力試し問題後半

（問題 11～20）

11 問題 C11：Election (Lv. 70)

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※1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 位となる値のことを指します。$K + 1$ 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 $\log_2 10^{15} \fallingdotseq 50$ 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が、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、十分余裕をもっ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++）

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

long long N, K;
double A[100009];

// 割り算の値が x であるときの議席数は？
long long check(double x) {
    long long sum = 0;
    for (int i = 1; i <= N; i++) sum += (long long)(A[i] / x);
    return sum;
}
```

※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。

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

    // 二分探索
    double Left = 1, Right = 1000000000, Mid;
    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）
    for (int i = 1; i <= 60; i++) {
        Mid = (Left + Right) / 2.0;

        // 割り算の値は Mid より大きいか？
        long long val = check(Mid);
        if (val >= K) {
                Left = Mid;
                Border = max(Border, Mid);
        }
        else {
                Right = Mid;
        }
    }

    // 出力
    for (int i = 1; i <= N; i++) {
        if (i >= 2) cout << " ";
        cout << (long long)(A[i] / Border);
    }
    cout << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

12 問題 C12：Taro the Novel Writer (Lv. 70)

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）。

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 $O(N^2MK)$ です。本問題の制約の上限は $N = 288, M = 50, K = 10$ であり、$288^2 \times 50 \times 10 \fallingdotseq 40000000$ 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++）

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

int N, K;
int M, A[59], B[59];
int dp[19][309];

// 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？
int score(int l, int r) {
    int cnt = 0;
    for (int i = 1; i <= M; i++) {
        if (l <= A[i] && B[i] <= r) cnt++;
    }
    return cnt;
}
```

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

    // 配列 dp の初期化
    for (int i = 0; i <= K; i++) {
        for (int j = 0; j <= N; j++) dp[i][j] = -1000000;
    }

    // 動的計画法（貰う遷移形式）
    dp[0][0] = 0;
    for (int i = 1; i <= K; i++) {
        for (int j = 1; j <= N; j++) {
                // k は「前の章がどのページで終わったか」
                for (int k = 0; k <= j - 1; k++) {
                        dp[i][j] = max(dp[i][j], dp[i - 1][k] + score(k + 1, j));
                }
        }
    }

    // 出力
    cout << dp[K][N] << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

13 問題 C13：Select 2 (Lv. 75)

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。

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

long long N, P;
long long A[100009];

int main() {
    // 入力
    cin >> N >> P;
    for (int i = 1; i <= N; i++) cin >> A[i];
    for (int i = 1; i <= N; i++) A[i] %= 1000000007LL; // オーバーフロー防止

    long long Answer = 0;
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= i - 1; j++) {
                if (A[j] * A[i] % 1000000007LL == P) Answer += 1;
        }
    }
    cout << Answer << endl;
    return 0;
}
```

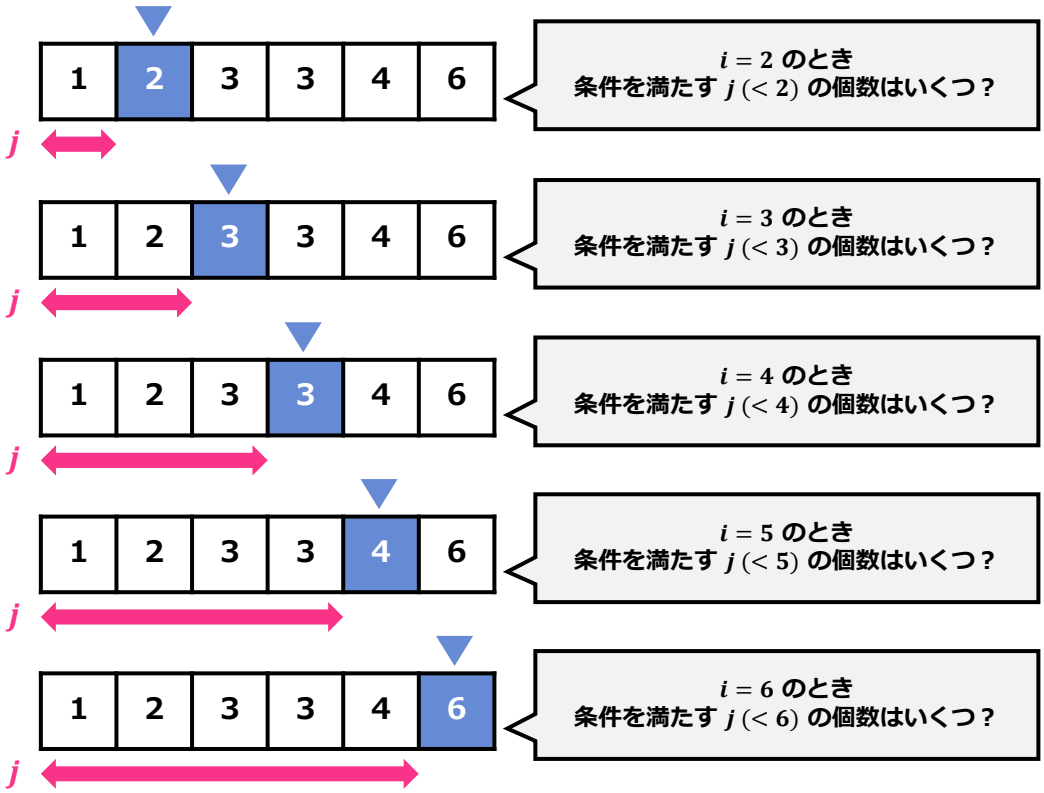
効率的な解法（$P \neq 0$ の場合）

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なります。

mod 1000000007 上での「$P \div A_i$」※3

※3　なぜなら、もう一方のカードを x とするとき、$(A_i \times x) \bmod 1000000007 = P$ 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 $A_j =$ (mod 1000000007 上での $P \div A_i$ 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。

$i = 1,2,\ldots,N$ 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 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÷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

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

const long long mod = 1000000007;
long long N, P;
long long A[100009];
map<long long, long long> Count;

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

    // カード j と i を選ぶとする (j < i)
    // 各 i に対して、何個の j で条件を満たすかを数える
    long long Answer = 0;
    for (int i = 1; i <= N; i++) {
        // A[i]*Goal mod 1000000007 = P を満たす整数が Goal
        // Divsion 関数は本の 175 ページ（5.5 節）を参照
        long long Goal = Division(P, A[i], mod);
        Answer += Count[Goal];
        Count[A[i]] += 1;
    }
    cout << Answer << endl;
    return 0;
}
```

残念ながら、$A_i = 0$ のケース※5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※6。

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	$A_i \times A_j \bmod 1000000007 = P$ 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÷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 Division(4, 0, mod) などを呼び出したときはランタイムエラーになりませんが、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++）

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

// a の b 乗を m で割った余りを返す関数
long long Power(long long a, long long b, long long m) {
    long long p = a, Answer = 1;
    for (int i = 0; i < 30; i++) {
        int wari = (1 << i);
        if ((b / wari) % 2 == 1) {
                Answer = (Answer * p) % m;
        }
        p = (p * p) % m;
    }
    return Answer;
}

// a ÷ b を m で割った余りを返す関数
long long Division(long long a, long long b, long long m) {
    return (a * Power(b, m - 2, m)) % m;
}

const long long mod = 1000000007;
long long N, P;
long long A[100009];
map<long long, long long> Count;

