6.5 最終確認問題 21-25 の解答

問題 21 (1)

関数 $f(x) = e^x$ に対して、その微分である $f'(x)$ も e^x になるという性質があります。

したがって、$y = e^x$ 上の点 $(1, e)$ における接線の傾きは、$f'(1) = e^1 = e$ となります。だから、接線の方程式は $y = ex + b$ という形で表せますが、このなかで点 $(1, e)$ を通るのは $b = 0$ のときです。

よって、点 $(1, e)$ における接線の方程式は $\boldsymbol{y = ex}$ です。

問題 21 (2)

接線 $y = ex$ と直線 $y = 2$ が交わるとき、$ex = 2$ になるので、$x = \frac{2}{e}$ です。したがって、この 2 直線の交点の座標は $\boldsymbol{\left(\frac{2}{e}, 2\right)}$ です。

この点の x 座標を小数で表すと、0.735758882… となります。

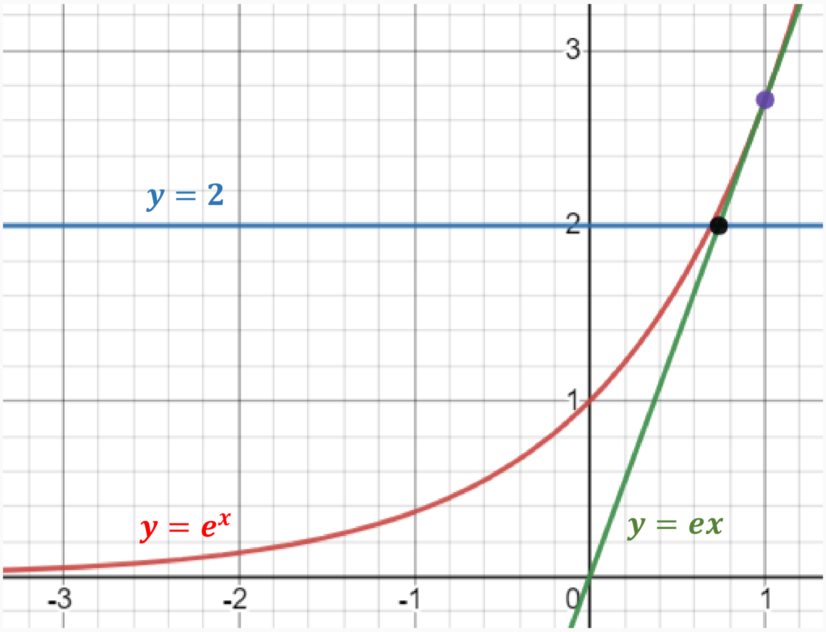


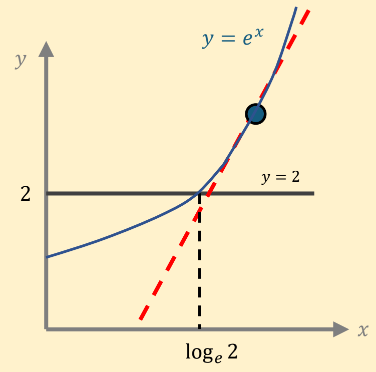
図. $y = e^x$ とその接線などのグラフ（desmos.com で描画）

問題 21 (3)

$\log_e 2$ の値はニュートン法（→**4.3 節**）を使って求められます。曲線 $y = e^x$ の y 座標が 2 となる点の x 座標が $\log_e 2$ であるから、以下の方針で求めることができます。

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

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



そのため、以下のようなプログラムを書けばよいです。なお、exp(x) は e^x を返す関数です。別の方法として、代わりに pow(2.718281828, x) と書いても、ほぼ同じ結果が得られます。

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

int main() {
    double r = 2.0; // y = e^x と y = 2 の交点を求めたいから
    double a = 1.0; // 初期値を適当に 1.0 にセットする

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

        // 接線の式 y = sessen_a * x + sessen_b を求める
        double sessen_a = zahyou_y;      ← コード 4.3.1 からの変更部分
        double sessen_b = zahyou_y - sessen_a * zahyou_x;

