**Department of Computing**

**CS250: Data Structures and Algorithms**

**BSDS- 1**

**Fall 2024**

**Name: Sana Khan Khitran**

**CMS: 464597**

Assignment 1: Linked Lists (Singly and Circular)

**CLO 1: Describe the fundamentals of data structures and algorithms (C-2 Understand)**

**CLO 2: Solve a given real time problem by applying appropriate data structure and algorithm (C-3 Apply).**

**Github Link:** [**https://github.com/sanakhitran22/SANA-DSA-ASSIGNMENT1**](https://github.com/sanakhitran22/SANA-DSA-ASSIGNMENT1)

# **Introduction:**

**This assignment focuses on demonstrating your understanding of circular linked lists and singly linked lists by solving two real-world problems. Explicitly mention any assumption you make, without invalidating the questions.**

## **Problem 1: Simple Process Scheduling Algorithm**

**Design and Implement a CPU Process Scheduling Algorithm using a Linked List. Processes will be represented as nodes in the list, and the scheduling algorithm should cycle through the processes, assigning CPU time to each process until all processes have completed their execution.**

### **Task:**

* **Each process should have the following attributes: process\_id, execution\_time, and remaining\_time.**
* **The scheduler assigns a fixed amount of CPU time to each process in each cycle. After the time is assigned, the process's remaining\_time will be reduced by that amount.**
* **If a process completes its execution (i.e., remaining\_time becomes 0), it should be removed from the circular linked list.**
* **The system should display the state of processes after each cycle, indicating which process is running and its remaining time.**
* **Show the state of the system after each cycle.**

### **Optional Task:**

**Your system should simulate a scenario where a new process can arrive at any time. For example, the program should handle an event that inserts a new process into the list while the scheduler is running.**

### **Output Example:**

Initial Processes: [(P1, 10), (P2, 5), (P3, 8)]

CPU Time per Process per Cycle: 3

Cycle 1: P1 (Remaining: 7), P2 (Remaining: 2), P3 (Remaining: 5)

Cycle 2: P1 (Remaining: 4), P2 (Completes), P3 (Remaining: 2)

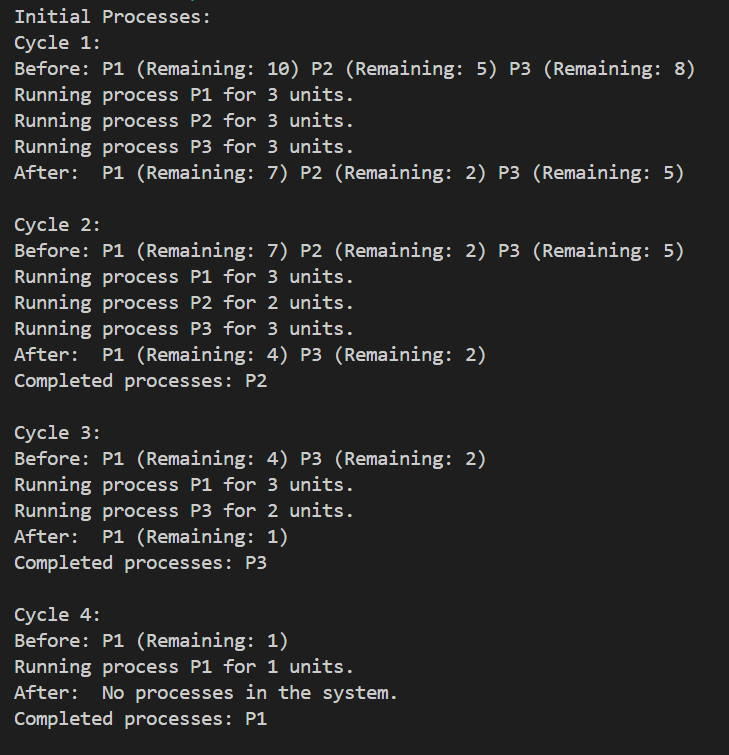
New process arrives: P4 (Remaining: 9)