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

    // カード j と i を選ぶとする (j < i)
    // 各 i に対して、何個の j で条件を満たすかを数える
    long long Answer = 0;
    for (int i = 1; i <= N; i++) {
        if (A[i] == 0) {
                if (P == 0) Answer += (i - 1);
                else Answer += 0;
        }
        else {
                // A[i]*Goal mod 1000000007 = P を満たす整数が Goal
                long long Goal = Division(P, A[i], mod);
                Answer += Count[Goal];
        }
        Count[A[i]] += 1;
    }
    cout << Answer << endl;
    return 0;
}
```

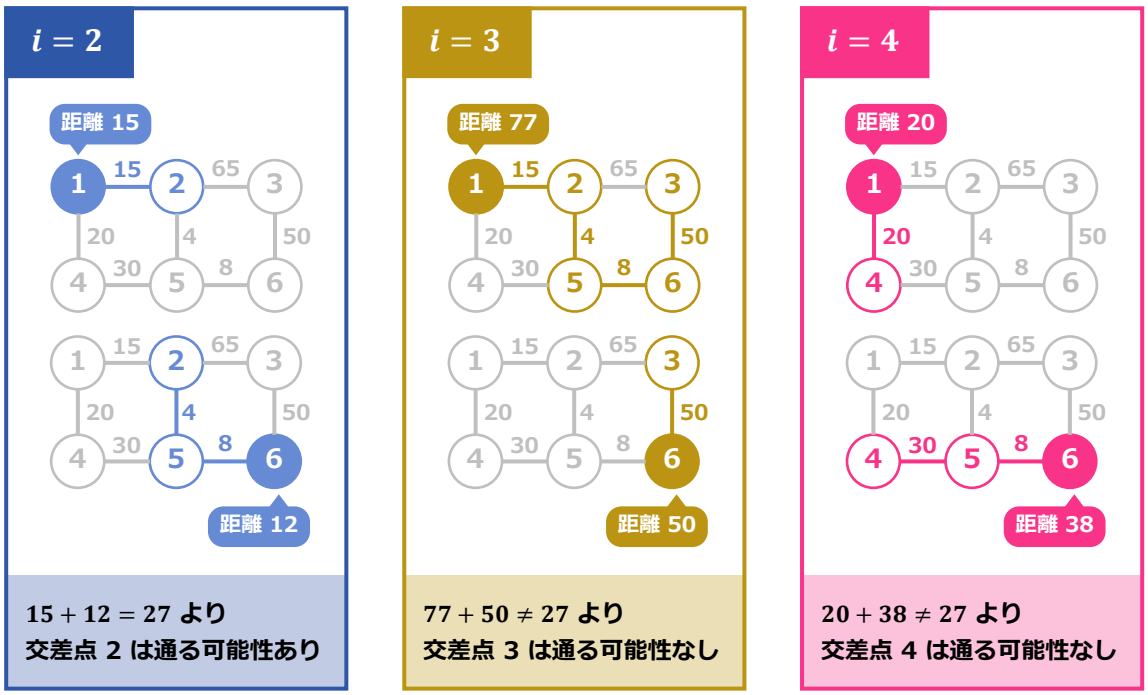
※Python のコードはサポートページをご覧ください

14 問題 C14：Commute Route (Lv. 75)

まず、交差点 p と交差点 q の最短経路長を $\mathrm{dist}(p, q)$ 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。

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

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 `dist1[i]`
- 交差点 N から各交差点までの最短経路長 `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、`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++）

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

// 入力・グラフ
int N, M, A[100009], B[100009], C[100009];
vector<pair<int, int>> G[100009];

// ダイクストラ法
int cur[100009]; bool kakutei[100009];
priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> Q;

// 頂点 1 からの距離、頂点 N からの距離
int dist1[100009];
int distN[100009];

// 始点 start でダイクストラ法を行う関数
void dijkstra(int start) {
    // 配列の初期化
    for (int i = 1; i <= N; i++) kakutei[i] = false;
    for (int i = 1; i <= N; i++) cur[i] = 2000000000;

    // スタート地点をキューに追加
    cur[start] = 0;
    Q.push(make_pair(cur[start], start));

    // ダイクストラ法
    while (!Q.empty()) {
        // 次に確定させるべき頂点を求める
        int pos = Q.top().second; Q.pop();
        if (kakutei[pos] == true) continue;

        // cur[x] の値を更新する
        kakutei[pos] = true;
        for (int i = 0; i < G[pos].size(); i++) {
                int nex = G[pos][i].first;
                int cost = G[pos][i].second;
                if (cur[nex] > cur[pos] + cost) {
                        cur[nex] = cur[pos] + cost;
                        Q.push(make_pair(cur[nex], nex));
                }
        }
    }
}

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

```
    // ダイクストラ法を行う
    dijkstra(1); for (int i = 1; i <= N; i++) dist1[i] = cur[i];
    dijkstra(N); for (int i = 1; i <= N; i++) distN[i] = cur[i];

    // 答えを求める
    int Answer = 0;
    for (int i = 1; i <= N; i++) {
        if (dist1[i] + distN[i] == dist1[N]) Answer += 1;
    }

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

※Python のコードはサポートページをご覧ください

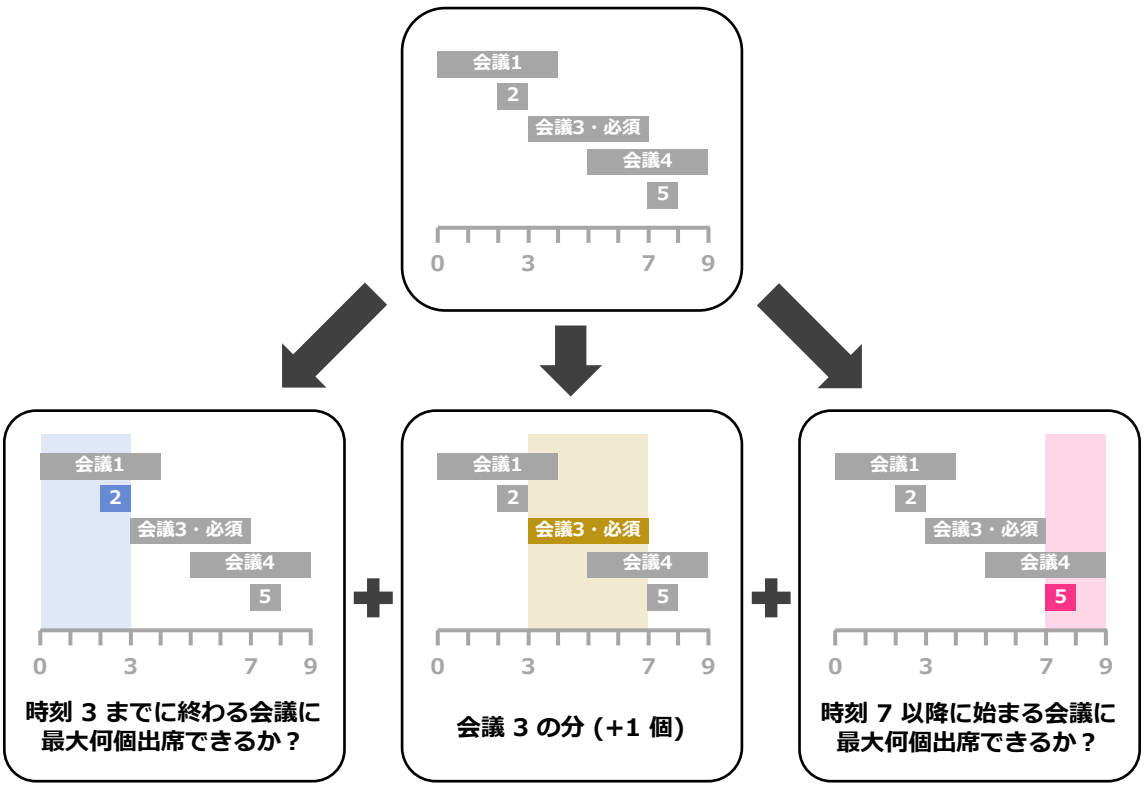
15 問題 C15：Many Meetings (Lv. 85)

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るので、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。



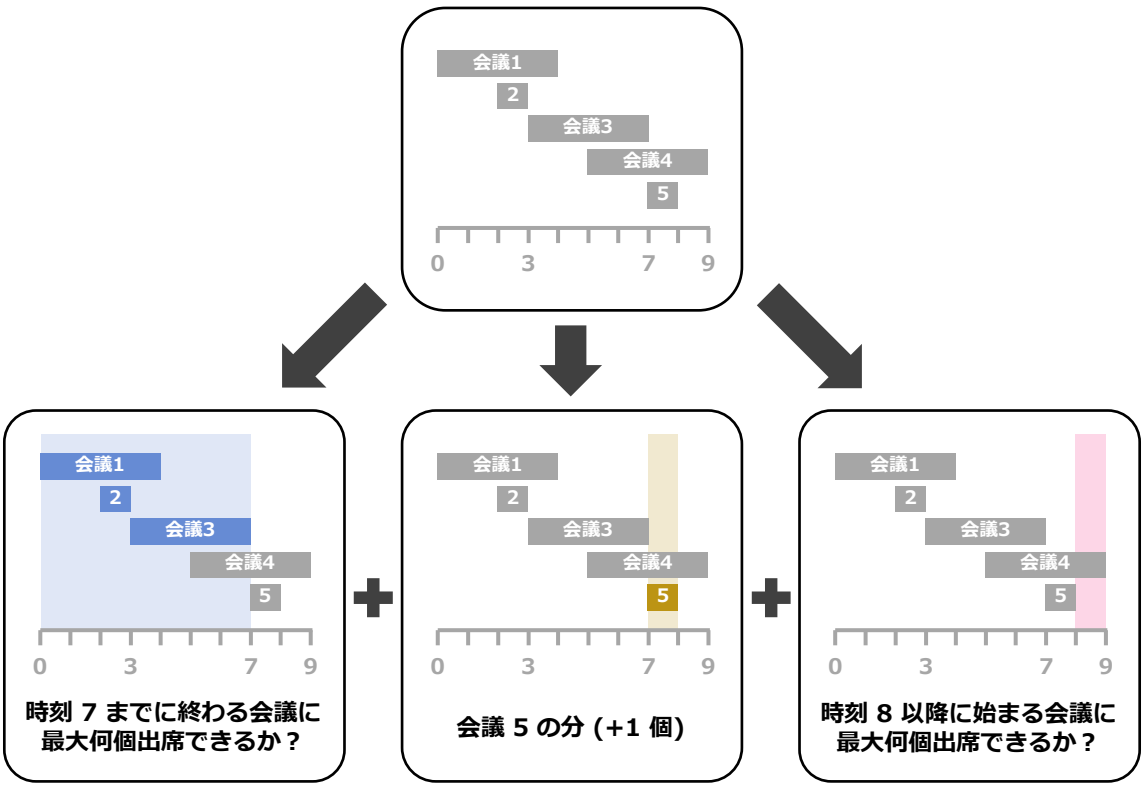
そこで、青線部分の値は 1 であり、赤線部分の値は 1 であるため、答えは 1+1+1=3 となります。

