

# 競技プログラミングの鉄則

アルゴリズムと思考力を  
高める 77 の技術

# 解答・解説

216 PAGES

## 第 1 章 アルゴリズムと計算量

応用問題 1.1 の解答	8
応用問題 1.2 の解答	9
応用問題 1.3 の解答	10
応用問題 1.4 の解答	12

## 第 2 章 累積和

応用問題 2.1 の解答	15
応用問題 2.2 の解答	17
応用問題 2.3 の解答	20
応用問題 2.4 の解答	22

## 第 3 章 二分探索

応用問題 3.1 の解答	26
応用問題 3.2 の解答	27
応用問題 3.3 の解答	29
応用問題 3.4 の解答	31

## 第4章 動的計画法

応用問題 4.1 の解答	35
応用問題 4.2 の解答	37
応用問題 4.3 の解答	39
応用問題 4.4 の解答	41
応用問題 4.5 の解答	44
応用問題 4.6 の解答	47
応用問題 4.7 の解答	50
応用問題 4.8 の解答	51
応用問題 4.9 の解答	54

## 第5章 数学的問題

応用問題 5.1 の解答	57
応用問題 5.2 の解答	58
応用問題 5.3 の解答	59
応用問題 5.4 の解答	60
応用問題 5.5 の解答	61
応用問題 5.6 の解答	63
応用問題 5.7 の解答	65
応用問題 5.8 の解答	67
応用問題 5.9 の解答	69

## 第 6 章 考察テクニック

応用問題 6.1 の解答	72
応用問題 6.2 の解答	74
応用問題 6.3 の解答	78
応用問題 6.4 の解答	81
応用問題 6.5 の解答	84
応用問題 6.6 の解答	86
応用問題 6.7 の解答	88
応用問題 6.8 の解答	91
応用問題 6.9 の解答	92
応用問題 6.10 の解答	94

## 第 7 章 ヒューリスティック

応用問題なし

## 第 8 章 データ構造とクエリ処理

応用問題 8.1 の解答	96
応用問題 8.2 の解答	99
応用問題 8.3 の解答	102
応用問題 8.4 の解答	105
応用問題 8.5 の解答	106
応用問題 8.6 の解答	108
応用問題 8.7 の解答	111
応用問題 8.8 の解答	113
応用問題 8.9 の解答	116

## 第 9 章 グラフアルゴリズム

応用問題 9.1 の解答	120
応用問題 9.2 の解答	122
応用問題 9.3 の解答	125
応用問題 9.4 の解答	128
応用問題 9.5 の解答	131
応用問題 9.6 の解答	135
応用問題 9.7 の解答	138
応用問題 9.8 の解答	140
応用問題 9.9 の解答	146

## 第 10 章 総合問題

力試し問題 1 の解答	150
力試し問題 2 の解答	151
力試し問題 3 の解答	152
力試し問題 4 の解答	154
力試し問題 5 の解答	156
力試し問題 6 の解答	158
力試し問題 7 の解答	160
力試し問題 8 の解答	162
力試し問題 9 の解答	164
力試し問題 10 の解答	166
力試し問題 11 の解答	170
力試し問題 12 の解答	174
力試し問題 13 の解答	177
力試し問題 14 の解答	181
力試し問題 15 の解答	184
力試し問題 16 の解答	189
力試し問題 17 の解答	194
力試し問題 18 の解答	197
力試し問題 19 の解答	200
力試し問題 20 の解答	203

# 第1章

## アルゴリズムと計算量

応用問題 1.1	8
応用問題 1.2	9
応用問題 1.3	10
応用問題 1.4	12

# 1.1

## 問題 B01 : A+B Problem (難易度：★1相当)

この問題では、2つの整数  $A$  と  $B$  を入力し、 $A + B$  の値を出力しなければなりません。

C++ の場合は `cin` を使って入力を行い、`cout` を使って出力を行うことができる、以下のようなプログラムを書くと正解が得られます。

### ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     // 入力
6     int A, B;
7     cin >> A >> B;
8
9     // 出力
10    cout << A + B << endl;
11    return 0;
12 }
```

※Python のコードはサポートページをご覧ください

# 1.2

## 問題 B02 : Divisor Check (難易度：★1相当)

この問題では、以下のように **A 以上 B 以下の整数を全探索する**ことで、正しい答えを求めることができます。

- $A$  は 100 の約数か？
- $A + 1$  は 100 の約数か？
- $A + 2$  は 100 の約数か？
- $\vdots$
- $B$  は 100 の約数か？

どれか 1 つでも Yes なら  
答えは Yes

全探索のアルゴリズムを C++ で実装すると、以下のようになります。ここで整数  $x$  が 100 の約数であるかどうかは、 $100 \% x == 0$  かどうかで判定できることに注意してください。

### ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     // 入力
6     int A, B;
7     cin >> A >> B;
8
9     // 答えを求める
10    bool Answer = false;
11    for (int i = A; i <= B; i++) {
12        if (100 % i == 0) Answer = true;
13    }
14
15    // 出力
16    if (Answer == true) cout << "Yes" << endl;
17    else cout << "No" << endl;
18    return 0;
19 }
```

※Python のコードはサポートページをご覧ください

## 1.3

# 問題 B03 : Supermarket 1

(難易度：★2相当)

この問題を全探索で解くプログラムは、三重の for 文を使って以下のように実装することができます。各変数は次のようにになっています。

変数名	説明
i	選んだ商品のうち 1 個目の番号（つまり値段は $A_i$ 円）
j	選んだ商品のうち 2 個目の番号（つまり値段は $A_j$ 円）
k	選んだ商品のうち 3 個目の番号（つまり値段は $A_k$ 円）

そして、各  $(i, j, k)$  に対しては合計価格が 1000 円になっているか、つまり  $A_i + A_j + A_k = 1000$  を満たすかどうかを調べています。なお、選ぶ商品はすべて異なる必要があるので、 $i, j, k$  が相異なるかどうかも条件分岐でチェックしています。

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     // 入力
6     int N, A[109];
7     cin >> N;
8     for (int i = 1; i <= N; i++) cin >> A[i];
9
10    // 答えを求める
11    bool Answer = false;
12    for (int i = 1; i <= N; i++) {
13        for (int j = 1; j <= N; j++) {
14            for (int k = 1; k <= N; k++) {
15                if (A[i] + A[j] + A[k] == 1000) { // 合計価格は 1000 円か？
16                    if (i!=j && j!=k && i!=k) { // 商品はすべて異なるか？
17                        Answer = true;
18                    }
19                }
20            }
21        }
22    }
23
24    // 出力
25    if (Answer == true) cout << "Yes" << endl;
26    else cout << "No" << endl;
27    return 0;
28 }
```

しかし、商品番号の小さい順に変数  $i, j, k$  を割り当てるようにする<sup>※1</sup>と、3つの値  $i, j, k$  がすべて異なるかどうかの判定をする必要がなくなり、実装が簡潔になります。具体的には次のようなループを行えば良いです。

- $j$  は  $i + 1$  以上  $N$  以下の範囲でループ
- $k$  は  $j + 1$  以上  $N$  以下の範囲でループ

全探索のアルゴリズムを C++ で実装すると以下のようになり、計算量は  $O(N^3)$  です。制約は  $N \leq 100$  ですので、実行時間制限の 1 秒には余裕を持つ間に合います。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     // 入力
6     int N, A[109];
7     cin >> N;
8     for (int i = 1; i <= N; i++) cin >> A[i];
9
10    // 答えを求める
11    bool Answer = false;
12    for (int i = 1; i <= N; i++) {
13        for (int j = i + 1; j <= N; j++) {
14            for (int k = j + 1; k <= N; k++) {
15                if (A[i] + A[j] + A[k] == 1000) Answer = true;
16            }
17        }
18    }
19
20    // 出力
21    if (Answer == true) cout << "Yes" << endl;
22    else cout << "No" << endl;
23    return 0;
24 }
```

※Python のコードはサポートページをご覧ください

※1 つまり、たとえば  $(i, j, k) = (3, 2, 1)$  などは探索しないということです

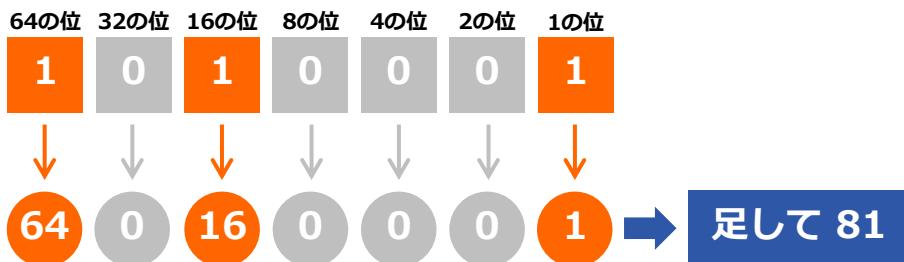
# 1.4

## 問題 B04 : Binary Representation 2 (難易度：★2相当)

本の 36 ページ (1.4 節) にも記した通り、以下のような方法で 2 進法を 10 進法に変換することができます。

2 進法の下の位から順に「1 の位」「2 の位」「4 の位」「8 の位」と倍々になるように付けていく。このとき、「数字×位」の総和が 10 進法に変換した値である。

たとえば、2 進法の「1010001」を 10 進法に変換すると、 $64+16+1=81$  となります。イメージ図を以下に示します。



このアルゴリズムを実装すると、以下の解答例のようになります。このプログラムでは、入力を文字列  $N$  として受け取っており、C++ の文字列は 0 文字目から始まるため、「数字×位」としては

```
N[i] * (1 << (N.size() - 1 - i))
```

を足しています。ここで  $N[i]$  は文字列の  $i$  文字目を表し、 $(1 << x)$  は  $2^x$  の意味します。

### ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
```

```
4 int main() {
5     // 入力
6     string N;
7     cin >> N;
8
9     // 答えを求める
10    int Answer = 0;
11    for (int i = 0; i < N.size(); i++) {
12        int keta;
13        int kurai = (1 << (N.size() - 1 - i));
14        if (N[i] == '0') keta = 0;
15        if (N[i] == '1') keta = 1;
16        Answer += keta * kurai;
17    }
18
19    // 出力
20    cout << Answer << endl;
21    return 0;
22 }
```

※Python のコードはサポートページをご覧ください

# 第2章

## 累積和

應用問題 2.1	· · · · ·	15
應用問題 2.2	· · · · ·	17
應用問題 2.3	· · · · ·	20
應用問題 2.4	· · · · ·	22

## 2.1

### 問題 B06 : Lottery

(難易度：★2相当)

この問題を解く最も単純な方法は全探索です。 $L$  回目から  $R$  回目までのアタリの数・ハズレの数を for 文で数えると、計算量  $O(R - L)$  で「アタリとハズレどちらが大きいか」という質問に答えることができます。

しかし、本問題の制約は  $N, Q \leq 100000$  であるため、残念ながら実行時間制限に間に合いません。一体どうすれば良いのでしょうか。

#### ◆ 累積和を使おう

まず、アタリの数・ハズレの数それぞれについて累積和をとることを考えます。具体的には、1回目から  $i$  回目までのアタリの数  $\text{Atari}[i]$ 、1回目から  $i$  回目までのハズレの数  $\text{Hazre}[i]$  を前もって計算します。入力例 1 のときの配列の値を以下に示します。

i	0	1	2	3	4	5	6	7
$\text{Atari}[i]$	0	0	1	2	2	3	3	3
$\text{Hazre}[i]$	0	1	1	2	2	3	3	4
結果		✗	○	○	✗	○	✗	✗

すると、 $L$  回目から  $R$  回目までのアタリの数・ハズレの数は、それぞれ以下のようにして計算することができます。

- アタリの数： $\text{Atari}[R] - \text{Atari}[L-1]$
- ハズレの数： $\text{Hazre}[R] - \text{Hazre}[L-1]$

したがって、次ページの解答例のような実装をすると、高速に答えを出すことができます。計算量は  $O(N + Q)$  です。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int N, A[100009];
5 int Q, L[100009], R[100009];
6 int Atari[100009], Hazre[100009];
7
8 int main() {
9     // 入力
10    cin >> N;
11    for (int i = 1; i <= N; i++) cin >> A[i];
12    cin >> Q;
13    for (int i = 1; i <= Q; i++) cin >> L[i] >> R[i];
14
15    // アタリの個数・ハズレの個数の累積和を求める
16    Atari[0] = 0;
17    Hazre[0] = 0;
18    for (int i = 1; i <= N; i++) {
19        Atari[i] = Atari[i - 1]; if (A[i] == 1) Atari[i] += 1;
20        Hazre[i] = Hazre[i - 1]; if (A[i] == 0) Hazre[i] += 1;
21    }
22
23    // 質問に答える
24    for (int i = 1; i <= Q; i++) {
25        int NumAtari = Atari[R[i]] - Atari[L[i] - 1];
26        int NumHazre = Hazre[R[i]] - Hazre[L[i] - 1];
27        if (NumAtari > NumHazre) cout << "win" << endl;
28        else if (NumAtari == NumHazre) cout << "draw" << endl;
29        else cout << "lose" << endl;
30    }
31    return 0;
32 }
```

※Python のコードはサポートページをご覧ください

## 2.2

# 問題 B07 : Convenience Store 2 (難易度 : ★3相当)

この問題では、 $t = 0, 1, \dots, T - 1$  について「時刻  $t + 0.5$  には何人働いているのか  $\text{Answer}[t]$ 」を求める必要があります。

それでは、 $\text{Answer}[t]$  の値はどうやって計算すれば良いのでしょうか。時刻  $L$  から時刻  $R$  まで働く人については、 $t = L, L + 1, \dots, R - 1$  について「時刻  $t + 0.5$  の労働者数」を 1 だけ増やすので、以下のプログラムによって正しく計算することができます。

```
1 // 入力
2 cin >> T >> N;
3 for (int i = 1; i <= N; i++) cin >> L[i] >> R[i];
4
5 // 答えを求める
6 for (int i = 0; i < T; i++) Answer[i] = 0;
7 for (int i = 1; i <= N; i++) {
8     for (int j = L[i]; j < R[i]; j++) Answer[j] += 1;
9 }
10
11 // 出力
12 for (int d = 0; d < T; d++) cout << Answer[d] << endl;
```

しかし、このプログラムの計算量は  $O(NT)$  です。本問題の制約は  $N, T \leq 10^5$  であるため、残念ながら実行時間制限に間に合いません。

## ◆ 差分を計算しよう

そこで、各時刻の労働者数  $\text{Answer}[t]$  の代わりに、**労働者数の前の時刻との差分  $B[t]$**  を計算することを考えます。時刻  $L$  から時刻  $R$  まで働く人については、 $B[L]$  に +1 して  $B[R]$  に -1 すれば良いです。

t	0	1	2	3	4	5	6	7
(労働者数)			+1	+1	+1	+1		
差分 B[t]			+1				-1	



すると、 $B[t]$  の累積和が求めるべき答え  $Answer[t]$  になります。たとえば  $N = 2, T = 10, (L, R) = (2, 6), (3, 9)$  の場合は下図のように計算できます。

t	0	1	2	3	4	5	6	7	8	9
差分 $B[t]$			+1				-1			

B[2] に +1  
 B[6] に -1  
 をする

t	0	1	2	3	4	5	6	7	8	9
差分 $B[t]$			+1	+1			-1			-1

B[3] に +1  
 B[9] に -1  
 をする

t	0	1	2	3	4	5	6	7	8	9
差分 $B[t]$			+1	+1			-1			-1
Answer[t]	0	0	1	2	2	2	1	1	1	0

累積和をとる

したがって、この問題を高速に解くプログラムは、以下の解答例のように実装することができます。計算量は  $O(N + T)$  であり、実行時間制限には余裕を持って間に合います。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 using namespace std;
3
4 int N, T;
5 int L[500009], R[500009];
6 int Answer[500009], B[500009];
7
8 int main() {
9     // 入力
10    cin >> T >> N;
11    for (int i = 1; i <= N; i++) cin >> L[i] >> R[i];
12
13    // 前日比に加算
14    for (int i = 0; i <= T; i++) B[i] = 0;

```

```
15  for (int i = 1; i <= N; i++) {
16      B[L[i]] += 1;
17      B[R[i]] -= 1;
18  }
19
20 // 累積和をとる
21 Answer[0] = B[0];
22 for (int d = 1; d <= T; d++) Answer[d] = Answer[d - 1] + B[d];
23
24 // 出力
25 for (int d = 0; d < T; d++) cout << Answer[d] << endl;
26 return 0;
27 }
```

※Python のコードはサポートページをご覧ください

## 2.3

# 問題 B08 : Counting Points (難易度 : ★4相当)

まず考えられる解法は、それぞれの点について「 $x$  座標が  $a$  以上  $c$  以下であり、 $y$  座標が  $b$  以上  $d$  以下であるかどうか」を直接調べることです。

しかしこの解法では、1 つの質問に答えるのに計算量  $O(N)$  かかってしまいます。質問の個数は  $Q$  個なので、全体の計算量は  $O(NQ)$  となり、残念ながら実行時間制限に間に合いません。

### ◆ 二次元累積和を考えよう

本問題では点の座標  $X_i, Y_i$  が 1 以上 1500 以下の整数となっているので、以下のような配列  $s[i][j]$ （大きさ約  $1500 \times 1500$ ）を用意します。

$s[i][j]$  : 座標  $(i, j)$  には何個の点が存在するか？

たとえば、点が座標  $(1, 1), (3, 4), (4, 3)$  に存在する場合、配列  $s[i][j]$  は下図左側のようになります。

そこで、質問の答えである「 $x$  座標が  $a$  以上  $c$  以下であり、 $y$  座標が  $b$  以上  $d$  以下である点の個数」は、下図右側のような長方形領域の総和となるため、**二次元累積和**を使って計算することができます。

1	2	3	4	5
1	1	0	0	0
2	0	0	0	0
3	0	0	0	1
4	0	0	1	0
5	0	0	0	0

配列  $s[i][j]$  の値

$a$ ►	$b$ ▼	$d$ ▼
1	0	0
0	0	0
0	0	1
0	0	0
0	0	0

答えはどの部分の総和か？

具体的には、配列  $S[i][j]$  の二次元累積和を  $T[i][j]$  とするとき、質問の答えは次の式で表すことができます。

$$T[c][d] + T[a-1][b-1] - T[a-1][d] - T[c][b-1]$$

したがって、以下の解答例のようなプログラムにより、計算量  $O(1)$  で各質問に答えることができます。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 // 入力で与えられる変数
5 int N, X[100009], Y[100009];
6 int Q, A[100009], B[100009], C[100009], D[100009];
7
8 // 各座標にある点の数 S[i][j]、二次元累積和 T[i][j]
9 int S[1509][1509];
10 int T[1509][1509];
11
12 int main() {
13     // 入力
14     cin >> N;
15     for (int i = 1; i <= N; i++) cin >> X[i] >> Y[i];
16     cin >> Q;
17     for (int i = 1; i <= Q; i++) cin >> A[i] >> B[i] >> C[i] >> D[i];
18
19     // 各座標にある点の数を数える
20     for (int i = 1; i <= N; i++) S[X[i]][Y[i]] += 1;
21
22     // 累積和をとる
23     for (int i = 0; i <= 1500; i++) {
24         for (int j = 0; j <= 1500; j++) T[i][j] = 0;
25     }
26     for (int i = 1; i <= 1500; i++) {
27         for (int j = 1; j <= 1500; j++) T[i][j] = T[i][j - 1] + S[i][j];
28     }
29     for (int i = 1; i <= 1500; i++) {
30         for (int j = 1; j <= 1500; j++) T[i][j] = T[i - 1][j] + T[i][j];
31     }
32
33     // 答えを求める
34     for (int i = 1; i <= Q; i++) {
35         cout << T[C[i]][D[i]] + T[A[i] - 1][B[i] - 1] - T[A[i] - 1][D[i]] -
36         T[C[i]][B[i] - 1] << endl;
37     }
38 }
```

※Python のコードはサポートページをご覧ください

# 2.4

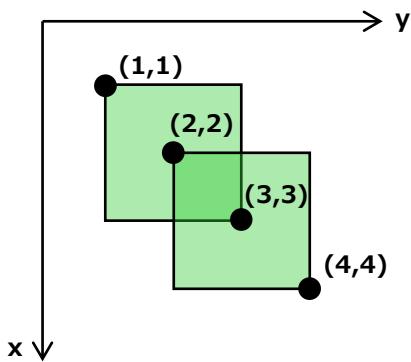
## 問題 B09 : Papers

(難易度：★4相当)

まずは以下の配列を考えます。 $T[i][j]$  が 1 以上になっている  $(i, j)$  の個数が、求めるべき答え（紙が 1 枚以上置かれている領域の面積）です。

$T[i][j]$  : 座標  $(i + 0.5, j + 0.5)$  には何枚の紙が置かれているか？

たとえば、隅の座標が  $(1, 1) \cdot (3, 3)$  である紙と、隅の座標が  $(2, 2) \cdot (4, 4)$  である紙が置かれている場合、 $T[i][j]$  の値は下図右側のようになります。

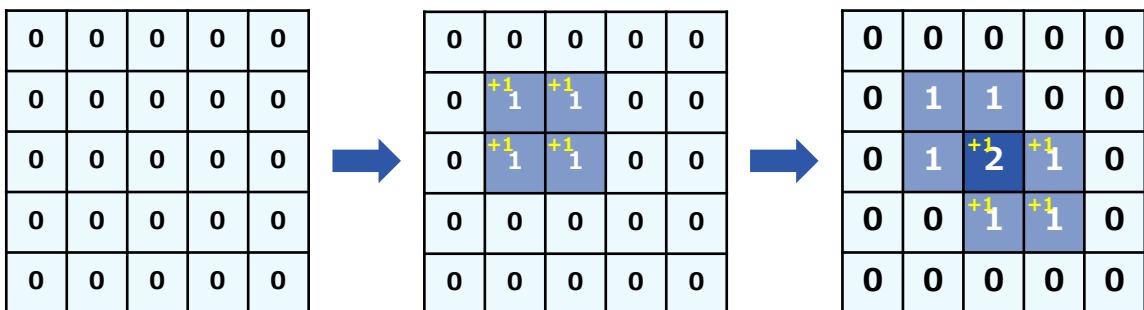


	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	0	0
2	0	1	2	1	0
3	0	0	1	1	0
4	0	0	0	0	0

配列  $T[i][j]$  の値

### ◆ $T[i][j]$ の計算

それでは、 $T[i][j]$  の値はどうやって計算すれば良いのでしょうか。もちろん、それぞれの紙について「対応する部分に +1 をする」という方法でも上手くいきますが、計算に時間がかかるてしまいます。



※1 説明の都合上、上下方向を x 軸、左右方向を y 軸にしています

そこで 67 ページ (2.4 節) で説明したように、四隅に +1/-1 する操作を行った後、最後に二次元累積和を取ると、より高速に  $T[i][j]$  の値を計算することができます。具体例を以下に示します。

0	0	0	0	0
0	+1	0	-1	0
0	0	0	0	0
0	-1	0	+1	0
0	0	0	0	0

0	0	0	0	0
0	+1	0	-1	0
0	0	+1	0	-1
0	-1	0	+1	0
0	0	-1	0	+1

0	0	0	0	0
0	+1	+1	0	0
0	+1	+2	+1	0
0	0	+1	+1	0
0	0	0	0	0

以上のアルゴリズムを実装すると、解答例のようになります。 $+1/-1$  を加算する位置が、問題 A09 と微妙に異なることに注意してください。

たとえば問題 A09 では配列の  $(c+1, d+1)$  番目に +1 をしていますが、今回は配列の  $(c, d)$  番目に +1 をしています。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 using namespace std;
3
4 // 入力で与えられる変数
5 int N;
6 int A[100009], B[100009], C[100009], D[100009];
7
8 // 座標 (i+0.5, j+0.5) に置かれている紙の数 T[i][j]
9 int T[1509][1509];
10
11 int main() {
12     // 入力
13     cin >> N;
14     for (int i = 1; i <= N; i++) cin >> A[i] >> B[i] >> C[i] >> D[i];
15
16     // 各紙について +1/-1 を加算
17     for (int i = 0; i <= 1500; i++) {
18         for (int j = 0; j <= 1500; j++) T[i][j] = 0;
19     }
20     for (int i = 1; i <= N; i++) {
21         T[A[i]][B[i]] += 1;
22         T[A[i]][D[i]] -= 1;
23         T[C[i]][B[i]] -= 1;
24         T[C[i]][D[i]] += 1;
25     }
}

```

```
26 // 二次元累積和をとる
27 for (int i = 1; i <= 1500; i++) {
28     for (int j = 1; j <= 1500; j++) T[i][j] = T[i][j - 1] + T[i][j];
29 }
30 for (int i = 1; i <= 1500; i++) {
31     for (int j = 1; j <= 1500; j++) T[i][j] = T[i - 1][j] + T[i][j];
32 }
33
34 // 面積を数える
35 int Answer = 0;
36 for (int i = 0; i <= 1500; i++) {
37     for (int j = 0; j <= 1500; j++) {
38         if (T[i][j] >= 1) Answer += 1;
39     }
40 }
41 cout << Answer << endl;
42 return 0;
43 }
```

※Python のコードはサポートページをご覧ください

# 第3章

## 二分探索

應用問題 3.1	· · · · ·	26
應用問題 3.2	· · · · ·	27
應用問題 3.3	· · · · ·	29
應用問題 3.4	· · · · ·	31

# 3.1

## 問題 B11 : Binary Search 2 (難易度：★3相当)

この問題は、以下のような方針で解くことができます。

- 手順1：配列  $A = [A_1, A_2, \dots, A_N]$  を小さい順にソートする。
- 手順2：二分探索を使って、それぞれの質問に答える。

C++ の場合、手順 1 は `sort` 関数を使って処理することができます。また、手順 2 は `lower_bound` 関数を使って処理することができます。

なお、20 行目の `lower_bound(A + 1, A + N + 1, X) - A` の値について、  
本の 86 ページには「 $A_i \geq X$  を満たす最小の  $i$  である」と書かれていますが、  
これは「配列  $A$  の中に  $X$  より小さい要素がいくつあるか」と一致することに  
注意してください。

### ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 int N, A[100009];
6 int Q, X[100009];
7
8 int main() {
9     // 入力
10    cin >> N;
11    for (int i = 1; i <= N; i++) cin >> A[i];
12    cin >> Q;
13    for (int i = 1; i <= Q; i++) cin >> X[i];
14    // 配列 X をソート
15    sort(A + 1, A + N + 1);
16
17    // 質間に答える
18    for (int i = 1; i <= Q; i++) {
19        int pos1 = lower_bound(A + 1, A + N + 1, X[i]) - A;
20        cout << pos1 - 1 << endl;
21    }
22    return 0;
23 }
```

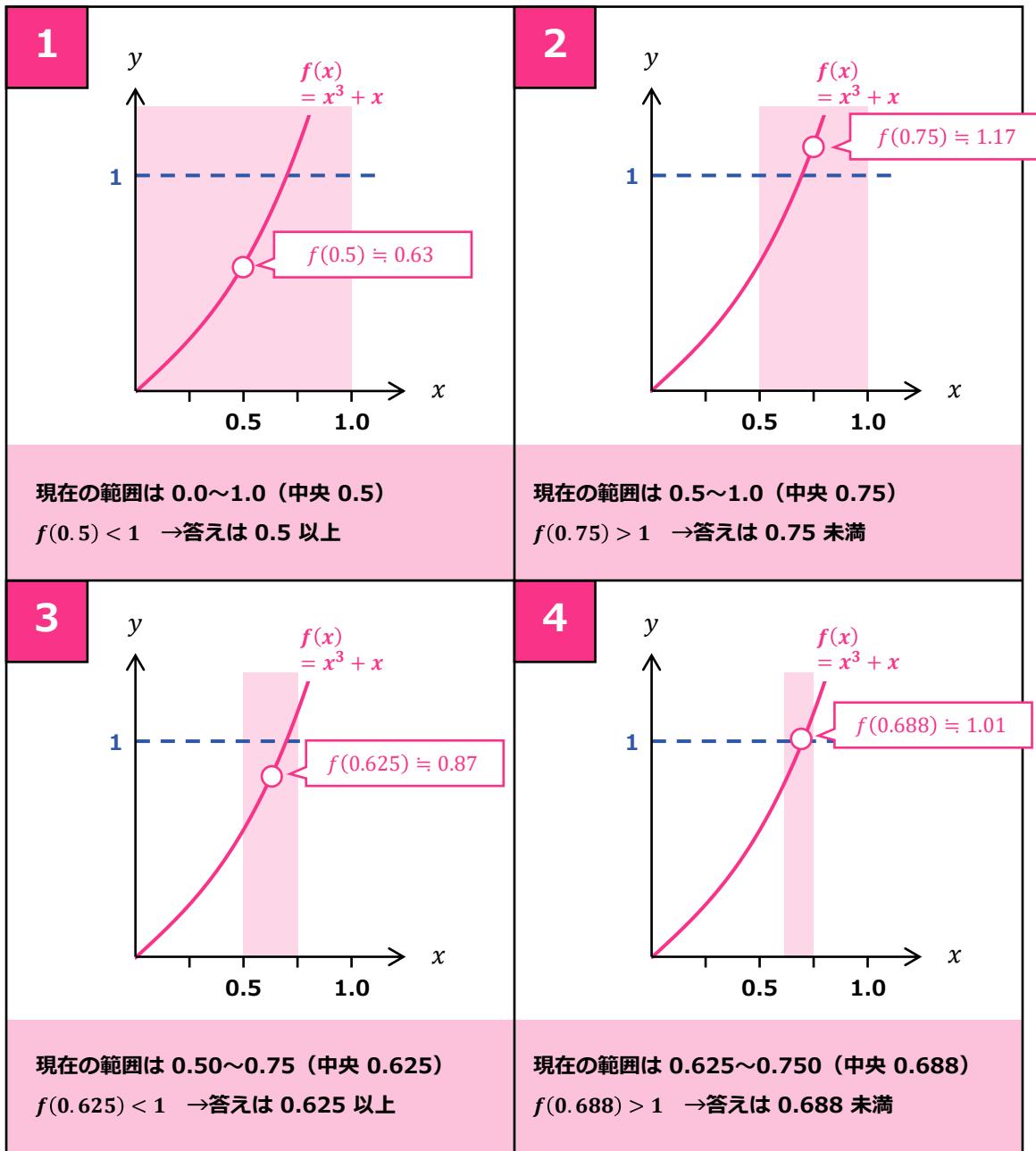
※Python のコードはサポートページをご覧ください

# 3.2

## 問題 B12 : Equation

(難易度：★4相当)

この問題は、**答えて二分探索**をして解くことができます。手始めに、 $N = 1$  のときの答えを求める考えましょう。まずは答えが 0 以上 1 以下の範囲であることが分かっているとします。



このように、たった 4 回の比較で、範囲が 0.625~0.688 まで絞られました。

それでは、本問題では何回の比較が必要なのでしょうか。まず、制約より  $N \leq 10^5$  であるため、明らかに答えは 0 以上 100 以下です ( $f(100) = 1000100$  ですので、既に  $10^5$  を超えています)。

また、答えと出力の絶対誤差が 0.001 未満であれば正解となるので、**元々 100 あつた幅を、二分探索によって 0.001 未満まで縮めなければなりません。**そこで 20 回の比較を行った場合は次のようになります。

範囲の幅は  $100 \div 2^{20} = 0.000095 \dots < 0.001$  より、十分である。

したがって、以下の解答例のように 20 回のループを行うプログラムを書くと、正解が得られます。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 // 関数 f
5 double f(double x) {
6     return x * x * x + x;
7 }
8
9 int main() {
10    // 入力
11    int N;
12    cin >> N;
13
14    // 二分探索
15    double Left = 0, Right = 100, Mid;
16    for (int i = 0; i < 20; i++) {
17        Mid = (Left + Right) / 2.0;
18        double val = f(Mid);
19
20        // 探索範囲を絞る
21        if (val > 1.0 * N) Right = Mid; // 左半分に絞られる
22        else Left = Mid; // 右半分に絞られる
23    }
24
25    // 出力
26    printf("%.12lf\n", Mid);
27    return 0;
28 }
```

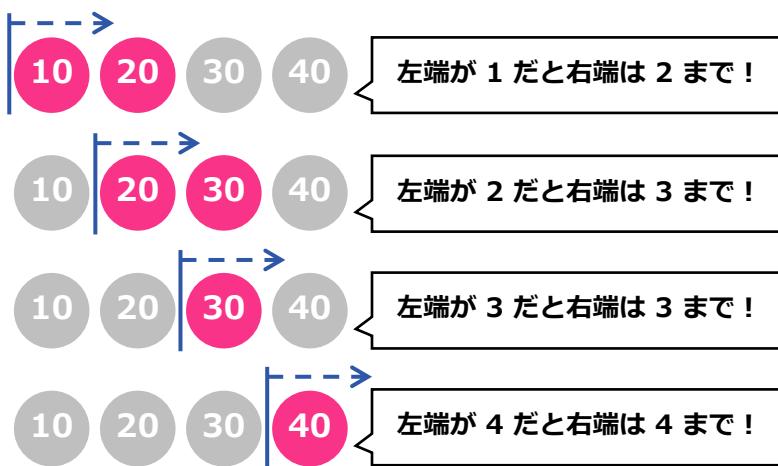
※Python のコードはサポートページをご覧ください

### 3.3

## 問題 B13 : Supermarket 2

(難易度：★4相当)

まず、「選ぶ品物の左端の番号を  $i$  とするとき、右端の番号はどこまで許されるか」を  $R_i$  とします。たとえば  $A = [10, 20, 30, 40], K = 50$  の場合、 $R_1 = 2$ 、 $R_2 = 3$ 、 $R_3 = 3$ 、 $R_4 = 4$  です。



このとき、本問題の答え（何通りの選び方があるか）は次式で表されます。

$$\frac{(R_1 - 1 + 1)}{\text{左端が } 1 \text{ のときのパターン数}} + \frac{(R_2 - 2 + 1)}{\text{左端が } 2 \text{ のときのパターン数}} + \cdots + \frac{(R_N - N + 1)}{\text{左端が } N \text{ のときのパターン数}}$$

### ◆ しゃくとり法で $R_i$ を求める

さて、本問題では明らかに  $R_1 \leq R_2 \leq \cdots \leq R_N$  を満たすため、 $R_i$  の値は以下のしゃくとり法（→本の 93 ページ）によって求めることができます。

- $R_i = R_{i-1}$  からスタート ( $i = 1$  の場合は  $R_i = 1$  から)
- 合計が  $K$  を超えないギリギリまで、 $R_i$  を 1 ずつ増やしていく

問題は、「これ以上  $R_i$  を増やすと合計価格が  $K$  円を超えるかどうか」を高速に判定することなのですが、これは 2 章で学んだ累積和を使えば良いです。

累積和  $A_1 + A_2 + \cdots + A_i$  の値を  $S_i$  とするとき、 $L$  番目から  $R$  番目の品物の合計価格は  $S_R - S_{L-1}$  となり、この値は計算量  $O(1)$  で求められます。

したがって、以下の解答例のような実装をすると、計算量  $O(N)$  で答えを求めるすることができます。なお、しゃくとり法では、「 $R_i$  を 1 増やす操作」が合計約  $N$  回しか行われないことに注意してください。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 long long N, K;
5 long long A[100009];
6 long long S[100009]; // 累積和
7 long long R[100009]; // 左端が決まったとき、右端はどこまで行けるか
8
9 // A[1] から A[r] までの合計値
10 long long sum(int l, int r) {
11     return S[r] - S[l - 1];
12 }
13
14 int main() {
15     // 入力
16     cin >> N >> K;
17     for (int i = 1; i <= N; i++) cin >> A[i];
18
19     // 累積和をとる
20     S[0] = 0;
21     for (int i = 1; i <= N; i++) S[i] = S[i - 1] + A[i];
22
23     // しゃくとり法
24     for (int i = 1; i <= N; i++) {
25         if (i == 1) R[i] = 0;
26         else R[i] = R[i - 1];
27         while (R[i] < N && sum(i, R[i] + 1) <= K) {
28             R[i] += 1;
29         }
30     }
31
32     // 答えを求める
33     long long Answer = 0;
34     for (int i = 1; i <= N; i++) Answer += (R[i] - i + 1);
35     cout << Answer << endl;
36     return 0;
37 }
```

※Python のコードはサポートページをご覧ください

## 3.4

# 問題 B14 : Another Subset Sum (難易度：★5相当)

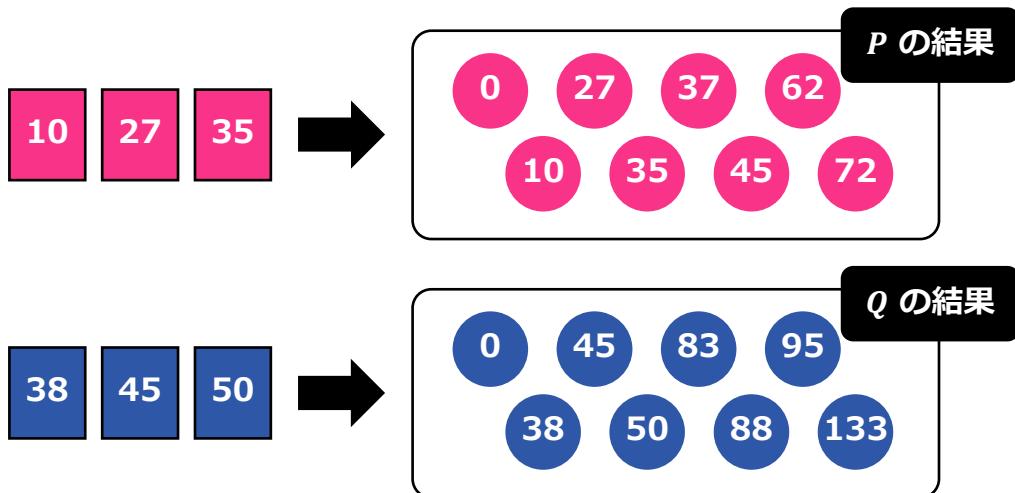
この問題は、半分全列挙を使った以下のような方針で解くことができます。

- 手順1：「前半  $N/2$  枚の中から何枚か選んだときの、カードに書かれた整数の総和として何があり得るか？」を全探索で求める
- 手順2：「後半  $N/2$  枚の中から何枚か選んだときの、カードに書かれた整数の総和として何があり得るか？」を全探索で求める
- 手順3：手順 1・2 の結果を合成させて、答えを求める

この説明だけではよく分からぬと思うので、例として  $N = 6, K = 100, A = [10, 27, 35, 38, 45, 50]$  のケースを考えてみましょう。手順 1 の結果を  $P$ 、手順 2 の結果を  $Q$  とするとき、

- $P = [0, 10, 27, 35, 37, 45, 62, 72]$
- $Q = [0, 38, 45, 50, 83, 88, 95, 131]$

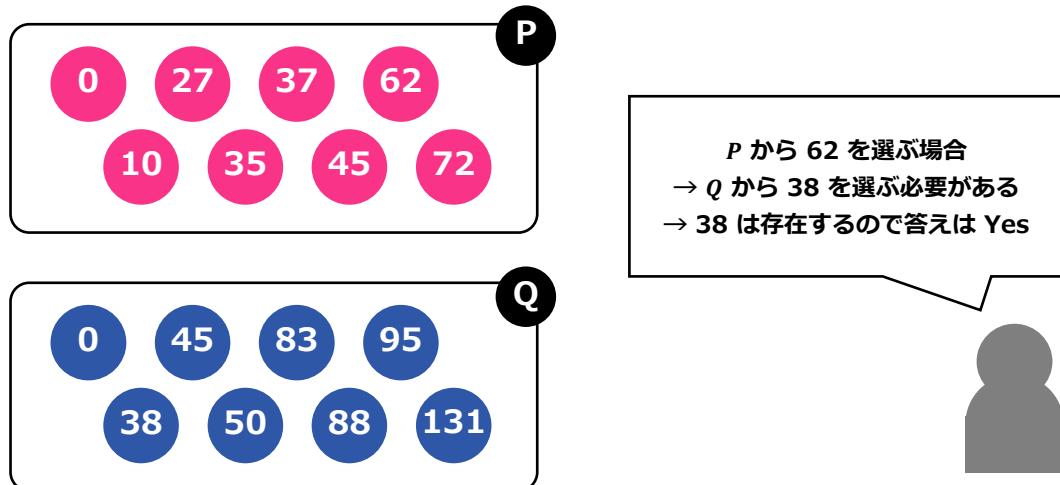
となります。イメージ図を以下に示します。



ここで、もし  $P, Q$  から合計が 100 になるように 1 つずつ取り出す方法が存在すれば、本問題の答えは Yes になるのですが、これはどうやって判定すれば良いでしょうか。もちろん  $8^2 = 64$  通りを全探索しても良いのですが、

計算に時間がかかってしまいます。そこで、「 $P$  の中からどれを選ぶか」を全探索するという方針が効率的です。

$P$  から選ばれた要素を  $x$  とするとき、 $Q$  からは  $100 - x$  を選ぶ必要があり、それが可能かどうかは配列の二分探索によって判定できるので<sup>※1</sup>、およそ  $8 \log 8$  回程度の計算しかする必要はありません（今回のケースでは、 $P$  の要素数も  $Q$  の要素数も 8 です）。



## ◆ 一般のケースでは？

ここまで  $N = 6$  の例を説明しましたが、一般的なケースでも同じように解くことができます。これを実装すると解答例のようになります。各ステップでの計算量は以下のようになります。

- 手順1：ビット全探索を使って  $O(N \times 2^{N/2})$
- 手順2：ビット全探索を使って  $O(N \times 2^{N/2})$
- 手順3： $P, Q$  の要素数は  $2^{N/2}$  なので、 $O(2^{N/2} \log 2^{N/2}) = O(N \times 2^{N/2})$

したがって、プログラム全体の計算量は  $O(N \times 2^{N/2})$  です。本問題の制約は  $N \leq 30$  であるため、 $2^{15} = 32768$  より十分余裕を持って間に合います。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
```

※1  $Q$  が小さい順にソートされている必要があることに注意してください

```

5 // 「配列 A にあるカードからいくつか選んだときの合計」として考えられるものを列挙
6 // ビット全探索を使う
7 vector<long long> Enumerate(vector<long long> A) {
8     vector<long long> SumList;
9     for (int i = 0; i < (1 << A.size()); i++) {
10         long long sum = 0; // 現在の合計値
11         for (int j = 0; j < A.size(); j++) {
12             int wari = (1 << j);
13             if ((i / wari) % 2 == 1) sum += A[j];
14         }
15         SumList.push_back(sum);
16     }
17     return SumList;
18 }
19
20 long long N, K;
21 long long A[39];
22
23 int main() {
24     // 入力
25     cin >> N >> K;
26     for (int i = 1; i <= N; i++) cin >> A[i];
27
28     // カードを半分ずつに分ける
29     vector<long long> L1, L2;
30     for (int i = 1; i <= N / 2; i++) L1.push_back(A[i]);
31     for (int i = N / 2 + 1; i <= N; i++) L2.push_back(A[i]);
32
33     // それぞれについて、「あり得るカードの合計」を全列挙
34     vector<long long> Sum1 = Enumerate(L1);
35     vector<long long> Sum2 = Enumerate(L2);
36     sort(Sum1.begin(), Sum1.end());
37     sort(Sum2.begin(), Sum2.end());
38
39     // 二分探索で  $\text{Sum1}[i] + \text{Sum2}[j] = K$  となるものが存在するかを見つける
40     for (int i = 0; i < Sum1.size(); i++) {
41         int pos = lower_bound(Sum2.begin(), Sum2.end(), K - Sum1[i]) - Sum2.begin();
42         if (pos < Sum2.size() && Sum2[pos] == K - Sum1[i]) {
43             cout << "Yes" << endl;
44             return 0;
45         }
46     }
47     cout << "No" << endl;
48     return 0;
49 }
```

※Python のコードはサポートページをご覧ください

# 第4章

## 動的計画法

應用問題 4.1	· · · · ·	35
應用問題 4.2	· · · · ·	37
應用問題 4.3	· · · · ·	39
應用問題 4.4	· · · · ·	41
應用問題 4.5	· · · · ·	44
應用問題 4.6	· · · · ·	47
應用問題 4.7	· · · · ·	50
應用問題 4.8	· · · · ·	51
應用問題 4.9	· · · · ·	54

この問題は、いきなり足場 1 から足場  $N$  までの最小コストを求めるのが難しくなってしまいます。しかし、 $i = 1, 2, \dots, N$  の順に「足場 1 から足場  $i$  までの最小コストはいくつ？」ということを考えていくと、計算量  $O(N)$  で解くことができます。

### ◆ 管理する配列

$dp[i]$  : 足場 1 から足場  $i$  まで移動するための最小コスト

### ◆ 配列 $dp$ の計算

まず、明らかに  $dp[1] = 0$  および  $dp[2] = |H_1 - H_2|$  が成り立ちます。次に  $dp[3]$  以降ですが、最後の行動で場合分けをすると、足場  $i$  に移動する方法としては以下の 2 つが考えられます：

- **方法A** : 1 回で、足場  $i - 1$  から足場  $i$  に移動する
- **方法B** : 1 回で、足場  $i - 2$  から足場  $i$  に移動する

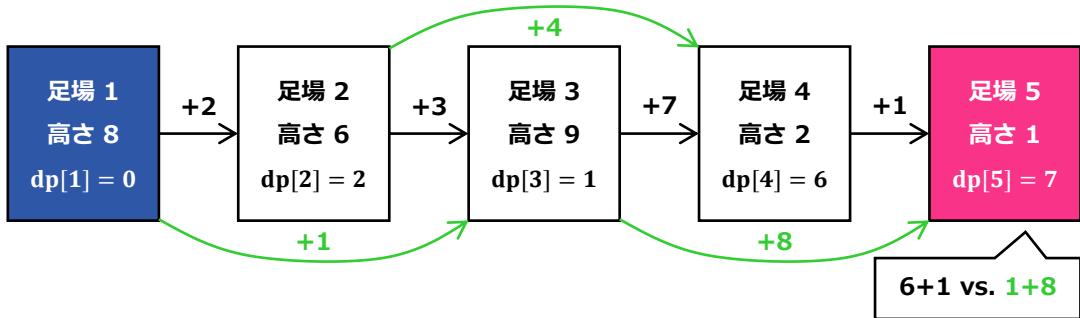
ここで、方法 A をとったときの合計コストは  $dp[i - 1] + |H_{i-1} - H_i|$ 、方法 B をとったときの合計コストは  $dp[i - 2] + |H_{i-2} - H_i|$  であるため、 $dp[3]$  以降は次の式によって計算することができます。

```
dp[i] = min(dp[i-1] + abs(H[i-1]-H[i]),  
            dp[i-2] + abs(H[i-2]-H[i]))
```

(次ページへ続きます)

## ◆ 具体例

たとえば、 $N = 5, A = [8, 6, 9, 2, 1]$  の場合、次のような計算により、答えが  $dp[5] = 7$  であると分かります。なお、矢印に書かれた整数は、1 回の移動コスト ( $|H_{i-1} - H_i|$  または  $|H_{i-2} - H_i|$ ) となっています。



## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <cmath>
3 #include <algorithm>
4 using namespace std;
5
6 int N, H[100009];
7 int dp[100009];
8
9 int main() {
10    // 入力
11    cin >> N;
12    for (int i = 1; i <= N; i++) cin >> H[i];
13
14    // 動的計画法
15    dp[1] = 0;
16    dp[2] = abs(H[1] - H[2]);
17    for (int i = 3; i <= N; i++) {
18        dp[i] = min(dp[i-1] + abs(H[i-1] - H[i]), dp[i-2] + abs(H[i-2] - H[i]));
19    }
20
21    // 出力
22    cout << dp[N] << endl;
23    return 0;
24 }
```