        // 次の a の値 next_a を求める
        double next_a = (r - sessen_b) / sessen_a;
        printf("Step #%d: a = %.15lf -> %.15lf¥n", i, a, next_a);
        a = next_a;
    }
    return 0;
}
```

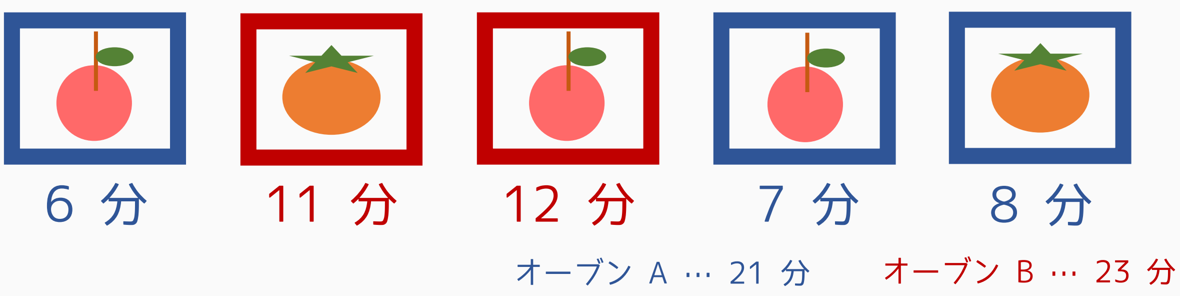
このとき、出力は以下のようになります。急激に $\log_e 2 = 0.693147180559945\ldots$ に近づき、たった 5 回で 15 桁目まで一致します。

```
Step #1: a = 1.000000000000000 -> 0.735758882342885
Step #2: a = 0.735758882342885 -> 0.694042299918915
Step #3: a = 0.694042299918915 -> 0.693147581059771
Step #4: a = 0.693147581059771 -> 0.693147180560026
Step #5: a = 0.693147180560026 -> 0.693147180559945
```

Python・JAVA・C のソースコードは、GitHub の chap6-21_25.md をご覧ください。

問題 22

使う 2 台のオーブンを「オーブン A」「オーブン B」とします。オーブン A で消費される時間を a、オーブン B で消費される時間を b とすると、料理にかかる全体の時間は $\max(a, b)$ となります。ここで、どのようにオーブンの割り当てを決めても、$a + b$ の値は $sumT = T_1 + T_2 + \cdots + T_N$ で変わらないので、a が決まれば $b = sumT - a$ と自動的に決まり、料理にかかる全体の時間も $\max(a, sumT - a)$ と決まります。



さて、どのような a が「実現可能」なのでしょうか？上図の例では、実現可能な a をすべて挙げると 0, 6, 7, 8, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 23, 24, 25, 26, 27, 29, 30, 31, 32, 33, 36, 37, 38, 44 分となります。

実は、実現可能な a は、**節末問題 3.7.4** と非常によく似た動的計画法のアルゴリズムを使って、計算量 $O(N \times sumT)$ ですべて挙げることができます。

用意する配列（二次元配列）

dp[i][j]：料理 i までの中から、オーブン A で消費する時間の和（以下、単に消費時間と呼ぶ）が j になる組合せが存在するなら `true`、そうでなければ `false`

動的計画法の遷移（$i = 0$）

明らかに「何も選ばない」という方法しか存在しないので、

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

となります。

動的計画法の遷移（$i = 1, 2, \ldots, N$ の順に計算）

総和が j になるように料理 i までの中から選ぶ方法は、以下の 2 つがあります。（最後の行動 [料理 i を焼くオーブン] で場合分けします）

- 料理 $i-1$ までの消費時間が $j - A_i$ であり、料理 i をオーブン A で焼く
- 料理 $i-1$ までの消費時間が j であり、料理 i をオーブン A では焼かない

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

最終的に、$dp[N][x] = \texttt{true}$ のとき $a = x$ が実現可能になります。実現可能な a の中で、$\max(a, sumT - a)$ が最小になるものが答えになります。

この解法を C++ で実装すると、以下のようになります。

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

int N, T[109]; bool dp[109][100009];

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

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