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。

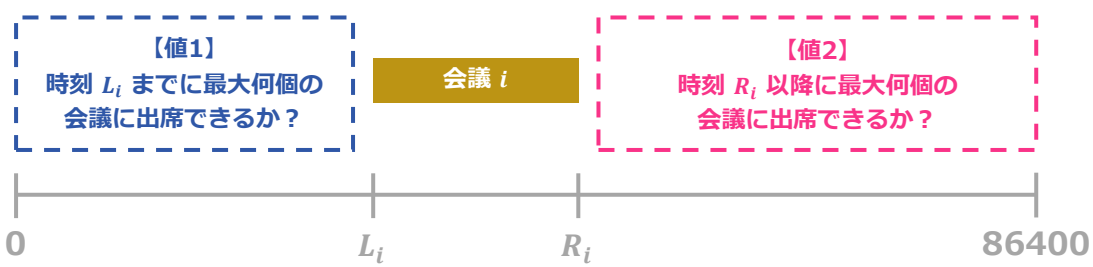
(時刻 7 までで最大何個の会議に出席できるか) + 1
+ (時刻 8 以降で最大何個の会議に出席できるか)

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

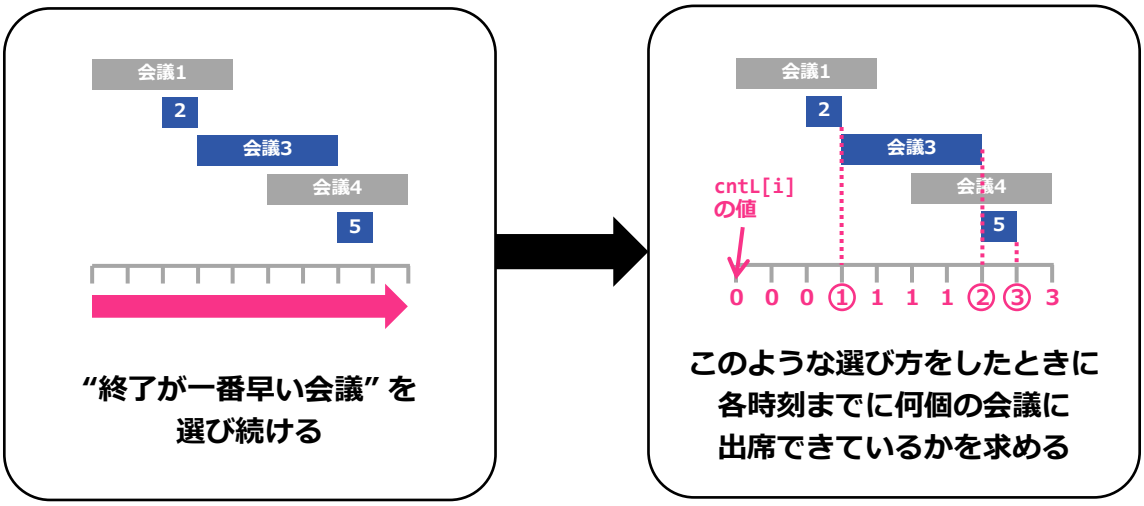
一般のケースでも同じように解くことができます。具体的には、「時刻 L_i から始まり、時刻 R_i に終わる会議に絶対出席しなければならない」というシナリオの場合、以下の 2 つの値を計算することで答えが分かります。



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※8、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、$i = 0,1,\ldots,86400$ に対して、時刻 i までに最大何個の会議に出席できるか `cntL[i]` を計算します（計算方法を以下に示します※9）。



同様に、$i = 0,1,\ldots,86400$ に対して、時刻 i 以降に最大何個の会議に出席できるか `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

```
cntL[L[i]] + 1 + 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 $i = 1,2,\ldots,N$ 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++）

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

// 入力で与えられる変数
int N, K;
int L[100009], R[100009];

// 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]
int cntL[200009];
int cntR[200009];

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

    // 時刻に補正をかける
    for (int i = 1; i <= N; i++) R[i] += K;

    // 左から区間スケジューリング（Part1：ソート）
    vector<pair<int, int>> tmp1;
    for (int i = 1; i <= N; i++) tmp1.push_back(make_pair(R[i], L[i]));
    sort(tmp1.begin(), tmp1.end());

    // 左から区間スケジューリング（Part2：貪欲法）
    int CurrentTime1 = 0; // 現在時刻
    int Num1 = 0; // 現在出席した会議の個数
    for (int i = 0; i < tmp1.size(); i++) {
        if (CurrentTime1 <= tmp1[i].second) {
                CurrentTime1 = tmp1[i].first;
                Num1 += 1;
                cntL[CurrentTime1] = Num1;
        }
    }

    // 右から区間スケジューリング（Part1：ソート）
    vector<pair<int, int>> tmp2;
    for (int i = 1; i <= N; i++) tmp2.push_back(make_pair(L[i], R[i]));
    sort(tmp2.begin(), tmp2.end());
    reverse(tmp2.begin(), tmp2.end());

    // 右から区間スケジューリング（Part2：貪欲法）
    int CurrentTime2 = 200000; // 現在時刻
    int Num2 = 0; // 現在出席した会議の個数
    for (int i = 0; i < tmp2.size(); i++) {
        if (CurrentTime2 >= tmp2[i].second) {
                CurrentTime2 = tmp2[i].first;
                Num2 += 1;
                cntR[CurrentTime2] = Num2;
        }
    }
```

```
    // cntL, cntR を求める
    // 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
    for (int i = 1; i <= 200000; i++) cntL[i] = max(cntL[i], cntL[i - 1]);
    for (int i = 199999; i >= 0; i--) cntR[i] = max(cntR[i], cntR[i + 1]);

    // 答えを求める
    for (int i = 1; i <= N; i++) {
        cout << cntL[L[i]] + 1 + cntR[R[i]] << endl;
    }
    return 0;
}
```

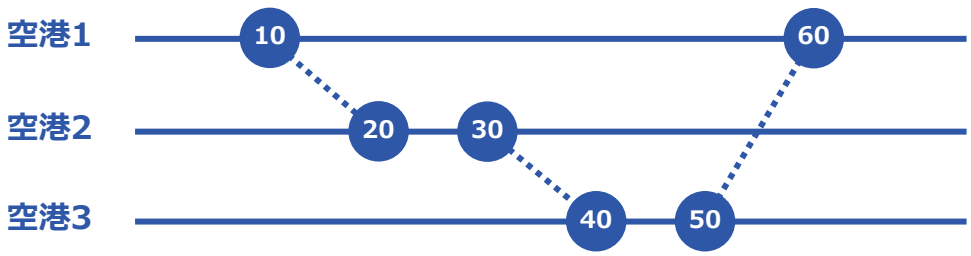
※Python のコードはサポートページをご覧ください

16 問題 C16：Flights (Lv. 85)

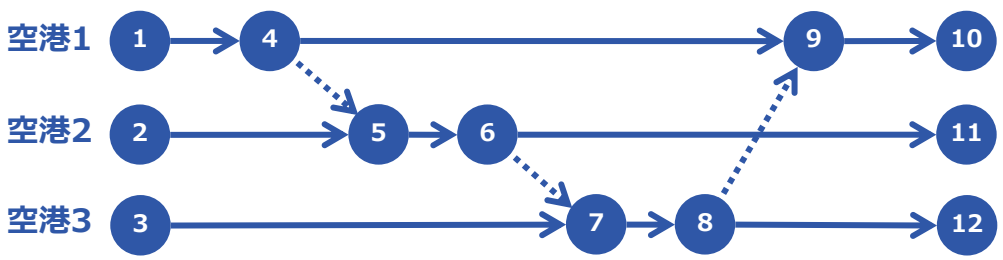
この問題では、必要な乗り継ぎ時間 K が設定されていることがあります。しかし、すべての飛行機の到着時刻 T_i を K 秒遅らせれば、$K = 0$ の場合に帰着させることができるので、本解説では $K = 0$ を仮定します。

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）。

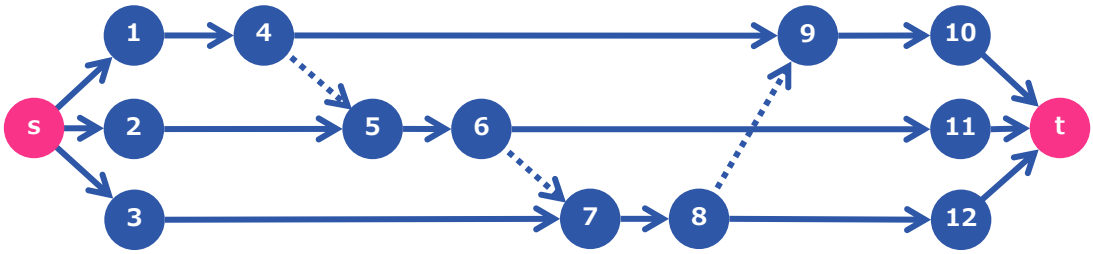


そこで、このダイヤグラムにおける出発時刻・到着時刻を頂点とすると、以下のようなグラフに変換することができます。実線は空港で待つこと、点線は飛行機で移動することに対応します。