※Python のコードはサポートページをご覧ください

## 4.2

# 問題 B17 : Frog 1 with Restoration

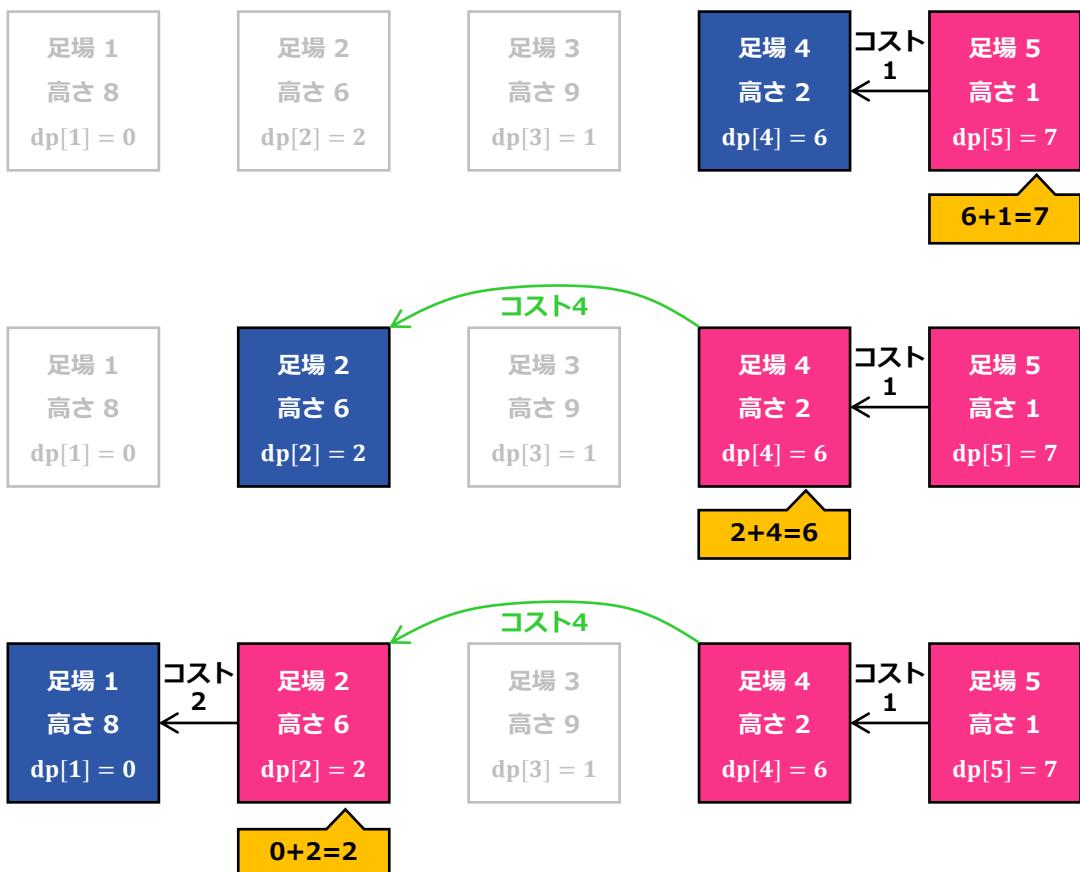
(難易度：★3相当)

$dp[i]$  の値を計算した後は、問題 A17 と同様に **ゴールから考えていくこと** で、「具体的な経路」を得ることができます。具体的には、いま部屋  $i$  にいるとき以下のようになります。

- $dp[i] = dp[i - 1] + |h_{i-1} - h_i|$  の場合：部屋  $i - 1$  に進むのが最適
- $dp[i] = dp[i - 2] + |h_{i-2} - h_i|$  の場合：部屋  $i - 2$  に進むのが最適

### ◆ 具体例

たとえば、 $N = 5, h = [8, 6, 9, 2, 1]$  の場合、次のような計算により、 $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$  という経路が最適だと分かります。



## 解答例 (C++)

```
1 #include <iostream>
2 #include <cmath>
3 #include <vector>
4 #include <algorithm>
5 using namespace std;
6
7 int N, H[100009];
8 int dp[100009];
9 vector<int> Answer;
10
11 int main() {
12     // 入力
13     cin >> N;
14     for (int i = 1; i <= N; i++) cin >> H[i];
15
16     // 動的計画法
17     dp[1] = 0;
18     dp[2] = abs(H[1] - H[2]);
19     for (int i = 3; i <= N; i++) {
20         dp[i] = min(dp[i-1] + abs(H[i-1] - H[i]), dp[i-2] + abs(H[i-2] - H[i]));
21     }
22
23     // 動的計画法の復元
24     int Place = N;
25     while (true) {
26         Answer.push_back(Place);
27         if (Place == 1) break;
28
29         // どちらに移動するかを求める
30         if (dp[Place-1] + abs(H[Place-1] - H[Place]) == dp[Place]) Place = Place-1;
31         else Place = Place - 2;
32     }
33     reverse(Answer.begin(), Answer.end());
34
35     // 答えを求める
36     cout << Answer.size() << endl;
37     for (int i = 0; i < Answer.size(); i++) {
38         if (i) cout << " ";
39         cout << Answer[i];
40     }
41     cout << endl;
42     return 0;
43 }
```

※Python のコードはサポートページをご覧ください

応用問題 4.2 と同様、「カード  $N$  を選ぶべきか?」「カード  $N - 1$  を選ぶべきか?」「カード  $N - 2$  を選ぶべきか?」という順番で答えを求めていくと、合計を  $S$  にするカードの選び方が分かります。

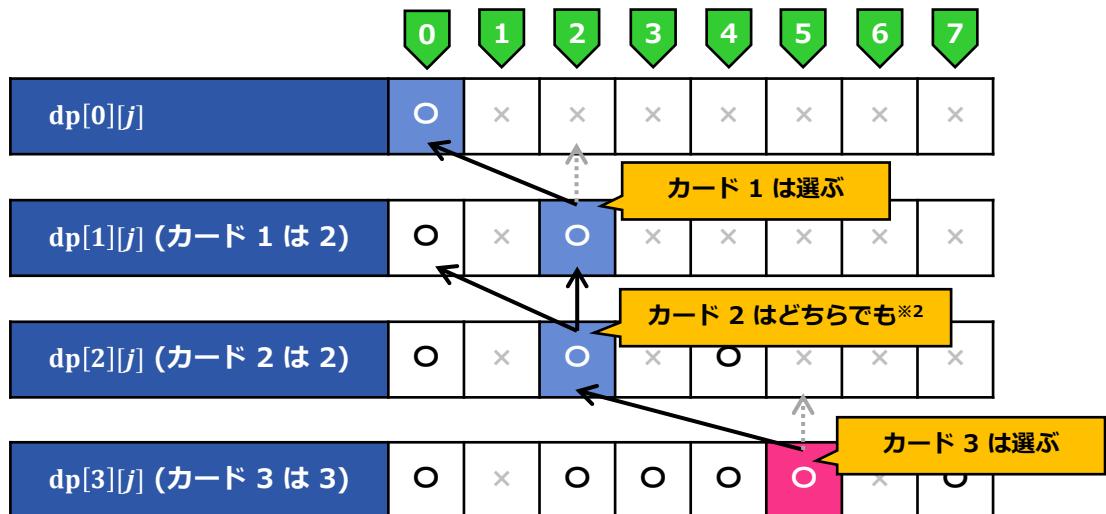
たとえばカード  $N$  を選ぶかどうかは、以下のように決めれば良いです。カード  $N - 1$  以降についても同様です。

- $\text{dp}[N][S]$  がマルの場合 : カード  $N$  を選ばない<sup>※1</sup>
- $\text{dp}[N][S - A_N]$  がマルの場合 : カード  $N$  を選ぶ

※2 つのうち、どちらか片方の条件は必ず満たす

## ◆ 具体例

たとえば、 $N = 5, S = 5, A = [2, 2, 3]$  の場合、次のようにして「カード 1 と 3 を選べば良い」ということが分かります。



※1 突然  $\text{dp}[N][S]$  などが出てきて混乱するかもしれません、動的計画法の復元を行う前に、本の 116 ページの通りに配列  $\text{dp}$  の値を計算しなければならないことに注意してください

※2 どちらでもと書いてありますが、ここでは「カード 2 を選ばない」方を選択していることに注意

## 解答例 (C++)

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5 int N, S, A[69];
6 bool dp[69][10009];
7 vector<int> Answer;
8
9 int main() {
10     // 入力
11     cin >> N >> S;
12     for (int i = 1; i <= N; i++) cin >> A[i];
13
14     // 動的計画法 (i = 0)
15     dp[0][0] = true;
16     for (int i = 1; i <= S; i++) dp[0][i] = false;
17     // 動的計画法 (i >= 1)
18     for (int i = 1; i <= N; i++) {
19         for (int j = 0; j <= S; j++) {
20             if (j < A[i]) {
21                 if (dp[i - 1][j] == true) dp[i][j] = true;
22                 else dp[i][j] = false;
23             }
24             if (j >= A[i]) {
25                 if (dp[i-1][j]==true || dp[i-1][j-A[i]]==true) dp[i][j]=true;
26                 else dp[i][j] = false;
27             }
28         }
29     }
30
31     // 選び方が存在しない場合
32     if (dp[N][S] == false) { cout << "-1" << endl; return 0; }
33
34     // 答えの復元 (Place は "現在の総和")
35     int Place = S;
36     for (int i = N; i >= 1; i--) {
37         if (dp[i - 1][Place] == true) Place = Place - 0; // カード i を選ばない
38         else {
39             Place = Place - A[i]; // カード i を選ぶ
40             Answer.push_back(i);
41         }
42     }
43     reverse(Answer.begin(), Answer.end());
44
45     // 出力
46     cout << Answer.size() << endl;
47     for (int i = 0; i < Answer.size(); i++) {
48         if (i >= 1) cout << " ";
49         cout << Answer[i];
50     }
51     cout << endl;
52     return 0;
53 }
```

※Python のコードはサポートページをご覧ください

## 4.4

# 問題 B19 : Knapsack 2 (難易度 : ★4相当)

例題で扱ったナップザック問題では、 $W$  の値が小さいため、「どの品物まで決めたか」「現在の合計重量」の 2 つを持つ動的計画法を考えました。

一方、今回は価値  $v_i$  の値が小さいので、「どの品物まで決めたか」「現在の合計価値」の 2 つを持つと、上手くいきます。

### ◆ 管理する配列

$dp[i][j]$  : 品物  $1, 2, \dots, i$  の中から、価値の合計が  $j$  となるように選ぶことを考える。このとき、合計重量としてあり得る最小値はいくつか。

※  $v_i$  の合計は高々 100000 なので、 $j \leq 100000$  まで考えれば良い

### ◆ 答えとなる値

$dp[N][i] \leq W$  を満たす最大の  $i$  が、求める答え（合計価値の最大値）となります。

### ◆ 配列 $dp$ の計算

まず、明らかに  $dp[0][0] = 0$  となります。 $dp[0][1], dp[0][2], \dots$  の値は、そもそも選び方が存在しないので  $\infty$  としておきます。

次に、 $i \geq 1$  に対する  $dp[i][j]$  の値はどうやって計算すれば良いのでしょうか。最後の行動で場合分けすると、 $dp[i][j]$  の状態になる方法として以下の 2 つがあると分かります。

選び方	合計重量の最小値
方法A : 品物 $i - 1$ 時点で合計価値 $j$ / 品物 $i$ を買わない	$dp[i - 1][j]$
方法B : 品物 $i - 1$ 時点で合計価値 $j - v_i$ / 品物 $i$ を買う	$dp[i - 1][j - v_i] + w_i$

したがって、 $dp[i][j]$  の値は次のようにして計算することができます。

$$dp[i][j] = \min(dp[i-1][j], dp[i-1][j-v[i]]+w[i])$$

## ◆ 具体例

たとえば、 $N = 5, W = 100, (w_i, v_i) = (55,2), (75,3), (40,2)$  の場合、次のようにして配列  $dp$  の値を計算することができます。 $dp[5][4] \leq 100$  なので、問題の答え（価値の最大値）は 4 です。

	0	1	2	3	4	5	6	7
$dp[0][j]$	0	$\infty$						
$dp[1][j]$ (重さ 55／価値 2)	0	$\infty$	55	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$dp[2][j]$ (重さ 75／価値 3)	0	$\infty$	55	75	$\infty$	130	$\infty$	$\infty$
$dp[3][j]$ (重さ 40／価値 2)	0	$\infty$	40	75	95	115	$\infty$	170

例 :  $\min(130, 75+40)=115$

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 long long N, W, w[109], v[109];
6 long long dp[109][100009];
7
8 int main() {
9     // 入力・配列の初期化
10    cin >> N >> W;
11    for (int i = 1; i <= N; i++) cin >> w[i] >> v[i];
12    for (int i = 0; i <= N; i++) {
13        for (int j = 0; j <= 100000; j++) dp[i][j] = 1'000'000'000'000'000LL;
14    }
}
```

```
15 // 動的計画法
16 dp[0][0] = 0;
17 for (int i = 1; i <= N; i++) {
18     for (int j = 0; j <= 100000; j++) {
19         if (j < v[i]) dp[i][j] = dp[i - 1][j];
20         else dp[i][j] = min(dp[i - 1][j], dp[i - 1][j - v[i]] + w[i]);
21     }
22 }
23
24 // 答えの出力
25 long long Answer = 0;
26 for (int i = 0; i <= 100000; i++) {
27     if (dp[N][i] <= W) Answer = i;
28 }
29 cout << Answer << endl;
30 return 0;
31 }
```

※Python のコードはサポートページをご覧ください

# 4.5

## 問題 B20 : Edit Distance (難易度 : ★6相当)

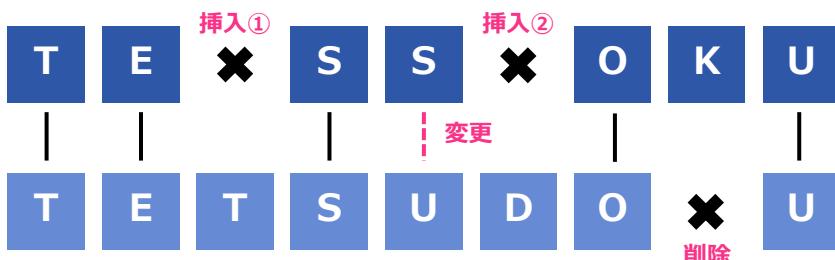
まず、文字列  $S$  を編集する操作は、**文字列  $S, T$  を一列に並べる操作**に対応します。具体的には以下のようになります。

操作1：文字列 $S$ から 1 つの文字を削除	$S$ の下に $T$ の文字を置かない
操作2：文字列 $S$ の 1 つの文字を変更	$S$ と $T$ の同じ位置に異なる文字を置く
操作3：文字列 $S$ に 1 つの文字を挿入	$T$ の上に $S$ の文字を置かない

### ◆ 具体例

たとえば、文字列  $S = "tessoku"$  を  $T = "tetsudou"$  に変更する操作の一例<sup>※3</sup>は、以下のような文字列の並びに対応します。

操作回数は、文字を置かなかった個数と、文字が一致していない個数を合計して 4 回です。



### ◆ 動的計画法へ

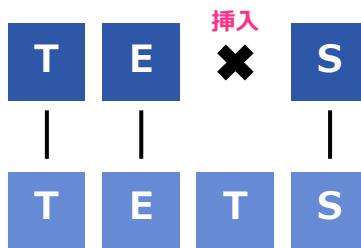
そこで、操作回数の最小値、すなわち**「文字を置かなかった個数と、文字が一致していない個数の合計」**（以下、合計コストとする）の最小値はどうやって計算すれば良いのでしょうか。まずは以下のような動的計画法を考えます。

$dp[i][j]$  : 文字列  $S$  の  $i$  文字目、文字列  $T$  の  $j$  文字目まで並べたとき、その時点での合計コストの最小値はいくつか。<sup>※4</sup>

※3 "tessoku" → "tetssoku" (挿入①) → "tetssdoku" (挿入②) → "tetsudoku" (変更)  
→ "tetsudou" (削除) という操作の場合

※4 文字列は左から順番に並べることを仮定している

たとえば  $S = \text{"tessoku"}$ 、 $T = \text{"tetsudou"}$  の場合、 $\text{dp}[3][4] = 1$  となります。なぜなら、 $S$  の 3 文字目まで (tes まで) と  $T$  の 4 文字目まで (tets まで) は、以下のようにしてコスト 1 で並べることができるからです。



## ◆ 配列 dp の計算

それでは、配列  $\text{dp}$  の値を計算することを考えましょう。まず明らかに  $\text{dp}[0][0]=0$  です。また  $\text{dp}[i][j]$  が指す状態に遷移する方法としては以下の 4 つが考えられます。

操作	合計コストの最小値
削除操作 ( $S$ の下に $T$ を置かない)	$\text{dp}[i-1][j]+1$
挿入操作 ( $T$ の下に $S$ を置かない)	$\text{dp}[i][j-1]+1$
変更操作 ( $S, T$ で異なる文字を置く)	$\text{dp}[i-1][j-1]+1$ $\because S_i \neq T_j$ の場合
何も変えない ( $S, T$ で同じ文字を置く)	$\text{dp}[i-1][j-1]$ $\because S_i = T_j$ の場合

したがって、配列  $\text{dp}$  の値は次のように計算すれば良いです。

$S_i = T_j$  の場合 :

- $\text{dp}[i][j] = \min(\text{dp}[i-1][j]+1, \text{dp}[i][j-1]+1, \text{dp}[i-1][j-1])$

$S_i \neq T_j$  の場合 :

- $\text{dp}[i][j] = \min(\text{dp}[i-1][j]+1, \text{dp}[i][j-1]+1, \text{dp}[i-1][j-1]+1)$

以上の内容を実装すると、次ページの解答例のようになります。ここで求めた答えは、 $S, T$  の長さをそれぞれ  $N, M$  とするとき、 $\text{dp}[N][M]$  であることに注意してください。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <string>
3 #include <algorithm>
4 using namespace std;
5
6 int N, M, dp[2009][2009];
7 string S, T;
8
9 int main() {
10     // 入力
11     cin >> S; N = S.size();
12     cin >> T; M = T.size();
13
14     // 動的計画法
15     dp[0][0] = 0;
16     for (int i = 0; i <= N; i++) {
17         for (int j = 0; j <= M; j++) {
18             if (i >= 1 && j >= 1 && S[i - 1] == T[j - 1]) {
19                 dp[i][j] = min({dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]});
20             }
21             else if (i >= 1 && j >= 1) {
22                 dp[i][j] = min({dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+1});
23             }
24             else if (i >= 1) {
25                 dp[i][j] = dp[i - 1][j] + 1;
26             }
27             else if (j >= 1) {
28                 dp[i][j] = dp[i][j - 1] + 1;
29             }
30         }
31     }
32
33     // 出力
34     cout << dp[N][M] << endl;
35
36 }
```

※Python のコードはサポートページをご覧ください

# 4.6

## 問題 B21 : Longest Subpalindrome (難易度 : ★6相当)

まず、文字列の中から回文を取り出す操作は、以下の操作に対応します。

- 最初に 1 つまたは 2 つの文字を選び、回文に追加する（2 つの文字を選ぶ場合は、同じ文字でなければならない）。
- その後、少しづつ範囲を広げていき、「**範囲の両端の文字が同じであれば、それらの文字を回文に追加する**」という操作を繰り返す。

たとえば文字列 “tanabata” から回文 “aabaa” を取り出す操作は、以下のような操作手順に対応します。

なお、この図では、回文として追加された部分を黒で表示しています。また、範囲を赤い矢印で示しています。



ここで、最初に 2 文字を選ぶ場合は、連続する 2 文字であると仮定してもかまいません。なぜなら、連続しない 2 文字を選ぶのは最適ではないからです（間の 1 文字を選ぶと、回文の長さが 1 増えます）。※5

※5 たとえば文字列 “kazan” の 2 文字目・4 文字目を最初に選ぶのは最適ではありません。なぜなら、3 文字目の「k」を追加すると、回文の長さが 1 増えるからです。

## ◆ 動的計画法を考える

そこで、最も多くの文字を回文として追加するにはどうすれば良いのでしょうか。以下のような動的計画法を考えます。

$dp[1][r]$  : 文字列の  $l$  文字目から  $r$  文字目までが範囲になっているとき、既に最大何文字を回文として追加できているか。

## ◆ 配列 $dp$ の計算

まず、初期状態は以下のようになります。

- 1 文字を選ぶ場合 :  $dp[i][i]=1$
- 2 文字を選ぶ場合 :  $dp[i][i+1]=2$  ( $S_i = S_{i+1}$  の場合)

次に、状態遷移を考えます。 $dp[1][r]$  の状態に遷移する方法として考えられるものは、以下の 3 つです。

操作	累計文字数の最大値
左端を 1 広げる	$dp[l+1][r]$
右端を 1 広げる	$dp[l][r-1]$
左端・右端を 1 広げ、回文に追加する	$dp[l+1][r-1]+2$ $\because S_l = S_r$ の場合

したがって、 $dp[1][r]$  の値は以下のようになります。

$S_l = T_t$  の場合 :

- $dp[1][r] = \max(dp[1][r-1], dp[l+1][r], dp[l+1][r-1]+2)$

$S_l \neq T_t$  の場合 :

- $dp[1][r] = \max(dp[1][r-1], dp[l+1][r])$

ここまで的内容を実装すると、次ページの解答例のようになります。計算量は  $O(N^2)$  です。なお、解答例では  $r-1$  の小さい順に  $dp[1][r]$  の値を計算していることに注意してください。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 int N;
6 int dp[1009][1009];
7 string S;
8
9 int main() {
10     // 入力
11     cin >> N;
12     cin >> S;
13
14     // 動的計画法 (初期状態)
15     for (int i = 0; i < N; i++) dp[i][i] = 1;
16     for (int i = 0; i < N - 1; i++) {
17         if (S[i] == S[i + 1]) dp[i][i + 1] = 2;
18         else dp[i][i + 1] = 1;
19     }
20
21     // 動的計画法 (状態遷移)
22     for (int LEN = 2; LEN <= N - 1; LEN++) {
23         for (int l = 0; l < N - LEN; l++) {
24             int r = l + LEN;
25
26             if (S[l] == S[r]) {
27                 dp[l][r] = max({ dp[l][r-1], dp[l+1][r], dp[l+1][r-1]+2 });
28             }
29             else {
30                 dp[l][r] = max({ dp[l][r-1], dp[l+1][r] });
31             }
32         }
33     }
34
35     // 答えを求める
36     cout << dp[0][N - 1] << endl;
37     return 0;
38 }
```

※Python のコードはサポートページをご覧ください

4.7

## 問題 B22 : Frog 1 with Restoration (難易度 : ★3相当)

部屋  $i$  にいるときの一手先の行動としては、「部屋  $i + 1$  に移動する」「部屋  $i + 2$  に移動する」の 2 つが考えられます。

そのため、部屋 1 から部屋  $i$  まで移動するための最小コストを  $dp[i]$  とするとき、 $dp[i]$  の値は以下のようにして計算することができます（問題 A16 とは異なり、配る遷移形式を使っています）。

最初、 $dp[1] = 0$  に設定し、 $dp[2], dp[3], \dots, dp[N] = \infty$  とする。

その後、 $i = 1, 2, \dots, N$  の順に、以下の操作を行う。

- $dp[i + 1]$  を  $\min(dp[i + 1], dp[i] + A_{i+1})$  に更新する
- $dp[i + 2]$  を  $\min(dp[i + 2], dp[i] + B_{i+2})$  に更新する

### ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 int N, A[100009], B[100009], dp[100009];
6
7 int main() {
8     // 入力
9     cin >> N;
10    for (int i = 2; i <= N; i++) cin >> A[i];
11    for (int i = 3; i <= N; i++) cin >> B[i];
12    // 配列 dp の初期化
13    dp[1] = 0;
14    for (int i = 2; i <= N; i++) dp[i] = 2000000000;
15
16    // 動的計画法
17    for (int i = 1; i <= N; i++) {
18        if (i<=N-1) dp[i+1] = min(dp[i+1], dp[i]+A[i+1]); // 部屋 i+1 に行く場合
19        if (i<=N-2) dp[i+2] = min(dp[i+2], dp[i]+B[i+2]); // 部屋 i+2 に行く場合
20    }
21    cout << dp[N] << endl;
22    return 0;
23 }
```

※Python のコードはサポートページをご覧ください

まず考えられる方法は、移動方法  $N!$  通りを全探索することです。しかし本問題の制約は  $N \leq 15$  であり、 $15!$  は  $10^{12}$  を超えるため、残念ながら実行時間制限に間に合いません。

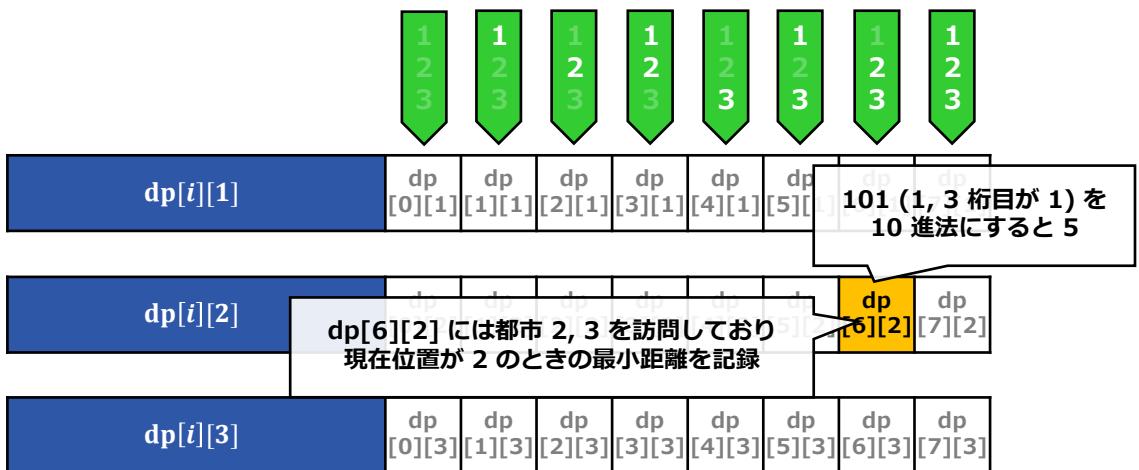
## ◆ 動的計画法（ビットDP）を考える

そこで、次のような動的計画法を考えます。既に訪問した都市  $i$  は、整数ではなく集合であることに注意してください。

$dp[i][j]$  : 既に訪問した都市の集合が  $i$  であり、現在位置が  $j$  であるときの、現時点での最小移動距離

なお、プログラム上では配列の添字として集合を設定することができず、整数にする必要があります。

整数にする方法としては、本の 141 ページ (4.8 節) に記したように、2 進法を使うなどの方法があります（下図参照）。



## ◆ 配列 dp はどうやって計算するか

まず、初期状態は  $dp[1][1]=0$  です※6。なぜなら、**最初は都市 1 から出発すると考えても一般性を失わない**からです。

次に、状態遷移はどのようにして行えば良いのでしょうか。配る遷移形式にしたがって考えると、次に訪れる都市を  $k$  とし、都市  $j$  と  $k$  の間の距離を  $dist(j, k)$  とするとき、以下のようにになります。

```
dp[i+(1<<k)][k] = min(dp[i+(1<<k)][k], dp[i][j] + dist(j,k));  
集合 i + 都市 k (値は i + 2k)
```

ここまで的内容を実装すると、次ページの解答例のようになります。答えは  $dp[2^N - 1][1]$  であることに注意してください。

なお、配列  $dp$  の要素数は  $N \times 2^N$  であり、各要素につき計算量  $O(N)$  かかるので、プログラム全体の計算量は  $O(N^2 \times 2^N)$  です。

## ◆ 解答例 (C++)

```
1 #include <iostream>  
2 #include <cmath>  
3 #include <algorithm>  
4 using namespace std;  
5  
6 int N, X[19], Y[19];  
7 double dp[1 << 16][19];  
8  
9 int main() {  
10    // 入力  
11    cin >> N;  
12    for (int i = 0; i < N; i++) cin >> X[i] >> Y[i];  
13  
14    // 配列 dp の初期化  
15    for (int i = 0; i < (1 << N); i++) {  
16        for (int j = 0; j < N; j++) dp[i][j] = 1e9;  
17    }  
18  
19    // 動的計画法 (dp[通った都市][今いる都市] となっている)  
20    dp[0][0] = 0;  
21    for (int i = 0; i < (1 << N); i++) {  
22        for (int j = 0; j < N; j++) {  
23            if (dp[i][j] >= 1e9) continue;
```

※6 最後に「スタート地点」に戻るため、ここではスタート地点を「訪問した都市の集合」に含めていません

```
24 // 都市 j から都市 k に移動したい！
25 for (int k = 0; k < N; k++) {
26     // 既に都市 k を通っていた場合
27     if ((i / (1 << k)) % 2 == 1) continue;
28
29     // 状態遷移
30     double DIST = sqrt(1.0*(X[j]-X[k])*(X[j]-X[k]) + 1.0*(Y[j]-
31 Y[k])*(Y[j]-Y[k]));
32     dp[i+(1<<k)][k] = min(dp[i+(1<<k)][k], dp[i][j] + DIST);
33 }
34 }
35
36 // 答えを出力
37 printf("%.12lf\n", dp[(1 << N) - 1][0]);
38 return 0;
39 }
```

※Python のコードはサポートページをご覧ください

4.9

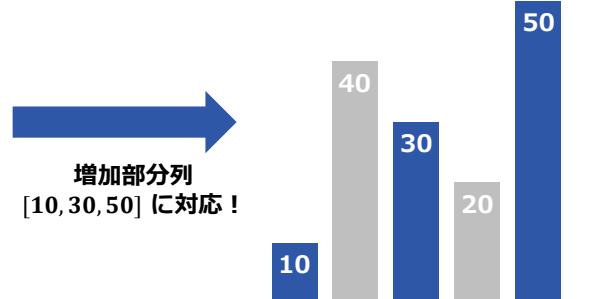
## 問題 B24 : Many Boxes

(難易度：★5相当)

まずは手始めに、 $X_1 < X_2 < \dots < X_N$  を満たす場合を考えます。少し衝撃的かもしれません、このようなケースでは、答えは**列  $[Y_1, Y_2, \dots, Y_N]$  の最長増加部分列問題の答え**と一致します。

なぜなら、1つの「箱の選び方」が、1つの「最長増加部分列」に対応するからです<sup>※7</sup>。箱 1・3・5 を選ぶ場合の具体例を以下に示します（注：最長増加部分列については、本の 144 ページをご覧ください）。

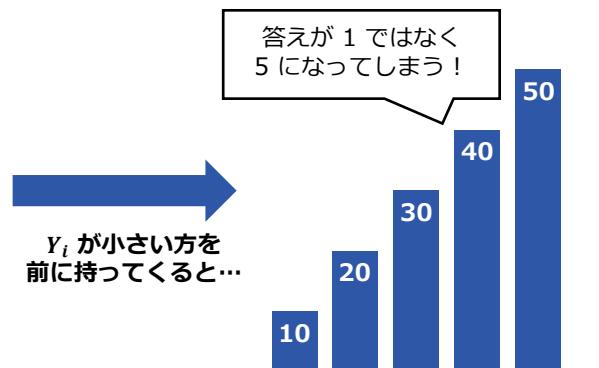
箱の番号	大きさ
箱1	$(X_1, Y_1) = (10, 10)$
箱2	$(X_2, Y_2) = (20, 40)$
箱3	$(X_3, Y_3) = (30, 30)$
箱4	$(X_4, Y_4) = (40, 20)$
箱5	$(X_5, Y_5) = (50, 50)$



それでは、 $X_1 < X_2 < \dots < X_N$  を満たさない場合はどうでしょうか。箱の番号は答えに関係ないので、あらかじめ  $X_i$  の小さい順に箱をソートしておけば良いです。

ただし、縦の長さ  $X_i$  が同じである場合は、横の長さ  $Y_i$  が大きい方を前に持ってくる必要があることに注意が必要です（下図参照）。

箱の番号	大きさ
箱1	$(X_1, Y_1) = (10, 10)$
箱2	$(X_2, Y_2) = (10, 20)$
箱3	$(X_3, Y_3) = (10, 30)$
箱4	$(X_4, Y_4) = (10, 40)$
箱5	$(X_5, Y_5) = (10, 50)$



※7 必ず増加部分列になる理由は、「箱を箱の中に入れるためには、縦の長さも横の長さも真に短くなつていなければならないこと」から分かります

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 int N, X[100009], Y[100009];
7 int LEN, L[100009];
8
9 // 配列 A の最長増加部分列 (LIS) の長さを計算する
10 // 配列 dp を使わない実装方法を利用している
11 int Get_LISvalue(vector<int> A) {
12     LEN = 0;
13     for (int i = 1; i <= A.size(); i++) L[i] = 0;
14
15     // 動的計画法
16     for (int i = 0; i < A.size(); i++) {
17         int pos = lower_bound(L + 1, L + LEN + 1, A[i]) - L;
18         L[pos] = A[i];
19         if (pos > LEN) LEN += 1;
20     }
21     return LEN;
22 }
23
24 int main() {
25     // 入力
26     cin >> N;
27     for (int i = 1; i <= N; i++) cin >> X[i] >> Y[i];
28
29     // ソート
30     vector<pair<int, int>> tmp;
31     for (int i = 1; i <= N; i++) tmp.push_back(make_pair(X[i], -Y[i]));
32     sort(tmp.begin(), tmp.end());
33
34     // 求める LIS の配列は?
35     vector<int> A;
36     for (int i = 0; i < tmp.size(); i++) {
37         A.push_back(-tmp[i].second);
38     }
39
40     // 出力
41     cout << Get_LISvalue(A) << endl;
42     return 0;
43 }
```

※Python のコードはサポートページをご覧ください

# 第 5 章

## 數學的問題

應用問題 5.1	· · · · ·	57
應用問題 5.2	· · · · ·	58
應用問題 5.3	· · · · ·	59
應用問題 5.4	· · · · ·	60
應用問題 5.5	· · · · ·	61
應用問題 5.6	· · · · ·	63
應用問題 5.7	· · · · ·	65
應用問題 5.8	· · · · ·	67
應用問題 5.9	· · · · ·	69

この問題は、本の 158 ページで説明した「エラトステネスのふるい」を使って解くことができます。

本問題の制約は  $N = 1000000$  ですが、エラトステネスのふるいの計算量は  $O(N \times \log \log N)$  ですので、 $\log \log 1000000 \approx 3$  より余裕を持って実行時間制限に間に合います。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int N;
5 bool Deleted[1000009]; // 整数 x が消されている場合に限り Deleted[x]=true
6
7 int main() {
8     // 入力
9     cin >> N;
10
11    // エラトステネスのふるい (i は  $\sqrt{N}$  以下の最大の整数までループする)
12    for (int i = 2; i <= N; i++) Deleted[i] = false;
13    for (int i = 2; i * i <= N; i++) {
14        if (Deleted[i] == true) continue;
15        for (int j = i * 2; j <= N; j += i) Deleted[j] = true;
16    }
17
18    // 答えを出力
19    for (int i = 2; i <= N; i++) {
20        if (Deleted[i] == false) cout << i << endl;
21    }
22    return 0;
23 }
```

※Python のコードはサポートページをご覧ください

## 5.2

# 問題 B27 : Calculate LCM

(難易度：★2相当)

まず、整数  $a, b$  の最大公約数と最小公倍数に関して、以下の性質が必ず成り立ちます。

$$a \times b = (\text{最大公約数}) \times (\text{最小公倍数})$$

たとえば 25 と 30 の最大公約数は 5、最小公倍数は 150 であり、たしかに等式「 $25 \times 30 = 5 \times 150$ 」が成り立っています<sup>※1</sup>。したがって、最小公倍数の値は、以下のアルゴリズムで高速に計算できます。

1. ユークリッドの互除法を使って、 $a, b$  の最大公約数を計算する
2.  $(\text{最小公倍数}) = a \times b \div (\text{最大公約数})$  を出力する

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int GCD(int A, int B) {
5     while (A >= 1 && B >= 1) {
6         if (A >= B) {
7             A = (A % B); // A の値を変更する場合
8         }
9     else {
10         B = (B % A); // B の値を変更する場合
11     }
12 }
13 if (A != 0) return A;
14 return B;
15 }
16
17 int main() {
18     long long A, B;
19     cin >> A >> B;
20     cout << A * B / GCD(A, B) << endl;
21     return 0;
22 }
```

※Python のコードはサポートページをご覧ください

※1 これが成り立つ理由の証明は少し難しいですが、詳しくは <https://manabitimes.jp/math/1032> をご覧ください

# 5.3

## 問題 B28 : Fibonacci Easy (難易度：★2相当)

自然に実装すると、以下のようになります。しかし、制約の最大値である  $N = 10^7$  の場合、第  $N$  項  $a_N$  の値は 200 万桁を超えて、残念ながらオーバーフローを起こしてしまいます。

```
1 a[1] = 1;
2 a[2] = 1;
3 for (int i = 3; i <= N; i++) a[i] = (a[i - 1] + a[i - 2]) % 1000000007;
4 cout << a[N] % 1000000007 << endl;
```

そこで、 $a_i$  の値を 1 回計算するごとに余りをとると、オーバーフローを防ぐことができます。解答例は以下の通りです。 $2 \times 1000000007 < 2^{31}$  より、int 型などの 32 ビット整数でも十分であることに注意してください。

### ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 const int mod = 1000000007;
5 int N, a[10000009];
6
7 int main() {
8     // 入力
9     cin >> N;
10
11    // フィボナッチ数列の計算
12    a[1] = 1;
13    a[2] = 1;
14    for (int i = 3; i <= N; i++) {
15        a[i] = (a[i - 1] + a[i - 2]) % mod;
16    }
17
18    // 出力
19    cout << a[N] << endl;
20    return 0;
21 }
```

※Python のコードはサポートページをご覧ください

例題 A29 の解答例（本の 172 ページ）では、30 回のループを行いました。しかし今回は制約が  $b \leq 10^{18}$  であるため、ループ回数を 60 回まで増やす必要があります（ $10^{18} < 2^{60}$  より、60 回あれば十分です）。

その他の注意点として、9 行目の変数 `wari` は `long long` 型などの 64 ビット整数にする必要があること、などが挙げられます。解答例は以下の通りです。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 // a の b 乗を m で割った余りを返す関数
5 // 変数 a は a^1 → a^2 → a^4 → a^8 → a^16 → … と変化
6 long long Power(long long a, long long b, long long m) {
7     long long p = a, Answer = 1;
8     for (int i = 0; i < 60; i++) {
9         long long wari = (1LL << i);
10        if ((b / wari) % 2 == 1) {
11            Answer = (Answer * p) % m; // 「a の 2^i 乗」が掛けられるとき
12        }
13        p = (p * p) % m;
14    }
15    return Answer;
16 }
17
18 int main() {
19     long long a, b;
20     cin >> a >> b;
21     cout << Power(a, b, 1000000007) << endl;
22     return 0;
23 }
```

※Python のコードはサポートページをご覧ください

# 5.5

## 問題 B30 : Combination 2 (難易度：★4相当)

マス  $(1, 1)$  からマス  $(H, W)$  まで行くためには、全部で  $H + W - 2$  回の移動を行う必要があります。その中の  $W - 1$  回が右方向である必要があります。

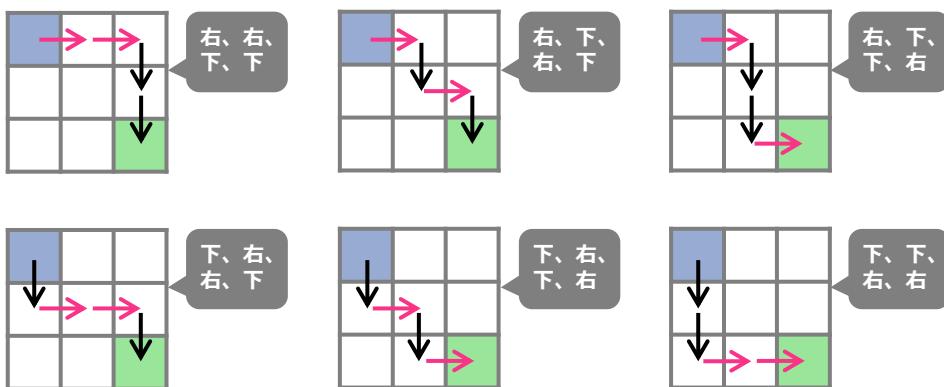
逆に、右方向の移動回数が  $W - 1$  回であれば、必ずマス  $(H, W)$  でゴールします。したがって、求める移動方法の数は、 $H + W - 2$  個の中から  $W - 1$  個を選ぶ方法の数  $_{H+W-2}C_{W-1}$  通りになります。



### ◆ 具体例

たとえば  $H = 3, W = 3$  の場合を考えましょう。左上のマス  $(1, 1)$  から右下のマス  $(3, 3)$  まで行くためには、全部で 4 回の移動を行う必要があり、その中の 2 回が右方向の移動である必要があります。

逆に、4 回中 2 回が右方向であれば、必ずマス  $(3, 3)$  にたどり着くので、移動方法の数は  ${}_4C_2 = 6$  通りとなります。



## 解答例 (C++)

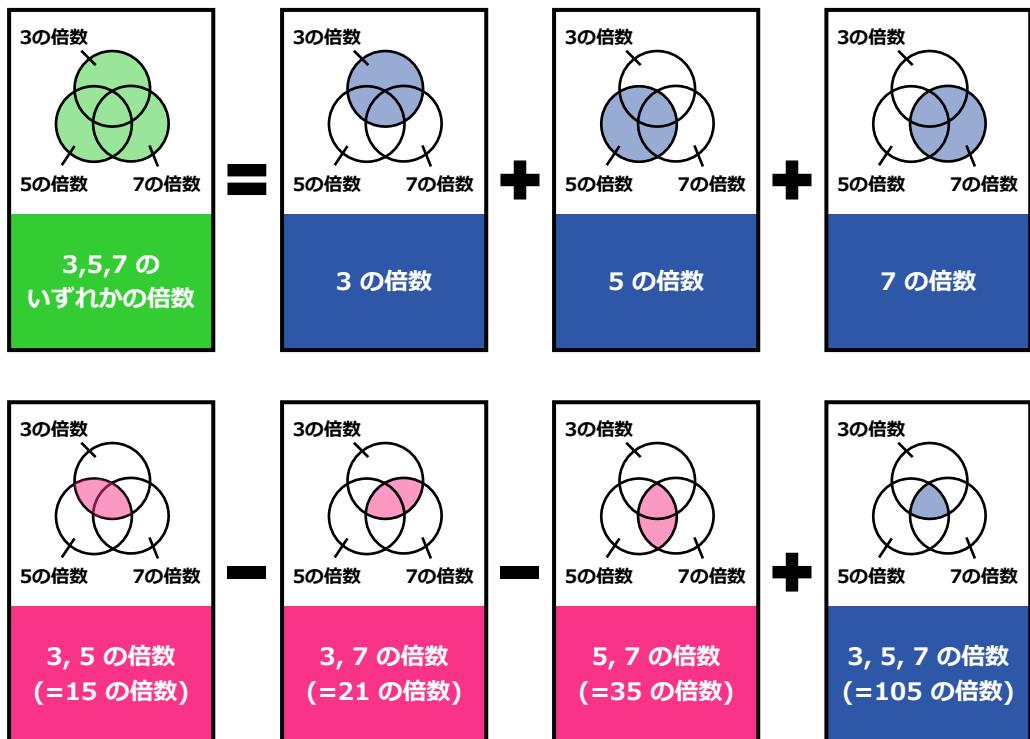
```
1 #include <iostream>
2 using namespace std;
3
4 // a の b 乗を m で割った余りを返す関数
5 // 変数 a は a^1 → a^2 → a^4 → a^8 → a^16 → … と変化
6 long long Power(long long a, long long b, long long m) {
7     long long p = a, Answer = 1;
8     for (int i = 0; i < 30; i++) {
9         int wari = (1 << i);
10        if ((b / wari) % 2 == 1) {
11            Answer = (Answer * p) % m; // 「a の 2^i 乗」が掛けられるとき
12        }
13        p = (p * p) % m;
14    }
15    return Answer;
16 }
17
18 // a ÷ b を m で割った余りを返す関数
19 long long Division(long long a, long long b, long long m) {
20     return (a * Power(b, m - 2, m)) % m;
21 }
22
23 // nCr mod 1000000007 を返す関数
24 long long ncr(int n, int r) {
25     const long long M = 1000000007;
26
27     // 手順 1: 分子 a を求める
28     long long a = 1;
29     for (int i = 1; i <= n; i++) a = (a * i) % M;
30
31     // 手順 2: 分母 b を求める
32     long long b = 1;
33     for (int i = 1; i <= r; i++) b = (b * i) % M;
34     for (int i = 1; i <= n - r; i++) b = (b * i) % M;
35
36     // 手順 3: 答えを求める
37     return Division(a, b, M);
38 }
39
40 int main() {
41     // 入力
42     long long H, W;
43     cin >> H >> W;
44
45     // 出力
46     cout << ncr(H + W - 2, W - 1) << endl;
47     return 0;
48 }
```

※Python のコードはサポートページをご覧ください

# 5.6

## 問題 B31 : Divisors Hard (難易度：★3相当)

3つの集合の包除原理（→本の 180 ページ）より、「3, 5, 7 のいずれかの倍数であるものの個数」は以下のようにして計算することができます。



そこで、1 以上  $N$  以下の  $a$  の倍数の個数は  $[N \div a]$  個（ $[x]$  は  $x$  の小数点以下切り捨て）であるため、本問題の答えは次式で表されます。

$$\left[ \frac{N}{3} \right] + \left[ \frac{N}{5} \right] + \left[ \frac{N}{7} \right] - \left[ \frac{N}{15} \right] - \left[ \frac{N}{21} \right] - \left[ \frac{N}{35} \right] + \left[ \frac{N}{105} \right]$$

この値を出力する、次ページの解答例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  $O(1)$  です。

## 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     long long N;
6     cin >> N;
7
8     long long A1 = (N / 3); // 3 で割り切れるものの個数
9     long long A2 = (N / 5); // 5 で割り切れるものの個数
10    long long A3 = (N / 7); // 5 で割り切れるものの個数
11    long long A4 = (N / 15); // 3, 5 で割り切れるもの (= 15 の倍数) の個数
12    long long A5 = (N / 21); // 3, 7 で割り切れるもの (= 21 の倍数) の個数
13    long long A6 = (N / 35); // 5, 7 で割り切れるもの (= 35 の倍数) の個数
14    long long A7 = (N / 105); // 3, 5, 7 で割り切れるもの (= 105 の倍数) の個数
15    cout << A1 + A2 + A3 - A4 - A5 - A6 + A7 << endl;
16    return 0;
17 }
```

※Python のコードはサポートページをご覧ください

この問題も、問題 A32（本の 181 ページ）と同じように、石の数が 0 個のときから順番に「勝ちの状態か負けの状態か」を求めていくと、答えが分かります。アルゴリズムの具体的な流れは以下の通りです。

$i = 0, 1, 2, \dots, N$  の順に、次の規則にしたがって「石が  $i$  個のときは勝ちの状態であるか」を求める。

### [規則]

- 石が  $i - a_1, \dots, i - a_K$  個のいずれかで負けの状態のとき：勝ちの状態
- 石が  $i - a_1, \dots, i - a_K$  個すべてで勝ちの状態のとき：負けの状態

このアルゴリズムを実装すると、以下の解答例のようになります。計算量は  $O(NK)$  です。なお、プログラム上では「石が  $i$  個」が勝ちの状態のとき `dp[i]=true`、負けの状態のとき `dp[i]=false` となっています。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 using namespace std;
3
4 // 配列 dp について: dp[x]=true のとき勝ちの状態、dp[x]=false のとき負けの状態
5 int N, K, A[109];
6 bool dp[100009];
7
8 int main() {
9     // 入力
10    cin >> N >> K;
11    for (int i = 1; i <= K; i++) cin >> A[i];
12
13    // 勝者を計算する
14    for (int i = 0; i <= N; i++) {
15        dp[i] = false;
16        for (int j = 1; j <= K; j++) {
17            if (i >= A[j] && dp[i - A[j]] == false) {
18                dp[i] = true; // 贠けの状態に遷移できれば、勝ちの状態
19            }
20        }
21    }
22
23    cout << dp[N] << endl;
24 }
```

```
20     }
21 }
22
23 // 出力
24 if (dp[N] == true) cout << "First" << endl;
25 else cout << "Second" << endl;
26 return 0;
27 }
```