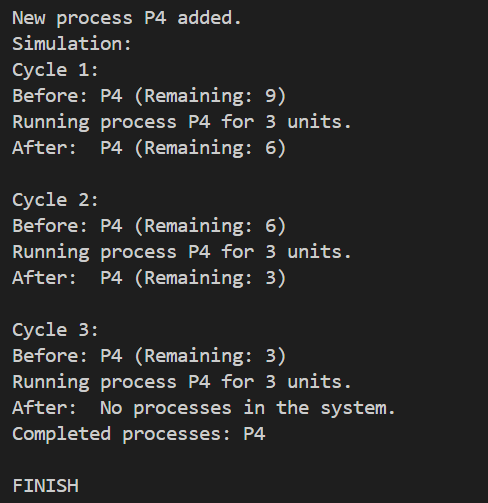
Cycle 3: P1 (Remaining: 1), P3 (Completes), P4 (Remaining: 6)

## **Solution:**

|  |
| --- |
| //ASSIGNMENT 1 DSA QUESTION 1  #include <iostream>  #include <string>  using namespace std;  class Process {  public:      int process\_id; // Unique ID of the process      int execution\_time; // Total time required for the process to complete      int remaining\_time; // Time left for the process to complete      Process\* next; // Pointer to the next process in the circular list      // Constructor      Process(int id, int time) : process\_id(id), execution\_time(time), remaining\_time(time), next(nullptr) {}  };  // Class to manage the process scheduling and simulate the CPU execution cycles  class ProcessScheduler {  private:      Process\* head;      const int cpu\_time\_per\_cycle = 3; // CPU time assigned to each process per cycle  public:      // Constructor      ProcessScheduler() : head(nullptr) {}      // Function to add a new process to the circular list      void addProcess(int process\_id, int execution\_time) {          Process\* newProcess = new Process(process\_id, execution\_time);  // Create a new process          if (!head) {  // If the list is empty, make the new process the head and point it to itself              head = newProcess;              newProcess->next = head;          } else {  // Otherwise, insert the new process at the end of the circular list              Process\* current = head;              while (current->next != head) {                  current = current->next;              }              current->next = newProcess;              newProcess->next = head; // Make the list circular          }      }      // Function to simulate a single CPU cycle where time is assigned to processes      void runCycle(int& cycleCount) {          if (!head) {  // If there are no processes, exit              return;          }          cout << "Cycle " << cycleCount << ":" << endl;          cout << "Before: ";          displayState();  // Display the state of all processes before running this cycle          Process\* current = head;       // Pointer to track the current process in the list          Process\* prev = nullptr;       // Pointer to track the previous process (for deletion)          Process\* completedHead = nullptr;  // Temporary list for completed processes          // Loop through the circular list of processes and assign CPU time          do {              // Display which process is running and for how much time              cout << "Running process P" << current->process\_id << " for " << min(cpu\_time\_per\_cycle, current->remaining\_time) << " units." << endl;              // Reduce the remaining time of the current process              current->remaining\_time -= cpu\_time\_per\_cycle;              // Check if the process has completed its execution              if (current->remaining\_time <= 0) {                  // Add completed process to a temporary list                  Process\* completed = new Process(current->process\_id, 0);                  completed->next = completedHead;                  completedHead = completed;                  // Remove the process from the circular linked list                  if (current->next == current) { // If it's the only process in the list                      delete current;                      head = nullptr; // Set head to null since there are no more processes                  } else {                      if (prev) {                          prev->next = current->next;  // Skip over the current process                      } else {                          head = current->next;        // Move head to the next process                      }                      Process\* temp = current;         // Delete the current process                      current = current->next;                      delete temp;                      continue;  // Continue to the next process                  }              } else {                  prev = current;  // Update prev to the current process              }              if (head == nullptr) break;  // If no processes remain, stop the loop              current = current->next;     // Move to the next process in the list          } while (current != head);          cout << "After:  ";          displayState();  // Display the state of processes after the cycle            displayCompleted(completedHead);  // Display completed processes          freeCompletedList(completedHead); // Free the memory for the completed process list            cycleCount++;  // Increment the cycle counter          cout << endl;      }      // Function to display the state of all processes in the system      void displayState() {          if (!head) {  // If there are no processes              cout << "No processes in the system." << endl;              return;          }          Process\* current = head;          do {              cout << "P" << current->process\_id << " (Remaining: " << current->remaining\_time << ") ";  // Display remaining time of each process              current = current->next;          } while (current != head);          cout << endl;      }      // Function to display the list of completed processes after a cycle      void displayCompleted(Process\* completed) {          if (!completed) {  // If no processes have completed, return              return;          }          cout << "Completed processes: ";          Process\* current = completed;          while (current) {  // Loop through the list of completed processes and display their IDs               cout << "P" << current->process\_id;              if (current->next)  cout << ", ";              current = current->next;          }          cout << endl;      }      // Function to free the memory of the completed process list      void freeCompletedList(Process\* completedHead) {          while (completedHead) {              Process\* temp = completedHead;  // Free each completed process node              completedHead = completedHead->next;              delete temp;          }      }      // Function to simulate the entire scheduling process, running cycles until all processes are completed      void simulate() {          int cycleCount = 1;  // Initialize cycle counter          while (head) {       // Keep running cycles as long as there are processes              runCycle(cycleCount);          }      }      // Destructor to free the memory of any remaining processes in the list      ~ProcessScheduler() {          if (!head) return;          Process\* current = head->next;          while (current != head) {              Process\* temp = current;              current = current->next;              delete temp;          }          delete head;      }  };  int main() {      ProcessScheduler scheduler;        // Create initial processes      scheduler.addProcess(1, 10);  // Process P1 with 10 units of execution time      scheduler.addProcess(2, 5);   // Process P2 with 5 units of execution time      scheduler.addProcess(3, 8);   // Process P3 with 8 units of execution time      // Simulate the process scheduling      cout << "Initial Processes:" << endl;      scheduler.simulate();      // Adding a new process P4 after the initial simulation      scheduler.addProcess(4, 9);   // Process P4 with 9 units of execution time      cout << "New process P4 added." << endl;      cout << "Simulation:" << endl;      scheduler.simulate();      return 0;  } |