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。



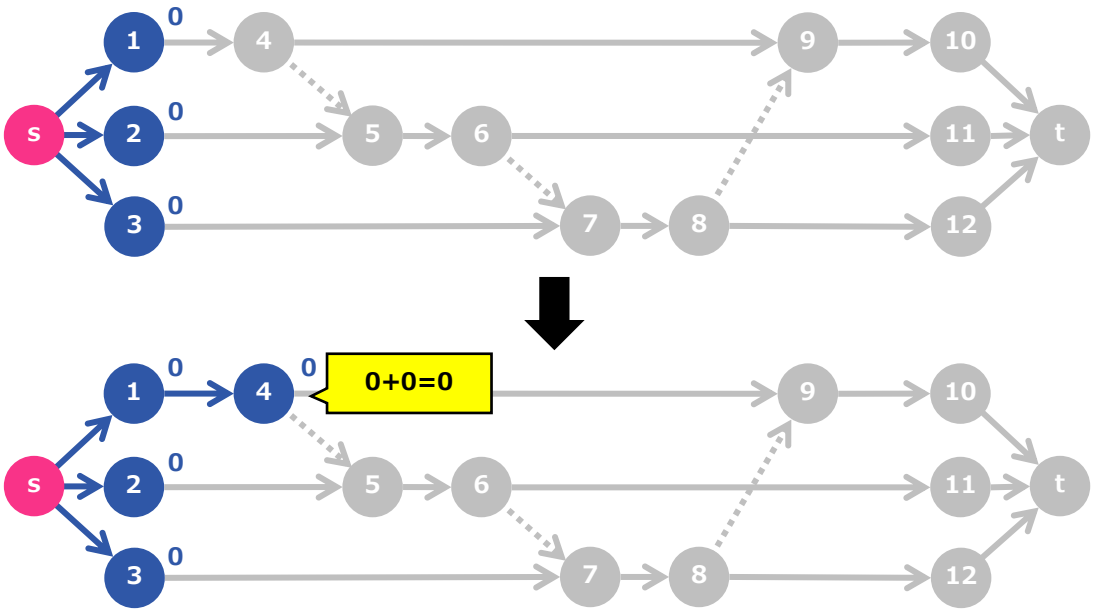
このとき、乗れる飛行機の数の最大値は、始点 s から終点 t まで行く経路における、通る点線の数の最大値となります。先ほどより問題がシンプルになりましたね。

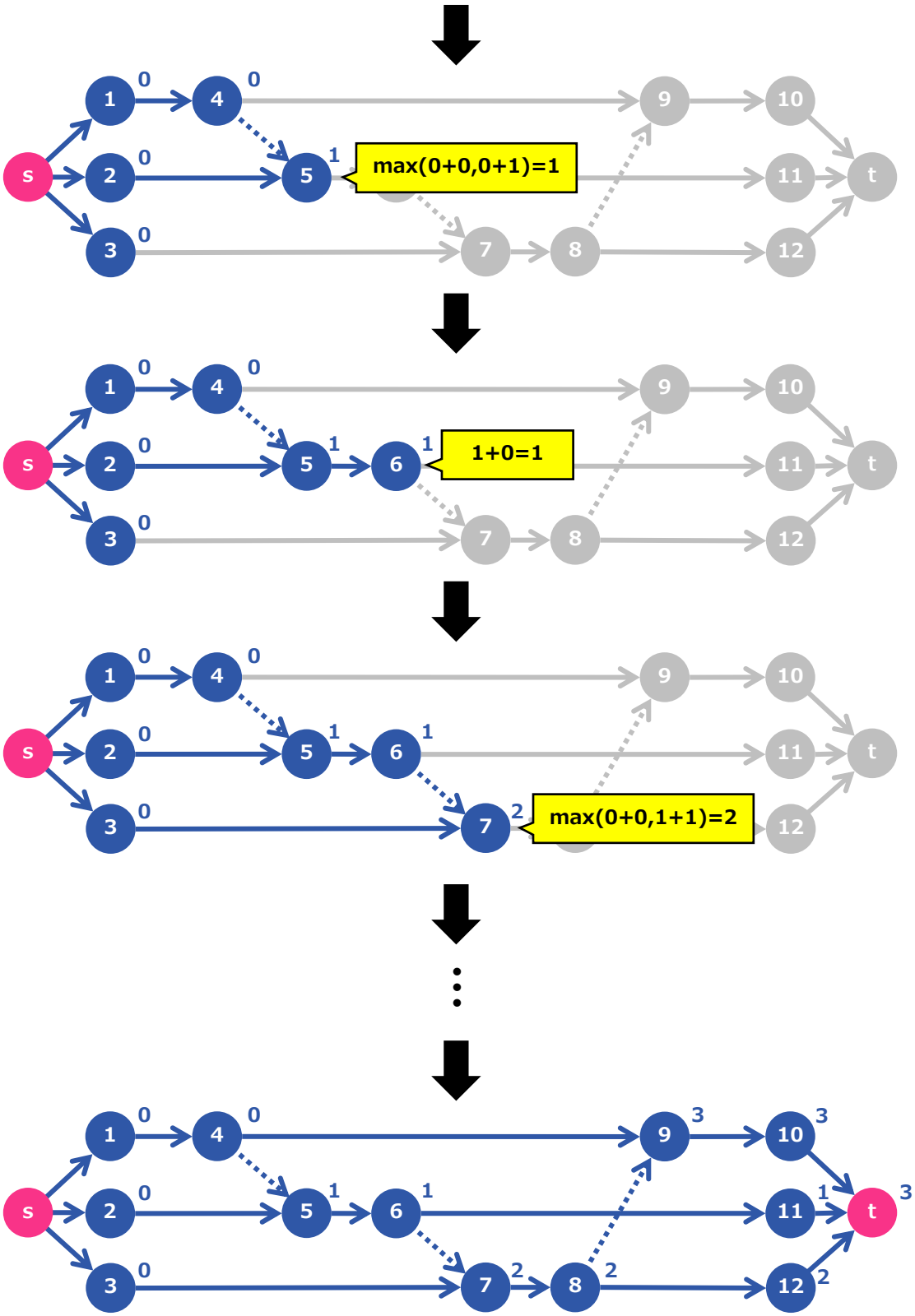
ステップ 2：答えを求める

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++）

```cpp
#include <iostream>
#include <tuple>
#include <vector>
#include <algorithm>
using namespace std;

// 入力部分
int N, M, K;
int A[100009], B[100009], S[100009], T[100009];

// (時刻, 路線番号または空港番号, 出発か到着か)
// 出発 = 2／到着 = 1／最初と最後 = 0
// 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
vector<tuple<int, int, int>> List;

// 頂点番号の情報
int VertS[100009]; // 路線 i の到着
int VertT[100009]; // 路線 i の出発
vector<int> Airport[100009];

// グラフおよび dp[i]
vector<pair<int, int>> G[400009];
int dp[400009];

int main() {
    // 入力
    cin >> N >> M >> K;
    for (int i = 1; i <= M; i++) {
        cin >> A[i] >> S[i] >> B[i] >> T[i];
        T[i] += K; // 到着時刻の補正
    }

    // 頂点となり得る (空港, 時刻) の組を「時刻の早い順に」ソート
    for (int i = 1; i <= M; i++) List.push_back(make_tuple(S[i], i, 2));
    for (int i = 1; i <= M; i++) List.push_back(make_tuple(T[i], i, 1));
```