※Python のコードはサポートページをご覧ください

この問題は、「何行目か」と「何列目か」を別々に考へることで、山の数が $2N$ のニムに帰着させることができます。具体的には以下の通りです。

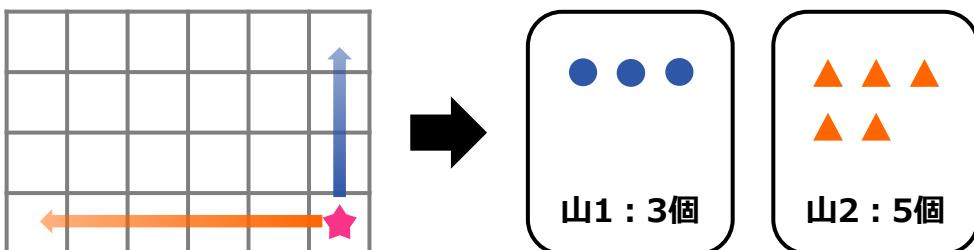
現在の*i*個目のコマの位置を、上から $a_i + 1$ 行目、左から $b_i + 1$ 列目とする。このとき、一回の操作では次のことができる。

- コマ*i*を左方向に動かす： $b_i$ を1以上減らす
- コマ*i*を上方向に動かす： $a_i$ を1以上減らす
- ただし、マス目の範囲に収めるため、 $a_i, b_i$ は0以上であるべき

つまり、一回の操作は「 $[a_1, a_2, \dots, a_N, b_1, b_2, \dots, b_N]$ の中から1つ選び、負の数にならない範囲で1以上減らすこと」に対応する。

これは山の数が $2N$ であり、各山の石の数が $[a_1, a_2, \dots, a_N, b_1, b_2, \dots, b_N]$ であるニムと全く等価である。

たとえばコマが1個であり、そのコマが上から4行目・左から6行目に存在する場合、「石が3, 5個ある2つの山のニム」と等価です（もちろん $3 \text{ XOR } 5 \neq 0$ なので、先手必勝です※2）。



したがって、本問題の答えは、 $(A_1 - 1) \text{ XOR } \dots \text{ XOR } (A_N - 1) \text{ XOR } (B_1 - 1) \text{ XOR } \dots \text{ XOR } (B_N - 1) = 0$ であれば後手必勝、そうでなければ先手必勝となります。次ページに解答例を示します。

※2 ニムでは、各山の石の数をすべてXORした値が0であれば後手必勝であったことを思い出しましょう。本の186ページに書かれています

## 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int H, W;
5 int N, A[200009], B[200009];
6
7 int main() {
8     // 入力
9     cin >> N >> H >> W;
10    for (int i = 1; i <= N; i++) cin >> A[i] >> B[i];
11
12    // 全部 XOR した値（二ム和）を求める
13    int XOR_Sum = 0;
14    for (int i = 1; i <= N; i++) XOR_Sum = (XOR_Sum ^ (A[i] - 1));
15    for (int i = 1; i <= N; i++) XOR_Sum = (XOR_Sum ^ (B[i] - 1));
16
17    // 出力
18    if (XOR_Sum != 0) cout << "First" << endl;
19    if (XOR_Sum == 0) cout << "Second" << endl;
20    return 0;
21 }
```

※Python のコードはサポートページをご覧ください

## 5.9

## 問題 B33 : Game 6

(難易度：★5相当)

まずは山が 1 つの場合を考えましょう。 $X = 2, Y = 3$  のとき、石の数が 0 個から 39 個のときの Grundy 数は以下の通りです（本の 191 ページのように、実際に手で計算してみましょう）。

石の数	0	1	2	3	4	5	6	7	8	9
Grundy 数	0	0	1	1	2	0	0	1	1	2
石の数	10	11	12	13	14	15	16	17	18	19
Grundy 数	0	0	1	1	2	0	0	1	1	2
石の数	20	21	22	23	24	25	26	27	28	29
Grundy 数	0	0	1	1	2	0	0	1	1	2
石の数	30	31	32	33	34	35	36	37	38	39
Grundy 数	0	0	1	1	2	0	0	1	1	2

0,0,1,1,2 が周期的に繰り返されていますね。それでは、この周期性は石の数が 40 個以上でも成り立つのでしょうか。答えは Yes です。少し難しいですが、以下のようにして証明することができます。

石が  $(i - 3, i - 2, i - 1)$  個の Grundy 数の値から、石が  $(i - 2, i - 1, i - 0)$  個の Grundy 数を求めることを考える。このとき<sup>※3</sup>、

1.  $(0, 0, 1)$  の次は  $(0, 1, 1)$  である [ $0, 0$  にない最小の非負整数は 1]
2.  $(0, 1, 1)$  の次は  $(1, 1, 2)$  である [ $0, 1$  にない最小の非負整数は 2]
3.  $(1, 1, 2)$  の次は  $(1, 2, 0)$  である [ $1, 1$  にない最小の非負整数は 0]
4.  $(1, 2, 0)$  の次は  $(2, 0, 0)$  である [ $1, 2$  にない最小の非負整数は 0]
5.  $(2, 0, 0)$  の次は  $(0, 0, 1)$  である [ $2, 0$  にない最小の非負整数は 1]

5 回でまた 1. に戻るので、5 個周期で「0→0→1→1→2」と繰り返す。

※3 遷移できる「石の数が  $i - 3, i - 2$  個の状態」の Grundy 数を赤色で示しています。

したがって、石の数が  $A_1$  個のとき、Grundy 数は下表のようになります。

石の数 $A_1$ を 5 で割った余り	0	1	2	3	4
Grundy 数	0	0	1	1	2

## ◆ 山が 2 個以上の場合

山が 2 つ以上の場合は、本の 192 ページに示したように、それぞれの山に対する Grundy 数を計算することで勝敗を判定することができます。

Grundy 数をすべて XOR した値 XOR\_Sum が 0 であるとき後手必勝となり、そうでなければ先手必勝です。実装例を以下に示します。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 long long N, X, Y, A[100009];
5
6 int main() {
7     // 入力
8     cin >> N >> X >> Y;
9     for (int i = 1; i <= N; i++) cin >> A[i];
10
11    // Grundy 数を計算
12    int XOR_Sum = 0;
13    for (int i = 1; i <= N; i++) {
14        if (A[i] % 5 == 0 || A[i] % 5 == 1) XOR_Sum ^= 0;
15        if (A[i] % 5 == 2 || A[i] % 5 == 3) XOR_Sum ^= 1;
16        if (A[i] % 5 == 4) XOR_Sum ^= 2;
17    }
18
19    // 出力
20    if (XOR_Sum != 0) cout << "First" << endl;
21    if (XOR_Sum == 0) cout << "Second" << endl;
22    return 0;
23 }
```

※Python のコードはサポートページをご覧ください

# 第6章

## 考察テクニック

應用問題 6.1	· · · · ·	72
應用問題 6.2	· · · · ·	74
應用問題 6.3	· · · · ·	78
應用問題 6.4	· · · · ·	81
應用問題 6.5	· · · · ·	84
應用問題 6.6	· · · · ·	86
應用問題 6.7	· · · · ·	88
應用問題 6.8	· · · · ·	91
應用問題 6.9	· · · · ·	92
應用問題 6.10	· · · · ·	94

# 6.1

## 問題 B36 : Switching Lights (難易度：★2相当)

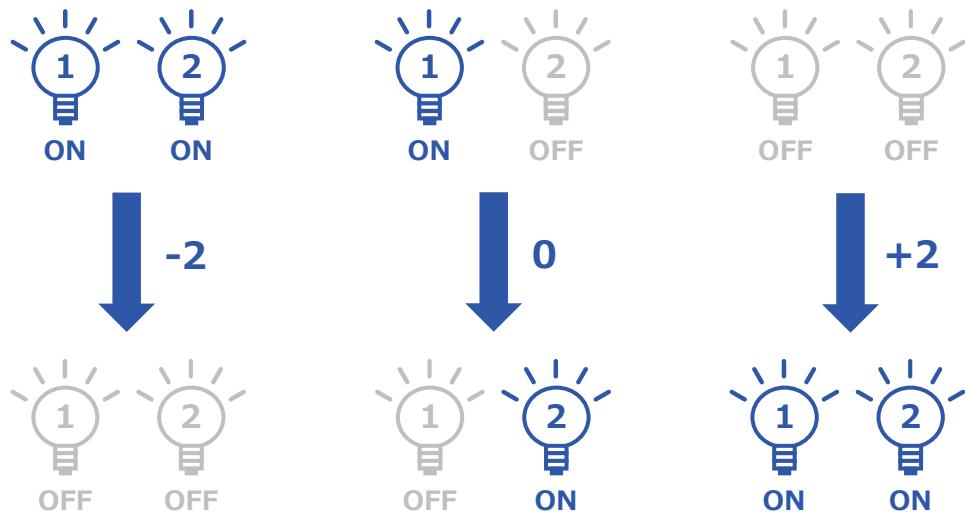
早速ですが結論から書きます。最初に ON になっている豆電球の個数を numON とするとき、答えは以下のようになります。

- numON の偶奇と  $K$  の偶奇が一致するとき : Yes
- numON の偶奇と  $K$  の偶奇が一致しないとき : No

したがって、次ページの解答例のようなプログラムにより、計算量  $O(N)$  で正しい答えを出すことができます。

### ◆ 解法の証明

それでは、なぜ偶奇が一致しなければ答えが No になるのでしょうか。この理由は、「2 つの豆電球の状態を反転させる」という操作を行っても、**ON になっている豆電球の個数の偶奇は変わらない**（個数の変化は  $-2/0/+2$  のいずれかである）ことから分かります。

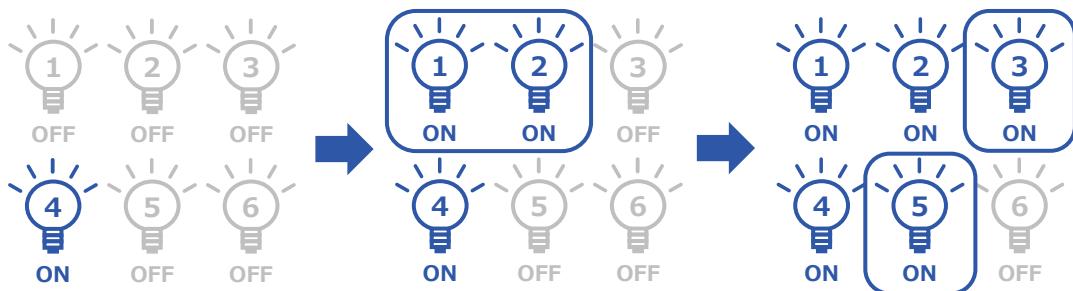


(次ページへ続く)

逆に、偶奇が一致している場合は、以下のような操作を行うことで  $K$  個の豆電球を ON にできるので、答えは Yes です。

numON<K の場合	OFF になっている豆電球を 2 つ選び、反転させる という操作を $(K - \text{numON})/2$ 回行う
numON>K の場合	ON になっている豆電球を 2 つ選び、反転させる という操作を $(\text{numON} - K)/2$ 回行う

たとえば  $A = (0,0,0,1,0,0), K = 5$  の場合、操作例は次のようにになります。2 回の操作で、ON の豆電球が 5 個になっています。



## ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int N, K;
6 string S;
7
8 int main() {
9     // 入力
10    cin >> N >> K;
11    cin >> S;
12
13    // ON となっているものの個数を数える
14    int numON = 0;
15    for (int i = 0; i < N; i++) {
16        if (S[i] == '1') numON += 1;
17    }
18
19    // 答えを出力
20    if (numON % 2 == K % 2) cout << "Yes" << endl;
21    else cout << "No" << endl;
22    return 0;
23 }
```

※Python のコードはサポートページをご覧ください

## 6.2

### 問題 B37 : Sum of Digits

(難易度：★5相当)

※この問題は例題と比べて相当難しいです。

この問題を解く最も単純な方法は、「1 の各桁の和は？」 「2 の各桁の和は？」といったように 1 つずつ調べていくことです。

もちろん、この方法でも正しい答えを出すことができますが、計算量は  $O(N)$  と遅く<sup>※1</sup>、 $N < 10^{15}$  の制約では実行時間制限に間に合いません。

そこで、以下のようなアルゴリズムを使うと、より高速に答えを導き出すことができます。

#### [手順1]

- 各  $(i, j)$  に対して、次の値  $R_{i,j}$  を計算する。
- 下から  $i$  桁目の数字が  $j$  となる、0 以上  $N$  以下の整数は何個か？<sup>※2</sup>

#### [手順2]

- 求める答えである「各桁の和の合計」は「 $j \times R_{i,j}$  の総和」である。

ここで、答えが  $j \times R_{i,j}$  の総和になる理由は、「下から  $i$  桁目が  $j$  であるときの答えへの寄与分」が  $j \times R_{i,j}$  となるからです。

たとえば  $N = 47, (i, j) = (0, 8)$  の場合を考えましょう。下から 0 桁目（つまり 1 の位）が 8 であるような整数は  $\{8, 18, 28, 38\}$  の 4 個あります。

そして、1 の位の 8 が「各桁の和の合計」に与える影響は、8 が 4 回足されているので  $8 \times 4 = 32$  です（これが  $j \times R_{i,j}$  の意味です）。



しかし、 $R_{i,j}$  の値を高速に求める部分が難所になりますので、 $R_{0,j}$  から順番に少しづつ考えていきましょう。

※1 実装によっては、計算量が  $O(N \log N)$  となります。

※2 1 以上より 0 以上の方がプログラムの実装が楽なので、0 以上の個数を考えています。

なお、存在しない位は 0 であるものとして考えます（例：8 の 10 の位は 0）。

## ◆ 一般のケースで $R_{i,j}$ を求める (1)

まずは 1 の位の個数  $R_{0,j}$  ( $0 \leq j \leq 9$ ) についてはどうでしょうか。手始めに  $N = 723$  のケースを考えると、次のようにになります。

以下の理由で  $R_{0,0} \sim R_{0,3}$  が 73、 $R_{0,4} \sim R_{0,9}$  が 72 となる：

- 0～719 の範囲では、1 の位に 0～9 が 72 回ずつ出現
- 720～723 の範囲では、1 の位に 0～3 が 1 回ずつ出現

一般の  $N$  の場合も同じように考えると、次表のようになります (0 以上  $[N \div 10] \times 10$  未満、 $[N \div 10] \times 10$  以上  $N$  以下) の 2 つに分割して考えています)。

条件	$R_{i,j}$ の値
$j \leq (1 \text{ の位の値})$ の場合	$[N \div 10] + 1$ 個※3
$j > (1 \text{ の位の値})$ の場合	$[N \div 10]$ 個

## ◆ 一般のケースで $R_{i,j}$ を求める (2)

次に 10 の位の個数  $R_{1,j}$  ( $0 \leq j \leq 9$ ) についてはどうでしょうか。手始めに  $N = 723$  のケースを考えると、次のようにになります。

以下の理由で  $R_{0,0} = R_{0,1} = 80$ 、 $R_{0,2} = 74$ 、 $R_{0,3} \sim R_{0,9}$  が 70 となる：

- 0～699 の範囲では、10 の位に 0～9 が 70 回ずつ出現
- 700～719 の範囲では、10 の位に 0～1 が 10 回ずつ出現
- 720～723 の範囲では、10 の位に 2 が 4 回出現

一般の  $N$  の場合も同じように考えると、次表のようになります (0 以上  $[N \div 100] \times 100$  未満、 $[N \div 100] \times 100$  以上  $[N \div 10] \times 10$  未満、 $[N \div 10] \times 10$  以上  $N$  以下) の 3 つに分割して考えています)。

条件	$R_{i,j}$ の値
$j < (10 \text{ の位の値})$ の場合	$[N \div 100] \times 10 + 10$ 個
$j = (10 \text{ の位の値})$ の場合	$[N \div 100] \times 10 + (N \bmod 10 + 1)$ 個※4
$j > (10 \text{ の位の値})$ の場合	$[N \div 100] \times 10$ 個

※3  $[x]$  は「 $x$  以下の小数点以下切り捨て」です。

※4  $a \bmod b$  は「 $a$  を  $b$  で割ったときの余り」です。

## ◆ 一般のケースで $R_{i,j}$ を求める (2)

最後に、100 の位以上についても同様に考えることができます。整数の組  $(i,j)$  に対する  $R_{i,j}$  の値は以下の通りです。

条件	$R_{i,j}$ の値
$j < (10^i \text{ の位の値})$ の場合	$[N \div 10^{i+1}] \times 10^i + 10^i \text{ 個}$
$j = (10^i \text{ の位の値})$ の場合	$[N \div 10^{i+1}] \times 10^i + (N \bmod 10^i + 1) \text{ 個}^{※4}$
$j > (10^i \text{ の位の値})$ の場合	$[N \div 10^{i+1}] \times 10^i \text{ 個}$

したがって、この問題を解くプログラムは以下のようにして実装できます。本問題の制約は  $N < 10^{15}$  ですので、少なくとも  $i = 14$  までの範囲で  $R_{i,j}$  を計算しなければならないことに注意してください。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 long long N;
6 long long R[18][10]; // R[i][j] は「下から i 行目が j となるような N 以下の整数の個数」
7 long long Power10[18];
8
9 int main() {
10     // 入力
11     cin >> N;
12
13     // 10 の N 乗を求める
14     Power10[0] = 1;
15     for (int i = 1; i <= 16; i++) Power10[i] = 10LL * Power10[i - 1];
16
17     // R[i][j] の値を計算
18     for (int i = 0; i <= 14; i++) {
19         // 下から i 行目の数字を求める
20         long long Digit = (N / Power10[i]) % 10LL;
21
22         // R[i][j] の値を求める
23         for (int j = 0; j < 10; j++) {
24             if (j < Digit) {
25                 R[i][j] = (N / Power10[i+1]) * Power10[i] + Power10[i];
26             }
27             if (j == Digit) {
28                 R[i][j] = (N / Power10[i+1]) * Power10[i] + (N % Power10[i]) + 1LL;
29             }
30             if (j > Digit) {
31                 R[i][j] = (N / Power10[i+1]) * Power10[i];
32             }
33         }
34     }
35 }
```

```
33     }
34 }
35
36 // 答えを求める
37 long long Answer = 0;
38 for (int i = 0; i <= 15; i++) {
39     for (int j = 0; j < 10; j++) Answer += 1LL * j * R[i][j];
40 }
41
42 // 出力
43 cout << Answer << endl;
44 return 0;
45 }
```

※Python のコードはサポートページをご覧ください

## 6.3

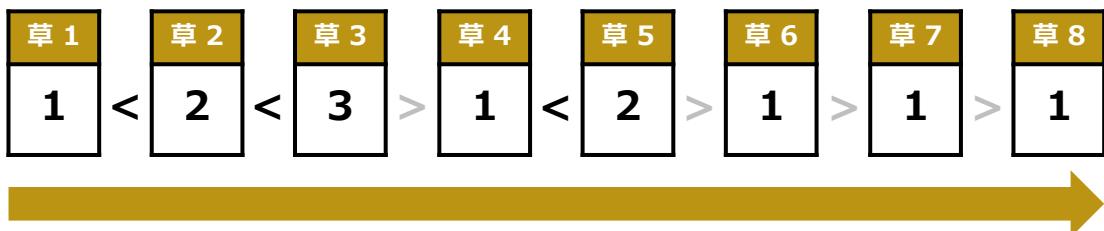
# 問題 B38 : Height of Grass (難易度 : ★4相当)

この問題を解くための重要なポイントは、「草の高さは絶対に〇〇以上である」という下限値を考えることです。

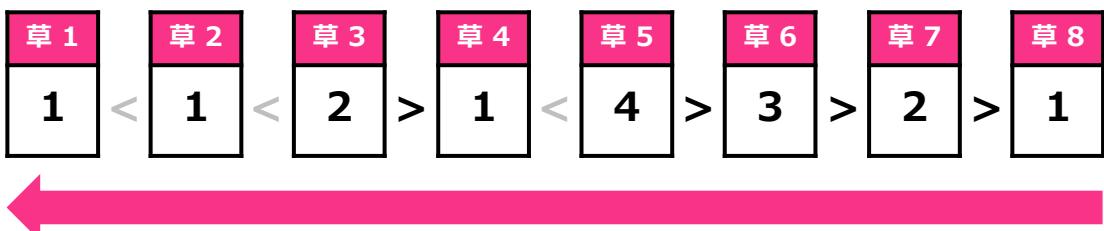
まずは手始めに、 $N = 8, S = \text{"AABABBBB"}$  というケースに対して下限値を求める考えましょう（イメージ図を以下に示します）。



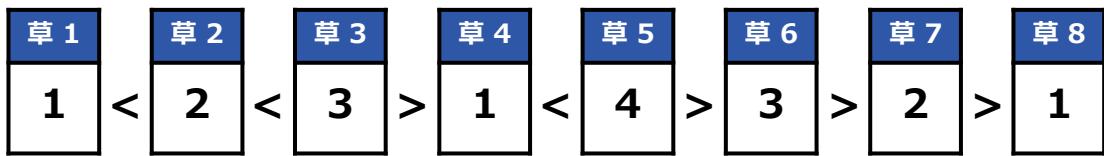
第一に、条件「<」から生まれる下限値を計算すると、以下のようになります。たとえば草 2 は草 1 より高くなければならないので、絶対に 2cm 以上であると分かります。また、草 3 は草 2 より高くなければならないので、絶対に 3cm 以上であると分かります。



第二に、条件「>」から生まれる下限値を計算すると、以下のようになります。たとえば草 7 は草 8 より高くなければならないので、絶対に 2cm 以上であると分かります。また、草 6 は草 7 より高くなければならないので、絶対に 3cm 以上であると分かります。



ここまで 2 つをまとめると、草の高さは下図の値以上であることが分かります（2 つの上限値の max をとっています）。



それでは、草の高さが [1, 2, 3, 1, 4, 3, 2, 1] の場合は条件を満たすのでしょうか。答えは Yes です。そのため、「草の高さの合計としてあり得る最小値」は、 $1+2+3+1+4+3+2+1=17$  であると分かります。

## ◆ 一般のケースを考える

一般のケースでも同様に下限値を求めてみましょう。

- 「<」の条件を考えたときの草  $i$  の高さの下限値を  $\text{ret1}[i]$
- 「>」の条件を考えたときの草  $i$  の高さの下限値を  $\text{ret2}[i]$

とするとき、それぞれの値は以下のようにして計算することができます。

```
1 // 「<」を考えたときの下限値を求める
2 int streak1 = 1; ret1[0] = 1; // streak1 は「何個 A が連続したか」 + 1
3 for (int i = 0; i < N - 1; i++) {
4     if (S[i] == 'A') streak1 += 1;
5     if (S[i] == 'B') streak1 = 1;
6     ret1[i + 1] = streak1;
7 }
8
9 // 「>」を考えたときの下限値を求める
10 int streak2 = 1; ret2[N - 1] = 1; // streak2 は「何個 B が連続したか」 + 1
11 for (int i = N - 2; i >= 0; i--) {
12     if (S[i] == 'B') streak2 += 1;
13     if (S[i] == 'A') streak2 = 1;
14     ret2[i] = streak2;
15 }
```

それでは、草  $i$  の高さが  $\max(\text{ret1}[i], \text{ret2}[i])$  のとき、すべての条件を満たすのでしょうか。証明は省略しますが※5、答えは Yes です。そのため、次ページの解答例のような実装により、正解を出すことができます。

※5 「<」と「>」の境界で分けて考えることが、証明のためのヒントになります。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <string>
3 #include <algorithm>
4 using namespace std;
5
6 int N, ret1[1 << 18], ret2[1 << 18];
7 string S;
8
9 int main() {
10     // 入力
11     cin >> N >> S;
12
13     // 答えを求める
14     int streak1 = 1; ret1[0] = 1; // streak1 は「何個 A が連続したか」 + 1
15     for (int i = 0; i < N - 1; i++) {
16         if (S[i] == 'A') streak1 += 1;
17         if (S[i] == 'B') streak1 = 1;
18         ret1[i + 1] = streak1;
19     }
20     int streak2 = 1; ret2[N - 1] = 1; // streak2 は「何個 B が連続したか」 + 1
21     for (int i = N - 2; i >= 0; i--) {
22         if (S[i] == 'B') streak2 += 1;
23         if (S[i] == 'A') streak2 = 1;
24         ret2[i] = streak2;
25     }
26
27     // 出力
28     long long Answer = 0;
29     for (int i = 0; i < N; i++) Answer += max(ret1[i], ret2[i]);
30     cout << Answer << endl;
31     return 0;
32 }
```

※Python のコードはサポートページをご覧ください

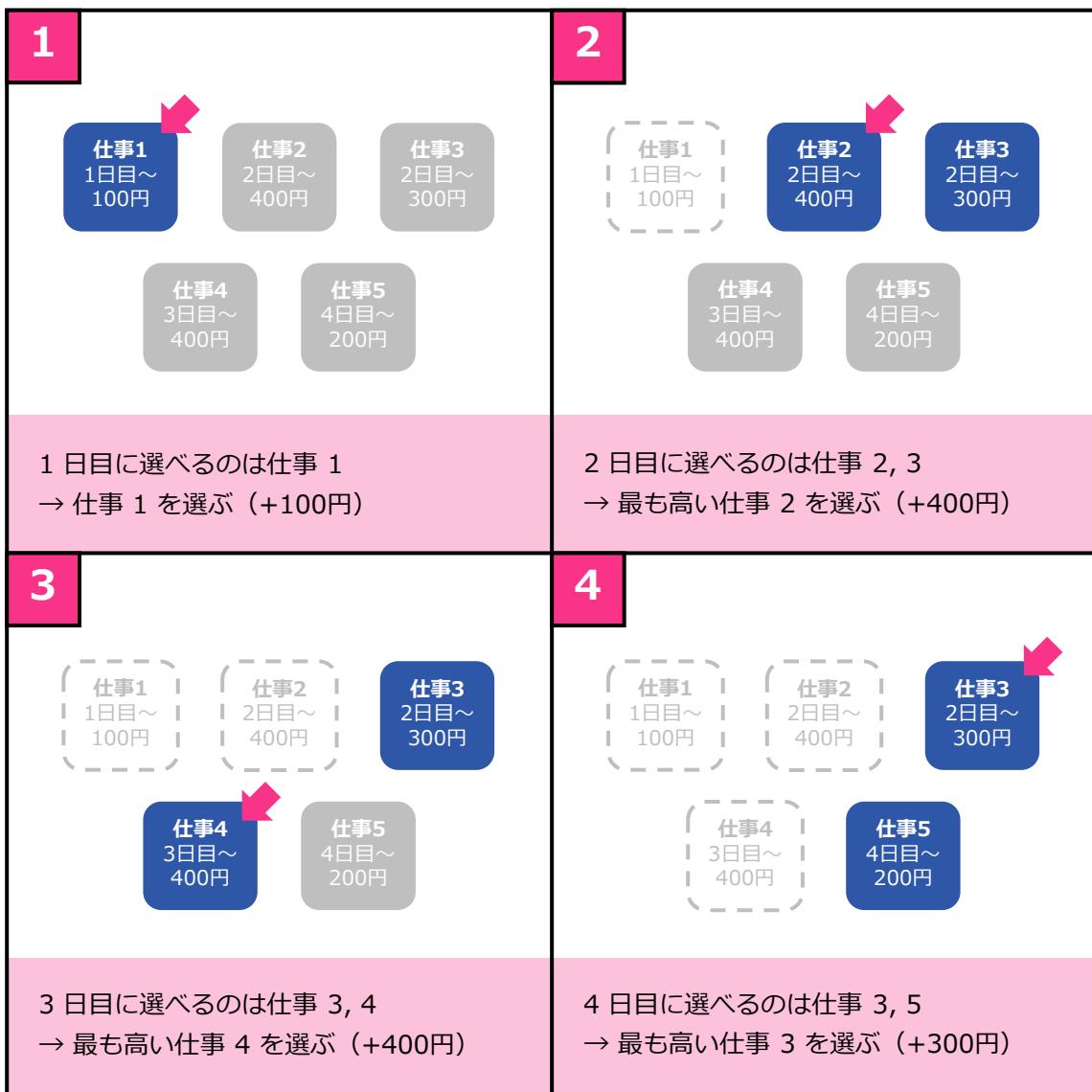
# 6.4

## 問題 B39 : Taro' s Job (難易度 : ★3相当)

実は次のような貪欲法を使うことで、最も多くの金額を稼ぐことができます。

1日目から順番に考えていく。それぞれの日について「今選べる仕事の中で最も給料の高いもの」を選ぶ。

たとえば  $N = 5, D = 4, (X_i, Y_i) = (1, 100), (2, 400), (2, 300), (3, 400), (4, 200)$  の場合、下図のようにして最大値 1200 円を稼ぐことができます。



## ◆ なぜ貪欲法が上手くいく？

それでは、なぜ「一番給料が高いものを選び続ける」という非常に直感的な方法が最適になるのでしょうか。少し難しいですが、以下のようにして証明することができます。

まず、「一番給料が高い仕事を選び続ける」以外の選び方をした場合、一番給料が高い仕事を選ばなかった“最初の日”が存在するので、これを  $d$  日目とする。

また、 $d$  日目に選んだ仕事の番号を  $a$  とし、その日に選べる最も給料の高い仕事の番号を  $b$  とする（ここで  $Y_b > Y_a$  が成り立つ）。

このとき、次の 2 つのことから、もし  $d$  日目に仕事  $b$  を選んでも、絶対に「稼げる合計金額」は減らないといえる。

- もし  $d + 1$  日目以降に仕事  $b$  を選んでいる場合、仕事  $a, b$  のやる日を交換しても合計金額は変わらない。
- もし  $d + 1$  日目以降に仕事  $b$  を選んでいない場合、 $d$  日目に仕事  $a$  の代わりに仕事  $b$  をやると合計金額が増える。

したがって、すべての日について「一番給料が高い仕事」を選択しても損することは絶対にない。

2 日目は ¥50K の  
仕事が一番高い！

1日目	2日目	3日目	4日目
¥40K 1日目～	¥30K 2日目～	¥20K 3日目～	¥50K 2日目～

選んで  
ない → ¥10K, 4日目～

↓  
2, 4 日目を交換しても  
合計金額は変わらない

1日目	2日目	3日目	4日目
¥40K 1日目～	¥50K 2日目～	¥20K 3日目～	¥30K 2日目～

選んで  
ない → ¥10K, 4日目～

2 日目は ¥50K の  
仕事が一番高い！

1日目	2日目	3日目	4日目
¥40K 1日目～	¥30K 2日目～	¥20K 3日目～	¥10K 4日目～

選んで  
ない → ¥50K, 2日目～

↓  
2 日目を ¥50K に変えると  
合計金額が増える

1日目	2日目	3日目	4日目
¥40K 1日目～	¥50K 2日目～	¥20K 3日目～	¥10K 4日目～

選んで  
ない → ¥30K, 2日目～

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int N, D;
5 int X[2009], Y[2009];
6 bool used[2009]; // used[i] は仕事 i を選んだかどうか
7 int Answer = 0;
8
9 int main() {
10     // 入力
11     cin >> N >> D;
12     for (int i = 1; i <= N; i++) cin >> X[i] >> Y[i];
13
14     // 答えを求める
15     for (int i = 1; i <= D; i++) {
16         int maxValue = 0; // 給料の最大値
17         int maxID = -1; // 給料が最大となる仕事の番号
18         for (int j = 1; j <= N; j++) {
19             if (used[j] == true) continue;
20             if (maxValue < Y[j] && X[j] <= i) {
21                 maxValue = Y[j];
22                 maxID = j;
23             }
24         }
25
26         // 選べる仕事がある場合
27         if (maxID != -1) {
28             Answer += maxValue;
29             used[maxID] = true;
30         }
31     }
32
33     // 出力
34     cout << Answer << endl;
35
36     return 0;
37 }
```

※Python のコードはサポートページをご覧ください

## 6.5

## 問題 B40 : Divide by 100

(難易度：★3相当)

まず、 $A_x + A_y$  の値が 100 の倍数になる条件を漏れなく列挙すると、以下のようになります。最初の 2 つ以外は逆順の場合 ( $A_x = 99, A_y = 1$  など) も含まれることに注意してください。

- (100 で割った余りが 0) + (100 で割った余りが 0) の形
- (100 で割った余りが 50) + (100 で割った余りが 50) の形
- (100 で割った余りが 1) + (100 で割った余りが 99) の形
- (100 で割った余りが 2) + (100 で割った余りが 98) の形
- (100 で割った余りが 3) + (100 で割った余りが 97) の形
- (100 で割った余りが 4) + (100 で割った余りが 96) の形
- ：
- (100 で割った余りが 49) + (100 で割った余りが 51) の形

そこで、 $A_i$  を 100 で割った余りが  $p$  となるような  $i$  の個数を  $\text{cnt}[p]$  とするとき、各条件を満たす組  $(x, y)$  の個数は次表の通りになります。

条件	組 $(x, y)$ の個数
0+0	$\text{cnt}[0] * (\text{cnt}[0]-1)/2$
50+50	$\text{cnt}[50] * (\text{cnt}[50]-1)/2$
1+99	$\text{cnt}[1] * \text{cnt}[99]$
2+98	$\text{cnt}[2] * \text{cnt}[98]$
3+97	$\text{cnt}[3] * \text{cnt}[97]$
4+96	$\text{cnt}[4] * \text{cnt}[96]$
：	：
49+51	$\text{cnt}[49] * \text{cnt}[51]$

この部分はそれぞれ  
 $\text{cnt}[0]C_2$  個、 $\text{cnt}[50]C_2$  個  
 になっている

これらを合計した値が答えですので、次ページの解答例に示すような実装をすれば良いです。計算回数は  $N + 100$  回程度と高速です。

## 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 long long N, A[200009];
5 long long cnt[109];
6 long long Answer = 0;
7
8 int main() {
9     // 入力
10    cin >> N;
11    for (int i = 1; i <= N; i++) cin >> A[i];
12
13    // 個数を数える
14    for (int i = 0; i < 100; i++) cnt[i] = 0;
15    for (int i = 1; i <= N; i++) cnt[A[i] % 100] += 1;
16
17    // 答えを求める
18    for (int i = 1; i < 50; i++) Answer += cnt[i] * cnt[100 - i];
19    Answer += cnt[0] * (cnt[0] - 1LL) / 2LL;
20    Answer += cnt[50] * (cnt[50] - 1LL) / 2LL;
21
22    // 出力
23    cout << Answer << endl;
24    return 0;
25 }
```

※Python のコードはサポートページをご覧ください

# 6.6

## 問題 B41 : Reverse of Euclid (難易度 : ★3相当)

この問題を解く重要なポイントは、**最後の操作から順番に考えていく**ことです。まず、最後の一手中は以下のようになります。

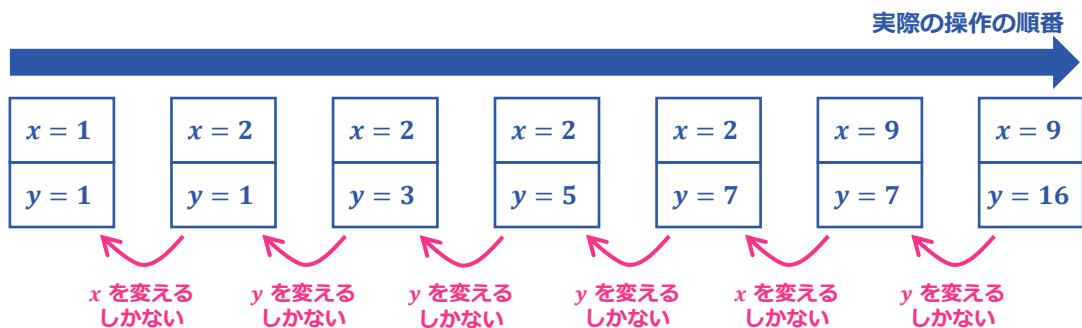
$X < Y$  の場合、最後の一手中として「 $x$  の値を  $x + y$  に変更する」という操作を選べない。なぜなら  $y$  より  $x$  の方が大きくなるからである。そのため、最後の一手中は残った「 $y$  の値を  $x + y$  に変更する」となる。

$X > Y$  の場合、最後の一手中として「 $y$  の値を  $x + y$  に変更する」という操作を選べない。なぜなら  $x$  より  $y$  の方が大きくなるからである。そのため、最後の一手中は残った「 $x$  の値を  $x + y$  に変更する」となる。

また、最後から 2 手目、3 手目、4 手目、…についても、 $x$  と  $y$  の大小関係によって「行える操作」が一通りに決まるので、最後の一手中のときと同じような方針で操作の列を求めることができます。

### ◆ 具体例を考えよう

たとえば  $(X, Y) = (9, 16)$  の場合、最後の一手中から順番に考えていくと以下のような操作の列が得られます。



### ◆ 補足

逆から考えたときの操作列は、ユークリッドの互除法とほぼ同じであるため、 $(X, Y)$  の最大公約数が 1 のときは  $(x, y) = (1, 1)$  で操作が終わります。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 int X, Y;
7
8 int main() {
9     // 入力
10    cin >> X >> Y;
11
12    // 逆から考えてい<
13    vector<pair<int, int>> Answer;
14    while (X >= 2 || Y >= 2) {
15        Answer.push_back(make_pair(X, Y));
16        if (X > Y) X -= Y;
17        else Y -= X;
18    }
19    reverse(Answer.begin(), Answer.end());
20
21    // 出力
22    cout << Answer.size() << endl;
23    for (int i = 0; i < Answer.size(); i++) {
24        cout << Answer[i].first << " " << Answer[i].second << endl;
25    }
26    return 0;
27 }
```

※Python のコードはサポートページをご覧ください

# 6.7

## 問題 B42 : Two Faced Cards (難易度 : ★5相当)

まず、(表の総和の絶対値)+(裏の総和の絶対値) を最大化する方法としては、以下の 4 つがあります。

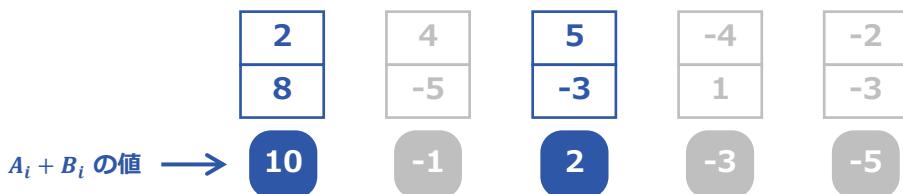
- ・ (表の総和) + (裏の総和) を最大化する。
- ・ (表の総和) - (裏の総和) を最大化する。
- ・ - (表の総和) + (裏の総和) を最大化する。
- ・ - (表の総和) - (裏の総和) を最大化する。

それぞれの場合について、どのようなカードの選び方を選べば最大になるかを考えてみましょう。

### ◆ (表の総和)+(裏の総和) の場合

まず、(表の総和)+(裏の総和) の値は、 **$A_i + B_i > 0$  であるカードをすべて選び、そうでないカードを全く選ばない**ときに最大になります。

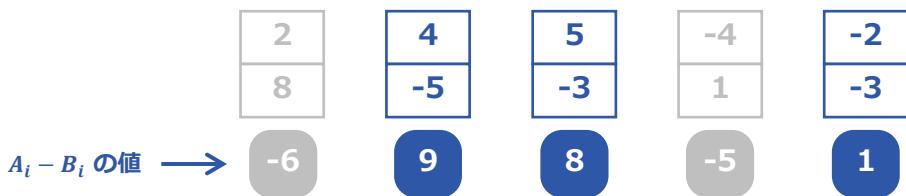
たとえば  $(A_i, B_i) = (2, 8), (4, -5), (5, -3), (-4, 1), (-2, -3)$  であるときは、1 枚目と 3 枚目のカードを選べば良いです。 (表の総和)+(裏の総和) は  $10+2=12$  となります。



### ◆ (表の総和)-(裏の総和) の場合

次に、(表の総和)-(裏の総和) の値は、 **$A_i - B_i > 0$  であるカードをすべて選び、そうでないカードを全く選ばない**ときに最大になります。

たとえば  $(A_i, B_i) = (2, 8), (4, -5), (5, -3), (-4, 1), (-2, -3)$  であるときは、  
 2・3・5 枚目のカードを選べば良いです。 (表の総和)-(裏の総和) は  
 $9+8+1=18$  となります。



### ◆ -(表の総和)+(裏の総和) の場合

次に、-(表の総和)+(裏の総和) の値は、 $-A_i + B_i > 0$  であるカードをすべて選び、そうでないカードを全く選ばないときに最大になります。

たとえば  $(A_i, B_i) = (2, 8), (4, -5), (5, -3), (-4, 1), (-2, -3)$  であるときは、1 枚目と 4 枚目のカードを選べば良いです。 (表の総和)+(裏の総和) は  $6+5=11$  となります。



### ◆ -(表の総和)-(裏の総和) の場合

次に、-(表の総和)-(裏の総和) の値は、 $-A_i - B_i > 0$  であるカードをすべて選び、そうでないカードを全く選ばないときに最大になります。

たとえば  $(A_i, B_i) = (2, 8), (4, -5), (5, -3), (-4, 1), (-2, -3)$  であるときは、  
 2・4・5 枚目のカードを選べば良いです。 (表の総和)+(裏の総和) は  
 $1+3+5=9$  となります。



## ◆ 結局求める答えは？

最後に、求めるべき答えは、4つの方法で得られた値のうち最大のものとなります。たとえば先程挙げた例では  $\max(12, 18, 11, 9) = 18$  です。

このように、「表と裏の目標とする符号」を全探索することで、計算量  $O(N)$  で正しい答えを出すことができます。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 long long N;
6 long long A[100009], B[100009];
7
8 // omote=1 のとき表の総和が正、ura=1 のとき裏の総和が正
9 // omote=2 のとき表の総和が負、ura=2 のとき裏の総和が負
10 long long solve(int omote, int ura) {
11     long long sum = 0;
12     for (int i = 1; i <= N; i++) {
13         long long card1 = A[i]; if (omote == 2) card1 = -A[i];
14         long long card2 = B[i]; if (ura == 2) card2 = -B[i];
15         // カード i は選ぶべきか？
16         if (card1 + card2 >= 0) {
17             sum += (card1 + card2);
18         }
19     }
20     return sum;
21 }
22
23 int main() {
24     // 入力
25     cin >> N;
26     for (int i = 1; i <= N; i++) cin >> A[i] >> B[i];
27
28     // 表の総和の正負と、裏の総和の正負を全探索
29     long long Answer1 = solve(1, 1);
30     long long Answer2 = solve(1, 2);
31     long long Answer3 = solve(2, 1);
32     long long Answer4 = solve(2, 2);
33
34     // 答えを出力
35     cout << max({ Answer1, Answer2, Answer3, Answer4 }) << endl;
36     return 0;
37 }
```

※Python のコードはサポートページをご覧ください

この問題を解くための重要なポイントは、**不正解数を数えること**です※6。生徒  $i$  の不正解数を  $\text{Incorrect}[i]$  をするとき、この生徒の正解数は  $M - \text{Incorrect}[i]$  となるため、もし不正解数さえ分かれば正解数を計算することができます。

## ◆ 不正解数を数えるには？

それでは、不正解数を効率的に数えるにはどうすれば良いのでしょうか。各問題には 1 名しか不正解者がいないので、以下のようなプログラムにより計算量  $O(N + M)$  で全生徒の不正解数が分かれます。

```
1 for (int i = 1; i <= N; i++) Incorrect[i] = 0;
2 for (int i = 1; i <= M; i++) Incorrect[A[i]] += 1;
```

最後に、入力や出力などを含めた全体のプログラムを以下に示します。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int N, M;
5 int A[200009];
6 int Incorrect[200009];
7
8 int main() {
9     // 入力
10    cin >> N >> M;
11    for (int i = 1; i <= M; i++) cin >> A[i];
12
13    // 不正解数を求める
14    for (int i = 1; i <= N; i++) Incorrect[i] = 0;
15    for (int i = 1; i <= M; i++) Incorrect[A[i]] += 1;
16
17    // 答えを出力
18    for (int i = 1; i <= N; i++) cout << M - Incorrect[i] << endl;
19    return 0;
20 }
```

※Python のコードはサポートページをご覧ください

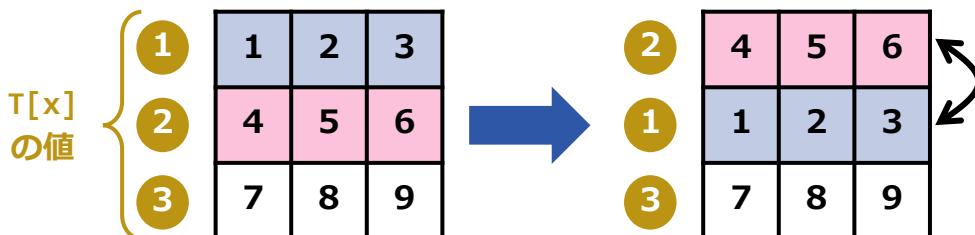
※6 なぜ「不正解数を数える」という発想になるのか：不正解は 1 名しかないので、その方が数えるのが速いからです。テストで最下位に近い成績を取ったとき、下から数える方が速いのと同じです

まず考えられる方法は、交換操作を直接行うことです。1回の操作で $2N$ 個のマスが変化するため、1回当たりの計算量は $O(N)$ となり、全体の計算量は $O(NQ)$ です。C++などの高速なプログラミング言語であれば、この解法でも十分間に合います。しかし、もう少し良い解法はあるのでしょうか。

## ◆ 工夫した解法

そこで、マス目の代わりに現在の $x$ 行目が元々の何行目であるか $T[x]$ を管理すると、より高速に解くことができます。まず $x$ 行目と $y$ 行目を入れ替える交換操作は、`swap(T[x], T[y])`の一発で処理することができます。

また、上から $x$ 行目・左から $y$ 列目の値を答える取得操作では、`A[T[x]][y]`を出力すれば良いです（`A`は元々のマス目を表す二次元配列）。



この解法を実装すると以下の解答例のようになります。計算量は $O(N + Q)$ と非常に高速です。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 using namespace std;
3
4 int N, A[509][509];
5 int Q, QueryType[200009], x[200009], y[200009];
6 int T[509];
7
8 int main() {
9     // 入力
10    cin >> N;

```

```
11  for (int i = 1; i <= N; i++) {
12      for (int j = 1; j <= N; j++) cin >> A[i][j];
13  }
14
15 // 配列 T を初期化
16 for (int i = 1; i <= N; i++) T[i] = i;
17
18 // クエリの処理
19 cin >> Q;
20 for (int i = 1; i <= Q; i++) {
21     cin >> QueryType[i] >> x[i] >> y[i];
22     if (QueryType[i] == 1) {
23         swap(T[x[i]], T[y[i]]);
24     }
25     if (QueryType[i] == 2) {
26         cout << A[T[x[i]]][y[i]] << endl;
27     }
28 }
29 return 0;
30 }
```

※Python のコードはサポートページをご覧ください

6.10

## 問題 B45 : Blackboard 2

(難易度 : ★2相当)

まず、1回の操作では  **$a + b + c$  の値が変わりません。**なぜなら、操作によって1つの整数が $+1$ され、別の1つの整数が $-1$ されるため、合計して $\pm 0$ となるからです。そのため、 $a + b + c$ の値が $0$ でなければ、答えは絶対にNoです。

### ◆ $a + b + c = 0$ の場合

逆に、 $a + b + c = 0$ の場合は必ず Yes になります。これは次のようにして証明することができます。

$a + b + c = 0$ の場合、「正の整数の絶対値の合計  $s_+$ 」と「負の整数の絶対値合計  $s_-$ 」は必ず一致する。たとえば  $(a, b, c) = (13, 7, -20)$ の場合、 $s_+ = s_- = 20$ となり確かに一致する。

そのため、「正の整数を一つ選んで $+1$ し、負の整数を一つ選んで $-1$ する」という操作を  $s_+$ 回行えば、必ず3つすべての整数が $0$ になる。

### ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 long long a, b, c;
5
6 int main() {
7     // 入力
8     cin >> a >> b >> c;
9
10    // 出力
11    if (a + b + c == 0) cout << "Yes" << endl;
12    else cout << "No" << endl;
13    return 0;
14 }
```

※Pythonのコードはサポートページをご覧ください

# 第8章

## データ構造とクエリ処理

応用問題 8.1	· · · · ·	96
応用問題 8.2	· · · · ·	99
応用問題 8.3	· · · · ·	102
応用問題 8.4	· · · · ·	105
応用問題 8.5	· · · · ·	106
応用問題 8.6	· · · · ·	108
応用問題 8.7	· · · · ·	111
応用問題 8.8	· · · · ·	113
応用問題 8.9	· · · · ·	116

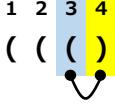
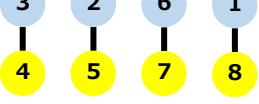
# 8.1