**CODE:**





## **Problem 2: Very large Prime Number Calculation**

**When it comes to storing very large numbers, one possible way is to use a linked list. For example, if you want to store a 1024-bit random number, you can store 32-bit integers in each node of the linked list. The linked list as a whole, then contains the 1024-bit number. Or you can take the digits of the big number, which can be fitted inside each node, and using a linked list can represent the number (e.g. to represent 1234567890123456789, you may keep 12345 in node 1, 67890 in node 2 and so on so forth). This was the easy part. You are required to evaluate and determine, probabilistically or deterministically, whether a 1024-bit random number is prime or not. Since primitive data types cannot store a 1024-bit number, the number will be split across multiple linked list nodes. You can use any methodology to split the number and store it in the linked list, however pay special attention to memory usage and primality test.**

### **Task**

* **Store a 1024-bit (309 digits) random number in a linked list.**
* **Each node of the linked list can contain up to 64 bit numbers.**
* **Implement any method to check for primality of this number.**
* **Make sure your chosen method, operates across multiple nodes in the linked list (No partial primality tests).**

### **Output Example:**

Input: A 1024-bit number represented as "1234567890123456789... "

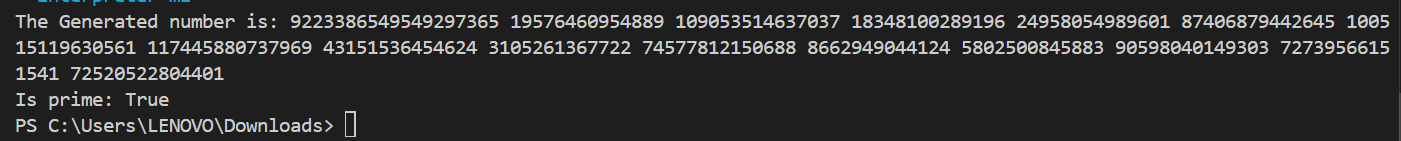
Smaller primes: [2, 3, 5, 7, 11, ...]

