の a の値 next_a を求める
double next_a = (r - sessen_b) / sessen_a;
printf("Step #%-d: a = %.12lf -> %.12lf\n", i, a, next_a);
a = next_a;
}
return 0;
}

```

このとき、出力は以下のようになります。急激に $\sqrt[3]{2} = 1.259921049894 \dots$ に近づき、たった 5 回で 12 行目まで一致します。

```

Step #1: a = 2.000000000000 -> 1.500000000000
Step #2: a = 1.500000000000 -> 1.296296296296
Step #3: a = 1.296296296296 -> 1.260932224742
Step #4: a = 1.260932224742 -> 1.259921860566
Step #5: a = 1.259921860566 -> 1.259921049895

```

なお、Python・JAVA・C のソースコードにつきましては、GitHub の chap4-3.md をご覧ください。

問題 4.3.3

二分探索法を用いて、手計算で $\sqrt{2}$ を求める過程を以下の表に示します。

操作回数	l	r	m	$m^2 < 2$ か？	範囲のイメージ
1 回目	1.00000	2.00000	1.50000	No	
2 回目	1.00000	1.50000	1.25000	Yes	
3 回目	1.25000	1.50000	1.37500	Yes	
4 回目	1.37500	1.50000	1.43750	No	
5 回目	1.37500	1.43750	1.40625	Yes	
6 回目	1.40625	1.43750	1.42188	No	
7 回目	1.40625	1.42188	1.41406	Yes	
8 回目	1.41406	1.42188	1.41797	No	
9 回目	1.41406	1.41797	1.41602	No	

9 回操作を行いましたが、 $\sqrt{2} = 1.41421 \dots$ の下 6 行とはなかなか一致しません。

そこで、以下のようなプログラムを作成し、何回の操作で一致する桁数が 6 桁に達するかを調べてみましょう。（Python・JAVA・C のプログラムは chap4-3.md をご覧ください）

```
#include <iostream>
using namespace std;

int main() {
    double l = 1.0;
    double r = 2.0;

    for (int i = 1; i <= 20; i++) {
        double m = (l + r) / 2.0;
        if (m * m < 2.0) l = m;
        else r = m;
        printf("Step #%-d: m = %.12lf\n", i, m);
    }
    return 0;
}
```

このとき、出力は以下のようになります。15 回目の操作でやっと 6 桁一致することが分かります。ニュートン法は 3 回なので、それに比べれば遅いです。

```
Step #1: m = 1.500000000000
Step #2: m = 1.250000000000
Step #3: m = 1.375000000000
Step #4: m = 1.437500000000
Step #5: m = 1.406250000000
Step #6: m = 1.421875000000
Step #7: m = 1.414062500000
Step #8: m = 1.417968750000
Step #9: m = 1.416015625000
Step #10: m = 1.415039062500
Step #11: m = 1.414550781250
Step #12: m = 1.414306640625
Step #13: m = 1.414184570312
Step #14: m = 1.414245605469
Step #15: m = 1.414215087891
Step #16: m = 1.414199829102
Step #17: m = 1.414207458496
Step #18: m = 1.414211273193
Step #19: m = 1.414213180542
Step #20: m = 1.414214134216
```

なお、このような二分探索法では、1 回の操作で精度が 2 倍になるため、精度を P 倍に上げるにはおよそ $\log_2 P$ 回の操作が必要です。今回の場合は $P = 10^5$ であり、操作回数は $\log_2 P \approx 16$ 回とほぼ一致します。

問題 4.3.4

指数法則（→**2.3.9項**）より、 $10^{0.3} = 1000^{0.1} = \sqrt[10]{1000}$ です。このため、たとえば以下のようない方法が考えられます。

なお、 x^5 のような累乗（整数乗）は、pow 関数を使わなくとも $x * x * x * x * x$ といったように四則演算だけで計算することができます。

方法1

$f(x) = x^{10}, r = 2$ として、一般化したニュートン法（→**4.3.6項**）を適用する。

ここで、 $f'(x) = 10x^9$ となる。

方法2

$x^{10} = 1000$ となるような x の値を二分探索（→**節末問題4.3.3**）によって求める。明らかに $1 < x < 2$ なので、初期値としては $l = 1, r = 2$ などを設定すれば良い。

ほかにも多数の方法がありますので、是非考えてみてください。

4.4

節末問題 4.4 の解答

問題 4.4.1 (1)

この問題は、多項式関数を積分する方法（→4.4.3項）の理解を問う問題です。

$$F(x) = \frac{1}{4}x^4 + x^3 + \frac{3}{2}x^2 + x$$

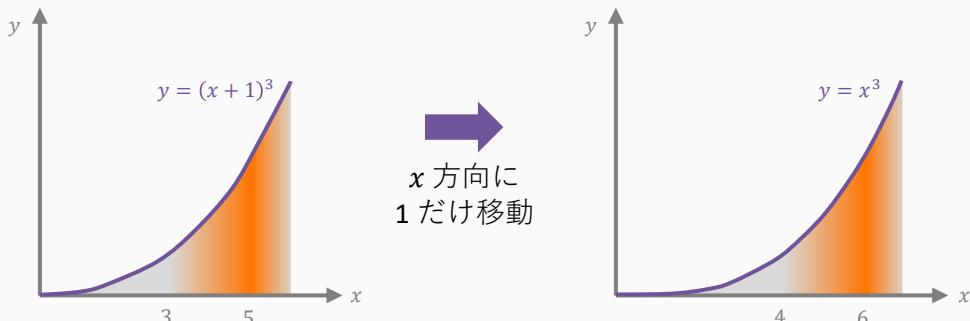
とすると、求める答えは以下のようになります。

$$\int_3^5 (x^3 + 3x^2 + 3x + 1) dx = F(5) - F(3)$$

そこで $F(5) = 323.75, F(3) = 63.75$ であるため、答えは $323.75 - 63.75 = \boxed{260}$ です。
なお、 $(x^3 + 3x^2 + 3x + 1) = (x + 1)^3$ であることを使うと、楽に計算できます。

$$\int_3^5 (x + 1)^3 dx = \int_4^6 x^3 dx = \frac{1}{4}(6^4 - 4^4) = 260$$

関数のグラフを右方向に 1 だけ平行移動させると、分かりやすいです。



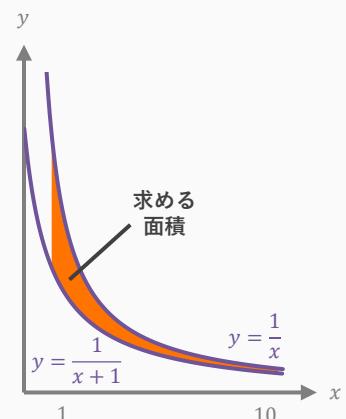
問題 4.4.1 (2)

この問題は、 $1/x$ を積分する方法（→4.4.3項）の理解を問う問題です。

積分は符号付き面積を求める操作に対応するため、

$$F(x) = \int_1^{10} \frac{1}{x} - \frac{1}{x+1} dx = \int_1^{10} \frac{1}{x} dx - \int_1^{10} \frac{1}{x+1} dx$$

が成り立ちます。イメージは右図の通りです。



さて、赤色部分・青色部分をそれぞれ求めると、以下のようになります。 $1/(x+1)$ の積分が分からぬ人は、4.4.5 項に戻って確認しましょう。

$$\int_1^{10} \frac{1}{x} dx = \log_e 10 - \log_e 1 = \log_e 10$$

$$\int_1^{10} \frac{1}{x+1} dx = \int_2^{11} \frac{1}{x} dx = \log_e 11 - \log_e 2 = \log_e 11/2$$

そこで、対数関数の公式（→2.3.10項）より、求める答えは

$$\log_e 10 - \log_e \frac{11}{2} = \log_e \left(10 \div \frac{11}{2} \right) = \boxed{\log_e \frac{20}{11}}$$

です。およそ 0.5978 となります。

問題 4.4.1 (3)

実は、以下の式が成り立ちます。

$$\frac{1}{x^2 + x} = \frac{1}{x} - \frac{1}{x+1}$$

したがって、求める答えは (2) と同じになります。

$$\int_1^{10} \frac{1}{x^2 + x} dx = \int_1^{10} \frac{1}{x} - \frac{1}{x+1} dx = \boxed{\log_e \frac{20}{11}}$$

問題 4.4.2

定積分の値は、約 1.2882263643059391197 であることが知られています。

それでは、どうやってこの値を求めるのでしょうか。多項式関数などの積分は手計算でも正確な値を計算できますが、 $f(x) = 2^{x^2}$ の場合は関数 $f(x)$ が複雑であるため、厳密な答えを計算するのは非常に難しいです。

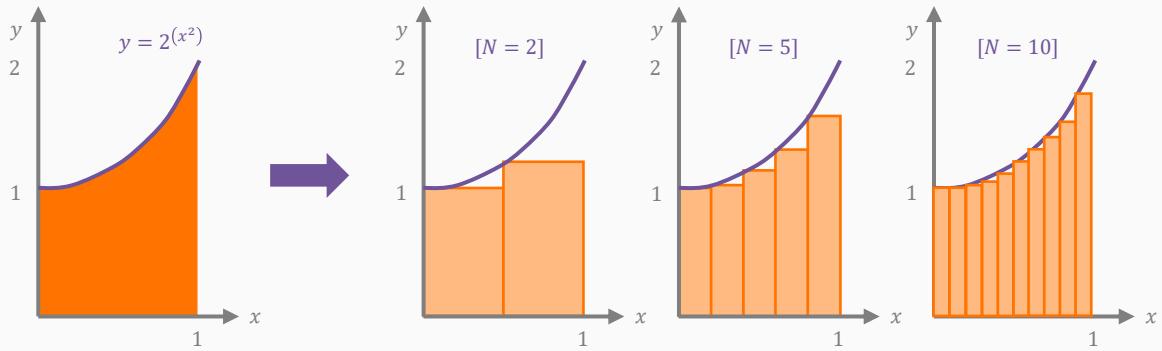
このような場合、代わりに答えの近似値を計算する数値計算（→4.3.7項）と呼ばれる手法がよく使われます。ここでは代表的な方法を 2 つ紹介します。

方法 1：単純な区分求積法

積分は面積を求める操作に対応するため、 $f(x) = 2^{(x^2)}$ とするとき、

$$\int_0^1 f(x) dx = \frac{f(0) + f\left(\frac{1}{N}\right) + f\left(\frac{2}{N}\right) + \cdots + f\left(\frac{N-1}{N}\right)}{N}$$

と近似することができます。 $N = 2, 5, 10$ の場合のイメージ図は以下の通りです。



この方法で定積分の値を求めるプログラムの例として、以下が考えられます。ここで、 N の値を増やせば増やすほど近似精度が上がります。

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    int N = 1000000;
    double Answer = 0.0;

    for (int i = 0; i < N; i++) {
        double x = 1.0 * i / N;
        double value = pow(2.0, x * x); // f(i/N) の値
        Answer += value;
    }
    printf("%.14lf\n", Answer / N);
    return 0;
}
```

しかし、この方法では絶対誤差を 10^{-12} 以下にすることは難しいです。実際、

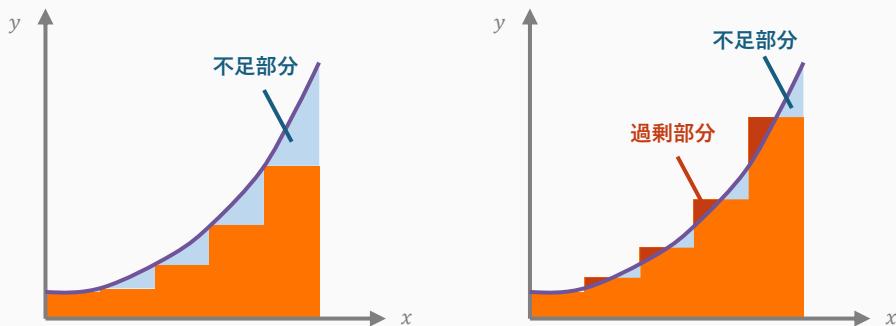
- $N = 1000$ のとき、出力は **1.28772659535497**
- $N = 1000000$ のとき、出力は **1.28822586430618**

となり、3~6桁しか一致していません。

方法 2：中央の値を使う

方法 1 では区間の左端を使いましたが、ここでは中央の値 $f(1/2N), f(3/2N), \dots$ を使って面積を求めてみましょう。

下図は $N = 5$ の場合の例を示しており、赤色は過剰な部分、青色は不足している部分を意味します。中央の値を使った場合、赤色と青色の面積がほぼ同じであり、正確に数えられているように思えます。



数式で表すと、定積分は以下のように近似することができます。

$$\int_0^1 f(x) dx = \frac{f\left(\frac{1}{2N}\right) + f\left(\frac{3}{2N}\right) + \cdots + f\left(\frac{2N-1}{2N}\right)}{N}$$

これを実装すると、以下のようになります。（Python・JAVA・C のプログラムは chap4-4.md をご覧ください）

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    int N = 1000000;
    double Answer = 0.0;

    for (int i = 0; i < N; i++) {
        double x = 1.0 * (2 * i + 1) / (2 * N);
        double value = pow(2.0, x * x); // f(i/N) の値
        Answer += value;
    }
    printf("%.14lf\n", Answer / N);
    return 0;
}
```

方法 1 と比べて精度がかなり良くなり、 $N = 1000000$ のとき絶対誤差 10^{-12} を実現することができます。

- $N = 1000$ のとき、出力は **1.28822624878143**
- $N = 1000000$ のとき、出力は **1.28822636430577**

なお、さらに効率的に定積分の近似値を求める方法として、シンプソンの公式などが知られています。興味のある方は、インターネットなどで調べてみてください。

問題 4.4.3

単純な解法として、以下のものが考えられます。

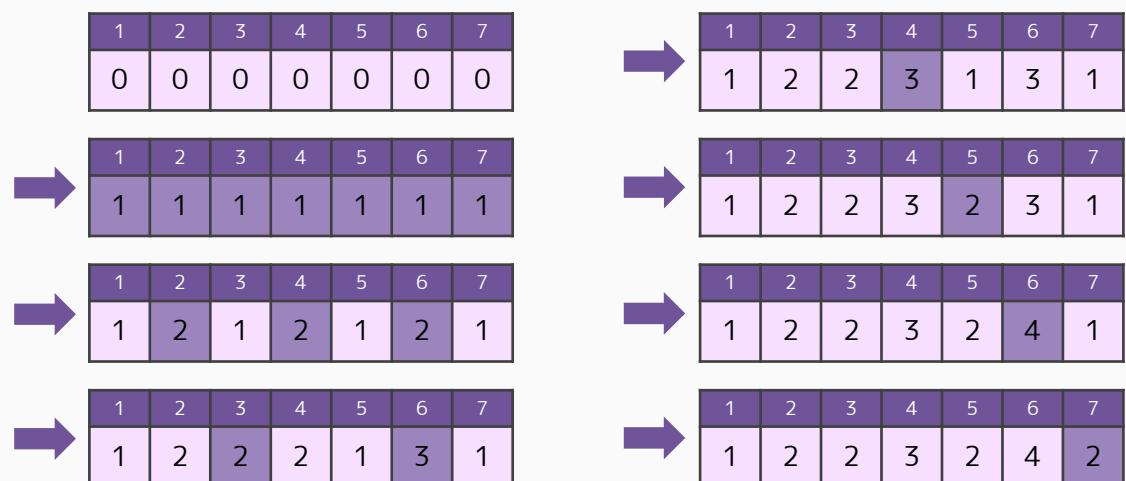
i = 1, 2, …, N の順に、約数をすべて列挙することによって $f(i)$ を計算する。

そうすると答えが分かる。約数列挙の計算量は $O(\sqrt{N})$ であるため、処理全体の計算量は $O(N^{1.5})$ である。

しかし、本問題の制約では実行時間制限超過 (TLE) となってしまいます。より高速に $f(1), f(2), \dots, f(N)$ を計算する方法として、以下が考えられます。

1. 最初、すべての i ($1 \leq i \leq N$) について $f(i) = 0$ とする。
2. 1 の倍数： $f(1), f(2), f(3), f(4), \dots$ に 1 を加算する。
3. 2 の倍数： $f(2), f(4), f(6), f(8), \dots$ に 1 を加算する。
4. 3 の倍数： $f(3), f(6), f(9), f(12), \dots$ に 1 を加算する。
5. $4, 5, 6, 7, \dots, N$ の倍数についても同じような操作をする。

$N = 7$ の場合における、 $f(1), f(2), \dots, f(N)$ を求める過程は以下のようになります。



それでは、このアルゴリズムの計算量を見積もってみましょう。 x の倍数は全部で N/x 個あるため、 x の倍数すべてに 1 を足す操作には計算量 $O(N/x)$ かかります。したがって、全体の計算回数は、

$$\frac{N}{1} + \frac{N}{2} + \cdots + \frac{N}{N} = O(N \log N)$$

となり、 $N = 10^7$ の場合でも十分高速に動作します（Python などの低速なプログラミング言語では TLE するかもしれません）。実装例は以下の通りです。

```
#include <iostream>
using namespace std;

long long N;
long long F[10000000];
long long Answer = 0;

int main() {
    // 入力 → 配列の初期化
    cin >> N;
    for (int i = 1; i <= N; i++) F[i] = 0;

    // F[1], F[2], ..., F[N] を計算する
    for (int i = 1; i <= N; i++) {
        // F[i], F[2i], F[3i], ... に 1 を加算
        for (int j = i; j <= N; j += i) F[j] += 1;
    }

    // 答えを求める → 出力
    for (int i = 1; i <= N; i++) {
        Answer += 1LL * i * F[i];
    }
    cout << Answer << endl;
    return 0;
}
```

なお、数学的考察編の「足された回数を考えるテクニック（→5.7節）」を使うと、この問題を計算量 $O(N)$ で解くことも可能です。

問題 4.4.4

答えは **6,000,022,499,693** であることが知られています。

- 出典：<https://oeis.org/A004080/b004080.txt>

(解説は次ページへ続きます)

単純な方法として、以下のプログラムのように $1/1 + 1/2 + 1/3 + \dots$ を順番に足していく方法が考えられます。しかし、現実的な時間では $N = 23$ 程度までしか実行が終わりません。

```
#include <iostream>
using namespace std;

int main() {
    // パラメータの設定・初期化
    long long cnt = 0;
    double LIMIT = 23; // これを 30 にすれば答えが求められる
    double Current = 0;

    // 1 つずつ足していく
    while (Current < LIMIT) {
        cnt += 1;
        Current += 1.0 / cnt;
    }

    // 答えを出力
    cout << cnt << endl;
    return 0;
}
```

そこで高速に求める代表的な方法として、以下の 2 つが挙げられます。他にも様々な方法がありますので、ぜひ考えてみてください。

方法 1：近似を使う

脚注で述べたように、オイラー定数を $\gamma = 0.57721566490153286 \dots$ 、 $1/1$ から $1/n$ までの和を H_n とすると、 H_n の値は $\log_e n - \gamma$ に非常に近い値となります。そこで、 $\log_e n - \gamma \geq 30$ となる最小の n は、

$$\lfloor e^{30+\gamma} \rfloor = \lfloor 6000022499693.369 \dots \rfloor = 6000022499693$$

となり、何と答えと一致します。

方法 2：並列計算を使う

計算回数が 10^{12} 回を超える場合、通常は現実的な時間で計算が終わりません。しかし、CUDA などのプログラミング言語を使って並列計算をすると、計算時間が 100 倍以上短縮されます。有名な「スパコン富岳」も実は並列計算で動いています。興味のある人は、インターネットなどで調べてみてください。

4.5

節末問題 4.5 の解答

問題 4.5.1

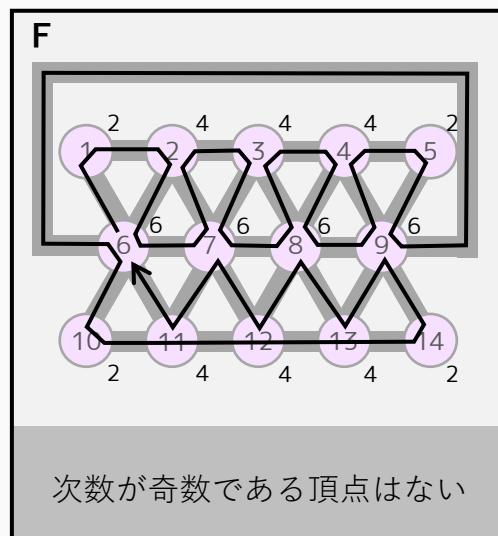
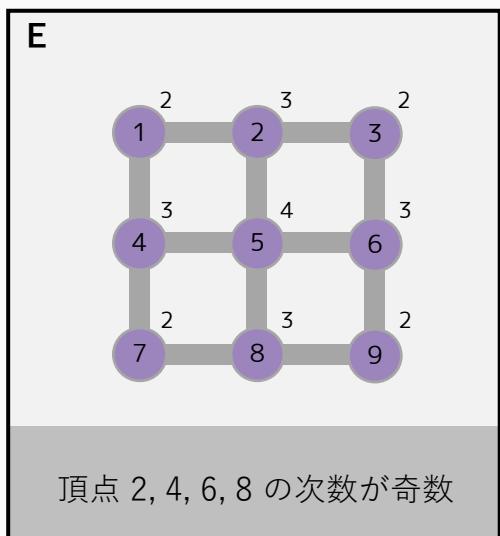
答えは以下の通りです。分からぬ人は、4.5.2 項・4.5.4 項を確認しましょう。

グラフの番号	グラフの種類	次数最大の頂点
A	重みなし無向グラフ	頂点 1 (次数=3)
B	重み付き無向グラフ	頂点 5 (次数=4)
C	重みなし有向グラフ	頂点 3 (出次数=2)
D	重み付き有向グラフ	頂点 2 (出次数=3)

問題 4.5.2

まず、グラフ E については次数が奇数である頂点が存在するため、すべての頂点を一度ずつ通って戻ってくる経路が存在しません。また、グラフ F はすべての頂点の次数が偶数であり、下図のような経路が存在します。

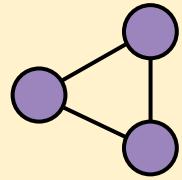
分からぬ人は、オイラーーグラフ（→4.5.2項）を確認しましょう。なお、下図の頂点に付いている番号は、その頂点の次数です。



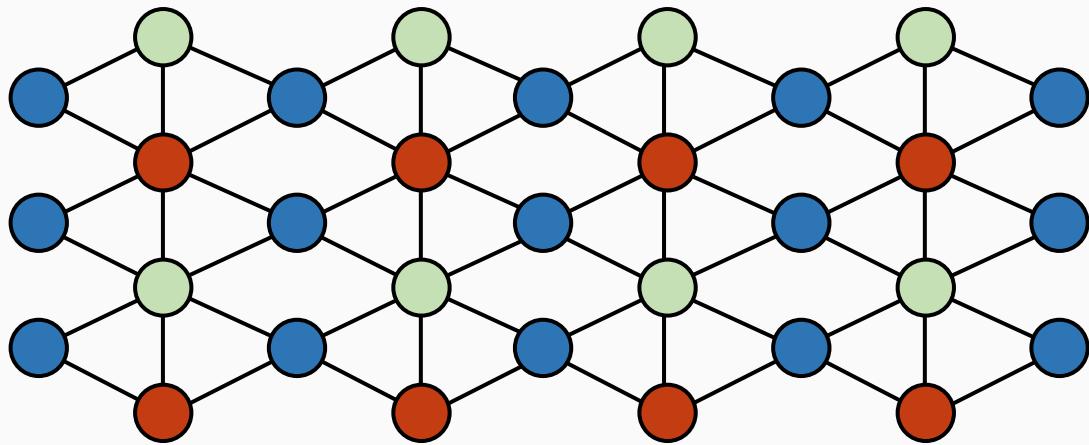
問題 4.5.3

まず、2色で塗り分けられない理由は以下の通りです。

グラフには右図のような三角形が含まれており、この三角形だけを考えても2色で塗ることは明らかに不可能だから。



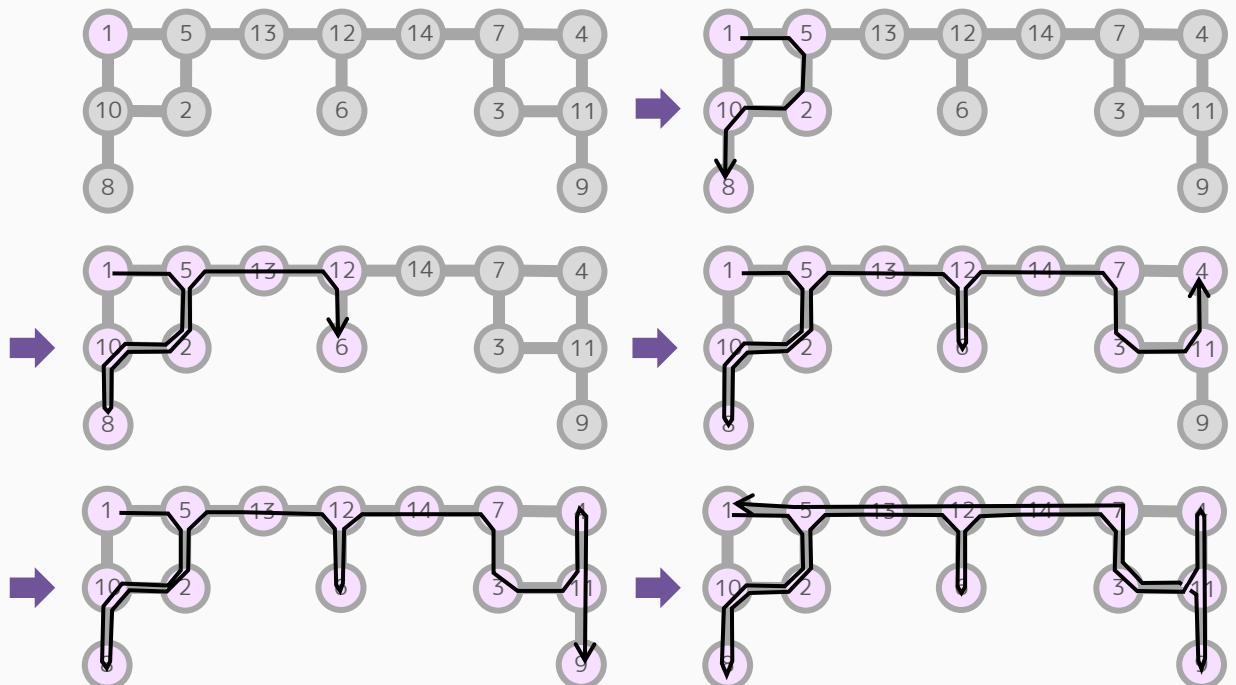
また、以下のように塗ると、3色で塗り分けることができます。



問題 4.5.4

訪問順序は順に **1, 5, 2, 10, 8, 13, 12, 6, 14, 7, 3, 11, 4, 9** です。

以下の図は、深さ優先探索の具体的な動きを示しています。



問題 4.5.5

この問題は、隣接リスト形式（→[4.5.5項](#)）の理解を問う問題です。たとえば以下のように実装すると、正解が得られます。

なお、変数 `cnt` は頂点 i に隣接する頂点のうち、番号が i 未満のものの個数を示しています。また、`G[i].size()` はリスト `G[i]` の要素数です。

```
#include <iostream>
#include <vector>
using namespace std;

int N, M;
int A[100009], B[100009];
vector<int> G[100009];

int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= M; i++) {
        cin >> A[i] >> B[i];
        G[A[i]].push_back(B[i]);
        G[B[i]].push_back(A[i]);
    }

    // 答えを求める
    int Answer = 0;
    for (int i = 1; i <= N; i++) {
        int cnt = 0;
        for (int j = 0; j < G[i].size(); j++) {
            // G[i][j] は頂点 i に隣接している頂点のうち j 個目
            if (G[i][j] < i) cnt += 1;
        }
        // 自分自身より小さい隣接頂点が 1 つであれば Answer に 1 を加算する
        if (cnt == 1) Answer += 1;
    }

    // 出力
    cout << Answer << endl;
    return 0;
}
```

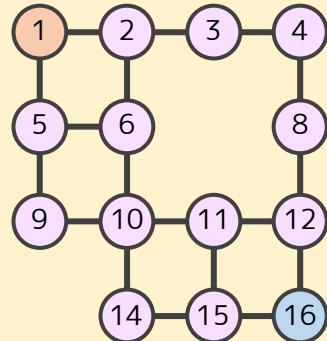
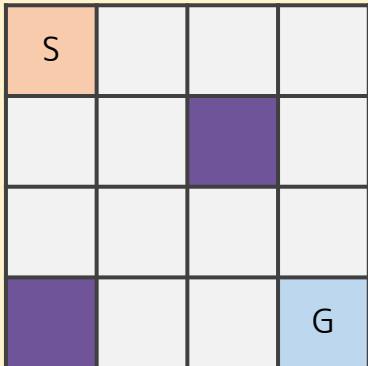
※ Python などのソースコードは `chap4-5.md` をご覧ください。

問題 4.5.6

以下のように頂点番号を設定すると、最短経路問題（→4.5.7項）に帰着することができます。頂点数は HW 、辺の数は $4HW$ 以下なので、1 秒以内で実行が終わります。

上から i 行目、左から j マス目の頂点番号を $(i - 1) \times W + j$ に設定する。

$H = 4, W = 4$ の場合の具体例は以下の図の通り。



したがって、以下のような実装で正解を出すことができます。なお、各マスに頂点番号を振るように、データを数値で表現して識別することをハッシュといいます。資格試験などでも頻出なので、興味のある人はインターネットなどで調べてみてください。

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

// 入力
int H, W;
int sx, sy, start; // スタートの座標 (sx, sy) と頂点番号 sx*W+sy
int gx, gy, goal; // ゴールの座標 (gx, gy) と頂点番号 gx*W+gy
char c[59][59];

// グラフ・最短経路
int dist[2509];
vector<int> G[2509];

int main() {
    // 入力
    cin >> H >> W;
    cin >> sx >> sy; start = sx * W + sy;
    cin >> gx >> gy; goal = gx * W + gy;
    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W; j++) cin >> c[i][j];
    }
}
```

```

// 横方向の辺 [(i, j) - (i, j+1)] をグラフに追加
for (int i = 1; i <= H; i++) {
    for (int j = 1; j <= W - 1; j++) {
        int idx1 = i * W + j; // 頂点 (i, j) の頂点番号
        int idx2 = i * W + (j+1); // 頂点 (i, j+1) の頂点番号
        if (c[i][j] == '.' && c[i][j+1] == '.'){
            G[idx1].push_back(idx2);
            G[idx2].push_back(idx1);
        }
    }
}

// 縦方向の辺 [(i, j) - (i+1, j)] をグラフに追加
for (int i = 1; i <= H - 1; i++) {
    for (int j = 1; j <= W; j++) {
        int idx1 = i * W + j; // 頂点 (i, j) の頂点番号
        int idx2 = (i+1) * W + j; // 頂点 (i+1, j) の頂点番号
        if (c[i][j] == '.' && c[i+1][j] == '.'){
            G[idx1].push_back(idx2);
            G[idx2].push_back(idx1);
        }
    }
}

// 以降は（頂点数などを除き）コード 4.5.3 と同じ
// 幅優先探索の初期化 (dist[i]=-1 のとき、未到達の白色頂点である)
for (int i = 1; i <= H * W; i++) dist[i] = -1;
queue<int> Q; // キュー Q を定義する
Q.push(start); dist[start] = 0; // Q に 1 を追加（操作 1）

// 幅優先探索
while (!Q.empty()) {
    int pos = Q.front(); // Q の先頭を調べる（操作 2）
    Q.pop(); // Q の先頭を取り出す（操作 3）
    for (int i = 0; i < (int)G[pos].size(); i++) {
        int nex = G[pos][i];
        if (dist[nex] == -1) {
            dist[nex] = dist[pos] + 1;
            Q.push(nex); // Q に nex を追加（操作 1）
        }
    }
}

// 答えを出力
cout << dist[goal] << endl;
return 0;
}

```

※ Python などのソースコードは chap4-5.md をご覧ください。

問題 4.5.7

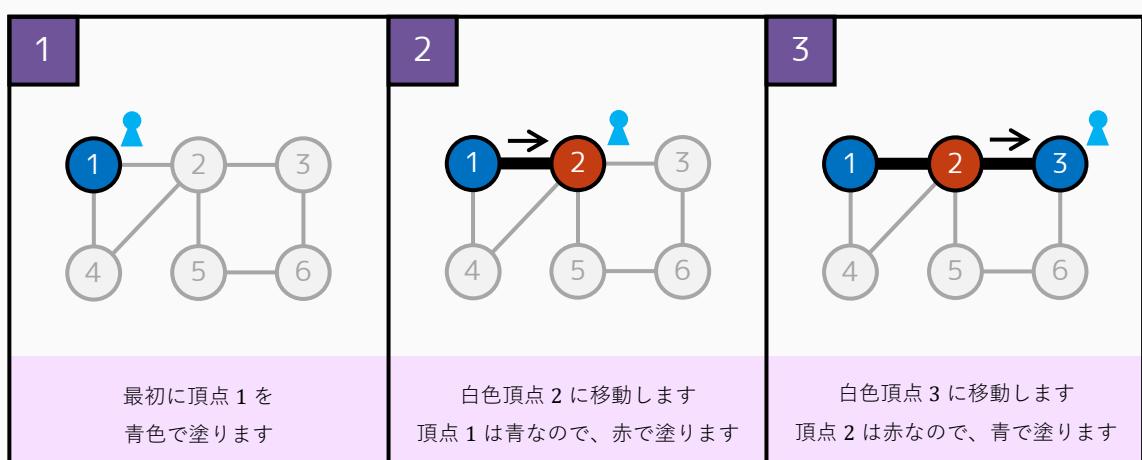
まず、グラフが連結である場合、1つの頂点の色さえ決まれば、グラフを青と赤の2色で塗る方法は高々1つに定まります（下図参照）。なぜなら、青の隣に赤、赤の隣に青、…と塗っていく必要があるからです。