## 問題 B51 : Bracket

(難易度：★4相当)

※この問題は例題と比べて相当難しいです。

まず、カッコ列  $((())())$  について対応関係を列挙することを考えましょう。左から順番に調べていくと、以下のようにして 4 つの対応関係（3-4 文字目・2-5 文字目・6-7 文字目・1-8 文字目）が分かります。

1  1 文字目は '(' である	2  2 文字目は '(' である	3  3 文字目は '(' である
4  4 文字目は ')' である 残っている一番右の '(' の 3 文字目と対応させる	5  5 文字目は ')' である 残っている一番右の '(' の 2 文字目と対応させる	6  6 文字目は '(' である
7  7 文字目は ')' である 残っている一番右の '(' の 6 文字目と対応させる	8  8 文字目は ')' である 残っている一番右の '(' の 1 文字目と対応させる	9  これですべての対応が わかった！

## ◆ どう実装するか

先程の例で説明したように、カッコ列の対応関係は、「残っている一番右の「(」を調べることによって列挙できます。

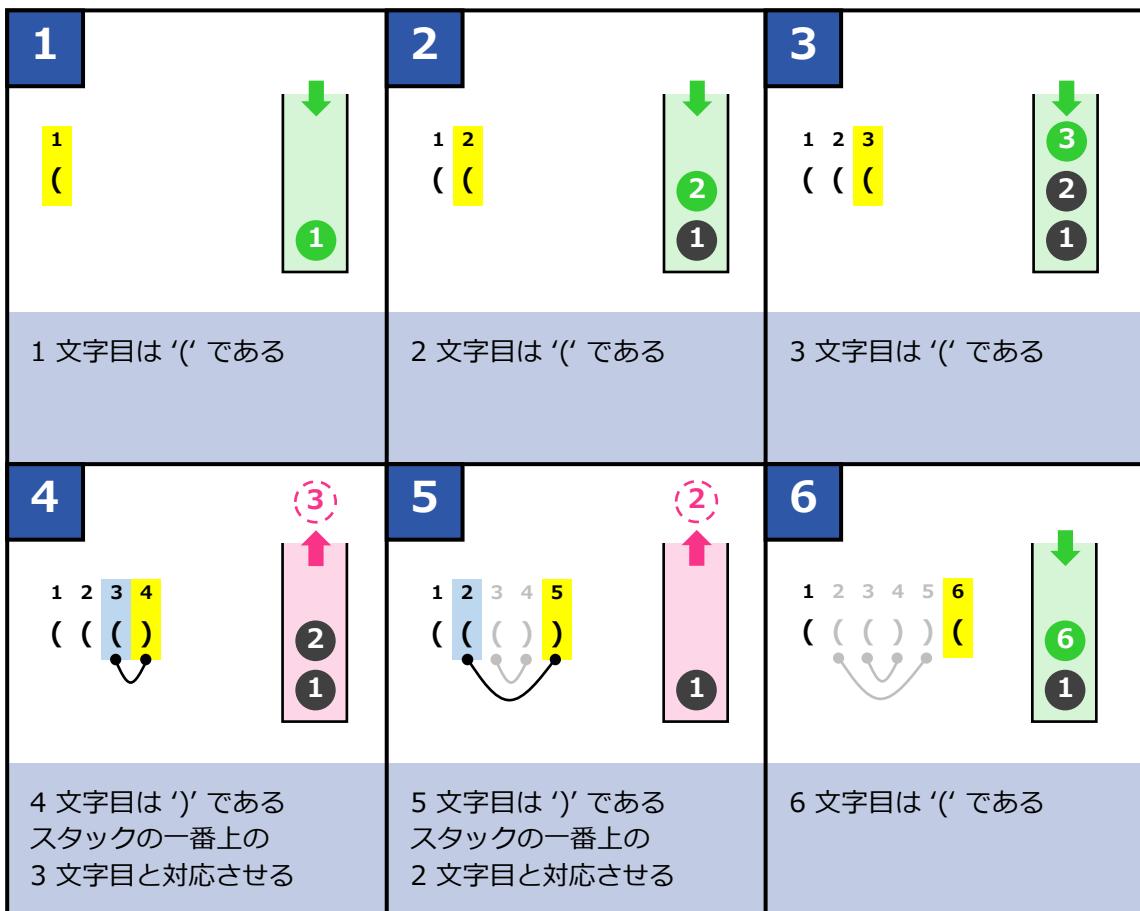
それでは、残っている一番右の「(」はどうやって効率的に調べられるのでしょうか。もちろん直接調べても良いですが、1 文字当たり計算量  $O(N)$ 、すなわち全体で計算量  $O(N^2)$  を要し、実行時間制限に間に合いません。

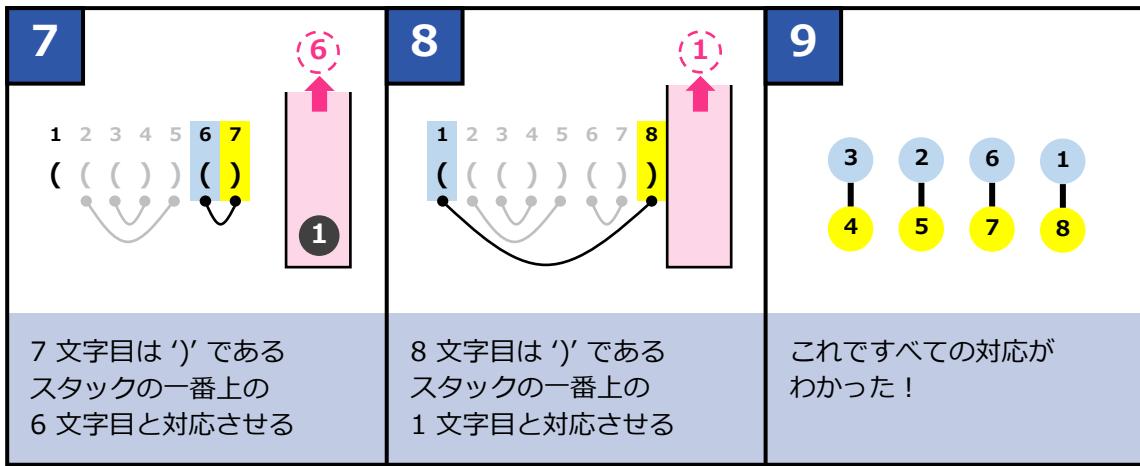
そこで、スタックに「今残っている「(」の位置」を左から順に記録すると効率的です。具体的なアルゴリズムの流れは以下の通りです。

左から順番に一文字ずつ読んでいき、次のことを行う。

- *i* 文字目が「(」のとき：スタックに *i* を追加する
- *i* 文字目が「)」のとき：スタックの一番上と *i* 文字目が対応することが分かる。その後、スタックの一番上の要素を消す。

たとえば、カッコ列が  $((())())$  の場合、アルゴリズムの挙動は以下のようになります（最初に説明したものと同じ例です）。





ここまででの内容を実装すると、以下の解答例のようになります。計算量は  $O(N)$  です。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <stack>
3 using namespace std;
4
5 int main() {
6     // 入力
7     string S;
8     cin >> S;
9
10    // 左から順番に見ていく
11    // 文字列は 0 文字目から始まることに注意
12    stack<int> Stack;
13    for (int i = 0; i < S.size(); i++) {
14        if (S[i] == '(') {
15            Stack.push(i + 1);
16        }
17        if (S[i] == ')') {
18            cout << Stack.top() << " " << i + 1 << endl;
19            Stack.pop();
20        }
21    }
22    return 0;
23 }
```

※Python のコードはサポートページをご覧ください

この問題は、問題文の通りに直接シミュレーションすることで解くことができます。実装例は以下のようになります。計算量は  $O(N)$  です。

## ◆ 解答例 (C++)

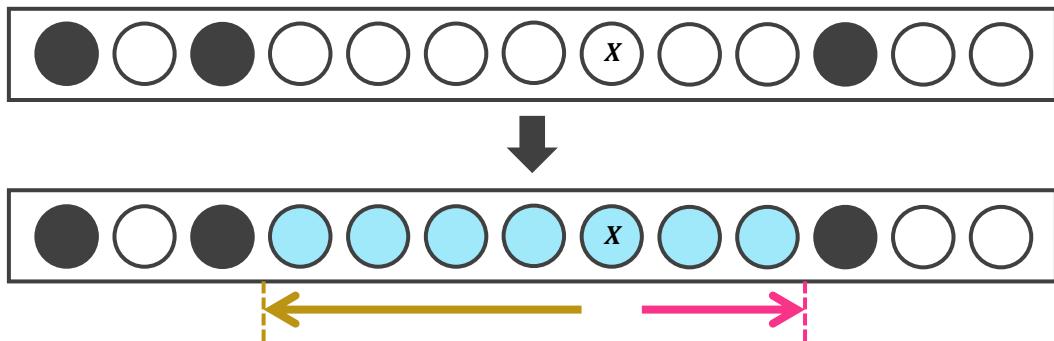
```
1 #include <iostream>
2 #include <queue>
3 using namespace std;
4
5 int N, X;
6 char A[100009];
7 queue<int> Q;
8
9 int main() {
10     // 入力
11     cin >> N >> X;
12     for (int i = 1; i <= N; i++) cin >> A[i];
13
14     // シミュレーション
15     Q.push(X); A[X] = '@';
16     while (!Q.empty()) {
17         int pos = Q.front(); Q.pop();
18         if (pos - 1 >= 1 && A[pos - 1] == '.') {
19             A[pos - 1] = '@';
20             Q.push(pos - 1);
21         }
22         if (pos + 1 <= N && A[pos + 1] == '.') {
23             A[pos + 1] = '@';
24             Q.push(pos + 1);
25         }
26     }
27
28     // 出力
29     for (int i = 1; i <= N; i++) cout << A[i];
30     cout << endl;
31     return 0;
32 }
```

※Python のコードはサポートページをご覧ください

(解説は次ページへ続く)

## ◆ どの部分が青く塗られるか？

それでは、このシミュレーションではどの部分が青く塗られるのでしょうか。実は、**ボール X から左に進んだときに黒にぶつかるまでの領域と、ボール X から右に進んだときに黒にぶつかるまでの領域**だけが、青く塗られます。具体例を以下に示します。



なぜなら、まずボール X がキューに追加され、ボール X と隣り合うボールがキューに追加され、それと隣り合うボールがキューに追加され、… といったように、**黒い“壁”にぶつかるまで連鎖的に「キューに追加される領域」が拡大していく**からです。具体例を以下に示します。

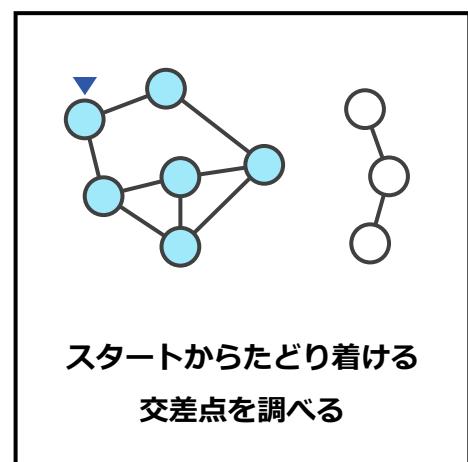
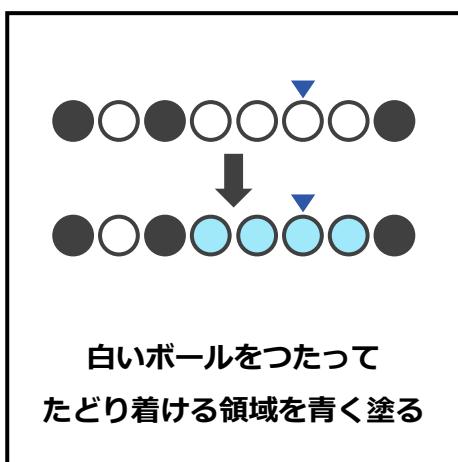
<b>1</b>		<b>2</b>	
ボール 4 をキューに追加		ボール 4 と隣接する 3, 5 をキューに追加	
<b>3</b>		<b>4</b>	
ボール 3 と隣接する 2 をキューに追加		ボール 5 と隣接する白色のボールはない	

## ◆ 幅優先探索との関連

今回行ったシミュレーションは、9.3 節で学ぶ「幅優先探索」と非常によく似ています。幅優先探索を使うと、

- ・ 道路網で、あるスタート地点からたどり着ける交差点はどこか
- ・ 最小何本の道路を通ることで、スタートからゴールへたどり着けるか

などの様々な実用的な問題を解くことができます。興味のある方は、ぜひ本の 357~361 ページをご覧ください。



問題設定もよく似ています

応用問題 6.4 の解説に書いた通り、この問題は「今選べる中で最も給料の高い仕事を選び続ける」という貪欲法で解くことができます。

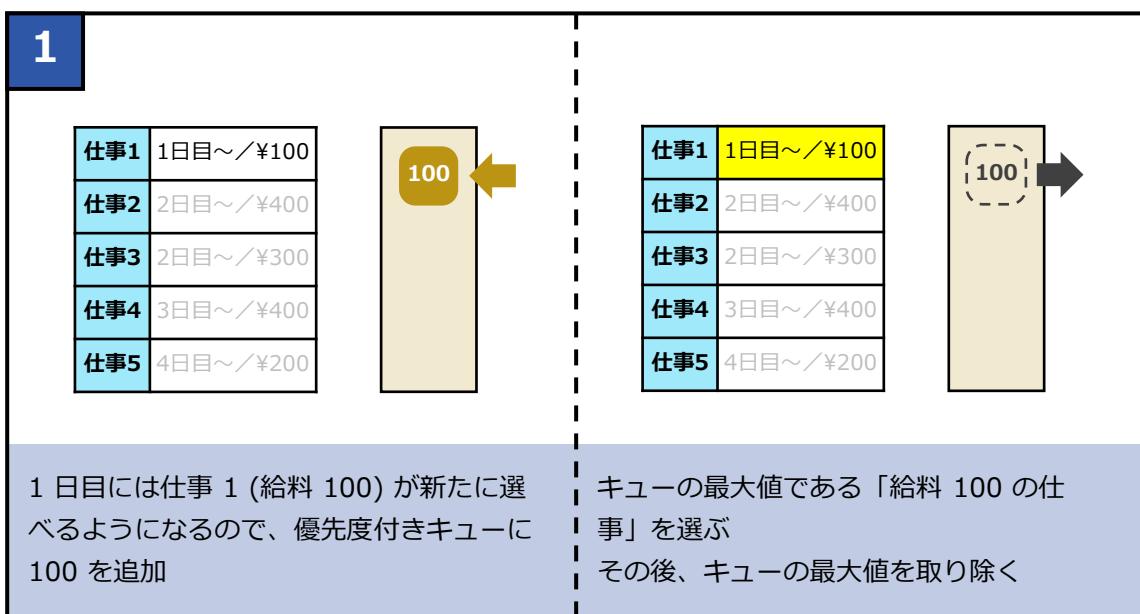
しかし、この貪欲法を自然に実装すると、1 日の仕事を選ぶのに計算量  $O(N)$  かかります。日数は  $D$  日なので、全体の計算量は  $O(ND)$  となり、残念ながら実行時間制限に間に合いません。

## ◆ 工夫した解法

そこで、1 日の仕事を計算量  $O(\log N)$  で選べるようにするために、「今選べる仕事の給料のリスト」を優先度付きキューで管理することを考えます。

たとえば、 $N = 5, D = 4, (X_i, Y_i) = (1, 100), (2, 400), (2, 300), (3, 400), (4, 200)$  の場合は以下のようになります。

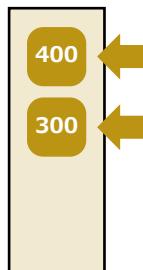
なお、使う優先度付きキューは、最大の要素を取り出すものであることに注意してください（つまり取り出す要素が“選ぶべき仕事”になります）。



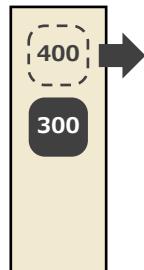
※表では、選べる仕事を黒字で、選べない仕事を灰色で示しています

2

仕事1	完了
仕事2	2日目～／¥400
仕事3	2日目～／¥300
仕事4	3日目～／¥400
仕事5	4日目～／¥200



仕事1	完了
仕事2	2日目～／¥400
仕事3	2日目～／¥300
仕事4	3日目～／¥400
仕事5	4日目～／¥200

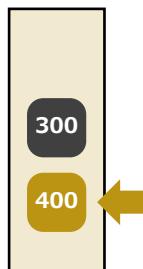


2日目には仕事2(給料400)と仕事3(給料300)が新たに選べるようになるので、優先度付きキューに400, 300を追加

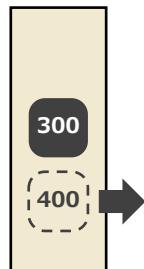
キューの最大値である「給料400の仕事」を選ぶ  
その後、キューの最大値を取り除く

3

仕事1	完了
仕事2	完了
仕事3	2日目～／¥300
仕事4	3日目～／¥400
仕事5	4日目～／¥200



仕事1	完了
仕事2	完了
仕事3	2日目～／¥300
仕事4	3日目～／¥400
仕事5	4日目～／¥200

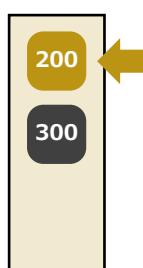


3日目には仕事4(給料400)が新たに選べるようになるので、優先度付きキューに400を追加

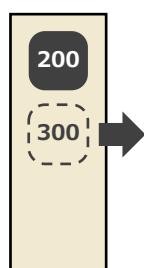
キューの最大値である「給料400の仕事」を選ぶ  
その後、キューの最大値を取り除く

4

仕事1	完了
仕事2	完了
仕事3	2日目～／¥300
仕事4	完了
仕事5	4日目～／¥200



仕事1	完了
仕事2	完了
仕事3	2日目～／¥300
仕事4	完了
仕事5	4日目～／¥200



4日目には仕事5(給料200)が新たに選べるようになるので、優先度付きキューに200を追加

キューの最大値である「給料300の仕事」を選ぶ  
その後、キューの最大値を取り除く

## ◆ 計算量について

最後に、計算量はどれくらいになるのでしょうか。優先度付きキューへの追加は仕事の個数と同じ  $N$  回行いますが、削除は日数と同じ  $D$  回行うので、優先度付きキューに対して行う操作の回数は合計  $N + D$  回となります。

したがって、アルゴリズム全体の計算量は  $O((N + D) \log N)$  です。以下に実装例を示します。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <vector>
3 #include <queue>
4 using namespace std;
5
6 long long N, D;
7 long long X[200009], Y[200009];
8 vector<long long> G[200009]; // G[i] は i 日目から始まる仕事の給料のリスト
9 long long Answer = 0;
10
11 int main() {
12     // 入力
13     cin >> N >> D;
14     for (int i = 1; i <= N; i++) {
15         cin >> X[i] >> Y[i];
16         G[X[i]].push_back(Y[i]);
17     }
18
19     // 答えを求める
20     priority_queue<long long> Q;
21     for (int i = 1; i <= D; i++) {
22         // i 日目から始まる仕事をキューに追加
23         for (int j : G[i]) Q.push(j);
24
25         // やる仕事を選択し、その仕事をキューから削除する
26         if (!Q.empty()) {
27             Answer += Q.top();
28             Q.pop();
29         }
30     }
31
32     // 出力
33     cout << Answer << endl;
34     return 0;
35 }
```

※Python のコードはサポートページをご覧ください

# 8.4

## 問題 B54 : Counting Same Values (難易度：★2相当)

この問題は、以下のようなアルゴリズムで解くことができます。計算量は  $O(N \log N)$  です。

各要素が現時点で何回出現したかを管理する連想配列 Map (初期値 0) を用意し、 $i = 1, 2, \dots, N$  の順に次の処理を行う：

- 手順1：答え Answer に  $\text{Map}[X[i]]$  を加算する
- 手順2： $\text{Map}[X[i]]$  に 1 を加算する

このアルゴリズムが上手くいく理由は、手順1で加算される  $\text{Map}[X[i]]$  の値が、 $X_j = X_i (j < i)$  を満たす  $j$  の個数であるからです。

### ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <map>
3 using namespace std;
4
5 int N, A[100009];
6 map<int, int> Map;
7
8 int main() {
9     // 入力
10    cin >> N;
11    for (int i = 1; i <= N; i++) cin >> A[i];
12
13    // 答えを求める
14    long long Answer = 0;
15    for (int i = 1; i <= N; i++) {
16        Answer += Map[A[i]];
17        Map[A[i]] += 1;
18    }
19
20    // 出力
21    cout << Answer << endl;
22    return 0;
23 }
```

※Python のコードはサポートページをご覧ください

まず、「差の最小値」を求めるクエリ 2 に答えるためには、以下の 2 つの値が分かっている必要があります。

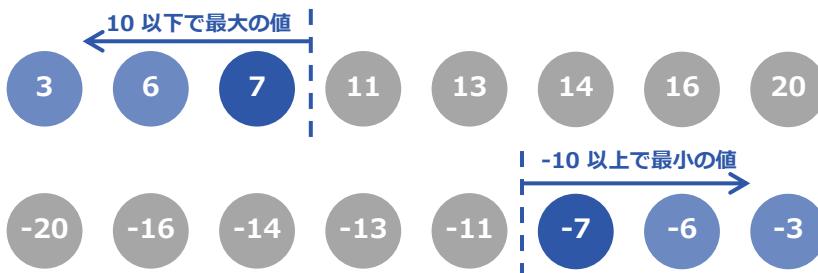
- 机にある  $x$  以下のカードのうち、最大の値  $v1$
- 机にある  $x$  以上のカードのうち、最小の値  $v2$

ここで、 $v2$  の値は例題 A55 (8.5 節) で学んだ通り、`lower_bound` 関数を使えば一発で求められます。しかし、 $v1$  についてはどうでしょうか。

## ◆ $v1$ を求める方法

解決策の一つとして<sup>※1</sup>、**「カードの値×(-1)」を記録した set を用意する**方法があります。

この方法を使った場合、**set における「 $-x$  以上で最小の値」が「 $x$  以下で最大の値」に対応する**ので、 $v2$  の値と同じようにして  $v1$  の値を求めることができます。実装例を以下に示します。



## ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <set>
3 #include <algorithm>
4 using namespace std;
5
6 long long N, QueryType[100009], x[100009];
7 set<long long> Set1, Set2;

```

<sup>※1</sup> 他にも、`itr++` や `itr-` などの高度な機能（本の 312 ページで解説）を使う手段もあります

```

8 // r 以下の最大値を返す
9 long long GetDown(long long r) {
10    auto itr = Set2.lower_bound(-r);
11
12    // r 以下のものが存在しない場合、非常に小さい値を返す
13    if (itr == Set2.end()) return -10000000000000LL;
14
15    // 存在する場合
16    return -(*itr);
17 }
18
19 // r 以上の最小値を返す
20 long long GetUp(long long r) {
21    auto itr = Set1.lower_bound(r);
22
23    // r 以上のものが存在しない場合、非常に大きい値を返す
24    if (itr == Set1.end()) return 10000000000000LL;
25
26    // 存在する場合
27    return (*itr);
28 }
29
30 int main() {
31    // 入力
32    cin >> N;
33    for (int i = 1; i <= N; i++) cin >> QueryType[i] >> x[i];
34
35    // クエリの処理
36    for (int i = 1; i <= N; i++) {
37        if (QueryType[i] == 1) {
38            Set1.insert(x[i]);
39            Set2.insert(-x[i]);
40        }
41        if (QueryType[i] == 2) {
42            long long v1 = GetDown(x[i]);
43            long long v2 = GetUp(x[i]);
44            long long Answer = min(x[i] - v1, v2 - x[i]);
45            if (Answer >= 100000000000LL) cout << "-1" << endl;
46            else cout << Answer << endl;
47        }
48    }
49    return 0;
50 }
```

※Python のコードはサポートページをご覧ください

まず、回文である条件は「前から読んでも後ろから読んでも文字列が同じこと」です。

そのため、もし  $S[l, r]$  を前から順に読んだときのハッシュ値と、 $S[l, r]$  を後ろから読んだときのハッシュ値が同じである場合、 $S[l, r]$  はほぼ 100% 回文であると分かります。



## ◆ 後ろからのハッシュ値はどう求める？

そこで、文字列  $S$  の長さを  $N$  とし、文字列  $S$  を逆順にした文字列を  $S'$  とするとき、以下の 2 つの値は一致します。

- $S[l, r]$  を後ろから読んだときのハッシュ値
- $S'[N - r + 1, N - l + 1]$  を前から読んだときのハッシュ値



したがって、以下のようなプログラムにより、回文かどうかを判定するクエリを処理することができます。

なお、プログラム中の関数 `GetHashLeft(l, r)` は、 $S[l, r]$  を前から読んだときのハッシュ値を返します。また、プログラム中の関数 `GetHashRight(l, r)` は、 $S[l, r]$  を後ろから読んだときのハッシュ値を返します。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <string>
3 #include <algorithm>
4 using namespace std;
5
6 // 入力で与えられる変数など
7 int N, Q, L[100009], R[100009];
8 string S;
9 string SRev; // S の逆順
10
11 // 文字列を数値に変換した値（それぞれ S, SRev に対応）
12 int T[100009];
13 int TRev[100009];
14
15 // ハッシュ値など
16 long long mod = 2147483647;
17 long long Power100[100009];
18 long long H[100009]; // S のハッシュ
19 long long HRev[100009]; // SRev のハッシュ
20
21 // 文字列の l～r 番目を前から読んだ時のハッシュ値を返す関数
22 long long GetHashLeft(int l, int r) {
23     long long val = H[r] - (Power100[r - 1 + 1] * H[l - 1] % mod);
24     if (val < 0) val += mod;
25     return val;
26 }
27
28 // 文字列の l～r 番目を後ろから読んだ時のハッシュ値を返す関数
29 long long GetHashRight(int l, int r) {
30     int true_l = N + 1 - r;
31     int true_r = N + 1 - l;
32     long long val = HRev[true_r] - (Power100[true_r - true_l + 1] * HRev[true_l - 1] % mod);
33     if (val < 0) val += mod;
34     return val;
35 }
36
37 int main() {
38     // 入力
39     cin >> N >> Q;
40     cin >> S;
41     for (int i = 1; i <= Q; i++) cin >> L[i] >> R[i];
```

```

42 SRev = S;
43 reverse(SRev.begin(), SRev.end());
44
45 // S, SRev の文字を数値に変換
46 for (int i = 1; i <= N; i++) T[i] = (int)(S[i - 1] - 'a') + 1;
47 for (int i = 1; i <= N; i++) TRRev[i] = (int)(SRev[i - 1] - 'a') + 1;
48
49 // 100 の n 乗を前計算
50 Power100[0] = 1;
51 for (int i = 1; i <= N; i++) Power100[i] = (100LL * Power100[i - 1]) % mod;
52
53 // S のハッシュ値を前計算
54 H[0] = 1;
55 for (int i = 1; i <= N; i++) H[i] = (100LL * H[i - 1] + T[i]) % mod;
56
57 // SRev のハッシュ値を前計算
58 HRev[0] = 1;
59 for (int i = 1; i <= N; i++) HRev[i] = (100LL * HRev[i - 1] + TRRev[i]) % mod;
60
61 // クエリの処理
62 for (int i = 1; i <= Q; i++) {
63     long long v1 = GetHashLeft(L[i], R[i]);
64     long long v2 = GetHashRight(L[i], R[i]);
65     // 左から読んだ時・右から読んだ時のハッシュ値が一致していれば回文
66     if (v1 == v2) cout << "Yes" << endl;
67     else cout << "No" << endl;
68 }
69 return 0;
70 }
```

※Python のコードはサポートページをご覧ください

この問題も例題 A57 (8.7 節) と同じように、以下の値を前計算する「ダブリング」を使うことで効率的に解けます。

- 整数  $i$  から 1 回操作した後の整数  $dp[0][i]$
- 整数  $i$  から 2 回操作した後の整数  $dp[1][i]$
- 整数  $i$  から 4 回操作した後の整数  $dp[2][i]$
- 整数  $i$  から 8 回操作した後の整数  $dp[3][i]$
- 16, 32, 64,... 回操作した後も同様

それでは、このアイデアはどうやって実装すれば良いのでしょうか。

## ◆ ダブリングの実装 (1)

まずは前計算について考えます。 $2^{d-1}$  回後の  $2^{d-1}$  回後は  $2^d$  回後ですので、前計算の部分は以下のようにして実装できます。

ここで、本問題の制約は  $K \leq 10^9$  であり、 $10^9 < 2^{30}$  ですので、 $dp[29][j]$  までを前計算しておけば十分です。

```

1 for (int i = 1; i <= N; i++) dp[0][i] = i - (i の各桁の和)
2
3 for (int d = 1; d <= 29; d++) {
4     for (int i = 1; i <= N; i++) dp[d][i] = dp[d - 1][dp[d - 1][i]];
5 }
```

## ◆ ダブリングの実装 (2)

次に、各整数に対して「 $K$  回の操作を行った後の値」を求める部分について考えます。

本の 322 ページでも述べた通り、 **$K$  を 2 進法で表したときの  $2^d$  の位が 1 であるときに限り「操作を  $2^d$  回進めること」を行えば、操作を  $K$  回進めることができるため、この部分は次ページのように実装できます。**

```

1 // 答えを求める
2 for (int i = 1; i <= N; i++) {
3     int CurrentNum = i; // 現在の整数
4     for (int d = 29; d >= 0; d--) {
5         if ((K / (1 << d)) % 2 != 0) CurrentNum = dp[d][CurrentNum];
6     }
7     cout << CurrentNum << endl;
8 }
```

最後に、以上の内容をまとめると、解答例のようになります。プログラム全体の計算量は  $O(N \log K)$  です。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int N, K;
6 int dp[32][300009];
7
8 int main() {
9     // 入力
10    cin >> N >> K;
11
12    // 1 回操作した後の値を求める
13    for (int i = 1; i <= N; i++) {
14        string str = to_string(i);
15        dp[0][i] = i;
16        for (int j = 0; j < str.size(); j++) {
17            dp[0][i] -= (int)(str[j] - '0');
18        }
19    }
20
21    // 前計算
22    for (int d = 1; d <= 29; d++) {
23        for (int i = 1; i <= N; i++) dp[d][i] = dp[d - 1][dp[d - 1][i]];
24    }
25
26    // 答えを求める
27    for (int i = 1; i <= N; i++) {
28        int CurrentNum = i; // 現在の整数
29        for (int d = 29; d >= 0; d--) {
30            if ((K / (1 << d)) % 2 != 0) CurrentNum = dp[d][CurrentNum];
31        }
32        cout << CurrentNum << endl;
33    }
34    return 0;
35 }
```

※Python のコードはサポートページをご覧ください

※この問題は例題と比べて相当難しいです。

この問題の解説を見る前に、まずは問題 A76（例題 10.6）を解くことをおすすめします。問題 B58 と非常によく似た問題ですが、難易度は ★4 程度と比較的簡単です。

なお、本を最初から順番に読んでいる方は、10 章まで読破してからこの問題に挑戦することを推奨します。

まず考えられる方法は、スタートからゴールまで移動する方法を全探索することです。しかし、移動方法は最大で  $2^{N-2}$  通りあるため、制約の上限である  $N = 100000$  はもちろん、 $N = 100$  でも絶望的です。

## ◆ 動的計画法を考える

そこで、以下の配列に動的計画法を適用させることを考えます。

$dp[i]$  : 足場 1 から足場  $i$  まで最小何回のジャンプで移動できるか？

まず初期状態は明らかに  $dp[1]=0$  です（スタート地点は足場 1 であるため）。次に状態遷移を考えます。

- $posL : X_{pos} \geq X_i - R$  を満たす最小の pos
- $posR : X_{pos} \leq X_i - L$  を満たす最大の pos

すると、足場  $i$  にたどり着くための最後の行動としては「足場  $posL$ ,  $posL+1$ , …,  $posR$  のいずれかから直接ジャンプする」しかありません。そのため、 $dp[i]$  の値は以下のとおりになります。

$$dp[i] = \min(dp[posL], dp[posL+1], \dots, dp[posR]) + 1$$

したがって、次ページに示すようなプログラムにより、ゴール地点までの最小ジャンプ回数が分かります。計算量は  $O(N^2)$  です。なお、 $posL$  や  $posR$  の値は、配列の二分探索によって求めることができます。

```

1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 int N, L, R, X[100009];
6 int dp[100009];
7
8 int main() {
9     // 入力
10    cin >> N >> L >> R;
11    for (int i = 1; i <= N; i++) cin >> X[i];
12
13    // 動的計画法
14    dp[1] = 0;
15    for (int i = 2; i <= N; i++) {
16        int posL = lower_bound(X + 1, X + N + 1, X[i] - R) - X;
17        int posR = lower_bound(X + 1, X + N + 1, X[i] - L + 1) - X - 1;
18
19        // dp[posL] から dp[posR] までの最大値を求める
20        dp[i] = 1000000000;
21        for (int j = posL; j <= posR; j++) dp[i] = min(dp[i], dp[j] + 1);
22    }
23
24    // 答えを出力
25    cout << dp[N] << endl;
26    return 0;
27 }
```

※Python のコードはサポートページをご覧ください

## ◆ 動的計画法の高速化

先程のプログラムはたしかに正しい答えを出力します。しかし、 $dp[posL]$  から  $dp[posR]$  までの区間の最大値を求める部分がボトルネックとなり、計算量が  $O(N^2)$  になってしまいます。本問題の制約は  $N \leq 100000$  ですので、このままでは実行時間制限に間に合いません。

そこで、 $dp[i]$  の値をセグメント木で管理すると、区間の最大値を計算量  $O(\log N)$  で求めることができます。プログラム全体の計算量が  $O(N \log N)$  まで削減されます。以下に実装例を示します。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 class SegmentTree {
6 public:
7     int dat[300000], siz = 1;
```

```

8 // 要素 dat の初期化を行う（最初は全部ゼロ）
9 void init(int N) {
10    siz = 1;
11    while (siz < N) siz *= 2;
12    for (int i = 1; i < siz * 2; i++) dat[i] = 0;
13 }
14
15 // クエリ 1 に対する処理
16 void update(int pos, int x) {
17    pos = pos + siz - 1;
18    dat[pos] = x;
19    while (pos >= 2) {
20        pos /= 2;
21        dat[pos] = min(dat[pos * 2], dat[pos * 2 + 1]);
22    }
23 }
24
25 // クエリ 2 に対する処理
26 // u は現在のセル番号、[a, b) はセルに対応する半開区間、[l, r) は求めたい半開区間
27 int query(int l, int r, int a, int b, int u) {
28    if (r <= a || b <= l) return 1000000000; // 一切含まれない場合
29    if (l <= a && b <= r) return dat[u]; // 完全に含まれる場合
30    int m = (a + b) / 2;
31    int AnswerL = query(l, r, a, m, u * 2);
32    int AnswerR = query(l, r, m, b, u * 2 + 1);
33    return min(AnswerL, AnswerR);
34 }
35 };
36
37 int N, L, R, X[100009];
38 int dp[100009];
39 SegmentTree Z;
40
41 int main() {
42    // 入力
43    cin >> N >> L >> R;
44    for (int i = 1; i <= N; i++) cin >> X[i];
45
46    // セグメント木の準備
47    Z.init(N);
48    dp[1] = 0;
49    Z.update(1, 0);
50
51    // 動的計画法
52    for (int i = 2; i <= N; i++) {
53        int posL = lower_bound(X + 1, X + N + 1, X[i] - R) - X;
54        int posR = lower_bound(X + 1, X + N + 1, X[i] - L + 1) - X - 1;
55        dp[i] = Z.query(posL, posR + 1, 1, Z.siz + 1, 1) + 1;
56        Z.update(i, dp[i]);
57    }
58
59    // 答えを出力
60    cout << dp[N] << endl;
61    return 0;
62 }

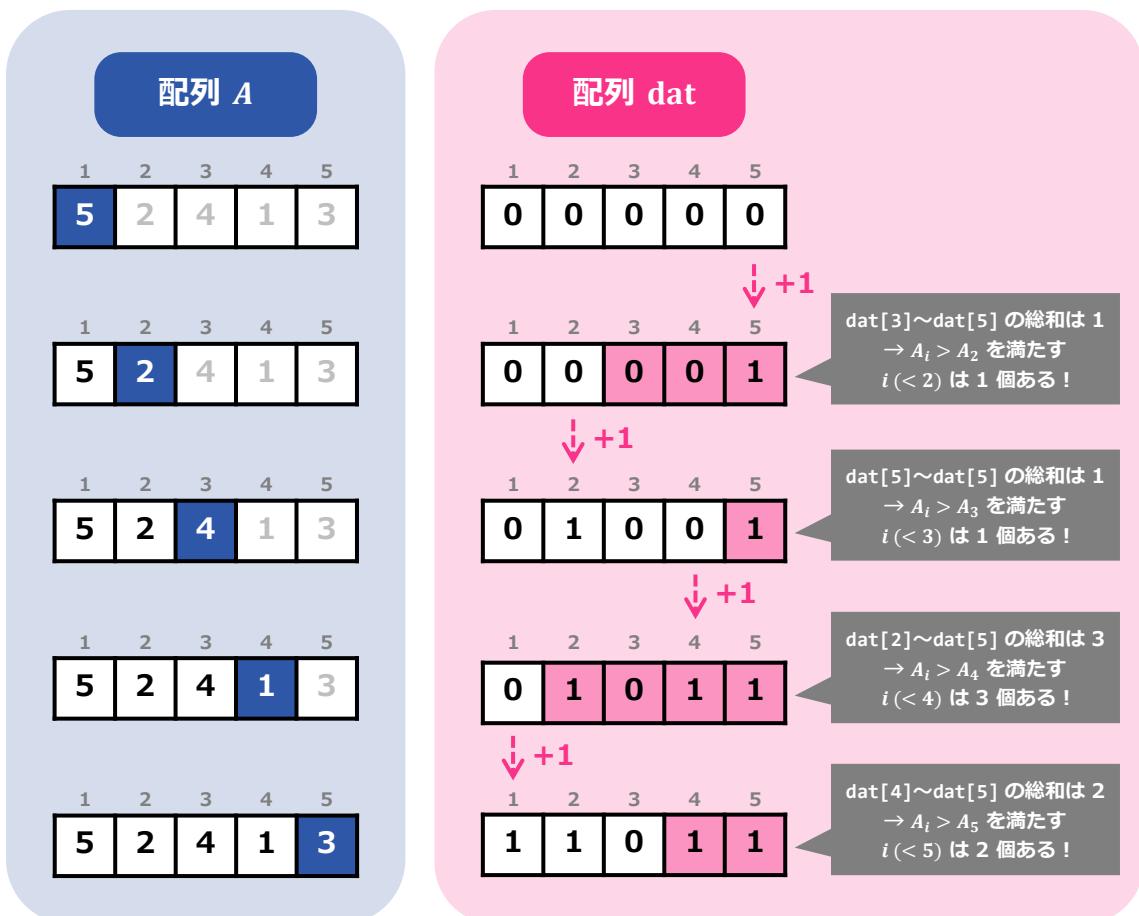
```

※Python のコードはサポートページをご覧ください

この問題を解く最も単純な方法は、すべての組  $(i, j)[1 \leq i < j \leq N]$  について  $A_i > A_j$  かどうかを直接調べることです。しかし、この方法では計算量が  $O(N^2)$  となり、残念ながら実行時間制限に間に合いません。

## ◆ 工夫した解法

そこで、現時点で整数  $x$  は何回出現したかを表す配列  $\text{dat}[x]$  を管理することを考えます。すると、下図のようにして  $A_i > A_j$  となる組  $(i, j)$  の個数を計算することができます（ $A = [5, 2, 4, 1, 3]$  のときの例を示しています）。



この解法を厳密に書き表すと、次のようにになります。

最初は  $\text{dat}[1], \text{dat}[2], \dots, \text{dat}[N]=0$  に初期化する。その後、 $j = 1, \dots, N$  の順に以下の処理を行う。

- 答え Answer に  $\text{dat}[\text{A}[j]+1]+\dots+\text{dat}[N]$  を加算する
- $\text{dat}[\text{A}[j]]$  に 1 を加算する

さて、この解法の計算量はどれくらいなのでしょうか。 $\text{dat}[\text{A}[j]+1]$  から  $\text{dat}[N]$  までの区間の総和を直接計算すると、計算量は  $O(N^2)$  となり、元々の解法と変わりません。

しかし、 $\text{dat}[i]$  の値をセグメント木を使って管理すると、区間の総和を計算量  $O(\log N)$  で求めることができるために、全体の計算量は見事に  $O(N \log N)$  まで削減されます。

最後に、以上の解法を実装すると、解答例のようになります。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 #include <iostream>
5 #include <algorithm>
6 using namespace std;
7
8 class SegmentTree {
9 public:
10     int dat[600000], siz = 1;
11
12     // 要素 dat の初期化を行う（最初は全部ゼロ）
13     void init(int N) {
14         siz = 1;
15         while (siz < N) siz *= 2;
16         for (int i = 1; i < siz * 2; i++) dat[i] = 0;
17     }
18
19     // クエリ 1 に対する処理
20     void update(int pos, int x) {
21         pos = pos + siz - 1;
22         dat[pos] = x;
23         while (pos >= 2) {
24             pos /= 2;
25             dat[pos] = dat[pos * 2] + dat[pos * 2 + 1]; // 8.8 節から変更した部分
26         }
27     }
}
```

```

28 // クエリ 2 に対する処理
29 int query(int l, int r, int a, int b, int u) {
30     if (r <= a || b <= l) return 0; // 8.8 節から変更した部分
31     if (l <= a && b <= r) return dat[u];
32     int m = (a + b) / 2;
33     int AnswerL = query(l, r, a, m, u * 2);
34     int AnswerR = query(l, r, m, b, u * 2 + 1);
35     return AnswerL + AnswerR; // 8.8 節から変更した部分
36 }
37 };
38
39 int N, A[150009];
40 SegmentTree Z;
41
42 int main() {
43     // 入力
44     cin >> N;
45     for (int i = 1; i <= N; i++) cin >> A[i];
46
47     // セグメント木の準備
48     Z.init(N);
49
50     // 答えを求める
51     long long Answer = 0;
52     for (int j = 1; j <= N; j++) {
53         Answer += Z.query(A[j] + 1, N + 1, 1, Z.siz + 1, 1);
54         Z.update(A[j], 1);
55     }
56
57     // 出力
58     cout << Answer << endl;
59     return 0;
60 }
```

※Python のコードはサポートページをご覧ください

# 第9章

## グラフアルゴリズム

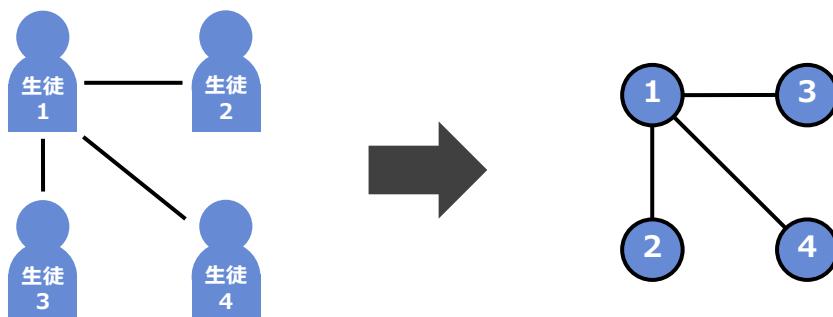
応用問題 9.1	120
応用問題 9.2	122
応用問題 9.3	125
応用問題 9.4	128
応用問題 9.5	131
応用問題 9.6	135
応用問題 9.7	138
応用問題 9.8	140
応用問題 9.9	146

9.1

## 問題 B61 : Influencer

(難易度：★2相当)

まず、この問題で出てくる友達関係は、以下のように重みなし無向グラフで表現することができます。



このとき、「生徒  $i$  の友達の人数」は「グラフの頂点  $i$  の次数」と一致するため、次数が最も大きい頂点の番号を出力すれば正解です。

隣接リスト表現を使ってグラフを管理したときの実装例を以下に示します。  
 $G[i].size()$  は頂点  $i$  の次数に対応することに注意してください。

### ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int N, M, A[100009], B[100009];
6 vector<int> G[100009];
7
8 int main() {
9     // 入力
10    cin >> N >> M;
11    for (int i = 1; i <= M; i++) {
12        cin >> A[i] >> B[i];
13        G[A[i]].push_back(B[i]); // 「頂点 A[i] に隣接する頂点」として B[i] を追加
14        G[B[i]].push_back(A[i]); // 「頂点 B[i] に隣接する頂点」として A[i] を追加
15    }
16
17    // 次数 (= 友達の数) が最大となる生徒の番号を求める
18    int MaxFriends = 0; // 友達の数の最大値
19    int MaxID = 0;      // 番号

```

```

20     for (int i = 1; i <= N; i++) {
21         if (MaxFriends < (int)G[i].size()) {
22             MaxFriends = (int)G[i].size();
23             MaxID = i;
24         }
25     }
26
27     // 出力
28     cout << MaxID << endl;
29     return 0;
30 }
```

※Python のコードはサポートページをご覧ください

## ◆ 補足：グラフを使わない解法

9 章のテーマは「グラファルゴリズム」なので、解答例ではグラフを使って実装しました。しかし、以下のようにグラフを使わない解法もあります。

- 生徒  $i$  の現在の友達の数を表す配列  $\text{cnt}[i]$  を用意する
- $i = 1, 2, \dots, M$  について、 $\text{cnt}[\text{A}[i]]$  と  $\text{cnt}[\text{B}[i]]$  に 1 を足す

これを実装すると、以下のようになります。計算量は  $O(N + M)$  です。