Output: True (if prime) / False (if not prime)

**CODE:**

|  |
| --- |
| #include <iostream>  #include <cstdlib>  #include <ctime>  #include <cstring>  #include <climits>  #include <cstdint>  #include <random>  using namespace std;  // Constants defining node and bit sizes  const int bits\_per\_node = 64; // Each node can hold 64 bits (uint64\_t)  const int nodes\_required = 1024 / bits\_per\_node; // Total nodes required to represent a 1024-bit number  const uint64\_t MAX\_UINT64 = UINT64\_MAX; // Maximum value for a 64-bit unsigned integer  // Node structure representing each part of a large number  struct Node {      uint64\_t value; // Value stored in the node      Node\* next; // Pointer to the next node in the linked list      Node(uint64\_t val = 0) : value(val), next(nullptr) {} // Constructor initializing node value  };  class LargeNumber {  private:      Node\* head;      // function to perform a deep copy of another LargeNumber      void deepCopy(const LargeNumber& other) {          Node\* otherCurrent = other.head; // Pointer to traverse the other LargeNumber          Node\*\* current = &head; // Pointer to insert new nodes into this LargeNumber          while (otherCurrent) {              \*current = new Node(otherCurrent->value); // Create a new node and copy the value              current = &((\*current)->next); // Move to the next pointer              otherCurrent = otherCurrent->next; // Move to the next node in the other LargeNumber          }      }  public:      // Default constructor      LargeNumber() : head(nullptr) {}      // Constructor to initialize with a single uint64\_t value      LargeNumber(uint64\_t val) : head(nullptr) {          head = new Node(val); // Create a new node for the value      }      // Copy constructor      LargeNumber(const LargeNumber& other) : head(nullptr) {          deepCopy(other); // Call deepCopy to copy nodes      }      // Assignment operator      LargeNumber& operator=(const LargeNumber& other) {          if (this != &other) { // Self-assignment check              // Deallocate existing nodes              while (head) {                  Node\* temp = head;                  head = head->next;                  delete temp; // Delete the old node              }              deepCopy(other); // Deep copy the new nodes          }          return \*this; // Return current object      }      // Generate a random 1024-bit number      void generateRandom() {          srand(time(nullptr)); // Seed random number generator          for (int i = 0; i < nodes\_required; ++i) {              // Generate a random 64-bit number              uint64\_t randomValue = ((uint64\_t)rand() << 32) | rand();              if (i == nodes\_required - 1) {                  randomValue |= (1ULL << 63); // Ensure the last node is set to 1024 bits              }              Node\* newNode = new Node(randomValue); // Create a new node with the random value              newNode->next = head; // Insert at the beginning of the linked list              head = newNode; // Update head pointer          }      }      // Check if the number is even      bool isEven() const {          return head && !(head->value & 1); // Check if the least significant bit is 0      }      // Check if the number is zero      bool isZero() const {          Node\* current = head; // Pointer to traverse the linked list          while (current) {              if (current->value != 0) return false; // Return false if any value is not zero              current = current->next; // Move to the next node          }          return true; // All values are zero      }      // Check if the number is one      bool isOne() const {          if (!head || head->value != 1) return false; // Check if the head is 1          Node\* current = head->next; // Pointer to traverse remaining nodes          while (current) {              if (current->value != 0) return false; // Return false if any subsequent value is not zero              current = current->next; // Move to the next node          }          return true; // The number is one      }      // Divide the large number by two      void divideByTwo() {          uint64\_t carry = 0; // Carry to keep track of division          Node\* current = head; // Pointer to traverse the linked list          while (current) {              uint64\_t newValue = (current->value >> 1) | (carry << 63); // Shift right and include carry              carry = current->value & 1; // Update carry with the least significant bit              current->value = newValue; // Update current node value              current = current->next; // Move to the next node          }          // Remove leading zero node          if (head && head->value == 0) {              Node\* temp = head;              head = head->next; // Update head to the next node              delete temp; // Delete the old head          }      }      // Subtract another LargeNumber from this one      void subtract(const LargeNumber& other) {          Node\* thisCurrent = head; // Pointer for this number          Node\* otherCurrent = other.head; // Pointer for other number          uint64\_t borrow = 0; // Borrow for subtraction          // Loop through