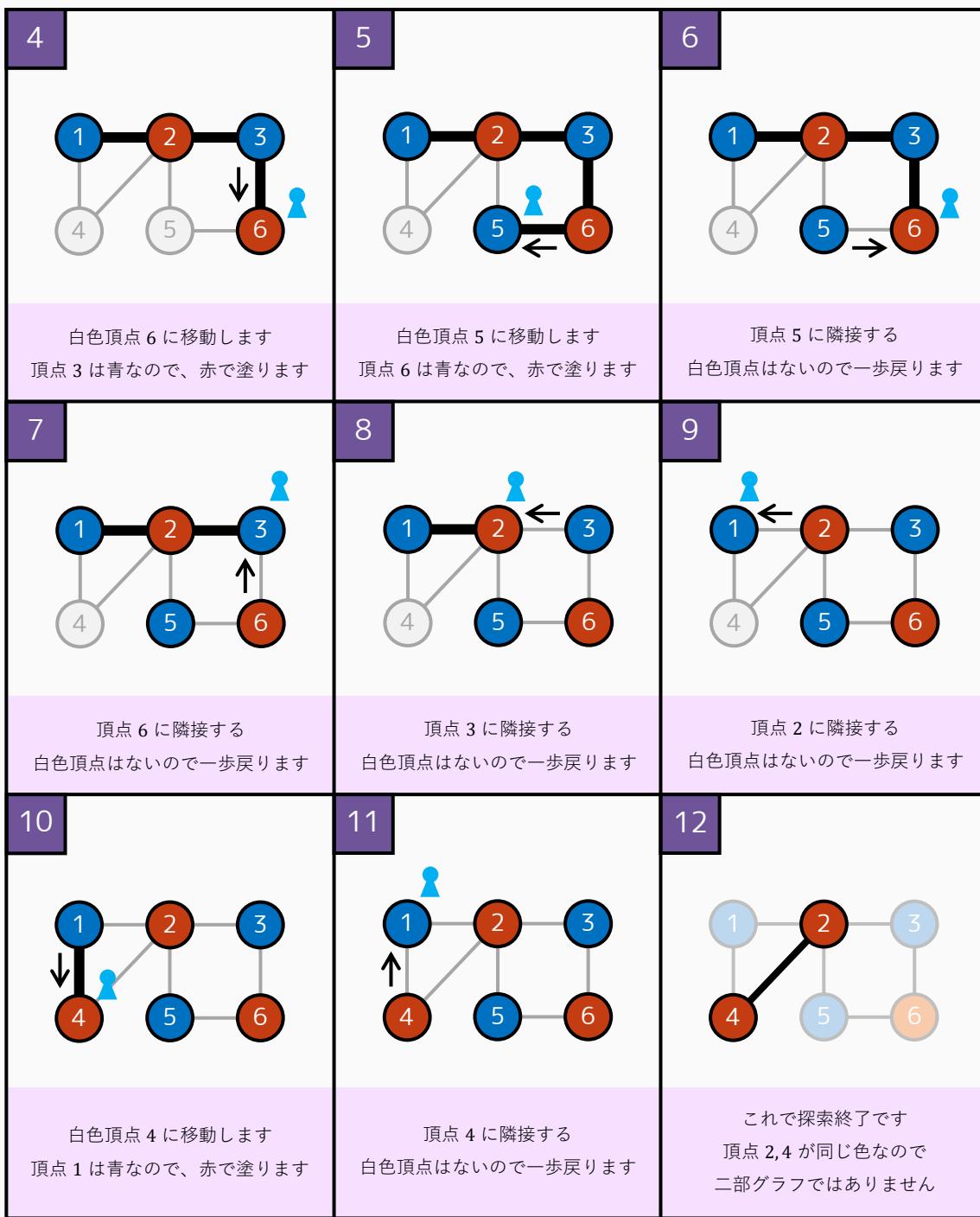


このため、以下のように深さ優先探索をすると、グラフを青と赤で塗る方法を1つ構成することができます。なお、青文字で示した部分は、4.5.6項で述べた連結判定のアルゴリズムと異なる部分です。

1. すべての頂点を白色で塗る。
2. 一番最初に頂点1を訪問し、頂点1を**青色**で塗る。
3. その後、以下の操作を繰り返す。
 - A) 隣接する白色頂点がない：一步戻る
 - B) 隣接する白色頂点がある：これらの中で番号が最小の頂点を訪問する。新たに頂点を訪問する際には、**前の頂点と異なる色**で塗る。
4. 最終的に、**どの隣り合う2頂点も異なる色で塗られていたら、グラフは二部グラフである。**

このアルゴリズムを具体的なグラフに適用させると、以下のようにになります。太線は移動経路の跡を示しています。





このアルゴリズムを実装すると、次ページのようになります。なお、プログラミングでは実際に頂点を白・青・赤で塗ることはできないので、

- $\text{color}[i]=0$ のとき：頂点 i が白色
- $\text{color}[i]=1$ のとき：頂点 i が青色
- $\text{color}[i]=2$ のとき：頂点 i が赤色

としています。また、グラフが連結でないケースが入力される可能性があるので、手順 2. に相当する操作を各連結成分に対して行わなければならないことに注意してください。（プログラムの「深さ優先探索」部分参照）

```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int N, M, A[200009], B[200009]; // N, M ≤ 200000 なので配列の大きさは 200009 にしておく
vector<int> G[200009];
int color[200009];

void dfs(int pos) {
    for (int i : G[pos]) { // 範囲 for 文
        if (color[i] == 0) {
            // color[pos]=1 のとき 2、color[pos]=2 のとき 1
            color[i] = 3 - color[pos];
            dfs(i);
        }
    }
}

int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= M; i++) {
        cin >> A[i] >> B[i];
        G[A[i]].push_back(B[i]);
        G[B[i]].push_back(A[i]);
    }

    // 深さ優先探索
    for (int i = 1; i <= N; i++) color[i] = 0;
    for (int i = 1; i <= N; i++) {
        if (color[i] == 0) {
            // 頂点 i は白である（まだ探索されていない連結成分である）
            color[i] = 1;
            dfs(i);
        }
    }

    // 二部グラフかどうかの判定
    bool Answer = true;
    for (int i = 1; i <= M; i++) {
        if (color[A[i]] == color[B[i]]) Answer = false;
    }
    if (Answer == true) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}

```

※ Pythonなどのソースコードは chap4-5.md をご覧ください。

問題 4.5.8

注意：この問題は、4.5.8 項「その他の代表的なグラフアルゴリズム」で紹介されたダイクストラ法を使います。初学者は解けなくて当然ですので、ご安心ください。

一般に、整数は「10 倍して 1 ~ 9 の値を足す」という操作の繰り返しによって作ることができます。たとえば整数 8691 は、

- 整数 0 から始める
- 10 倍して 8 を足す ($0 \times 10 + 8 = 8$ になる)
- 10 倍して 6 を足す ($8 \times 10 + 6 = 86$ になる)
- 10 倍して 9 を足す ($86 \times 10 + 9 = 869$ になる)
- 10 倍して 1 を足す ($869 \times 10 + 1 = 8691$ になる)

そこで、以下のような重み付き有向グラフを考えてみましょう。頂点 i ($0 \leq i < K$) は「 K で割って i 余る整数」を指します。

頂点について

- K 個の頂点を用意する。
- 頂点番号はそれぞれ $0, 1, 2, 3, \dots, K - 1$ とする。

辺について

各頂点 i ($0 \leq i < K$) について、以下の 10 個の辺を追加する。

- 頂点 i から頂点 $(10i + 0) \bmod K$ に向かう、重み 0 の辺※
- 頂点 i から頂点 $(10i + 1) \bmod K$ に向かう、重み 1 の辺
- 頂点 i から頂点 $(10i + 2) \bmod K$ に向かう、重み 2 の辺
- 頂点 i から頂点 $(10i + 3) \bmod K$ に向かう、重み 3 の辺
- 頂点 i から頂点 $(10i + 4) \bmod K$ に向かう、重み 4 の辺
- 頂点 i から頂点 $(10i + 5) \bmod K$ に向かう、重み 5 の辺
- 頂点 i から頂点 $(10i + 6) \bmod K$ に向かう、重み 6 の辺
- 頂点 i から頂点 $(10i + 7) \bmod K$ に向かう、重み 7 の辺
- 頂点 i から頂点 $(10i + 8) \bmod K$ に向かう、重み 8 の辺
- 頂点 i から頂点 $(10i + 9) \bmod K$ に向かう、重み 9 の辺

※ 実装の都合上、頂点 $0 \rightarrow 0$ の辺は例外的に追加しません。

ここで、一つの辺を通ることが 1 回の操作に対応します。たとえば、 $K = 13$ で 86 を 869 にする操作は、頂点 $8 \rightarrow 11$ に向かう重み 9 の辺を通過することに対応します。
 $(86 \bmod 13 = 8, 869 \bmod 13 = 11)$

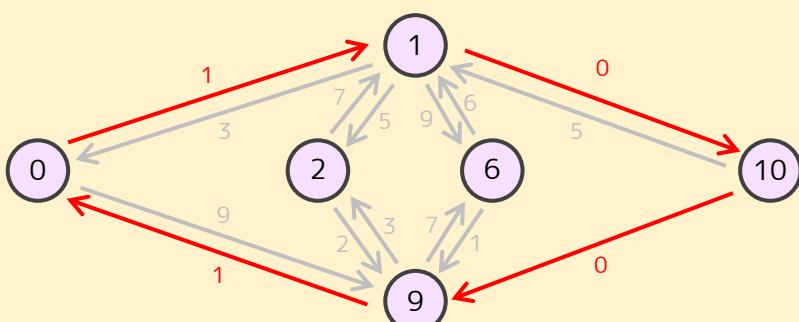


したがって、頂点 0 から頂点 i (≥ 1) までの最短経路長が、 K で割って i 余る数における各桁の和の最小値となります。

同様に、頂点 0 から頂点 0 へ戻る最短経路の長さが、 K の倍数における各桁の和の最小値になります。具体例は以下の通りです。

たとえば $K = 13$ の場合、各桁の和の最小値は 2 (1001 という数) ですが、これは $0 \rightarrow 1 \rightarrow 10 \rightarrow 9 \rightarrow 0$ という経路に対応します。（補足： $1 \bmod 13 = 1$ 、 $10 \bmod 13 = 10$ 、 $100 \bmod 13 = 9$ 、 $1001 \bmod 13 = 0$ ）

この経路は、前ページの方法で構成したグラフにおける最短経路になっています。（見やすさの都合上、一部の頂点と辺を省略しています）



よって、ダイクストラ法（→4.5.8項）を使って重み付きグラフの最短経路長を求めれば、正しい答えを出すことができます。

ただし、自然に実装すると頂点 0 から 0 までの最短経路長は 0 になってしまうので、少し工夫を行う必要があります。詳しくは、次ページの実装例を参考にしてください。

※ より丁寧な公式解説もご覧ください。（リンクは chap4-5.md にあります）

```

#include <bits/stdc++.h>
using namespace std;

int K, dist[100009];
bool used[100009];
vector<pair<int, int>> G[100009];
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> Q;

// ダイクストラ法
void dijkstra() {
    // 配列の初期化など
    for (int i = 0; i < K; i++) dist[i] = (1 << 30);
    for (int i = 0; i < K; i++) used[i] = false;
    Q.push(make_pair(0, 0)); // ここで dist[0] = 0 にはしないことに注意！

    // キューの更新
    while (!Q.empty()) {
        int pos = Q.top().second; Q.pop();
        if (used[pos] == true) continue;
        used[pos] = true;
        for (pair<int, int> i : G[pos]) {
            int to = i.first, cost = dist[pos] + i.second;
            if (pos == 0) cost = i.second; // 頂点 0 の場合は例外
            if (dist[to] > cost) {
                dist[to] = cost;
                Q.push(make_pair(dist[to], to));
            }
        }
    }
}

int main() {
    // 入力
    cin >> K;

    // グラフの辺を追加
    for (int i = 0; i < K; i++) {
        for (int j = 0; j < 10; j++) {
            if (i == 0 && j == 0) continue;
            G[i].push_back(make_pair((i * 10 + j) % K, j));
        }
    }

    // ダイクストラ法・出力
    dijkstra();
    cout << dist[0] << endl;
    return 0;
}

```

※ Python などのソースコードは chap4-5.md をご覧ください。

4.6

節末問題 4.6 の解答

問題 4.6.1 (1)

掛け算の式はどのようなタイミングで余りをとっても答えが変わらないため、

- $21 \times 41 \times 61 \times 81 \times 101 \times 121$ を 20 で割った余り
- $1 \times 1 \times 1 \times 1 \times 1 \times 1$ を 20 で割った余り

は等しいです。後者は明らかに 1 なので、この問題の答えは **1** です。

問題 4.6.1 (2)

計算前にすべて 100 で割った余りをとっても答えは変わらないため、

- 202112^5 を 100 で割った余り
- 12^5 を 100 で割った余り

は一致します。 $12^5 = 248832$ なので答えは **32** だと分かりますが、手計算では少々面倒です。そこで、以下のように計算途中でも余りをとりましょう。

- $12 \times 12 = 144 \equiv 44 \pmod{100}$
- $44 \times 12 = 528 \equiv 28 \pmod{100}$
- $28 \times 12 = 336 \equiv 36 \pmod{100}$
- $36 \times 12 = 432 \equiv 32 \pmod{100}$

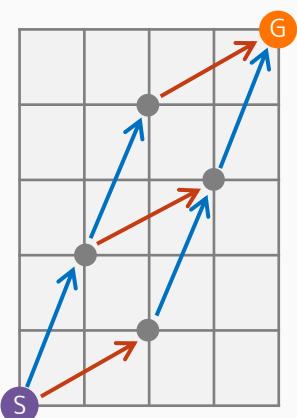
そうすると、高々 3 行の数の計算で答えを求めることができます。

問題 4.6.2

まずは具体例から考えましょう。コマをマス (4, 5) まで移動させるには、

- $(i, j) \rightarrow (i + 1, j + 2)$ の移動を 2 回
- $(i, j) \rightarrow (i + 2, j + 1)$ の移動を 1 回

行う必要があります。逆にその条件さえ満たせば必ず目的の場所にたどり着きます。3 回中 2 回が $(i + 1, j + 2)$ への移動なので、移動方法の数は ${}_3C_2 = 3$ 通りです。



次に、一般化したケースを考えましょう。

- $(i, j) \rightarrow (i + 1, j + 2)$ の移動を a 回（移動 A とする）
- $(i, j) \rightarrow (i + 2, j + 1)$ の移動を b 回（移動 B とする）

行うとき、マス (X, Y) に移動させるには以下の 3 条件を満たす必要があります。

- a, b は非負整数である。
- x 座標の制約： $a + 2b = X$
- y 座標の制約： $2a + b = Y$

ここで、2 つ目・3 つ目を同時に満たす組は、 $(a, b) = \left(\frac{2Y-X}{3}, \frac{2X-Y}{3}\right)$ のみです。

しかし、1 つ目の条件より、答えが 0 通りになる場合があります。

- a, b は整数なので、 $2Y - X, 2X - Y$ が両方 3 の倍数でなければ 0 通り
- a, b は 0 以上なので、 $2Y - X < 0$ または $2X - Y < 0$ であれば 0 通り

そうでない場合は、全体の移動回数 $(X + Y)/3$ 回のうち $(2Y - X)/3$ 回を移動 A にすれば必ず目的のマスにたどり着けるため、求める答えは $\binom{X+Y}{3} C_{(2Y-X)/3}$ 通りとなります。コード 4.6.5 を少し変えた以下のような実装をすると、正解が得られます。

```
#include <iostream>
using namespace std;

const long long mod = 1000000007;
int X, Y;

long long modpow(long long a, long long b, long long m) {
    // 繰り返し二乗法 (p は a^1, a^2, a^4, a^8, ... といった値をとる)
    long long p = a, Answer = 1;
    for (int i = 0; i < 30; i++) {
        if ((b & (1 << i)) != 0) { Answer *= p; Answer %= m; }
        p *= p; p %= m;
    }
    return Answer;
}

// Division(a, b, m) は a ÷ b mod m を返す関数
long long Division(long long a, long long b, long long m) {
    return (a * modpow(b, m - 2, m)) % m;
}
```

```

int main() {
    // 入力
    cin >> X >> Y;

    // 場合分け (a, b が負になってしまう場合)
    if (2 * Y - X < 0 || 2 * X - Y < 0) {
        cout << "0" << endl;
        return 0;
    }

    // 場合分け (a, b が整数にならない場合)
    if ((2 * Y - X) % 3 != 0 || (2 * X - Y) % 3 != 0) {
        cout << "0" << endl;
        return 0;
    }

    // 二項係数の分子と分母を求める (手順 1. / 手順 2.)
    long long bunshi = 1, bunbo = 1;
    long long a = (2 * Y - X) / 3, b = (2 * X - Y) / 3;
    for (int i = 1; i <= a + b; i++) { bunshi *= i; bunshi %= mod; }
    for (int i = 1; i <= a; i++) { bunbo *= i; bunbo %= mod; }
    for (int i = 1; i <= b; i++) { bunbo *= i; bunbo %= mod; }

    // 答えを求める (手順 3.)
    cout << Division(bunshi, bunbo, mod) << endl;
    return 0;
}

```

※ Python などのソースコードは chap4-6.md をご覧ください。

問題 4.6.3

まず、以下の式が成り立ちます。たとえば $N = 2$ のとき $4^0 + 4^1 + 4^2 = 1 + 4 + 16 = 21$ である一方、 $(4^{N+1} - 1)/3 = 63 \div 3 = 21$ であり、2つの値が一致します。

$$4^0 + 4^1 + 4^2 + \cdots + 4^N = \frac{4^{N+1} - 1}{3}$$

証明はやや難しいですが、長さ $4^{N+1}/3$ の棒を $3/4$ だけ切り取る操作を $N+1$ 回繰り返すと、1回目から順に長さ $4^N, 4^{N-1}, \dots, 4^0$ のものが取り出され、最終的に長さ $1/3$ だけ残ることを考えると良いです。（参考：和の公式 → 2.5.10項）



したがって、以下のような方法で答えを $M = 1000000007$ で割った余りを求めることができます。

- $4^{N+1} - 1$ を M で割った余り V を求める (\rightarrow **4.6.7項**)
- $V \div 3$ を M で割った余りを求める (\rightarrow **4.6.8項**)

実装例は以下の通りです。なお、関数 `Division(a, b, m)` は、 $a \div b$ を m で割った余りを返すものとなっています。また、制約が $N \leq 10^{18}$ であるため、`modpow` 関数のループ回数が $\log_2(10^{18}) \approx 60$ 回程度必要であることに注意してください。

```
#include <iostream>
using namespace std;

const long long mod = 1000000007;
long long N;

long long modpow(long long a, long long b, long long m) {
    // 繰り返し二乗法 (p は a^1, a^2, a^4, a^8, ... といった値をとる)
    long long p = a, Answer = 1;
    for (int i = 0; i < 60; i++) {
        if ((b & (1LL << i)) != 0) { Answer *= p; Answer %= m; }
        p *= p; p %= m;
    }
    return Answer;
}

// Division(a, b, m) は a÷b mod m を返す関数
long long Division(long long a, long long b, long long m) {
    return (a * modpow(b, m - 2, m)) % m;
}

int main() {
    // 入力
    cin >> N;

    // 答えの計算
    long long V = modpow(4, N + 1, mod) - 1;
    long long Answer = Division(V, 3, mod);

    // 出力
    cout << Answer << endl;
    return 0;
}
```

※ Python などのソースコードは chap4-6.md をご覧ください。

4.7

節末問題 4.7 の解答

問題 4.7.1

まず、行列の掛け算は以下のようになります（行列も整数と同様、掛け算を優先して計算します）。分からぬ人は、4.7.3 項に戻って確認しましょう。

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 2 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} [1 \ 1 \ 1] = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

求める答えは、青く示されている 2 つの 2×3 行列の和なので、以下の通りです。

$$\begin{bmatrix} 2 & 0 & 2 \\ 1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 3 \\ 3 & 3 & 3 \end{bmatrix}$$

問題 4.7.2

まず、 $a_3 = 2a_2 + a_1$ 、 $a_2 = a_2$ であることから、以下の式が成り立ちます。

$$\begin{bmatrix} a_3 \\ a_2 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_2 \\ a_1 \end{bmatrix}$$

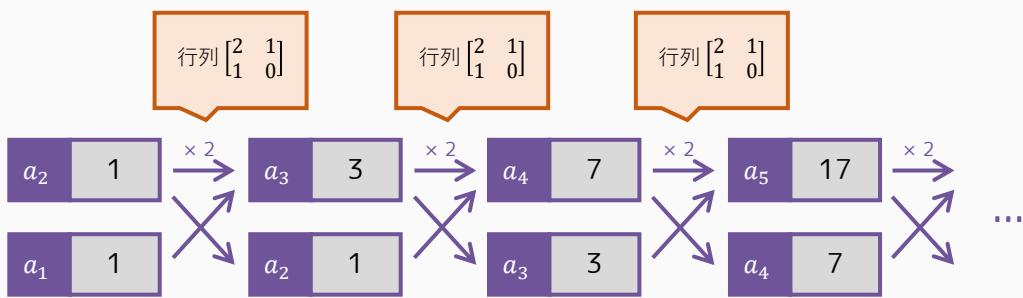
同様に、 $a_4 = 2a_3 + a_2$ 、 $a_3 = a_3$ であることから、以下の式が成り立ちます。

$$\begin{bmatrix} a_4 \\ a_3 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_3 \\ a_2 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_3 \\ a_2 \end{bmatrix}$$

同じような計算を a_5 以降についても繰り返すと、以下のようになります。このことから、 A を $2, 1, 1, 0$ からなる行列とするとき、 a_N の値が A^{N-1} の (2, 1) 成分と (2, 2) 成分を足した値であることが分かります。

$$\begin{bmatrix} a_{N+1} \\ a_N \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}^N \begin{bmatrix} a_2 \\ a_1 \end{bmatrix} = A^{N-1} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

以下の図は、行列と数列 $a = (a_1, a_2, a_3, \dots)$ の関係を表したものです。



したがって、以下のような実装をすると、正しい答えを出すことができます。なお、コード 4.7.1 と異なる部分を濃い青色で示しています。（逆にいえば、このプログラムはコード 4.7.1 とほぼ同一です。）

```
#include <iostream>
using namespace std;

struct Matrix {
    long long p[2][2] = { {0, 0}, {0, 0} };
};

Matrix Multiplication(Matrix A, Matrix B) { // 2×2 行列 A, B の積を返す関数
    Matrix C;
    for (int i = 0; i < 2; i++) {
        for (int k = 0; k < 2; k++) {
            for (int j = 0; j < 2; j++) {
                C.p[i][j] += A.p[i][k] * B.p[k][j];
                C.p[i][j] %= 1000000007;
            }
        }
    }
    return C;
}

Matrix Power(Matrix A, long long n) { // A の n 乗を返す関数
    Matrix P = A, Q;
    bool flag = false;
    for (int i = 0; i < 60; i++) {
        if ((n & (1LL << i)) != 0LL) {
            if (flag == false) { Q = P; flag = true; }
            else { Q = Multiplication(Q, P); }
        }
        P = Multiplication(P, P);
    }
    return Q;
}

int main() {
    // 入力 → 累乗の計算 (N が 2 以上でなければ正しく動作しないので注意
}
```

```

long long N;
cin >> N;

// 行列 A の作成
Matrix A;
A.p[0][0] = 2; A.p[0][1] = 1; A.p[1][0] = 1;

// B=A^{N-1} の計算
Matrix B = Power(A, N - 1);

// 答えの出力
cout << (B.p[1][0] + B.p[1][1]) % 1000000007 << endl;
return 0;
}

```

※ Python などのソースコードは chap4-7.md をご覧ください。

問題 4.7.3

まず、 $a_4 = a_3 + a_2 + a_1, a_3 = a_3, a_2 = a_2$ より、以下の式が成り立ちます。

$$\begin{bmatrix} a_4 \\ a_3 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_3 \\ a_2 \\ a_1 \end{bmatrix}$$

同様に、 $a_5 = a_4 + a_3 + a_2, a_4 = a_4, a_3 = a_3$ より、以下の式が成り立ちます。

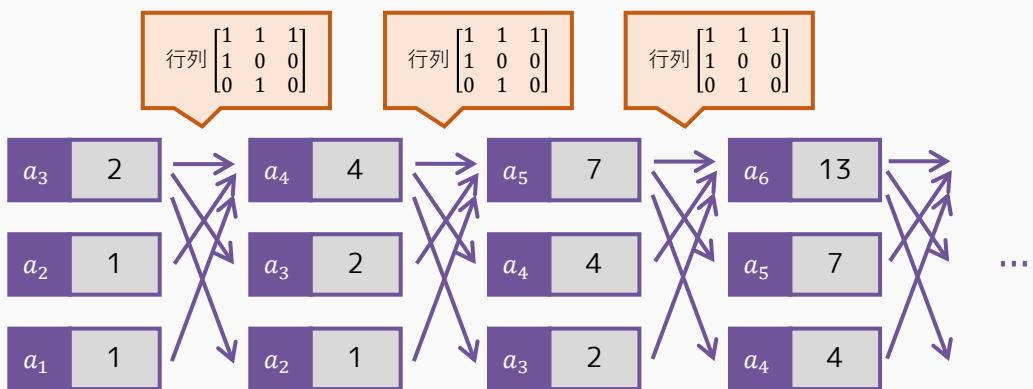
$$\begin{bmatrix} a_5 \\ a_4 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_4 \\ a_3 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^2 \begin{bmatrix} a_3 \\ a_2 \\ a_1 \end{bmatrix}$$

同じような計算を a_6 以降も繰り返すと、以下のようになります。ただし、 $1, 1, 1, 1, 0, 0, 0, 1, 0$ からなる 3×3 行列を A とします。

$$\begin{bmatrix} a_{N+2} \\ a_{N+1} \\ a_N \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^{N-1} \begin{bmatrix} a_3 \\ a_2 \\ a_1 \end{bmatrix} = A^{N-1} \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$$

このことから、 a_N の値が $[A^{N-1} の (3, 1) 成分 \times 2 + (3, 2) 成分 + (3, 3) 成分]$ で表されることが分かります。

以下の図は、行列と数列 $a = (a_1, a_2, a_3, \dots)$ の関係を表したものです。



したがって、以下のような実装をすると、正しい答えを出すことができます。なお、行列の大きさが 3×3 になっていることに注意してください。（コード 4.7.1 と異なる部分を濃い青色で示しています。）

```
#include <iostream>
using namespace std;

struct Matrix {
    long long p[3][3] = { {0, 0, 0}, {0, 0, 0}, {0, 0, 0} };
};

Matrix Multiplication(Matrix A, Matrix B) { // 3×3 行列 A, B の積を返す関数
    Matrix C;
    for (int i = 0; i < 3; i++) {
        for (int k = 0; k < 3; k++) {
            for (int j = 0; j < 3; j++) {
                C.p[i][j] += A.p[i][k] * B.p[k][j];
                C.p[i][j] %= 1000000007;
            }
        }
    }
    return C;
}

Matrix Power(Matrix A, long long n) { // A の n 乗を返す関数
    Matrix P = A, Q;
    bool flag = false;
    for (int i = 0; i < 60; i++) {
        if ((n & (1LL << i)) != 0LL) {
            if (flag == false) { Q = P; flag = true; }
            else { Q = Multiplication(Q, P); }
        }
        P = Multiplication(P, P);
    }
    return Q;
}
```

```

int main() {
    // 入力 → 累乗の計算 (N が 2 以上でなければ正しく動作しないので注意)
    long long N;
    cin >> N;

    // 行列 A の作成
    Matrix A;
    A.p[0][0] = 1; A.p[0][1] = 1; A.p[0][2] = 1; A.p[1][0] = 1; A.p[2][1] = 1;

    // B=A^{N-1} の計算
    Matrix B = Power(A, N - 1);

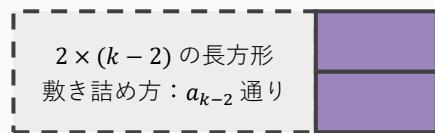
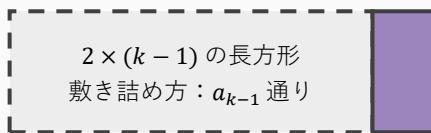
    // 答えの出力
    cout << (2LL * B.p[2][0] + B.p[2][1] + B.p[2][2]) % 1000000007 << endl;
    return 0;
}

```

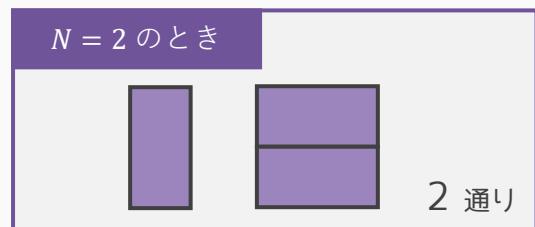
※ Python などのソースコードは chap4-7.md をご覧ください。

問題 4.7.4 (1)

$2 \times k$ の長方形を左から順番に敷き詰めていくとき、最後に置くピースは以下の 2 つのうちいずれかになります。 ($k \geq 2$ の場合)



したがって、 $2 \times k$ の長方形を敷き詰める方法の数を a_k とすると、 $a_k = a_{k-1} + a_{k-2}$ という漸化式が成り立ちます。また、 $N = 1$ のときの答えは $a_1 = 1$ 、 $N = 2$ のときの答えは $a_2 = 1$ です。

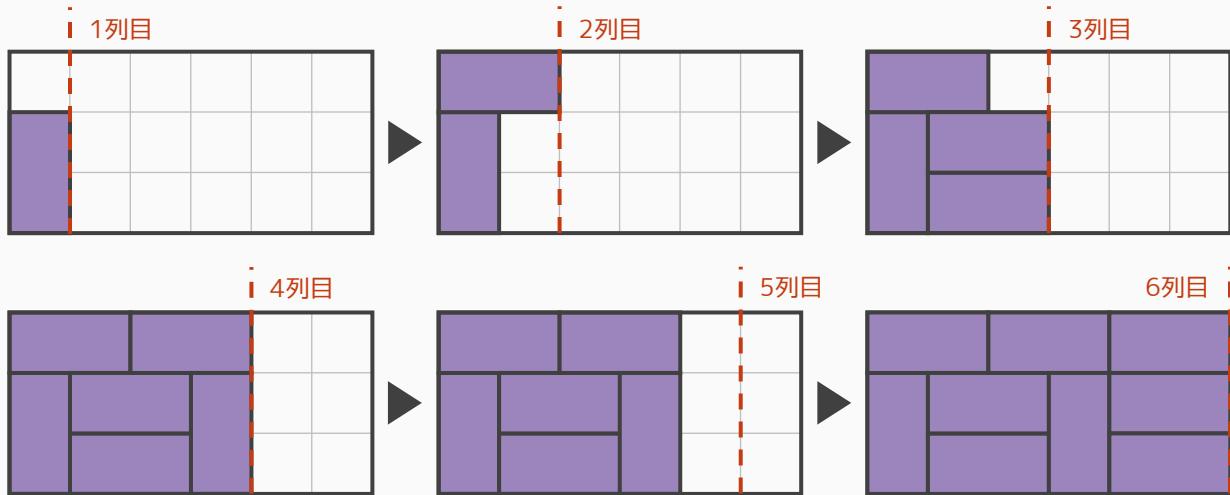


よって、 $2 \times N$ のときの答えは以下のようにフィボナッチ数の第 $N + 1$ 項となるため、それを出力するプログラム（→4.7.1項）を作成すれば正解が得られます。

N	1	2	3	4	5	6	7	8
a_N	1	2	3	5	8	13	21	34

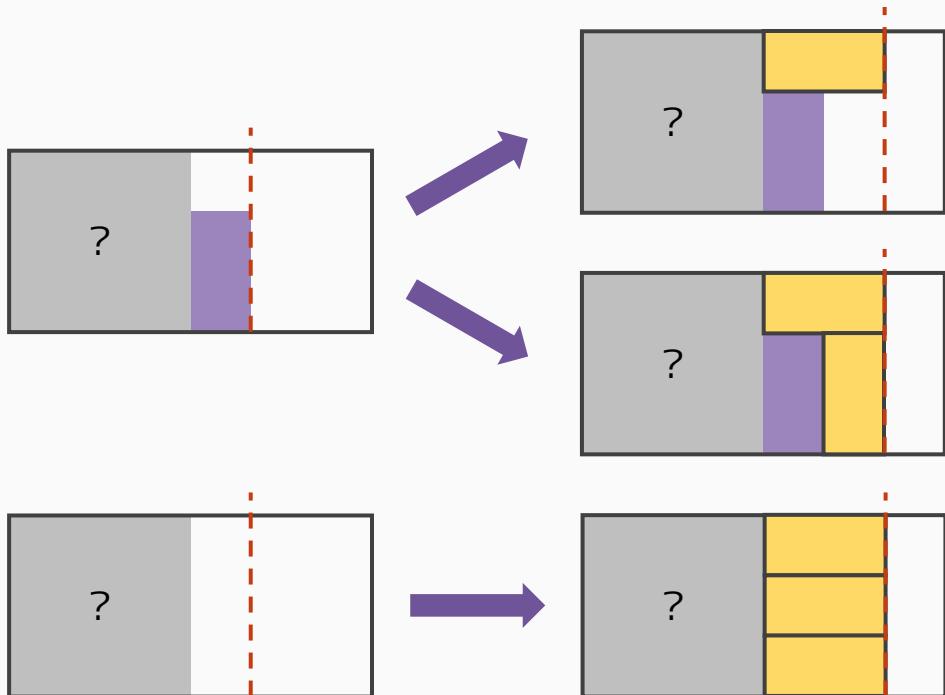
問題 4.7.4 (2), (3)

