

ABC215 分析レポート

AtCoder ABC215におけるあなたの提出と、上位3名 (jiangly, snuke, noimi) の提出データを比較・解析しました。

あなたはA, B, C, D, Eの5問を完答しており、非常に順調なパフォーマンスです。特にE問題（500点）までを30分以内に通しているのは素晴らしいスピードです。

上位勢との主な差分は、「標準ライブラリ/コンパイラ組み込み関数の活用度」、「計算量に対する定数倍の意識（メモリ管理）」、そして「典型的な難問（F問題以降）へのアプローチ」にあります。

以下に詳細な分析とアドバイスを記述します。

概要

- コンテスト: ABC215
- 対象ユーザー: kakira
- 比較対象上位者: jiangly (1位), snuke (2位), noimi (3位)

提出タイムラインとパフォーマンス分析

問題	あなたのタイム	上位平均タイム(目安)	差分・状態
:---	:---	:---	:---
A	01:10	00:50	Excellent (上位勢と遜色なし)
B	08:16	01:30	Good (少し実装に時間をかけたが許容範囲)
C	09:41	02:30	Good (Bの直後に通しており、実装速度は速い)
D	18:17	05:30	Good (実装方針は正しい)
E	29:35	10:00 - 12:00	Great (500点問題をこの時間で通すのは上位レベル)
F	未提出	16:00 - 22:00	Missed (上位勢はここを確実に通してくる)
G	未提出	22:00 - 26:00	Missed

全体的な考察

- 序盤～中盤 (A-E): 非常に安定しています。特にE問題までの立ち回りは素晴らしいです。
- 壁 (F): F問題（最大値の最小化・最小値の最大化系）で手が止まってしまったようです。ここは典型アルゴリズム（二分探索）の適用範囲なので、ここを突破できると青～黄色パフォーマンスが安定します。

問題別 詳細分析とアドバイス

A - Your First Judge

- あなたのコード: `if (S == "Hello,World!")` で分岐。
- 上位勢: 全員同じ方針です。
- アドバイス: 完璧です。改善点はありません。

B - log2(N)

- あなたの方針:

- `k` を0からループさせ、`now *= 2` しながら `now <= N` を満たす最大の `k` を探す。
- 上位勢の方針 (jiangly, snuke):
- **jiangly:** `std::__lg(n)` を使用。
- **snuke:** ビットシフト `(2ll<<k) <= n` を使用したループ。
- 比較・アドバイス:
- あなたのアプローチで正解ですが、競技プログラミングでは「2の何乗か」や「最上位ビットの位置」を求める操作は頻出です。
- GCCコンパイラ (AtCoder環境) では、`__lg(n)` や `63 - __builtin_clzll(n)` (Count Leading Zeros) を使うと $O(1)$ かつ実装量ほぼゼロで求まります。
- **Code Snippet (jiangly):**

```
std::cout << std::__lg(n) << "\n";
```

C - One More aab aba baa

- あなたの方針:

- `next_permutation` を `do-while` で回し、カウンター `cnt` が `N` (入力のK) になるまでループ。
- 上位勢の方針:
- `next_permutation` を単純に `K-1` 回呼び出す。
- 比較・アドバイス:
- あなたのコード :

```
int cnt = 0;
do {
    if (cnt == N) { cout << s << endl; return; }
    cnt++;
} while(next_permutation(all(s)));
```

- `next_permutation` は辞書順で「次の」順列に書き換える関数なので、中身を確認する必要はなく、単に回数分空回しすればOKです。

- **Code Snippet (jiangly):**

```
for (int i = 0; i < k; i++) {
    std::next_permutation(s.begin(), s.end());
}
std::cout << s << "\n";
```

- これにより、`do-while` の条件分岐や `return` の位置を気にする必要がなくなります。

D - Coprime 2

- あなたの方針:

- 入力 A_i ごとに $O(\sqrt{A_i})$ で素因数分解。
- 素因数を `set` に入れて重複排除。
- エラトステネスの篩の要領で、素因数の倍数を `flag` 配列で落とす。
- 上位勢の方針 (snuke):
- **前計算 (Linear Sieve / SPF):** 10^5 までの整数に対し、最小素因数 (SPF) を前計算。
- **高速素因数分解:** SPFテーブルを使うことで、各 A_i の素因数分解を $O(\log A_i)$ で行う。

- 比較・アドバイス:
- 今回の制約 $A_i \leq 10^5$ ではあなたの $O(N\sqrt{A})$ でも間に合いますが、 $A_i \leq 10^6$ や N が大きい場合、TLEする可能性があります。
- 改善点1(データ構造): `set<int> S` を使っていますが、`set` は挿入に $O(\log(\text{要素数}))$ かかります。ここは `vector` に入れて `sort` & `unique` するか、あるいは出現する素数は 10^5 以下なので `bool used_prime[100005]` のような配列で管理すると高速です。
- 改善点2(アルゴリズム): 頻出テクニックとして **OSA_k法 (SPF利用)** をライブラリ化しておくと、この手の問題で圧倒的に有利になります。