```
    // 動的計画法
    for (int i = 1; i <= N; i++) {
        for (int j = 0; j <= sumT; j++) {
            if (j < T[i]) {
                if (dp[i - 1][j] == true) dp[i][j] = true;
                else dp[i][j] = false;
            }
            if (j >= T[i]) {
                if (dp[i-1][j] == true || dp[i-1][j-T[i]] == true) dp[i][j] = true;
                else dp[i][j] = false;
            }
        }
    }

    // 答えを計算して出力
    int answer = (1 << 30);
    for (int i = 0; i <= sumT; i++) {
        if (dp[N][i] == true) {
            int cooking_time = max(i, sumT - i);
            answer = min(answer, cooking_time);
        }
    }
    cout << answer << endl;
    return 0;
}
```

Python・JAVA・C のソースコードは、GitHub の chap6-21_25.md をご覧ください。

問題 23

エラトステネスのふるい（→**4.4.1 項**）を使うと、N 以下の素数を $O(N \log \log N)$ 時間で列挙できます。しかし、この問題で列挙すべきなのは「L 以上 R 以下の素数」であります。以下のようにアルゴリズムを少し変えてみましょう。

1. 最初、整数 $L, L+1, \dots, R$ を書く。
2. 書かれている全ての 2 の倍数に × を付ける。例外として、2 には × を付けない。
3. 書かれている全ての 3 の倍数に × を付ける。例外として、3 には × を付けない。
4. 書かれている全ての 4 の倍数に × を付ける。例外として、4 には × を付けない。
5. （中略）
6. 書かれている全ての $\lfloor\sqrt{R}\rfloor$ の倍数に × を付ける。例外として、$\lfloor\sqrt{R}\rfloor$ には × を付けない。
7. 無印のまま残った整数だけが素数である。

次に、実装方法を考えます。プログラミングでは整数を直接書くことはできないので、代わりに長さ $R-L+1$ の配列 `prime` を持って、`prime[x]` には「整数 $x+L$ が無印かどうか」（無印なら `true`、×が付けられているなら `false`）を記録すると、うまく実装できます。

i 番目の操作で×が付けられるのは $\lceil L/i \rceil \times i, \lceil L/i+1 \rceil \times i, \dots, \lfloor R/i \rfloor \times i$ であることに注意してください。×が付けられるのは大体 $(R-L)/i$ 個なので、全体計算量は

$$\frac{R-L}{2}+\frac{R-L}{3}+\cdots+\frac{R-L}{\sqrt{R}}=O\left((R-L)\log\sqrt{R}\right)$$

となります。この証明は、**4.4,4 節・4.4.5 節**を参照してください。

この解法を C++ で実装すると、以下のようになります。

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

long long L, R; bool prime[500009];

int main() {
	// 入力
	cin >> L >> R;

	// 配列の初期化・L=1 のときの場合分け（コーナーケース）
	for (long long i = 0; i <= R - L; i++) {
		prime[i] = true;
	}
	if (L == 1) prime[0] = false;

	// ふるい
	for (long long i = 2; i * i <= R; i++) {
		long long min_value = ((L + i - 1) / i) * i; // L 以上で最小の i の倍数
		// L 以上 R 以下の（i を除く）i の倍数すべてにバツを付ける
		for (long long j = min_value; j <= R; j += i) {
			if (j == i) continue;
			prime[j - L] = false;
		}
	}

	// 個数を数えて出力
	long long answer = 0;
	for (long long i = 0; i <= R - L; i++) {
		if (prime[i] == true) answer += 1;
	}
	cout << answer << endl;
	return 0;
}
```

さらに工夫すると、計算量 $O\left(\left(\sqrt{R}+(R-L)\right)\log\log\sqrt{R}\right)$ で解くこともできます。あらかじめエラトステネスのふるいで $\sqrt{R}$ 以下の素数を求めておけば、「合成数 (4, 6, 8, 9, …) の倍数に × を付ける」といった無駄な操作が省けて、この計算量になります。