以下のように、左から一列ずつ敷き詰めていくことを考えましょう。 x 列目までの敷き詰めを考えるとき、 x 列目と $x + 1$ 列目にまたがるピースは含めないものとします。

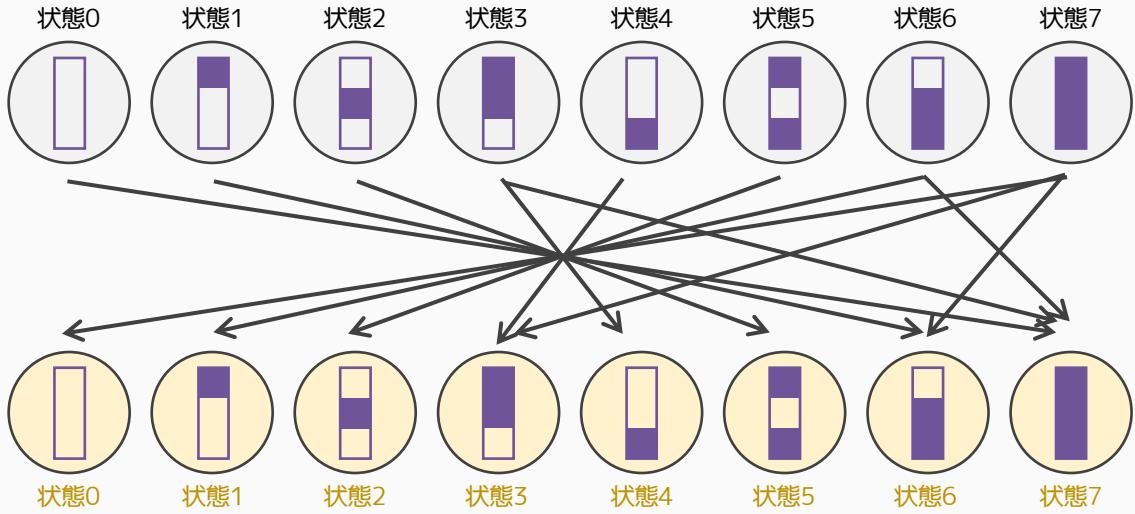


ここで、「 x 列目までの敷き詰めでは必ず $x - 1$ 列目までがすべて埋まっている」という重要な性質が成り立つため、この後の敷き詰め方は最後の列 (x 列目) にのみ依存します。

たとえば、 $K = 3$ で下の 2 行にのみ既に置かれていた場合、以下の 2 通りの置き方があります。その他の場合も同様に考えることができます。



さて、「 x 列目の段階で最後の列が○○のとき、 $x + 1$ 列目の段階で最後の列が△△であるようにする方法は何通りあるか？」ということを考えましょう。例えば $K = 3$ の場合は下図のとおりであり、矢印 1 個につき 1 通りです。



そこで、最後の列の状態を $0, 1, \dots, 2^K - 1$ と番号を振るとき、 $dp[x][y]$ を「 x 列目まで敷き詰めた時点で状態 y である場合の数」とすると、以下の式が成り立ちます。

ただし、 $A_{p,q}$ は状態 p から状態 q に遷移する方法の数（たとえば矢印が引かれていなければ $A_{p,q} = 0$ ）とします。また、 $L = 2^K - 1$ とします。

$$\begin{bmatrix} dp[x+1][0] \\ dp[x+1][1] \\ dp[x+1][2] \\ \vdots \\ dp[x+1][L] \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \cdots & A_{0,L} \\ A_{1,0} & A_{1,1} & A_{1,2} & \cdots & A_{1,L} \\ A_{2,0} & A_{2,1} & A_{2,2} & \cdots & A_{2,L} \\ \vdots & \vdots & \vdots & & \vdots \\ A_{L,0} & A_{L,1} & A_{L,2} & \cdots & A_{L,L} \end{bmatrix} \begin{bmatrix} dp[x][0] \\ dp[x][1] \\ dp[x][2] \\ \vdots \\ dp[x][L] \end{bmatrix}$$

この式は少し複雑ですが、たとえば左辺の $(1, 1)$ 成分である $dp[x+1][0]$ の値は、「 x 列目まで敷き詰めて状態○○である場合の数 × 状態○○から状態 0 に遷移する方法の数」をすべての状態について足し合わせたものだと考えれば良いです。

したがって、 $dp[N][0], dp[N][1], \dots, dp[N][N-1]$ の値は以下のようになります。（上の式を繰り返し適用すれば良いです）

$$\begin{bmatrix} dp[N][0] \\ dp[N][1] \\ dp[N][2] \\ \vdots \\ dp[N][L] \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \cdots & A_{0,L} \\ A_{1,0} & A_{1,1} & A_{1,2} & \cdots & A_{1,L} \\ A_{2,0} & A_{2,1} & A_{2,2} & \cdots & A_{2,L} \\ \vdots & \vdots & \vdots & & \vdots \\ A_{L,0} & A_{L,1} & A_{L,2} & \cdots & A_{L,L} \end{bmatrix}^N \begin{bmatrix} dp[0][0] \\ dp[0][1] \\ dp[0][2] \\ \vdots \\ dp[0][L] \end{bmatrix} = A^N \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

たとえば $K = 3$ の場合、 $dp[N][0], dp[N][1], dp[N][2], \dots, dp[N][7]$ の値は以下の通りです。（ $K = 4$ の場合は長くなるため省略します）

$$\begin{bmatrix} dp[N][0] \\ dp[N][1] \\ dp[N][2] \\ dp[N][3] \\ dp[N][4] \\ dp[N][5] \\ dp[N][6] \\ dp[N][7] \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}^N \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

そこで最後の行（ N 行目）はすべて埋まつていなければならないので、求める答えは $dp[N][L] = dp[N][2^K - 1]$ です。

よって、繰り返し二乗法を使って $2^K \times 2^K$ 行列の累乗を求めてことで、計算量 $O((2^K)^3 \times \log N)$ でこの問題を解くことができました。以下は C++ の実装例です。

```
#include <iostream>
using namespace std;

// K=2 の場合の遷移
long long Mat2[4][4] = {
    {0, 0, 0, 1},
    {0, 0, 1, 0},
    {0, 1, 0, 0},
    {1, 0, 0, 1}
};

// K=3 の場合の遷移
long long Mat3[8][8] = {
    {0, 0, 0, 0, 0, 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 1, 0},
    {0, 0, 0, 0, 0, 1, 0, 0},
    {0, 0, 0, 0, 1, 0, 0, 1},
    {0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 1, 0, 0, 0, 0, 0},
    {0, 1, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 1, 0, 0, 1, 0}
};

// K=4 の場合の遷移
long long Mat4[16][16] = {
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0}
};
```

```

    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},  

    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},  

    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},  

    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1},  

    {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},  

    {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},  

    {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},  

    {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

    {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

    {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},  

    {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0},  

    {0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0},  

    {1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1}
};

struct Matrix {
    int size_ = 0; // 行列の大きさ (size_ × size_ の正方行列とする)
    long long p[16][16];
};

Matrix Multiplication(Matrix A, Matrix B) { // 行列 A, B の積を返す関数
    Matrix C;

    // 行列 C の初期化
    C.size_ = A.size_;
    for (int i = 0; i < A.size_; i++) {
        for (int j = 0; j < A.size_; j++) C.p[i][j] = 0;
    }

    // 行列の掛け算
    for (int i = 0; i < A.size_; i++) {
        for (int k = 0; k < A.size_; k++) {
            for (int j = 0; j < A.size_; j++) {
                C.p[i][j] += A.p[i][k] * B.p[k][j];
                C.p[i][j] %= 1000000007;
            }
        }
    }
    return C;
}

Matrix Power(Matrix A, long long n) { // A の n 乗を返す関数
    Matrix P = A, Q;
    bool flag = false;
    for (int i = 0; i < 60; i++) {
        if ((n & (1LL << i)) != 0LL) {
            if (flag == false) { Q = P; flag = true; }
            else { Q = Multiplication(Q, P); }
        }
        P = Multiplication(P, P);
    }
    return Q;
}

```

```
int main() {
    // 入力
    long long K, N;
    cin >> K >> N;

    // 行列 A の作成
    Matrix A; A.size_ = (1 << K);
    for (int i = 0; i < (1 << K); i++) {
        for (int j = 0; j < (1 << K); j++) {
            if (K == 2) A.p[i][j] = Mat2[i][j];
            if (K == 3) A.p[i][j] = Mat3[i][j];
            if (K == 4) A.p[i][j] = Mat4[i][j];
        }
    }

    // B=A^N の計算
    Matrix B = Power(A, N);

    // 答えの出力
    cout << B.p[(1 << K) - 1][(1 << K) - 1] << endl;
    return 0;
}
```

第 5 章

問題解決のための
数学的考察

5.2

節末問題 5.2 の解答

問題 5.2.1

フィボナッチ数の第 12 項までと、それらを 4 で割った余りは以下の通りです。

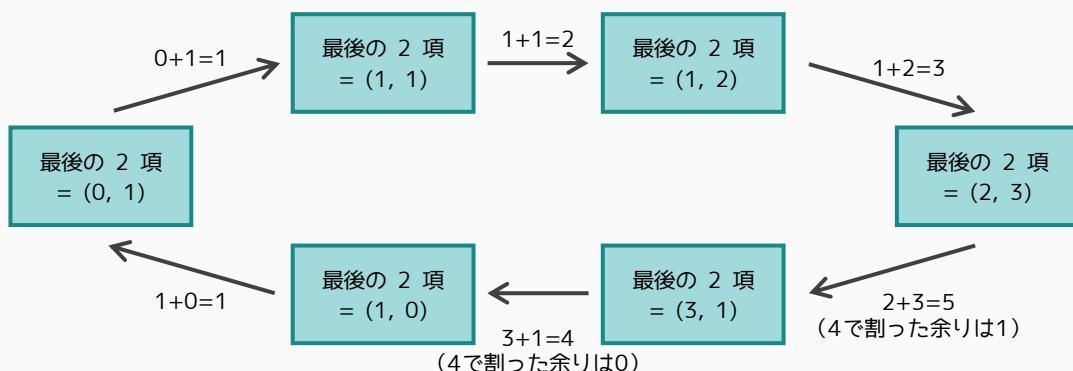
$1 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 0 \rightarrow \dots$ が周期的に繰り返されていますね。

N	1	2	3	4	5	6	7	8	9	10	11	12
第 N 項	1	1	2	3	5	8	13	21	34	55	89	144
4 で割った余り	1	1	2	3	1	0	1	1	2	3	1	0

この周期性は N が大きくなっても成り立つのでしょうか。実は

- ・ フィボナッチ数において、項の値は直前の二項のみから決まること
- ・ (第 1 項, 第 2 項) と (第 7 項, 第 8 項) が一致していること

から証明できます。イメージ図は以下の通りです。



したがって、フィボナッチ数の第 N 項を 4 で割った余りは、第 $(N \bmod 6)$ 項を 4 で割った余りと一致します (N が 6 の倍数は第 6 項と一致)。

$10000 \bmod 6 = 4$ なので、フィボナッチ数の第 10000 項は、第 4 項を 4 で割った余りである 3 となります。

(解説は次ページへ続きます)

問題 5.2.2

まず、石が 1 個の状態から石を取ることはできないため、 $N = 1$ は負けの状態（後手必勝）です。一方、石が 2 個のときは先手が 1 個の石を取ると後手が手を打てなくなるので、 $N = 2$ は勝ちの状態です。

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
状態	負	勝													

次に $N = 3$ の場合を考えましょう。一般に、ゲームは負けの状態に遷移可能な場合のみ勝つことができます（→5.2.2項）。しかし、「石を 1 個取り、残り 2 個（勝ちの状態）に減らす」という操作しかできないため、 $N = 3$ は負けの状態です。

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
状態	負	勝	負												



次に、石が 4, 5, 6 個の状態からは一手で石の数を 3 個（負けの状態）に減らすことができるため、それらは勝ちの状態です。一方、石が 7 個の状態からは、

- 石を 1 個取り、石の数を 6 個に減らす
- 石を 2 個取り、石の数を 5 個に減らす
- 石を 3 個取り、石の数を 4 個に減らす

という 3 つの操作方法がありますが、すべて勝ちの状態に遷移します。よって、 $N = 7$ は負けの状態です。

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
状態	負	勝	負	勝	勝	勝	負								



同じように考察を進めていくと、 $N = 8, 9, 10, 11, 12, 13, 14$ は勝ちの状態、 $N = 15$ は負けの状態であることが分かります。

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
状態	負	勝	負	勝	勝	勝	負	勝	勝	勝	勝	勝	勝	勝	負



ここまで時点では $1, 3, 7, 15$ 個が負けの状態であるため、勘の良い人は「 $2^k - 1$ で表される数だけが負けの状態ではないか」という周期性が頭に浮かぶと思います。（浮かばなかった人は、 $N = 16$ 以降も調べてみてください）

実は、この周期性は N が大きくなても成り立ちます（証明略）。したがって、 $N = 2^k - 1$ となるかどうかを $1 \leq k \leq 60$ の範囲で全探索する以下のようなプログラムを書くと、正解が得られます。

なお、本問題の制約は $N \leq 10^{18}$ であり、 $2^{60} > 10^{18}$ であるため、 $k \leq 60$ までの探索で十分です）

```
#include <iostream>
using namespace std;

int main() {
    // 入力
    long long N;
    cin >> N;

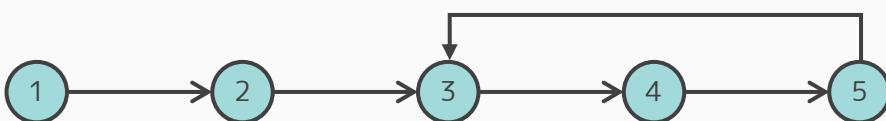
    // N = 2^k-1 の形で表されるかどうかを調べる
    bool flag = false;
    for (int k = 1; k <= 60; k++) {
        if (N == (1LL << k) - 1LL) flag = true;
    }

    // 出力
    if (flag == true) cout << "Second" << endl;
    else cout << "First" << endl;
    return 0;
}
```

※ Python などのソースコードは chap5-2.md をご覧ください。

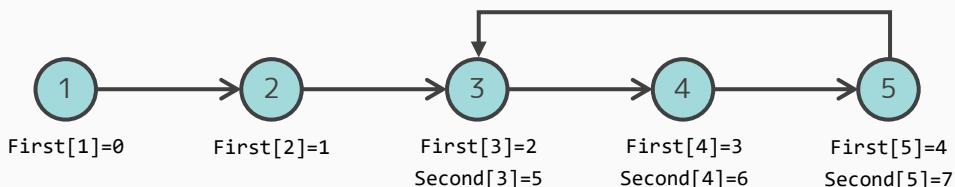
問題 5.2.3

この問題は少し複雑なので、まずは $N = 5, A = (2, 3, 4, 5, 3)$ の場合を考えましょう。テレポータの転送は下図のようになり、町 1 から出発した場合、 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow \dots$ と周期的に移動します。



この周期性は一般的なケースでも成り立ちます。 N 回以内の移動で、既に訪問した町に戻ってきて、その後は周期的な移動を行うことが証明できます。

ここで、最初に町 u に訪れたときに $\text{First}[u]$ 回テレポーターを使っており、二度目に訪れたときに $\text{Second}[u]$ 回テレポーターを使っているとしましょう。（下図は具体例となります）



$L = \text{Second}[u] - \text{First}[u]$ （周期の長さ）とすると、町 u にはテレポーターを $\text{First}[u], \text{First}[u] + L, \text{First}[u] + 2L, \dots$ 回使ったときに訪問します。上の例では、

- 町 3：テレポーターを $2, 5, 8, 11, 14, 17, \dots$ 回使ったときに訪問
- 町 4：テレポーターを $3, 6, 9, 12, 15, 18, \dots$ 回使ったときに訪問
- 町 5：テレポーターを $4, 7, 10, 13, 16, 19, \dots$ 回使ったときに訪問

となります。

したがって、 $(K - \text{First}[u]) \bmod L = 0$ のとき、 K 回の移動で町 u に到着します。このような u を調べる以下ののようなプログラムを提出すると、正解が得られます。

```
#include <iostream>
using namespace std;

long long N, K;
long long A[200009];
long long First[200009], Second[200009];

int main() {
    // 入力
    cin >> N >> K;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 配列の初期化
    for (int i = 1; i <= N; i++) First[i] = -1;
    for (int i = 1; i <= N; i++) Second[i] = -1;

    // 答えを求める
    // cur は現在いる町の番号
    long long cnt = 0, cur = 1;
    while (true) {
        // First, Second の更新
        if (cur == 1) {
            First[cur] = cnt;
            Second[cur] = cnt;
        } else {
            if (First[cur] != -1) {
                Second[cur] = cnt;
            }
        }
        cur = (cur + A[cur]) % N;
        if (cur == 1) {
            if (First[cur] == -1) {
                First[cur] = cnt;
            }
        }
        if (cnt == K) break;
        cnt++;
    }
}
```

```
if (First[cur] == -1) First[cur] = cnt;
else if (Second[cur] == -1) Second[cur] = cnt;

// K 回の移動後に町 cur にいるかどうかの判定
if (cnt == K) {
    cout << cur << endl;
    return 0;
}
else if (Second[cur] != -1 && (K-First[cur]) % (Second[cur]-First[cur]) == 0) {
    cout << cur << endl;
    return 0;
}

// 位置の更新
cur = A[cur];
cnt += 1;
}
return 0;
}
```

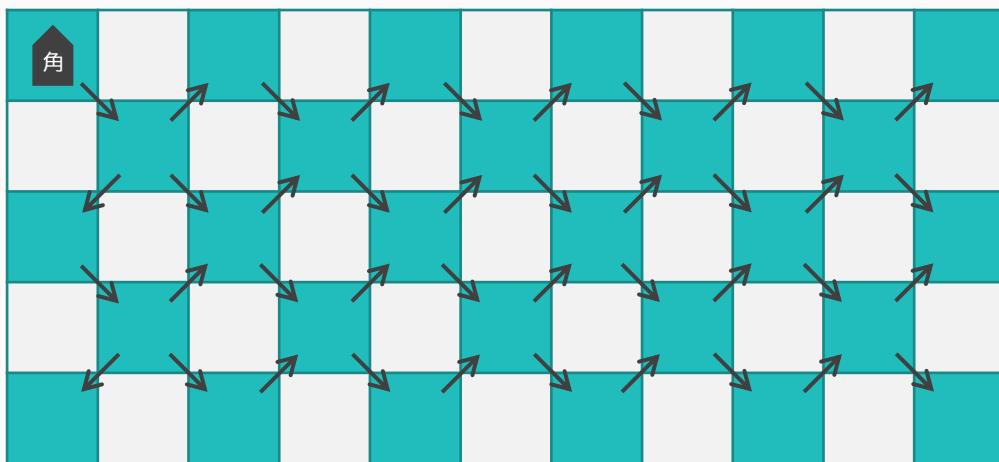
※ Python などのソースコードは chap5-2.md をご覧ください。

5.3

節末問題 5.3 の解答

問題 5.3.1

まず、 $H = 5, W = 11$ のケースを考えてみましょう。上から x 行目、左から y 列目のマスを (x, y) とするとき、 $x + y$ が偶数のマスにのみたどり着けます。



実は、 $H \geq 2, W \geq 2$ の場合は必ず $x + y$ が偶数のマス（全部で $[HW/2]$ 個）に限り到達可能です。奇数のマスに移動できない理由は以下の通りです。

角行は斜め方向への移動なので、隣り合うマスのみを考えると、

- マス $(x, y) \rightarrow$ マス $(x + 1, y + 1)$
- マス $(x, y) \rightarrow$ マス $(x + 1, y - 1)$
- マス $(x, y) \rightarrow$ マス $(x - 1, y + 1)$
- マス $(x, y) \rightarrow$ マス $(x - 1, y - 1)$

の移動ができる。しかし、 $[x$ 座標] + [y 座標] の値の増加は $-2, 0, 2$ のいずれかとなるため、偶奇は変わらない。

スタート位置（左上マス）については $[x$ 座標] + [y 座標] = 2（偶数）なので、奇数のマスには移動できない。

したがって、次ページのように実装すると正解が得られます。なお、 $H = 1$ または $W = 1$ のとき、答えが 1 であることに注意してください。このような場合分けを必要とするケースを「コーナーケース」といいます。

```
#include <iostream>
using namespace std;

int main() {
    // 入力
    long long H, W;
    cin >> H >> W;

    // 場合分け
    if (H == 1 || W == 1) {
        cout << "1" << endl;
    }
    else {
        cout << (H * W + 1) / 2 << endl;
    }
    return 0;
}
```

※ Python などのソースコードは chap5-3.md をご覧ください。

問題 5.3.2

以下の手順で数の選び方を決めていくことを考えましょう。

- **手順 1**： 2, 3, 4, 5, 6, 7, 8, 9, 10 の選び方を決める
- **手順 2**： 1 の選び方を決める

まず、手順 1 における選び方は全部で $2^9 = 512$ 通りあります（→3.3.2項）。一方、手順 1 が終わった時点で、最終的な選んだ数の総和を奇数にするような手順 2 の選び方は必ず 1 通りだけ存在します。



したがって、求める答えは $512 \times 1 = \boxed{512 \text{ 通り}}$ (全体のちょうど半分) です。

5.4

節末問題 5.4 の解答

問題 5.4.1

まず、「一つ以上 6 の目が出ること」の余事象は「すべて 5 以下」であるため、

$$(\text{一つ以上 } 6 \text{ の目が出る確率}) = 1 - (\text{すべて } 5 \text{ 以下となる確率})$$

という式が成り立ちます。そこで、すべて 5 以下となる確率は、積の法則（→3.3.2 項）より $(5/6) \times (5/6) \times (5/6) = 125/216$ です。よって、答えは以下の通りです。

$$1 - \frac{125}{216} = \frac{\textcolor{red}{91}}{\textcolor{red}{216}}$$

問題 5.4.2

5.4.4 項で解説された通りに実装すれば良いです。実装例として以下が考えられます。

なお、各変数・配列は以下ののような意味を持っています。

- `gyou[i]` : i 行目の総和
- `retu[j]` : j 列目の総和
- `Answer[i][j]` : i 行目・ j 列目のマスに対する答え

```
#include <iostream>
using namespace std;

int H, W, A[2009][2009];
int gyou[2009]; // 行の総和
int retu[2009]; // 列の総和
int Answer[2009][2009];

int main() {
    // 入力
    cin >> H >> W;
    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W; j++) cin >> A[i][j];
    }

    // 行の総和を計算する
    for (int i = 1; i <= H; i++) {
```

```

        gyou[i] = 0;
        for (int j = 1; j <= W; j++) gyou[i] += A[i][j];
    }

    // 列の総和を計算する
    for (int j = 1; j <= W; j++) {
        retu[j] = 0;
        for (int i = 1; i <= H; i++) retu[j] += A[i][j];
    }

    // 各マスに対する答えを計算する
    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W; j++) {
            Answer[i][j] = gyou[i] + retu[j] - A[i][j];
        }
    }

    // 空白区切りで出力
    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W; j++) {
            if (j >= 2) cout << " ";
            cout << Answer[i][j];
        }
        cout << endl;
    }
    return 0;
}

```

※ Python などのソースコードは chap5-4.md をご覧ください。

問題 5.4.3 (1)

一般に、 N 以下の整数のうち M の倍数であるものの個数は $\lfloor N/M \rfloor$ 個であるため、

- 3 の倍数は $A_1 = \lfloor 1000 \div 3 \rfloor = \text{333 個}$
- 5 の倍数は $A_2 = \lfloor 1000 \div 5 \rfloor = \text{200 個}$
- 7 の倍数は $A_3 = \lfloor 1000 \div 7 \rfloor = \text{142 個}$
- 15 の倍数は $A_4 = \lfloor 1000 \div 15 \rfloor = \text{66 個}$
- 21 の倍数は $A_5 = \lfloor 1000 \div 21 \rfloor = \text{47 個}$
- 35 の倍数は $A_6 = \lfloor 1000 \div 35 \rfloor = \text{28 個}$
- 105 の倍数は $A_7 = \lfloor 1000 \div 105 \rfloor = \text{9 個}$

となります。

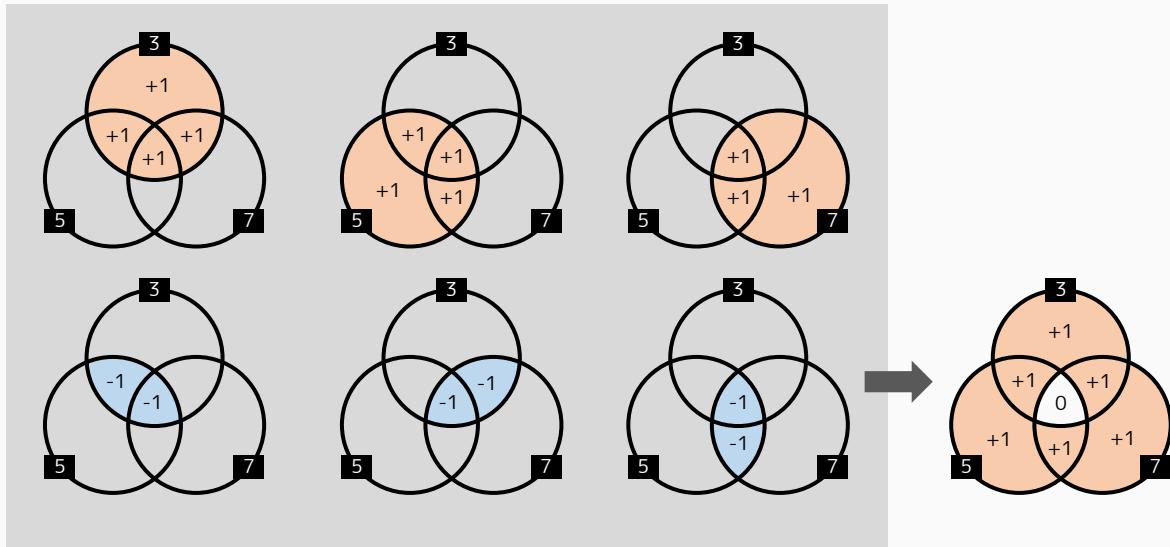
問題 5.4.3 (2), (3), (4), (5)

(2) $A_1 + A_2 + A_3$ と数えた場合、たとえば **15 という数**が 2 回数えられています。

(A_1 と A_2 両方にカウントされています)

(3) $A_1 + A_2 + A_3 - A_4 - A_5 - A_6$ と数えた場合、**105 という数**は A_1, A_2, A_3 で 3 回分足されていますが、 A_4, A_5, A_6 で 3 回分引かれているため、全く数えられていないのと同然です。

なお、下図に示すように、正しく数えられていない数は 105 の倍数のみです。



(4) (3) でカウントされなかった 105 の倍数を足せば良いです。答えは **$A_1 + A_2 + A_3 - A_4 - A_5 - A_6 + A_7$** となります。

(5) (1) の答えより、 $333 + 200 + 142 - 66 - 47 - 28 + 9 = \textcolor{red}{543} \text{ 個}$ です。

問題 5.4.4

集合 $S_1, S_2, S_3, \dots, S_N$ の和集合（どれか一つでも含まれる部分）の要素数は、以下の式で表されます。

N 個の集合の中から 1 つ以上を選ぶ方法は $2^N - 1$ 通りあるが、それらすべてに対して以下の値を加算したもの。

- 奇数個選んだとき：選んだ集合の共通部分の要素数
- 偶数個選んだとき：選んだ集合の共通部分の要素数 $\times (-1)$

たとえば、集合 S_1, S_2, S_3 の和集合の要素数は、以下をすべて足した値です。

- S_1 の要素数
- S_2 の要素数
- S_3 の要素数
- S_1 と S_2 の共通部分 $\times (-1)$
- S_1 と S_3 の共通部分 $\times (-1)$
- S_2 と S_3 の共通部分 $\times (-1)$
- S_1 と S_2 と S_3 の共通部分 $\times (-1)$

S_1 に「1000 以下の 3 の倍数」、 S_2 に「1000 以下の 5 の倍数」、 S_3 に「1000 以下の 7 の倍数」を当てはめてみてください。問題 5.4.3 と同じ結果になると思います。

問題 5.4.5

集合 S_1 を「 N 以下の V_1 の倍数」、集合 S_2 を「 N 以下の V_2 の倍数」、…、集合 S_K を「 N 以下の V_K の倍数」とするとき、求めるべき答えは S_1, S_2, \dots, S_K の共通部分となります。

したがって、 N 個の集合の選び方（どの倍数を選ぶか）を $2^N - 1$ 通り全探索する以下のようなプログラムを書くと、正解が得られます。ビット全探索（→コラム1）という実装方法を使っています。

なお、 P_1, P_2, \dots, P_M すべての倍数である N 以下の整数の個数は、

$$\frac{N}{P_1, P_2, P_3, \dots, P_M \text{ の最小公倍数}}$$

という式で表されます。（3 個以上の最小公倍数の求め方は →節末問題3.2.3）

```
#include <iostream>
using namespace std;

long long N, K;
long long V[20];
long long Answer = 0;

// 最大公約数を返す関数
long long GCD(long long A, long long B) {
    if (B == 0) return A;
    return GCD(B, A % B);
}
```

```

// 最小公倍数を返す関数
long long LCM(long long A, long long B) {
    return (A / GCD(A, B)) * B;
}

int main() {
    // 入力
    cin >> N >> K;
    for (int i = 1; i <= K; i++) cin >> V[i];

    // ビット全探索
    for (int i = 1; i < (1 << K); i++) {
        long long cnt = 0; // 選んだ数の個数
        long long lcm = 1; // 最小公倍数
        for (int j = 0; j < K; j++) {
            if (((i & (1 << j))) != 0) {
                cnt += 1;
                lcm = LCM(lcm, V[j + 1]);
            }
        }
        long long num = N / lcm; // 選ばれた数すべての倍数であるものの個数
        if (cnt % 2 == 1) Answer += num;
        if (cnt % 2 == 0) Answer -= num;
    }

    // 出力
    cout << Answer << endl;
    return 0;
}

```

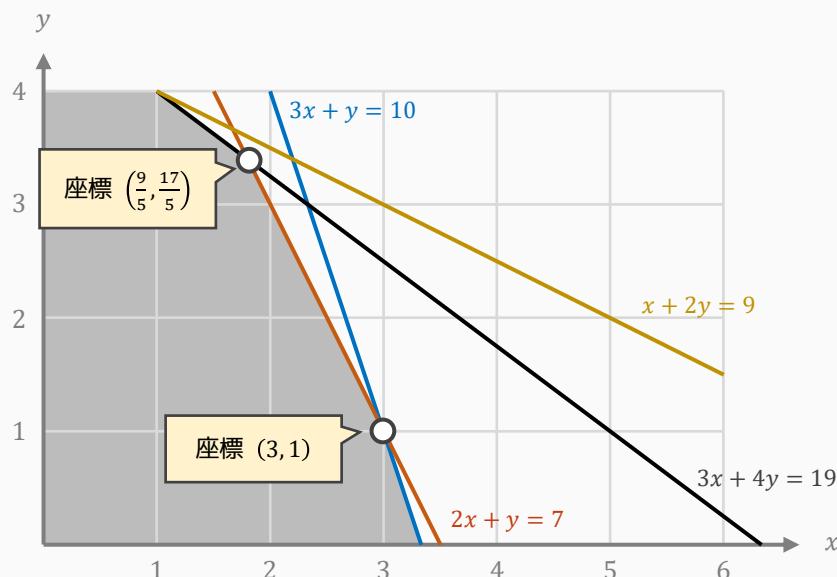
※ Python などのソースコードは chap5-4.md をご覧ください。

5.5

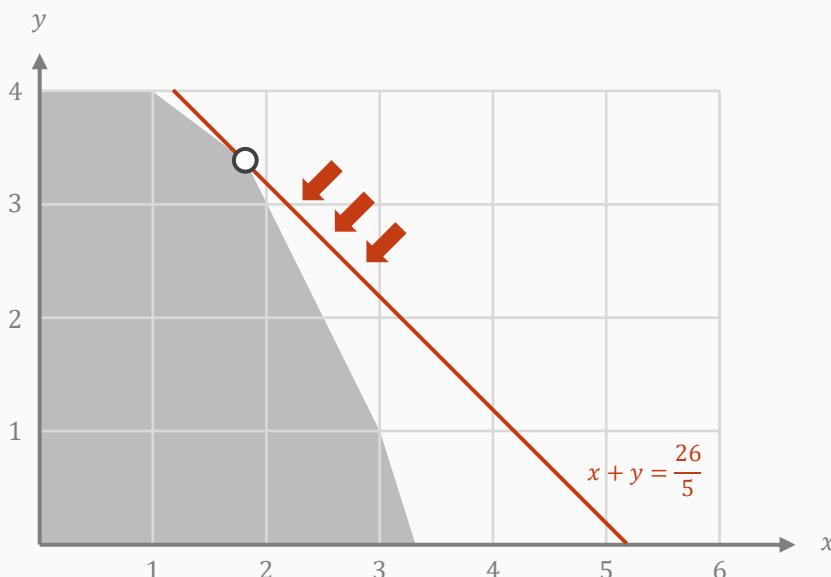
節末問題 5.5 の解答

問題 5.5.1

まず、 $3x + y = 10$, $2x + y = 7$, $3x + 4y = 19$, $x + 2y = 9$ のグラフを同じ座標平面上に描くと以下のようにになります。灰色は問題文の 4 つの条件をすべて満たす領域を示しています。