both numbers          while (thisCurrent && otherCurrent) {              if (thisCurrent->value >= otherCurrent->value + borrow) {                  thisCurrent->value -= otherCurrent->value + borrow; // Direct subtraction                  borrow = 0; // No borrow              } else {                  // Need to borrow                  thisCurrent->value = (MAX\_UINT64 - otherCurrent->value - borrow + 1 + thisCurrent->value);                  borrow = 1; // Set borrow for next iteration              }              thisCurrent = thisCurrent->next; // Move to the next node              otherCurrent = otherCurrent->next; // Move to the next node          }          // Handle remaining nodes in this number          while (thisCurrent && borrow) {              if (thisCurrent->value >= borrow) {                  thisCurrent->value -= borrow; // Subtract borrow                  borrow = 0; // Clear borrow              } else {                  // Need to wrap around                  thisCurrent->value = MAX\_UINT64 - borrow + 1;                  borrow = 1; // Set borrow for next node              }              thisCurrent = thisCurrent->next; // Move to the next node          }          // Remove leading zeros          while (head && head->next && head->value == 0) {              Node\* temp = head;              head = head->next; // Update head              delete temp; // Delete old head          }      }      // Calculate modulus of this number with another LargeNumber      LargeNumber mod(const LargeNumber& m) const {          LargeNumber result = \*this; // Copy this number          while (result >= m) { // While result is greater than or equal to modulus              result.subtract(m); // Subtract modulus from result          }          return result; // Return the result      }      // Perform modular multiplication of this number with another LargeNumber      LargeNumber modMul(const LargeNumber& other, const LargeNumber& m) const {          LargeNumber result; // Result of multiplication          LargeNumber temp = \*this; // Temporary variable for calculations          Node\* otherCurrent = other.head; // Pointer for the other LargeNumber          while (otherCurrent) {              for (int i = 0; i < 64; ++i) { // Loop through bits                  if (otherCurrent->value & (1ULL << i)) {                      result = (result + temp).mod(m); // Add temp to result and take modulus                  }                  temp = (temp + temp).mod(m); // Double temp and take modulus              }              otherCurrent = otherCurrent->next; // Move to the next node          }          return result; // Return the result      }      // Perform modular exponentiation      LargeNumber modPow(const LargeNumber& exponent, const LargeNumber& m) const {          if (m.isOne()) return LargeNumber(0); // If modulus is one, return zero          LargeNumber result(1); // Initialize result to 1          LargeNumber base = this->mod(m); // Reduce base with modulus          LargeNumber exp = exponent; // Copy exponent          while (!exp.isZero()) { // While exponent is not zero              if (!exp.isEven()) {                  result = result.modMul(base, m); // Multiply result with base and take modulus              }              base = base.modMul(base, m); // Square base and take modulus              exp.divideByTwo(); // Divide exponent by two          }          return result;      }      // Comparison operator for equality      bool operator==(const LargeNumber& other) const {          Node\* thisCurrent = head; // Pointer for this number          Node\* otherCurrent = other.head; // Pointer for other number          // Compare node values          while (thisCurrent && otherCurrent) {              if (thisCurrent->value != otherCurrent->value) return false; // If values differ, return false              thisCurrent = thisCurrent->next; // Move to the next node              otherCurrent = otherCurrent->next; // Move to the next node          }          return !thisCurrent && !otherCurrent; // Return true if both lists are fully traversed      }      // Comparison operator for greater than or equal to      bool operator>=(const LargeNumber& other) const {          Node\* thisCurrent = head; // Pointer for this number          Node\* otherCurrent = other.head; // Pointer for other number          // Skip leading zeros          while (thisCurrent && thisCurrent->value == 0) thisCurrent = thisCurrent->next;          while (otherCurrent && otherCurrent->value == 0) otherCurrent = otherCurrent->next;          // Compare number of significant nodes          int thisNodes = 0, otherNodes = 0; // Count significant nodes          Node\* temp = thisCurrent;          while (temp) {              thisNodes++; // Count nodes in this number              temp = temp->next;          }          temp = otherCurrent;          while (temp) {              otherNodes++; // Count nodes in other number              temp = temp->next;          }          if (thisNodes > otherNodes) return true; // This number has more significant nodes          if (thisNodes < otherNodes) return false; // Other number has more significant nodes          // Compare