Python・JAVA・C のソースコードは、GitHub の chap6-21_25.md をご覧ください。

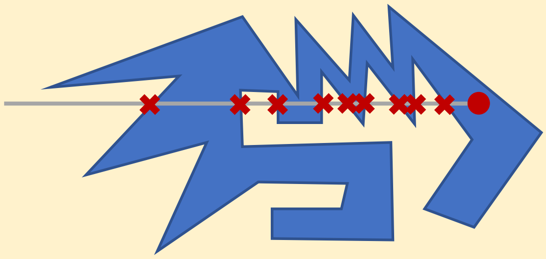
問題 24

計算幾何学（→**4.1 節**）の問題です。この問題では、多角形が凸（内角がすべて 180 度未満の多角形）とは限らないので、複雑に入れ組んでいる場合もあります。このような場合も含めて、点が多角形の内部に入っているか判定するには、どうすればよいのでしょうか？

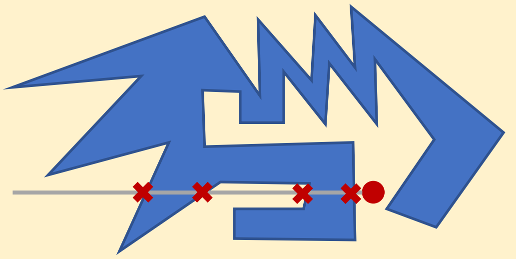
実は、以下のような単純な方法で判定できます。

点 (A, B) が多角形に含まれるかの判定

1. 点 (A, B) から左に向かって半直線を引きます。
2. この半直線が、多角形の辺と交わった回数を数えます。これが奇数回ならば点 (A, B) は多角形の内部に、偶数回なら多角形の外部にあります。



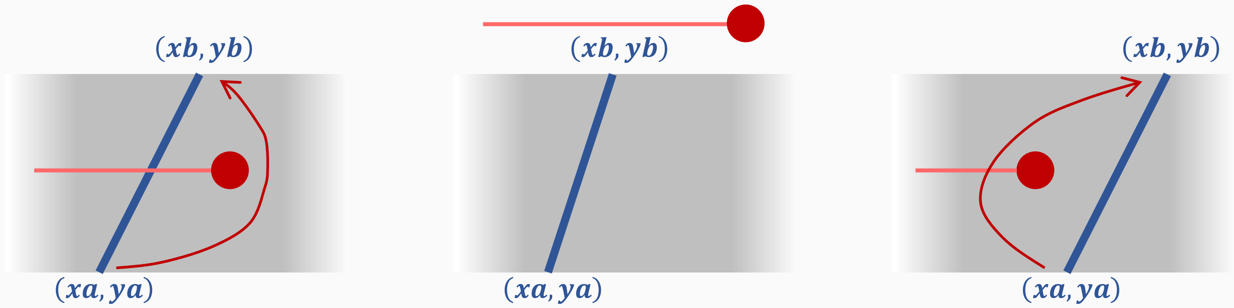
9 回交わるので **内部**



4 回交わるので **外部**

つまり、多角形の各辺に対して、点 (A, B) から左に向かって引いた半直線と交わるかどうか判定することになります。辺が (xa, ya) と (xb, yb) を結ぶ線分（ただし $ya < yb$）だとして、**基本的には**以下の条件を満たしたときに交わります。