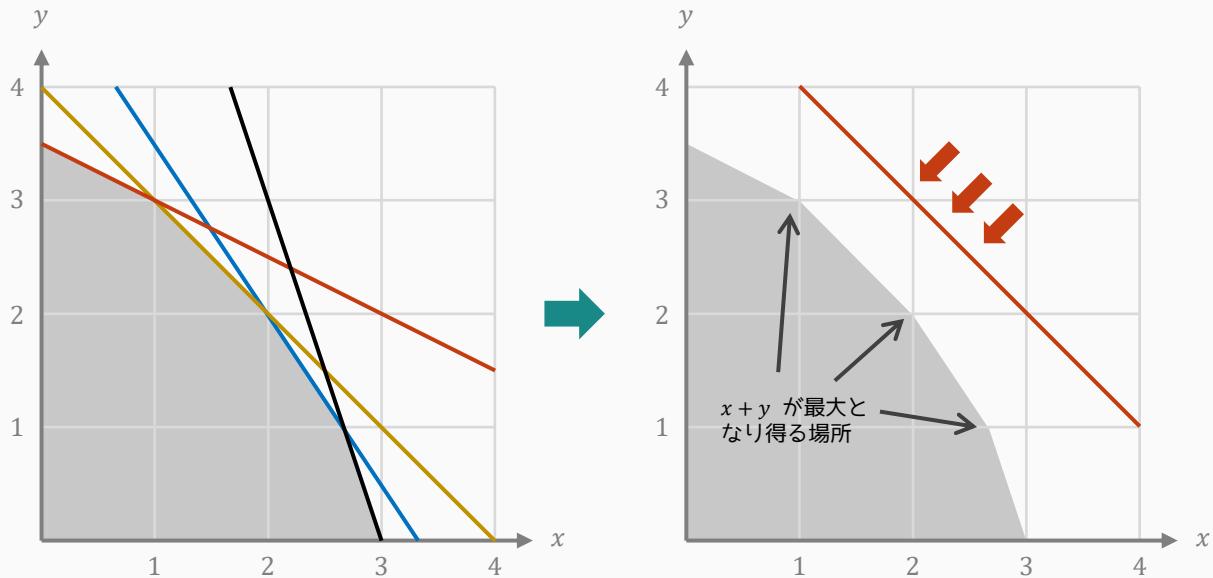


そこで、直線 $x + y = a$ を少しずつ下ろしてみましょう（ a を小さくしてみましょう）。そうすると、 $a = 26/5$ のときに初めて灰色部分とぶつかるので、 $x + y$ の最大値は $26/5$ です。



問題 5.5.2

線形計画問題では、2つの直線の交点で $x + y$ が最大となります。なぜなら、条件を満たす部分（下図の灰色に相当）は、2直線の交点でしか曲がらないからです。



したがって、以下のようなアルゴリズムによって答えを求めることができます。

なお、 x, y が正の実数のときに最大値をとることが制約で保証されていることに注意してください。（そうでないケースでは最大値が無限大になる可能性があり、場合分けが必要です）

すべての整数の組 (i, j) $[1 \leq i < j \leq N]$ について、以下のことを行う。

- 直線 $a_i x + b_i y + c_i = 0$ と直線 $a_j x + b_j y + c_j = 0$ の交点を求める
- N 個の条件式すべてを満たすかどうかを判定する

条件を満たす交点の中で $(x$ 座標) + $(y$ 座標) の値が最大となるものが答え。

なお、直線 $a_i x + b_i y + c_i = 0$ と直線 $a_j x + b_j y + c_j = 0$ の交点は以下の方法で求められます。

- $a_i b_j = a_j b_i$ のとき：2直線は平行（交差しない）
- $a_i b_j \neq a_j b_i$ のとき：交点は以下の座標となる（本書では扱っていない連立方程式を使うと使うと導出できます。興味のある人は調べましょう）

$$\left(\frac{c_i b_j - c_j b_i}{a_i b_j - a_j b_i}, \quad \frac{c_i a_j - c_j a_i}{b_i a_j - b_j a_i} \right)$$

したがって、以下のようなプログラムを書くと正解が得られます。関数 `check(x, y)` は実数の組 (x, y) が N 個の条件式をすべて満たす場合 `true`、そうでない場合 `false` を返します。

なお、このプログラムは `check` 関数を $O(N^2)$ 回呼び出しており、`check` 関数の計算量は $O(N)$ であるため、プログラム全体の計算量は $O(N^3)$ となります。

```
#include <iostream>
using namespace std;

int N;
double A[509], B[509], C[509];

bool check(double x, double y) {
    for (int i = 1; i <= N; i++) {
        if (A[i] * x + B[i] * y > C[i]) return false;
    }
    return true;
}

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i] >> B[i] >> C[i];

    // 交点を全探索
    double Answer = 0.0;
    for (int i = 1; i <= N; i++) {
        for (int j = i + 1; j <= N; j++) {
            // 交点をもたない場合
            if (A[i] * B[j] == A[j] * B[i]) continue;

            // i 番目の直線（条件式の境界）と j 番目の直線（条件式の境界）の交点を求める
            double px = (C[i] * B[j] - C[j] * B[i]) / (A[i] * B[j] - A[j] * B[i]);
            double py = (C[i] * A[j] - C[j] * A[i]) / (B[i] * A[j] - B[j] * A[i]);
            bool ret = check(px - 0.00000001, py - 0.00000001);
            if (ret == true) {
                Answer = max(Answer, px + py);
            }
        }
    }

    // 答えを出力
    printf("%.12lf\n", Answer);
    return 0;
}
```

※ Python などのソースコードは chap5-5.md をご覧ください。

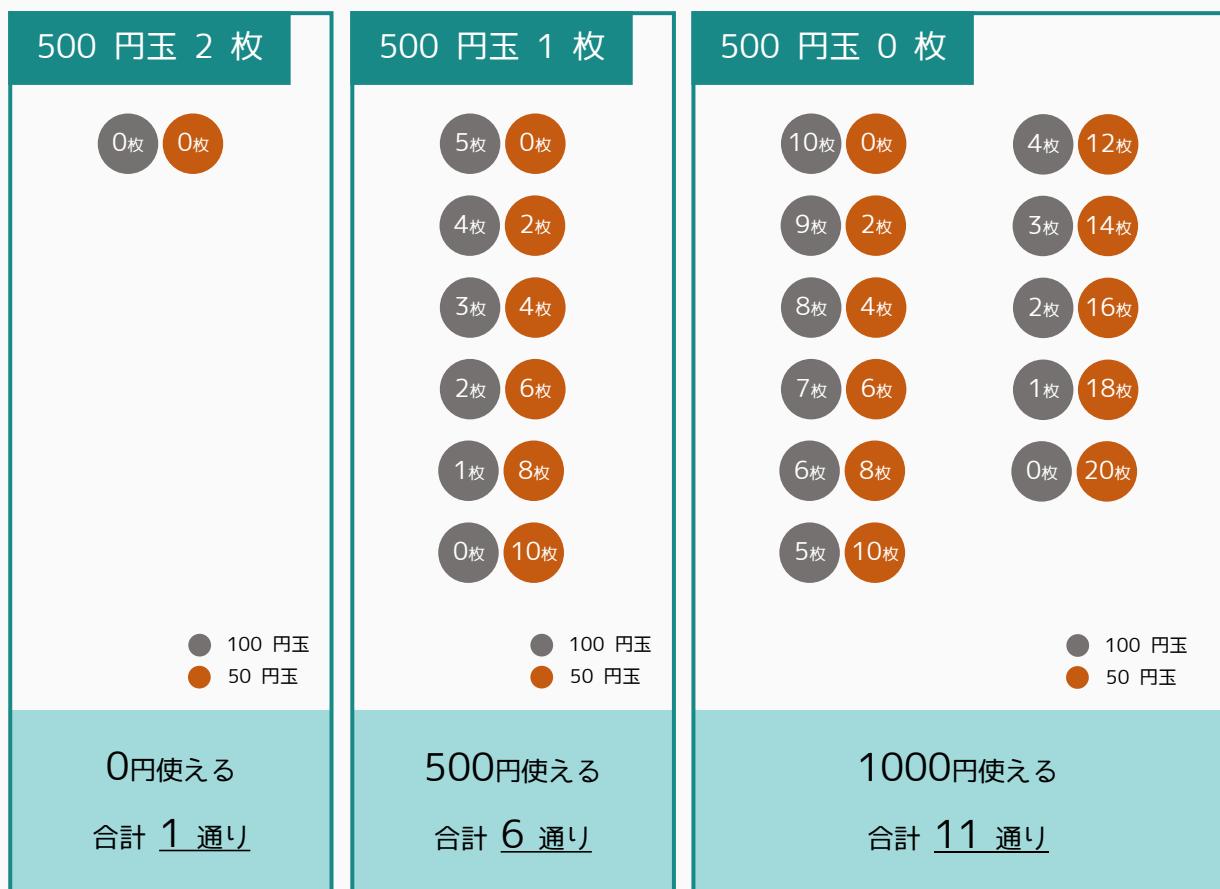
5.6

節末問題 5.6 の解答

問題 5.6.1 (1), (2), (3)

500 円玉を 2 枚・1 枚・0 枚使ったとき、100 円玉と 50 円玉合わせた金額はそれぞれ 0 円・500 円・1000 円となります。

したがって、それぞれの支払う方法の数は下図に示すとおりです。



問題 5.6.1 (4)

問題を右図のように「500 円玉の枚数」で分解することを考えます。

そうすると、(1), (2), (3) の結果より、答えが $1 + 6 + 11 = \textcolor{red}{18} \text{ 通り}$ であることが分かります。

元の問題

500,100,
50円玉で
1000円を支払
う方法の数

小問題1
500円玉が2枚

小問題2
500円玉が1枚

小問題3
500円玉が0枚

問題 5.6.2

まず、問題を以下のように分解することを考えましょう。

- ・ 小問題 1：選んだ整数の最大値が A_1 となる選び方は何通り？
- ・ 小問題 2：選んだ整数の最大値が A_2 となる選び方は何通り？
- ・ 小問題 3：選んだ整数の最大値が A_3 となる選び方は何通り？
- ・ … :
- ・ 小問題 N ：選んだ整数の最大値が A_N となる選び方は何通り？

このとき、求めるべき答えは以下のようになります。

$$(\text{小問題 1 の答え}) \times A_1 + \cdots + (\text{小問題 } N \text{ の答え}) \times A_N$$

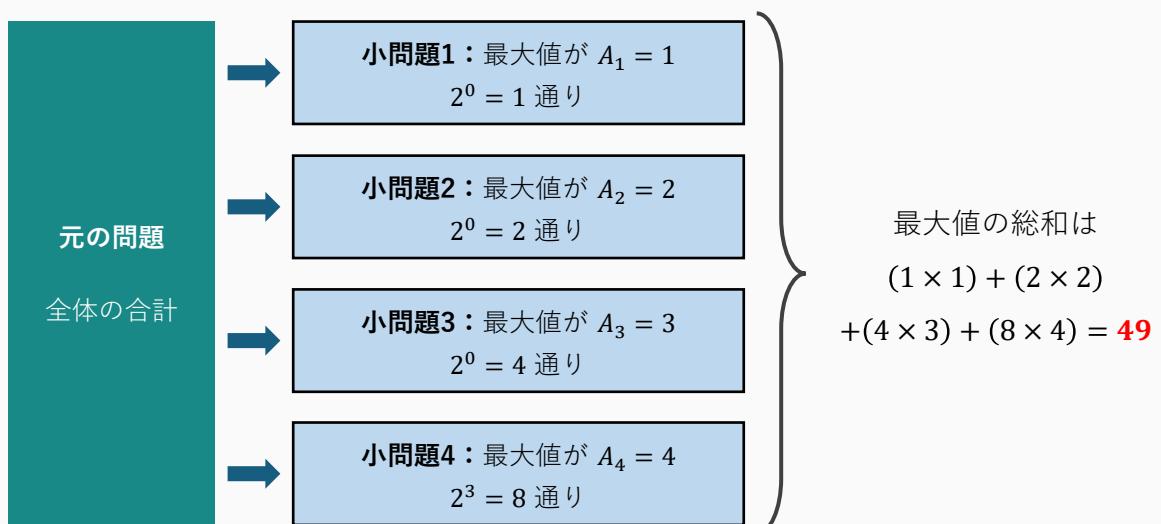
そこで、選んだ整数の最大値が A_i となるような選び方の条件は以下のとおりです。

- ・ A_i を選ぶ。
- ・ $A_1, A_2, A_3, \dots, A_{i-1}$ の中から 0 個以上を選ぶ。

$i - 1$ 個のものについて Yes/No を選択できるので、積の法則（→3.3.2項）より選び方は全部で 2^{i-1} 通りあります。したがって、求める答えは以下の通りです。

$$\sum_{i=1}^N 2^{i-1} \times A_i = (2^0 \times A_1) + (2^1 \times A_2) + \cdots + (2^{N-1} \times A_N)$$

たとえば、 $N = 4, (A_1, A_2, A_3, A_4) = (1, 2, 3, 4)$ の場合、下図のようにして答えが 49 だと分かります。



これをプログラムで実装すると、以下のようになります。なお、変数 `power[i]` は 2^i を 1000000007 で割った余りとなっています。

```
#include <iostream>
using namespace std;

const long long mod = 1000000007;
long long N;
long long A[300009];
long long power[300009];

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    //  $2^i$  を求める
    power[0] = 1;
    for (int i = 1; i <= N; i++) {
        power[i] = (2 * power[i - 1]) % mod;
    }

    // 答えを求める
    long long Answer = 0;
    for (int i = 1; i <= N; i++) {
        Answer += power[i - 1] * A[i];
        Answer %= mod;
    }

    // 出力
    cout << Answer << endl;
    return 0;
}
```

※ Python などのソースコードは chap5-6.md をご覧ください。

5.7

節末問題 5.7 の解答

問題 5.7.1

足し算の式の中に 2021 は 4 個、1234 は 5 個あります。

したがって、求める答えは $2021 \times 4 + 1234 \times 5 = 8084 + 6170 = \textcolor{red}{14254}$ です。

問題 5.7.2

美しさの期待値を以下の ${}_6C_2 = 15$ 個のパートに分解することを考えます。

- ・ パーツ 1：1 番目と 2 番目のサイコロの出方により加算される美しさ
- ・ パーツ 2：1 番目と 3 番目のサイコロの出方により加算される美しさ
- ・ パーツ 3：1 番目と 4 番目のサイコロの出方により加算される美しさ
- ・ パーツ 4：1 番目と 5 番目のサイコロの出方により加算される美しさ
- ・ … :
- ・ パーツ 15：5 番目と 6 番目のサイコロの出方により加算される美しさ

そこで、以下の理由により、各パートにおける「加算される美しさ」の期待値は $1/6$ となります。

サイコロの出目は右図の $6 \times 6 = 36$ 通り
が等確率で起こり得ます。

一方、そのうち 2 つの出目が同じとなる
ものは 6 通りであるため、その確率はは
 $6/36 = 1/6$ です。分からぬ人は 3.4 節
に戻って確認しましょう。

		サイコロ 2					
		1	2	3	4	5	6
サイコロ 1	1	○	✗	✗	✗	✗	✗
	2	✗	○	✗	✗	✗	✗
	3	✗	✗	○	✗	✗	✗
	4	✗	✗	✗	○	✗	✗
	5	✗	✗	✗	✗	○	✗
	6	✗	✗	✗	✗	✗	○

したがって、求める答えは $1/6 \times 15 = \textcolor{red}{5/2}$ となります。（期待値の線形性 [→3.4 節] より、全体の美しさの期待値は、各パートの期待値の和となります）

問題 5.7.3

まず、 $A_1 \leq A_2 \leq A_3 \leq \dots \leq A_N$ のケースを考えましょう。このとき、以下の式が成り立つため、答えは例題 2 (\rightarrow 5.7.3項) と同一となります。

$$|A_j - A_i| = A_j - A_i \quad (1 \leq i \leq j \leq N)$$

よって

$$\sum_{i=1}^N \sum_{j=i+1}^N |A_j - A_i| = \sum_{i=1}^N \sum_{j=i+1}^N A_j - A_i$$

一方、求めるべき答えは「異なる 2 つの要素の差を全部足した値」であるため、 $A_1, A_2, A_3, \dots, A_N$ の順序を入れ替えても答えは変わりません。

たとえば、 $A = (1, 4, 2, 3)$ の場合の答えは 10 であり、それを並べ替えた $A = (1, 2, 3, 4)$ の場合の答えも 10 です。

したがって、数列 $A = (A_1, A_2, \dots, A_N)$ を昇順にソート (\rightarrow 3.6節) した後、例題 2 と同じ処理を行う以下のようなプログラムを作成すると、正解が得られます。

```
#include <iostream>
#include <algorithm>
using namespace std;

long long N, A[200009];
long long Answer = 0;

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // ソート (コード 5.7.1 から追加した唯一の部分)
    sort(A + 1, A + N + 1);

    // 答えを求める → 答えの出力
    for (int i = 1; i <= N; i++) Answer += A[i] * (-N + 2LL * i - 1LL);
    cout << Answer << endl;
    return 0;
}
```

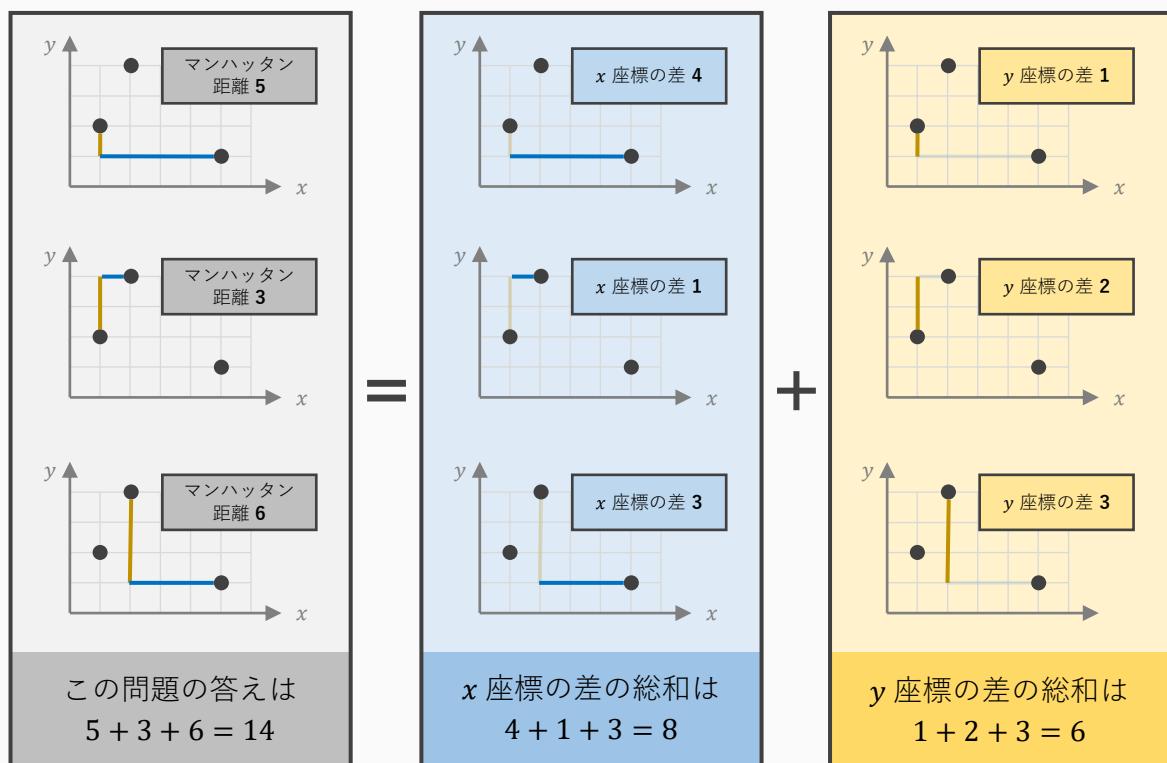
※ Python などのソースコードは chap5-7.md をご覧ください。

問題 5.7.4

まず、2点間のマンハッタン距離は、 x 座標の差の絶対値と y 座標の差の絶対値を足した値です。したがって、求めるべき「マンハッタン距離の総和」は、以下の2つのパートの答えを足した値となります。

- パーツ1： x 座標の差の絶対値の総和
- パーツ2： y 座標の差の絶対値の総和

たとえば、座標 $(1, 2), (5, 1), (2, 4)$ に点がある場合を考えましょう。マンハッタン距離の総和は $5 + 3 + 6 = 14$ である一方、 x 座標の差の絶対値の総和は 8、 y 座標の差の絶対値の総和は 6 です。



そこで、パート1・パート2の答えは、以下のような式で表されます。これは節末問題 5.7.3 と同じ形であり、 (x_1, x_2, \dots, x_N) と (y_1, y_2, \dots, y_N) を小さい順にソートすると、あとは計算量 $O(N)$ で式の値が求められます。

$$Part1 = \sum_{i=1}^N \sum_{j=i+1}^N |x_i - x_j|$$

$$Part2 = \sum_{i=1}^N \sum_{j=i+1}^N |y_i - y_j|$$

よって、以下のようなプログラムを書くと、正解が得られます。なお、C++ では標準ライブラリ `std::sort` を使うことで、配列の要素を小さい順にソートすることができます。 (→3.6.1項)

```
#include <iostream>
#include <algorithm>
using namespace std;

long long N;
long long X[200009], Y[200009];

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> X[i] >> Y[i];

    // 配列をソートする
    sort(X + 1, X + N + 1);
    sort(Y + 1, Y + N + 1);

    // パーツ 1 の答え (x 座標の差の絶対値の総和)
    long long Part1 = 0;
    for (int i = 1; i <= N; i++) Part1 += X[i] * (-N + 2LL * i - 1LL);

    // パーツ 2 の答え (y 座標の差の絶対値の総和)
    long long Part2 = 0;
    for (int i = 1; i <= N; i++) Part2 += Y[i] * (-N + 2LL * i - 1LL);

    // 出力
    cout << Part1 + Part2 << endl;
    return 0;
}
```

※ Python などのソースコードは chap5-7.md をご覧ください。

5.8

節末問題 5.8 の解答

問題 5.8.1

5.8.3 項で述べた通り、年齢差の関係を表したグラフの頂点 1 から頂点 i までの最短経路長を $dist[i]$ とすると、人 i の年齢の最大値は $\min(dist[i], 120)$ となります。

したがって、最短経路長を求めるプログラム（コード 4.5.3）の出力部分のみ変更した、以下のようなプログラムを提出すると、正解が得られます。

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

int N, M, A[100009], B[100009];
int dist[100009];
vector<int> G[100009];

int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= M; i++) {
        cin >> A[i] >> B[i];
        G[A[i]].push_back(B[i]);
        G[B[i]].push_back(A[i]);
    }

    // 幅優先探索の初期化 (dist[i]=-1 のとき、未到達の白色頂点である)
    for (int i = 1; i <= N; i++) dist[i] = -1;
    queue<int> Q; // キュー Q を定義する
    Q.push(1); dist[1] = 0; // Q に 1 を追加 (操作 1)

    // 幅優先探索
    while (!Q.empty()) {
        int pos = Q.front(); // Q の先頭を調べる (操作 2)
        Q.pop(); // Q の先頭を取り出す (操作 3)
        for (int i = 0; i < (int)G[pos].size(); i++) {
            int nex = G[pos][i];
            if (dist[nex] == -1) {
                dist[nex] = dist[pos] + 1;
                Q.push(nex); // Q に nex を追加 (操作 1)
            }
        }
    }
}
```

```

    }
}

// 頂点 1 から各頂点までの最短距離を出力
// 頂点 1 から到達できない場合は 120 歳にできる
for (int i = 1; i <= N; i++) {
    if (dist[i] == -1) cout << "120" << endl;
    else cout << min(dist[i], 120) << endl;
}
return 0;
}

```

← コード 4.3.1 からの変更部分

※ Python などのソースコードは chap5-8.md をご覧ください。

問題 5.8.2

この問題は解法が見えづらいので、 N が小さいケースを調べてみましょう。

- $N = 2$ のとき、 $(P_1, P_2) = (2, 1)$ で最大スコア 1
- $N = 3$ のとき、 $(P_1, P_2, P_3) = (2, 3, 1)$ で最大スコア 3
- $N = 4$ のとき、 $(P_1, P_2, P_3, P_4) = (2, 3, 4, 1)$ で最大スコア 6

となります。（全探索をすることで、手計算でも分かります）

$N \geq 5$ の場合も同じように、 $(P_1, P_2, \dots, P_{N-1}, P_N) = (2, 3, \dots, N, 1)$ とすると、スコアは

$$\sum_{i=1}^N (i \bmod P_i) = 1 + 2 + 3 + \dots + (N-1) + 0 = \frac{N(N-1)}{2}$$

となります（和の公式 → **2.5.10項**）。しかし、これが本当に最大なのでしょうか。

答えは Yes であり、以下のようにして証明できます。

まずは簡単のため、 $N = 4$ の最大値が 6 であることを証明します。スコアは

$$\sum_{i=1}^4 (i \bmod P_i) = (1 \bmod P_1) + (2 \bmod P_2) + (3 \bmod P_3) + (4 \bmod P_4)$$

ですが、mod の性質より以下のことがいえます。

- $1 \bmod P_1$ は $P_1 - 1$ 以下
- $2 \bmod P_2$ は $P_2 - 1$ 以下
- $3 \bmod P_3$ は $P_3 - 1$ 以下
- $4 \bmod P_4$ は $P_4 - 1$ 以下

したがって、スコアはそれらを足した値 $P_1 + P_2 + P_3 + P_4 - 4$ 以下です。

しかし、 (P_1, P_2, P_3, P_4) は $(1, 2, 3, 4)$ の並べ替えであるため、

$$P_1 + P_2 + P_3 + P_4 = 1 + 2 + 3 + 4 = 10$$

$$P_1 + P_2 + P_3 + P_4 - 4 = 1 + 2 + 3 + 4 - 4 = 6$$

となり、スコアが **6 以下** であることが分かります。よって、スコアは $(P_1, P_2, P_3, P_4) = (2, 3, 4, 1)$ のとき最大値 **6** となります。

一般の場合の証明

$N = 4$ の場合と同じような方法で証明できます。まず、 $i \bmod P_i \leq P_i - 1$ が成り立つため、スコアは $(P_1 - 1) + (P_2 - 1) + \cdots + (P_N - 1) = P_1 + \cdots + P_N - N$ 以下であるといえます。

一方、 $(P_1, P_2, P_3, \dots, P_N)$ は $(1, 2, 3, \dots, N)$ の並べ替えであるため、

$$\begin{aligned} P_1 + P_2 + \cdots + P_N &= 1 + 2 + \cdots + N = \frac{N(N+1)}{2} \\ P_1 + P_2 + \cdots + P_N - N &= \frac{N(N+1)}{2} - N = \frac{N(N-1)}{2} \end{aligned}$$

となり、スコアが **$N(N-1)/2$ 以下** であることが分かります。

したがって、 $N(N-1)/2$ を出力する以下のようないくつかのプログラムを提出すると、正解が得られます。

本問題の制約は $N \leq 10^9$ と大きく、答えが 10^{17} を超える可能性があるため、`int` 型などの 32 ビット整数を使うとオーバーフローを起こすことに注意してください。

```
#include <iostream>
using namespace std;

int main() {
    // 入力
    long long N;
    cin >> N;

    // 出力
    cout << N * (N - 1) / 2 << endl;
    return 0;
}
```

※ Python などのソースコードは chap5-8.md をご覧ください。

問題 5.8.3

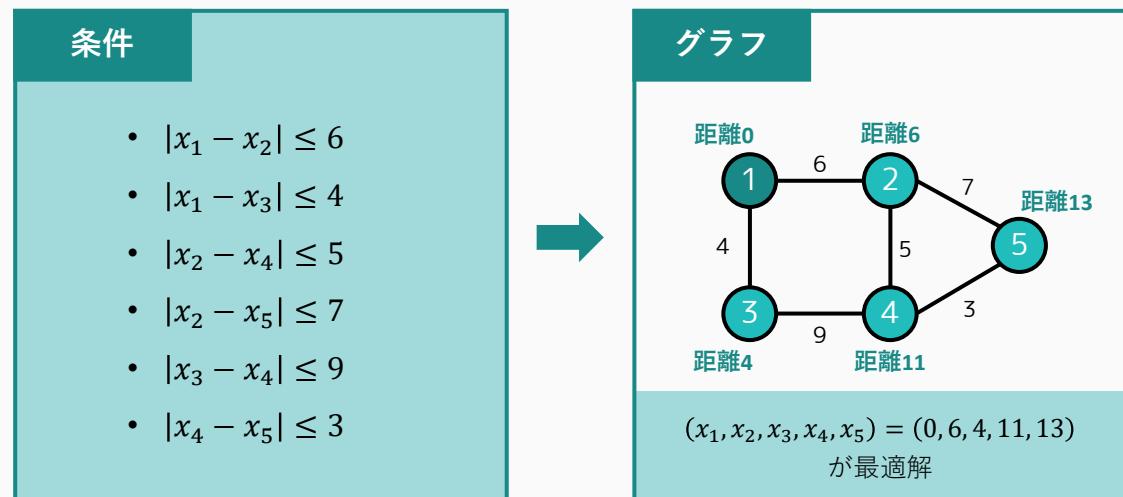
注意：この問題は、4.5.8 項「その他の代表的なグラフアルゴリズム」で紹介されたダイクストラ法を使います。初学者は解けなくて当然ですので、ご安心ください。

まず、以下のような重み付き無向グラフを考えましょう。

- 頂点は N 個あり、1 から N までの番号が振られている。
- 辺は M 個あり、 $|x_{A_i} - x_{B_i}| \leq C_i$ という条件式について、頂点 A_i と B_i を結ぶ重み C_i の辺を追加する。

このとき、 x_N の最大値は頂点 1 から頂点 N までの最短経路長 $dist[N]$ となります。

具体例は以下の通りであり、 $x_i = dist[i]$ とすると確かに M 個の条件式すべてを満たします。（詳しい証明は本解説では扱わないことにしますが、重みなしグラフの場合と同じような方法で証明できます）



したがって、ダイクストラ法によって頂点 1 から N までの最短経路長 $dist[N]$ を求め、それを出力する以下のようなプログラムを提出すると、正解が得られます。

```
#include <bits/stdc++.h>
using namespace std;

long long N, M;
long long A[500009], B[500009], C[500009];

// グラフ
bool used[500009];
long long dist[500009]; // dist[i] は頂点 1 → 頂点 i の最短経路長
vector<pair<int, long long>> G[500009];
priority_queue<pair<long long, int>, vector<pair<long long, int>>, greater<pair<long long, int>>> Q;
```

```

// ダイクストラ法
void dijkstra() {
    // 配列の初期化など
    for (int i = 1; i <= N; i++) dist[i] = (1LL << 60);
    for (int i = 1; i <= M; i++) used[i] = false;
    dist[1] = 0;
    Q.push(make_pair(0, 1));

    // キューの更新
    while (!Q.empty()) {
        int pos = Q.top().second; Q.pop();
        if (used[pos] == true) continue;
        used[pos] = true;
        for (pair<int, int> i : G[pos]) {
            int to = i.first, cost = dist[pos] + i.second;
            if (pos == 0) cost = i.second; // 頂点 0 の場合は例外
            if (dist[to] > cost) {
                dist[to] = cost;
                Q.push(make_pair(dist[to], to));
            }
        }
    }
}

int main() {
    // 入力・グラフの辺の追加
    cin >> N >> M;
    for (int i = 1; i <= M; i++) {
        cin >> A[i] >> B[i] >> C[i];
        G[A[i]].push_back(make_pair(B[i], C[i]));
        G[B[i]].push_back(make_pair(A[i], C[i]));
    }

    // ダイクストラ法
    dijkstra();

    // 答えの出力
    if (dist[N] == (1LL << 60)) cout << "-1" << endl;
    else cout << dist[N] << endl;
    return 0;
}

```

※ Python などのソースコードは chap5-8.md をご覧ください。

5.9

節末問題 5.9 の解答

問題 5.9.1

コード 5.9.1 では、以下のように while 文を用いて「ギリギリまで払ったときの紙幣の枚数」を数えていました。

```
while (N >= 10000) { N -= 10000; Answer += 1; }
while (N >= 5000) { N -= 5000; Answer += 1; }
while (N >= 1) { N -= 1000; Answer += 1; }
```

これは割り算を用いて計算することもできます。残り金額が N 円のとき、使える紙幣の枚数の最大値は以下の表のようになります。

札の種類	10000 円札	5000 円札	1000 円札
使える最大枚数	$\left\lfloor \frac{N}{10000} \right\rfloor$ 枚	$\left\lfloor \frac{N}{5000} \right\rfloor$ 枚	$\left\lfloor \frac{N}{1000} \right\rfloor$ 枚
ギリギリまで使ったときの残り金額	$N \bmod 10000$ 円	$N \bmod 5000$ 円	$N \bmod 1000$ 円

したがって、以下のようなプログラムを書くと、計算量 $O(1)$ で答えを求めることができます。 $N = 10^{18}$ 程度の入力をしても、一瞬で実行が終わります。

```
#include <iostream>
using namespace std;