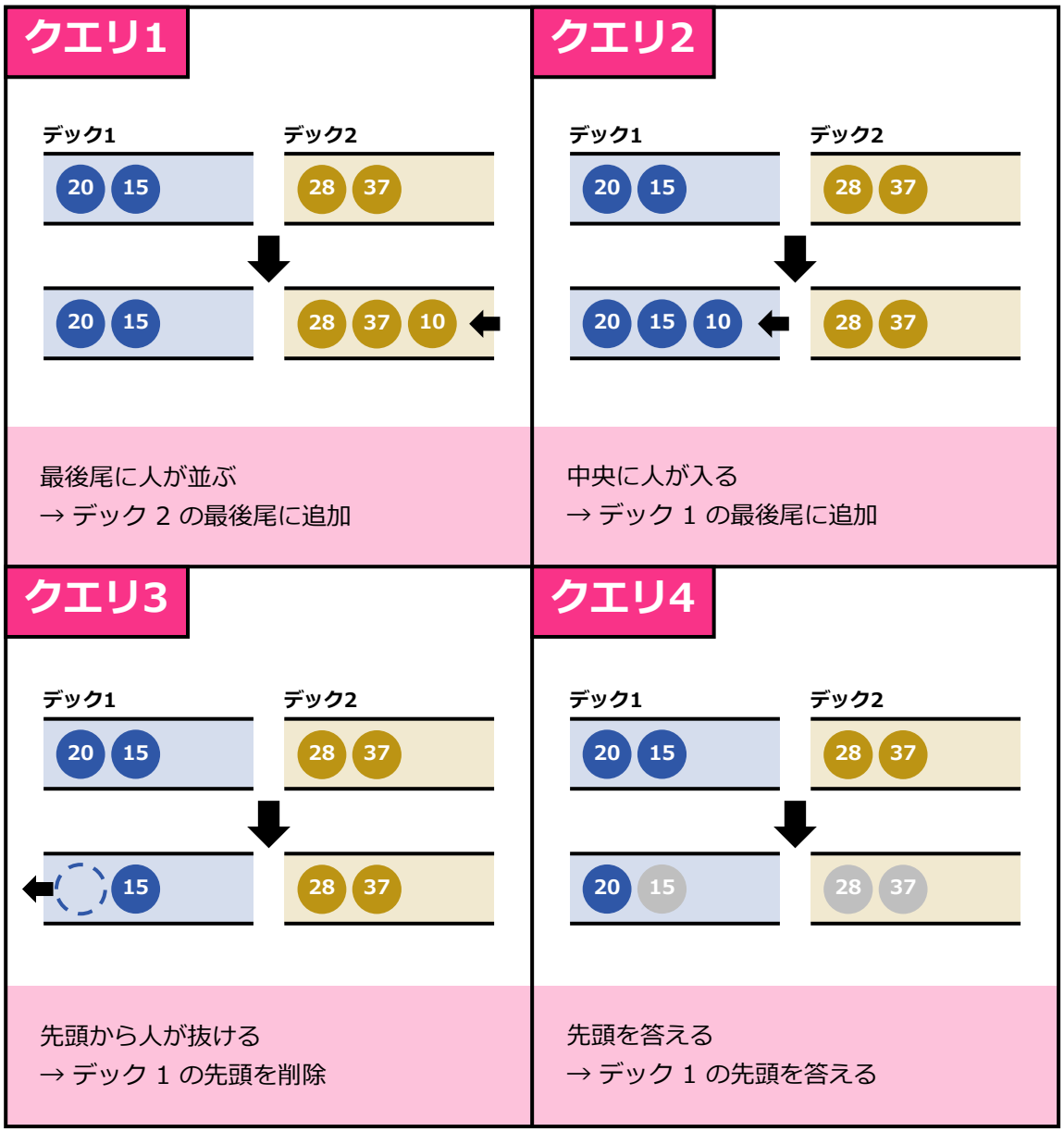
```cpp
    for (int i = 1; i <= N; i++) List.push_back(make_tuple(-1, i, 0));
    for (int i = 1; i <= N; i++) List.push_back(make_tuple(2100000000, i, 0));
    sort(List.begin(), List.end());

    // 各路線の頂点番号を求める
    // ここで、頂点番号は時刻の早い順に 1, 2, ..., List.size() となる
    for (int i = 0; i < List.size(); i++) {
        if (get<2>(List[i]) == 2) VertS[get<1>(List[i])] = i + 1;
        if (get<2>(List[i]) == 1) VertT[get<1>(List[i])] = i + 1;
    }

    // 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）
    for (int i = 0; i < List.size(); i++) {
        if (get<2>(List[i]) == 0) Airport[get<1>(List[i])].push_back(i + 1);
        if (get<2>(List[i]) == 1) Airport[B[get<1>(List[i])]].push_back(i + 1);
        if (get<2>(List[i]) == 2) Airport[A[get<1>(List[i])]].push_back(i + 1);
    }

    // グラフを作る（辺が逆向きになっていることに注意！）
    for (int i = 1; i <= M; i++) {
        G[VertT[i]].push_back(make_pair(VertS[i], 1)); // 路線に対応する辺（点線）
    }
    for (int i = 1; i <= N; i++) {
        for (int j = 0; j < (int)Airport[i].size() - 1; j++) {
                int idx1 = Airport[i][j];
                int idx2 = Airport[i][j + 1];
                G[idx2].push_back(make_pair(idx1, 0)); // 空港で待つことに対応する辺
        }
    }

    // グラフに始点（頂点 0）と終点（頂点 List.size()+1）を追加
    for (int i = 1; i <= N; i++) {
        G[Airport[i][0]].push_back(make_pair(0, 0));
        G[List.size() + 1].push_back(make_pair(Airport[i][Airport[i].size() - 1], 0));
    }

    // 動的計画法によって dp[i] の値を求める
    // 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
    dp[0] = 0;
    for (int i = 1; i <= List.size() + 1; i++) {
        for (int j = 0; j < G[i].size(); j++) {
                dp[i] = max(dp[i], dp[G[i][j].first] + G[i][j].second);
        }
    }

    // 出力
    cout << dp[List.size() + 1] << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

17 問題 C17：Strange Data Structure (Lv. 90)

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※10」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。

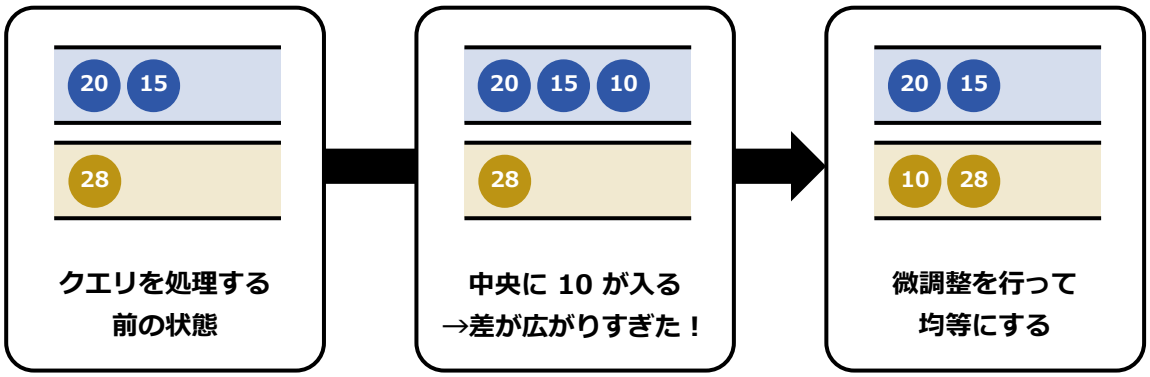
これらのデックを用意した場合、4 種類すべてのクエリを計算量 $O(1)$ で処理することができます。列が $[20, 28, 15, 37]$ のときのデックの変化を以下に示します。



※10　デックについては、本の 301 ページをご覧ください。

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※11、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++）

```cpp
#include <iostream>
#include <deque>
#include <string>
using namespace std;

// 入力で与えられる変数
int Q;
char QueryType[300009]; string X[300009];

// デック
deque<string> Z1, Z2;

int main() {
    // 入力
    cin >> Q;
    for (int i = 1; i <= Q; i++) {
        cin >> QueryType[i];
        if (QueryType[i] == 'A' || QueryType[i] == 'B') cin >> X[i];
    }

    // クエリの処理
    for (int i = 1; i <= Q; i++) {
        // [A] 最後尾に入る
        if (QueryType[i] == 'A') {
                Z2.push_back(X[i]);
        }
        // [B] 中央に入る
        if (QueryType[i] == 'B') {
                Z1.push_back(X[i]);
        }
```

※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。

```
        // [C] 先頭が抜ける
        if (QueryType[i] == 'C') {
                Z1.pop_front();
        }
        // [D] 先頭を答える
        if (QueryType[i] == 'D') {
                cout << Z1.front() << endl;
        }