1. $ya \leq B \leq yb$ である。
2. 点 (xa, ya)、点 (A, B)、点 (xb, yb) がこの順で反時計回りになっている。

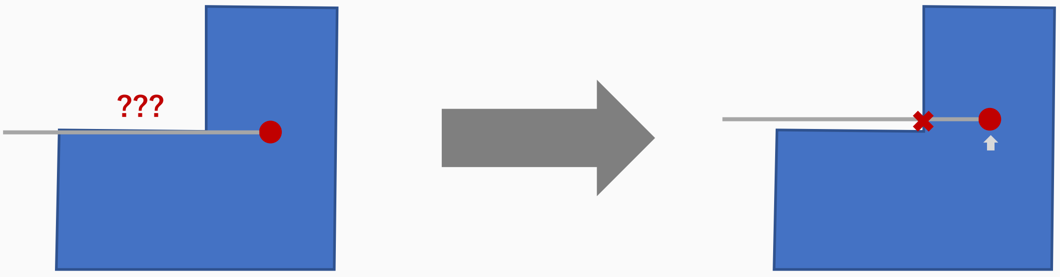


条件 1・2 を両方満たす

条件 1 を満たさない
（$ya \leq B \leq yb$ ではない）

条件 2 を満たさない
（時計回りになっている）

しかし、多角形の辺が $y = B$ で水平になる場合は例外で、この辺を見ただけでは本当に交わっているのか判断がつきません。このようなケースをなくすために、点をほんの少しだけ上にずらして考えます（そうしても答えは変わりません）。



すると、条件 1 が少しだけ変わり、以下のようになります。なぜなら「$ya \leq B + \epsilon \leq yb$（$\epsilon$ は無限に小さい数）」が条件になるからです。

1. $ya \leq B < yb$ である。
2. 点 (xa, ya)、点 (A, B)、点 (xb, yb) がこの順で反時計回りになっている。

条件 2 は外積を使って判定できます（→4.1.5 項）。多角形の辺のなかでこの条件を満たすものを数え、これが奇数個か偶数個かを判定することでこの問題が解けます。

この解法を C++ で実装すると、以下のようになります。

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

int N; long long X[100009], Y[100009], A, B;

int main() {
```

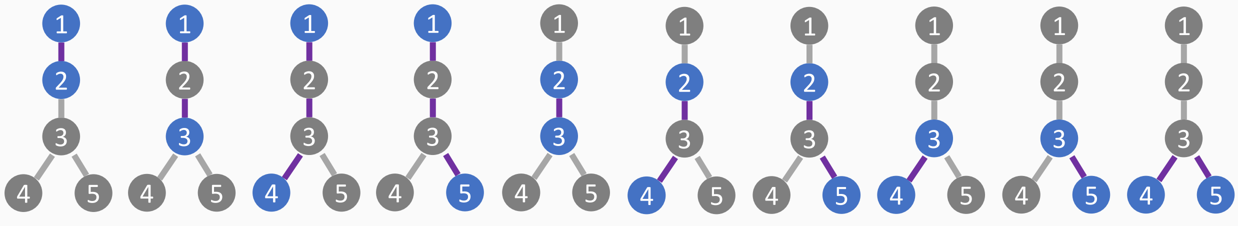
```
    // 入力
    cin >> N;
    for (int i = 0; i < N; i++) cin >> X[i] >> Y[i];
    cin >> A >> B;

    // 交差する回数を数える
    int cnt = 0;
    for (int i = 0; i < N; i++) {
        long long xa = X[i] - A, ya = Y[i] - B;
        long long xb = X[(i + 1) % N] - A, yb = Y[(i + 1) % N] - B;
        if (ya > yb) {
            swap(xa, xb);
            swap(ya, yb);
        }
        if (ya <= 0 && 0 < yb && xa * yb - xb * ya < 0) {
            cnt += 1;
        }
    }

    // 答えを出力
    if (cnt % 2 == 1) cout << "INSIDE" << endl;
    else cout << "OUTSIDE" << endl;