E - Chain Contestant

- あなたの方針:
- `vector<vector<vector<int>> dp(N + 1, vector<vector<int>>(1 << K, vector<int>(11)));`
- 3次元DP: `dp[i文字目][使用済み集合mask][最後に選んだコンテスト]`
- 上位勢の方針(jiangly, snake):
- 次元削減: `i文字目` の次元を削除し、`dp[mask][last]` の2次元配列を使い回す(in-place更新)。
- メモリ配置: `vector` の多重入れ子はメモリ確保とアクセスのオーバーヘッドが大きいです。上位勢は `int f[1 << 10][10];` のように固定長配列、あるいは1次元化した `vector` を使用しています。
- 比較・アドバイス:
- 解法自体は完璧です。
- 実装の改善: `vector<vector<vector<int>>` は、AtCoderの制限時間内では定数倍が重く、TLEの原因になりがちです(今回2秒制限かつ $N = 1000$ なので余裕でしたが)。
- 特にDPなどサイズが決まっている場合は、生配列 `int dp[1005][1024][11]` や `vector` の1次元化を検討してください。
- Code Snippet(jiangly):

```
// iのループ内で、dp配列を直接更新している
// 配列サイズは [1<<10][10] のみ
Z f[1 << 10][10];
for (int i = 0; i < n; i++) {
    int x = s[i] - 'A';
    // maskを降順に回すことでの直前の状態からの遷移を1つの配列で処理可能にするテクニックも使えるが
    // jianglyは単純に遷移させている(遷移元と遷移先が被らない工夫をしている)
    // ...
}
```

F - Dist Max 2(未提出)

- 問題概要: 2点間の距離 $\min(|x_i - x_j|, |y_i - y_j|)$ の最大値を求める。
- 上位勢の方針:
- 答えで二分探索: 「距離を K 以上にできるか?」という判定問題にする。
- 条件式: $|x_i - x_j| \geq K$ かつ $|y_i - y_j| \geq K$ となるペアが存在するか。
- 判定の実装:
- 点を x 座標でソートする。
- 各点 i について、 $x_j \leq x_i - K$ を満たす点 j の範囲(尺取り法や二分探索で求まる)を考える。
- その範囲内の点 j について、 $\max(y_j)$ と $\min(y_j)$ を保持しておけば、 $|y_i - y_j| \geq K$ の判定が $O(1)$ で可能。
- アドバイス:
- 「最大値の最小化」や「最小値の最大化」を見たら、条件反射で「答えで二分探索(Binary Search on Answer)」を疑ってください。
- この問題は典型的な「二分探索 + 平面走査(または尺取り法)」のパターンです。Eまで解ける実力があれば、このパターンを知っていれば解けたはずです。

G - Colorful Candies 2 (未提出)

- **問題概要:** K 個選んだときの色数の期待値。
- **上位勢の方針:**
- **期待値の線形性:** 「色 c が選ばれる確率」を全色について足し合わせる。
- 色 c が A_c 個あるとき、その色が「1個も選ばれない確率」を考える方が簡単。
- 選ばれない確率 = $\frac{\binom{N-A_c}{K}}{\binom{N}{K}}$
- よって、色 c が選ばれる確率 = $1 - \frac{\binom{N-A_c}{K}}{\binom{N}{K}}$
- 同じ個数 A_c を持つ色はまとめて計算することで高速化。
- **アドバイス:**
- 「種類の数」の期待値問題は、「各種類が含まれる確率の総和」に分解するのが鉄則です。

総合アドバイス

- 「**答えて二分探索**」の習得 (F問題対策)
 - F問題のような「最小値の最大化」は、ABCのF問題クラスで頻出です。
- **Action:** 過去問で「Binary Search on Answer」タグの問題を数問解いて、判定パートを $O(N)$ や $O(N \log N)$ で書く練習をしてください。
- **多次元vectorの回避 (E問題の最適化)**
 - `vector<vector<vector<int>>` は非常に遅いです。
 - **Action:** 競技プログラミングにおいては、多次元配列が必要な場合、可能な限り `int dp[100][100][100]` のような固定長配列（グローバル変数として宣言）を使うか、`atcoder::dsu` のように1次元 `vector` をインデックス計算でアクセスする癖をつけると、定数倍高速化で救われる場面が増えます。
- **数学的典型パターンのストック (G問題対策)**
 - 期待値問題が出たら「線形性」と「主客転倒（個数カウントではなく、各要素が寄与する確率を考える）」を即座に想起できるようにしましょう。

あなたの実装力はすでに高く、A～Eをスムーズに解く力があります。あとは「典型アルゴリズムの引き出し」をいくつか増やすだけで、黄色コーダーへの道が開けるはずです。