```

1 // 入力
2 int cnt[100009];
3 cin >> N >> M;
4 for (int i = 1; i <= N; i++) cnt[i] = 0;
5 for (int i = 1; i <= M; i++) {
6     cin >> A[i] >> B[i];
7     cnt[A[i]] += 1; cnt[B[i]] += 1;
8 }
9
10 // 友達の数が最大となる生徒の番号を求める
11 int MaxFriends = 0; // 友達の数の最大値
12 int MaxID = 0; // 番号
13 for (int i = 1; i <= N; i++) {
14     if (MaxFriends < cnt[i]) {
15         MaxFriends = cnt[i];
16         MaxID = i;
17     }
18 }
19 cout << MaxID << endl;
```

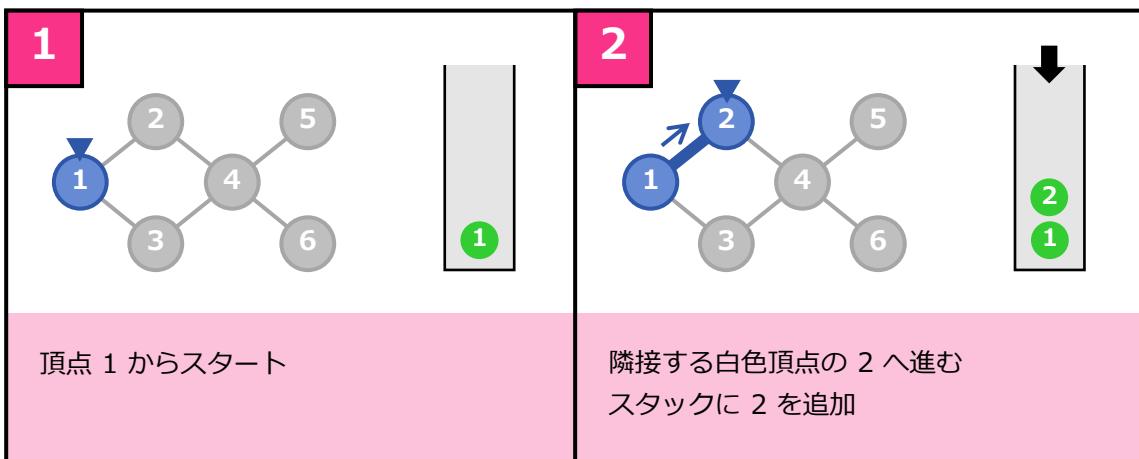
この 9.2 節では、「進めるだけ進み、行き詰まつたら一步戻る」という深さ優先探索を使って、頂点 1 から各頂点まで到達可能かどうかを判定する方法を紹介しました。しかし、頂点 1 から  $N$  までの具体的な経路を得るにはどうすれば良いのでしょうか。

## ◆ 経路を得る方法

まず、以下のようにして、**深さ優先探索における移動経路の跡（本の 354 ページの図の青線部分に対応）**を管理することを考えます。

- ・ スタック Path を用意する。
- ・ 頂点 pos へ進んだときには、Path の一番上に pos を追加する。
- ・ 一步戻るときには、Path の一番上の要素を削除する。

このとき、頂点 1 から  $N$  までの経路は、「頂点  $N$  へ進んだ時点での移動経路の跡」となります。具体例を以下に示します。



<p><b>3</b></p> <p>隣接する白色頂点の 4 へ進む スタックに 4 を追加</p>	<p><b>4</b></p> <p>隣接する白色頂点の 3 へ進む スタックに 3 を追加</p>
<p><b>5</b></p> <p>これ以上進めないので一步戻る スタックの一番上を削除</p>	<p><b>6</b></p> <p>隣接する白色頂点の 5 へ進む スタックに 5 を追加</p>
<p><b>7</b></p> <p>これ以上進めないので一步戻る スタックの一番上を削除</p>	<p><b>8</b></p> <p>隣接する白色頂点の 6 へ進む スタックに 6 を追加 答えとなる経路は「1→2→4→6」</p>

ここまで的内容を実装すると、以下の解答例のようになります。頂点  $N$  にたどり着いた時点でのスタックの要素を、下から順番に出力すれば良いです。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <vector>
3 #include <stack>
4 #include <algorithm>
5 using namespace std;
```

```

6 int N, M, A[100009], B[100009];
7 vector<int> G[100009]; // グラフ
8 bool visited[100009]; // 頂点 i が青か白か
9 stack<int> Path, Answer; // 移動経路の跡
10
11 void dfs(int pos) {
12     // ゴール地点にたどり着いた！
13     if (pos == N) {
14         Answer = Path;
15         return;
16     }
17
18     // その他の場合
19     visited[pos] = true;
20     for (int i = 0; i < G[pos].size(); i++) {
21         int nex = G[pos][i];
22         if (visited[nex] == false) {
23             Path.push(nex); // 頂点 nex を経路に追加
24             dfs(nex);
25             Path.pop(); // 頂点 nex を経路から削除
26         }
27     }
28     return;
29 }
30
31 int main() {
32     // 入力
33     cin >> N >> M;
34     for (int i = 1; i <= M; i++) {
35         cin >> A[i] >> B[i];
36         G[A[i]].push_back(B[i]);
37         G[B[i]].push_back(A[i]);
38     }
39
40     // 深さ優先探索
41     for (int i = 1; i <= N; i++) visited[i] = false;
42     Path.push(1); // 頂点 1 (スタート地点) を経路に追加
43     dfs(1);
44
45     // スタックの要素を「下から順に」に記録
46     vector<int> Output;
47     while (!Answer.empty()) {
48         Output.push_back(Answer.top());
49         Answer.pop();
50     }
51     reverse(Output.begin(), Output.end());
52
53     // 答えの出力
54     for (int i = 0; i < Output.size(); i++) {
55         if (i >= 1) cout << " ";
56         cout << Output[i];
57     }
58     cout << endl;
59     return 0;
60 }

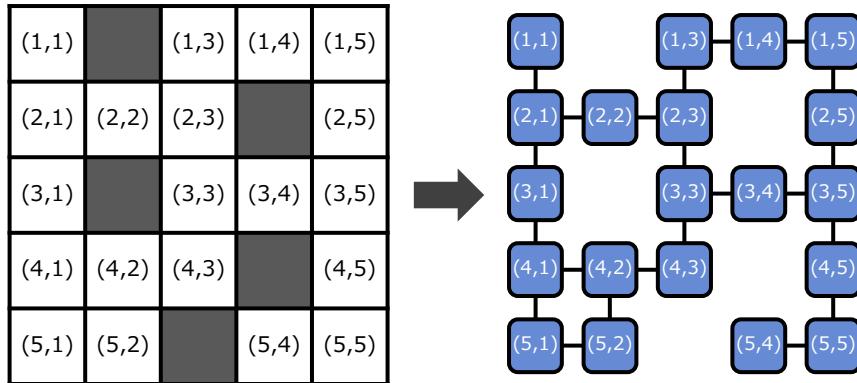
```

※Python のコードはサポートページをご覧ください

## 問題 B63：幅優先探索

(難易度：★4相当)

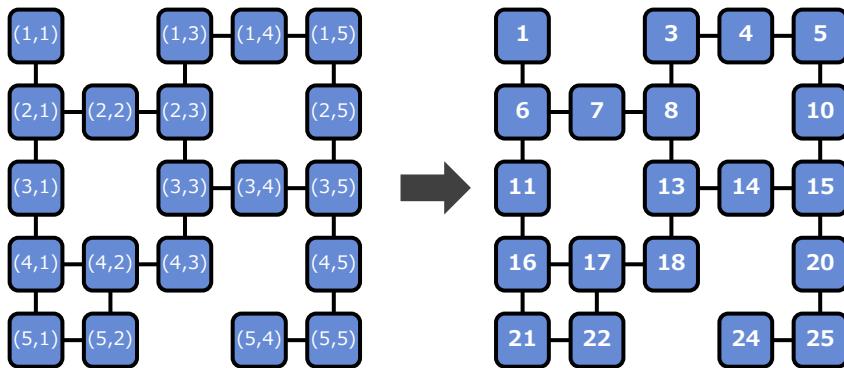
まず、迷路を以下のようなグラフとして表現することを考えます。マスが頂点、隣接するマスの関係が辺に対応しています。



すると、左上マスから右下マスまでの最短手数は、**頂点  $(1, 1)$  から頂点  $(H, W)$  までの最短経路長**になります。

そのため、例題 A63 (9.3 節) でやったように、幅優先探索で最短経路長を計算すれば正解が得られます。

なお、次ページの実装例では、頂点  $(i, j)$  の番号を 1 つの整数  $(i - 1) \times W + j$  で表していることに注意してください<sup>※1</sup>。



※1 もちろん、二次元配列 `vector<int> G[i][j]` を使って隣接リストを管理すれば、頂点番号がそのままでも幅優先探索を行うことができますが、1 つの整数の方が実装が楽です

## 解答例 (C++)

```
1 #include <iostream>
2 #include <vector>
3 #include <queue>
4 using namespace std;
5
6 // 入力
7 int H, W;
8 int sx, sy, start; // スタートの座標 (sx, sy) と頂点番号 sx*H+sy
9 int gx, gy, goal; // ゴールの座標 (gx, gy) と頂点番号 gx*W+gy
10 char c[59][59];
11
12 // グラフ・最短経路
13 int dist[2509];
14 vector<int> G[2509];
15
16 int main() {
17     // 入力
18     cin >> H >> W;
19     cin >> sx >> sy; start = sx * W + sy;
20     cin >> gx >> gy; goal = gx * W + gy;
21     for (int i = 1; i <= H; i++) {
22         for (int j = 1; j <= W; j++) cin >> c[i][j];
23     }
24
25     // 横方向の辺 [(i, j) - (i, j+1)] をグラフに追加
26     for (int i = 1; i <= H; i++) {
27         for (int j = 1; j <= W - 1; j++) {
28             int idx1 = i * W + j; // 頂点 (i, j) の頂点番号
29             int idx2 = i * W + (j + 1); // 頂点 (i, j+1) の頂点番号
30             if (c[i][j] == '.' && c[i][j + 1] == '.') {
31                 G[idx1].push_back(idx2);
32                 G[idx2].push_back(idx1);
33             }
34         }
35     }
36
37     // 縦方向の辺 [(i, j) - (i+1, j)] をグラフに追加
38     for (int i = 1; i <= H - 1; i++) {
39         for (int j = 1; j <= W; j++) {
40             int idx1 = i * W + j; // 頂点 (i, j) の頂点番号
41             int idx2 = (i + 1) * W + j; // 頂点 (i+1, j) の頂点番号
42             if (c[i][j] == '.' && c[i + 1][j] == '.') {
43                 G[idx1].push_back(idx2);
44                 G[idx2].push_back(idx1);
45             }
46         }
47     }
48
49     // 幅優先探索の初期化
50     for (int i = 1; i <= H * W; i++) dist[i] = -1;
51     queue<int> Q;
52     Q.push(start); dist[start] = 0;
```

```
53 // 幅優先探索
54 while (!Q.empty()) {
55     int pos = Q.front();
56     Q.pop();
57     for (int i = 0; i < G[pos].size(); i++) {
58         int to = G[pos][i];
59         if (dist[to] == -1) {
60             dist[to] = dist[pos] + 1;
61             Q.push(to);
62         }
63     }
64 }
65
66 // 答えを出力
67 cout << dist[goal] << endl;
68 return 0;
69 }
```

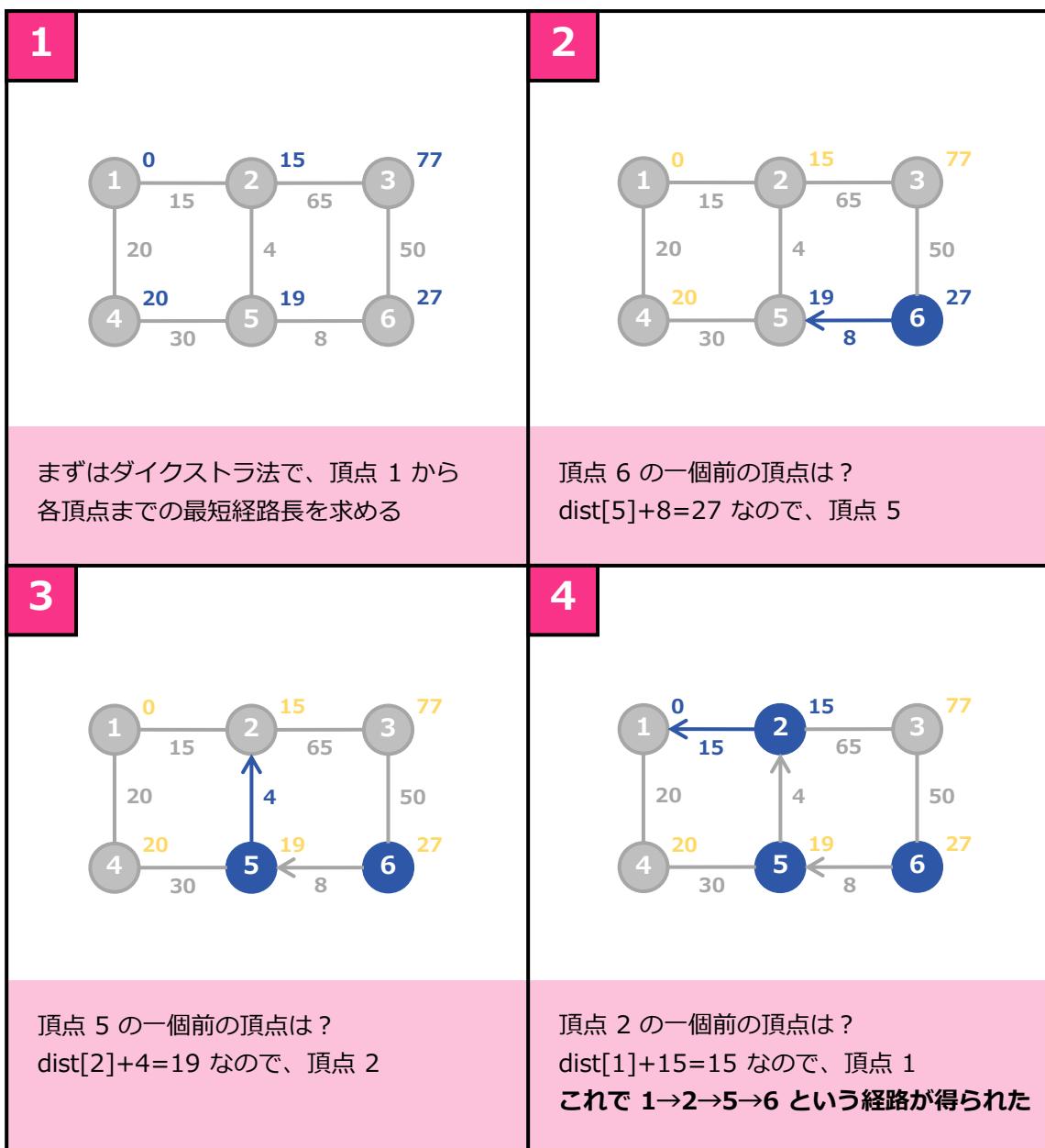
※Python のコードはサポートページをご覧ください

9.4

## 問題 B64 : Shortest Path (難易度：★4相当)

4.2 節では、動的計画法で具体的な解を求める方法として、「逆から順番に考える方法」を紹介しました。

今回も同じように、**ゴール地点の頂点 N から順番に考えれば**、頂点 1 から頂点 N までの具体的な最短経路を求めることができます。例を以下に示します（本の 362 ページの入力例と同じです）。



## 解答例 (C++)

```
1 #include <iostream>
2 #include <queue>
3 #include <vector>
4 #include <algorithm>
5 using namespace std;
6
7 // 入力・グラフ
8 int N, M, A[100009], B[100009], C[100009];
9 vector<pair<int, int>> G[100009];
10
11 // ダイクストラ法
12 int cur[100009]; bool kakutei[100009];
13 priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> Q;
14
15
16 int main() {
17     // 入力
18     cin >> N >> M;
19     for (int i = 1; i <= M; i++) {
20         cin >> A[i] >> B[i] >> C[i];
21         G[A[i]].push_back(make_pair(B[i], C[i]));
22         G[B[i]].push_back(make_pair(A[i], C[i]));
23     }
24
25     // 配列の初期化
26     for (int i = 1; i <= N; i++) kakutei[i] = false;
27     for (int i = 1; i <= N; i++) cur[i] = 2000000000;
28
29     // スタート地点をキューに追加
30     cur[1] = 0;
31     Q.push(make_pair(cur[1], 1));
32
33     // ダイクストラ法
34     while (!Q.empty()) {
35         // 次に確定させるべき頂点を求める
36         int pos = Q.top().second; Q.pop();
37
38         // Q の最小要素が「既に確定した頂点」の場合
39         if (kakutei[pos] == true) continue;
40
41         // cur[x] の値を更新する
42         kakutei[pos] = true;
43         for (int i = 0; i < G[pos].size(); i++) {
44             int nex = G[pos][i].first;
45             int cost = G[pos][i].second;
46             if (cur[nex] > cur[pos] + cost) {
47                 cur[nex] = cur[pos] + cost;
48                 Q.push(make_pair(cur[nex], nex));
49             }
50         }
51     }
}
```

```
52 // 答えの復元 (Place は現在の位置：ゴールから出発)
53 vector<int> Answer;
54 int Place = N;
55 while (true) {
56     Answer.push_back(Place);
57     if (Place == 1) break;
58
59     // Place の前の頂点としては、一体どこが良いのか？
60     for (int i = 0; i < G[Place].size(); i++) {
61         int nex = G[Place][i].first;
62         int cost = G[Place][i].second;
63         if (cur[nex] + cost == cur[Place]) {
64             Place = nex;
65             break;
66         }
67     }
68 }
69 reverse(Answer.begin(), Answer.end());
70
71 // 出力
72 for (int i = 0; i < Answer.size(); i++) {
73     if (i >= 1) cout << " ";
74     cout << Answer[i];
75 }
76 cout << endl;
77 return 0;
78 }
```

※Python のコードはサポートページをご覧ください

9.5

## 問題 B65 : Road to Promotion Hard (難易度 : ★4相当)

まずは簡単のため、問題 A65（本の 9.5 節）と同じように、社員 1 が上司であり、番号の小さい社員の方が偉い場合を考えましょう。

社員  $x$  の直属の部下の階級がそれぞれ  $r_1, r_2, \dots, r_k$  であるとき、社員  $x$  の階級は次式で表される：

$$(社員 x の階級) = \max(r_1, r_2, \dots, r_k) + 1$$

番号の小さい方が上司なので、社員  $N, \dots, 2, 1$  の順に、上式の通りに階級を計算していくけば、計算量  $O(N)$  で答えがわかる。

$N = 5$  のときの具体例を以下に示します。

<b>1</b>	<b>2</b>	<b>3</b>
 社員 5 は部下がないので 階級 0	 社員 4 は部下がないので 階級 0	 社員 3 は部下がないので 階級 0
<b>4</b>	<b>5</b>	<b>6</b>
 社員 2 の階級は $\max(0,0)+1=1$	 社員 1 の階級は $\max(1,0)+1=2$	 すべての社員の 階級がわかった！

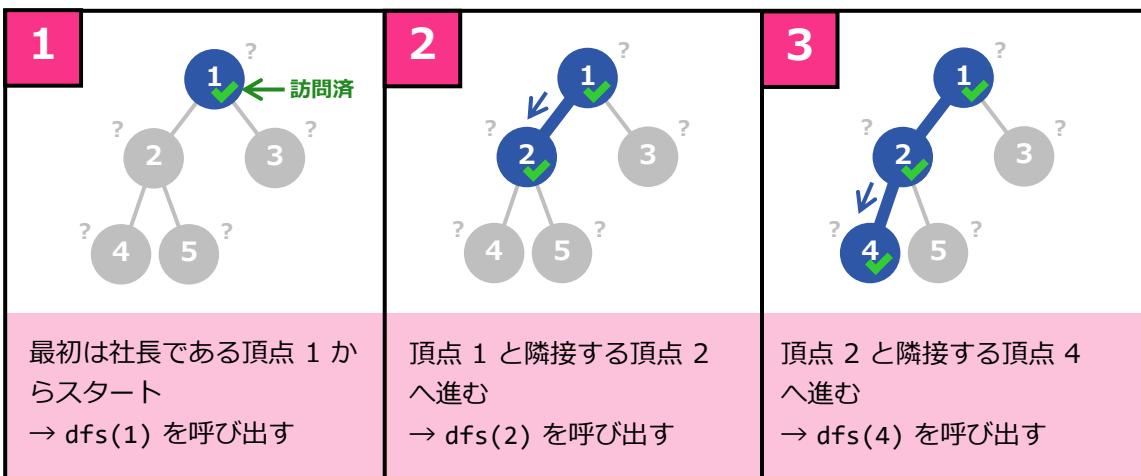
## ◆ 一般の場合を解く

ここまででは、番号の小さい方が偉いという簡単なケースの解き方を説明しました。ところが実際はそうとは限らないため、社員  $N, \dots, 2, 1$  の順番に求める方法が通用しません。

そこで、深さ優先探索を使って、**社長と遠い方から順番に階級を求めていく**と上手くいきます（注：社長との距離が近い方が上司であるという性質を使っています）。実装例を以下に示します。社長の番号を  $X$  とするとき、最初は  $\text{dfs}(X)$  が呼び出されます※2。

```
1 // 深さ優先探索を行う関数 (pos は現在位置)
2 // 戻り値は社員 pos の階級
3 int dfs(int pos) {
4     // 最初、社員 pos の階級を 0 に設定する
5     visited[pos] = true;
6     Answer[pos] = 0;
7
8     // 探索をする
9     for (int i = 0; i < G[pos].size(); i++) {
10         int nex = G[pos][i];
11         if (visited[nex] == false) {
12             int ret = dfs(nex);
13             Answer[pos] = max(Answer[pos], ret + 1); // 階級を更新する
14         }
15     }
16
17     // 値を返す
18     return Answer[pos];
19 }
```

ここで、再帰関数の動きは少し複雑ですので、以下に例を示します（最初の例と同じ）。イメージとしては、「隣接する頂点をすべて訪問して一歩戻るときに、階級を計算する」といった感じです。



※2  $G$  は上下関係を表すグラフの隣接リスト、 $\text{Answer}[i]$  は社員  $i$  の階級を表します

<b>4</b>		<b>5</b>	
行き詰まつたので一步戻る 社員 4 の階級は 0	頂点 2 と隣接する頂点 5 へ進む $\rightarrow \text{dfs}(5)$ を呼び出す	行き詰まつたので一步戻る 社員 5 の階級は 0	
<b>7</b>		<b>8</b>	
行き詰まつたので一步戻る 社員 2 の階級は $\max(0,0)+1=1$	頂点 1 と隣接する頂点 3 へ進む $\rightarrow \text{dfs}(3)$ を呼び出す	行き詰まつたので一步戻る 社員 3 の階級は 0	
<b>10</b>		<b>11</b>	
行き詰まつたので計算終了 社員 1 の階級は $\max(1,0)+1=2$	これですべての社員の階級がわかった！		

最後に、入力部分・出力部分・グラフの隣接リストを作る部分などを加えると、実装は以下の解答例のように穴ります。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;

```

```

5 // 入力される変数・答え
6 int N, T, A[100009], B[100009];
7 int Answer[100009];
8
9 // グラフ・深さ優先探索
10 vector<int> G[100009];
11 bool visited[100009];
12
13 // 深さ優先探索を行う関数 (pos は現在位置)
14 // 戻り値は社員 pos の階級
15 int dfs(int pos) {
16     // 最初、社員 pos の階級を 0 に設定する
17     visited[pos] = true;
18     Answer[pos] = 0;
19
20     // 探索をする
21     for (int i = 0; i < G[pos].size(); i++) {
22         int nex = G[pos][i];
23         if (visited[nex] == false) {
24             int ret = dfs(nex);
25             Answer[pos] = max(Answer[pos], ret + 1); // 階級を更新する
26         }
27     }
28
29     // 値を返す
30     return Answer[pos];
31 }
32
33 int main() {
34     // 入力
35     cin >> N >> T;
36     for (int i = 1; i <= N - 1; i++) {
37         cin >> A[i] >> B[i];
38         G[A[i]].push_back(B[i]); // A[i]→B[i] の方向に辺を追加
39         G[B[i]].push_back(A[i]); // B[i]→A[i] の方向に辺を追加
40     }
41
42     // 深さ優先探索
43     dfs(T);
44
45     // 出力
46     for (int i = 1; i <= N; i++) {
47         if (i >= 2) cout << " ";
48         cout << Answer[i];
49     }
50     cout << endl;
51     return 0;
52 }
```

※Python のコードはサポートページをご覧ください

9.6

## 問題 B66 : Typhoon

(難易度: ★5相当)

※この問題は例題と比べて相当難しいです。

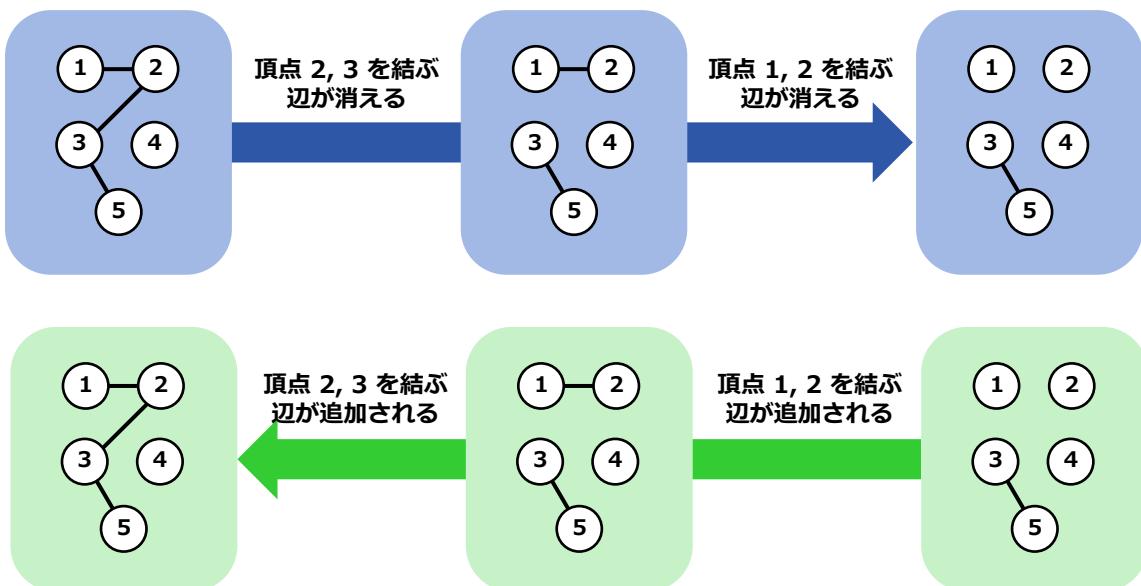
まず、駅を頂点とし、鉄道路線を辺としたグラフを考えると、クエリ 1・2 はそれぞれ以下のような処理に対応します。

クエリ1: $x$ 本目の路線が運休になる	$x$ 番目の辺が消える
クエリ2: 駅 $s$ から駅 $t$ へ移動できるかを答える	頂点 $s$ と $t$ は同じ連結成分かを答える

これらの処理は Union-Find で行えるクエリ（本の 375 ページ）と非常によく似ていますが、残念ながら辺を消す操作だけはできません。一体どうすれば良いのでしょうか。

## ◆ 一般の場合を解く

解決策の一つとして、クエリを逆順に処理するという方法があります。もし逆順になれば、「辺が消える操作」は「辺を追加する操作」に変わるために、Union-Find で処理できる形になります。



このアイデアを実装すると、次ページの解答例のようになります。なお、最後のクエリの時点でも運休になっていない路線が存在する場合があるため、このような辺を事前に追加しておく必要があることに注意してください。

## 解答例 (C++)

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 class UnionFind {
6 public:
7     int par[100009];
8     int siz[100009];
9
10    // N 頂点の Union-Find を作成
11    void init(int N) {
12        for (int i = 1; i <= N; i++) par[i] = -1; // 最初は親が無い
13        for (int i = 1; i <= N; i++) siz[i] = 1; // 最初はグループの頂点数が 1
14    }
15
16    // 頂点 x の根を返す関数
17    int root(int x) {
18        while (true) {
19            if (par[x] == -1) break; // 1 個先（親）がなければ、ここが根
20            x = par[x]; // 1 個先（親）に進む
21        }
22        return x;
23    }
24
25    // 要素 u と v を統合する関数
26    void unite(int u, int v) {
27        int RootU = root(u);
28        int RootV = root(v);
29        if (RootU == RootV) return; // u と v が同グループのときは処理を行わない
30        if (siz[RootU] < siz[RootV]) {
31            par[RootU] = RootV;
32            siz[RootV] = siz[RootU] + siz[RootV];
33        }
34        else {
35            par[RootV] = RootU;
36            siz[RootU] = siz[RootU] + siz[RootV];
37        }
38    }
39
40    // 要素 u と v が同一のグループかどうかを返す関数
41    bool same(int u, int v) {
42        if (root(u) == root(v)) return true;
43        return false;
44    }
45};
46
47 // 入力で与えられる変数・答え
48 int N, M, A[100009], B[100009];
49 int Q, QueryType[100009], x[100009], u[100009], v[100009];
50 string Answer[100009];
51
52 // その他の変数
53 UnionFind UF;
54 bool cancelled[100009];
```

```

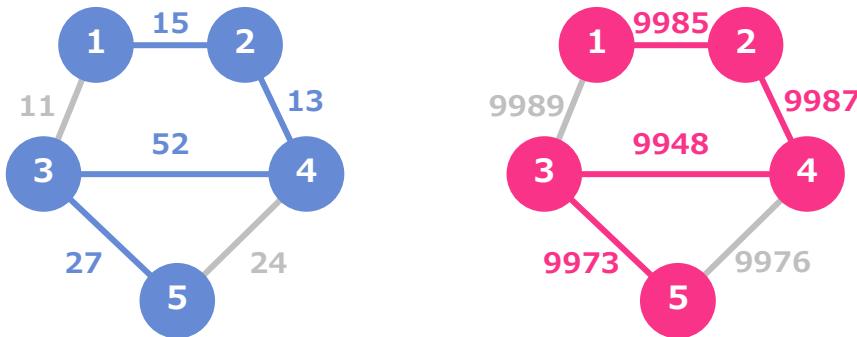
55 int main() {
56     // 入力
57     cin >> N >> M;
58     for (int i = 1; i <= M; i++) cin >> A[i] >> B[i];
59     cin >> Q;
60     for (int i = 1; i <= Q; i++) {
61         cin >> QueryType[i];
62         if (QueryType[i] == 1) cin >> x[i];
63         if (QueryType[i] == 2) cin >> u[i] >> v[i];
64     }
65
66     // 最初に運休になっている路線を求める
67     for (int i = 1; i <= M; i++) cancelled[i] = false;
68     for (int i = 1; i <= Q; i++) {
69         if (QueryType[i] == 1) cancelled[x[i]] = true;
70     }
71
72     // Union-Find の初期化 (その日の最後の状態にする)
73     UF.init(N);
74     for (int i = 1; i <= M; i++) {
75         if (cancelled[i] == false && UF.same(A[i], B[i]) == false) {
76             UF.unite(A[i], B[i]);
77         }
78     }
79
80     // クエリを逆から処理
81     for (int i = Q; i >= 1; i--) {
82         if (QueryType[i] == 1) {
83             // 駅 A[x[i]] と駅 B[x[i]] を結ぶ路線が開通
84             if (UF.same(A[x[i]], B[x[i]]) == false) UF.unite(A[x[i]], B[x[i]]);
85         }
86         if (QueryType[i] == 2) {
87             if (UF.same(u[i], v[i]) == true) Answer[i] = "Yes";
88             else Answer[i] = "No";
89         }
90     }
91
92     // 出力
93     for (int i = 1; i <= Q; i++) {
94         if (QueryType[i] == 2) cout << Answer[i] << endl;
95     }
96     return 0;
97 }
```

※Python のコードはサポートページをご覧ください

グラフの最大全域木は、**大きい辺から順番に追加していく**という非常に単純な貪欲法で求めることができます。

なぜなら、「グラフ  $G$  の最大全域木」と「各辺の長さを  $10000 - x$  に変えたグラフ  $G'$  の最小全域木」が一致するからです※3。

そして、最小全域木は本の 383 ページで説明したように、小さい辺から順番に追加していくという貪欲法（クラスカル法）で求められるからです。



### ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 // Union-Find クラスの実装は本の 9.6 節 (379~380 ページ) 参照
7 int N, M;
8 int A[100009], B[100009], C[100009];
9 UnionFind UF;
10
11 int main() {
12     // 入力
13     cin >> N >> M;
14     for (int i = 1; i <= M; i++) cin >> A[i] >> B[i] >> C[i];

```

※3 このことは、「もしグラフ  $G$  で選んだ辺の重みの総和が  $w$  であり、グラフ  $G'$  でもまったく同じ辺を選ぶとき、グラフ  $G'$  での辺の重みの総和が  $10000(N - 1) - w$  になり、 $w$  の値が大きいほど小さくなること」から説明できます ( $N$  を頂点数とする)。

たとえば上図の例のように頂点が 5 個である場合、グラフ  $G'$  の重みの総和が  $40000 - w$  となり、この値は  $w$  が最大のとき最小になります。

```
15 // 辺を長さの大きい順にソートする
16 vector<pair<int, int>> EdgeList;
17 for (int i = 1; i <= M; i++) EdgeList.push_back(make_pair(C[i], i));
18 sort(EdgeList.begin(), EdgeList.end());
19 reverse(EdgeList.begin(), EdgeList.end()); // 問題 A67 と異なる唯一の部分
20
21 // 最大全域木を求める
22 int Answer = 0; UF.init(N);
23 for (int i = 0; i < EdgeList.size(); i++) {
24     int idx = EdgeList[i].second;
25     if (UF.same(A[idx], B[idx]) == false) {
26         UF.unite(A[idx], B[idx]);
27         Answer += C[idx];
28     }
29 }
30 cout << Answer << endl;
31 return 0;
32 }
```

※Python のコードはサポートページをご覧ください

9.8

## 問題 B68 : ALGO Express

(難易度 : ★7相当)

※この問題は例題と比べて相当難しいです。

この問題は、いきなり  $N$  が大きい場合や  $P_i$  が負の場合を考えると難しくなってしまうので、まずは以下のケースを考えましょう。

ALGO 鉄道には  $N = 3$  個の駅があり、各駅に対して以下のコストがかかる（特急駅に指定しない場合もコストがかかることに注意）。

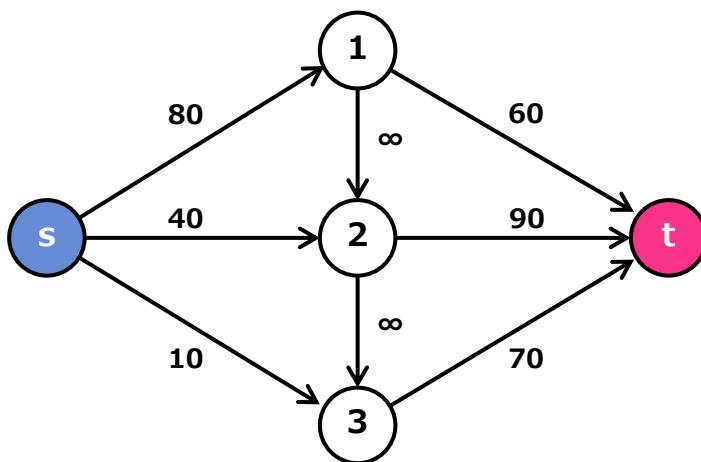
選択	駅 1 のコスト	駅 2 のコスト	駅 3 のコスト
特急駅に指定する場合	60 円	90 円	70 円
特急駅に指定しない場合	80 円	40 円	10 円

また、以下の  $M = 2$  個の条件がある。

- 駅 1 が特急駅ならば駅 2 も特急駅でなければならない
- 駅 2 が特急駅ならば駅 3 も特急駅でなければならない

すべての条件を満たすときの最小コストは何円か。

実は、この問題の答えは、以下のグラフの最大フロー、すなわち最小カットの重みの総和と一致することが知られています。



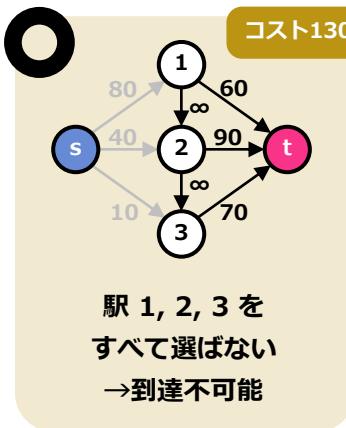
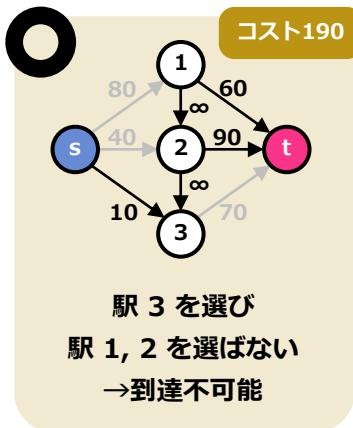
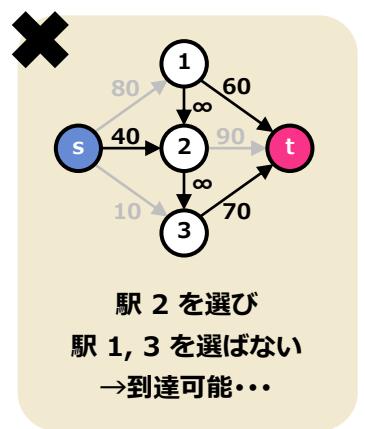
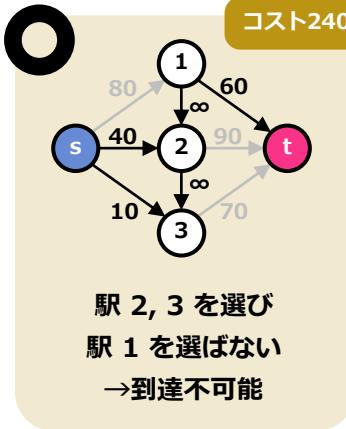
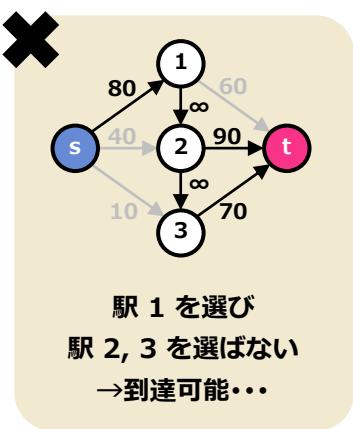
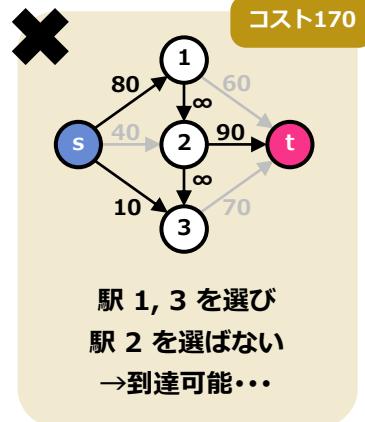
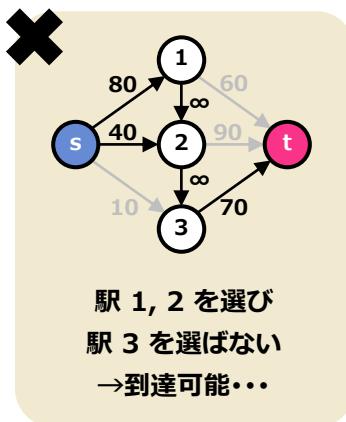
では、なぜそうなるのでしょうか。最小カット問題において、

- 駅  $i$  を特急駅に指定しない場合、 $s$  と頂点  $i$  を結ぶ辺を切る
- 駅  $i$  を特急駅に指定する場合、頂点  $i$  と  $t$  を結ぶ辺を切る

という辺の切り方をすることを考えます（切った辺の重みの総和がコストの合計となります）。

このとき、***M = 2 個すべての条件を満たす場合は既に s から t へ到達不可***能になっており、追加で辺を消す必要がありません。しかし 1 つでも満たさない条件がある場合は、重み  $\infty$  の辺を消す必要があり、最小カットが  $\infty$  になってしまいます。

そのため、最小カットは「すべての条件を満たす場合の最小コスト」に対応します。以下に例を示します。



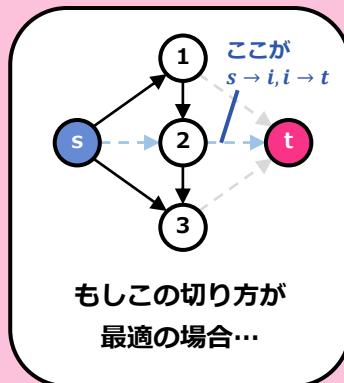
最小コストは  
130



もしかしたら、連結のままである場合に、重さ  $\infty$  の辺の代わりに他の辺を消す方が良いのではないか（つまり、ある頂点  $i$  に対して辺  $s \rightarrow i$  と辺  $i \rightarrow t$  を両方消す）と思うかもしれません。

しかし、このような消し方は絶対に最適にはなりません。これは次のように証明することができます（非常に難しいので読み飛ばしてもかまいません）。

仮にこのような消し方が最適である場合、辺  $s \rightarrow i$  だけを追加しても、辺  $i \rightarrow t$  だけを追加しても、始点  $s$  から終点  $t$  まで到達可能になるはずである（たとえば辺  $s \rightarrow i$  を追加しても到達不可能である場合、辺  $s \rightarrow i$  を消さない方が明らかに最適に近い）。



辺  $i \rightarrow t$  を消しても  
到達可能なはず！

辺  $s \rightarrow i$  を消しても  
到達可能なはず！<sup>※4</sup>

ところが、辺  $s \rightarrow i$  だけを追加しても、辺  $i \rightarrow t$  だけを追加しても、到達不可能なものが到達可能になるようなケースは絶対に存在しない。

なぜなら、辺  $s \rightarrow i$  だけを追加したときの  $s$  から  $t$  までの経路は「 $s \rightarrow i \rightarrow (\text{適当なパスA}) \rightarrow t$ 」という形で必ず表されるからである。

また、辺  $i \rightarrow t$  だけを追加したときの  $s$  から  $t$  までの経路は「 $s \rightarrow (\text{適当なパスB}) \rightarrow i \rightarrow t$ 」という形で必ず表されるからである。

そして、辺  $s \rightarrow i$  も辺  $i \rightarrow t$  も追加しない元の切り方の場合でも、「 $s \rightarrow (\text{適当なバスA}) \rightarrow i \rightarrow (\text{適当なバスB}) \rightarrow t$ 」という経路を通ることで、 $s$  から  $t$  まで到達可能になってしまうからである。

これは「元の切り方で始点  $s$  から終点  $t$  へ到達不可能である」という仮定に矛盾する（背理法）。

## ◆ 実際の問題を解く

それではいよいよ本題に入ります。駅  $i$  特急駅に指定するときの利益  $P_i$  が正・負どちらもあり得る場合は、どうやって解けば良いのでしょうか。たとえば以下のケースを考えましょう。

ALGO 鉄道には  $N = 3$  個の駅があり、各駅を特急駅に指定した場合、以下のようない利益が得られる。

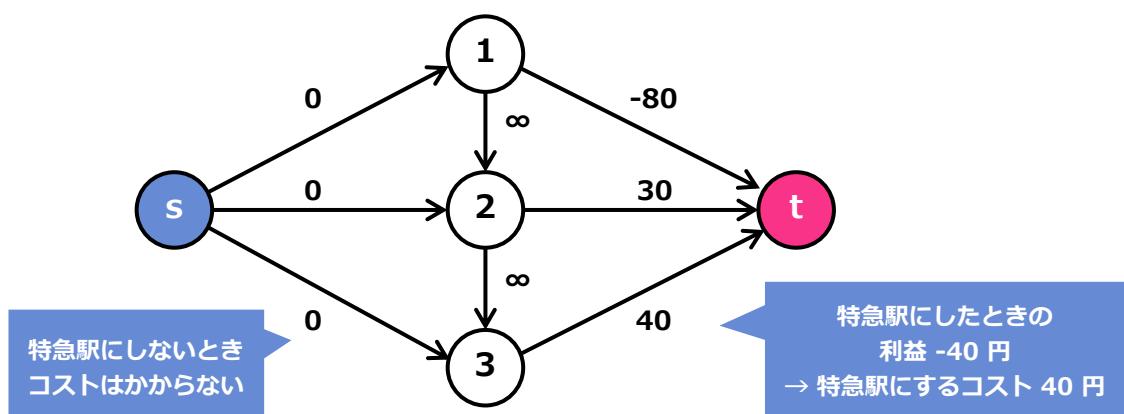
選択	駅 1 のコスト	駅 2 のコスト	駅 3 のコスト
特急駅に指定するときの利益	80円	-30円	-40円

また、以下の  $M = 2$  個の条件がある。

- 駅 1 が特急駅ならば駅 2 も特急駅でなければならない
- 駅 2 が特急駅ならば駅 3 も特急駅でなければならない

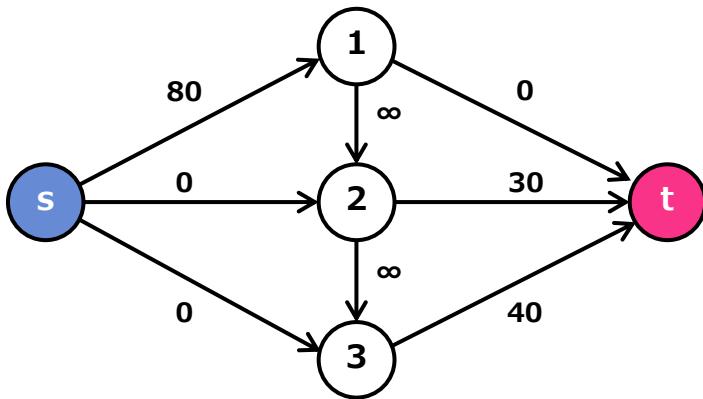
すべての条件を満たすときの利益の最大値は何円か。

一見すると、このケースでの答えは、以下のようなグラフの最小カットを計算することで求められると思うかもしれません。



しかし Ford-Fulkerson 法は、重みが負の辺があるときに正しく動作しません。そこで負の辺を消すために、「**特急駅にしたときのコストが  $-x$  円であること**」を「**特急駅にしないときのコストが  $x$  円であること**」に言い換えることを考えましょう。

たとえば先程の例の場合、グラフは以下のように修正されます。「特急駅にしたときのコストが  $-80$  円」が「特急駅にしないときのコストが  $80$  円」に言い換えられています。



このとき、求める最大の利益は **80-(最小カット)** になります。ここで最小カットが足し算ではなく引き算になっている理由は、最小カットで求めているのはコストの最小値であって、利益の最大値ではないからです。

また  $80$  という値は、「元々特急駅にしたときコスト  $-80$  円・しないとき  $0$  円だったのが、特急駅にしたときコスト  $0$  円・しないとき  $80$  円」になり、一律  $80$  円上がったことに対する補正から出ています。

## ◆ 一般のケースで解く

ここまでのお題は分かりましたでしょうか。最後に一般のケースの解き方を説明します。まずは以下のようないかんグラフを作ります。

辺の始点／終点	コスト
スタート $s \rightarrow$ 頂点 $i$	$P_i < 0$ のとき $0$ 、 $P_i > 0$ のとき $P_i$
頂点 $i \rightarrow$ ゴール $t$	$P_i > 0$ のとき $0$ 、 $P_i < 0$ のとき $-P_i$
頂点 $A_j \rightarrow$ 頂点 $B_j$	$∞$

そこで、 $P_i > 0$  であるような  $P_i$  の合計を  $Offset$  とするとき、求める最大の利益は  $Offset - (\text{グラフの最小カット})$  となります。

したがって、以下の解答例のように実装すると、正しい答えが得られます。  
なお、このプログラムでは頂点番号を 1 つの整数で表すため、スタート地点の番号を  $N + 1$  に設定し、ゴール地点の番号を  $N + 2$  に設定していることに注意してください。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 // MaximumFlow クラスは本の 9.8 節 (393~394 ページ) 参照
7 int N, P[159];
8 int M, A[159], B[159];
9 MaximumFlow Z;
10
11 int main() {
12     // 入力
13     cin >> N >> M;
14     for (int i = 1; i <= N; i++) cin >> P[i];
15     for (int i = 1; i <= M; i++) cin >> A[i] >> B[i];
16
17     // グラフを作る (前半パート)
18     int Offset = 0;
19     Z.init(N);
20     for (int i = 1; i <= N; i++) {
21         // 効果が正の場合は、特急駅にしない場合 (始点→i) のコストを P[i] に設定
22         if (P[i] >= 0) {
23             Z.add_edge(N + 1, i, P[i]);
24             Offset += P[i];
25         }
26         // 効果が負の場合は、特急駅にする場合 (i→終点) のコストを -P[i] に設定
27         if (P[i] < 0) {
28             Z.add_edge(i, N + 2, -P[i]);
29         }
30     }
31
32     // グラフを作る (後半パート)
33     for (int i = 1; i <= M; i++) {
34         Z.add_edge(A[i], B[i], 1000000000);
35     }
36
37     // 答えを求める
38     int Answer = Offset - Z.max_flow(N + 1, N + 2);
39     cout << Answer << endl;
40     return 0;
41 }
```

※Python のコードはサポートページをご覧ください

まず、以下のような  $N + 26$  頂点のグラフを考えます。

[頂点の情報]

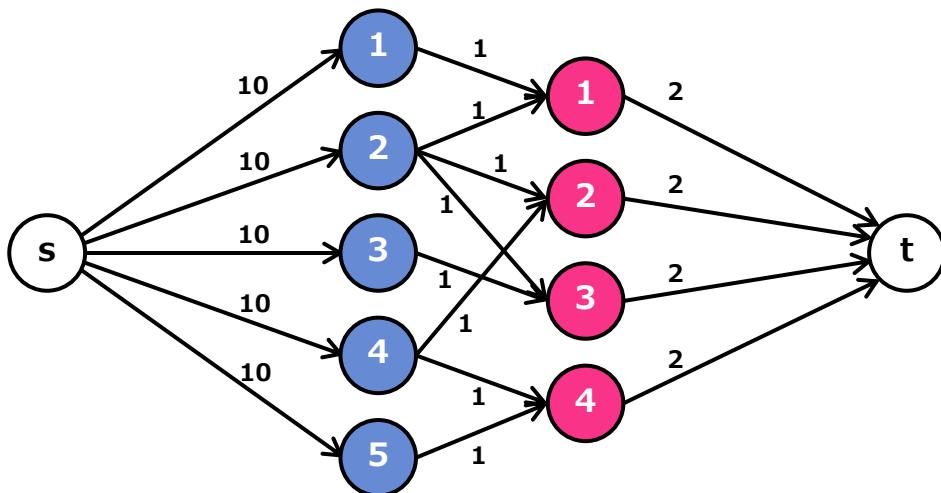
- 青色頂点は  $N$  個あり、従業員に対応する
- 赤色頂点は 24 個あり、時間帯に対応する
- 他にも、スタート地点  $s$  とゴール地点  $t$  が存在する