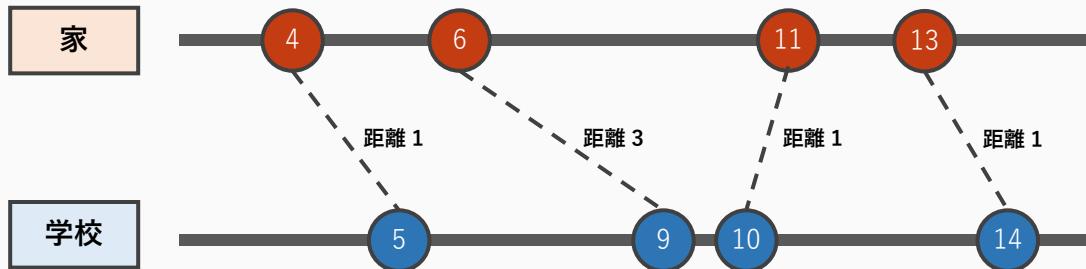
int main() {
    // 入力
    long long N, Answer = 0;
    cin >> N;

    // 10000 円札を支払う
    Answer += (N / 10000); N %= 10000;
    // 5000 円札を支払う
    Answer += (N / 5000); N %= 5000;
    // 1000 円札を支払う
    Answer += (N / 1000); N %= 1000;

    // 答えを出力
    cout << Answer << endl;
    return 0;
}
```

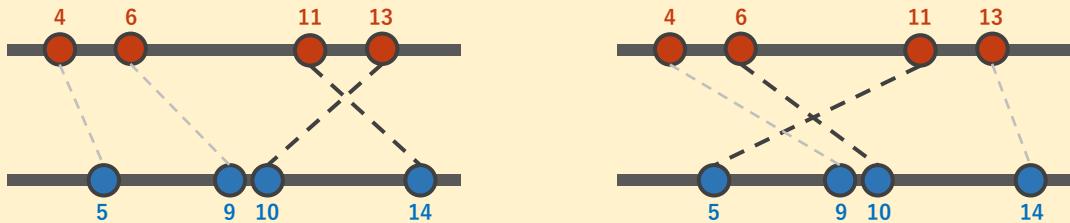
問題 5.9.2

下図のように、「左から 1 番目の家と左から 1 番目の小学校」「左から 2 番目の家と左から 2 番目の小学校」…「左から N 番目の家と左から N 番目の小学校」を繋ぐと、家と通う学校の距離の合計が最小となります。

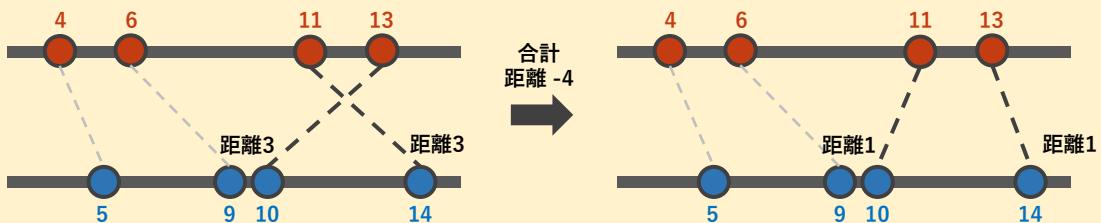


この事実は、以下のようにして証明することができます。（少し難易度が高いので読み飛ばしても構いません）

まず、上で述べた方法（以下、方法 A と記す）を除くすべての繋ぎ方には「交差する箇所」が少なくとも 1 つ存在します。



また、交差する箇所を繋ぎ換えて交差を消すと、合計距離は減るか変わらないかのいずれかになります。下図がその一例です。



そして、交差を消す操作ができなくなるまで繰り返すと、最終的には必ず方法 A になります※。このことから、方法 A より距離の合計が短い繋ぎ方（家と学校の割り当て方）が存在しないことが証明できました。

※交差する点線の組の数（最大 ${}_N C_2$ 組）が必ず 1 以上減ることから証明できます。

したがって、配列 (A_1, A_2, \dots, A_N) を小さい順にソートしたものを $(A'_1, A'_2, \dots, A'_N)$ 、配列 (B_1, B_2, \dots, B_N) を小さい順にソートしたものを $(B'_1, B'_2, \dots, B'_N)$ とするとき、距離の合計は以下の式で表されます。

$$\sum_{i=1}^N |A'_i - B'_i| = \underbrace{|A'_1 - B'_1|}_{\substack{\text{左から } 1 \text{ 番目の家と} \\ \text{左から } 1 \text{ 番目の学校の距離}} + \underbrace{|A'_2 - B'_2|}_{\substack{\text{左から } N \text{ 番目の家と} \\ \text{左から } N \text{ 番目の学校の距離}}} + \cdots + \underbrace{|A'_N - B'_N|}_{\substack{\text{左から } N \text{ 番目の家と} \\ \text{左から } N \text{ 番目の学校の距離}}}$$

よって、以下のようなプログラムを提出すると、正解が得られます。

```
#include <iostream>
#include <cmath>
#include <algorithm>
using namespace std;

long long N;
long long A[100009], B[100009];

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];
    for (int i = 1; i <= N; i++) cin >> B[i];

    // ソート
    sort(A + 1, A + N + 1);
    sort(B + 1, B + N + 1);

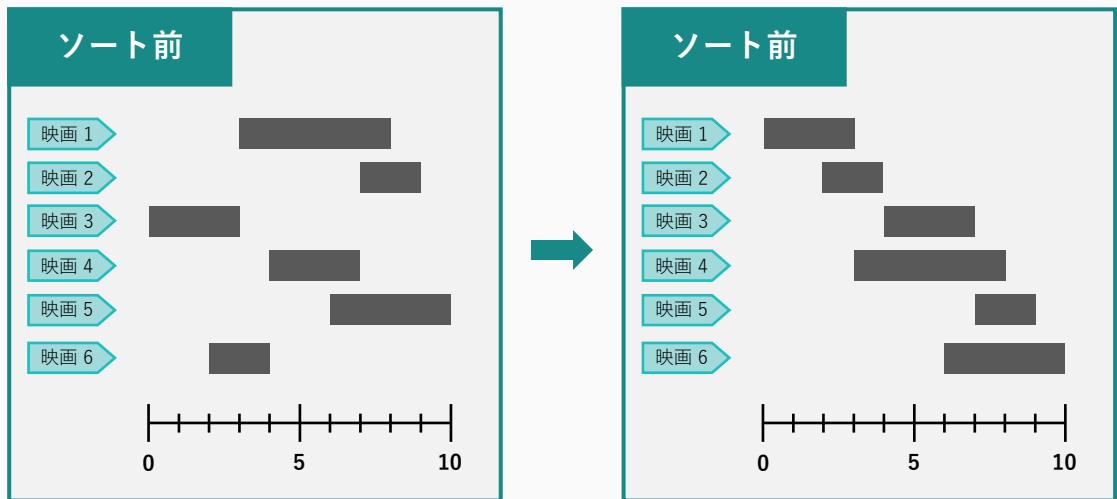
    // 答えを求める
    long long Answer = 0;
    for (int i = 1; i <= N; i++) Answer += abs(A[i] - B[i]);
    cout << Answer << endl;
    return 0;
}
```

※ Python などのソースコードは chap5-9.md をご覧ください。

問題 5.9.3

コード 5.9.2 のアルゴリズムは確かに正しい答えを出すことができますが、計算が遅いです。「いま選べる中で最も終了時刻が早い映画」を調べるのに計算量 $O(N)$ かかるため、 N 個すべての映画を選べるようなケースでは、計算量が $O(N^2)$ となってしまいます。一体どうすれば高速化できるのでしょうか。

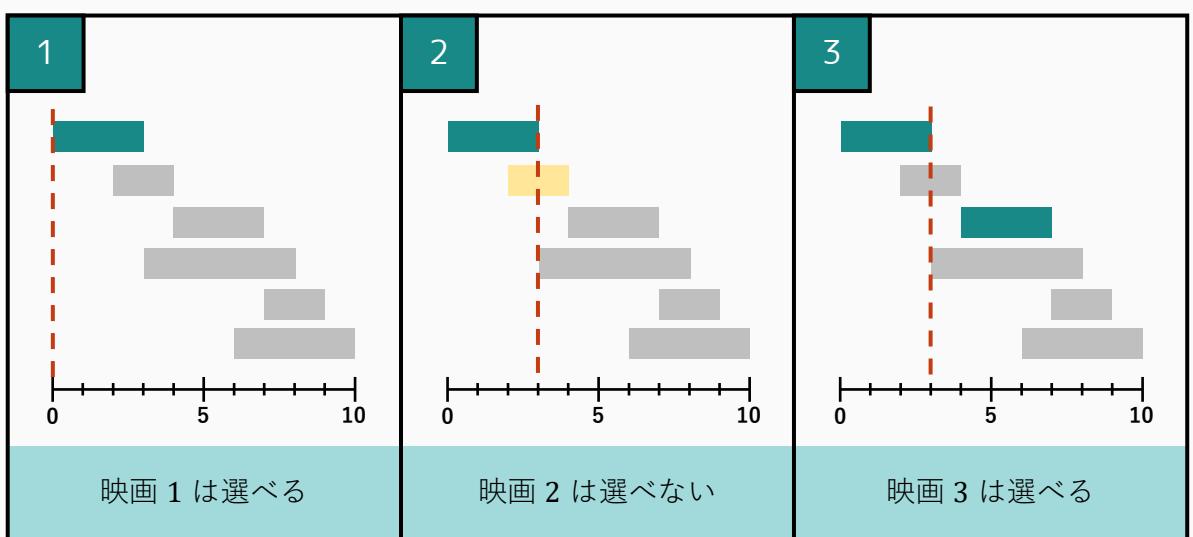
そこで、映画を終了時刻の早い順にソートし、最も終了が早いものを「映画 1」、最も終了が遅いものを「映画 N 」としましょう。

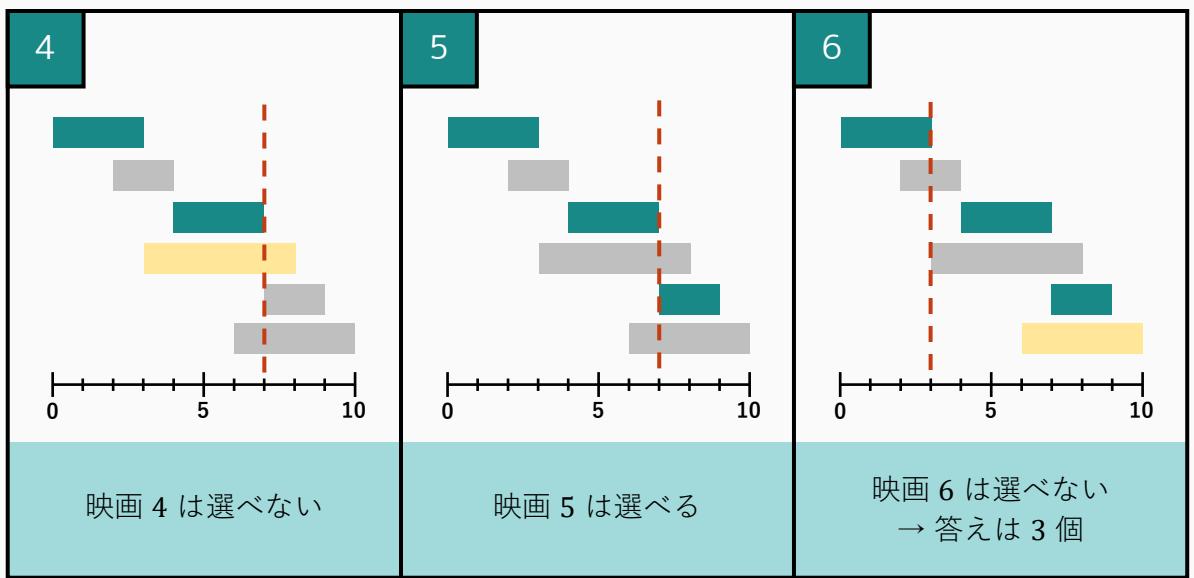


そうすると、以下のようなアルゴリズムで「最も終了が早いもの」を効率的に選び続けることができます。

- ・ 映画 1 を選ぶ。
- ・ (開始時刻的に) 映画 2 を選べるならば、選ぶ。
- ・ (開始時刻的に) 映画 3 を選べるならば、選ぶ。
- ・ :
- ・ (開始時刻的に) 映画 N を選べるならば、選ぶ。

このアルゴリズムを具体的な例に適用すると、以下のようになります。





たとえば C++ での実装例は以下のようになります。なお、終了時刻の早い順にソートするために、Movie という型を使っています。Movie 型の変数 A があったとき、

- A.l：映画の開始時刻
- A.r：映画の終了時刻

を意味し、A.r の大きい方が「大きい」と判定されます (bool operator< の部分)。そのため、sort 関数によって A.r の小さい順に整列されます。

```
#include <iostream>
#include <algorithm>
using namespace std;

// Movie 型
struct Movie {
    int l, r;
};

// Movie 型の比較関数
bool operator<(const Movie &a1, const Movie &a2) {
    if (a1.r < a2.r) return true;
    if (a1.r > a2.r) return false;
    if (a1.l < a2.l) return true;
    return false;
}

int N;
Movie A[300009];
int CurrentTime = 0; // 現在時刻 (最後に選んだ映画の終了時刻)
int Answer = 0; // 現在見た映画の数

int main() {
    // 入力
    cin >> N;
```

```
for (int i = 1; i <= N; i++) cin >> A[i].l >> A[i].r;

// ソート
sort(A + 1, A + N + 1);

// 終了時刻が最も早いものを選び続ける
for (int i = 1; i <= N; i++) {
    if (CurrentTime <= A[i].l) {
        CurrentTime = A[i].r;
        Answer += 1;
    }
}

// 出力
cout << Answer << endl;
return 0;
}
```

※ Python などのソースコードは chap5-9.md をご覧ください。

5.10

節末問題 5.10 の解答

問題 5.10.1

分配法則（→5.10.2項）を使うと、以下のように楽な計算で解くことができます。

問題 (1)

$$\begin{aligned} & 37 \times 39 + 37 \times 61 \\ &= 37 \times (39 + 61) \\ &= 37 \times 100 \\ &= \mathbf{3700} \end{aligned}$$

問題 (2)

$$\begin{aligned} & 2021 \times 333 + 2021 \times 333 + 2021 \times 334 \\ &= 2021 \times (333 + 333 + 334) \\ &= 2021 \times 1000 \\ &= \mathbf{2021000} \end{aligned}$$

問題 5.10.2

この問題は、例題 2（→5.10.2項）の一般化です。

分配法則を使うと、シグマ記号の式について以下のことが分かります。

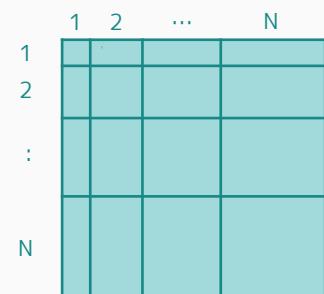
- $i = 1$ のときの総和 : $(1 \times 1) + (1 \times 2) + \cdots + (1 \times N) = \mathbf{1 \times (1 + 2 + \cdots + N)}$
- $i = 2$ のときの総和 : $(2 \times 1) + (2 \times 2) + \cdots + (2 \times N) = \mathbf{2 \times (1 + 2 + \cdots + N)}$
- \vdots
- $i = N$ のときの総和 : $(N \times 1) + (N \times 2) + \cdots + (N \times N) = \mathbf{N \times (1 + 2 + \cdots + N)}$

求めるべき答えは青色で示した値の総和であるため、分配法則より

$$\sum_{i=1}^N \sum_{j=1}^N ij = (1 + 2 + \cdots + N) \times (1 + 2 + \cdots + N) = \frac{N(N+1)}{2} \times \frac{N(N+1)}{2}$$

となります。イメージが湧かない人は、右図の正方形の面積を考えてみると良いと思います。

縦の長さが $N(N+1)/2$ 、横の長さが $N(N+1)/2$ となっています。



したがって、以下のように答えを出力するプログラムを提出すると、正解が得られます。なお、この問題は制約が $N \leq 10^9$ と大きく、 $N(N+1)/2 \times N(N+1)/2$ の値が 10^{30} を超える可能性があります。計算途中で余りを取る（→4.6.1項）などの工夫を行わなければ、`long long` 型などの 64 ビット整数でもオーバーフローを起こす可能性があるので注意してください。

```
#include <iostream>
using namespace std;

const long long mod = 1000000007;
long long N;

int main() {
    // 入力
    cin >> N;

    // 答えを求める
    long long val = N * (N + 1) / 2;
    val %= mod;
    cout << val * val % mod << endl;
    return 0;
}
```

※ Python などのソースコードは chap5-10.md をご覧ください。

問題 5.10.3

以下のような立方体を考えると、この問題の答えが

$$\sum_{i=1}^A \sum_{j=1}^B \sum_{k=1}^C ijk = (1 + \dots + A)(1 + \dots + B)(1 + \dots + C)$$

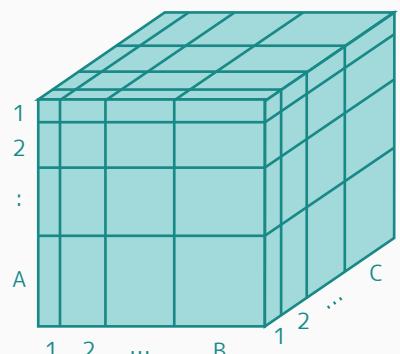
であることが分かります。そこで和の公式（→2.5.10 項）より、次式が成り立ちます。

$$1 + 2 + \dots + A = \frac{A(A+1)}{2}$$

$$1 + 2 + \dots + B = \frac{B(B+1)}{2}$$

$$1 + 2 + \dots + C = \frac{C(C+1)}{2}$$

よって、答えは $\frac{A(A+1)}{2} \times \frac{B(B+1)}{2} \times \frac{C(C+1)}{2}$ です。



したがって、以下のように答えを出力するプログラムを提出すると、正解となります。

なお、本問題は制約が $A, B, C \leq 10^9$ と大きいので、オーバーフローを防ぐために

- $D = A(A + 1)/2$
- $E = B(B + 1)/2$
- $F = C(C + 1)/2$

と変数をおいたうえで、計算途中で余りをとるなどの工夫をしています。

```
#include <iostream>
using namespace std;

const long long mod = 998244353;
long long A, B, C;

int main() {
    // 入力
    cin >> A >> B >> C;

    // 計算
    long long D = A * (A + 1) / 2; D %= mod;
    long long E = B * (B + 1) / 2; E %= mod;
    long long F = C * (C + 1) / 2; F %= mod;

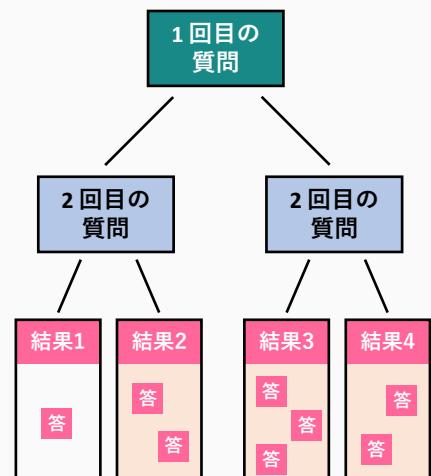
    // 答えを出力
    // ここで (D * E * F) % mod にしても、途中で 10^27 を扱う可能性がある
    // そのため、long long 型でもオーバーフローすることに注意！
    cout << (D * E % mod) * F % mod << endl;
    return 0;
}
```

※ Python などのソースコードは chap5-10.md をご覧ください。

問題 5.10.4

まず、太郎君の思い浮かべている数として 8 通りが考えられますが、2 回の質問に対する回答の組合せは「Yes→Yes」「Yes→No」「No→Yes」「No→No」の 4 通りしかありません。

$8 > 4$ ですから、右図のように「結果が 1 通りに定まっているもの」が必ず存在します。そのため、2 回で確実に当てることは不可能です。



問題 5.10.5

自然に実装すると、以下のようになります。

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    // 入力
    double a, b, c;
    cin >> a >> b >> c;

    // 左辺と右辺の計算
    double v1 = log2(a);
    double v2 = b * log2(c);

    // 出力
    if (v1 < v2) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

しかし、このプログラムを提出すると 100 ケース中 15 ケースで不正解となってしまいます。その理由は誤差（→5.10.1項）です。

たとえば $(a, b, c) = (10^{18} - 1, 18, 10)$ のケースでは、本当の答えは Yes なのに間違つて No と出力しています。実際、

$$\log_2 a = 59.7947057079725222602 \dots$$

$$b \log_2 c = 59.7947057079725222616 \dots$$

であり、左辺と右辺の相対誤差は 10^{-19} 程度です。あまりにも近すぎるので、コンピュータの限界を超えてしまい、「同じ数である」と判定してしまうのです。

改善方法①

それでは、誤差による不正解を防ぐにはどうすれば良いのでしょうか。一つの方法は全部整数で扱うことです。対数の性質（→2.3.10項）より、

$b \log_2 c = \log_2(c^b)$ であるから
 $\log_2 a < b \log_2 c$ のとき、 $\log_2 a < \log_2(c^b)$
よって、 $a < c^b$

↓
log の中身だけを取っても
大小関係は変わらない

であるため、 $a < c^b$ であれば Yes、そうでなければ No と出力すれば良いです。

これを実装すると、以下のようになります。

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    // 入力
    long long a, b, c;
    cin >> a >> b >> c;

    // 右辺の計算 (c の b 乗)
    long long v = 1;
    for (long long i = 1; i <= b; i++) {
        v *= c;
    }

    // 出力
    if (a < v) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

しかし、これは不正解となってしまいます。その理由はオーバーフロー（→[5.10.1項](#)）です。このプログラムは c^b の値をそのまま計算しますが、制約が $a, b, c \leq 10^{18}$ と大きく、最悪の場合 10^{18} の 10^{18} 乗を計算することになります。C++ はもちろん、Python でも計算することができません。

改善方法②

次に、オーバーフローを防ぐにはどうすれば良いのでしょうか。典型的な方法として「計算の途中で余りを取る」などが考えられますが、本問題は余りの計算ではないため、この方法は通用しません。

そこで、累乗を計算している途中で右辺の値が a を上回ったらこの時点で Yes 確定なので、ループ処理を打ち切るという対策が有効です。自然に実装すると、以下のようにになります。

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    // 入力
```

```

#include <iostream>
#include <cmath>
using namespace std;

int main() {
    // 入力
    long long a, b, c;
    cin >> a >> b >> c;

    // 右辺の計算 (c の b 乗)
    long long v = 1;
    for (long long i = 1; i <= b; i++) {
        if (a / c < v) {
            // この条件分岐は a < (v * c) を言い換えただけ
            // 条件の言い換えをした理由は、v, c が 10^{18} 程度になる可能性があるため
            // a < v * c になると最悪の場合 v * c = 10^{36} になりオーバーフローするから
            // 注：long long 型の限界は 2^{63}-1 (約 10^{19})
            cout << "Yes" << endl;
            return 0;
        }
        v *= c;
    }

    // ループが打ち切られない場合
    cout << "No" << endl;
    return 0;
}

```

しかし、このプログラムは 100 ケース中 2 ケースで実行時間制限超過 (TLE) となります。その原因是 $c = 1$ のケースです。

たとえば、 $(a, b, c) = (2, 10^{18}, 1)$ のケースを考えましょう。1 は何乗しても 1 なので、「現在の右辺の値 v が a を超えたら打ち切る」という処理は効きません。そのため、 $b = 10^{18}$ 回のループを行ってしまいます。

なお、 $2^{60} > 10^{18}$ であるため、 $c \geq 2$ のケースでは必ず 60 回以内のループで処理が終わるといえます。

改善方法③

最後に $c = 1$ のケースで場合分けをしましょう。この問題の制約では $a \geq 1$ なので、 $c^b = 1$ より、答えは必ず No となります。したがって、次ページのようなプログラムを書くと、ようやく正解が得られます。

```

#include <iostream>
#include <cmath>
using namespace std;

int main() {
    // 入力
    long long a, b, c;
    cin >> a >> b >> c;

    // c = 1 のときの場合分け
    if (c == 1) {
        cout << "No" << endl;
        return 0;
    }

    // 右辺の計算 (c の b 乗)
    long long v = 1;
    for (long long i = 1; i <= b; i++) {
        if (a / c < v) {
            // この条件分岐は a < (v * c) を言い換えただけ
            // 条件の言い換えをした理由は、v, c が 10^18 程度になる可能性があるため
            // a < v * c にすると最悪の場合 v * c = 10^36 になりオーバーフローするから
            cout << "Yes" << endl;
            return 0;
        }
        v *= c;
    }

    // ループが打ち切られない場合
    cout << "No" << endl;
    return 0;
}

```

※ Python などのソースコードは chap5-10.md をご覧ください。

問題 5.10.6

まず、 $m = 1, 2, \dots, N$ についてそれぞれ調べる方法が考えられますが、制約が $N \leq 10^{11}$ と大きいため、実行時間制限超過 (TLE) となってしまいます。そこで、 m としてあり得るパターンの数より $f(m)$ としてあり得るパターンの数の方が圧倒的に小さいため、以下のアルゴリズムが効率的です。

- $f(m)$ としてあり得る候補を全列挙する。
- $f(m)$ が決まれば $m = f(m) + B$ と決まるので、それぞれの候補について m の各桁の積が $f(m)$ と一致するかどうかをチェックする。

それでは $f(m)$ の候補はどうやって全列挙すれば良いのでしょうか。実は、1123 や 12233599 のような単調増加な数 m について $f(m)$ を計算するだけで良いです。なぜなら、数の順番を並べ替えても一般性を失わないからです。たとえば、

- $m = 1123$ のとき $f(m) = 1 \times 1 \times 2 \times 3 = 6$
- $m = 2131$ のとき $f(m) = 2 \times 1 \times 3 \times 1 = 6$
- $m = 3112$ のとき $f(m) = 3 \times 1 \times 1 \times 2 = 6$

とすべて同じになります。なお、単調増加な 11 衔以内の数はおよそ 30 万個しか存在せず、全列挙は十分現実的です。

したがって、以下のようなプログラムを書くと、正解が得られます。なお、単調増加な数 m は再帰関数 `func(digit, m)` で全列挙しており、`digit` は現在の桁数を表したもので、`func` が分からぬ人は、3.6 節に戻って確認しましょう。

また、関数 `product(m)` は整数 m の各桁の積を返すものです。数を 10 で割り続けていくことで、各桁の値を計算しています。2 進数に変換するアルゴリズム（→2.1.9項）と似ています。

※注意：この C++ プログラムは、本書では扱っていない `set` 型を利用しています。知らない方は、インターネットなどで調べてみてください。（GitHub に掲載されている Python・JAVA のソースコードでも `set` 型が用いられています）

```
#include <iostream>
#include <set>
using namespace std;

// f(m) としてあり得る候補
// set 型についてはインターネットで調べてみてください！
set<long long> fm_cand;

// m の各桁の積を返す関数
long long product(long long m) {
    if (m == 0) {
        return 0;
    }
    else {
        long long ans = 1;
        while (m >= 1) {
            ans *= (m % 10);
            m /= 10;
        }
        return ans;
    }
}
```

```

    }
}

void func(int digit, long long m) {
    // m の桁数は 11 桁以下
    // 注：余った桁を 1 で埋めれば、全部 11 桁と仮定しても良い
    int min_value = (m % 10);
    for (int i = min_value; i <= 9; i++) {
        // 10 * m + i は m の後ろに数字 i を付けたもの
        func(digit + 1, 10 * m + i);
    }
}

int main() {
    // f(m) の候補を列挙
    func(0, 0);

    // 入力
    long long N, B;
    cin >> N >> B;

    // m - f(m) == B になるかどうかチェック
    long long Answer = 0;
    for (long long fm : fm_cand) {
        long long m = fm + B;
        long long prod_m = product(m);
        if (m - prod_m == B && m <= N) {
            Answer += 1;
        }
    }

    // 出力
    cout << Answer << endl;
    return 0;
}

```

※ Python などのソースコードは chap5-10.md をご覧ください。

問題 5.10.7 (1)

この問題は様々な解法が考えられるので、そのうち一つを紹介します。

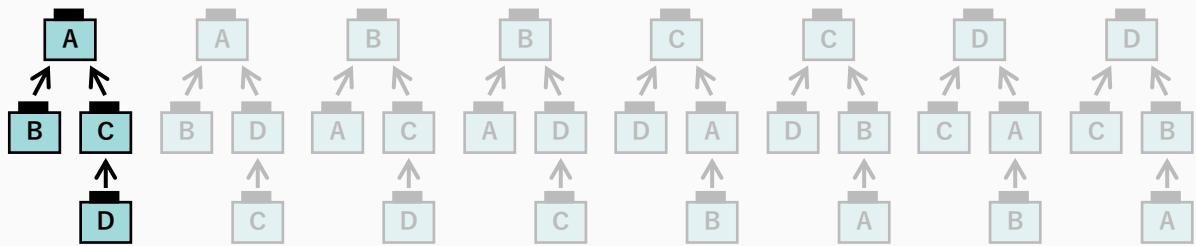
なお、ここでは説明の都合上、それぞれのおもりに A, B, C, D, E というラベルを付けることにします。



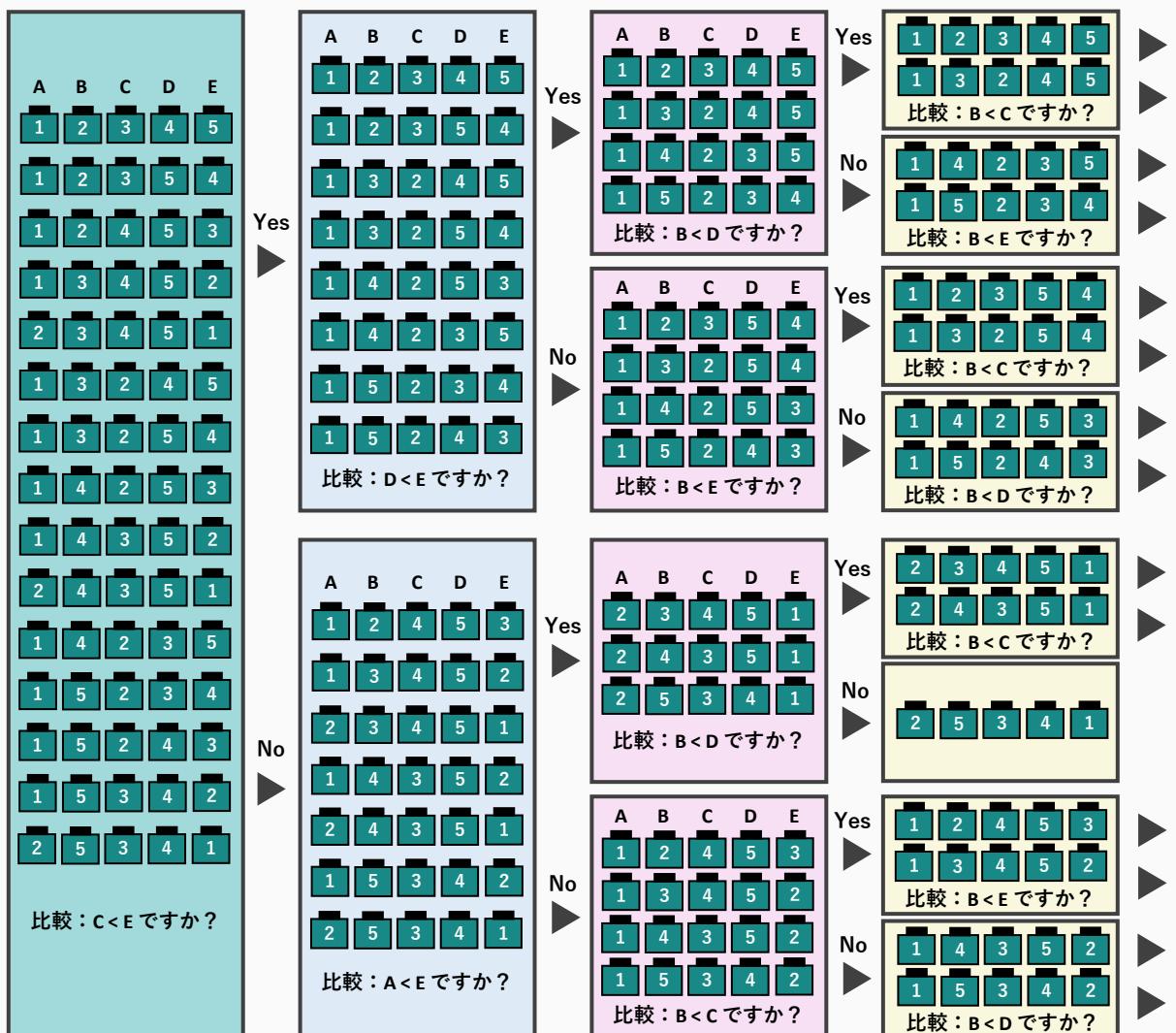
まず、最初の3回は以下のような比較を行います。

1. おもり A とおもり B を比較する。
2. おもり C とおもり D を比較する。
3. 1. の軽い方と 2. で軽い方を比較する。

3回の質問の結果は以下の8通りがあり得ますが、対称性より「おもり A が最も軽く、おもり C よりおもり D の方が重い」という一番左のパターンであることを仮定しても、一般性を失いません。



さて、一番左のパターンとなるおもりの重さの組合せは15通りありますが、以下のような比較により、必ず4回で当てることが可能です。（数字は重さ [kg]）



問題 5.10.7 (2)

おもりの重さの組合せは $5! = 120$ 通り存在する一方、6回の比較の結果（左側・右側のどちらが重いか）の組合せは $2^6 = 64$ 通りです。後者の方が小さいため、6回で当てることはできません。

問題 5.10.7 (3)

まず、以下のようにして最小回数が 45 回以上であることが証明できます。

おもりの重さの組合せを P 通りとすると、 L 回で比較を行うためには $2^L \geq P$ すなわち $L \geq \log_2 P$ を満たす必要があります。

そこでおもりが 16 個のとき $\log_2 P = \log_2 16! = 44.2501 \dots$ となるため、少なくとも 45 回の比較が必要です。

それでは、何回が最小なのでしょうか。まず、マージソート（→3.6節）をこの問題に適用すると、以下のような操作を行うことになります。

- 「1 個のおもりの列 2 つを Merge する」 × 8 回
- 「2 個のおもりの列 2 つを Merge する」 × 4 回
- 「4 個のおもりの列 2 つを Merge する」 × 2 回
- 「8 個のおもりの列 2 つを Merge する」 × 1 回

l 個のおもりの列に対する Merge 操作では $2l - 1$ 回の比較を行うため（→3.6.10 項）、合計比較回数は $(1 \times 8) + (3 \times 4) + (7 \times 2) + (15 \times 1) = 49$ 回となります。

また、2021 年 12 月現在、46 回以内で確実に当てられる方法が考案されています。詳しく知りたい方は、以下の論文をお読みください。

- Peczarski, Marcin (2011). “Towards Optimal Sorting of 16 Elements”. *Acta Universitatis Sapientiae*. 4 (2): 215–224.