    return 0;
}
```

Python・JAVA・C のソースコードは、GitHub の chap6-21_25.md をご覧ください。

問題 25

この問題で、幅優先探索（→**4.5.7 項**）を使って最短経路を求めていると、計算量 $O(N^2)$ かかってしまいますが、「足された回数を考える」テクニック（→5.7 節）を巧みに使うと計算量 $O(N)$ で解くことができます。

まずは具体例といて、以下の 5 頂点の木を考えてみましょう。頂点のペアは 10 通りあり、それぞれ最短経路は以下のようになっています。

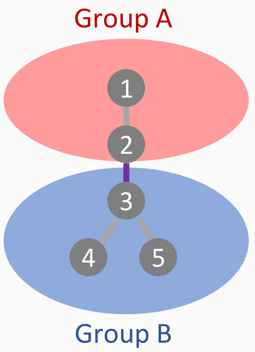


図．10 通りのペアそれぞれに対する最短経路

これを見ると、以下のことが分かります。

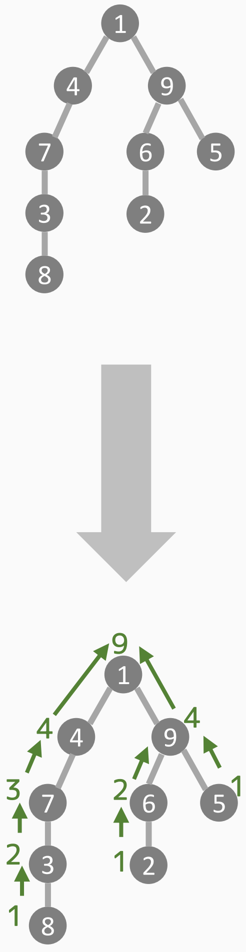
- 頂点 1-2 を結ぶ辺は、4 つの最短経路（1・2・3・4 番目）に含まれている
- 頂点 2-3 を結ぶ辺は、6 つの最短経路（2・3・4・5・6・7 番目）に含まれている
- 頂点 3-4 を結ぶ辺は、4 つの最短経路（3・6・8・10 番目）に含まれている
- 頂点 3-5 を結ぶ辺は、4 つの最短経路（4・7・9・10 番目）に含まれている

だから答えは 4 + 6 + 4 + 4 = 18、と求める方針が「足された回数を考える」テクニックを使ったやり方です。



では、特定の辺が何個の最短経路に含まれているかは、どうやって分かるのでしょうか？右図のように、辺は木を 2 つの部分に分けます。それぞれ A 頂点のグループ・$N-A$ 頂点のグループに分かれたとすると、辺は $A \times (N-A)$ 個の最短経路に含まれています。なぜなら、Group A の頂点から Group B の頂点までの最短経路に必ず含まれるからです。

したがって、それぞれの辺について、何頂点のグループと何頂点のグループに分かれるかが求まれば、この問題が解けます。



さて、この木を頂点 1 を持ってぶら下げることを考えてみましょう。すると、右図のような根っこのようなグラフの形ができます。このような木を「頂点 1 を根とする根付き木」といいます。（例えば、上司と部下の関係を表すグラフ（→**4.5.3 項**）が根付き木として表せます）

頂点 v の配下に（自分自身を含め） c_v 個の頂点があるとします。すると、頂点 v とその真上にある頂点を結ぶ辺は、グラフを c_v 頂点と $N-c_v$ 頂点のグループに分けることになります。

c_v は動的計画法を使って求められます。頂点 v の直下にある頂点を $s_1, s_2, \ldots, s_k$ とすると

$$c_v = c_{s_1} + c_{s_2} + \cdots + c_{s_k} + 1$$

と計算できるので、c_v を根付き木の下から順に求めていくと、計算量 $O(N)$ ですべての c_v が求まります。この問題の答えは、$c_v \times (N-c_v)$ $(v = 2, 3, \ldots, N)$ の総和になります。

この解法を C++ で実装すると、以下のようになります。なお、動的計画法の部分は、深さ優先探索（→**4.5.6 項**）と同様に再帰関数を用いると、比較的簡単に実装できます。

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

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

int dp[100009]; bool visited[100009];
void dfs(int pos) {
    visited[pos] = true;
    dp[pos] = 1;
    for (int i : G[pos]) {
        if (visited[i] == false) {
            dfs(i);
            dp[pos] += dp[i];
        }
    }
}

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

    // 深さ優先探索 (DFS) を使った動的計画法
    for (int i = 1; i <= N; i++) {
        visited[i] = false;
    }
    dfs(1);

    // 答えを計算して出力
    long long answer = 0;
    for (int i = 2; i <= N; i++) {
        answer += 1LL * dp[i] * (N - dp[i]);
    }
    cout << answer << endl;

    return 0;
}
```

Python・JAVA・C のソースコードは、GitHub の chap6-21_25.md をご覧ください。