        // 微調整（前半のデック Z1 が大きすぎる場合）
        while ((int)Z1.size() - (int)Z2.size() >= 2) {
                string r = Z1.back();
                Z1.pop_back();
                Z2.push_front(r);
        }
        // 微調整（後半のデック Z2 が大きすぎる場合）
        while ((int)Z1.size() - (int)Z2.size() <= -1) {
                string r = Z2.front();
                Z2.pop_front();
                Z1.push_back(r);
        }
    }
    return 0;
}
```

※Python のコードはサポートページをご覧ください

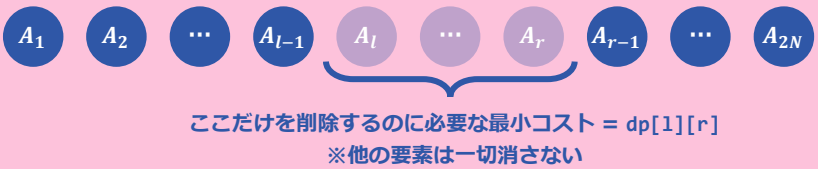
18 問題 C18：Pick Two (Lv. 95)

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[l][r]：範囲 $[A_l, A_{l+1}, \ldots, A_r]$ 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[l][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]=...=dp[2*N-1][2*N]=0 である。

状態遷移

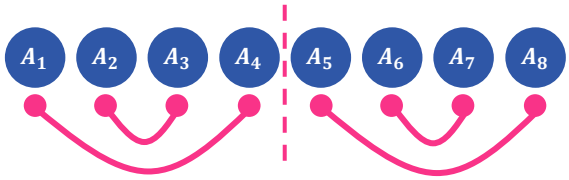
整数列 $[A_l, A_{l+1}, \ldots, A_r]$ を全部消すための方法の一つを以下に示す：

- 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[l+1][r-1]+abs(A[l]-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[l][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]+abs(A[l]-A[r])`
区間 $[A_l, \ldots, A_m]$ と区間 $[A_{m+1}, \ldots, A_r]$ に分割 **※ m は $l \leq m < r$ を満たせば何でも良い**	`dp[l][m]+dp[m+1][r]`

求める答え

`dp[1][2*N]`

ここまで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。計算量は $O(N^3)$ です。なお、dp[l][r] の値は r-l 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。

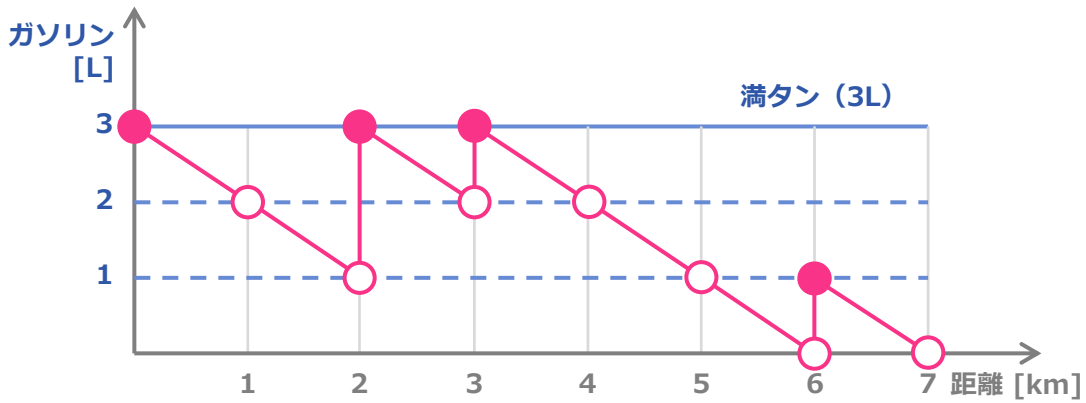
解答例（C++）

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

int N, A[409];
int dp[409][409];

int main() {
    // 入力
    cin >> N;
    for (int i = 1; i <= 2 * N; i++) cin >> A[i];

    // 配列 dp の初期化
    for (int i = 1; i <= 2 * N; i++) {
        for (int j = 1; j <= 2 * N; j++) dp[i][j] = 1000000000;
    }

    // 動的計画法（初期状態）
    for (int i = 1; i <= 2 * N - 1; i++) {
        dp[i][i + 1] = abs(A[i] - A[i + 1]);
    }

    // 動的計画法（遷移）
    for (int LEN = 2; LEN <= 2 * N - 1; LEN++) {
        for (int l = 1; l <= 2 * N - LEN; l++) {
                int r = l + LEN;

                // l 番目と r 番目を消す場合
                dp[l][r] = min(dp[l][r], dp[l + 1][r - 1] + abs(A[l] - A[r]));

                // 2 つの区間を合成させる場合
                for (int m = l; m < r; m++) {
                        dp[l][r] = min(dp[l][r], dp[l][m] + dp[m + 1][r]);
                }
        }
    }

    // 出力
    cout << dp[1][2 * N] << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

19 問題 C19：Gasoline Optimization (Lv. 95)

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（一般のケース）

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。

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

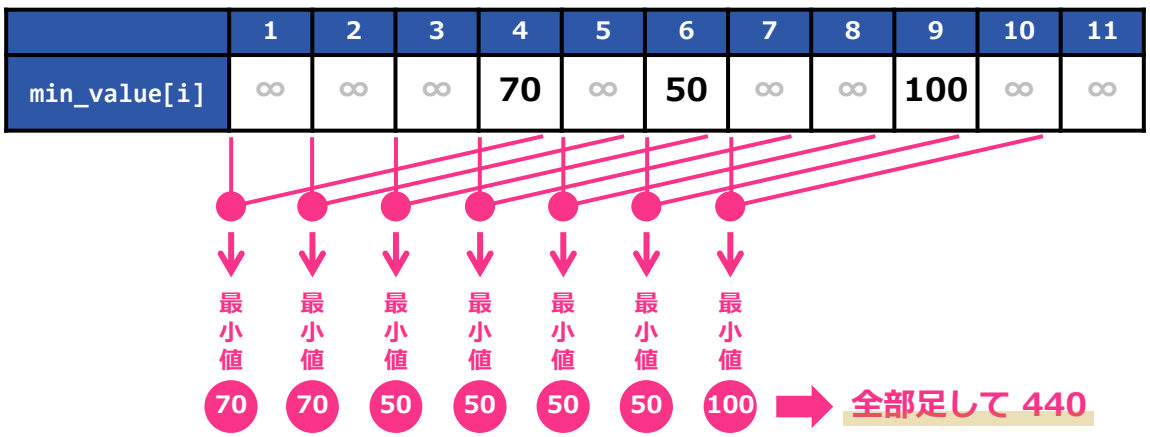
答えを求める

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], …, 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、$i=1,2,\ldots,L-K$ 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。



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], …, min_value[i+K-1]` の値を求めると、計算量が $O(N\log N)$ まで削減されます。

解答例（C++）

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

class SegmentTree {
public:
    long long dat[2100000], siz = 1;
```

```
    // 要素 dat の初期化を行う（最初は全部ゼロ）
    void init(int N) {
        siz = 1;
        while (siz < N) siz *= 2;
        for (int i = 1; i < siz * 2; i++) dat[i] = 0;
    }
    // クエリ 1 に対する処理
    void update(int pos, long long x) {
        pos = pos + siz - 1;
        dat[pos] = x;
        while (pos >= 2) {
                pos /= 2;
                dat[pos] = min(dat[pos * 2], dat[pos * 2 + 1]);
        }
    }
    // クエリ 2 に対する処理
    // 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
    long long query(int l, int r, int a, int b, int u) {
        if (r <= a || b <= l) return (1LL << 60); // 一切含まれない場合
        if (l <= a && b <= r) return dat[u]; // 完全に含まれる場合
        int m = (a + b) / 2;
        long long AnswerL = query(l, r, a, m, u * 2);
        long long AnswerR = query(l, r, m, b, u * 2 + 1);
        return min(AnswerL, AnswerR);
    }
};

long long N, L, K;
long long A[700009], C[700009];
long long Min_Value[700009];
SegmentTree Z;

int main() {
    // 入力
    cin >> N >> L >> K;
    for (int i = 1; i <= N; i++) cin >> A[i] >> C[i];
    // 各地点での「値段の最小値」を求める
    for (int i = 1; i <= L - 1; i++) Min_Value[i] = (1LL << 60);
    for (int i = 1; i <= N; i++) Min_Value[A[i]] = min(Min_Value[A[i]], C[i]);

    // セグメント木に載せる
    Z.init(L);
    for (int i = 1; i <= L - 1; i++) Z.update(i, Min_Value[i]);

    // 答えを求める
    long long Answer = 0;
    for (int i = 1; i <= L - K; i++) {
        long long val = Z.query(i, i + K, 1, Z.siz + 1, 1);
        if (val == (1LL << 60)) {
                cout << "-1" << endl;
                return 0;
        }
        Answer += val;
    }
    cout << Answer << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

20 問題 C20：Mayor’ s Challenge (Lv. 99)

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※12、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。

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

int N, K, L, A[401], B[401], C[51][51];

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

    // 出力
    for (int i = 1; i <= K; i++) {
        cout << rand() % L + 1 << endl; // 1 以上 L 以下のランダムな整数
    }

    return 0;
}
```

この解法の欠点は、特別区に割り当てられ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。

※12　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）。

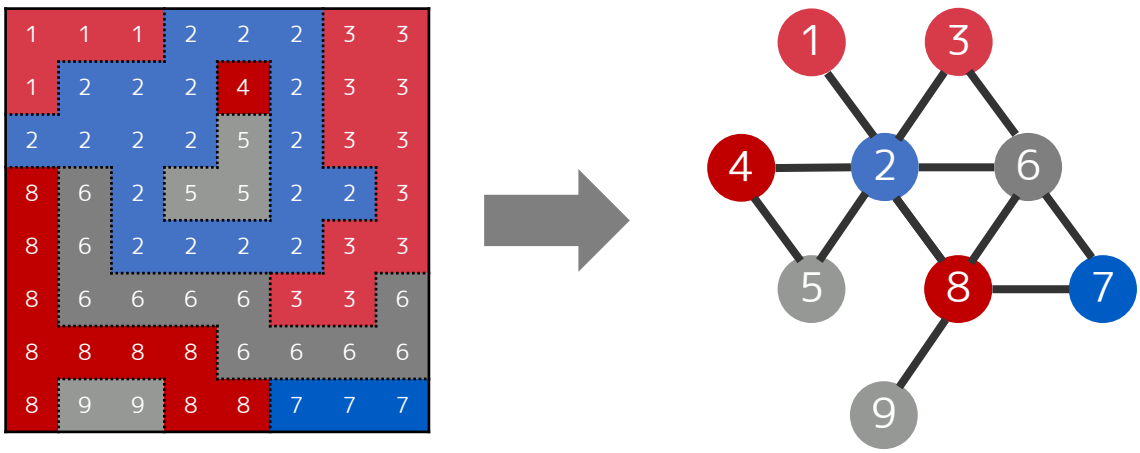
```
1   // 解のランダム生成を 10,000 回
2   double best_score = 0.0;
3   int answer[401];
4   for (int repeat = 1; repeat <= 10000; repeat++) {
5       // 特別区を 20 個ずつの地区にランダムに割り当て
6       int region[401];
7       for (int i = 1; i <= K; i++) {
8           region[i] = i % L + 1;
9       }
10      random_shuffle(region + 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は…

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。