しかしながら、最小回数が 45 回であるか、はたまた 46 回であるのかは誰も知りません。興味を持たれた方は、この未解決問題にぜひ挑戦してみましょう。

第 6 章

最終確認問題

6.1

最終確認問題 1-5 の解答

問題 1

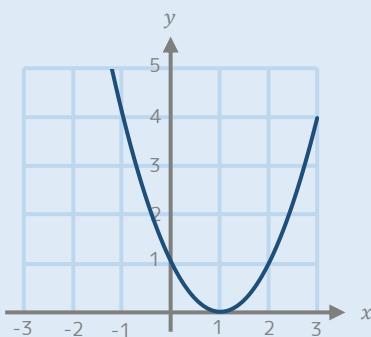
1. $a + 2b + 3c + 4d = (1 \times 12) + (2 \times 34) + (3 \times 56) + (4 \times 78)$
 $= 12 + 68 + 168 + 312$
 $= \textcolor{red}{560}$
2. $a^2 + b^2 + c^2 + d^2 = 12^2 + 34^2 + 56^2 + 78^2$
 $= 144 + 1156 + 3136 + 6084$
 $= \textcolor{red}{10520}$
3. $abcd \bmod 10 = (12 \times 34 \times 56 \times 78) \bmod 10$
 $= (2 \times 4 \times 6 \times 8) \bmod 10$
 $= 288 \bmod 10 = \textcolor{red}{8}$
4. $\sqrt{b+d-a} = \sqrt{34+78-12}$
 $= \sqrt{100} = \textcolor{red}{10}$

なお、3. では計算途中で余りを取っても正しい答えが得られる性質（→4.6.1項）を使っています。

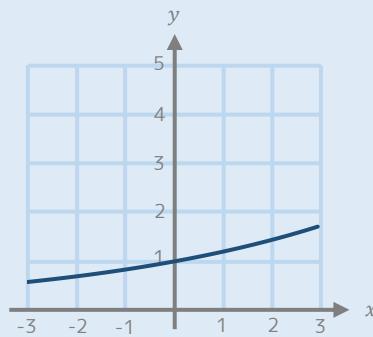
問題 2

答えは以下のようになります。分からぬ人は、関数（→2.3節）に戻って確認しましょう。なお、(4) の $y = 2^{3x}$ は $y = 8^x$ と同じです。

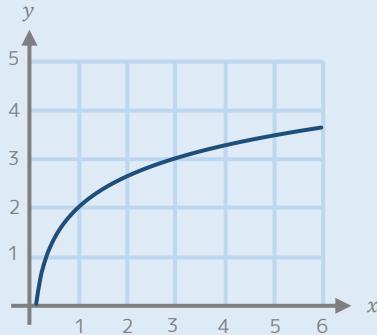
関数 $y = x^2 - 2x + 1$



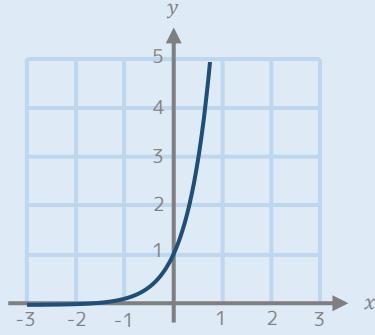
関数 $y = 1.2^x$



関数 $y = \log_3 x + 2$



関数 $y = 2^{3x}$



問題 3 (1), (2)

1. ${}_4P_3 = (4 \times 3 \times 2) = \mathbf{24}$

${}_{10}P_5 = (10 \times 9 \times 8 \times 7 \times 6) = \mathbf{30240}$

${}_{2021}P_1 = \mathbf{2021}$

2. ${}_4C_3 = {}_4P_3 \div 3! = 24 \div 6 = \mathbf{4}$

${}_{10}C_5 = {}_{10}P_5 \div 5! = 30240 \div 120 = \mathbf{252}$

${}_{2021}C_1 = {}_{2021}P_1 \div 1! = 2021 \div 1 = \mathbf{2021}$

${}_{2021}C_{2020} = \underline{{}_{2021}C_1} = \mathbf{2021}$

$${}_{2021}C_{2020} = \frac{2021!}{2020! \times 1!}$$

$${}_{2021}C_1 = \frac{2021!}{1! \times 2020!} \text{ であるため。}$$

問題 3 (3), (4), (5)

3. 積の法則（→3.3.2項）より、選び方の総数は $160 \times 250 \times 300 = \mathbf{12000000} \text{ 通り}$ （1200 万通り）です。

4. 積の法則（→3.3.2項）より、書き方の総数は $4^5 = \mathbf{1024} \text{ 通り}$ です。

5. N 個のものを並べる方法の数は $N! = 1 \times 2 \times \cdots \times N$ 通り（→3.3.3項）なので、長さ 8 の数列は全部で $8! = \mathbf{40320} \text{ 通り}$ です。

問題 4

まず、平均値（→3.5.4項）は以下のようになります。

$$\frac{182 + 182 + 188 + 191 + 192 + 195 + 197 + 200 + 205 + 217}{10} = \mathbf{195 \text{ cm}}$$

次に標準偏差を計算します。各部員の身長の平均との差は次表の通りです。

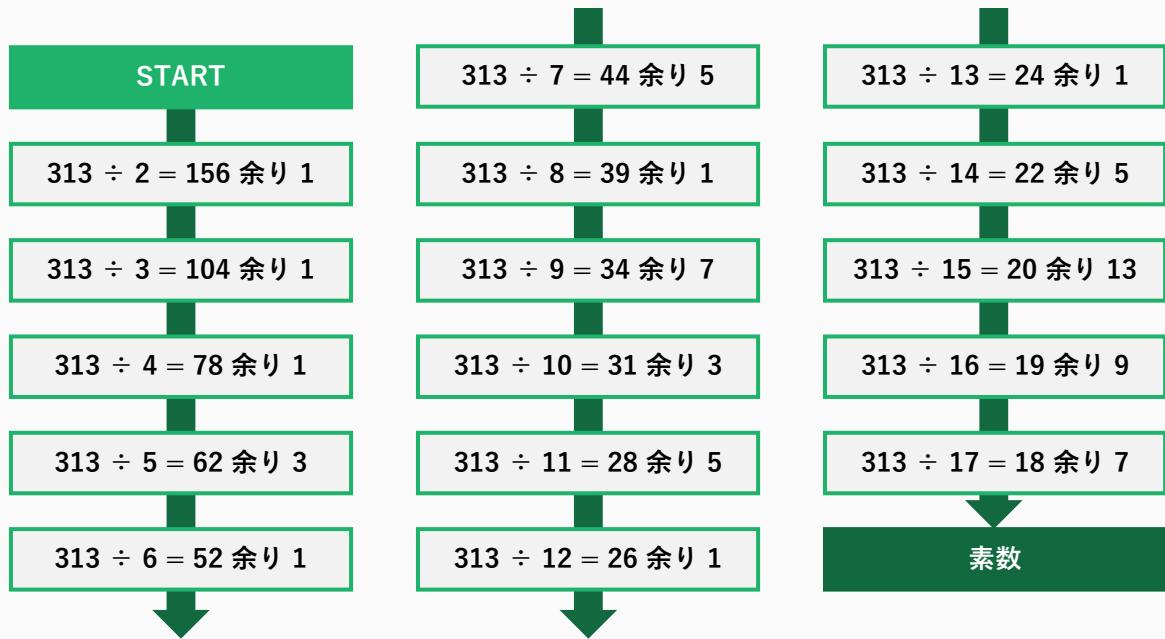
身長	182	183	188	191	192	195	197	200	205	217
平均との差	13	12	7	4	3	0	2	5	10	22

したがって、標準偏差（→3.5.4項）は次のようにになります。

$$\sqrt{\frac{13^2 + 12^2 + 7^2 + 4^2 + 3^2 + 0^2 + 2^2 + 5^2 + 10^2 + 22^2}{10}} = \mathbf{10 \text{ cm}}$$

問題 5 (1)

$\sqrt{313} = 17.69 \dots$ なので、以下のように 2 から 17 まで割って行けばよいです。いずれも割り切れないため、313 は素数です。



問題 5 (2)

723 と 207 の最大公約数をユークリッドの互除法で求めると、以下のようになります。求める最大公約数は 3 です。



6.2

最終確認問題 6-10 の解答

問題 6

(1) $1/x$ の積分 (\rightarrow 4.3.4項) より、

$$\int_1^{10000} \frac{1}{x} dx = \log_e 10000 = 9.2103 \dots$$

となります。これを小数第一位で四捨五入すると、答えは 9 です。

(2) それぞれの i におけるプログラムのループ回数は、次の通りです。

- $i = 1$ のとき : $\lfloor N/1 \rfloor + 1000$ 回
- $i = 2$ のとき : $\lfloor N/2 \rfloor + 1000$ 回
- $i = 3$ のとき : $\lfloor N/3 \rfloor + 1000$ 回
- \vdots
- $i = N$ のとき : $\lfloor N/N \rfloor + 1000$ 回

よって、全体のループ回数 L は次のようにになります。

$$\underbrace{\left\lfloor \frac{N}{1} \right\rfloor + \left\lfloor \frac{N}{2} \right\rfloor + \left\lfloor \frac{N}{3} \right\rfloor + \cdots + \left\lfloor \frac{N}{N} \right\rfloor}_{+ 1000N}$$

逆数の和の性質 (\rightarrow 4.4.4項) より、下線部分の総和はおよそ $N \log_e N$ です。したがって $L \approx N \log_e N + 1000N$ となります。これをランダウの O 記法を用いて表すと、計算量は $O(N \log N)$ です。

問題 7

答えは次の表のようになります。

関数	N^2	N^3	2^N	3^N	$N!$
1 億	10000	464	26	17	11
5 億	22361	793	28	18	12
10 億	31623	1000	29	18	12

問題 8

(1) 答えは以下の通りです。前から一つずつ計算すると良いです。 (→3.7.1項)

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
1	1	5	9	29	65	181	441	1165	2929

(2) 答えは以下の通りです。分からぬ人は、4.7節に戻って確認しましょう。

$$A + B = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 5 & 8 \\ 10 & 20 \end{bmatrix} = \begin{bmatrix} 1+5 & 4+8 \\ 1+10 & 0+20 \end{bmatrix} = \begin{bmatrix} 6 & 12 \\ 11 & 20 \end{bmatrix}$$

$$AB = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 5 & 8 \\ 10 & 20 \end{bmatrix} = \begin{bmatrix} 1 \times 5 + 4 \times 10 & 1 \times 8 + 4 \times 20 \\ 1 \times 5 + 0 \times 10 & 1 \times 8 + 0 \times 20 \end{bmatrix} = \begin{bmatrix} 45 & 88 \\ 5 & 8 \end{bmatrix}$$

(3) 答えは以下の通りです。分からぬ人は、4.7節に戻って確認しましょう。

$$A^2 = A \times A = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 1 & 4 \end{bmatrix}$$

$$A^3 = A^2 \times A = \begin{bmatrix} 5 & 4 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 9 & 20 \\ 5 & 4 \end{bmatrix}$$

$$A^4 = A^3 \times A = \begin{bmatrix} 9 & 20 \\ 5 & 4 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 29 & 36 \\ 9 & 20 \end{bmatrix}$$

$$A^5 = A^4 \times A = \begin{bmatrix} 29 & 36 \\ 9 & 20 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 65 & 181 \\ 29 & 36 \end{bmatrix}$$

(4) これは行列累乗でフィボナッチ数列が表せることと同じような理由です。まず、
 $a_n = a_{n-1} + 4a_{n-2}$ 、 $a_{n-1} = a_{n-1}$ より、次式が成り立ちます。

$$\begin{bmatrix} a_n \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_{n-1} \\ a_{n-2} \end{bmatrix}$$

この式を繰り返し適用させると、以下のように累乗を用いた式で表されます。

$$\begin{bmatrix} a_n \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_{n-1} \\ a_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} a_{n-2} \\ a_{n-3} \end{bmatrix}$$

= ...

$$= \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} a_2 \\ a_1 \end{bmatrix} = A^{n-2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

よって、 a_n の値は A^{n-2} の (1, 1) 成分と (1, 2) 成分を足した値と等しいです。

(次ページへ続く)

そこで、次の式が成り立つため、「 A^{n-2} の $(1, 1)$ 成分と $(1, 2)$ 成分を足した値」は「 A^{n-1} の $(1, 1)$ 成分」と一致します。

$$A^{n-2} \times A = \begin{bmatrix} (1, 1) \text{ 成分} & (1, 2) \text{ 成分} \\ (2, 1) \text{ 成分} & (2, 2) \text{ 成分} \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} (1, 1) \text{ 成分} + (1, 2) \text{ 成分} & (1, 1) \text{ 成分} \times 4 \\ (2, 1) \text{ 成分} + (2, 2) \text{ 成分} & (2, 1) \text{ 成分} \times 4 \end{bmatrix}$$

これが、 $(1, 1)$ 成分の値が数列に出現する値になっている理由です。

問題 9

$1 \text{ XOR } 2 \text{ XOR } 3 \text{ XOR } \cdots \text{ XOR } N$ の値を、いきなり $N = 1000000007$ のケースで求めるのは難しいので、まずは小さいケースで調べてみましょう（→5.2節）。

N	1	2	3	4	5	6	7	8	9	10	11
答え	1	3	0	4	1	7	0	8	1	11	0

$N = 3, 7, 11$ では $1 \text{ XOR } 2 \text{ XOR } \cdots \text{ XOR } N = 0$ であるため、この時点で「 N を 4 で割ると 3 余るのではないか」という規則性が頭に浮かぶと思います。

それでは、この規則性は N が大きくなっても成り立つのでしょうか。答えは Yes であり、以下のようにして証明できます。

まず、 x を 4 の倍数とするととき、以下の式が成り立ちます。

$$\begin{aligned} & x \text{ XOR } (x+1) \text{ XOR } (x+2) \text{ XOR } (x+3) \\ &= \cancel{\{x \text{ XOR } (x+1)\}} \text{ XOR } \cancel{\{(x+2) \text{ XOR } (x+3)\}} \\ &= 1 \text{ XOR } 1 = 0 \end{aligned}$$

したがって、 $N \bmod 4 = 3$ であるとき、求めるべき値は以下の通りです。

$$\begin{aligned} & \frac{1 \text{ XOR } 2 \text{ XOR } 3 \text{ XOR } \cancel{4 \text{ XOR } 5 \text{ XOR } 6 \text{ XOR } 7} \text{ XOR } \cdots \text{ XOR } \cancel{(N-3) \text{ XOR } (N-2) \text{ XOR } (N-1) \text{ XOR } N}}{\quad \downarrow \quad \quad \quad \downarrow \quad \quad \quad \downarrow} \\ &= \quad \quad 0 \quad \text{XOR} \quad \quad \quad 0 \quad \quad \quad \text{XOR} \cdots \text{ XOR} \quad \quad \quad 0 \\ &= \quad \quad \quad 0 \end{aligned}$$

そこで、 $1000000007 \bmod 4 = 3$ であるため、答えは **0** です。

問題 10

この問題は直接計算しても解けますが、プログラミングを使わなければ面倒です。そこで、上から i 番目・左から j 番目のマスに $4i + j$ が書かれていることを使って、以下のように各マスの値を分解しましょう。

4	4	4	4	4	4	4	4	4
8	8	8	8	8	8	8	8	8
12	12	12	12	12	12	12	12	12
16	16	16	16	16	16	16	16	16
20	20	20	20	20	20	20	20	20
24	24	24	24	24	24	24	24	24
28	28	28	28	28	28	28	28	28
32	32	32	32	32	32	32	32	32



1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

すべてのマスの総和

64 個のマス全体では、 $[4, 8, 12, 16, 20, 24, 28, 32]$ が 8 回ずつ、 $[1, 2, 3, 4, 5, 6, 7, 8]$ が 8 回ずつ足されています。したがって、求める答えは以下の通りです。

$$\begin{aligned} & (4 + 8 + 12 + 16 + 20 + 24 + 28 + 32) \times 8 + (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8) \times 8 \\ &= 144 \times 8 + 36 \times 8 \\ &= \mathbf{1440} \end{aligned}$$

緑色のマスの総和

すべての行・すべての列について、8 個中 4 個（半分）が緑色で塗られています。すなわち $[4, 8, 12, 16, \dots]$ などが足された回数も半分になるため、求める答えは $1440 \div 2 = \mathbf{720}$ です。

※足された回数を考えるテクニックが分からぬ人は、5.7 節を確認してください。

6.3

最終確認問題 11-15 の解答

問題 11

- 積の法則は確率にも適用できるので、答えは $(1/2)^8 = 1/256$ となります。
- まず、あるマスが白になる可能性も黒になる可能性も $1/2$ なので、
 - 白マルの個数の期待値 : $64 \times (1/2) = 32$
 - 黒マルの個数の期待値 : $64 \times (1/2) = 32$となります。期待値の線形性（→3.4.3項）より、[白の個数の期待値] $\times 2 +$ [黒の個数の期待値] $= (32 \times 2) + 32 = 96$ です。
- 期待値の線形性より、求める答えは以下の式で表されます。

$$\begin{aligned} [\text{答え}] &= [\text{1 行目が全部白になる確率}] + \cdots + [\text{8 行目が全部白になる確率}] \\ &\quad + [\text{1 列目が全部白になる確率}] + \cdots + [\text{8 列目が全部白になる確率}] \\ &\quad + [\text{1 つ目の対角線が全部白になる確率}] \\ &\quad + [\text{2 つ目の対角線が全部白になる確率}] \end{aligned}$$

そこで、すべての行・列・対角線は 8 個のマスから構成されるので、それが全部白になる確率は $(1/2)^8 = 1/256$ です。したがって、答えは $1/256 + 1/256 + \cdots + 1/256 = 18 \times (1/256) = 9/128$ となります。

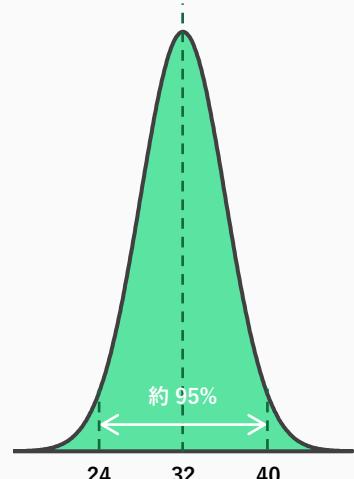
- 各マスが白マルとなる確率は $p = 0.5$ 、マスの数は全部で $n = 64$ 個あるので、白マルの個数は

$$\text{平均 } \mu : np = 64 \times 0.5 = 32$$

$$\text{標準偏差 } \sigma : \sqrt{np(1-p)} = \sqrt{64 \times 0.5 \times 0.5} = 4$$

の正規分布に近似します（→節末問題 3.5.1）。そこで

$\mu - 2\sigma = 24$ 、 $\mu + 2\sigma = 40$ であるため、68-95-99.7 則より白マスが 24 ~ 40 個となる確率は約 95% であるといえます。（右図参照）



問題 12

「123 を含まない数の個数」を考えると難しくなってしまうので、その余事象（→5.4.1項）に相当する「123 を含む数の個数」を考えてみましょう。

まず、123 を含む 999999 以下の数として、以下の 4 パターンが考えられます。ただし、場合分けが少なくなるように、5 衔以下のものは先頭を 0 で埋めた数を考えるものとします。（たとえば 1237 → 001237）

1 123???	2 ?123??	3 ??123?	4 ???123
下 3 衔を決められるので 1000 通り (123 のみ)	3 つの桁を決められるので 1000 通り (123 のみ)	3 つの桁を決められるので 1000 通り (123 のみ)	3 つの桁を決められるので 1000 通り (123 のみ)

すべてのパターンについて、3 つの桁を 0 ~ 9 の範囲で自由に選べるので、積の法則（→3.3.2項）より $10 \times 10 \times 10 = 1000$ 通りがあります。そのため、「123 を含む数は全部で 4000 個だ」と思うかもしれません。

しかし、123123 という数はパターン 1 と 4 両方に数えられてしまっているため、実際の個数は $4000 - 1 = 3999$ 個です。

1 123123	2 ?123??	3 ??123?	4 123123
-------------	-------------	-------------	-------------

したがって、「123 を含む数の個数」は全体のパターン数 999999 から「123 を含む数の個数（3999 個）」を引いた値 **996000 個** となります。

問題 13

関数 `func(N)` の計算時間を a_N とすると、この関数が `func(N-1)`、`func(N-2)`、`func(N-3)`、`func(N-3)` を順に呼び出していることから、次式が成り立ちます。

$$a_N = a_{N-1} + a_{N-2} + a_{N-3} + a_{N-3}$$

そこで、 $a_N = 2^N$ とすると $2^N = 2^{N-1} + 2^{N-2} + 2^{N-3} + 2^{N-3}$ となるため、つじつまが合います。したがって、`func(N)` の呼び出しの計算量は $O(2^N)$ です。

※注：「なんで $a_N = 2^N$ を当てはめる発想になるのだ！」と思った方は、実際に `func(N)` の実行時間を測定してみると答えのヒントが得られます。たとえば著者環境では、`func(25)` は 0.128 秒、`func(26)` は 0.259 秒であり、ほぼ 2 倍の差です。

問題 14 (1)

5人の順位の組合せは $5! = 120$ 通りですが、4回の質問の結果（誰が一番速いか）の組合せは $3^4 = 81$ 通りしかありません。後者の方が小さいため、4回で当てることはできません。（→**5.10.6項**）

問題 14 (2)

この問題には様々な解法が考えられるので、そのうち一つを紹介します。

ステップ 1

まず、5人のうち最も速かった選手を、以下の方法によって調べます。

1. A・B・C を選び、誰が一番速かったかを聞く。
2. D・E・(1. で一番速かった選手) を選び、誰が一番速かったかを聞く。

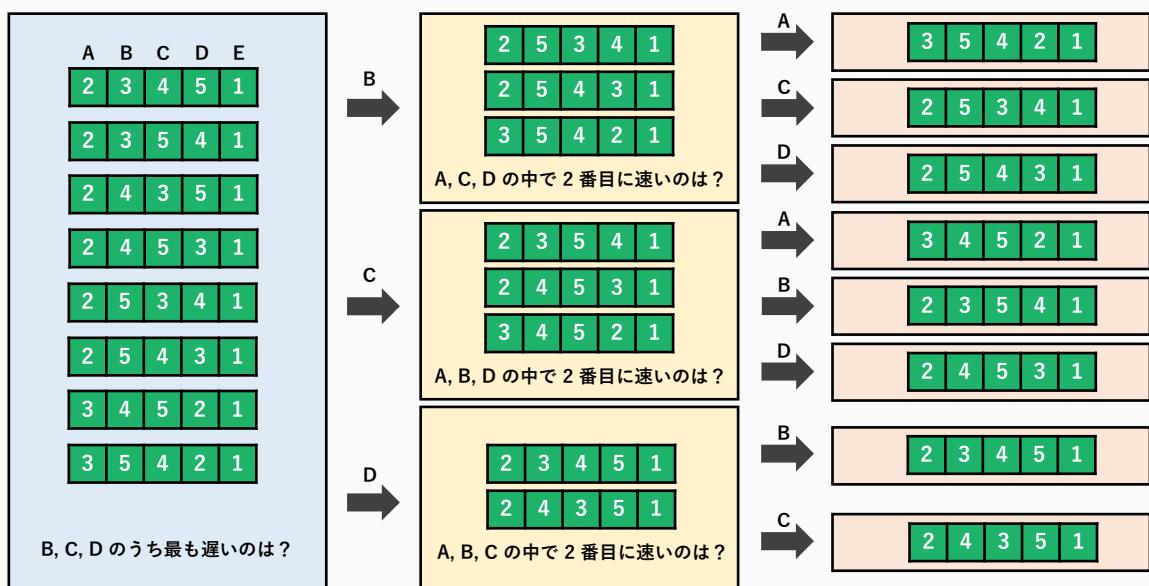
そうすると、残った4人の選手の順位を3回で当てる問題になります。以降、最も速かった選手を E であると仮定します。

ステップ 2

A・B・C の中で最も速い選手を聞きます。それが A である場合、順位の組合せは以下の8通りに絞られます。（数字は順位）

<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr><tr><td>2</td><td>3</td><td>4</td><td>5</td><td>1</td></tr></table>	A	B	C	D	E	2	3	4	5	1	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr><tr><td>2</td><td>3</td><td>5</td><td>4</td><td>1</td></tr></table>	A	B	C	D	E	2	3	5	4	1	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr><tr><td>2</td><td>4</td><td>3</td><td>5</td><td>1</td></tr></table>	A	B	C	D	E	2	4	3	5	1	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr><tr><td>2</td><td>4</td><td>5</td><td>3</td><td>1</td></tr></table>	A	B	C	D	E	2	4	5	3	1
A	B	C	D	E																																							
2	3	4	5	1																																							
A	B	C	D	E																																							
2	3	5	4	1																																							
A	B	C	D	E																																							
2	4	3	5	1																																							
A	B	C	D	E																																							
2	4	5	3	1																																							
<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr><tr><td>2</td><td>5</td><td>3</td><td>4</td><td>1</td></tr></table>	A	B	C	D	E	2	5	3	4	1	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr><tr><td>2</td><td>5</td><td>4</td><td>3</td><td>1</td></tr></table>	A	B	C	D	E	2	5	4	3	1	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr><tr><td>3</td><td>4</td><td>5</td><td>2</td><td>1</td></tr></table>	A	B	C	D	E	3	4	5	2	1	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr><tr><td>3</td><td>5</td><td>4</td><td>2</td><td>1</td></tr></table>	A	B	C	D	E	3	5	4	2	1
A	B	C	D	E																																							
2	5	3	4	1																																							
A	B	C	D	E																																							
2	5	4	3	1																																							
A	B	C	D	E																																							
3	4	5	2	1																																							
A	B	C	D	E																																							
3	5	4	2	1																																							

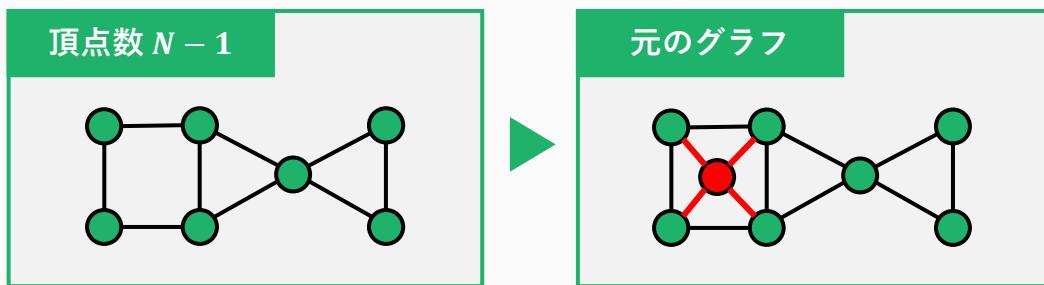
その後は以下のように質問することで、必ず2回で当てることが可能です。このようなプロセスにより、5人全員の順位を計5回で当てることができました。



問題 15

まず、平面グラフには「頂点数が辺の数の 3 倍未満である」という性質があるため、次数が 5 以下の頂点が少なくとも一つ存在します。

ですから、頂点数が $N - 1$ の平面グラフに次数 5 以下の頂点を追加することで、元のグラフを作ることができます。



そこで、頂点数 $N - 1$ の平面グラフを 5 彩色できると仮定した場合、頂点を追加したグラフが 5 彩色できることを証明しましょう (★)。以下のようにになります。

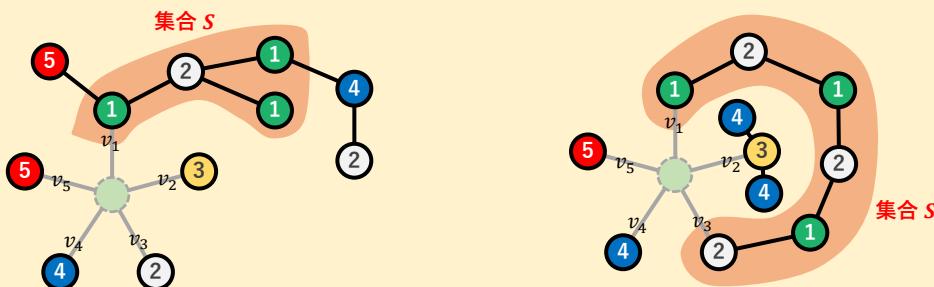
パターン 1：追加した頂点 u の次数が 4 以下である場合

下図のように、隣り合うどの頂点とも異なる色を塗れば良い。（色の選択肢は 5 つあるため、このような色は必ず存在する）



パターン 2：追加した頂点 u の次数が 5 である場合

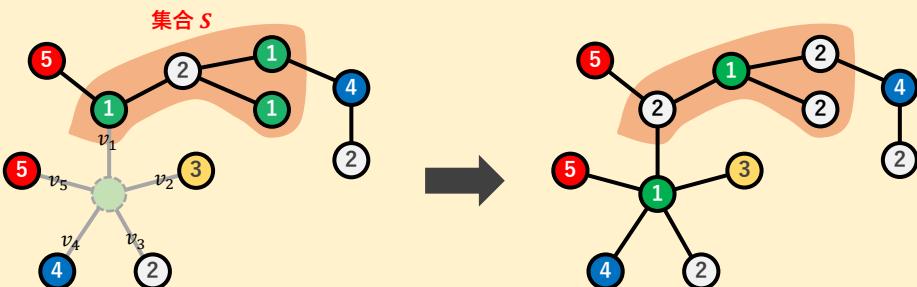
u に隣接する頂点を時計回りに v_1, v_2, v_3, v_4, v_5 とし、 v_1 の色を 1、 v_3 の色を 2 とする。また、（頂点数 $N - 1$ のグラフにおいて）頂点 v_1 から色 1, 2 の頂点だけを通ってたどり着ける頂点の集合を S とする。



そこで、 v_3 が集合 S に含まれていなかった場合、以下の操作を行うと、色 1 が空く（追加頂点 u を色 1 に設定することができるという意味）。

- S に含まれる色 1 の頂点をすべて色 2 にする。
- S に含まれる色 2 の頂点をすべて色 1 にする。

下図は、操作の具体例を示している。



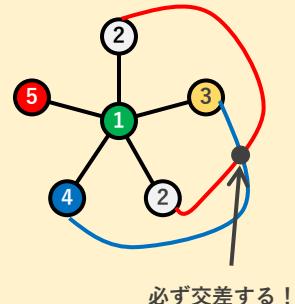
一方、 v_3 が集合 S に含まれている場合は、頂点 v_2 （色 3）と v_4 （色 4）で同じことをやれば良い。具体的には、頂点 v_2 から色 3, 4 の頂点だけを通ってたどり着ける頂点の集合を T として、

- T に含まれる色 3 の頂点をすべて色 4 にする
- T に含まれる色 4 の頂点をすべて色 3 にする

という操作を行うと、色 3 が空く。なお、

- 頂点 v_1 から v_3 へ行く、頂点 u を通らない経路
- 頂点 v_2 から v_4 へ行く、頂点 u を通らない経路

は必ず交差するため、 v_4 は絶対に集合 T に含まれない。



最後に、頂点数 1 のグラフは 5 彩色可能なので、(★) より頂点数 2 のグラフは OK、頂点数 3 のグラフは OK、頂点数 4 のグラフは OK…となり、最終的に元のグラフも 5 彩色可能であることが証明できました。

なお、より分かりやすい証明を知りたい方は、chap6-11_15.md に掲載されている「高校数学の美しい物語」のWebサイトをご覧ください。

6.4

最終確認問題 16-20 の解答

問題 16

あり得るすべてのパターンをしらみつぶしに調べる方法を「全探索」（→[2.4.5 項](#)）といいます。

この問題では、 $1 \leq a < b < c \leq N, a + b + c = X$ となる整数 (a, b, c) の組み合わせを完全に無駄なく列挙することは決して簡単ではありません。しかし、 $1 \leq a < b < c \leq N$ となる (a, b, c) を列挙することは簡単であり、そのすべてを試して、その中で $a + b + c = X$ となる個数を数える、といった方法が最もシンプルです。

この解法を C++ で実装すると、以下のようになります。 (a, b, c) のループ処理は、[コード 3.3.1](#) と似ています。

```
#include <iostream>
using namespace std;

int main() {
    // 入力
    int N, X;
    cin >> N >> X;

    // すべての (a, b, c) の組み合わせを試す
    int answer = 0;
    for (int a = 1; a <= N; a++) {
        for (int b = a + 1; b <= N; b++) {
            for (int c = b + 1; c <= N; c++) {
                if (a + b + c == X) {
                    answer += 1;
                }
            }
        }
    }

    // 答えを出力
    cout << answer << endl;
    return 0;
}
```

※ Python などのソースコードは chap6-16_20.md をご覧ください。

問題 17

長方形の面積は（縦の長さ）×（横の長さ）と計算されます。これがどちらも整数であるから、面積が N となるためには、縦の長さ・横の長さが「 N の約数」になる必要があります。