[辺の情報]

以下の 3 種類の辺を追加する :

辺の始点／終点	容量	備考
スタート $s \rightarrow$ 青色頂点 $i$	10	10 は労働時間の上限に対応
青色頂点 $i \rightarrow$ 赤色頂点 $j$	1	$C_{i,j} = 1$ のとき（働けるとき）のみ追加
赤色頂点 $j \rightarrow$ ゴール $t$	$M$	$M$ は必要労働者数に対応

$N = 5, M = 2$  のときのグラフの例を以下に示します（ここでは図の大きさの都合上、1 日が 24 時間ではなく 4 時間の場合を例としています）。



このとき、最大フローの総流量が  $24 \times M$  であるときに限り答えは Yes、そうでなければ答えは No となります。

この理由は、以下の 2 つのポイントから説明できます。

### [ポイント1]

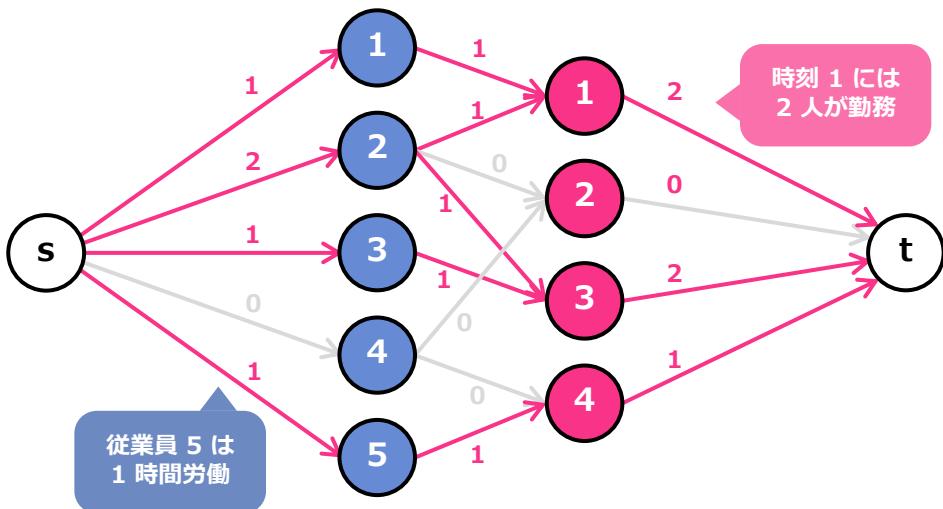
フローにおける各辺の流量は、以下のような値に対応する。

辺の始点／終点	値の意味
スタート $s \rightarrow$ 青色頂点 $i$	従業員 $i$ の労働時間
青色頂点 $i \rightarrow$ 赤色頂点 $j$	流量が 1 のとき、従業員 $i$ が時刻 $j$ に働く 流量が 0 のとき、従業員 $i$ が時刻 $j$ に働くない
赤色頂点 $j \rightarrow$ ゴール $t$	時刻 $j$ に勤務している社員数

### [ポイント2]

すべての時刻について  $M$  人が勤務している場合に限り、フローの総流量は  $24 \times M$  になる<sup>※4</sup>。

ポイント 1 の例を下図に示します（赤い数字は流量）。この図のようにフローを流した場合、たとえば従業員 2 は（時刻 1・3 に）2 時間労働することに対応します。



## ◆ 実装について

ここまでアイデアを実装すると次ページの解答例のようになります。MaximumFlow クラスでは頂点番号を整数にする必要があるため、青色頂点の番号を  $1 \sim N$ 、赤色頂点の番号を  $N + 1 \sim N + 24$ 、スタート地点の番号を  $N + 25$ 、ゴール地点の番号を  $N + 26$  に再設定していることに注意してください。

※4 ここで、 $M + 1$  人以上を勤務させるのは明らかに無駄であることに注意してください

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 // MaximumFlow クラスは本の 9.8 節 (393~394 ページ) 参照
7 int N, M;
8 char C[59][24];
9 MaximumFlow Z;
10
11 int main() {
12     // 入力
13     cin >> N >> M;
14     for (int i = 1; i <= N; i++) {
15         for (int j = 0; j <= 23; j++) cin >> C[i][j];
16     }
17
18     // グラフを作る (前半パート)
19     Z.init(N + 26);
20     for (int i = 1; i <= N; i++) {
21         Z.add_edge(N + 25, i, 10); // 従業員は 10 時間までしか働けない
22     }
23     for (int i = 0; i <= 23; i++) {
24         Z.add_edge(N + i, N + 26, M); // シフトは M 人以上欲しい
25     }
26
27     // グラフを作る (後半パート)
28     for (int i = 1; i <= N; i++) {
29         for (int j = 0; j <= 23; j++) {
30             if (C[i][j] == '1') Z.add_edge(i, N + j, 1);
31         }
32     }
33
34     // 答えを求める
35     int Answer = Z.max_flow(N + 25, N + 26);
36     if (Answer == 24 * M) cout << "Yes" << endl;
37     else cout << "No" << endl;
38     return 0;
39 }
```

※Python のコードはサポートページをご覧ください

# 力試し問題前半

## (問題 1~10)

力試し問題 1	150
力試し問題 2	151
力試し問題 3	152
力試し問題 4	154
力試し問題 5	156
力試し問題 6	158
力試し問題 7	160
力試し問題 8	162
力試し問題 9	164
力試し問題 10	166

消費税率は 10% ですので、税抜価格が  $N$  円のときの税込価格は  $1.1N$  円です<sup>※1</sup>。したがって、 $1.1 * N$  を出力する以下のプログラムを提出すると、正解が得られます。

## ◆ 実装上の注意

C++ の場合、`(int)(1.1 * N)` のように整数型に変換してから出力しなければ不正解となる可能性があります。なぜなら、たとえば  $N = 1000000$  のような大きな値の場合、`1.1e+06` のように指数表記で出力されるからです。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     // 入力
6     int N;
7     cin >> N;
8
9     // 出力
10    cout << (int)(1.1 * N) << endl;
11    return 0;
12 }
```

※Python のコードはサポートページをご覧ください

※1 制約より  $N$  は 100 の倍数であるため、税込価格  $1.1N$  は必ず整数になります。

この問題は、**選ぶ 2 つのボールを全探索**することによって、計算量  $O(N^2)$  で解くことができます。

実装例を以下に示します。このプログラムでは、選ぶボールのうち 1 つ目の番号を  $i$  とし、2 つ目の番号を  $j$  としています。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 int N, A[109];
6 int Answer = 0;
7
8 int main() {
9     // 入力
10    cin >> N;
11    for (int i = 1; i <= N; i++) cin >> A[i];
12
13    // 答えを求める (全探索)
14    for (int i = 1; i <= N; i++) {
15        for (int j = i + 1; j <= N; j++) Answer = max(Answer, A[i] + A[j]);
16    }
17
18    // 出力
19    cout << Answer << endl;
20    return 0;
21 }
```

※Python のコードはサポートページをご覧ください

## ◆ 想定される別解

実は、「最も重いボール」と「2 番目に重いボール」を選ぶのが絶対に最適になります。

そのため、 $A = [A_1, \dots, A_N]$  を大きい順にソートし、 $A_1 + A_2$  を出力するという方法もあります。計算量は  $O(N \log N)$  であり、全探索よりも高速です。

この問題は、次のようなアルゴリズムで解くことができます。

### [手順1]

- まず、各  $i$  に対して  $i$  日目の株価  $\text{Price}[i]$  を計算する。

### [手順2]

- その後、それぞれの質問に答える。
- $S_j, T_j$  日目の株価のどちらが高いかは、 $\text{Price}[S[j]] < \text{Price}[T[j]]$  かどうかで判定できる。

ここで  $\text{Price}[i]$  の値は、以下の式にしたがって 1 日目から順番に計算すると求められます。計算方法が、累積和を高速に計算するとき（本の 54 ページ参照）とよく似ていますね。

- $\text{Price}[1] = A[1]$
- $\text{Price}[i] = \text{Price}[i-1] + A[i] \ (i >= 2)$

※ここで、 $A[i]$  は  $i$  日目の前日比

以上の内容を実装すると解答例のようになります。計算量は  $O(N + Q)$  です。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 using namespace std;
3
4 // 入力で与えられる変数
5 int D, X, A[200009];
6 int Q, S[200009], T[200009];
7
8 // 各日の株価
9 int Price[200009];
10
11 int main() {
12     // 入力
13     cin >> D >> X;
14     for (int i = 2; i <= D; i++) cin >> A[i];

```

```
15     cin >> Q;
16     for (int i = 1; i <= Q; i++) cin >> S[i] >> T[i];
17
18     // 各日の株価を求める（累積和）
19     Price[1] = X;
20     for (int i = 2; i <= D; i++) Price[i] = Price[i - 1] + A[i];
21
22     // 答えを出力する
23     for (int i = 1; i <= Q; i++) {
24         if (Price[S[i]] > Price[T[i]]) cout << S[i] << endl;
25         else if (Price[S[i]] < Price[T[i]]) cout << T[i] << endl;
26         else cout << "Same" << endl;
27     }
28     return 0;
29 }
```

※Python のコードはサポートページをご覧ください

## 4

# 問題 C04 : Divisor Enumeration

(Lv. 20)

まず考えられる方法は、「1 は  $N$  の約数か？」 「2 は  $N$  の約数か？」 といったように一つずつ調べていく方法です。しかし、計算量は  $O(N)$  と遅く、本問題の制約では実行時間制限に間に合いません。

	約数		約数		約数		約数
1	○	10	×	19	×	28	×
2	○	11	×	20	×	29	×
3	○	12	○	21	×	30	×
4	○	13	×	22	×	31	×
5	×	14	×	23	×	32	×
6	○	15	×	24	×	33	×
7	×	16	×	25	×	34	×
8	×	17	×	26	×	35	×
9	○	18	○	27	×	36	○

( $N = 36$  の場合)  
全探索すると  
時間がかかる…

## ◆ 効率的な解法

実は、1 から  $N$  までを全部調べなくても、1 から  $\sqrt{N}$  までを調べれば、すべての約数を列挙することができます。

なぜなら、整数  $i$  が約数であると分かったら、整数  $N/i$  も約数であることが分かるからです ( $i \times (N/i) = N$  であるため) <sup>※2</sup>。たとえば  $N = 36$  のとき、3 が約数とわかれば 12 も約数であると分かります。

	約数
1	○
2	○
3	○
4	○
5	×
6	○

- $36 \div 1 = 36$  も約数！
- $36 \div 2 = 18$  も約数！
- $36 \div 3 = 12$  も約数！
- $36 \div 4 = 9$  も約数！
- $36 \div 6 = 6$  も約数！

36 の約数は  
1, 2, 3, 4, 6, 9, 12, 18, 36  
効率的に列挙できた！

※2 もしかしたら、 $\sqrt{N}$  まで調べても、 $i$  にも  $N/i$  にも含まれない約数があるのではないかと思う方もいるかもしれません。しかし、 $N$  のすべての「 $\sqrt{N}$  を超える約数  $A$ 」について、 $A \times B = N$  を満たす整数  $B$  ( $\sqrt{N}$  以下) が存在するため、そんなことは絶対にあり得ません。たとえば  $N = 36$  の場合、 $9 \times 4 = 36$  より、約数 9 は 4 を調べる時点で見つかってしまいます

したがって、以下のようなプログラムにより、すべての約数を列挙することができます。計算量は  $O(\sqrt{N})$  です。

なお、この問題では約数を小さい順に出力しなければ不正解となることに注意してください（そのため、21 行目ではソートを使っています）。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 int main() {
7     // 入力
8     long long N;
9     cin >> N;
10
11     // 約数を列挙
12     vector<long long> Yakusuu;
13     for (long long i = 1; i * i <= N; i++) {
14         if (N % i == 0) {
15             Yakusuu.push_back(i);
16             if (i != N / i) Yakusuu.push_back(N / i);
17         }
18     }
19
20     // 小さい順にソート
21     sort(Yakusuu.begin(), Yakusuu.end());
22
23     // 出力
24     for (int i = 0; i < Yakusuu.size(); i++) {
25         cout << Yakusuu[i] << endl;
26     }
27     return 0;
28 }
```

※Python のコードはサポートページをご覧ください

## 5

## 問題 C05 : Lucky Numbers

(Lv. 30)

まず、 $N$  番目のラッキー数は「 $N - 1$  の 2 進法表記の 0,1 を 4,7 に書き換えた値」となります。40 番目までを以下に示します※3。

$N$	$N$ 番目のラッキー数	$N - 1$ の 2 進法
1	4444444444	0000000000
2	4444444447	0000000001
3	4444444474	0000000010
4	4444444477	0000000011
5	4444444744	0000000100
6	4444444747	0000000101
7	4444444774	0000000110
8	4444444777	0000000111
9	4444447444	0000001000
10	4444447447	0000001001
11	4444447474	0000001010
12	4444447477	0000001011
13	4444447744	0000001100
14	4444447747	0000001101
15	4444447774	0000001110
16	4444447777	0000001111
17	4444474444	0000010000
18	4444474447	0000010001
19	4444474474	0000010010
20	4444474477	0000010011

$N$	$N$ 番目のラッキー数	$N - 1$ の 2 進法
21	4444474744	0000000100
22	4444474747	0000000101
23	4444474774	0000000110
24	4444474777	0000010111
25	4444477444	0000011000
26	4444477447	0000011001
27	4444477474	0000011010
28	4444477477	0000011011
29	4444477744	0000011100
30	4444477747	0000011101
31	4444477774	0000011110
32	4444477777	0000011111
33	4444744444	0000100000
34	4444744447	0000100001
35	4444744474	0000100010
36	4444744477	0000100011
37	4444744744	0000100100
38	4444744747	0000100101
39	4444744774	0000100110
40	4444744777	0000100111

したがって、次ページの解答例のような実装をすると、正解が得られます。  
10 進法を 2 進法に変換する方法が分からぬ方は、本の 36 ページに戻って復習しましょう。

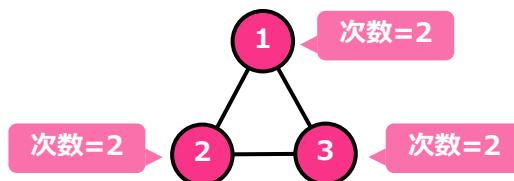
※3 「なぜ 2 進法というアイデアがひらめくのか？」と思った方は、ラッキー数で使われる数が 4, 7 ではなく 0, 1 である場合に単純化して考えてみると良いでしょう

## ◆ 解答例 (C++)

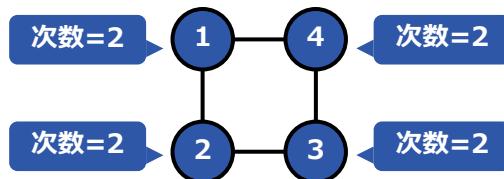
```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     // 入力
6     int N;
7     cin >> N; N -= 1;
8
9     // 出力
10    for (int x = 9; x >= 0; x--) {
11        int wari = (1 << x);
12        if ((N / wari) % 2 == 0) cout << "4"; // 2 進法の 0 が 4 に対応
13        if ((N / wari) % 2 == 1) cout << "7"; // 2 進法の 1 が 7 に対応
14    }
15    cout << endl;
16    return 0;
17 }
```

※Python のコードはサポートページをご覧ください

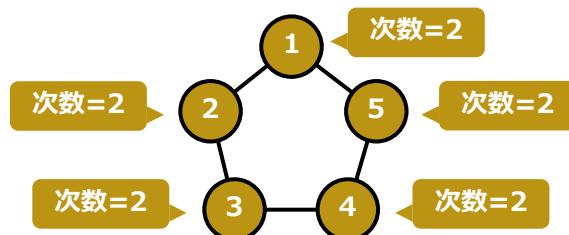
まずは  $N = 3$  の場合を解いてみましょう。少し考察すると、すべての頂点が次数 2 のグラフとして、以下のものが見つかると思います。



次に  $N = 4$  の場合を解いてみましょう。これも少し考察すると、すべての頂点が次数 2 のグラフとして、以下のものが見つかると思います。

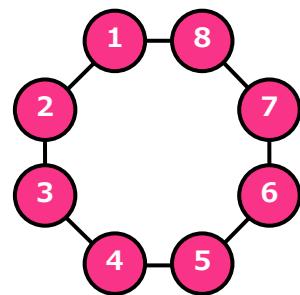
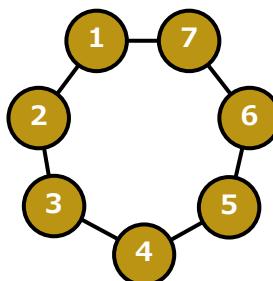
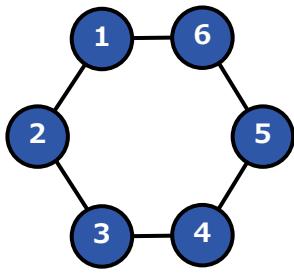


続いて  $N = 5$  の場合を解いてみましょう。これはやや難しいですが、すべての頂点が次数 2 のグラフとして、以下のものが見つかると思います。



勘の良い人であれば、この時点で「全頂点を一度ずつ通るループのような形にすれば、すべての頂点の次数が 2 になるのではないか」という予測が立つのではないでしょうか。

実際、この予測は正しいです。したがって、ループのようなグラフを出力する次ページの解答例のようなプログラムを書くと、正解となります。



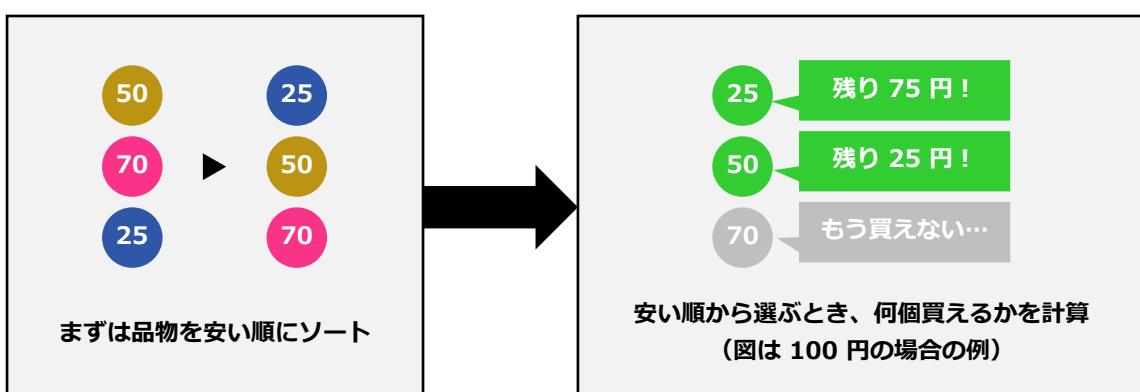
## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     // 入力
6     int N;
7     cin >> N;
8
9     // 出力
10    cout << N << endl;
11    for (int i = 1; i <= N; i++) {
12        cout << i << " " << i % N + 1 << endl;
13    }
14    return 0;
15 }
```

※Python のコードはサポートページをご覧ください

まず、限られた値段の中で最も多くの品物を買う方法は、「安い品物から順番に買っていくこと」です。

そのため、あらかじめ品物を値段の安い順にソートしておき、その後各質問に対して「どこまで買えるか」を直接計算すると、正しい答えが分かります。



しかし、計算量は質問に答える部分がボトルネックとなって  $O(NQ)$  となります。残念ながら実行時間制限に間に合いません。

## ◆ 効率的な解法

そこで、 $i$  個の品物を買うのに必要な最小の金  $\text{Price}[i]$  を前もって計算することを考えましょう。

$A_1, \dots, A_N$  が小さい順にソートされているとき、 $\text{Price}[i] = A[1] + \dots + A[i]$  となるため、 $\text{Price}[i]$  の値は累積和を求めるとき（本の 54 ページ）と同じようにして計算することができます。

すると、 $X[j]$  円以内で買える品物の個数の最大値は、 $\text{Price}[i] \leq X[j]$  を満たす最大の  $i$  となります。そしてこの値は、配列の二分探索によって計算量  $O(\log N)$  で計算することができます。

以上の考察により、アルゴリズム全体の計算量が  $O((N + Q) \log N)$  まで削減されました。本問題の制約の上限値である  $N, Q = 100000$  でも、もちろん実行時間制限に間に合います。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 long long N, C[100009], S[100009];
6 long long Q, X[100009];
7
8 int main() {
9     // 入力
10    cin >> N;
11    for (int i = 1; i <= N; i++) cin >> C[i];
12    cin >> Q;
13    for (int i = 1; i <= Q; i++) cin >> X[i];
14
15    // C[i] を小さい順にソート
16    sort(C + 1, C + N + 1);
17
18    // 累積和 S[i] をとる
19    // S[i] は「i 個の品物を買うときの最小金額」
20    S[0] = 0;
21    for (int i = 1; i <= N; i++) S[i] = S[i - 1] + C[i];
22
23    // 質問に答える
24    for (int i = 1; i <= Q; i++) {
25        int pos = upper_bound(S, S + N + 1, X[i]) - S;
26        cout << pos - 1 << endl;
27    }
28    return 0;
29 }
```

※Python のコードはサポートページをご覧ください

この問題は全探索によって解くことができます。具体的には、0000～9999の全パターンについて、「もし当選番号が△△△△ならば抽選券  $N$  枚の等級はすべて正しいか？」ということを調べれば良いです。

実装は少し大変ですが、C++ の場合は数値を文字列に変換する関数 `to_string` などを使えば少し簡単になります。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 using namespace std;
5
6 int N;
7 string S[1009]; int T[1009];
8
9 // 当選番号が A2 のとき、A1 は何等かを返す
10 int Hantei(string A1, string A2) {
11     int Diff = 0;
12     for (int i = 0; i < 4; i++) {
13         if (A1[i] != A2[i]) Diff += 1;
14     }
15     if (Diff == 0) return 1; // 全く同じとき 1 等
16     if (Diff == 1) return 2; // 桁が 1 つだけ違うとき 2 等
17     return 3;
18 }
19
20 int main() {
21     // 入力
22     cin >> N;
23     for (int i = 1; i <= N; i++) cin >> S[i] >> T[i];
24
25     // 全探索
26     vector<string> Answer;
27     for (int num = 0; num <= 9999; num++) {
28         // 整数 num を 4 桁の文字列に置き換える
29         string ID = to_string(num);
30         while (ID.size() < 4) ID = "0" + ID;
31
32         // すべての情報が正しいかどうかを確認
33         bool flag = true;
```

```
34     for (int i = 1; i <= N; i++) {
35         if (Hantei(S[i], ID) != T[i]) flag = false;
36     }
37
38     // もしすべての情報が正しかった場合
39     if (flag == true) {
40         Answer.push_back(ID);
41     }
42 }
43
44 // 出力
45 if (Answer.size() != 1) {
46     cout << "Can't Solve" << endl;
47 }
48 else {
49     cout << Answer[0] << endl;
50 }
51 return 0;
52 }
```

※Python のコードはサポートページをご覧ください

まず考えられる解法は、どの日に勉強するかを全探索することです。もちろんこの解法でも最終的には正しい答えを出すことができます。しかし、全部で  $2^N$  通りを探索することになるため、計算量の面で絶望的です。

そこで、以下のような動的計画法を考えます。前の日に勉強したかどうかで場合分けするために、配列を 2 つ持つというアイデアが基本です。

### 管理する配列

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

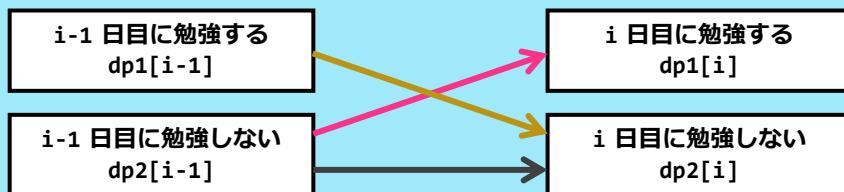
### 初期状態

1 日目が始まる前の時点では、何も実力が上がっていないため、 $dp1[0]=0$  および  $dp2[0]=0$  となる。

### 状態遷移

状態遷移のイメージは、下図に示すとおりである。

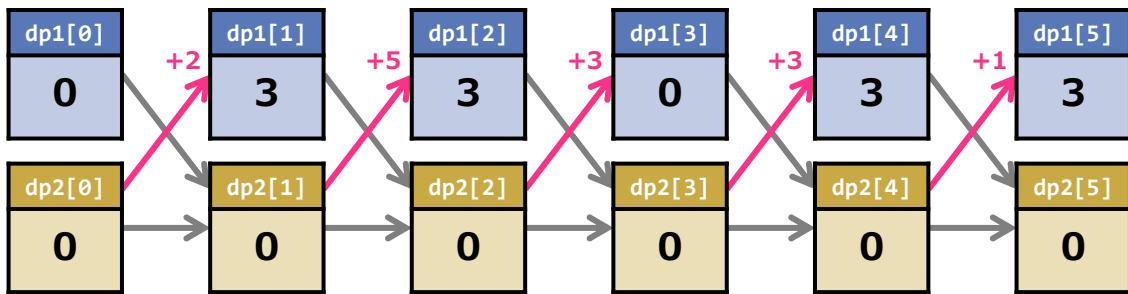
- $dp1[i] = dp2[i-1] + A[i]$
- $dp2[i] = \max(dp1[i-1], dp2[i-1])$



### 求める答え

$$\max(dp1[N], dp2[N])$$

たとえば、 $N = 5, (A_1, A_2, A_3, A_4, A_5) = (2, 5, 3, 3, 1)$  の場合、 $dp1[i]$  および  $dp2[i]$  の値は以下のようになります。



ここまで解法を実装すると、以下のようになります。計算量は  $O(N)$  と高速です。なお、本問題の制約は  $A_i \leq 10^9$  と大きく、答えが最大で  $10^{14}$  を超える可能性もあるため、C++ の場合は long long 型などの 64 ビット整数を利用する必要があることに注意してください（注：Python の場合は関係ありません）。

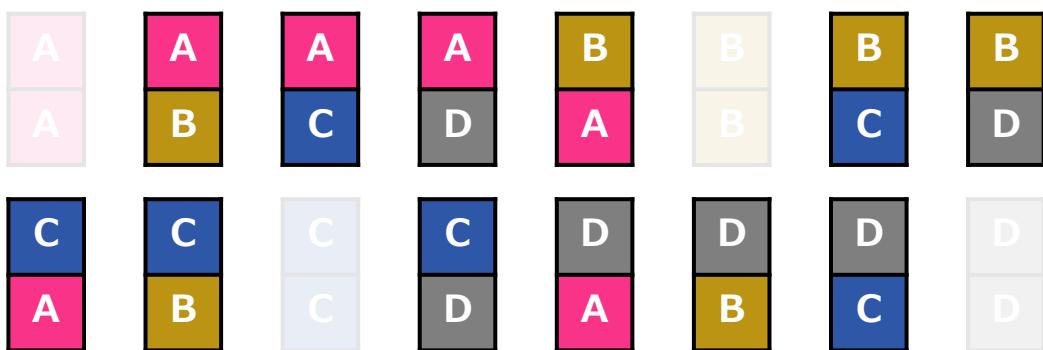
## ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 long long N, A[500009];
6 long long dp1[500009], dp2[500009];
7
8 int main() {
9     // 入力
10    cin >> N;
11    for (int i = 1; i <= N; i++) cin >> A[i];
12
13    // 配列の初期化
14    dp1[0] = 0;
15    dp2[0] = 0;
16
17    // 動的計画法
18    for (int i = 1; i <= N; i++) {
19        dp1[i] = dp2[i - 1] + A[i];
20        dp2[i] = max(dp1[i - 1], dp2[i - 1]);
21    }
22
23    // 答えを出力
24    cout << max(dp1[N], dp2[N]) << endl;
25    return 0;
26 }
```

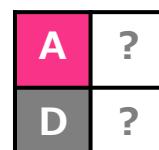
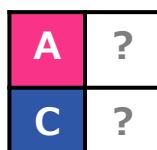
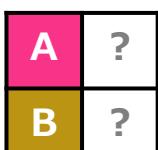
※Python のコードはサポートページをご覧ください

まずは  $W = 1$  の場合を考えましょう。マス目に色を塗る方法は全部で  $4 \times 4 = 16$  通りありますが、その中の 4 通りは「隣接する 2 マスが同じ色」になってしまふので、条件を満たす塗り方は **12 通り**です。



### ◆ $W = 2$ の場合を考える

次に  $W = 2$  の場合ですが、「どの隣接する 2 マスも同じ色にならないよう に 2 列目を塗る方法の数」は 1 列目の塗り方に依存します。そこで最初に、1 列目の塗り方が「A・B」の場合から考えましょう。

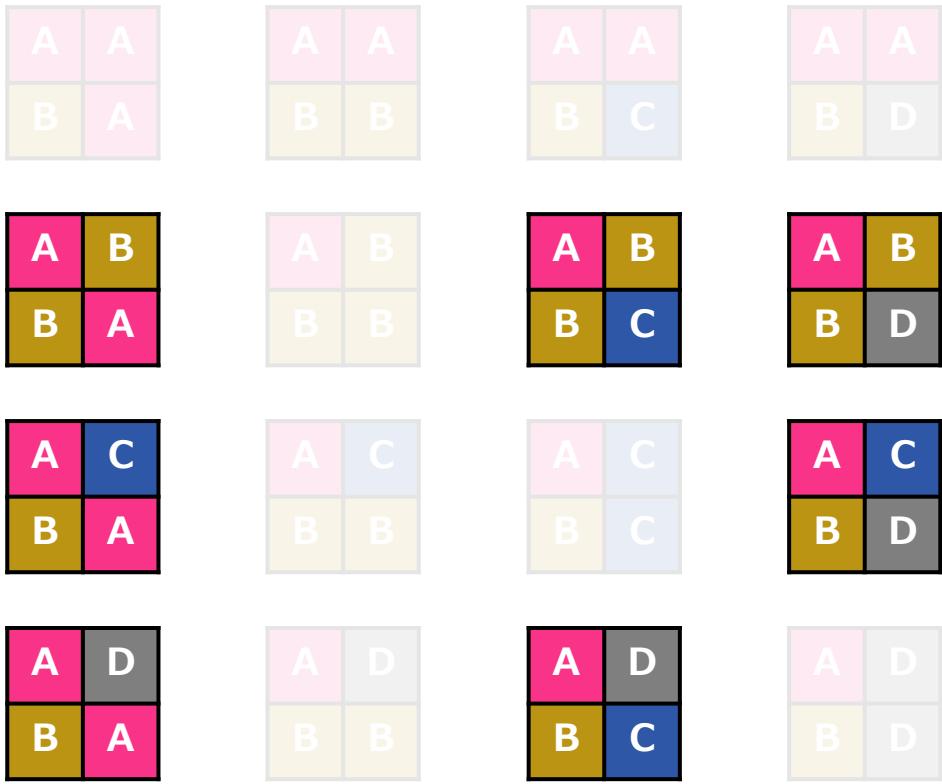


? の塗り方は何通りか  
(まずはここから考える)

? の塗り方は何通りか

? の塗り方は何通りか

次ページの図に示す通り、1 列目が「A・B」のときの 2 列目の塗り方は全部で **7 通り**あります。



## ◆ 最初の列が「A・B」以外の場合

それでは、最初の列が「A・B」以外の 11 通りについてはどうでしょうか。1 列目の 2 マスの色が異なるという点で「A・B」の場合と一致しているので、実質的には「A・B」の場合と同じように考えることができます。

したがって、全パターンについて 2 列目の塗り方は 7 通りです。このことから、 $W = 2$  のときの答えが  $12 \times 7 = 84$  通りであると分かります。

## ◆ $W$ が 3 以上の場合は？

さて、 $W = 3$  の場合はどうでしょうか。3 列目の塗り方は 2 列目の塗り方に依存しますが、先程述べたように、前の列の 2 マスが異なる色の場合、その列の塗り方は必ず 7 通りだけあります。したがって、 $W = 3$  のときの答えは  $84 \times 7 = 588$  通りです。

$W \geq 4$  の場合も同様に考えると、答えが  $12 \times 7^{W-1}$  通りであると分かります。本問題の制約は  $W \leq 10^{18}$  と非常に大きいので、答えを直接計算すると実行時間制限に間に合いませんが、繰り返し二乗法（本の 170 ページ）を使って計算すると、計算量が  $O(\log W)$  となり効率的です。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 // a の b 乗を m で割った余りを返す関数
5 long long Power(long long a, long long b, long long m) {
6     long long p = a, Answer = 1;
7     for (int i = 0; i < 60; i++) {
8         long long wari = (1LL << i);
9         if ((b / wari) % 2 == 1) {
10             Answer = (Answer * p) % m;
11         }
12         p = (p * p) % m;
13     }
14     return Answer;
15 }
16
17 int main() {
18     // 入力
19     const long long mod = 1000000007;
20     long long W;
21     cin >> W;
22
23     // 出力
24     cout << 12LL * Power(7, W - 1, mod) % mod << endl;
25     return 0;
26 }
```

※Python のコードはサポートページをご覧ください

# 力試し問題後半

## (問題 11~20)

力試し問題 11	·····	170
力試し問題 12	·····	174
力試し問題 13	·····	177
力試し問題 14	·····	181
力試し問題 15	·····	184
力試し問題 16	·····	189
力試し問題 17	·····	194
力試し問題 18	·····	197
力試し問題 19	·····	200
力試し問題 20	·····	203

この問題は、ボーダーとなる「票数÷議席数」の値で二分探索をすることで効率的に答えを求められます。

例として、入力が  $N = 4, K = 10, A = [1000000, 700000, 300000, 180000]$  であり、ボーダー<sup>※1</sup>が 140000 以上 300000 以下であることが分かっているケースを考えましょう。

## ◆ 1 回目の探索

現在の範囲 140000～300000 の中央は 220000 ですので、「ボーダーは 220000 以下か？」という質問を考えます。

もしボーダーが 220000 ちょうどである場合、獲得する議席数の合計は 8 となります。この数字は  $K = 10$  を下回っているので、ボーダーが 220000 以下であることが分かります。

**※注：ある党の獲得票数が  $a$  議席、ボーダーが  $b$  議席である場合、その党の獲得議席数は  $\lfloor a \div b \rfloor$  議席となります。**

政党1 100万÷22万=4議席	政党2 70万÷22万=3議席	政党3 30万÷22万=1議席	政党4 18万÷22万=0議席
1,000,000	700,000	300,000	180,000
500,000	350,000	150,000	90,000
333,333	233,333	100,000	60,000
250,000	175,000	75,000	45,000
200,000	140,000	60,000	36,000

※1 ここで、ボーダーは「票数÷議席数」が  $K$  位となる値のこと指します。  
 $K + 1$  位ではないことに注意してください。

## ◆ 2 回目の探索

現在の範囲 140000～220000 の中央は 180000 ですので、「ボーダーは 180000 以下か？」という質問を考えます。

もしボーダーが 180000 ちょうどである場合、獲得する議席数の合計は 10 となります。この数字は  $K = 10$  以上となっているため、ボーダーが 180000 以上であることが分かります。

政党1 100万÷18万=5議席	政党2 70万÷18万=3議席	政党3 30万÷18万=1議席	政党4 18万÷18万=1議席
1,000,000	700,000	300,000	180,000
500,000	350,000	150,000	90,000
333,333	233,333	100,000	60,000
250,000	175,000	75,000	45,000
200,000	140,000	60,000	36,000

## ◆ 3 回目の探索

現在の範囲 180000～220000 の中央は 200000 ですので、「ボーダーは 200000 以下か？」という質問を考えます。

もしボーダーが 200000 ちょうどである場合、獲得する議席数の合計は 9 となります。この数字は  $K = 10$  を下回っているため、ボーダーが 200000 以下であることが分かります。

政党1 100万÷20万=5議席	政党2 70万÷20万=3議席	政党3 30万÷20万=1議席	政党4 18万÷20万=0議席
1,000,000	700,000	300,000	180,000
500,000	350,000	150,000	90,000
333,333	233,333	100,000	60,000
250,000	175,000	75,000	45,000
200,000	140,000	60,000	36,000

このように、たった 3 回の探索で、ボーダーの範囲を 180000～200000 まで絞ることができました。

## ◆ 何回の探索が必要か？

それでは、二分探索を使って各政党の議席数を正しく求めるためには、何回の探索が必要なのでしょうか。

まず、本問題の制約より、最初の時点であり得る「ボーダーの値」は 1 以上  $10^9$  以下 となります。

また、ボーダーと次点の相対誤差は  $10^{-6}$  を超えるため、あり得るボーダーの範囲が  $10^{-6}$  を下回るようになれば、確実に正しい議席数を求めることができるといえます。※2

そこで、範囲の大きさが  $10^9$  であったところを  $10^{-6}$  にするためには、範囲を  $10^{15}$  分の 1 に縮める必要があるため、少なくとも  $\log_2 10^{15} \approx 50$  回の探索が必要であるといえます。

さて、50 回といえば多いように感じるかもしれません、1 回の探索に必要な計算量は  $O(N)$  です。制約より  $N \leq 100000$  であるため、十分余裕をもつて実行時間制限に間に合います。

## ◆ 実装について

最後に C++ の実装例を以下に示します。プログラム中の関数 `check(x)` は、ボーダーとなる「票数÷議席数」が  $x$  であるときの合計獲得議席数を返すものとなっています。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 long long N, K;
6 double A[100009];
7
8 // 割り算の値が x であるときの議席数は？
9 long long check(double x) {
10     long long sum = 0;
11     for (int i = 1; i <= N; i++) sum += (long long)(A[i] / x);
12     return sum;
13 }
```

※2 最悪のケースは、ボーダーが  $1 + 10^{-6}$ 、次点が 1 である場合です。

```
15 int main() {
16     // 入力
17     cin >> N >> K;
18     for (int i = 1; i <= N; i++) cin >> A[i];
19
20     // 二分探索
21     double Left = 1, Right = 1000000000, Mid;
22     double Border = 0; // 現在のボーダー (合計議席数が K 以上となった最大の値)
23     for (int i = 1; i <= 60; i++) {
24         Mid = (Left + Right) / 2.0;
25
26         // 割り算の値は Mid より大きいか?
27         long long val = check(Mid);
28         if (val >= K) {
29             Left = Mid;
30             Border = max(Border, Mid);
31         }
32         else {
33             Right = Mid;
34         }
35     }
36
37     // 出力
38     for (int i = 1; i <= N; i++) {
39         if (i >= 2) cout << " ";
40         cout << (long long)(A[i] / Border);
41     }
42     cout << endl;
43     return 0;
44 }
```

※Python のコードはサポートページをご覧ください

まず考えられる方法は、小説を分割する方法を全探索することです。しかし  $N = 288, K = 10$  のケースでは、全部で  $10^{16}$  通り以上を調べる必要があり、実行時間制限の面では絶望的です。

そこで、以下のような動的計画法を考えます。1 章、2 章、3 章、… の順にどこで区切るかを決めていくというアイデアが基礎になっています。

### 管理する配列

$dp[i][j]$  : 現時点で  $i$  章までの割り当てが決まっており、 $i$  章の最後のページが  $j$  ページ目であることを考える。この時点での小説の良さの最大値はいくつか（以下にイメージ図を示す）。



	0 ページ	1 ページ	2 ページ	3 ページ	4 ページ	5 ページ	6 ページ
0 章まで決定	$dp[0][0]$	$dp[0][1]$	$dp[0][2]$	$dp[0][3]$	$dp[0][4]$	$dp[0][5]$	$dp[0][6]$
1 章まで決定	$dp[1][0]$	$dp[1][1]$	$dp[1][2]$	$dp[1][3]$	$dp[1][4]$	$dp[1][5]$	$dp[1][6]$
2 章まで決定	$dp[2][0]$	$dp[2][1]$	$dp[2][2]$	$dp[2][3]$	$dp[2][4]$	$dp[2][5]$	$dp[2][6]$
3 章の時点で 4 ページのとき							
3 章まで決定	$dp[3][0]$	$dp[3][1]$	$dp[3][2]$	$dp[3][3]$	$dp[3][4]$	$dp[3][5]$	$dp[3][6]$
4 章まで決定	$dp[4][0]$	$dp[4][1]$	$dp[4][2]$	$dp[4][3]$	$dp[4][4]$	$dp[4][5]$	$dp[4][6]$

### 初期状態

最初の時点では何も決まっていないので、 $dp[0][0]=0$  とする。

### 状態遷移（貰う遷移形式）

$dp[i][j]$  の状態になる方法としては、次ページの表のようなものが考えられる。ただし、 $score(l, r)$  を「 $l$  ページ目から  $r$  ページ目までを同じ章にしたときの、小説の良さの増分」とする。

i-1 章まで	i 章	小説の良さの最大値
0 ページ	1 ページ目から j ページ目	$dp[i-1][0] + score(1, j)$
1 ページ	2 ページ目から j ページ目	$dp[i-1][1] + score(2, j)$
2 ページ	3 ページ目から j ページ目	$dp[i-1][2] + score(3, j)$
:	:	:
j-2 ページ	j-1 ページ目から j ページ目	$dp[i-1][j-2] + score(j-1, j)$
j-1 ページ	j ページ目のみ	$dp[i-1][j-1] + score(j, j)$

$dp[i][j]$  の値は、表の一番右の列に書かれた値の最大値となります（for 文を使って計算量  $O(N)$  で計算できます）。

### 求める答え

$dp[K][N]$

ここまで内容を実装すると、以下の解答例のようになります。計算量は  $O(N^2 MK)$  です。本問題の制約の上限は  $N = 288, M = 50, K = 10$  であり、 $288^2 \times 50 \times 10 = 40000000$  であるため、実行時間制限の 3 秒には十分余裕をもって間に合います。

なお、 $score(l, r)$  の値を前もって計算しておくと、計算量を  $O(N^2(K + M))$  まで減らすことも可能です。興味のある方はぜひ実装してみてください。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 int N, K;
6 int M, A[59], B[59];
7 int dp[19][309];
8
9 // l ページ目から r ページ目までの間に、何個のつながりがあるか？
10 int score(int l, int r) {
11     int cnt = 0;
12     for (int i = 1; i <= M; i++) {
13         if (l <= A[i] && B[i] <= r) cnt++;
14     }
15     return cnt;
16 }
```

```
17 int main() {
18     // 入力
19     cin >> N >> M >> K;
20     for (int i = 1; i <= M; i++) cin >> A[i] >> B[i];
21
22     // 配列 dp の初期化
23     for (int i = 0; i <= K; i++) {
24         for (int j = 0; j <= N; j++) dp[i][j] = -1000000;
25     }
26
27     // 動的計画法（貰う遷移形式）
28     dp[0][0] = 0;
29     for (int i = 1; i <= K; i++) {
30         for (int j = 1; j <= N; j++) {
31             // k は「前の章がどのページで終わったか」
32             for (int k = 0; k <= j - 1; k++) {
33                 dp[i][j] = max(dp[i][j], dp[i - 1][k] + score(k + 1, j));
34             }
35         }
36     }
37
38     // 出力
39     cout << dp[K][N] << endl;
40     return 0;
41 }
```

※Python のコードはサポートページをご覧ください

この問題を解く最も単純な方法は、すべてのカードの選び方を全探索することです。選ぶ 2 枚のカードを  $A_j, A_i$  ( $1 \leq j < i \leq N$ ) とするとき、以下のように二重の for 文を使って全探索することができます。

しかし計算量が  $O(N^2)$  と遅く、 $N = 100000$  のケースでは実行時間制限に間に合いません。

```

1 #include <iostream>
2 using namespace std;
3
4 long long N, P;
5 long long A[100009];
6
7 int main() {
8     // 入力
9     cin >> N >> P;
10    for (int i = 1; i <= N; i++) cin >> A[i];
11    for (int i = 1; i <= N; i++) A[i] %= 1000000007LL; // オーバーフロー防止
12
13    long long Answer = 0;
14    for (int i = 1; i <= N; i++) {
15        for (int j = 1; j <= i - 1; j++) {
16            if (A[j] * A[i] % 1000000007LL == P) Answer += 1;
17        }
18    }
19    cout << Answer << endl;
20    return 0;
21 }
```

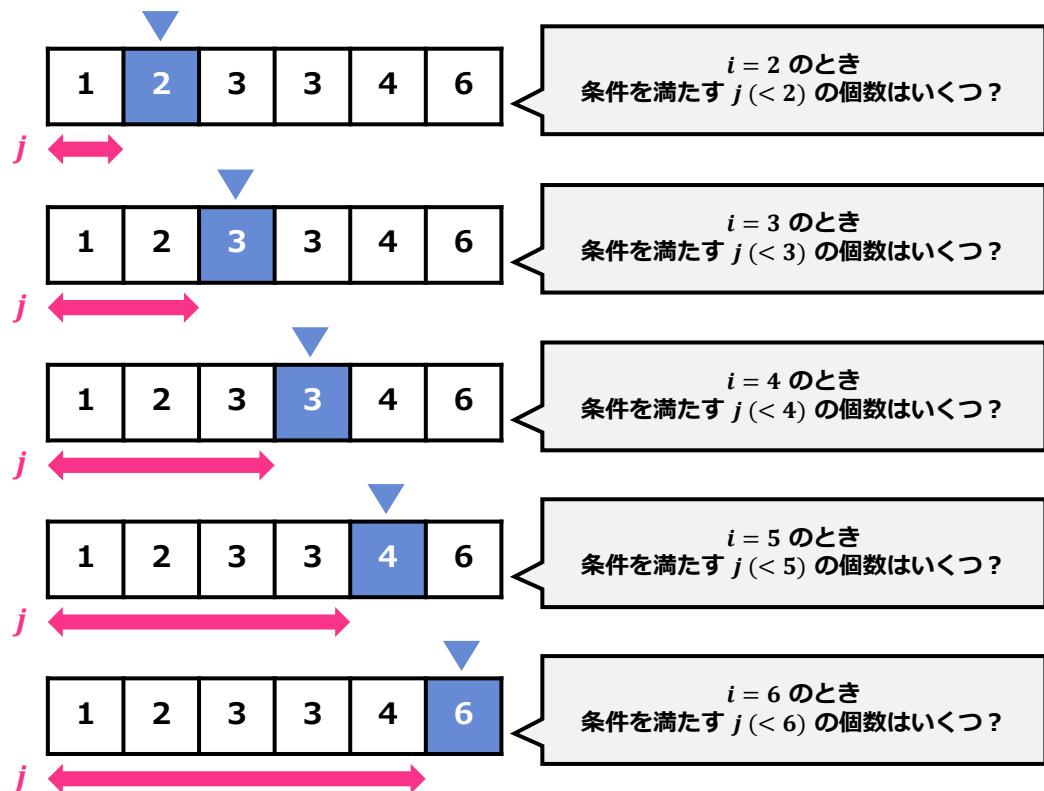
## ◆ 効率的な解法 ( $P \neq 0$ の場合)

まず、**1 枚のカードが決まればもう 1 枚も自動的に決まる**という重要な性質があります。具体的には、一方のカードの値が  $A_i$  であるとき、もう一方のカードの値は次のようにになります。

$\text{mod } 1000000007$  上での  $\lceil P \div A_i \rceil$  <sup>※3</sup>

※3 なぜなら、もう一方のカードを  $x$  とするとき、 $(A_i \times x) \text{ mod } 1000000007 = P$  を満たす必要があるからです

そこで、**大きい方の番号  $i$  だけを全探索**することを考えます。このとき、各  $i$  について「条件を満たす  $j$  の個数は何個か」をどう高速に求めるかが問題になります。 $P = 12, A = [1, 2, 3, 3, 4, 6]$  のときのイメージ図を以下に示します。



それでは、条件を満たす  $j$  の個数、すなわち  $A_j = (\text{mod } 1000000007 \text{ 上での } P \div A_i \text{ の値})$  を満たす  $j$  の個数※4を効率的に計算するにはどうすれば良いのでしょうか。

一つの方法として、問題 A54（本の 8.4 節）でやったように、 $A_j = x$  となる  $j$  の個数を**連想配列** `Count[x]` に記録するという方法があります。アルゴリズムを厳密に書き下すと、以下のとおりになります。