```
// グラフの作成
vector<int> G[401];
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        if (i != N && C[i][j] != 0 && C[i + 1][j] != 0 && C[i][j] != C[i + 1][j]) {
                G[C[i][j]].push_back(C[i + 1][j]);
                G[C[i + 1][j]].push_back(C[i][j]);
        }
        if (j != N && C[i][j] != 0 && C[i][j + 1] != 0 && C[i][j] != C[i][j + 1]) {
                G[C[i][j]].push_back(C[i][j + 1]);
                G[C[i][j + 1]].push_back(C[i][j]);
        }
    }
}
// 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）
for (int i = 1; i <= K; i++) {
    sort(G[i].begin(), G[i].end());
    G[i].erase(unique(G[i].begin(), G[i].end()), G[i].end());
}
```

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めます。

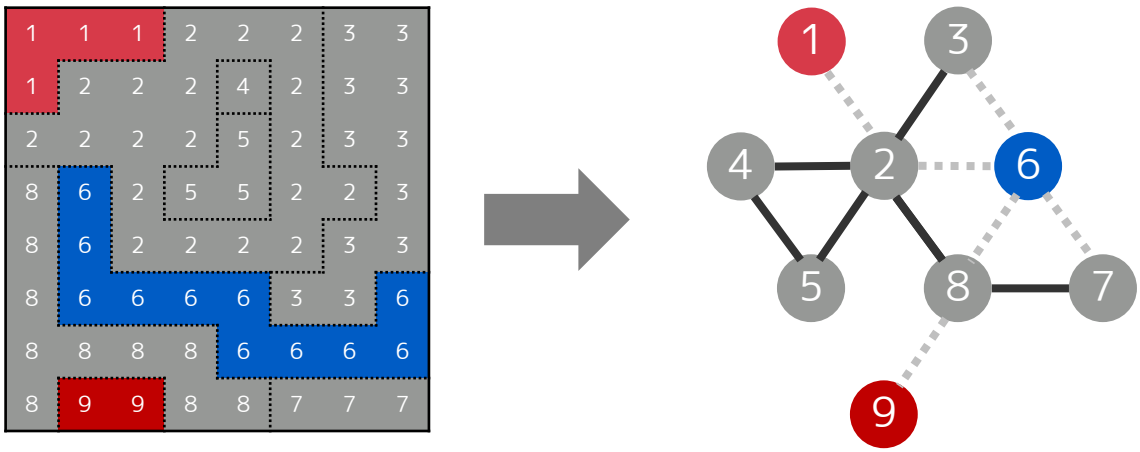
$K = 400$ 頂点のグラフが与えられる。これを $L = 20$ 個の**連結な**部分グラフに分割する方法のなかで、それぞれの部分グラフの「A_i の合計」および「B_i の合計」をできるだけ平等にするものを見つけよ。

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なものです。

- 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※13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。

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

int N, K, L, A[401], B[401], C[51][51]; vector<int> G[401];

// 深さ優先探索を行う関数 dfs（9.2 節を参照）
int counter = 0;
bool visited[401]; int ranking[401];
void dfs(int pos) {
    visited[pos] = true;
    counter += 1;
    ranking[pos] = counter;
    for (int i = 0; i < G[pos].size(); i++) {
        int nex = G[pos][i];
        if (visited[nex] == false) {
                dfs(nex);
        }
    }
}

int main() {
    // 入力（省略・C20 の解説 1 ページ目を参照）
:
    // グラフの作成（省略・C20 の解説 3 ページ目を参照）
:
    // 深さ優先探索（DFS）を行う（頂点 1 からスタートする）
    for (int i = 1; i <= K; i++) {
        visited[i] = false;
    }
    dfs(1);

    // 出力
    for (int i = 1; i <= K; i++) {
        if (ranking[i] <= K - L + 1) {
                // ranking[i] が 381 以下なら、特別区 1 に割り当て
                cout << 1 << endl;
        }
        else {
                // ranking[i] が 382 以上なら、特別区 2～20 に割り当て
                cout << ranking[i] - (K - L) << endl;
        }
    }

    return 0;
}
```

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ってきます。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

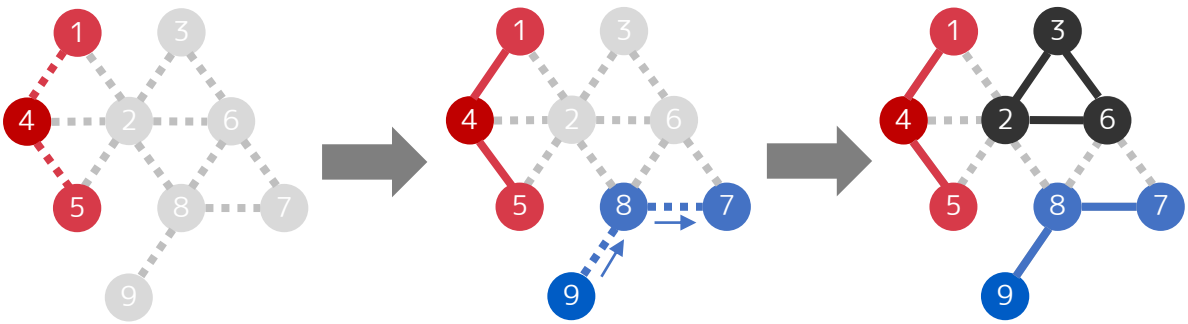
本書の 6.4 節・7.1 節で紹介した貪欲法は、一手先のことしか考えずに答えを決めていく方法です。この問題では、まず次のような貪欲法のアイデアが思いつくでしょう。

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。

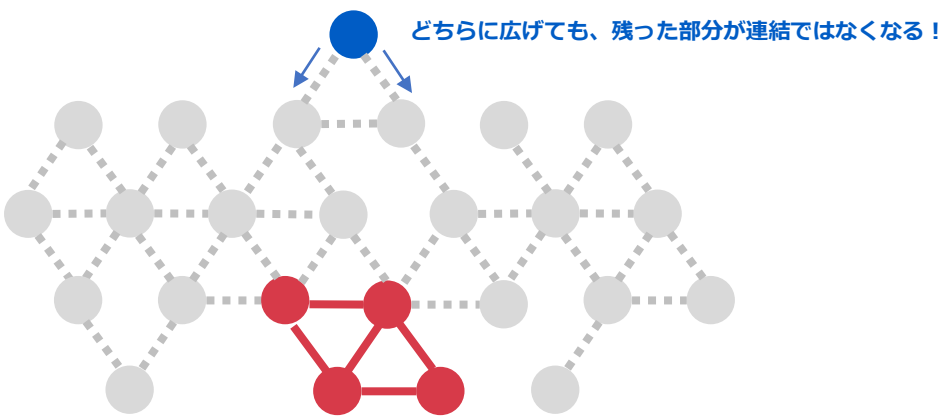
- まだ残っている頂点を起点とし、頂点数が 20 になるまで広げていく。ただし、残った部分も連結にならなければならない。