36 の約数 … 1, 2, 3, 4, 6, 9, 12, 18, 36



このため、3.1.5 項の方法で約数列挙すれば、可能な長方形を全て調べ上げることができます。この中で周の長さが最小になるものを求める事になります。

一方で、よりシンプルな解法もあります。（縦の長さ） \leq （横の長さ）としても一般性を失わない（→5.10.4 項）ことを利用すると、（縦の長さ） $\leq \sqrt{N}$ になります。つまり、縦の長さ x を 1 以上 \sqrt{N} 以下で全探索して、

- 横の長さ $N \div x$ が整数になるもの
- その中で周の長さ $2x + 2(N \div x)$ が最小になるもの

を求める事になります。計算量は $O(\sqrt{N})$ で、C++ での実装は以下のようになります。

```
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    // 入力
    long long N;
    cin >> N;

    // 縦の長さを 1 から  $\sqrt{N}$  まで全探索
    long long answer = (1LL << 60);
    for (long long x = 1; x * x <= N; x++) {
        if (N % x == 0) {
            answer = min(answer, 2 * x + 2 * (N / x));
        }
    }

    // 答えを出力
    cout << answer << endl;
    return 0;
}
```

※ Python などのソースコードは chap6-16_20.md をご覧ください。

問題 18

整数 A, B の最大公約数を $\text{GCD}(A, B)$ 、最小公倍数を $\text{LCM}(A, B)$ とします。すると、 $A \times B = \text{GCD}(A, B) \times \text{LCM}(A, B)$ という関係（→2.5.2 項）が成り立ちます。したがって、 $\text{GCD}(A, B)$ がユークリッドの互除法（→3.2 節）で計算できれば、最小公倍数は

$$\text{LCM}(A, B) = A \times B \div \text{GCD}(A, B)$$

と求められます。計算量は $O(\log(A + B))$ です。

しかし、このまま求める C++ などの言語ではオーバーフローします。なぜなら、`long long` 型でも 19 桁程度までの数しか扱えず、 $A \times B$ を計算する時点でこれを超える可能性があるからです。この問題は、以下の 2 つの方法で対処できます。

- 1 計算順序を「 $A \times B \div \text{GCD}(A, B)$ 」から「 $A \div \text{GCD}(A, B) \times B$ 」に変える
- 2 最小公倍数が 10^{18} を超えるかどうかを、うまい方法で判定する

2. に関しては、 $A \times B \div \text{GCD}(A, B) > 10^{18}$ つまり $A \div \text{GCD}(A, B) > 10^{18} \div B$ であるかを判定すればよいです。左辺は必ず整数なので、右辺は整数に切り捨ててもうまくいきます。したがって、以下のようなプログラムを書けば正解できます。

```
#include <iostream>
using namespace std;

long long GCD(long long A, long long B) {
    if (B == 0) return A;
    return GCD(B, A % B);
}

int main() {
    // 入力
    long long A, B;
    cin >> A >> B;

    // 最小公倍数が 10^18 を超えるかどうか判定
    if (A / GCD(A, B) > 1000000000000000000000000 / B) {
        cout << "Large" << endl;
    }
    else {
        cout << A / GCD(A, B) * B << endl;
    }
    return 0;
}
```

※ Python などのソースコードは chap6-16_20.md をご覧ください。

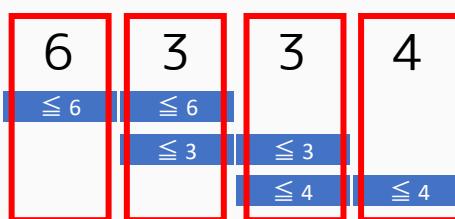
問題 19

この問題は、上界を考える（→5.8 節）テクニックを使うことで解けます。

一般の場合を考えるのは簡単ではないので、まずは具体例として $N = 4, B_1 = 6, B_2 = 3, B_3 = 4$ の場合を考えましょう。このとき、

- $\max(A_1, A_2) \leq 6 \cdots [A_1 \text{ と } A_2 \text{ は両方 } 6 \text{ 以下}]$
- $\max(A_2, A_3) \leq 3 \cdots [A_2 \text{ と } A_3 \text{ は両方 } 3 \text{ 以下}]$
- $\max(A_3, A_4) \leq 4 \cdots [A_3 \text{ と } A_4 \text{ は両方 } 4 \text{ 以下}]$

言い換えられます。したがって、 A_1 は 6 以下、 A_2 は「6 以下かつ 3 以下」だから 3 以下、 A_3 は「3 以下かつ 4 以下」だから 3 以下、 A_4 は 4 以下と分かります。



一般の場合も同様にして、 $A_1 \leq B_1, A_i \leq \min(B_{i-1}, B_i)$ ($2 \leq i \leq N - 1$), $A_N \leq B_{N-1}$ という上界が得られます。実際に、この上界を当てはめた数列 A は条件を満たし、その合計を求めるプログラムを書けば正解です。以下が C++ での実装例です。

```
#include <iostream>
#include <algorithm>
using namespace std;

int N, B[109];
int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N - 1; i++) {
        cin >> B[i];
    }

    // 数列 A の要素の合計を求める → 答えの出力
    int answer = B[1] + B[N - 1];
    for (int i = 2; i <= N - 1; i++) {
        answer += min(B[i - 1], B[i]);
    }
    cout << answer << endl;
    return 0;
}
```

※ Python などのソースコードは chap6-16_20.md をご覧ください。

問題 20

この問題は、各質問に対して合計を以下のプログラムのように求めると、計算量 $O(NQ)$ となり、実行時間制限オーバー (TLE) になってしまいます。

```
int answer1 = 0, answer2 = 0;
for (int i = L; i <= R; i++) {
    if (C[i] == 1) answer1 += P[i];
    if (C[i] == 2) answer2 += P[i];
}
```

これを高速化するために、累積和（→4.2 節）を使いましょう。まずは、1組の合計得点だけを求めることを考えます。学籍番号 i の生徒が1組のとき $A_i = P_i$ 、2組のとき $A_i = 0$ とすると、1組の合計得点は $A_L + A_{L+1} + \dots + A_R$ と計算されます。したがって、累積和を使えば各質問に $O(1)$ で答えられます。

学籍番号・組	1	2	3	4	5
得点 P_i	50	80	100	30	40
1組の得点 A_i	50	80	0	30	0
累積和	50	130	130	160	160

… 1組
… 2組

2組に対しても同様のことができます。すると、全体計算量 $O(N + Q)$ でこの問題が解けます。この解法を C++ で実装すると、以下のようになります。

```
#include <iostream>
using namespace std;

int N, C[100009], P[100009], L[100009], R[100009], S1[100009], S2[100009];
int main() {
    // 入力 → 累積和を求める
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> C[i] >> P[i];
    for (int i = 1; i <= N; i++) S1[i] = S1[i - 1] + (C[i] == 1 ? P[i] : 0);
    for (int i = 1; i <= N; i++) S2[i] = S2[i - 1] + (C[i] == 2 ? P[i] : 0);

    // 質間に答える
    cin >> Q;
    for (int i = 1; i <= Q; i++) {
        cin >> L[i] >> R[i];
        cout << S1[R[i]] - S1[L[i] - 1] << " " << S2[R[i]] - S2[L[i] - 1] << endl;
    }
    return 0;
}
```

※ Pythonなどのソースコードは chap6-16_20.md をご覧ください。

6.5

最終確認問題 21-25 の解答

問題 21 (1)

関数 $f(x) = e^x$ に対して、その微分である $f'(x)$ も e^x になるという性質があります。

したがって、 $y = e^x$ 上の点 $(1, e)$ における接線の傾きは、 $f'(1) = e^1 = e$ となります。だから、接線の方程式は $y = ex + b$ という形で表せますが、このなかで点 $(1, e)$ を通るのは $b = 0$ のときです。

よって、点 $(1, e)$ における接線の方程式は $\textcolor{red}{y = ex}$ です。

問題 21 (2)

接線 $y = ex$ と直線 $y = 2$ が交わるとき、 $ex = 2$ になるので、 $x = \frac{2}{e}$ です。したがって、この 2 直線の交点の座標は $\left(\frac{2}{e}, 2\right)$ です。

この点の x 座標を小数で表すと、 $0.735758882\cdots$ となります。

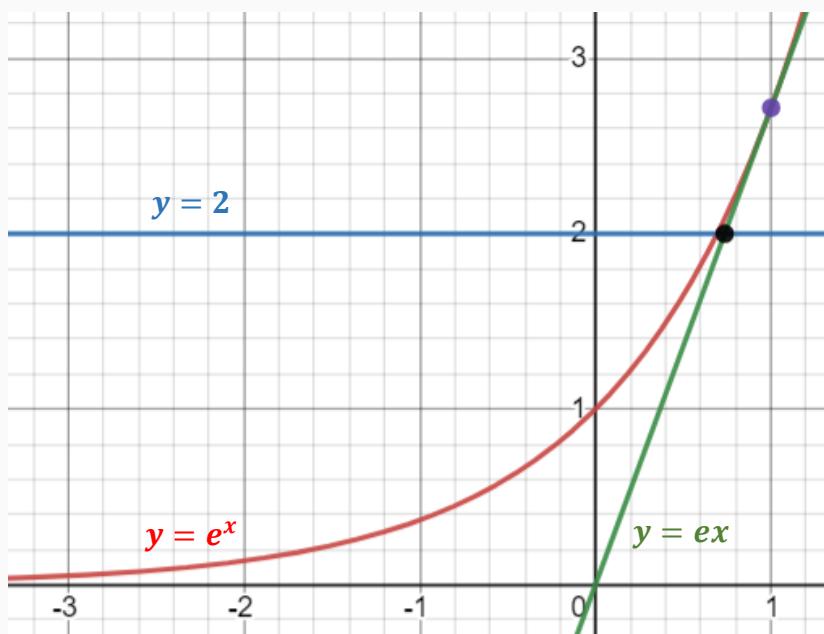


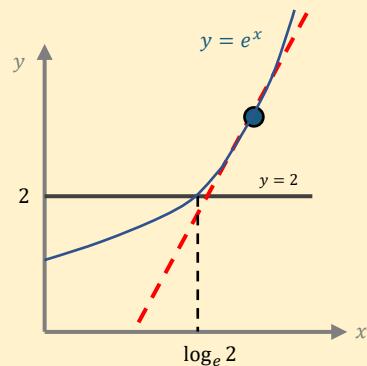
図. $y = e^x$ とその接線などのグラフ (desmos.com で描画)

問題 21 (3)

$\log_e 2$ の値はニュートン法（→4.3 節）を使って求められます。曲線 $y = e^x$ の y 座標が 2 となる点の x 座標が $\log_e 2$ であるから、以下の方針で求めることができます。

- $f(x) = e^x$ とする。ここで $f'(x) = e^x$ 。
- 最初、適当な初期値 a を設定する。
- その後、 a の値を以下に更新し続ける。

点 $(a, f(a))$ における接線と直線
 $y = 2$ の交点の x 座標



そのため、以下のようなプログラムを書けばよいです。なお、`exp(x)` は e^x を返す関数です。別の方法として、代わりに `pow(2.718281828, x)` と書いてても、ほぼ同じ結果が得られます。

```
#include <cmath>
#include <iostream>
using namespace std;

int main() {
    double r = 2.0; // y = e^x と y = 2 の交点を求めたいから
    double a = 1.0; // 初期値を適当に 1.0 にセットする

    for (int i = 1; i <= 5; i++) {
        // 点 (a, f(a)) の x 座標と y 座標を求める
        double zahyou_x = a;
        double zahyou_y = exp(a);      ← コード 4.3.1 からの変更部分

        // 接線の式 y = sessen_a * x + sessen_b を求める
        double sessen_a = zahyou_y;    ← コード 4.3.1 からの変更部分
        double sessen_b = zahyou_y - sessen_a * zahyou_x;

        // 次の a の値 next_a を求める
        double next_a = (r - sessen_b) / sessen_a;
        printf("Step #%d: a = %.15lf -> %.15lf\n", i, a, next_a);
        a = next_a;
    }
    return 0;
}
```

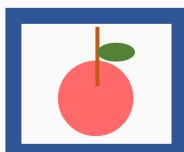
このとき、出力は以下のようになります。急激に $\log_e 2 = 0.693147180559945 \dots$ に近づき、たった 5 回で 15 桁目まで一致します。

```
Step #1: a = 1.000000000000000 -> 0.735758882342885  
Step #2: a = 0.735758882342885 -> 0.694042299918915  
Step #3: a = 0.694042299918915 -> 0.693147581059771  
Step #4: a = 0.693147581059771 -> 0.693147180560026  
Step #5: a = 0.693147180560026 -> 0.693147180559945
```

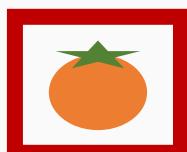
Python・JAVA・C のソースコードは、GitHub の chap6-21_25.md をご覧ください。

問題 22

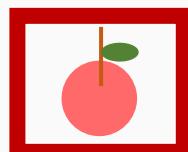
使う 2 台のオーブンを「オーブン A」「オーブン B」とします。オーブン A で消費される時間を a 、オーブン B で消費される時間を b とすると、料理にかかる全体の時間は $\max(a, b)$ となります。ここで、どのようにオーブンの割り当てを決めて、 $a + b$ の値は $sumT = T_1 + T_2 + \dots + T_N$ で変わらないので、 a が決まれば $b = sumT - a$ と自動的に決まり、料理にかかる全体の時間も $\max(a, sumT - a)$ と決まります。



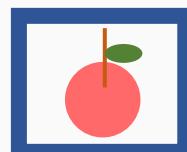
6 分



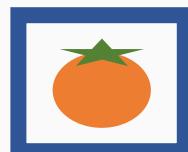
11 分



12 分



7 分



オーブン A … 21 分 オーブン B … 23 分

さて、どのような a が「実現可能」なのでしょうか？上図の例では、実現可能な a をすべて挙げると 0, 6, 7, 8, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 23, 24, 25, 26, 27, 29, 30, 31, 32, 33, 36, 37, 38, 44 分となります。

実は、実現可能な a は、**節末問題 3.7.4** と非常によく似た動的計画法のアルゴリズムを使って、計算量 $O(N \times sumT)$ ですべて挙げることができます。

用意する配列（二次元配列）

$dp[i][j]$: 料理 i までのなかから、オープン A で消費する時間の和（以下、単に消費時間と呼ぶ）が j になる組合せが存在するなら `true`、そうでなければ `false`

動的計画法の遷移 ($i = 0$)

明らかに「何も選ばない」という方法しか存在しないので、

- $dp[0][j] = \text{true}$ ($j = 0$)
- $dp[0][j] = \text{false}$ ($j \neq 0$)

となります。

動的計画法の遷移 ($i = 1, 2, \dots, N$ の順に計算)

総和が j になるように料理 i までのなかから選ぶ方法は、以下の 2 つがあります。

(最後の行動 [料理 i を焼くオープン] で場合分けします)

- 料理 $i - 1$ までの消費時間が $j - A_i$ であり、料理 i をオープン A で焼く
 - 料理 $i - 1$ までの消費時間が j であり、料理 i をオープン A では焼かない
- したがって、 $dp[i - 1][j - A_i], dp[i - 1][j]$ のうち少なくとも一方が `true` の場合 $dp[i][j] = \text{true}$ 、そうでなければ `false` となります。

最終的に、 $dp[N][x] = \text{true}$ のとき $a = x$ が実現可能になります。実現可能な a の中で、 $\max(a, sumT - a)$ が最小になるものが答えになります。

この解法を C++ で実装すると、以下のようになります。

```
#include <iostream>
#include <algorithm>
using namespace std;

int N, T[109]; bool dp[109][100009];

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> T[i];

    // 配列の初期化
    int sumT = 0;
    for (int i = 1; i <= N; i++) sumT += T[i];
    for (int i = 1; i <= sumT; i++) dp[0][i] = false;
    dp[0][0] = true;
```

```

// 動的計画法
for (int i = 1; i <= N; i++) {
    for (int j = 0; j <= sumT; j++) {
        if (j < T[i]) {
            if (dp[i - 1][j] == true) dp[i][j] = true;
            else dp[i][j] = false;
        }
        if (j >= T[i]) {
            if (dp[i-1][j] == true || dp[i-1][j-T[i]] == true) dp[i][j] = true;
            else dp[i][j] = false;
        }
    }
}

// 答えを計算して出力
int answer = (1 << 30);
for (int i = 0; i <= sumT; i++) {
    if (dp[N][i] == true) {
        int cooking_time = max(i, sumT - i);
        answer = min(answer, cooking_time);
    }
}
cout << answer << endl;
return 0;
}

```

Python・JAVA・C のソースコードは、GitHub の chap6-21_25.md をご覧ください。

問題 23

エラトステネスのふるい（→4.4.1 項）を使うと、 N 以下の素数を $O(N \log \log N)$ 時間で列挙できます。しかし、この問題で列挙すべきなのは「 L 以上 R 以下の素数」であります。以下のようにアルゴリズムを少し変えてみましょう。

- 最初、整数 $L, L + 1, \dots, R$ を書く。
- 書かれている全ての 2 の倍数に \times を付ける。例外として、2 には \times を付けない。
- 書かれている全ての 3 の倍数に \times を付ける。例外として、3 には \times を付けない。
- 書かれている全ての 4 の倍数に \times を付ける。例外として、4 には \times を付けない。
- （中略）
- 書かれている全ての $\lfloor \sqrt{R} \rfloor$ の倍数に \times を付ける。例外として、 $\lfloor \sqrt{R} \rfloor$ には \times を付けない。
- 無印のまま残った整数だけが素数である。

次に、実装方法を考えます。プログラミングでは整数を直接書くことはできないので、代わりに長さ $R - L + 1$ の配列 `prime` を持って、`prime[x]` には「整数 $x + L$ が無印かどうか」（無印なら `true`、 \times が付けられているなら `false`）を記録すると、うまく実装できます。

i 番目の操作で \times が付けられるのは $[L/i] \times i, [L/i + 1] \times i, \dots, [R/i] \times i$ であることに注意してください。 \times が付けられるのは大体 $(R - L)/i$ 個なので、全体計算量は

$$\frac{R - L}{2} + \frac{R - L}{3} + \dots + \frac{R - L}{\sqrt{R}} = O((R - L) \log \sqrt{R})$$

となります。この証明は、**4.4.4 節・4.4.5 節**を参照してください。

この解法を C++ で実装すると、以下のようになります。

```
#include <iostream>
using namespace std;

long long L, R; bool prime[500009];

int main() {
    // 入力
    cin >> L >> R;

    // 配列の初期化・L=1 のときの場合分け（コーナーケース）
    for (long long i = 0; i <= R - L; i++) {
        prime[i] = true;
    }
    if (L == 1) prime[0] = false;

    // ふるい
    for (long long i = 2; i * i <= R; i++) {
        long long min_value = ((L + i - 1) / i) * i; // L 以上で最小の i の倍数
        // L 以上 R 以下の (i を除く) i の倍数すべてにバツを付ける
        for (long long j = min_value; j <= R; j += i) {
            if (j == i) continue;
            prime[j - L] = false;
        }
    }

    // 個数を数えて出力
    long long answer = 0;
    for (long long i = 0; i <= R - L; i++) {
        if (prime[i] == true) answer += 1;
    }
    cout << answer << endl;
    return 0;
}
```

さらに工夫すると、計算量 $O\left(\left(\sqrt{R} + (R - L)\right) \log \log \sqrt{R}\right)$ で解くこともできます。あらかじめエラトステネスのふるいで \sqrt{R} 以下の素数を求めておけば、「合成数 (4, 6, 8, 9, ...) の倍数に × を付ける」といった無駄な操作が省けて、この計算量になります。

Python・JAVA・C のソースコードは、GitHub の chap6-21_25.md をご覧ください。

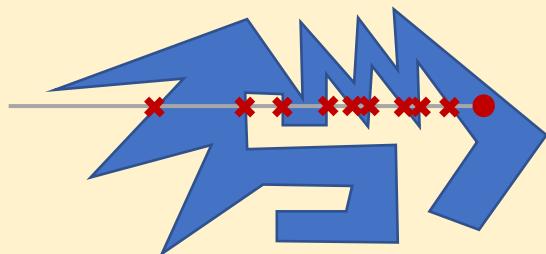
問題 24

計算幾何学（→4.1 節）の問題です。この問題では、多角形が凸（内角がすべて 180 度未満の多角形）とは限らないので、複雑に入れ組んでいる場合もあります。このような場合も含めて、点が多角形の内部に入っているか判定するには、どうすればよいのでしょうか？

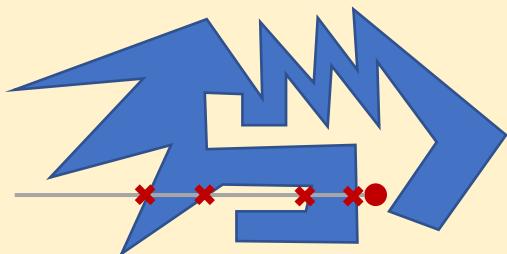
実は、以下のような単純な方法で判定できます。

点 (A, B) が多角形に含まれるかの判定

1. 点 (A, B) から左に向かって半直線を引きます。
2. この半直線が、多角形の辺と交わった回数を数えます。これが奇数回ならば点 (A, B) は多角形の内部に、偶数回なら多角形の外部にあります。



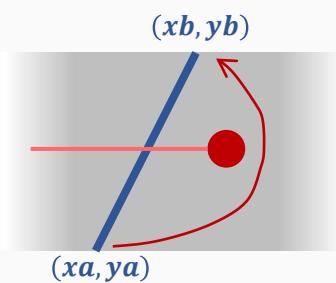
9 回交わるので 内部



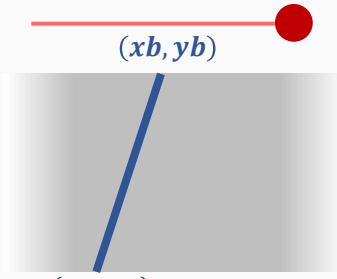
4 回交わるので 外部

つまり、多角形の各辺に対して、点 (A, B) から左に向かって引いた半直線と交わるかどうか判定することになります。辺が (xa, ya) と (xb, yb) を結ぶ線分（ただし $ya < yb$ ）だとして、**基本的には**以下の条件を満たしたときに交わります。

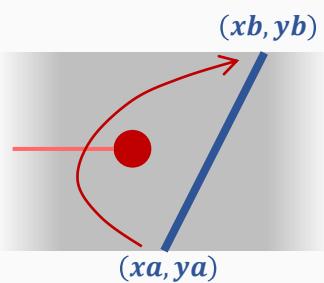
- 1 $ya \leq B \leq yb$ である。
- 2 点 (xa, ya) 、点 (A, B) 、点 (xb, yb) がこの順で反時計回りになっている。



条件 1・2 を両方満たす

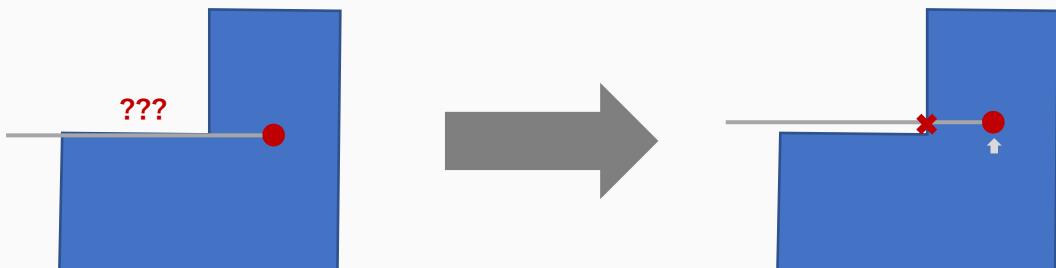


条件 1 を満たさない
($ya \leq B \leq yb$ ではない)



条件 2 を満たさない
(時計回りになっている)

しかし、多角形の辺が $y = B$ で水平になる場合は例外で、この辺を見ただけでは本当に交わっているのか判断がつきません。このようなケースをなくすために、点をほんの少しだけ上にずらして考えます（そうしても答えは変わりません）。



すると、条件 1 が少しだけ変わり、以下のようになります。なぜなら「 $ya \leq B + \epsilon \leq yb$ (ϵ は無限に小さい数)」が条件になるからです。

- 1 $ya \leq B < yb$ である。
- 2 点 (xa, ya) 、点 (A, B) 、点 (xb, yb) がこの順で反時計回りになっている。

条件 2 は外積を使って判定できます（→4.1.5 項）。多角形の辺のなかでこの条件を満たすものを数え、これが奇数個か偶数個かを判定することでこの問題が解けます。

この解法を C++ で実装すると、以下のようになります。

```
#include <iostream>
#include <algorithm>
using namespace std;

int N; long long X[100009], Y[100009], A, B;

int main() {
```

```

// 入力
cin >> N;
for (int i = 0; i < N; i++) cin >> X[i] >> Y[i];
cin >> A >> B;

// 交差する回数を数える
int cnt = 0;
for (int i = 0; i < N; i++) {
    long long xa = X[i] - A, ya = Y[i] - B;
    long long xb = X[(i + 1) % N] - A, yb = Y[(i + 1) % N] - B;
    if (ya > yb) {
        swap(xa, xb);
        swap(ya, yb);
    }
    if (ya <= 0 && 0 < yb && xa * yb - xb * ya < 0) {
        cnt += 1;
    }
}
}

// 答えを出力
if (cnt % 2 == 1) cout << "INSIDE" << endl;
else cout << "OUTSIDE" << endl;

return 0;
}

```

Python・JAVA・C のソースコードは、GitHub の chap6-21_25.md をご覧ください。

問題 25

この問題で、幅優先探索（→4.5.7 項）を使って最短経路を求めていると、計算量 $O(N^2)$ かかってしまいますが、「足された回数を考える」テクニック（→5.7 節）を巧みに使うと計算量 $O(N)$ で解くことができます。

まずは具体例として、以下の 5 頂点の木を考えてみましょう。頂点のペアは 10 通りあり、それぞれ最短経路は以下のようになっています。

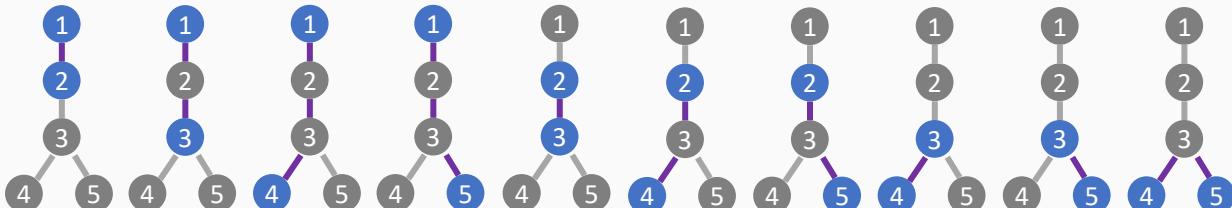


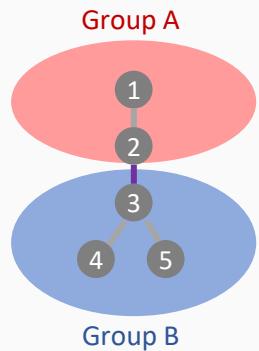
図. 10 通りのペアそれぞれに対する最短経路

これを見ると、以下のことが分かります。

- 頂点 1-2 を結ぶ辺は、4 つの最短経路 (1・2・3・4 番目) に含まれている
- 頂点 2-3 を結ぶ辺は、6 つの最短経路 (2・3・4・5・6・7 番目) に含まれている
- 頂点 3-4 を結ぶ辺は、4 つの最短経路 (3・6・8・10 番目) に含まれている
- 頂点 3-5 を結ぶ辺は、4 つの最短経路 (4・7・9・10 番目) に含まれている

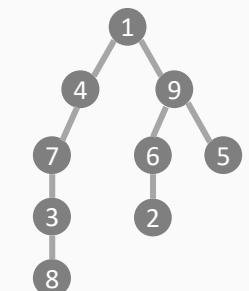
だから答えは $4 + 6 + 4 + 4 = 18$ 、と求める方針が「足された回数を考える」テクニックを使ったやり方です。

では、特定の辺が何個の最短経路に含まれているかは、どうやって分かるのでしょうか？右図のように、辺は木を 2 つの部分に分けます。それぞれ A 頂点のグループ・ $N - A$ 頂点のグループに分かれるとすると、辺は $A \times (N - A)$ 個の最短経路に含まれています。なぜなら、Group A の頂点から Group B の頂点までの最短経路に必ず含まれるからです。



したがって、それぞれの辺について、何頂点のグループと何頂点のグループに分かれかかるかが求まれば、この問題が解けます。

さて、この木を頂点 1 を持つてぶら下げる考えをみてみましょう。すると、右図のような根っこのようなグラフの形ができます。このような木を「頂点 1 を根とする根付き木」といいます。（例えば、上司と部下の関係を表すグラフ（→4.5.3 項）が根付き木として表せます）

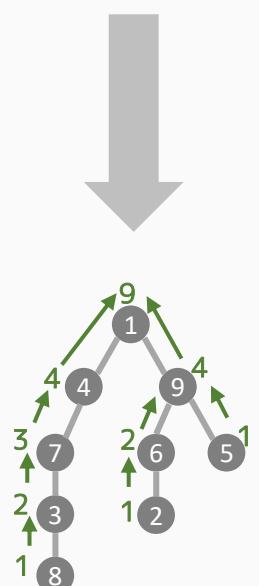


頂点 v の配下に（自分自身を含め） c_v 個の頂点があるとします。すると、頂点 v とその真上にある頂点を結ぶ辺は、グラフを c_v 頂点と $N - c_v$ 頂点のグループに分けることになります。

c_v は動的計画法を使って求められます。頂点 v の直下にある頂点を s_1, s_2, \dots, s_k とすると

$$c_v = c_{s_1} + c_{s_2} + \dots + c_{s_k} + 1$$

と計算できるので、 c_v を根付き木の下から順に求めていくと、計算量 $O(N)$ すべての c_v が求まります。この問題の答えは、 $c_v \times (N - c_v)$ ($v = 2, 3, \dots, N$) の総和になります。



この解法を C++ で実装すると、以下のようになります。なお、動的計画法の部分は、深さ優先探索（→4.5.6 項）と同様に再帰関数を用いると、比較的簡単に実装できます。

```
#include <vector>
#include <iostream>
using namespace std;

int N, M, A[100009], B[100009]; vector<int> G[100009];

int dp[100009]; bool visited[100009];
void dfs(int pos) {
    visited[pos] = true;
    dp[pos] = 1;
    for (int i : G[pos]) {
        if (visited[i] == false) {
            dfs(i);
            dp[pos] += dp[i];
        }
    }
}

int main() {
    // 入力
    int N;
    cin >> N;
    for (int i = 1; i <= N - 1; ++i) {
        cin >> A[i] >> B[i];
        G[A[i]].push_back(B[i]);
        G[B[i]].push_back(A[i]);
    }

    // 深さ優先探索 (DFS) を使った動的計画法
    for (int i = 1; i <= N; i++) {
        visited[i] = false;
    }
    dfs(1);

    // 答えを計算して出力
    long long answer = 0;
    for (int i = 2; i <= N; i++) {
        answer += 1LL * dp[i] * (N - dp[i]);
    }
    cout << answer << endl;

    return 0;
}
```

Python・JAVA・C のソースコードは、GitHub の chap6-21_25.md をご覧ください。

6.6

最終確認問題 26-30 の解答

問題 26

実験開始から n 秒後の物質 A, B, C の量をそれぞれ a_n, b_n, c_n とします。 a_n, b_n, c_n は、以下の漸化式によって決まります。

- $a_{n+1} = (1 - X)a_n + Yb_n$
- $b_{n+1} = (1 - Y)b_n + Zc_n$
- $c_{n+1} = (1 - Z)c_n + Xa_n$

この (a_n, b_n, c_n) と $(a_{n+1}, b_{n+1}, c_{n+1})$ の関係は、行列（→4.7 節）を使って以下のように表せます。

$$\begin{bmatrix} a_{n+1} \\ b_{n+1} \\ c_{n+1} \end{bmatrix} = \begin{bmatrix} 1 - X & Y & 0 \\ 0 & 1 - Y & Z \\ X & 0 & 1 - Z \end{bmatrix} \begin{bmatrix} a_n \\ b_n \\ c_n \end{bmatrix}$$

この漸化式を繰り返し適用すると、 a_T, b_T, c_T は以下の式で表せます。（実験開始時点での物質 A, B, C の量はそれぞれ 1 グラムずつです）

$$\begin{bmatrix} a_T \\ b_T \\ c_T \end{bmatrix} = \begin{bmatrix} 1 - X & Y & 0 \\ 0 & 1 - Y & Z \\ X & 0 & 1 - Z \end{bmatrix}^T \begin{bmatrix} a_0 \\ b_0 \\ c_0 \end{bmatrix} = \begin{bmatrix} 1 - X & Y & 0 \\ 0 & 1 - Y & Z \\ X & 0 & 1 - Z \end{bmatrix}^T \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

3×3 行列の累乗は、繰り返し二乗法（→4.6.7 項）を行列に適用することで、計算量 $O(\log T)$ で計算できます。これで、この問題の答えが求まります。

