以下を通常のテキスト表示にしてください。
1. 堅牢性（エラーハンドリング・安全性）
入力値の検証:
関数（例: find_max, find_min）に渡される引数（arr, size）が、期待される有効な値であるかを確認していますか？
例: arr が NULL ポインタでないか？ size が 0 以上であるか？
もし無効な値が渡された場合、プログラムがクラッシュしたり、予期せぬ動作をしたりする可能性はありませんか？
境界条件の考慮:
配列の要素数が1つだけの場合、空の配列の場合など、極端なケースで正しく動作しますか？
エラー処理:
関数内で問題が発生した場合（例: 無効な入力があった場合）、どのように呼び出し元にエラーを伝えていますか？（戻り値、エラーコードなど）
2. 可読性・保守性
命名規則:
変数名、関数名が、その役割や目的を明確に表す、分かりやすい名前になっていますか？
例: n よりも array_count の方が分かりやすい、など。
命名規則（例: 小文字とアンダースコアで区切る snake_case）が一貫していますか？
コメント:
複雑な処理や、なぜそのように実装したのかが分かりにくい箇所に、適切なコメントが書かれていますか？
関数の目的、引数（入力）、戻り値（出力）、そしてその関数を使う上での注意点などが、コメントで説明されていますか？
マジックナンバー:
プログラム中に、意味が分からない数字（例: 6 など）が直接書かれていませんか？
そのような数字は、定数（#define や const）として名前をつけて定義し、その名前を使っていますか？
コードの構造:
一つの関数やブロックが長すぎず、一つの役割に集中していますか？（「単一責任の原則」）
コードが読みやすいように、適切な場所で改行や空白行が使われていますか？
3. 効率性
アルゴリズムの選択:
現在の処理方法よりも、もっと効率的に同じ結果を得られる方法はありませんか？
例: ループの回数を減らす、計算をまとめるなど。
無駄な処理:
同じ計算が何度も繰り返されていませんか？
必要のないメモリの確保やコピーが発生していませんか？
4. コーディングスタイル
インデント:
コードの階層構造（ブロックの深さ）に合わせて、一貫したインデント（字下げ）が使われていますか？（例: スペース2つ、スペース4つ、タブなど）
括弧の位置:
{ や } といった括弧の位置が、プロジェクトやチームのルール（または一般的な慣習）に沿って一貫していますか？
スペースの利用:
演算子（+, =, > など）の周りや、カンマの後ろなどに、適切なスペースが使われて

いて読みやすいですか？
ヘッダーコメント:
ファイルの先頭に、ファイルの内容や著作権情報などを説明するコメントがありますか？（これはプロジェクトによって有無が異なります）