COURSE PROJECT

Design, implement, and evaluate an advanced algorithm of your choice, demonstrating deep understanding of algorithmic design, implementation, and performance evaluation.

1. Choose Your Algorithm

Algorithm Selection:

- **Option 1**: Pick an advanced algorithm from the *Introduction to Algorithms* (CLRS) textbook, such as the Red-Black tree.
- **Option 2**: Choose an algorithm from papers presented in recent research conferences (e.g., STOC, FOCS, NeurIPS, ICML, SIGMOD). You can search for algorithmic improvements or novel algorithms that address specific computational challenges.
- **Tip**: Select an algorithm you're genuinely interested in or one that aligns with your academic or career goals.

2. Implement the Algorithm

Programming Language: Use any language you're comfortable with (e.g., Python, Java, C++).

Code Quality:

- **Structure**: Organize code into functions or classes.
- **Naming**: Use clear, descriptive variable and function names.
- **Comments**: Add comments explaining each section, especially where the algorithm's logic is implemented.

Implementation Steps:

1. **Understand the Algorithm**: Study the algorithm's design in detail, including any pseudocode provided in the textbook or research paper.
2. **Translate to Code**: Write the algorithm step-by-step, testing as you go.
3. **Modularize**: Break down the code into functions/modules for easier testing and debugging.
4. **Optimize for Performance**: Depending on the language, some built-in data structures (like set or map in C++) can improve performance.

3. Write the Technical Report

Technical Report Sections:

3.1 Algorithm Introduction

- **Description**: Explain the algorithm's purpose and how it works. Include pseudocode or a flowchart for clarity if necessary.
- **Background**: Discuss any theoretical background, like graph theory, probability, or linear algebra.
- **Complexity**: Analyze the time and space complexity and provide justifications.
- **Comparisons**: Briefly compare your chosen algorithm with others that solve similar problems, highlighting strengths and weaknesses.

3.2 Implementation Details

- **Design Choices**: Describe why you chose certain data structures (e.g., heaps, trees).
- **Challenges**: Mention any technical issues faced and how you solved them.
- **Architecture**: Provide an overview of your code structure, with diagrams if necessary.

3.4 Benchmarking Methodology

- **Workload**: Define typical input cases you'll use for testing.
- **Dataset**: Describe the dataset(s) used. If using standard datasets, provide details. If synthetic, explain generation.
- **Setup**: Describe your hardware/software environment (e.g., CPU, memory, programming language).
- **Results**: Present your benchmark results, with clear graphs or tables to interpret findings.
  - **Time Complexity**: Measure how the algorithm's runtime changes with input size.
  - **Space Complexity**: Evaluate the memory usage, either theoretically or by monitoring during tests.

    - Compare your implementation’s performance to other implementations, if possible.
    - Include graphs showing runtime, memory usage, or both against input sizes.

4. Submission Requirements

- **Technical Report**: Export your document as a PDF. Using ACM or IEEE style is recommended.
- **Source Code**: Submit all files, following all requirements above.
- **Benchmark Data**: Include raw results if necessary, or processed in Excel/CSV for easier grading.

5. Evaluation Criteria

- Check each criterion as you complete the project to ensure full points:
    - **Algorithmic Understanding**: Ensure clear explanations in the report.
    - **Implementation Correctness**: Test your code thoroughly for bugs.
    - **Performance Analysis Depth**: Analyze multiple test cases.
    - **Code Quality and Documentation**: Make sure comments and structure are clear.
    - **Presentation Clarity**: Aim for a professional, well-organized report.

6. Academic Integrity

- Cite the original algorithm source, any reference implementations, and any online resources you used.
- Write the code independently. Even if you consult examples, ensure your implementation reflects your understanding.