この解法を C++ で実装すると、以下のようになります。

```
#include <iostream>
using namespace std;

struct matrix {
    double x[3][3] = {
        { 0.0, 0.0, 0.0 },
        { 0.0, 0.0, 0.0 },
        { 0.0, 0.0, 0.0 }
    };
};
```

```

// 行列の掛け算
matrix multiplication(matrix A, matrix B) {
    matrix C;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 3; k++) {
                C.x[i][j] += A.x[i][k] * B.x[k][j];
            }
        }
    }
    return C;
}

// 行列の累乗
matrix power(matrix A, int n) {
    matrix P = A, Q;
    bool flag = false;
    for (int i = 0; i < 30; i++) {
        if ((n & (1 << i)) != 0) {
            if (flag == false) { Q = P; flag = true; }
            else Q = multiplication(Q, P);
        }
        P = multiplication(P, P);
    }
    return Q;
}

int main() {
    int Q, T; double X, Y, Z;
    cin >> Q;
    for (int t = 1; t <= Q; t++) {
        // 入力 → 行列 A の構築
        cin >> X >> Y >> Z >> T;
        matrix A;
        A.x[0][0] = 1.0 - X; A.x[2][0] = X;
        A.x[1][1] = 1.0 - Y; A.x[0][1] = Y;
        A.x[2][2] = 1.0 - Z; A.x[1][2] = Z;

        // 行列累乗の計算 → 答えを出力
        matrix B = power(A, T);
        double answerA = B.x[0][0] + B.x[0][1] + B.x[0][2];
        double answerB = B.x[1][0] + B.x[1][1] + B.x[1][2];
        double answerC = B.x[2][0] + B.x[2][1] + B.x[2][2];
        printf("%.12lf %.12lf %.12lf\n", answerA, answerB, answerC);
    }
    return 0;
}

```

Python・JAVA・C のソースコードは、GitHub の chap6-26_30.md をご覧ください。

問題 27

書かれている整数の差が k 以上になるボールの選び方の数を求める問題を「問題 k 」ということになります。問題 k は、以下のように小問題に分解（→5.6 節）できます。

- （差が k 以上になるように）1 個のボールを選ぶ方法は何通り？
- 差が k 以上になるように 2 個のボールを選ぶ方法は何通り？
- 差が k 以上になるように 3 個のボールを選ぶ方法は何通り？
- （中略）
- 差が k 以上になるように $[N/k]$ 個のボールを選ぶ方法は何通り？

$[N/k]$ 個で打ち切ってよい理由は、どう上手く選んでも最大で $[N/k]$ 個のボールしか選べないからです。このように、問題 k は $[N/k]$ 個の「より解きやすそうな小問題」に分解できました。これをもとに問題 $1, 2, 3, \dots, N$ を解くと、全体で

$$\left\lceil \frac{N}{1} \right\rceil + \left\lceil \frac{N}{2} \right\rceil + \left\lceil \frac{N}{3} \right\rceil + \cdots + \left\lceil \frac{N}{N} \right\rceil$$

個の小問題を解くことになります。これは $O(N \log N)$ 個であり、逆数 $1/x$ の和が $O(\log N)$ になる（→4.4.4 項）性質から分かります。

では、各小問題がどうやって解けるか考えてみましょう。

“小問題”はどんな問題か？

$1, 2, \dots, N$ が書かれたボールから、どの 2 つの数も m 以上離れるように、 k 個のボールを選ぶ方法は何通りか？

小問題の答え

この小問題の答えは $\textcolor{red}{N-(k-1)(m-1)C_m}$ 通りです。

理由は、ボール x を選ぶときに以下の図のようにボール $x, x+1, \dots, x+m-1$ を囲むことを考えると、ボール $1, 2, \dots, N+m-1$ から k 個の互いに重ならない「長さ m の囲い」を配置する問題になるからです。これは、単体のボール $N+m-1-km$ 個と囲い k 個を並び替える問題に読み替えられます。



$N = 9, m = 3, k = 3$ の場合の例

2 個の単体のボールと 3 個の囲いを並び替えるので、 ${}_5C_3 = 10$ 通り

二項係数は、階乗を前計算する方法（→[コード 4.6.6](#)）を使うと、 $M = 10^9 + 7$ として $O(\log M)$ 時間で求められます。先ほど述べたように、 $O(N \log N)$ 個の小問題を解くことになるので、この問題は全体計算量 $O(N \log N \log M)$ で解けることになります。

この解法を C++ で実装すると、以下のようになります。

```
#include <iostream>
using namespace std;

const long long mod = 1000000007;
int N; long long fact[100009];

long long modpow(long long a, long long b, long long m) {
    // 繰り返し二乗法 (p は a^1, a^2, a^4, a^8, ... といった値をとる)
    long long p = a, Answer = 1;
    for (int i = 0; i < 30; i++) {
        if ((b & (1 << i)) != 0) { Answer *= p; Answer %= m; }
        p *= p; p %= m;
    }
    return Answer;
}

long long Division(long long a, long long b, long long m) {
    return (a * modpow(b, m - 2, m)) % m;
}

long long ncr(int n, int r) {
    // ncr は  $n!$  を  $r! \times (n-r)!$  で割った値
    return Division(fact[n], fact[r] * fact[n - r] % mod, mod);
}

int main() {
    // 配列の初期化 (fact[i] は i の階乗を 1000000007 で割った余り)
    fact[0] = 1;
    for (int i = 1; i <= 100000; i++) {
        fact[i] = 1LL * i * fact[i - 1] % mod;
    }

    // 入力 → 答えを計算して出力
    cin >> N;
    for (int i = 1; i <= N; i++) {
        int answer = 0;
        for (int j = 1; j <= (N + i - 1) / i; j++) {
            answer += ncr(N - (i - 1) * (j - 1), j);
            answer %= mod;
        }
        cout << answer << endl;
    }
    return 0;
}
```

Python・JAVA・C のソースコードは、GitHub の chap6-26_30.md をご覧ください。

問題 28

この問題は、5.7.5 項の足し算ピラミッドの問題と似ていますが、色に対して規則が定められているので取り掛かりにくそうに見えます。ここで、色を次のように ID に直して考えると、後々都合がよいです。

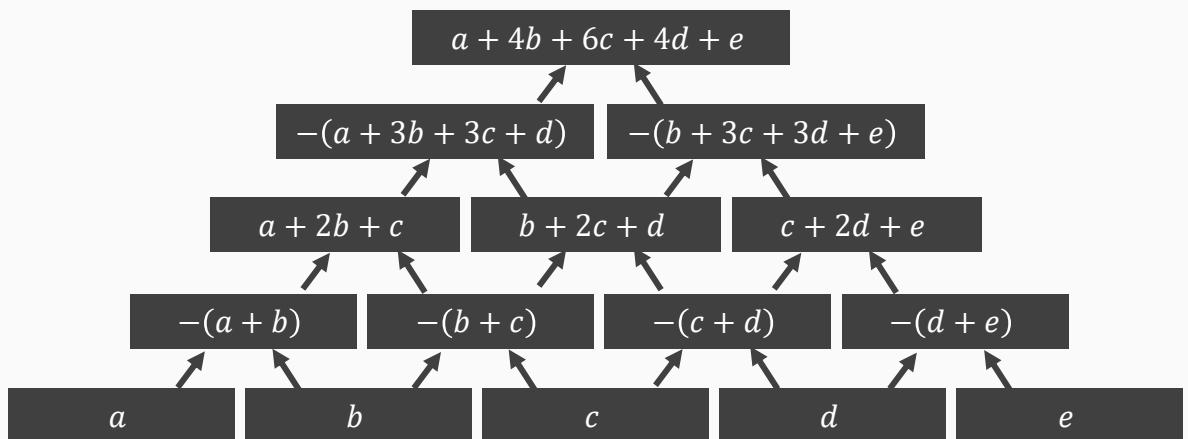
- 「青」→ 0、「白」→ 1、「赤」→ 2

直下にある 2 つのブロックの色の ID を x, y とすると、上のブロックの色は右表のようになります。

$y \backslash x$	0	1	2
0	0	2	1
1	2	1	0
2	1	0	2

つまり、直下のブロックの色の ID を x, y とすると、上のブロックの色の ID は $-(x + y) \bmod 3$ と計算できることになります※。

これを $N = 5$ 段のピラミッドの場合でも考えてみましょう。下のブロックの色を a, b, c, d, e とすると、各ブロックの色は下図のようになります（すべて $\bmod 3$ で考えています）。



一般の場合も同様に、一番上の色の ID は

$$(-1)^{N-1} \cdot (c_1 \cdot_{N-1} C_0 + c_2 \cdot_{N-1} C_1 + \cdots + c_N \cdot_{N-1} C_{N-1}) \bmod 3$$

で求まることになります。

では、 $nCr \bmod 3$ はどうやって求めるのでしょうか？3 は n, r に比べて小さいので、逆元を使った方法（→4.6.8 項）は通用しません。

n を 3 進法で表すと $n_{d-1}n_{d-2}\dots n_1n_0$ 、 k を 3 進法で表すと $k_{d-1}k_{d-2}\dots k_1k_0$ と表されます。このとき、

ここで、リュカの定理を使いましょう。

n を 3 進法で表すと $n_{d-1}n_{d-2} \dots n_1n_0$ 、 r を 3 進法で表すと $r_{d-1}r_{d-2} \dots r_1r_0$ と表さるとします。このとき、 $nCr \bmod 3$ は

$$(n_{d-1}Cr_{d-1} \times n_{d-2}Cr_{d-2} \times \dots \times n_1Cr_1 \times n_0Cr_0) \bmod 3$$

と計算されます。ただし、式の途中で $n_i < r_i$ となる場合、 $nCr \bmod 3 = 0$ となります。一般の $\bmod M$ に対しても同様の定理が成り立ちます。

nCr は計算量 $O(\log n)$ で計算でき、この問題は全体計算量 $O(N \log N)$ で解けました。

この解法を C++ で実装すると、以下のようになります。なお、 nCr の計算は、再帰関数を用いて実装されています。（[2.1.9 項](#)の方法で 3 進法に変換することも可能です）

```
#include <string>
#include <iostream>
using namespace std;

// リュカの定理で ncr mod 3 を計算
int ncr(int x, int y) {
    if (x < 3 && y < 3) {
        if (x < y) return 0;
        if (x == 2 && y == 1) return 2;
        return 1;
    }
    return ncr(x / 3, y / 3) * ncr(x % 3, y % 3) % 3;
}

int main() {
    // 入力
    int N; string C;
    cin >> N >> C;

    // 答えを求める
    int answer = 0;
    for (int i = 0; i < N; i++) {
        int code;
        if (C[i] == 'B') code = 0;
        if (C[i] == 'W') code = 1;
        if (C[i] == 'R') code = 2;
        answer += code * ncr(N - 1, i);
        answer %= 3;
    }

    // 答えを (-1)^(N-1) で掛ける
    if (N % 2 == 0) {
        answer = (3 - answer) % 3;
    }
}
```

```

// 答えを出力 ("BWR" の answer 文字目)
cout << "BWR"[answer] << endl;

return 0;
}

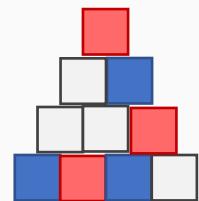
```

Python・JAVA・C のソースコードは、GitHub の chap6-26_30.md をご覧ください。

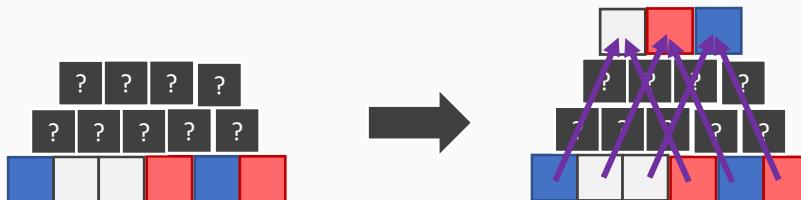
問題 28 — 別解

この問題には別解があります。「規則性を考える」（→5.2 節）を使った解法です。

$N = 4$ の場合を考えてみましょう。実は、どのような場合でも、一番上のブロックの色は、一番左のブロックと一番右のブロックだけで決まります。2, 3 番目のブロックは関係ありません。右図の例では、一番左のブロックは青、一番右のブロックは白なので、一番上のブロックは赤になります。



$N = 10$ の場合も同様で、一番左のブロックと一番右のブロックだけで決まり、2, 3, 4, 5, 6, 7, 8, 9 番目のブロックは関係ありません。 $N = 28, 82, 244, 730, 2188, 6562, \dots$ の場合も同様です。一般に、影響範囲の一番左のブロックと一番右のブロックを見るだけで、 3^k 段上のブロックの配置が求まることがあります。



これを巧みに利用すれば、 $O(\log N)$ 段のブロックしか計算する必要がなくなります。例えば $N = 23$ の場合、3 進法を利用すると $22 = 9 + 9 + 3 + 1$ なので、1 段目のブロックを入力してから、10 段目を求め、19 段目を求め、22 段目を求め、23 段目を求める、といった流れになります。各ステップには計算量 $O(N)$ しかかかりず、全体で $O(\log N)$ ステップがあるので、全体計算量 $O(N \log N)$ でこの問題が解けます。

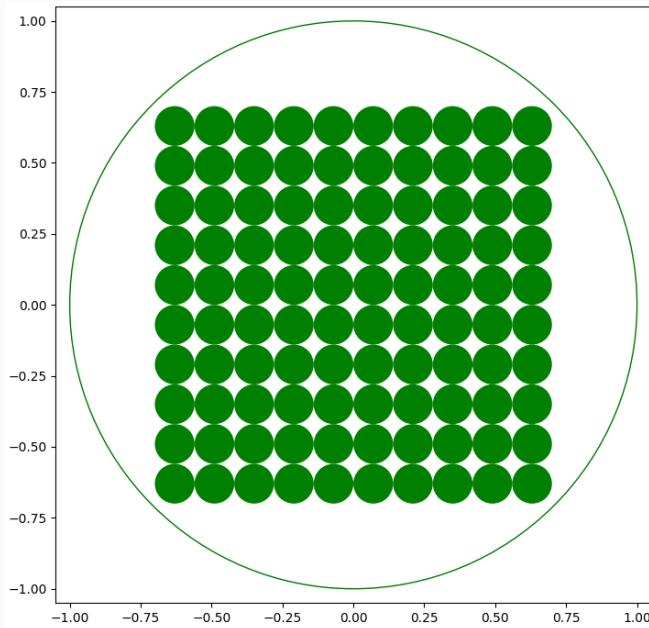
実装コードは、紙面の都合上省略します。

問題 29

この問題は、半径 1 の円にできるだけ半径の大きな 100 個の小さな円を敷き詰める問題です。半径が大きい配置ほど高い得点が得られる形式なので、この問題に対しては様々な解法が考えられます。

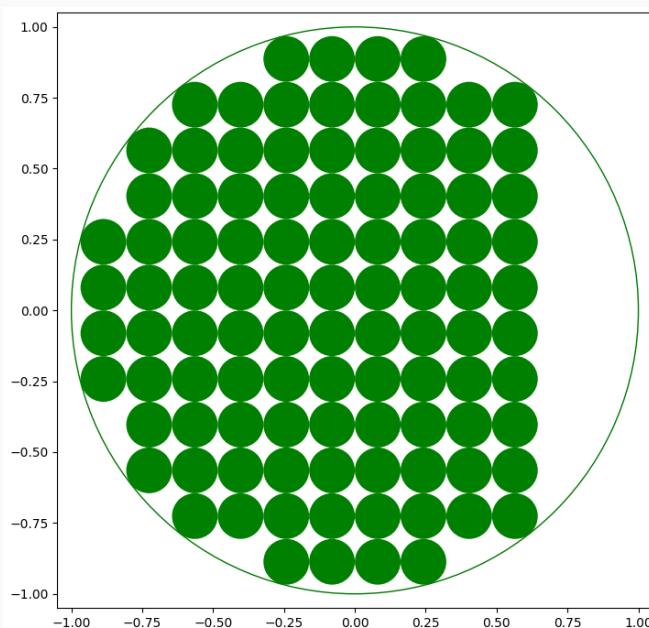
解法 0 — AtCoder 上の出力例 ($R = 0.07$)

これは AtCoder の問題文の出力例にも書かれた、半径 $R = 0.07$ の敷き詰め方です。



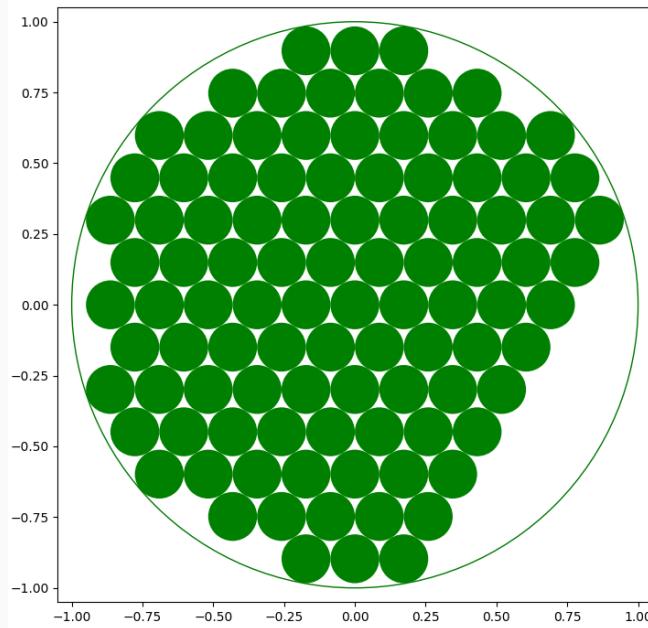
解法 1 — 正方形状に敷き詰め ($R = 0.0806$)

解法 0 の敷き詰め方だと、上下左右にまだスペースがあります。ここも敷き詰めると、 $R = 0.0806$ が達成できます。



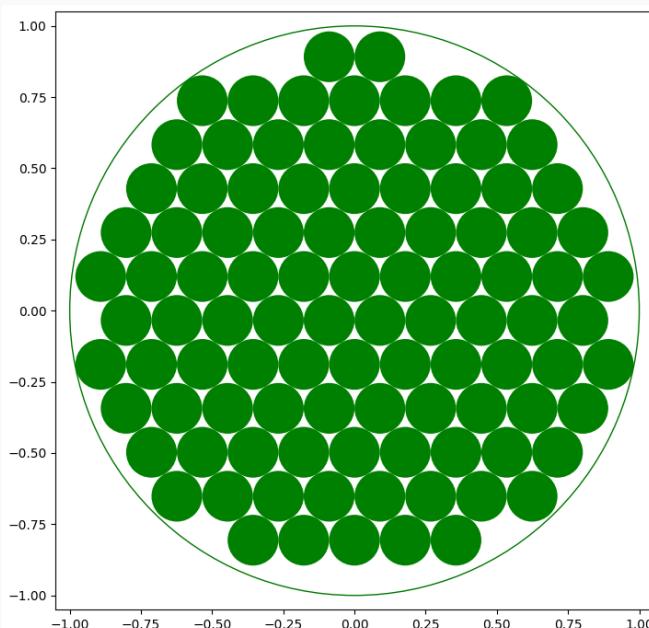
解法 2 — 六角形状に敷き詰め ($R = 0.0863$)

六角形の形に敷き詰めると、さらに効率的に敷き詰められます。敷き詰め可能な半径を二分探索を使って求めると、 $R = 0.0863$ まで作れることが分かります。



解法 3 — 敷き詰めの中心位置をずらす ($R = 0.0891$)

解法 2 では、最も真ん中の円の中心座標を $(0, 0)$ に固定していました。これをずらすと、さらに効率的な敷き詰めができる場合があります。中心座標を何万パターンも試してみると、 $R = 0.0891$ の敷き詰め方が見つかりました。



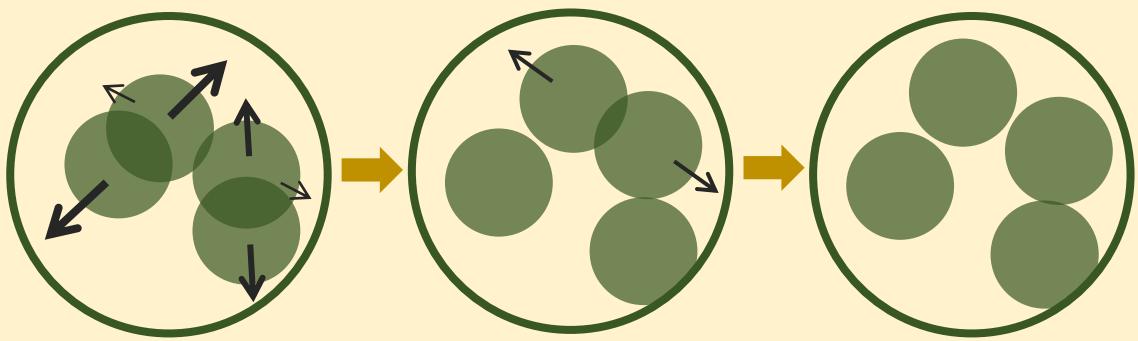
解法 4 — 勾配降下法を使う ($R = 0.0899$)

実は、勾配降下法（→コラム 5）を使うことで、もっと良い解を出すことができます。

解法のアイデア

最初、目標とする半径 R を固定します。（例えば $R = 0.0895$ など）

その後、ランダムに中心座標を決めて、 $N = 100$ 個の円を描きます。いくつかの円は重なりますが、「円を少しずらす」ことによって円の重なりをどんどん減らしていき、最終的にどの円も重ならないようにするのが目標です。



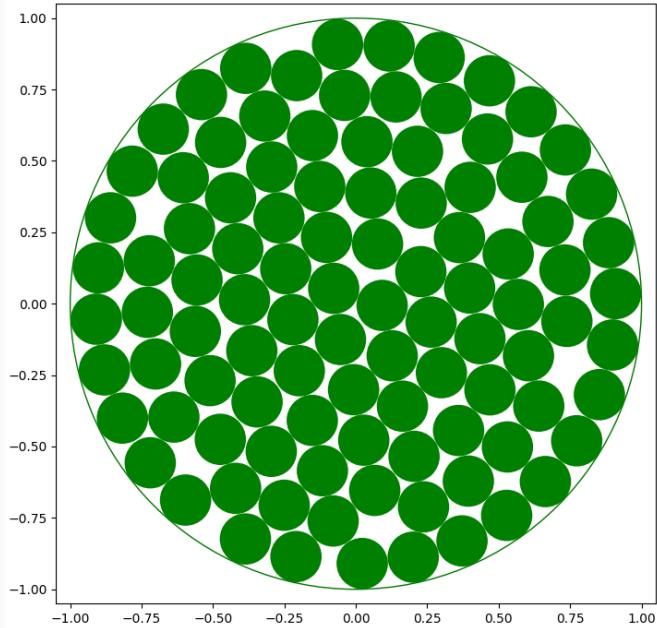
これは勾配降下法を用いて実装できます。例えば、ペナルティーを

$$P = \sum_{i=1}^n \max \left(\left((1-R) - \sqrt{x_i^2 + y_i^2} \right)^2 - R^2, 0 \right) + \sum_{1 \leq i < j \leq n} \max \left((x_i - x_j)^2 + (y_i - y_j)^2 - R^2, 0 \right)$$

などに設定して、 P ができるだけ小さくなる方向に $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ を勾配降下法で移動させていきます。すると、最終的には P が局所的最適解になります。 $P = 0$ になれば、目標達成です。

もちろん、1回の試行では $P = 0$ の解が見つからない可能性があるので、勾配降下法のステップをある程度繰り返したところで打ち切り、リセットして $N = 100$ 個のランダムな円の配置からやり直します。目標とする半径 R の値によっては、この試行を何十回・何百回と繰り返して、やっと $P = 0$ の解にたどり着く場合もあります。

実際にやってみたところ、 $R = 0.0895$ 程度なら数回の試行で $P = 0$ の解が見つかりました。一方、 R が大きくなると解は見つかりにくくなり、 $R = 0.0899$ では C++ のプログラムを 40 分程度回し、勾配降下法の試行を 3000 回ほど行ったところで、解が見つかりました。見つかった解は、次ページに載せています。



さらに効率的な敷き詰め方

この問題はよく研究されており、2021年12月現在、 $R = 0.0902352$ の解が見つかっています。詳しく知りたい方は、以下の論文をお読みください。

- Grosso, A., Jamali, A. R. M. J. U., Locatelli, M., & Schoen, F. (2010). Solving the problem of packing equal and unequal circles in a circular container. *Journal of Global Optimization*, 47(1), 63-81.

しかしながら、現時点での最も良い解は2008年に発見されたものです。この問題は、最適解がまだ知らない未解決問題であり、将来この解がさらに改善される可能性も十分あります。

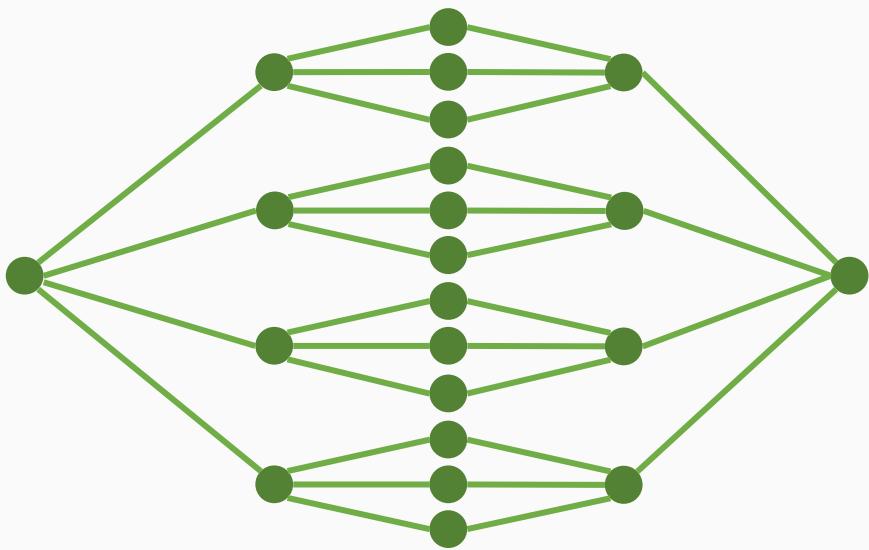
興味を持たれた方は、この未解決問題にぜひ挑戦してみましょう。

問題 30

次数 d 以下、直径 k 以下のできるだけ頂点数が多いグラフを作る問題は「次数直径問題」といい、よく研究されています。本問題は、 $d = 4, k = 4$ の場合です。できるだけ頂点数が多くなるほど得点が上がる形式であり、様々なグラフの作り方が考えられます。

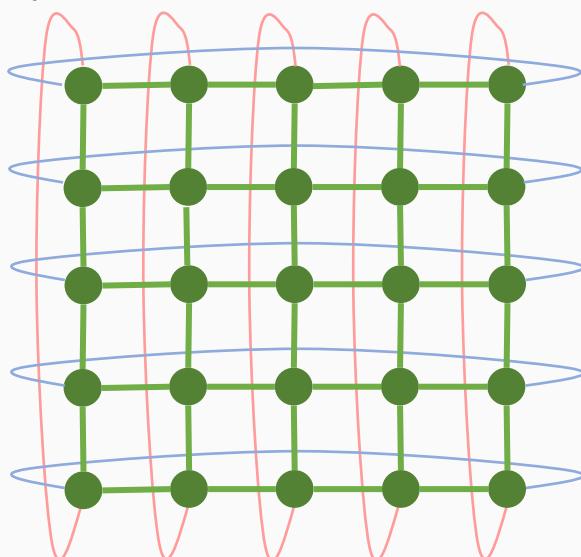
解法 0 — 問題文中の例（22 頂点）

これは、本の問題文にも例示されている、22 頂点のグラフです。



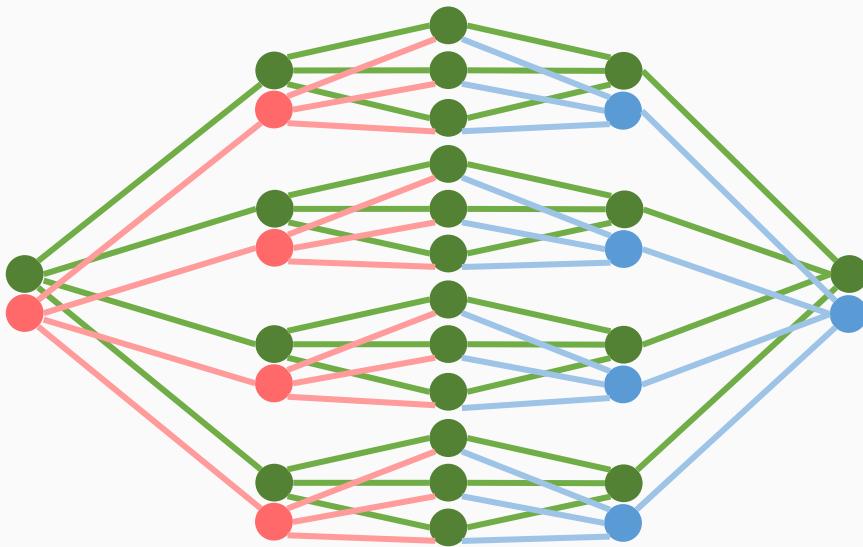
解法 1 — 5×5 のマス目のようなグラフ（25 頂点）

5×5 のマス目のようなグラフを作ると、左上から右下までの最短経路長が 8 になりますが、上と下の頂点、左と右の頂点をつなぐことによって、どの 2 頂点間の最短経路も 4 以下にできます。



解法 2 — 解法 0 [問題文中の例] を工夫する (32 頂点)

先ほどの解法 0 は、真ん中の 12 頂点の次数が 2 であるという欠点があります。この 12 頂点をもとにグラフを以下のように拡張することができます。



それ以外にも、解法 0・1・2 で挙げたような「規則的な」グラフの作り方はたくさんあります。しかし、規則的なグラフを手で作るのにも限界があり、特に 40 頂点以上のグラフを見つけるのは難しいです。しかし、アルゴリズムの力で、70~80 頂点程度のグラフを作ることもできてしまいます。

解法 3 — 山登り法を使ったアルゴリズム (73 頂点)

山登り法は、**コラム 5** でも言及されていますが、解を少しずつ改善していくことでできるだけ良い解を得るアルゴリズムです。山登り法はシンプルながらも非常に効果的なアルゴリズムであり、この問題にも適用することができます。

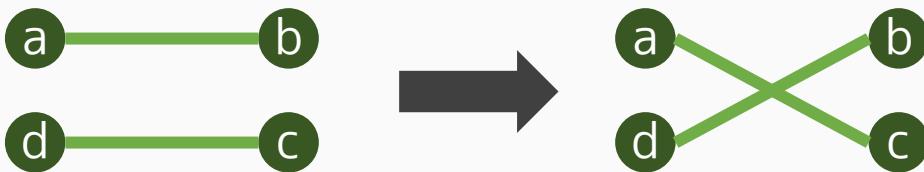
山登り法の方針は、以下の通りです。

1. 頂点数 N を決め、 N 頂点の全ての次数が 4 のグラフ G をランダムに生成する。
2. 以下の操作をたくさん繰り返す
 - グラフの 2 辺をランダムに選び、これをつなぎ変える操作を行う
 - つなぎ変えたことで「最短距離が 5 以上の頂点の組の個数」が増加した場合のみ、つなぎえたグラフを元に戻す

3. 最終的に「最短距離が 5 以上の頂点の組の個数」が 0 になれば、目的のグラフが作れた

ただし、辺 $a - b$ と辺 $c - d$ をつなぎ変える操作とは、以下の操作のことを指します。

- 辺 $a - b$ と辺 $c - d$ を削除し、新たに辺 $a - c$ と辺 $b - d$ を追加する。
- ただし、現在のグラフに辺 $a - c$ または辺 $b - d$ が存在しない場合のみ、つなぎ変える操作ができる。



この操作を行っても、各頂点の次数は 4 で変わらず、グラフを「少しづつ改変」できるため、山登り法にとって都合のよい変化の方法になります。

実際にやってみると、 $N = 73$ にセットしても、山登り法でどんどんグラフが改善されていき、数秒で条件を満たすグラフが求まります。このグラフは規則性がなく複雑であるため、ここには掲載しないことにします。

また、山登り法を改良した「焼きなまし法」を使ったプログラムを 10 分ほど実行したところ、 $N = 79$ 頂点のグラフも求まりました。

さらに頂点数の多いグラフ

2021 年 12 月現在、次数が 4 以下で直径（2 頂点間の最短経路長の最大値）が 4 以下のグラフとして、98 頂点のものが発見されていますが、これが最適であるかは分かつておらず、さらに頂点数の多いグラフが発見される可能性もあります。詳しく知りたい方は、以下のウェブサイトをご覧ください。

- COMBINATORICS WIKI, The Degree Diameter Problem for General Graphs

(次数 d , 直径 k) = (4, 4) の場合だけでなく、他の多くの (d, k) に対しても、改善の余地がまだ残されているどころか、現時点で最適性が知られているのは、自明なものを除き $(d, k) = (3, 2), (4, 2), (5, 2), (6, 2), (7, 2), (3, 3), (4, 3)$ の 7 種類しかありません。

興味を持たれた方は、この未解決問題にぜひ挑戦してみましょう。

第7章

おわりに

全 184 ページにわたる解説もこれで終わりです。分かりづらかった部分などもあったのではないかと思いますが、最後までお読みいただきありがとうございました。

また、演習問題の分量はかなり多く、全部こなすのは相当大変な労力を要したのではないかでしょうか。実際、節末問題と最終確認問題を合わせたら 148 問あり、一問当たり 30 分でも 70 時間以上かかります。しかし、これらの問題で少しでも力をつけいただき、読者の役に立てれば本当に嬉しいです。

本当に最後になりますが、解説作成にあたっては米田寛峻さん (square1001) に協力していただきました。全体の 2 割ほどが彼によって執筆されました。大変感謝しております。

2021 年 12 月 29 日

米田 優峻

問題解決のための
「アルゴリズム × 数学」が
基礎からしっかり身につく本 解答・解説

2021 年 12 月 29 日 バージョン 1 作成

作成者 米田 優峻

協力 米田 寛峻

発行 GitHub