$i = 1, 2, \dots, N$  に対して、以下のことを行う：

- 答え `Answer` に `Count[Goal]` の値を加算する。ただし `Goal` は  $\text{mod } 1000000007$  上での  $P \div A_i$  の値である。
- その後、`Count[A[i]]` に 1 を加算する。

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。さて、これを提出すると正解になるのでしょうか。

※4 たとえば上の例で  $A_i = 6$  を選んだ場合、 $12 \div 6 = 2$  より、数えなければならないのは「 $A_j = 2$ 」となる  $j$  の個数です

```

1 #include <iostream>
2 #include <map>
3 using namespace std;
4
5 const long long mod = 1000000007;
6 long long N, P;
7 long long A[100009];
8 map<long long, long long> Count;
9
10 int main() {
11     // 入力
12     cin >> N >> P;
13     for (int i = 1; i <= N; i++) cin >> A[i];
14     for (int i = 1; i <= N; i++) A[i] %= mod;
15
16     // カード j と i を選ぶとする (j < i)
17     // 各 i に対して、何個の j で条件を満たすかを数える
18     long long Answer = 0;
19     for (int i = 1; i <= N; i++) {
20         // A[i]*Goal mod 1000000007 = P を満たす整数が Goal
21         // Division 関数は本の 175 ページ (5.5 節) を参照
22         long long Goal = Division(P, A[i], mod);
23         Answer += Count[Goal];
24         Count[A[i]] += 1;
25     }
26     cout << Answer << endl;
27     return 0;
28 }
```

残念ながら、 $A_i = 0$  のケース<sup>※5</sup>で不正解 (WA) となってしまいます。なぜなら、mod 上でも「0 で割る割り算」を行うことはできないからです<sup>※6</sup>。

そのため、 $A_i = 0$  のときに限り場合分けをする必要があります。具体的には以下のようになります。

ケース	$A_i \times A_j \text{ mod } 1000000007 = P$ となる条件	$j$ の個数
$P = 0$ のとき	$A_j$ は何でも良い	$i - 1$
$P \neq 0$ のとき	$A_j$ に関わらず絶対に条件を満たさない	0

この場合分けを入れると、プログラムは次ページの解答例のようになります。これでようやく正解に至ることができました。

※5 通常、C++ で  $4 \div 0$  のような「0 で割る割り算」を行った場合はランタイムエラーが起こります。一方、上の実装で  $\text{Division}(4, 0, \text{mod})$  などを呼び出したときはランタイムエラーにななりませんが、 $\text{Goal}$  の値が 0 になるため正しい答えが求められません。

※6 厳密には、 $A_i = 3000000021$  のように  $1000000007$  で割った余りが 0 である場合も含まれます。

## 解答例 (C++)

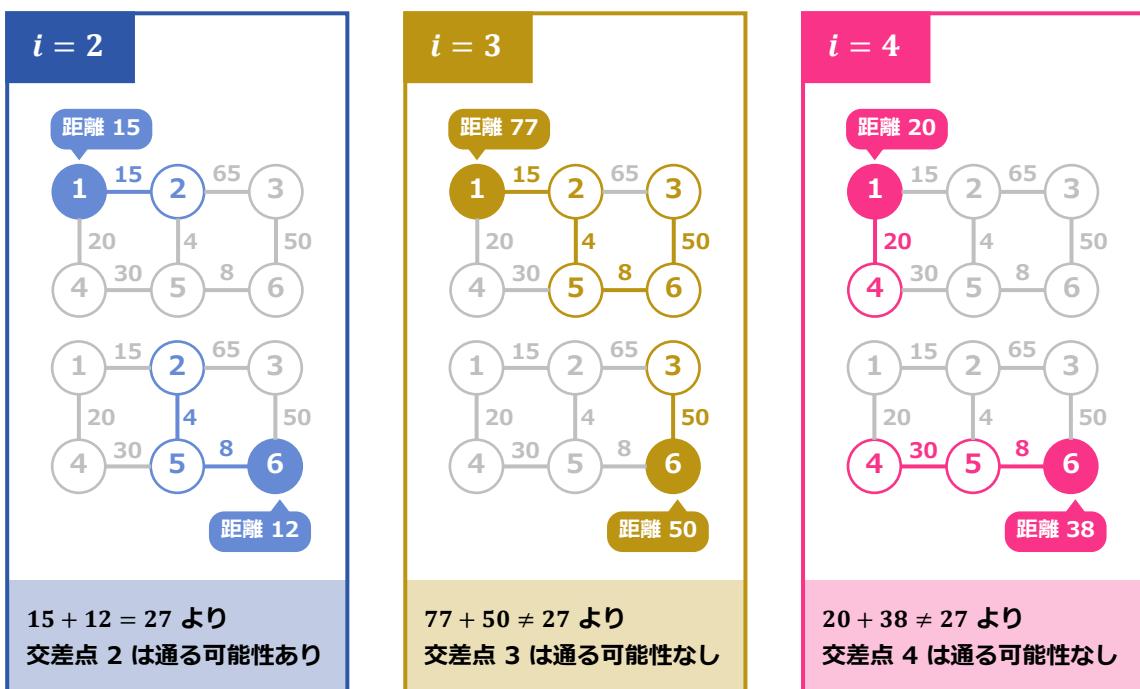
```
1 #include <iostream>
2 #include <map>
3 using namespace std;
4
5 // a の b 乗を m で割った余りを返す関数
6 long long Power(long long a, long long b, long long m) {
7     long long p = a, Answer = 1;
8     for (int i = 0; i < 30; i++) {
9         int wari = (1 << i);
10        if ((b / wari) % 2 == 1) {
11            Answer = (Answer * p) % m;
12        }
13        p = (p * p) % m;
14    }
15    return Answer;
16 }
17
18 // a ÷ b を m で割った余りを返す関数
19 long long Division(long long a, long long b, long long m) {
20     return (a * Power(b, m - 2, m)) % m;
21 }
22
23 const long long mod = 1000000007;
24 long long N, P;
25 long long A[100009];
26 map<long long, long long> Count;
27
28 int main() {
29     // 入力
30     cin >> N >> P;
31     for (int i = 1; i <= N; i++) cin >> A[i];
32     for (int i = 1; i <= N; i++) A[i] %= mod;
33
34     // カード j と i を選ぶとする (j < i)
35     // 各 i に対して、何個の j で条件を満たすかを数える
36     long long Answer = 0;
37     for (int i = 1; i <= N; i++) {
38         if (A[i] == 0) {
39             if (P == 0) Answer += (i - 1);
40             else Answer += 0;
41         }
42         else {
43             // A[i]*Goal mod 1000000007 = P を満たす整数が Goal
44             long long Goal = Division(P, A[i], mod);
45             Answer += Count[Goal];
46         }
47         Count[A[i]] += 1;
48     }
49     cout << Answer << endl;
50     return 0;
51 }
```

※Python のコードはサポートページをご覧ください

まず、交差点  $p$  と交差点  $q$  の最短経路長を  $\text{dist}(p, q)$  とするとき、交差点  $i$  を通る可能性があることは、以下の条件を満たすことと同じです。

$$\text{dist}(1, N) = \text{dist}(1, i) + \text{dist}(i, N)$$

自動採点システムの入力例に対応した具体例を以下に示します（注：交差点 1 から 6 までの最短経路長は 27 です）。



したがって、以下の 2 つを前もってダイクストラ法で計算しておくと、交差点  $i$  を通る可能性があるかどうかを計算量  $O(1)$  で判定できます。※7。

- ・ 交差点 1 から各交差点までの最短経路長  $\text{dist1}[i]$
- ・ 交差点  $N$  から各交差点までの最短経路長  $\text{distN}[i]$

実装例を次ページに示します。プログラム全体の計算量は  $O(M \log N)$  です。

※7 ここで、「交差点  $i$  から交差点  $N$  までの最短経路長」は「交差点  $N$  から交差点  $i$  までの最短経路長」と一致するため、 $\text{distN}[i]$  から計算できることに注意してください（グラフが無向グラフであることが関係しています）。

## 解答例 (C++)

```
1 #include <iostream>
2 #include <queue>
3 #include <vector>
4 #include <algorithm>
5 using namespace std;
6
7 // 入力・グラフ
8 int N, M, A[100009], B[100009], C[100009];
9 vector<pair<int, int>> G[100009];
10
11 // ダイクストラ法
12 int cur[100009]; bool kakutei[100009];
13 priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>> Q;
14
15
16 // 頂点 1 からの距離、頂点 N からの距離
17 int dist1[100009];
18 int distN[100009];
19
20 // 始点 start でダイクストラ法を行う関数
21 void dijkstra(int start) {
22     // 配列の初期化
23     for (int i = 1; i <= N; i++) kakutei[i] = false;
24     for (int i = 1; i <= N; i++) cur[i] = 2000000000;
25
26     // スタート地点をキューに追加
27     cur[start] = 0;
28     Q.push(make_pair(cur[start], start));
29
30     // ダイクストラ法
31     while (!Q.empty()) {
32         // 次に確定させるべき頂点を求める
33         int pos = Q.top().second; Q.pop();
34         if (kakutei[pos] == true) continue;
35
36         // cur[x] の値を更新する
37         kakutei[pos] = true;
38         for (int i = 0; i < G[pos].size(); i++) {
39             int nex = G[pos][i].first;
40             int cost = G[pos][i].second;
41             if (cur[nex] > cur[pos] + cost) {
42                 cur[nex] = cur[pos] + cost;
43                 Q.push(make_pair(cur[nex], nex));
44             }
45         }
46     }
47 }
48
49 int main() {
50     // 入力
51     cin >> N >> M;
52     for (int i = 1; i <= M; i++) {
53         cin >> A[i] >> B[i] >> C[i];
54         G[A[i]].push_back(make_pair(B[i], C[i]));
55         G[B[i]].push_back(make_pair(A[i], C[i]));
56     }
57 }
```

```
57 // ダイクストラ法を行う
58 dijkstra(1); for (int i = 1; i <= N; i++) dist1[i] = cur[i];
59 dijkstra(N); for (int i = 1; i <= N; i++) distN[i] = cur[i];
60
61 // 答えを求める
62 int Answer = 0;
63 for (int i = 1; i <= N; i++) {
64     if (dist1[i] + distN[i] == dist1[N]) Answer += 1;
65 }
66
67 // 出力
68 cout << Answer << endl;
69
70 }
```

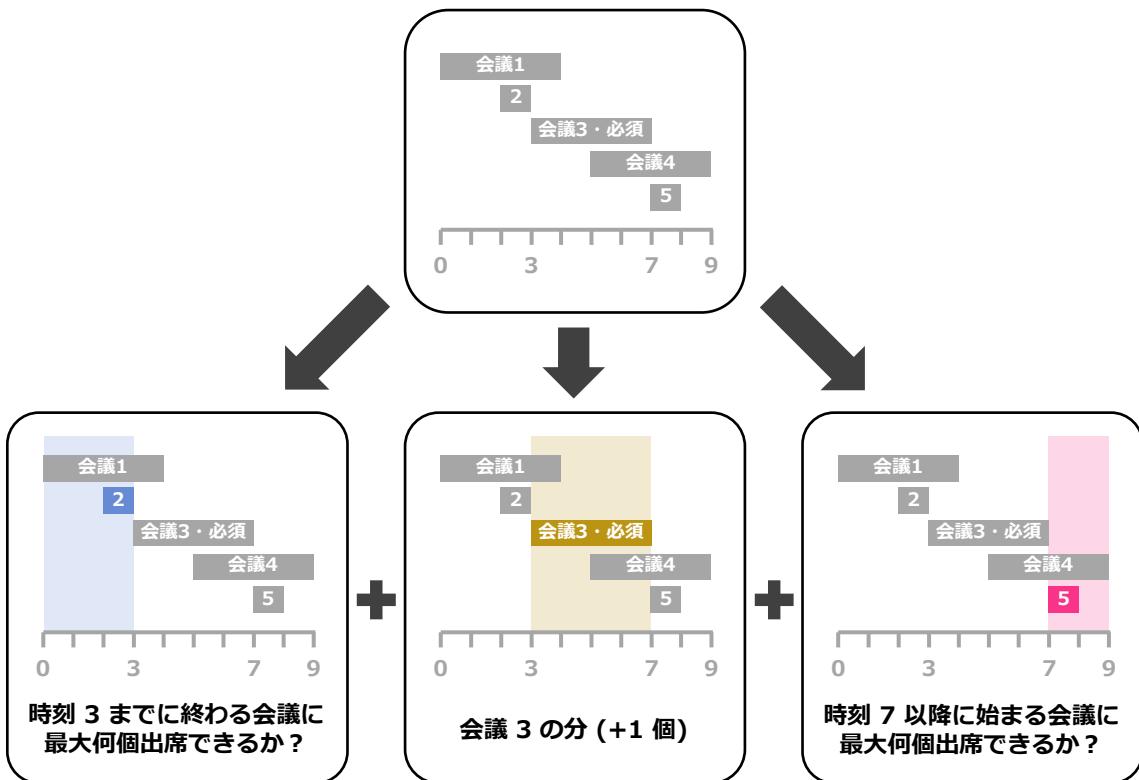
※Python のコードはサポートページをご覧ください

この問題では、空けるべき最小間隔  $K$  [秒] が設定されていることがあります。しかし、すべての会議の終了時刻  $R_i$  を  $K$  秒遅らせれば、 $K = 0$  の場合に帰着させることができます。本解説では  $K = 0$  を仮定します。

### ◆ 具体例を考えよう (1)

手始めに、入力が  $N = 5, K = 0, (L_i, R_i) = (0, 4), (1, 2), (3, 7), (5, 9), (7, 8)$  であり、3番目の会議に絶対出席しなければならない場合を考えてみましょう。

まず、出席できる会議の個数の最大値は、(時刻 3までで最大何個の会議に出席できるか) + 1 + (時刻 7以降で最大何個の会議に出席できるか) という式で表されます。イメージ図を以下に示します。



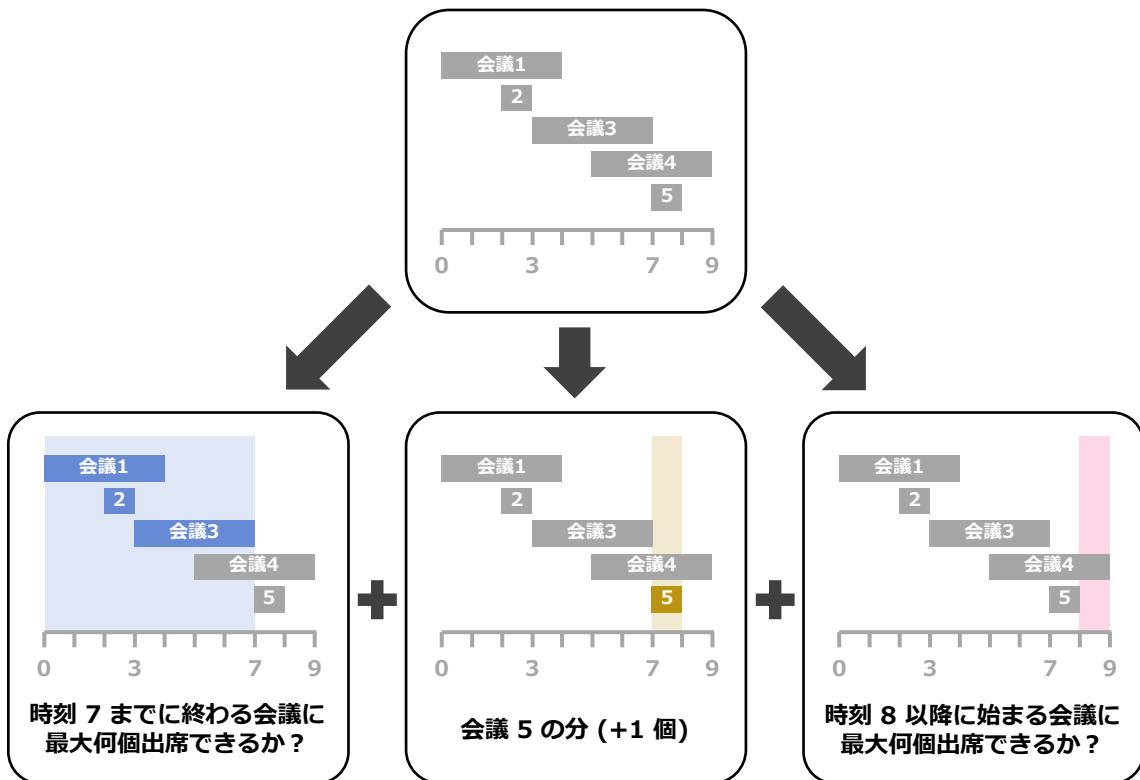
そこで、青線部分の値は 1 であり、赤線部分の値は 1 であるため、答えは  $1+1+1=3$  となります。

## ◆ 具体例を考えよう (2)

次に、5番目の会議に絶対出席しなければならない場合を考えましょう。まず答えは以下の式で表されます。

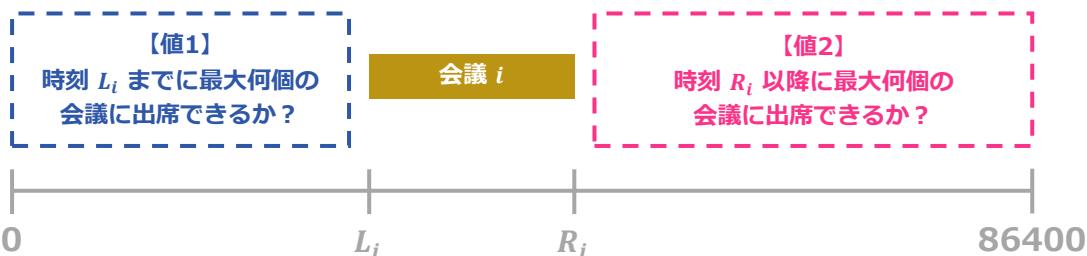
$$\begin{aligned} & (\text{時刻 } 7 \text{まで最大何個の会議に出席できるか}) + 1 \\ & + (\text{時刻 } 8 \text{以降で最大何個の会議に出席できるか}) \end{aligned}$$

そこで、青線部分の値は2、赤線部分の値は0であるため、答えは $2+1+0=3$ となります。



## ◆ 一般のケースと計算量

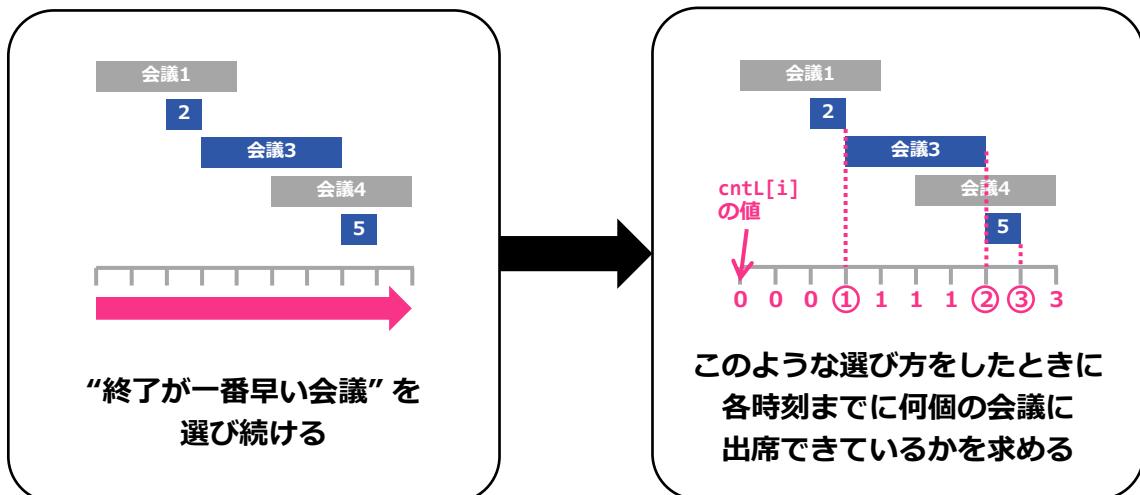
一般的なケースでも同じように解くことができます。具体的には、「時刻  $L_i$  から始まり、時刻  $R_i$  に終わる会議に絶対出席しなければならない」というシナリオの場合、以下の2つの値を計算することで答えが分かります。



そして、2つの値は区間スケジューリング問題を解くことで計算できます。しかし計算量は  $O(N^2 \log N)$  となり<sup>※8</sup>、実行時間制限に間に合いません。

## ◆ 解法を工夫しよう

そこで、 $i = 0, 1, \dots, 86400$  に対して、時刻  $i$  までに最大何個の会議に出席できるか  $\text{cntL}[i]$  を計算します（計算方法を以下に示します<sup>※9</sup>）。



同様に、 $i = 0, 1, \dots, 86400$  に対して、時刻  $i$  以降に最大何個の会議に出席できるか  $\text{cntR}[i]$  を計算します（この値は、先程の方法を右からやるようにして計算することができます）。

すると、時刻  $L_i$  から始まり、時刻  $R_i$  に終わる会議に絶対出席しなければならないというシナリオに対する答えは

$$\text{cntL}[L[i]] + 1 + \text{cntR}[R[i]]$$

となり、計算量  $O(1)$  で求められます。

## ◆ 実装について

最後に実装例を次ページに示します。プログラム全体の計算量は、ソートがボトルネックとなり  $O(N \log N)$  です。

※8 区間スケジューリング問題を1回解くのに必要な計算量は  $O(N \log N)$  ですが、今回の問題では  $i = 1, 2, \dots, N$  について「 $i$ 番目の会議に出席しなければならないときの、出席できる会議の最大数」を求める必要があります。

※9 終了が一番早い会議を選び続けたとき、どの時刻  $i$  に対しても「時刻  $i$  までに出席できる会議の数」が最大になります。

## 解答例 (C++)

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 using namespace std;
5
6 // 入力で与えられる変数
7 int N, K;
8 int L[100009], R[100009];
9
10 // 時刻 i までに何個出席できるか cntL[i] / 時刻 i から何個出席できるか cntR[i]
11 int cntL[200009];
12 int cntR[200009];
13
14 int main() {
15     // 入力
16     cin >> N >> K;
17     for (int i = 1; i <= N; i++) cin >> L[i] >> R[i];
18
19     // 時刻に補正をかける
20     for (int i = 1; i <= N; i++) R[i] += K;
21
22     // 左から区間スケジューリング (Part1 : ソート)
23     vector<pair<int, int>> tmp1;
24     for (int i = 1; i <= N; i++) tmp1.push_back(make_pair(R[i], L[i]));
25     sort(tmp1.begin(), tmp1.end());
26
27     // 左から区間スケジューリング (Part2 : 貪欲法)
28     int CurrentTime1 = 0; // 現在時刻
29     int Num1 = 0; // 現在出席した会議の個数
30     for (int i = 0; i < tmp1.size(); i++) {
31         if (CurrentTime1 <= tmp1[i].second) {
32             CurrentTime1 = tmp1[i].first;
33             Num1 += 1;
34             cntL[CurrentTime1] = Num1;
35         }
36     }
37
38     // 右から区間スケジューリング (Part1 : ソート)
39     vector<pair<int, int>> tmp2;
40     for (int i = 1; i <= N; i++) tmp2.push_back(make_pair(L[i], R[i]));
41     sort(tmp2.begin(), tmp2.end());
42     reverse(tmp2.begin(), tmp2.end());
43
44     // 右から区間スケジューリング (Part2 : 貪欲法)
45     int CurrentTime2 = 200000; // 現在時刻
46     int Num2 = 0; // 現在出席した会議の個数
47     for (int i = 0; i < tmp2.size(); i++) {
48         if (CurrentTime2 >= tmp2[i].second) {
49             CurrentTime2 = tmp2[i].first;
50             Num2 += 1;
51             cntR[CurrentTime2] = Num2;
52         }
53     }
}
```

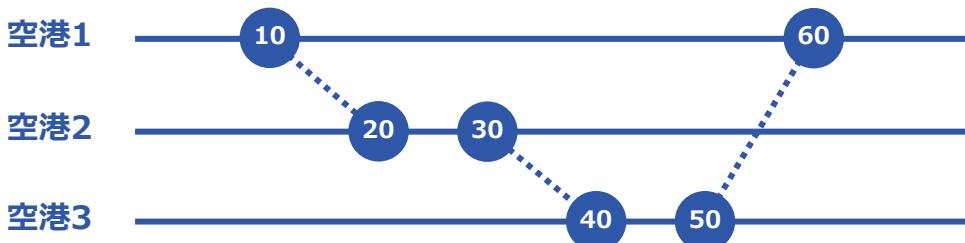
```
54 // cntL, cntR を求める
55 // ここで、補正後の R[i] の値は絶対に 200000 を超えないことに注意
56 for (int i = 1; i <= 200000; i++) cntL[i] = max(cntL[i], cntL[i - 1]);
57 for (int i = 199999; i >= 0; i--) cntR[i] = max(cntR[i], cntR[i + 1]);
58
59 // 答えを求める
60 for (int i = 1; i <= N; i++) {
61     cout << cntL[L[i]] + 1 + cntR[R[i]] << endl;
62 }
63 return 0;
64 }
```

※Python のコードはサポートページをご覧ください

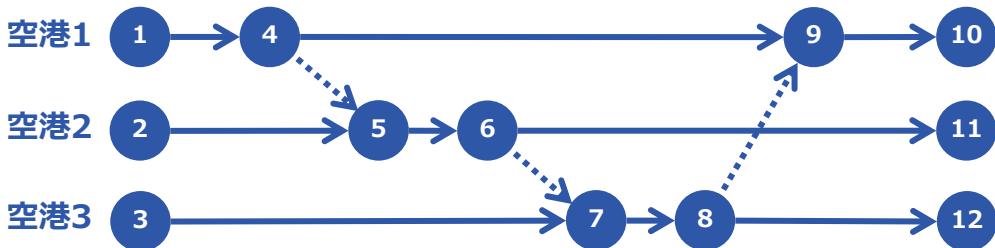
この問題では、必要な乗り継ぎ時間  $K$  が設定されていることがあります。しかし、すべての飛行機の到着時刻  $T_i$  を  $K$  秒遅らせれば、 $K = 0$  の場合に帰着させることができるので、本解説では  $K = 0$  を仮定します。

### ◆ ステップ 1：飛行機をグラフで表す

まず、その日に飛ぶ飛行機の情報は、以下のようなダイヤグラムで表すことができます（マルの中に書かれた数字は時刻）。

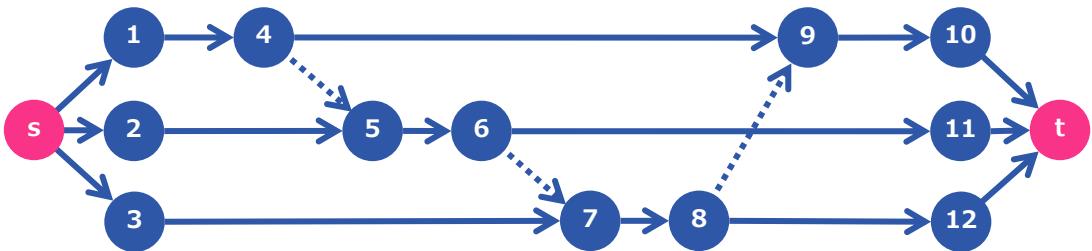


そこで、このダイヤグラムにおける出発時刻・到着時刻を頂点とすると、以下のようないわゆるグラフに変換することができます。実線は空港で待つこと、点線は飛行機で移動することに対応します。



そして、この問題の答えである「乗れる飛行機の数の最大値」は、頂点  $\{1, 2, 3\}$  のいずれかから出発し、頂点  $\{10, 11, 12\}$  のいずれかに到着する経路における、通る点線の数の最大値となります。

次に、始点  $s$  から頂点  $\{1, 2, 3\}$  に向かう実線および、頂点  $\{10, 11, 12\}$  から終点  $t$  に向かう実線を追加することを考えます。



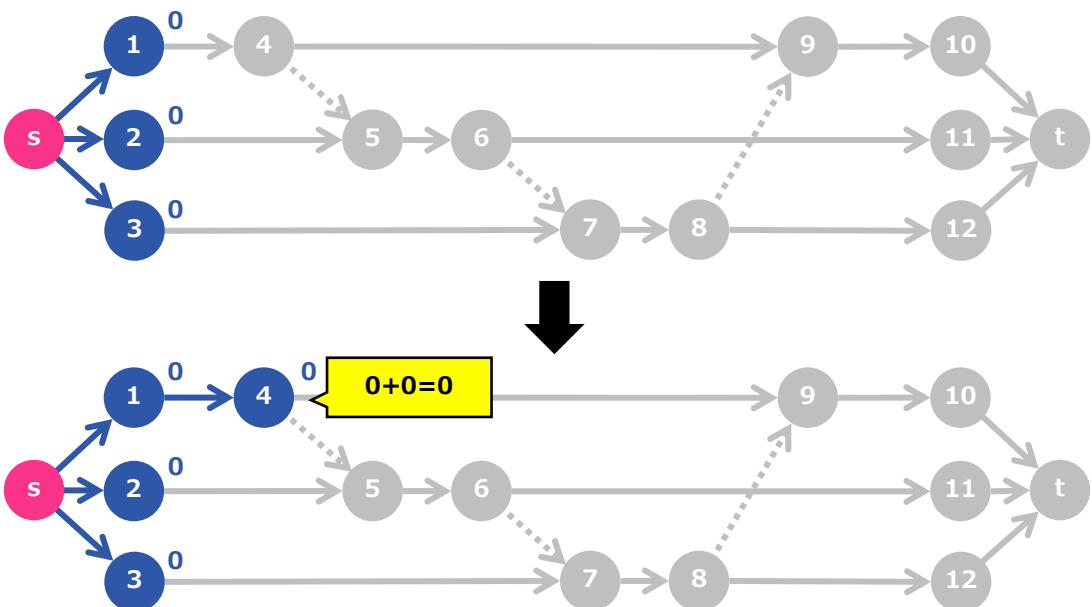
このとき、乗れる飛行機の数の最大値は、始点  $s$  から終点  $t$  まで行く経路における、通る点線の数の最大値となります。先ほどより問題がシンプルになりましたね。

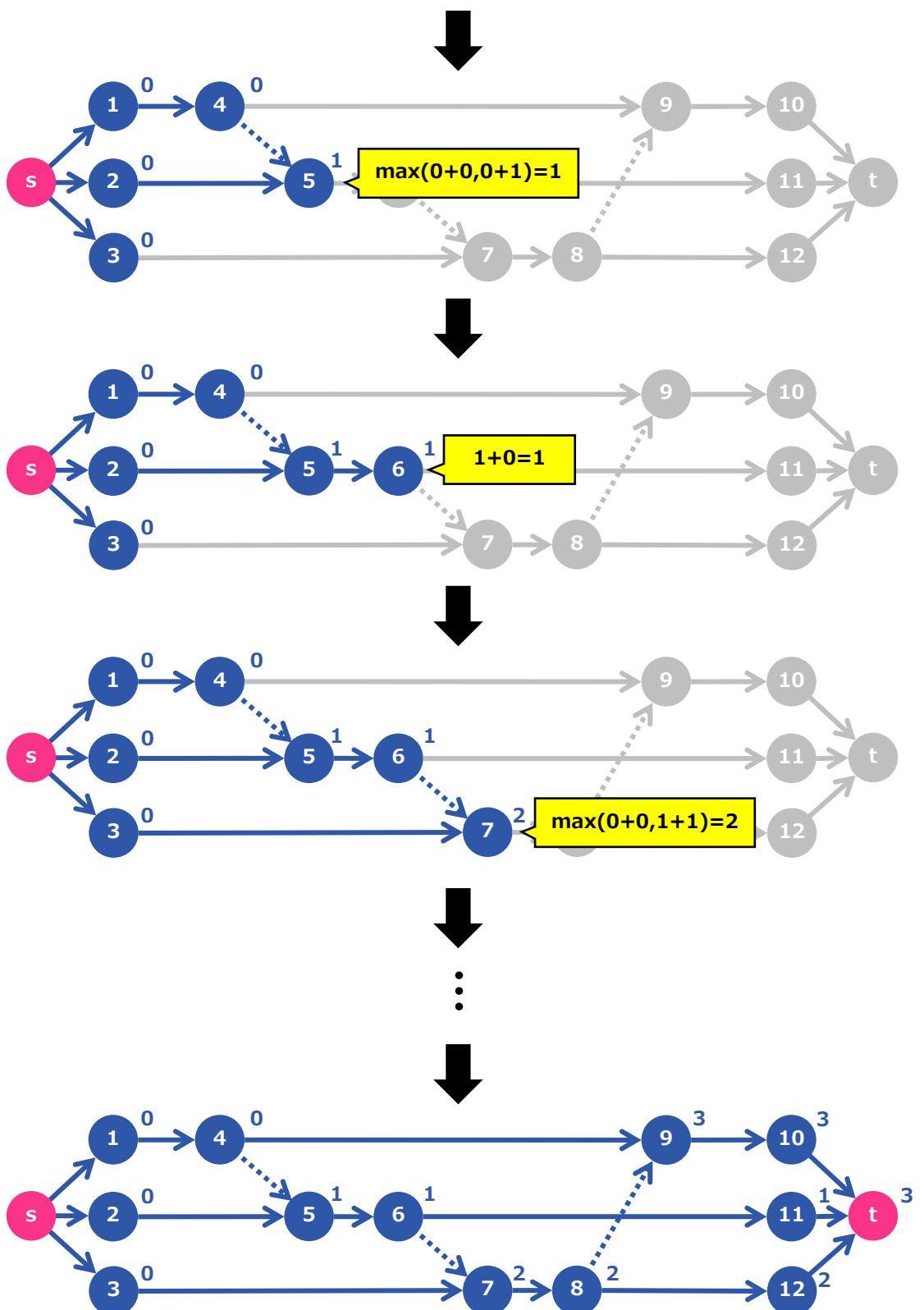
## ◆ ステップ 2：答えを求める

さて、通る点線の数の最大値はどうやって求めれば良いのでしょうか。**すべての辺は「時刻が早い方から遅い方」に向かっている**ので、時刻が早い頂点から順番に、以下の値を計算すれば良いです。

$dp[i]$  : 始点  $s$  から頂点  $i$  までたどり着くまでに、最大何本の点線を通ることができるか

上図の例に対応した計算過程を以下に示します（頂点の右上に書かれた数字が  $dp[i]$  の値です）。





## ◆ 実装について

以上のアルゴリズムを実装すると、解答例のようになります。実装上の注意点を次ページにいくつか示します。

- 動的計画法による  $dp[i]$  の計算をしやすくするため、頂点番号は時刻が早い方が小さくなるように設定しています。特に、スタートの頂点番号は 0、ゴールの頂点番号は最大値  $List.size() + 1$  です。
- 配る遷移形式で  $dp[i]$  の値を計算しやすくするため、辺を逆向きにして管理しています。また、無向グラフの代わりに、実線の重みを 0、点線の重みを 1 とした有向グラフを管理しています。
- 空港も時刻も同じ場合は、到着の方の頂点番号が小さくなるようにしています（到着と同時に出発しても良いようにするため）。

## ◆ 解答例 (C++)

```

1 #include <iostream>
2 #include <tuple>
3 #include <vector>
4 #include <algorithm>
5 using namespace std;
6
7 // 入力部分
8 int N, M, K;
9 int A[100009], B[100009], S[100009], T[100009];
10
11 // (時刻, 路線番号または空港番号, 出発か到着か)
12 // 出発 = 2 / 到着 = 1 / 最初と最後 = 0
13 // ここで、出発の方が番号が大きい理由は、同じ時刻のときに到着をより早くするため
14 vector<tuple<int, int, int>> List;
15
16 // 頂点番号の情報
17 int VertS[100009]; // 路線 i の到着
18 int VertT[100009]; // 路線 i の出発
19 vector<int> Airport[100009];
20
21 // グラフおよび dp[i]
22 vector<pair<int, int>> G[400009];
23 int dp[400009];
24
25 int main() {
26     // 入力
27     cin >> N >> M >> K;
28     for (int i = 1; i <= M; i++) {
29         cin >> A[i] >> S[i] >> B[i] >> T[i];
30         T[i] += K; // 到着時刻の補正
31     }
32
33     // 頂点となり得る(空港, 時刻)の組を「時刻の早い順に」ソート
34     for (int i = 1; i <= M; i++) List.push_back(make_tuple(S[i], i, 2));
35     for (int i = 1; i <= M; i++) List.push_back(make_tuple(T[i], i, 1));

```

```

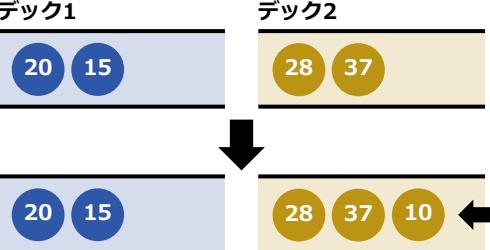
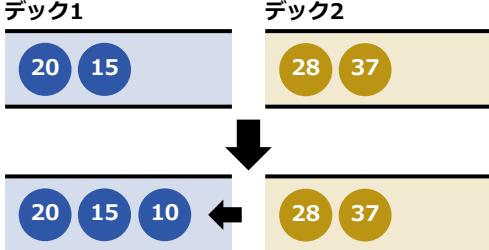
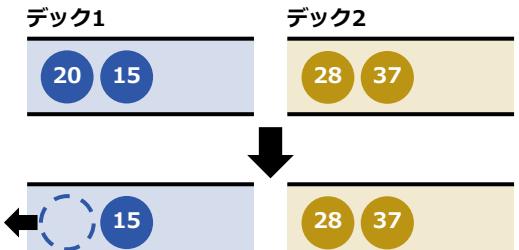
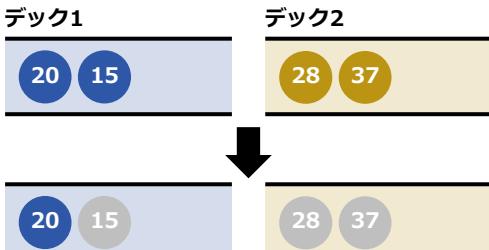
36 for (int i = 1; i <= N; i++) List.push_back(make_tuple(-1, i, 0));
37 for (int i = 1; i <= N; i++) List.push_back(make_tuple(2100000000, i, 0));
38 sort(List.begin(), List.end());
39
40 // 各路線の頂点番号を求める
41 // ここで、頂点番号は時刻の早い順に 1, 2, ..., List.size() となる
42 for (int i = 0; i < List.size(); i++) {
43     if (get<2>(List[i]) == 2) VertS[get<1>(List[i])] = i + 1;
44     if (get<2>(List[i]) == 1) VertT[get<1>(List[i])] = i + 1;
45 }
46
47 // 各空港の頂点番号を求める（空港で待つことに対応する実線を求めるときに使う）
48 for (int i = 0; i < List.size(); i++) {
49     if (get<2>(List[i]) == 0) Airport[get<1>(List[i])].push_back(i + 1);
50     if (get<2>(List[i]) == 1) Airport[B[get<1>(List[i])]].push_back(i + 1);
51     if (get<2>(List[i]) == 2) Airport[A[get<1>(List[i])]].push_back(i + 1);
52 }
53
54 // グラフを作る（辺が逆向きになっていることに注意！）
55 for (int i = 1; i <= M; i++) {
56     G[VertT[i]].push_back(make_pair(VertS[i], 1)); // 路線に対応する辺（点線）
57 }
58 for (int i = 1; i <= N; i++) {
59     for (int j = 0; j < (int)Airport[i].size() - 1; j++) {
60         int idx1 = Airport[i][j];
61         int idx2 = Airport[i][j + 1];
62         G[idx2].push_back(make_pair(idx1, 0)); // 空港で待つことに対応する辺
63     }
64 }
65
66 // グラフに始点（頂点 0）と終点（頂点 List.size()+1）を追加
67 for (int i = 1; i <= N; i++) {
68     G[Airport[i][0]].push_back(make_pair(0, 0));
69     G[List.size() + 1].push_back(make_pair(Airport[i][Airport[i].size() - 1], 0));
70 }
71
72 // 動的計画法によって dp[i] の値を求める
73 // 頂点番号は時刻の早い順になっているので、dp[1] から順に計算すれば良い
74 dp[0] = 0;
75 for (int i = 1; i <= List.size() + 1; i++) {
76     for (int j = 0; j < G[i].size(); j++) {
77         dp[i] = max(dp[i], dp[G[i][j].first] + G[i][j].second);
78     }
79 }
80
81 // 出力
82 cout << dp[List.size() + 1] << endl;
83 return 0;
84 }

```

※Python のコードはサポートページをご覧ください

この問題を解く重要なポイントは、「**列の前半を管理したデック<sup>※10</sup>**」と**「列の後半を管理したデック」**の2つを用意することです。

これらのデックを用意した場合、4種類すべてのクエリを計算量  $O(1)$  で処理することができます。列が  $[20, 28, 15, 37]$  のときのデックの変化を以下に示します。

クエリ1	クエリ2
 最後尾に人が並ぶ → デック 2 の最後尾に追加	 中央に人が入る → デック 1 の最後尾に追加
クエリ3	クエリ4
 先頭から人が抜ける → デック 1 の先頭を削除	 先頭を答える → デック 1 の先頭を答える

厳密には、デック 1 とデック 2 にちょうど半分ずつ（要素数が奇数の場合はデック 1 の方が 1 個だけ多い）入る必要があるため<sup>※11</sup>、クエリが終わった後に下図のような微調整を行わなければならない場合があります。

しかし、この微調整も計算量  $O(1)$  ですので、各クエリ当たりの計算量が  $O(1)$  であることには変わりありません。



## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <deque>
3 #include <string>
4 using namespace std;
5
6 // 入力で与えられる変数
7 int Q;
8 char QueryType[300009]; string X[300009];
9
10 // デック
11 deque<string> Z1, Z2;
12
13 int main() {
14     // 入力
15     cin >> Q;
16     for (int i = 1; i <= Q; i++) {
17         cin >> QueryType[i];
18         if (QueryType[i] == 'A' || QueryType[i] == 'B') cin >> X[i];
19     }
20
21     // クエリの処理
22     for (int i = 1; i <= Q; i++) {
23         // [A] 最後尾に入る
24         if (QueryType[i] == 'A') {
25             Z2.push_back(X[i]);
26         }
27         // [B] 中央に入る
28         if (QueryType[i] == 'B') {
29             Z1.push_back(X[i]);
30         }
31     }
32 }
```

※11 ちょうど半分ずつにしなければならない理由は、クエリ 2 を正しく処理するためです。

```
31     // [C] 先頭が抜ける
32     if (QueryType[i] == 'C') {
33         Z1.pop_front();
34     }
35     // [D] 先頭を答える
36     if (QueryType[i] == 'D') {
37         cout << Z1.front() << endl;
38     }
39
40     // 微調整（前半のデック z1 が大きすぎる場合）
41     while ((int)Z1.size() - (int)Z2.size() >= 2) {
42         string r = Z1.back();
43         Z1.pop_back();
44         Z2.push_front(r);
45     }
46     // 微調整（後半のデック z2 が大きすぎる場合）
47     while ((int)Z1.size() - (int)Z2.size() <= -1) {
48         string r = Z2.front();
49         Z2.pop_front();
50         Z1.push_back(r);
51     }
52 }
53 return 0;
54 }
```

※Python のコードはサポートページをご覧ください

この問題を解く最も単純な方法は、操作方法を全探索することです。しかし残念ながら  $N = 10$  の時点で 6 億通り以上の操作方法があります。制約の最大値である  $N = 200$  では絶望的です。

そこで、以下のような動的計画法を考えます。

### 管理する配列

$dp[1][r]$  : 範囲  $[A_l, A_{l+1}, \dots, A_r]$  だけを削除するのに必要な最小コスト



ここだけを削除するのに必要な最小コスト =  $dp[1][r]$

※他の要素は一切消さない

### 初期状態

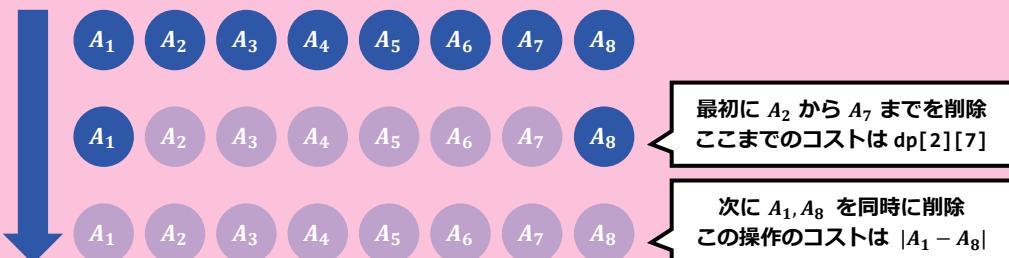
最初は“隣接する 2 要素”のうちどれかを削除するところから始まるので、 $dp[1][2]=dp[2][3]=\dots=dp[2*N-1][2*N]=0$  である。

### 状態遷移

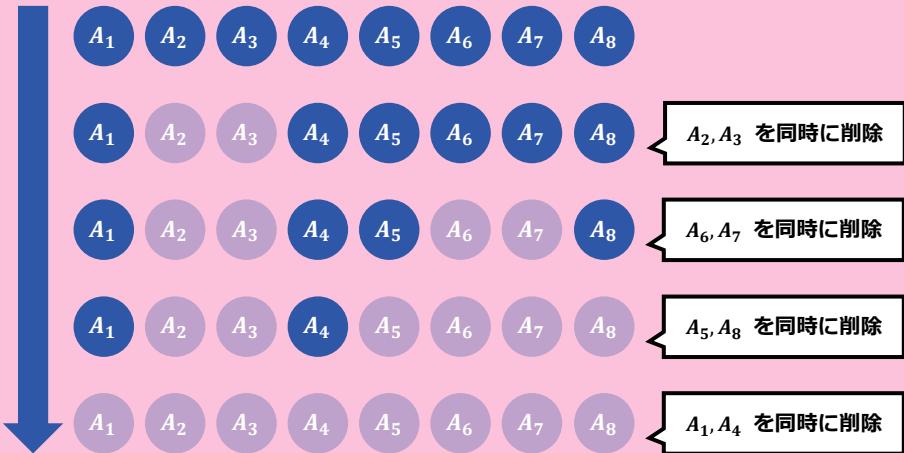
整数列  $[A_l, A_{l+1}, \dots, A_r]$  を全部消すための方法の一つを以下に示す：

- まず、 $A_{l+1}$  から  $A_{r-1}$  までの範囲を削除する
- 次に、 $A_l$  と  $A_r$  を同時に削除する

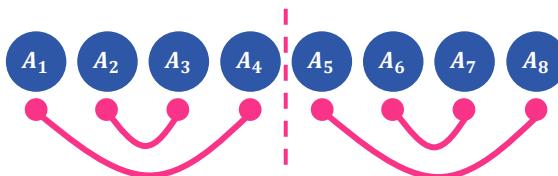
このときの合計コストは  $dp[1+1][r-1] + \text{abs}(A[1]-A[r])$  である。以下に例を示す。



また、それ以外の方法で削除を行う場合、必ず 2 つの区間に分割して考えることができる。たとえば以下の例の場合、「 $A_1$  から  $A_4$  まで」と「 $A_5$  から  $A_8$  まで」に分けられる。



同時に削除した要素を点で結ぶと・・・



$A_4$  と  $A_5$  の間で分割して考えてても問題ない！

※ 分けられた 2 つの領域にまたがる消し方は一度も行っていない

したがって、 $dp[1][r]$  の値は、以下の表の右側の最大値である。

操作方法	コストの最小値
最後に $A_l, A_r$ を同時に削除	$dp[l+1][r-1] + \text{abs}(A[l] - A[r])$
区間 $[A_l, \dots, A_m]$ と区間 $[A_{m+1}, \dots, A_r]$ に分割 ※ $m$ は $l \leq m < r$ を満たせば何でも良い	$dp[1][m] + dp[m+1][r]$