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）。



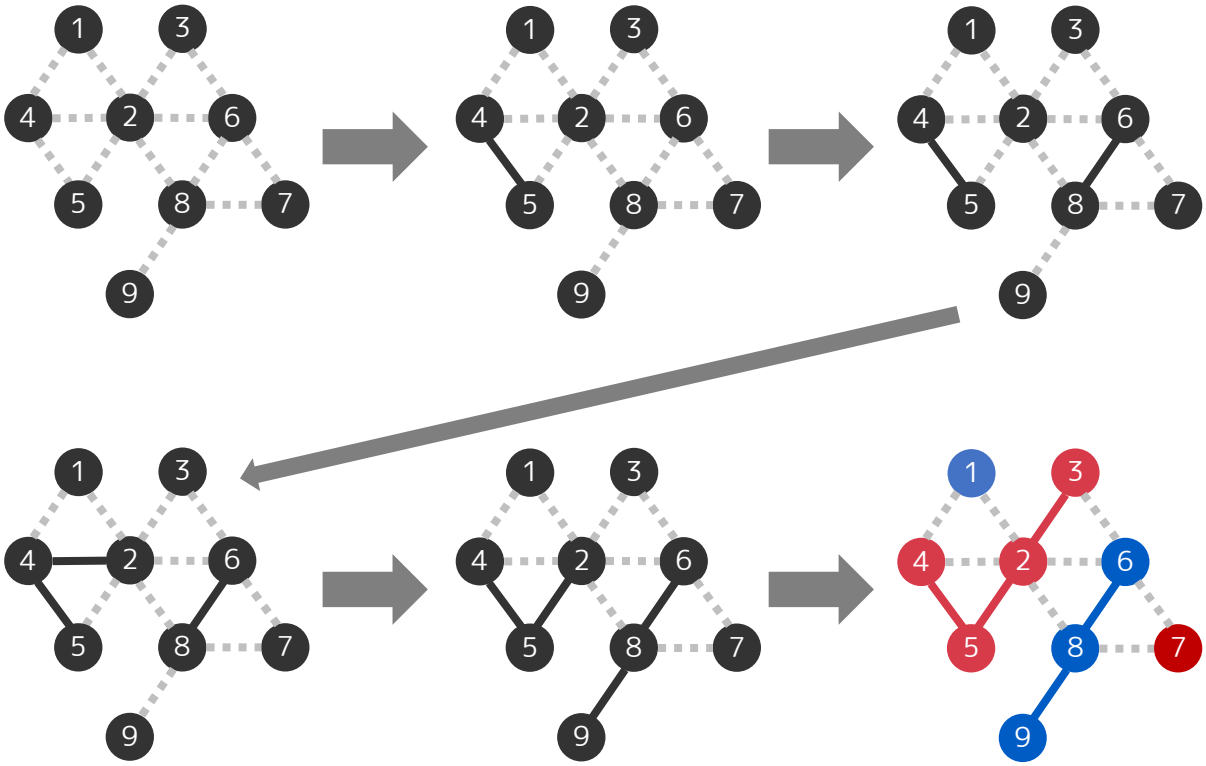
しかし、この貪欲法はあまり上手くいきません。なぜなら、残った部分の連結性を保ったまま頂点数が 20 になるまで広げられない場合が多く、結果として半分程度の頂点が使われずに残ってしまうからです。そのため、これを実装したプログラムを提出しても、得点は 464,392 点にとどまります。



「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ったので、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 の地区を合併するのではなく、頂点 2 の地区と頂点 3 の地区を合併すべきだった、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、size[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。

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

// Union-Find 木（省略・実装は本書の 9.6 節を参照）
:
int N, K, L, A[401], B[401], C[51][51]; vector<int> G[401];
int answer[401];

int main() {
    // 入力（省略・C20 の解説 1 ページ目を参照）
:
    // グラフの作成（省略・C20 の解説 3 ページ目を参照）
:
    // 貪欲法（併合を K - L = 380 回繰り返す）
    UnionFind uf;
    uf.init(K);
    for (int i = 1; i <= K - L; i++) {
        int min_size = 1000000000, vertex1 = -1, vertex2 = -1;
        for (int j = 1; j <= K; j++) {
                for (int k = 0; k < G[j].size(); k++) {
                        int v = G[j][k];
                        if (uf.same(j, v) == false) {
                                // 頂点 j の地区と頂点 v の地区を併合すると…？
                                int size1 = uf.siz[uf.root(j)];
                                int size2 = uf.siz[uf.root(v)];
                                if (min_size > size1 + size2) {
                                        min_size = size1 + size2;
                                        vertex1 = j;
                                        vertex2 = v;
                                }
                        }
                }
        }
        uf.unite(vertex1, vertex2);
    }

    // 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）
    vector<int> roots;
    for (int i = 1; i <= K; i++) {
        roots.push_back(uf.root(i));
    }
    sort(roots.begin(), roots.end());
    roots.erase(unique(roots.begin(), roots.end()), roots.end());
    for (int i = 1; i <= K; i++) {
        for (int j = 0; j < roots.size(); j++) {
                if (roots[j] == uf.root(i)) {
                        answer[i] = j + 1;
                }
        }
    }

```

```
    // 出力
    for (int i = 1; i <= K; i++) {
        cout << answer[i] << endl;
    }

    return 0;
}
```

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。

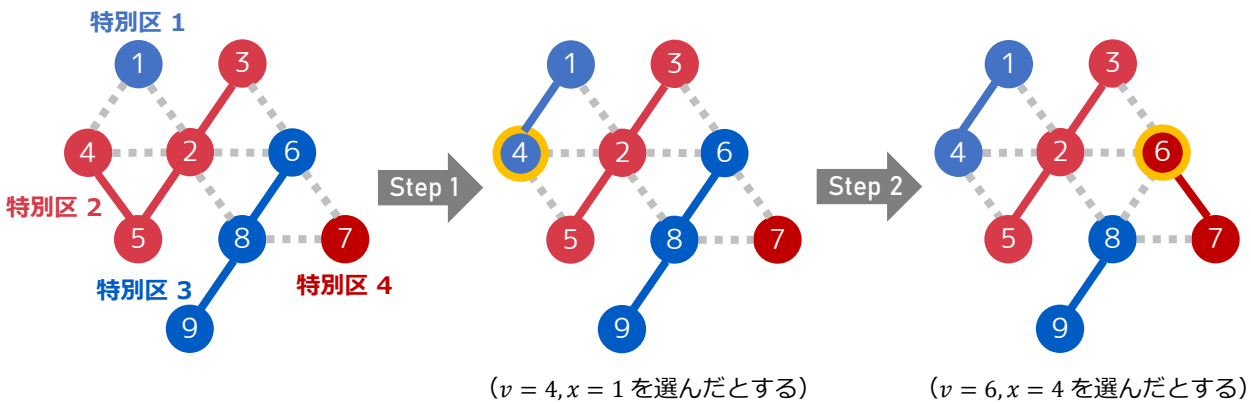
山登り法を使ってみよう

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きます。



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ったか」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っておきましょう※14。

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。

```
// 深さ優先探索（9.2 節を参照）
bool visited[401];
void dfs(int pos) {
    visited[pos] = true;
    for (int i = 0; i < G[pos].size(); i++) {
        int nex = G[pos][i];
        if (answer[nex] == answer[pos] && visited[nex] == false) dfs(nex);
    }
}

// どのくらい格差が小さいかのスコアを返す関数
double get_score() {
    // 答えが正しいかを深さ優先探索（DFS）を使って確認
    for (int i = 1; i <= K; i++) visited[i] = false;
    for (int i = 1; i <= L; i++) {
        // 特別区 i に属する頂点 pos を探す
        int pos = -1;
        for (int j = 1; j <= K; j++) {
                if (answer[j] == i) pos = j;
        }
        if (pos == -1) return 0.0; // 存在しない特別区がある！
        dfs(pos);
    }
    for (int i = 1; i <= K; i++) {
        if (visited[i] == false) return 0.0; // 連結ではない特別区がある！
    }
    // スコアの計算
    int p[21], q[21];
    for (int i = 1; i <= L; i++) {
        p[i] = 0;
        q[i] = 0;
    }
    for (int i = 1; i <= K; i++) {
        p[answer[i]] += A[i];
        q[answer[i]] += B[i];
    }
    int pmin = *min_element(p + 1, p + L + 1);
    int pmax = *max_element(p + 1, p + L + 1);
    int qmin = *min_element(q + 1, q + L + 1);
    int qmax = *max_element(q + 1, q + L + 1);
    return min(double(pmin) / pmax, double(qmin) / qmax);
}
```

※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。

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

// Union-Find 木（省略・実装は本書の 9.6 節を参照）
:
int N, K, L, A[401], B[401], C[51][51]; vector<int> G[401];
int answer[401];

// 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）
:
int main() {
    // 入力（省略・C20 の解説 1 ページ目を参照）
:
    // グラフの作成（省略・C20 の解説 3 ページ目を参照）
:
    // 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）
:
    // 山登り法（0.95 秒ループを回す）
    double TIME_LIMIT = 0.95;
    int ti = clock();
    double current_score = get_score();
    while (double(clock() - ti) / CLOCKS_PER_SEC < TIME_LIMIT) {
        int v = rand() % K + 1; // 1 以上 K 以下のランダムな整数
        int x = rand() % L + 1; // 1 以上 L 以下のランダムな整数
        int old_x = answer[v];
        // とりあえず変更し、スコアを評価する
        answer[v] = x;
        double new_score = get_score();
        // 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）
        if (new_score!=0.0 && current_score<=new_score) current_score = new_score;
        else answer[v] = old_x;
    }

    // 出力
    for (int i = 1; i <= K; i++) {
        cout << answer[i] << endl;
    }

    return 0;
}
```

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ています。

※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。

```
int v, x;
do {
    v = rand() % K + 1; // 1 以上 K 以下のランダムな整数
    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
} 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。

```
// スコアの変化に応じて、変更を採用する確率を決める
double rand_value = double(rand() + 0.5) / (RAND_MAX + 1.0); // 0〜1 のランダムな実数
double temp = 0.0040 - 0.0039 * (double(clock() - ti) / CLOCKS_PER_SEC / TIME_LIMIT);
if (new_score != 0.0 && rand_value < exp((new_score - current_score) / temp)) {
    current_score = new_score;
}
else {
    answer[v] = old_x;
}
```

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。