求める答え

$dp[1][2^N]$

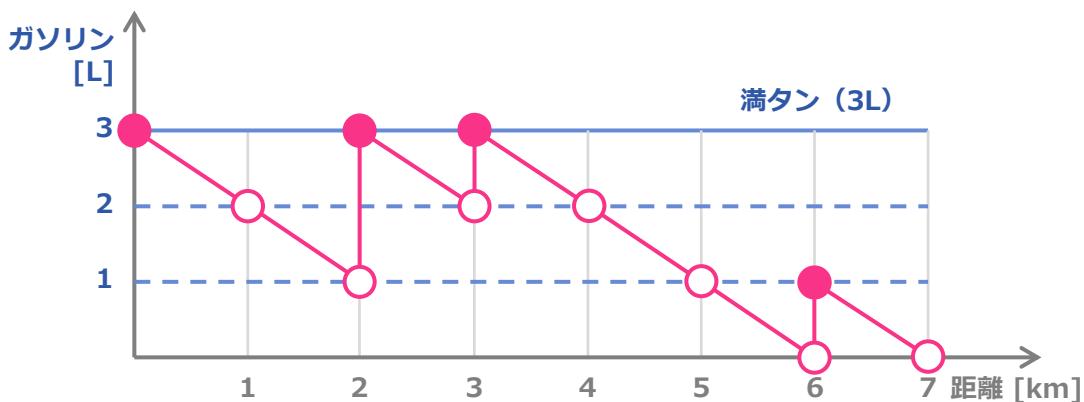
ここまで内容を実装すると、次ページの解答例のようになります。計算量は  $O(N^3)$  です。なお、 $dp[1][r]$  の値は  $r-1$  の小さい順に計算しなければならないことに注意してください。

## 解答例 (C++)

```
1 #include <iostream>
2 #include <cmath>
3 #include <algorithm>
4 using namespace std;
5
6 int N, A[409];
7 int dp[409][409];
8
9 int main() {
10     // 入力
11     cin >> N;
12     for (int i = 1; i <= 2 * N; i++) cin >> A[i];
13
14     // 配列 dp の初期化
15     for (int i = 1; i <= 2 * N; i++) {
16         for (int j = 1; j <= 2 * N; j++) dp[i][j] = 1000000000;
17     }
18
19     // 動的計画法 (初期状態)
20     for (int i = 1; i <= 2 * N - 1; i++) {
21         dp[i][i + 1] = abs(A[i] - A[i + 1]);
22     }
23
24     // 動的計画法 (遷移)
25     for (int LEN = 2; LEN <= 2 * N - 1; LEN++) {
26         for (int l = 1; l <= 2 * N - LEN; l++) {
27             int r = l + LEN;
28
29             // l 番目と r 番目を消す場合
30             dp[l][r] = min(dp[l][r], dp[l + 1][r - 1] + abs(A[l] - A[r]));
31
32             // 2 つの区間を合成させる場合
33             for (int m = l; m < r; m++) {
34                 dp[l][r] = min(dp[l][r], dp[l][m] + dp[m + 1][r]);
35             }
36         }
37     }
38
39     // 出力
40     cout << dp[1][2 * N] << endl;
41     return 0;
42 }
```

※Python のコードはサポートページをご覧ください

まず、 $i$  リットル目のガソリンを入れた場所  $E_i$  について考えます。たとえば下図の場合は  $E = [2, 2, 3, 6]$  ですが、ガソリンの残量が負になったり満タンを突破したりしないためには、 $E_i$  がどのような条件を満たす必要があるのでしょうか。



## ◆ $E_i$ が満たすべき条件（一般的なケース）

まず、 $E_i > i + K - 1$  を満たす場合、ガソリンの残量が負になってしまいます。なぜなら、 $i$  リットル目のガソリンを入れる直前の容量  $K - E_i + (i - 1)$  が 0 を下回ってしまうからです。

また、 $E_i < i$  を満たす場合、ガソリンの容量が満タンを突破してしまいます。なぜなら、 $i$  リットル目のガソリンを入れた直後の容量  $K - E_i + i$  が  $K$  を上回ってしまうからです。

したがって、 $E_i$  の値は次の条件を満たす必要があります。逆にこれらの条件をすべての  $i$  で満たせば、ガソリンの残量は常に 0 以上  $K$  以下となります。

$$i \leq E_i \leq i + K - 1$$

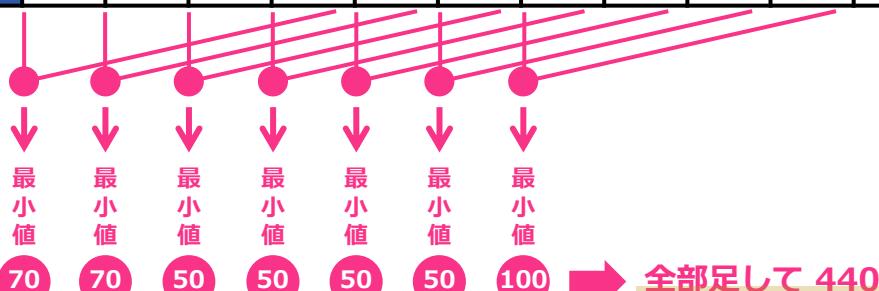
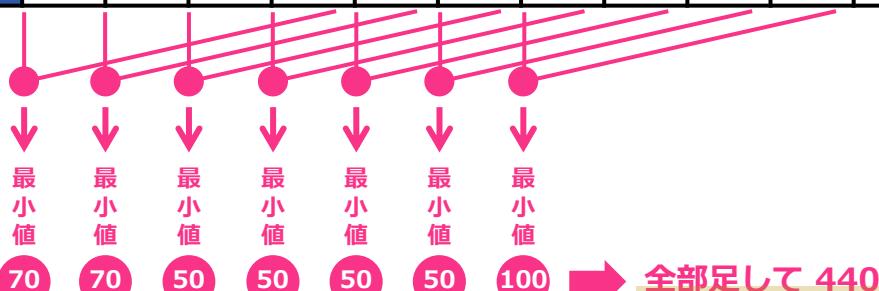
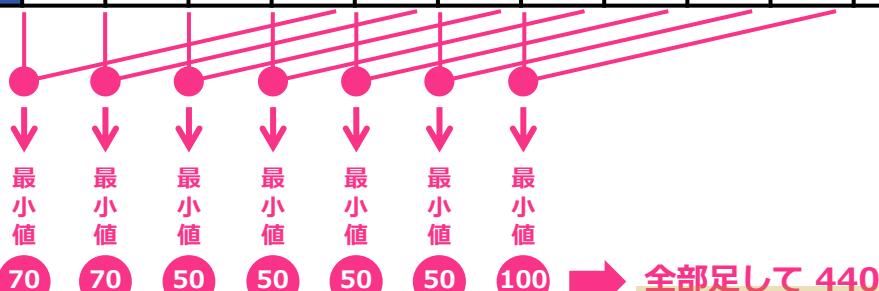
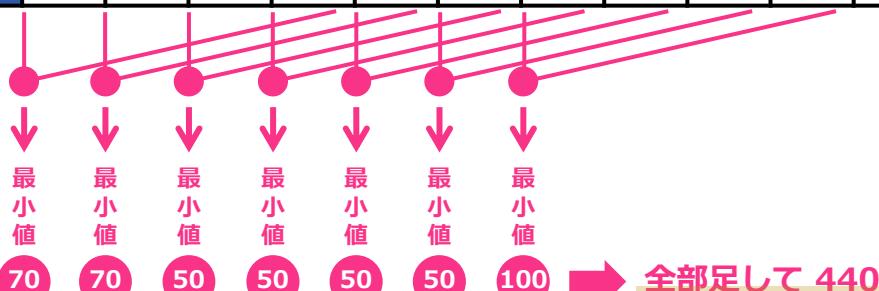
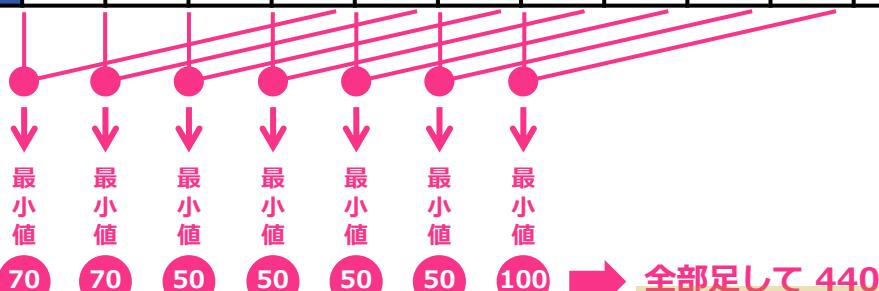
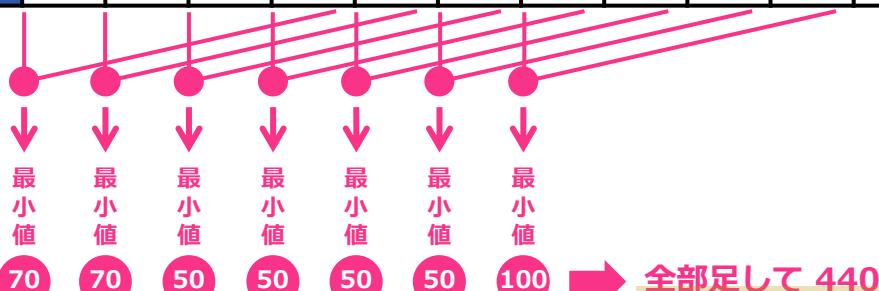
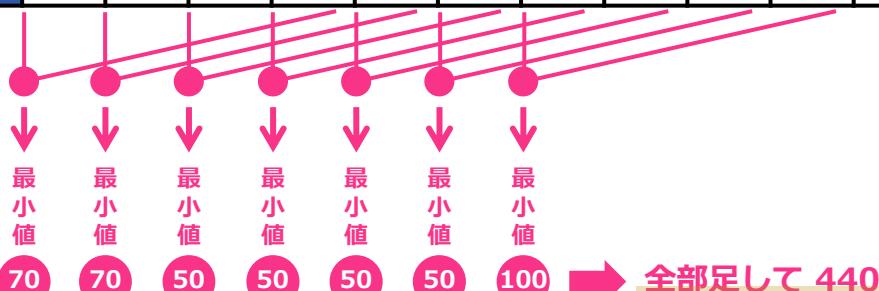
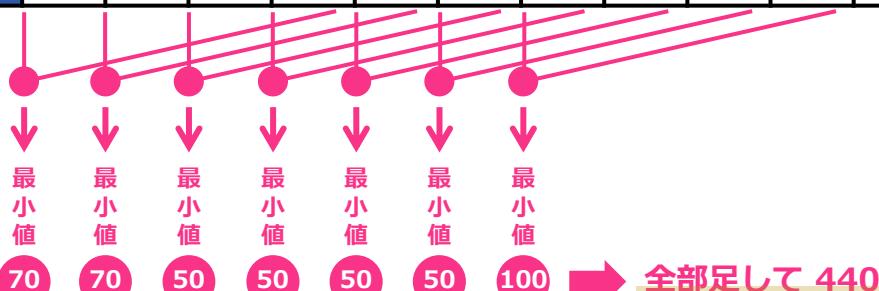
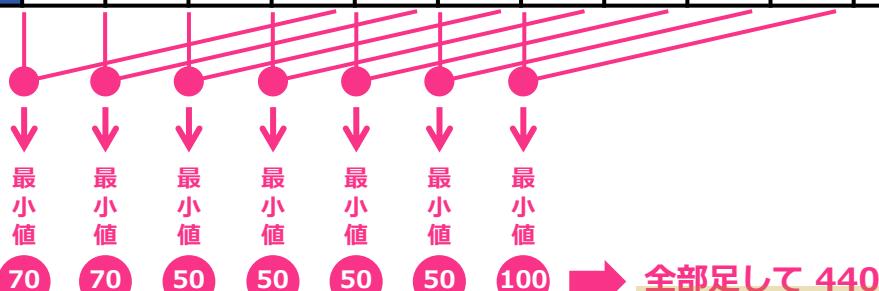
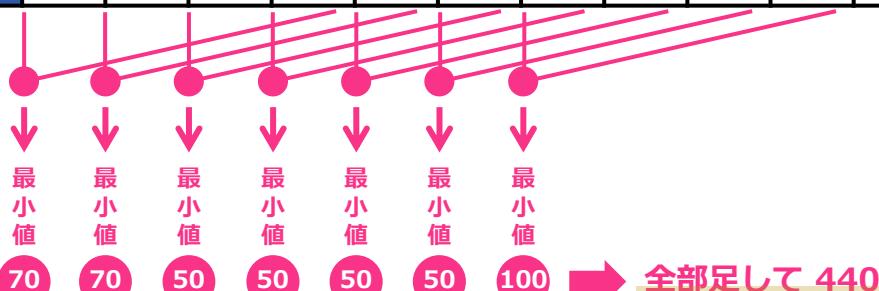
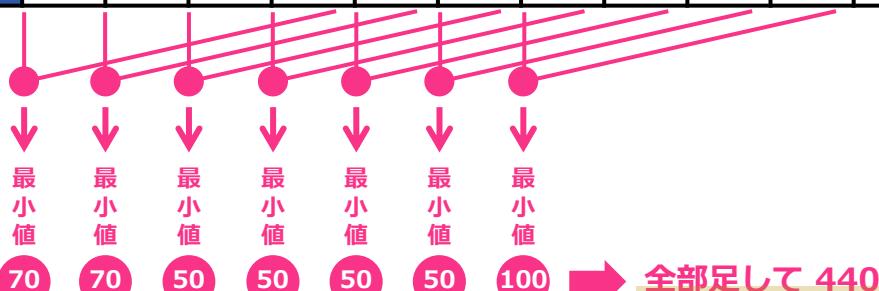
## ◆ 答えを求める

以上のことから、 $x$  キロ地点で給油できるガソリンの価格の最小値を  $\min\_value[x]$  とするとき、 $i$  リットル目のガソリンを得るための最小価格は次式で表されます。

$$\min(\min\_value[i], \min\_value[i+1], \dots, \min\_value[i+K-1])$$

そして、ガソリン残量 0 でゴールするのが最適なので、給油すべきガソリンの総量は  $L - K$  リットルです。したがって、 $i = 1, 2, \dots, L - K$  について上式の値を合計した値が答えになります。

たとえば入力例 1 ( $N = 3, L = 11, K = 4, (A_i, C_i) = (4, 70), (6, 50), (9, 100)$  の場合) では、下図の通り、答えが 440 となります。

	1	2	3	4	5	6	7	8	9	10	11
$\min\_value[i]$	$\infty$	$\infty$	$\infty$	70	$\infty$	50	$\infty$	$\infty$	100	$\infty$	$\infty$
											

## ◆ 実装と計算量

最後に、このアルゴリズムを実装すると解答例のようになります。単純に実装すると計算量が  $O(N^2)$  となり実行時間制限に間に合いませんが、セグメント木を使って  $\min(\min\_value[i], \min\_value[i+1], \dots, \min\_value[i+K-1])$  の値を求めるとき、計算量が  $O(N \log N)$  まで削減されます。

## ◆ 解答例 (C++)

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 class SegmentTree {
6 public:
7     long long dat[2100000], siz = 1;
```

```

8 // 要素 dat の初期化を行う（最初は全部ゼロ）
9 void init(int N) {
10    siz = 1;
11    while (siz < N) siz *= 2;
12    for (int i = 1; i < siz * 2; i++) dat[i] = 0;
13 }
14 // クエリ 1 に対する処理
15 void update(int pos, long long x) {
16    pos = pos + siz - 1;
17    dat[pos] = x;
18    while (pos >= 2) {
19        pos /= 2;
20        dat[pos] = min(dat[pos * 2], dat[pos * 2 + 1]);
21    }
22 }
23 // クエリ 2 に対する処理
24 // u は現在のセル番号、[a, b) はセルに応する半開区間、[l, r) は求めたい半開区間
25 long long query(int l, int r, int a, int b, int u) {
26    if (r <= a || b <= l) return (1LL << 60); // 一切含まれない場合
27    if (l <= a && b <= r) return dat[u]; // 完全に含まれる場合
28    int m = (a + b) / 2;
29    long long AnswerL = query(l, r, a, m, u * 2);
30    long long AnswerR = query(l, r, m, b, u * 2 + 1);
31    return min(AnswerL, AnswerR);
32 }
33 };
34
35 long long N, L, K;
36 long long A[700009], C[700009];
37 long long Min_Value[700009];
38 SegmentTree Z;
39
40 int main() {
41    // 入力
42    cin >> N >> L >> K;
43    for (int i = 1; i <= N; i++) cin >> A[i] >> C[i];
44    // 各地点での「値段の最小値」を求める
45    for (int i = 1; i <= L - 1; i++) Min_Value[i] = (1LL << 60);
46    for (int i = 1; i <= N; i++) Min_Value[A[i]] = min(Min_Value[A[i]], C[i]);
47
48    // セグメント木に載せる
49    Z.init(L);
50    for (int i = 1; i <= L - 1; i++) Z.update(i, Min_Value[i]);
51
52    // 答えを求める
53    long long Answer = 0;
54    for (int i = 1; i <= L - K; i++) {
55        long long val = Z.query(i, i + K, 1, Z.siz + 1, 1);
56        if (val == (1LL << 60)) {
57            cout << "-1" << endl;
58            return 0;
59        }
60        Answer += val;
61    }
62    cout << Answer << endl;
63    return 0;
64 }

```

※Python のコードはサポートページをご覧ください

この問題は、本書の第7章で扱ったジャンルであるヒューリスティックの問題です。地区に分割する際に、人口・役所職員数ができるだけ等しくなるほど高得点が得られるという形式ですので、様々な解法が考えられます。

## ◆はじめの一歩

まずは最初の一手として、特別区の連結性は考えずに<sup>※12</sup>、とりあえず正の得点を得るプログラムを書いてみましょう。たとえば、各地区をランダムな特別区に割り当てる以下のコードを提出すると、36,348点が得られます。

```

1 #include <iostream>
2 using namespace std;
3
4 int N, K, L, A[401], B[401], C[51][51];
5
6 int main() {
7     // 入力
8     cin >> N >> K >> L;
9     for (int i = 1; i <= K; i++) {
10         cin >> A[i] >> B[i];
11     }
12     for (int i = 1; i <= N; i++) {
13         for (int j = 1; j <= N; j++) {
14             cin >> C[i][j];
15         }
16     }
17
18     // 出力
19     for (int i = 1; i <= K; i++) {
20         cout << rand() % L + 1 << endl; // 1 以上 L 以下のランダムな整数
21     }
22
23     return 0;
24 }
```

この解法の欠点は、特別区に割り当てる地区の個数が平均的なケースで10~30個とばらつき、結果として人口・役所職員数の格差が大きくなってしまうことです。

<sup>※12</sup> 特別区が連結でなくても、連結な場合の1/1000の得点が取れるので、最大100,000点が狙えます。

これを改善するため、各特別区に 20 個ずつ地区を割り当ててみましょう。一例として、先ほどのプログラムの 18~21 行目を以下のように変えると、83,544 点が得られます。

```
18 // 出力
19 for (int i = 1; i <= K; i++) {
20     cout << i % L + 1 << endl;
21 }
```

## ◆ 少し工夫して 9 万点を狙おう

先ほどは「各特別区に 20 個ずつ地区を割り当てる」ような方法を 1 つ決め打ちして出力していました。これをランダムにたくさんの回数行って、その中で最も良い解を出力する、というアルゴリズムにすると、さらに良い解が見つけられそうです。例えば、10,000 回試行した以下ののようなプログラムを提出すると、90,977 点が得られます（入力・出力等の部分は省略）。

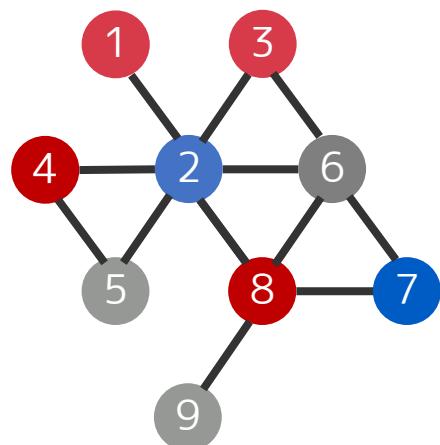
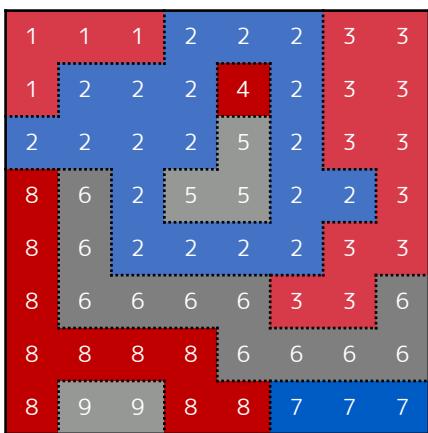
```
1 // 解のランダム生成を 10,000 回
2 double best_score = 0.0;
3 int answer[401];
4 for (int repeat = 1; repeat <= 10000; repeat++) {
5     // 特別区を 20 個ずつの地区にランダムに割り当てる
6     int region[401];
7     for (int i = 1; i <= K; i++) {
8         region[i] = i % L + 1;
9     }
10    random_shuffle(region + 1, region + K + 1);
11    // p[i], q[i] の計算
12    int p[21], q[21];
13    for (int i = 1; i <= L; i++) {
14        p[i] = 0;
15        q[i] = 0;
16    }
17    for (int i = 1; i <= K; i++) {
18        p[region[i]] += A[i];
19        q[region[i]] += B[i];
20    }
21    // スコア (= min(pmin/pmax, qmin/qmax)) の計算
22    int pmin = *min_element(p + 1, p + L + 1);
23    int pmax = *max_element(p + 1, p + L + 1);
24    int qmin = *min_element(q + 1, q + L + 1);
25    int qmax = *max_element(q + 1, q + L + 1);
26    double score = min(double(pmin) / pmax, double(qmin) / qmax);
27    // 現在得られている最良の解より良かったら、解を更新する
28    if (best_score < score) {
29        best_score = score;
30        for (int i = 1; i <= K; i++) {
31            answer[i] = region[i];
32        }
33    }
34 }
```

## ◆ ここから得点を上げるために

先ほどのように、各特別区が連結になっているかを考えずにプログラムを書くと得点が 0.001 倍になるため、最大でも 100,000 点しか得られません。より高い得点を取るためには、**すべての特別区を連結にする必要**が出てきます。

## ◆ グラフの問題として考える

この問題では、本書の第 9 章で扱った「グラフ」を使うと考えやすいです。各地区を頂点、隣接する地区を辺で表したグラフを考えましょう。



以下のようなプログラムで、グラフを作ることができます。

```
1 // グラフの作成
2 vector<int> G[401];
3 for (int i = 1; i <= N; i++) {
4     for (int j = 1; j <= N; j++) {
5         if (i != N && C[i][j] != 0 && C[i + 1][j] != 0 && C[i][j] != C[i + 1][j]) {
6             G[C[i][j]].push_back(C[i + 1][j]);
7             G[C[i + 1][j]].push_back(C[i][j]);
8         }
9         if (j != N && C[i][j] != 0 && C[i][j + 1] != 0 && C[i][j] != C[i][j + 1]) {
10            G[C[i][j]].push_back(C[i][j + 1]);
11            G[C[i][j + 1]].push_back(C[i][j]);
12        }
13    }
14 }
15 // G[i] の重複を取り除く (erase/unique については p.103 のコードを参照)
16 for (int i = 1; i <= K; i++) {
17     sort(G[i].begin(), G[i].end());
18     G[i].erase(unique(G[i].begin(), G[i].end()), G[i].end());
19 }
```

すると、元の問題では地理的な連結性を考える必要があったのを、グラフの問題として考えることで、以下のような単純化された問題として取り組みます。

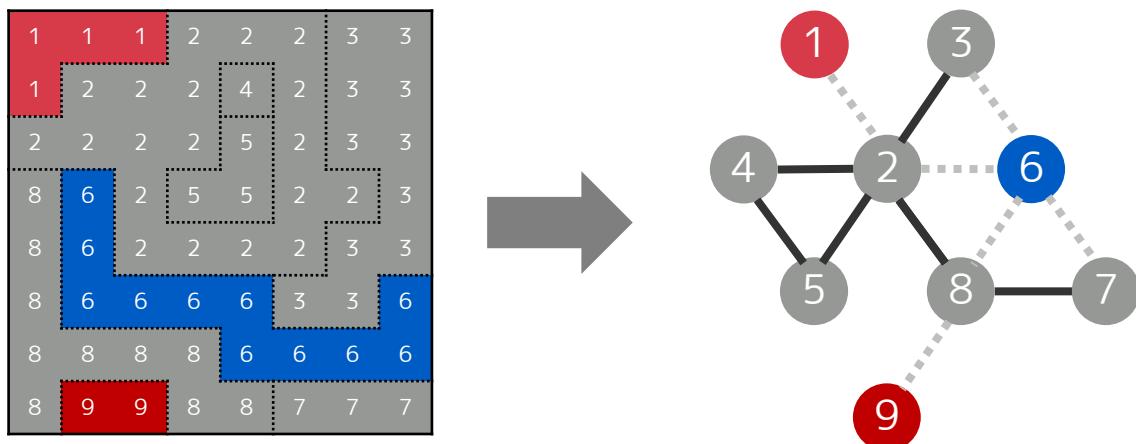
$K = 400$  頂点のグラフが与えられる。これを  $L = 20$  個の**連結な部分グラフ**に分割する方法のなかで、それぞれの部分グラフの「 $A_i$  の合計」および「 $B_i$  の合計」をできるだけ平等にするものを見つけよ。

## ◆ 特別区を連結にしてみよう！

今は特別区の格差のことなど考えずに、とりあえず連結な特別区に分けることを考えてみましょう。ここで一番簡単な方法は以下のよう�습니다。

- ある特定の特別区は  $K - L + 1 = 381$  個の地区を含み、それ以外の  $L - 1 = 19$  個の特別区は 1 つだけの地区からなるようにする。

下図は  $K = 9$  個の地区を  $L = 4$  個の特別区に分けるときの例です。サイズ 6 の連結な部分グラフさえ見つければ、残りは独立した特別区にすればいい、という発想です。



つまり、サイズ 381 の連結な部分グラフさえ見つけられれば OK です。これには色々な方法がありますが、例えば「深さ優先探索（9.2 節）で到達した順で 381 番目までに入る頂点を選ぶ」などの方法<sup>※13</sup>で、確実に見つけることができます。次ページのプログラムでは、ここで選ばれた頂点を特別区 1 に、それ以外の頂点を特別区 2~20 に割り当てています。このプログラムを提出すると、179,697 点が得られます。

※13 それ以外にも「幅優先探索（9.3 節）で特定の頂点からの最短距離を求め、その距離が小さい順に 381 個を選ぶ」や「Union-Find 木を使って『連結性を保ったまま頂点を 1 個削除する』を 19 回繰り返す」などの方法が使えます。

```

1 #include <vector>
2 #include <iostream>
3 #include <algorithm>
4 using namespace std;
5
6 int N, K, L, A[401], B[401], C[51][51]; vector<int> G[401];
7
8 // 深さ優先探索を行う関数 dfs (9.2 節を参照)
9 int counter = 0;
10 bool visited[401]; int ranking[401];
11 void dfs(int pos) {
12     visited[pos] = true;
13     counter += 1;
14     ranking[pos] = counter;
15     for (int i = 0; i < G[pos].size(); i++) {
16         int nex = G[pos][i];
17         if (visited[nex] == false) {
18             dfs(nex);
19         }
20     }
21 }
22
23 int main() {
24     // 入力 (省略・C20 の解説 1 ページ目を参照)
25     :
26     // グラフの作成 (省略・C20 の解説 3 ページ目を参照)
27     :
28     // 深さ優先探索 (DFS) を行う (頂点 1 からスタートする)
29     for (int i = 1; i <= K; i++) {
30         visited[i] = false;
31     }
32     dfs(1);
33
34     // 出力
35     for (int i = 1; i <= K; i++) {
36         if (ranking[i] <= K - L + 1) {
37             // ranking[i] が 381 以下なら、特別区 1 に割り当て
38             cout << 1 << endl;
39         }
40         else {
41             // ranking[i] が 382 以上なら、特別区 2~20 に割り当て
42             cout << ranking[i] - (K - L) << endl;
43         }
44     }
45
46     return 0;
47 }
48 }
```

残りの得点は、いかにして都市ごとの格差をなくすかにかかります。現状では各特別区は連結になっているものの、地区の数にして 381 倍、平均的なケースで約 560 倍の格差が生じており、これを減らしていく必要があります。ここからは、本書の 7.1 節で扱った貪欲法、7.2 節で扱った山登り法、7.3 節で扱った焼きなまし法などが力を発揮します。

## ◆ 貪欲法を使ってみよう

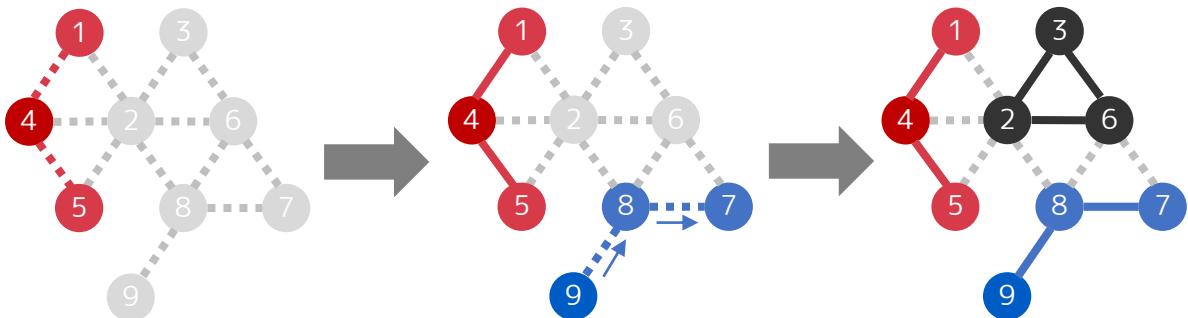
本書の 6.4 節・7.1 節で紹介した貪欲法は、一手先のことしか考えずに答えを決めていく方法です。この問題では、まず次のような貪欲法のアイデアが思いつくでしょう。

特別区 1, 2, …, 19 の順に、サイズ 20 の連結な部分グラフを見つけるために、以下の処理を繰り返す。

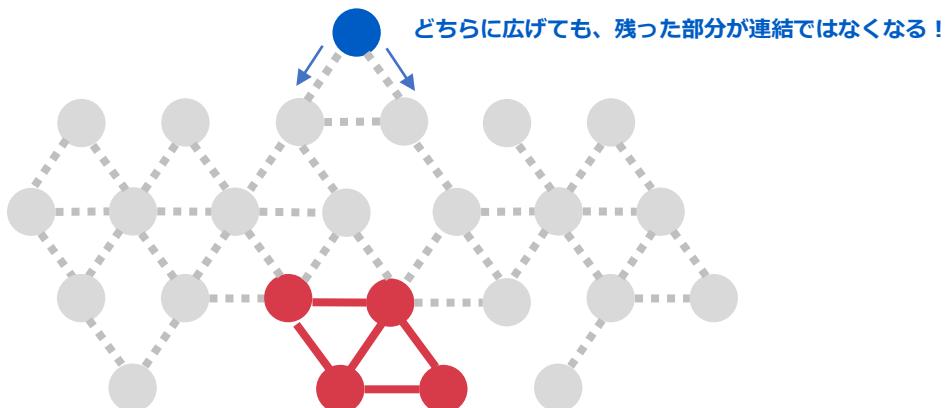
- まだ残っている頂点を起点とし、頂点数が 20 になるまで広げていく。ただし、残った部分も連結にならなければならない。

最後に残った頂点は特別区 20 に割り当てる。

以下の図はこの貪欲法のイメージです（9 個の地区を 3 分割する例）。



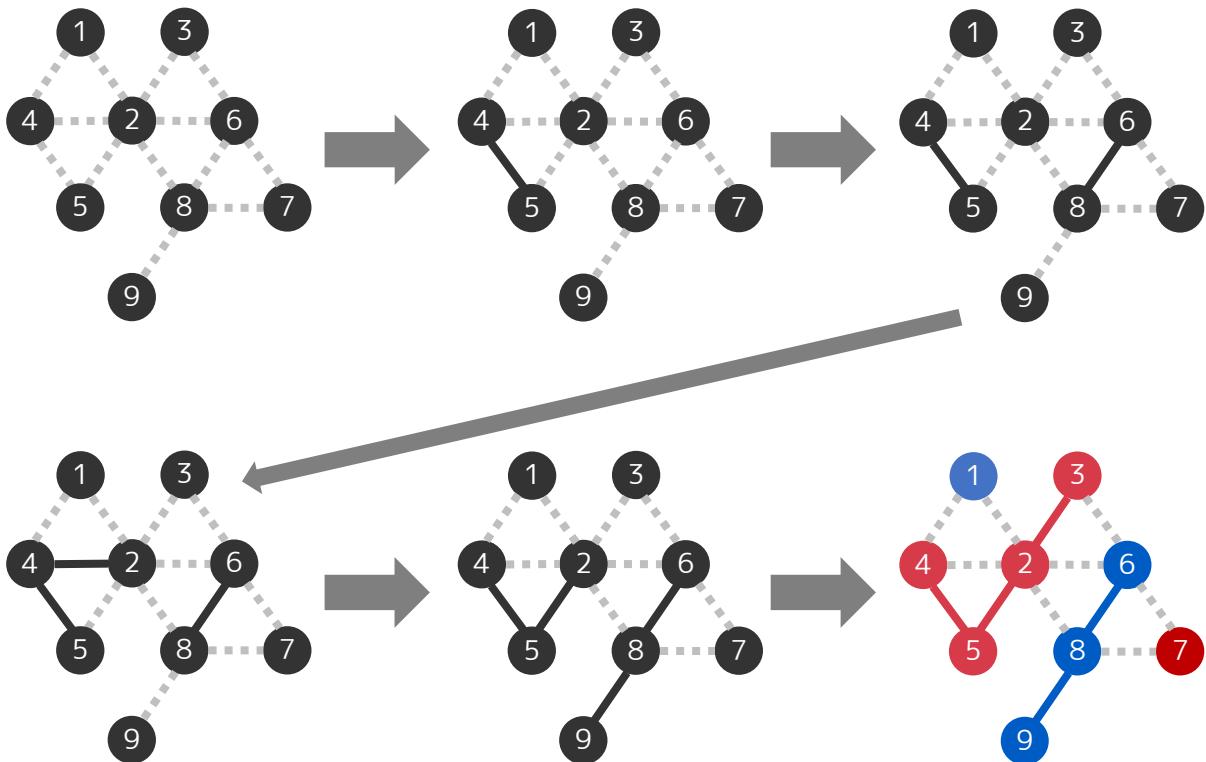
しかし、この貪欲法はあまり上手くいきません。なぜなら、残った部分の連結性を保ったまま頂点数が 20 になるまで広げられない場合が多く、結果として半分程度の頂点が使われずに残ってしまうからです。そのため、これを実装したプログラムを提出しても、得点は 464,392 点にとどまります。



## ◆ 「バランスの良い」貪欲法

先ほどのように「特別区を 1 個ずつ決めていく」方法では上手くいかなかつたので、少し発想を変えて「地区の合併を繰り返して  $L$  個になるまで減らしていく」という方法で答えを決めることを考えましょう。

以下の図は、9 個の地区を 4 つに「合併」する例です。そのためには 5 回の合併を行う必要があります。 $(K = 400$  個の地区を  $L = 20$  個の特別区にするためには、380 回の合併を行うことになります)



しかし、あとさき考えずに合併を行っていくと、上図の例のように特別区のサイズに大きな偏りが生じてしまいます。どのようにすれば偏りを減らせるか少し考えてみましょう。たとえば、以下のような貪欲法が考えられます。

- 地区の個数が 20 個になるまで、以下の要領で合併を繰り返す。
- 2 つの地区を併合すると新しい 1 つの大きな地区ができるが、この新しい地区のサイズができるだけ小さくなるような方法で併合を行う。

つまり、たとえば上図の 3 番目の合併操作では、頂点 2 の地区と頂点 4, 5 の地区を併合するのではなく、頂点 2 の地区と頂点 3 の地区を併合すべきだった、と考えます。

実装には Union-Find 木（9.6 節）を使うと便利です。特に、ある頂点  $x$  が属する連結成分のサイズは、 $\text{size}[\text{root}(x)]$  で求められます。例えば C++ では、以下のプログラムのように実装できます。

```
1 #include <vector>
2 #include <iostream>
3 #include <algorithm>
4 using namespace std;
5
6 // Union-Find 木 (省略・実装は本書の 9.6 節を参照)
7 :
8
9 int N, K, L, A[401], B[401], C[51][51]; vector<int> G[401];
10 int answer[401];
11
12 int main() {
13     // 入力 (省略・C20 の解説 1 ページ目を参照)
14     :
15     // グラフの作成 (省略・C20 の解説 3 ページ目を参照)
16     :
17     // 貪欲法 (併合を  $K - L = 380$  回繰り返す)
18     UnionFind uf;
19     uf.init(K);
20     for (int i = 1; i <= K - L; i++) {
21         int min_size = 1000000000, vertex1 = -1, vertex2 = -1;
22         for (int j = 1; j <= K; j++) {
23             for (int k = 0; k < G[j].size(); k++) {
24                 int v = G[j][k];
25                 if (uf.same(j, v) == false) {
26                     // 頂点 j の地区と頂点 v の地区を併合すると... ?
27                     int size1 = uf.size[uf.root(j)];
28                     int size2 = uf.size[uf.root(v)];
29                     if (min_size > size1 + size2) {
30                         min_size = size1 + size2;
31                         vertex1 = j;
32                         vertex2 = v;
33                     }
34                 }
35             }
36         }
37         uf.unite(vertex1, vertex2);
38     }
39
40     // Union-Find 木の状態から答えを出す (erase/unique については p.103 のコードを参照)
41     vector<int> roots;
42     for (int i = 1; i <= K; i++) {
43         roots.push_back(uf.root(i));
44     }
45     sort(roots.begin(), roots.end());
46     roots.erase(unique(roots.begin(), roots.end()), roots.end());
47     for (int i = 1; i <= K; i++) {
48         for (int j = 0; j < roots.size(); j++) {
49             if (roots[j] == uf.root(i)) {
50                 answer[i] = j + 1;
51             }
52         }
53     }
54 }
```

```

121 // 出力
122 for (int i = 1; i <= K; i++) {
123     cout << answer[i] << endl;
124 }
125
126 return 0;
127 }
```

このプログラムを提出すると、43,128,693 点が得られます。これは先ほどよりも大きく上がった値です。この貪欲法では、特別区のサイズが最小のものと最大のもので 2 倍程度しか差が生まれず、バランス良く分割できています。

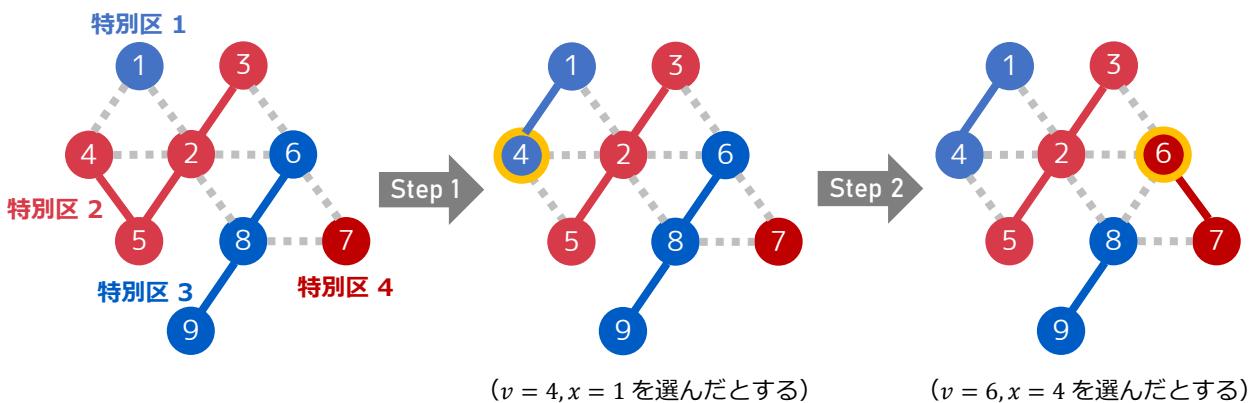
## ◆ 山登り法を使ってみよう

本書の 7.2 節で紹介した山登り法（局所探索法）は「小さな変更をランダムに行い、スコアが良くなればその変更を採用する」ことを繰り返して、答えをだんだん良くしていく手法です。

この問題では、先ほど説明した貪欲法で作るような初期解から、どんな「小さな変更」を行うのが良い解への道筋としてよいのでしょうか？たとえば、以下のようなものを考えてみましょう。

適当な地区  $v$  ( $1 \leq v \leq K$ ) と特別区の番号  $x$  ( $1 \leq x \leq L$ ) を選び、地区  $v$  の割り当てを特別区  $x$  に変更する。

以下の図が、山登り法のイメージです。小さな変更を繰り返すことで、元々偏りがあった特別区の割り当てが、だんだんバランス良くなっています。



## ◆ 山登り法の実装（1）：スコアの計算

山登り法では「小さな変更を行ったらスコアが良くなつたか」によって変更を採用するかどうか判断します。なので、どのくらい格差が小さいかのスコア  $\min(p_{\min}/p_{\max}, q_{\min}/q_{\max})$  を計算する関数を作つておきましょう<sup>※14</sup>。

特に、ここでは「 $L$  個の特別区がすべて存在し、かつ連結である」必要があるので、それを満たさないものは“スコア 0”とみなして、実装のメインの部分でもそう分かるようにしておくと良いでしょう。

```
1 // 深さ優先探索 (9.2 節を参照)
2 bool visited[401];
3 void dfs(int pos) {
4     visited[pos] = true;
5     for (int i = 0; i < G[pos].size(); i++) {
6         int nex = G[pos][i];
7         if (answer[nex] == answer[pos] && visited[nex] == false) dfs(nex);
8     }
9 }
10
11 // どのくらい格差が小さいかのスコアを返す関数
12 double get_score() {
13     // 答えが正しいかを深さ優先探索 (DFS) を使って確認
14     for (int i = 1; i <= K; i++) visited[i] = false;
15     for (int i = 1; i <= L; i++) {
16         // 特別区 i に属する頂点 pos を探す
17         int pos = -1;
18         for (int j = 1; j <= K; j++) {
19             if (answer[j] == i) pos = j;
20         }
21         if (pos == -1) return 0.0; // 存在しない特別区がある！
22         dfs(pos);
23     }
24     for (int i = 1; i <= K; i++) {
25         if (visited[i] == false) return 0.0; // 連結ではない特別区がある！
26     }
27     // スコアの計算
28     int p[21], q[21];
29     for (int i = 1; i <= L; i++) {
30         p[i] = 0;
31         q[i] = 0;
32     }
33     for (int i = 1; i <= K; i++) {
34         p[answer[i]] += A[i];
35         q[answer[i]] += B[i];
36     }
37     int pmin = *min_element(p + 1, p + L + 1);
38     int pmax = *max_element(p + 1, p + L + 1);
39     int qmin = *min_element(q + 1, q + L + 1);
40     int qmax = *max_element(q + 1, q + L + 1);
41     return min(double(pmin) / pmax, double(qmin) / qmax);
42 }
```

※14 本の p.277 で述べたように、ヒューリスティック系コンテストでは実装が数百行と長くなることが多いため、部分部分に分けてプログラムを書くのが実装しやすいとされています。

## ◆ 山登り法の実装 (2)：メインの部分

次に、山登り法の本体の部分を実装します。ここでは、初期解に「貪欲法で生成した解」（本問題 C20 の解説 7~9 ページ目を参照）を使用します※15。

```
1 #include <vector>
2 #include <iostream>
3 #include <algorithm>
4 using namespace std;
5
6 // Union-Find 木 (省略・実装は本書の 9.6 節を参照)
7 :
8
9 int N, K, L, A[401], B[401], C[51][51]; vector<int> G[401];
10 int answer[401];
11
12 // スコアの計算 (深さ優先探索も含む) / 省略・実装は C20 の解説 10 ページ目を参照
13 :
14
15 int main() {
16     // 入力 (省略・C20 の解説 1 ページ目を参照)
17 :
18
19     // グラフの作成 (省略・C20 の解説 3 ページ目を参照)
20 :
21
22     // 貪欲法で初期解を生成する (省略・C20 の解説 8~9 ページ目を参照)
23 :
24
25     // 山登り法 (0.95 秒ループを回す)
26     double TIME_LIMIT = 0.95;
27     int ti = clock();
28     double current_score = get_score();
29     while (double(clock() - ti) / CLOCKS_PER_SEC < TIME_LIMIT) {
30         int v = rand() % K + 1; // 1 以上 K 以下のランダムな整数
31         int x = rand() % L + 1; // 1 以上 L 以下のランダムな整数
32         int old_x = answer[v];
33         // とりあえず変更し、スコアを評価する
34         answer[v] = x;
35         double new_score = get_score();
36         // スコアに応じて採用／不採用を決める (解が不正なら当然採用しない)
37         if (new_score!=0.0 && current_score<=new_score) current_score = new_score;
38         else answer[v] = old_x;
39     }
40
41     // 出力
42     for (int i = 1; i <= K; i++) {
43         cout << answer[i] << endl;
44     }
45
46     return 0;
47 }
```

このプログラムを提出すると、86,943,388 点というかなりの高得点が得られます※16。実際に、特別区のサイズはほぼ均等になっており、人口や役所職員数もある程度考慮した分割ができます。

※15 一般的に、初期解には極端な解を設定しない方がよいとされます。たとえば、初期解に本問題 C20 の解説 4~5 ページ目のものを用いると、964,573 点しか得られません。

※16 ループ回数は 10 万回程度です。これにはまだ改良の余地があり、さらに 10 倍程度高速化できます。

## ◆さらなる高みへ

前ページの解答例では、ランダムに頂点を選びその割り当てをランダムに変えていました。しかしそれでは、周りの頂点と異なる特別区に設定され明らかに連結にならない場合があります。この対処として先ほどのコード 171～172 行目の部分を以下のように変えると、88,205,697 点が得られます。

```
1 int v, x;
2 do {
3     v = rand() % K + 1; // 1 以上 K 以下のランダムな整数
4     x = answer[G[v][rand() % G[v].size()]]; // 頂点 v に隣接する特別区をランダムに選ぶ
5 } while (answer[v] == answer[x]);
```

さらに、山登り法の代わりに焼きなまし法（7.3 節）を使うこともできます。先ほどのコード 177～179 行目の部分を以下のように変えてみましょう。すると、89,056,925 点が得られます。もし AtCoder Heuristic Contest の 4 時間コンテストで出題されれば、トップ 30 が狙えるような得点でしょう※6。

```
1 // スコアの変化に応じて、変更を採用する確率を決める
2 double rand_value = double(rand() + 0.5) / (RAND_MAX + 1.0); // 0～1 のランダムな実数
3 double temp = 0.0040 - 0.0039 * (double(clock() - ti) / CLOCKS_PER_SEC / TIME_LIMIT);
4 if (new_score != 0.0 && rand_value < exp((new_score - current_score) / temp)) {
5     current_score = new_score;
6 }
7 else {
8     answer[v] = old_x;
9 }
```

## ◆問題 C20 の解説のまとめ

本書の第 7 章で扱ったような「ヒューリスティック型課題」には実に多種多様な問題が出題され、それぞれの問題の構造に特化したアルゴリズムを作ることが求められます。

本解説では、貪欲法・山登り法・焼きなまし法を使って得点を伸ばしたという側面もありますが、実際に典型的な手法をあてはめることを超えて「どうやって問題に取り組むのか」ということも重要な側面でした。

実は、ここからさらに改善して 9500 万点を得る方法もあります。本解説では触れませんが、焼きなまし法を高速化したり、あるいはバランスの良い解に到達しやすくする工夫をしたりと、まだまだたくさんの工夫の余地があります。実力に自信のある方はぜひ挑戦してみましょう。

※6 2022 年 9 月時点。筆者の主観ではありますが、パフォーマンス 2400 程度になると思われます。

# 最終章

あとがき

## あとがき

---

全 216 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でしょうか。実際、応用問題と力試し問題を合わせたら 79 問あり、一問当たり 1 時間でも 80 時間近くかかります。しかし、これらの問題で少しでも力を付けていただき、読者の役に立てれば本当に嬉しいです。

最後になりますが、解説作成にあたっては米田寛峻氏（square1001）に協力していただきました。力試し問題 20 の解説が彼によって執筆されました。大変感謝しております。

2022 年 9 月 28 日  
米田 優峻

## 競技プログラミングの鉄則

### 解答・解説

---

2022 年 9 月 28 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