node values from most significant to least          while (thisCurrent && otherCurrent) {              if (thisCurrent->value > otherCurrent->value) return true; // This number is greater              if (thisCurrent->value < otherCurrent->value) return false; // Other number is greater              thisCurrent = thisCurrent->next; // Move to the next node              otherCurrent = otherCurrent->next; // Move to the next node          }          return true; // Numbers are equal      }      // Addition operator      LargeNumber operator+(const LargeNumber& other) const {          LargeNumber result; // Result of addition          Node\*\* resultCurrent = &result.head; // Pointer to insert nodes into result          Node\* thisCurrent = head; // Pointer for this number          Node\* otherCurrent = other.head; // Pointer for other number          uint64\_t carry = 0; // Carry for addition          // Loop through both numbers          while (thisCurrent || otherCurrent || carry) {              uint64\_t sum = carry; // Start with carry              if (thisCurrent) {                  sum += thisCurrent->value; // Add value from this number                  thisCurrent = thisCurrent->next; // Move to the next node              }              if (otherCurrent) {                  sum += otherCurrent->value; // Add value from the other number                  otherCurrent = otherCurrent->next; // Move to the next node              }              \*resultCurrent = new Node(sum & MAX\_UINT64); // Create a new node for the result              resultCurrent = &((\*resultCurrent)->next); // Move to the next pointer              carry = sum >> 64; // Update carry for next iteration          }          return result;      }      // Subtraction operator      LargeNumber operator-(const LargeNumber& other) const {          LargeNumber result = \*this; // Copy this number          result.subtract(other); // Subtract other from the copy          return result; // Return the result      }      // Check if the number is prime using Miller-Rabin primality test      bool isPrime() {          if (isEven()) return false; // Exclude even numbers            LargeNumber two(2);          LargeNumber one(1);          LargeNumber d = \*this - one; // d = n - 1          int s = 0; // Count of factors of 2 in d          while (d.isEven()) {              d.divideByTwo(); // Divide d by 2              s++; // Increase count of factors of 2          }          for (int i = 0; i < 5; i++) { // 5 iterations for Miller-Rabin              LargeNumber a;              a.generateRandom(); // Generate random number              a = a.mod(\*this); // Reduce it modulo n              if (a.isZero() || a.isOne()) continue; // Skip if a is 0 or 1              LargeNumber x = a.modPow(d, \*this); // x = a^d mod n              if (x.isOne() || x == \*this - one) continue; // x is either 1 or n - 1              bool composite = true; // Assume number is composite              for (int j = 0; j < s - 1; j++) {                  x = x.modMul(x, \*this); // Square x and take modulus                  if (x == \*this - one) { // Check if x is n - 1                      composite = false; // Found a witness, it's not composite                      break;                  }              }              if (composite) return false; // If still composite, n is not prime          }          return true; // n is probably prime      }      // Display the number      void display() const {          Node\* current = head; // Pointer to traverse the linked list          while (current) {              cout << current->value << " "; // Print each node value              current = current->next; // Move to the next node          }          cout << endl;      }      // Destructor to clean up memory      ~LargeNumber() {          while (head) {              Node\* temp = head; // Pointer to the current head              head = head->next; // Move head to the next node              delete temp; // Delete the old head          }      }  };  //MAIN FUNCTION  int main() {      LargeNumber largeNum; // Create a LargeNumber object      largeNum.generateRandom(); // Generate a random large number      cout << "The Generated number is: ";      largeNum.display();      bool isPrime = largeNum.isPrime(); // Check if the number is prime      cout << "Is prime: " << (isPrime ? "True" : "False") << endl;      return 0;  } |

**OUTPUT:**



## Submission:

Submit a .zip or .rar file on LMS containing:

* Source code for both problems (Problem1.cpp and Problem2.cpp or equivalent).
* A short README containing:
  + Approach,
  + Assumptions
  + Publicly accessible GitHub link
  + Challenges faced
  + Screenshots of your output for sample inputs.

Grading Rubric:  
Total Marks = 26

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Criteria | 0 | 1 | 2 | 3 | 4 |
| Understanding of Data Structures (CLO1) | No submission/ Plagiarism | Very limited or incorrect understanding of circular or singly linked lists. | Basic understanding, but with significant errors or missing concepts in linked list usage. | Demonstrates good understanding with some issues in linked list structure or node handling. | Demonstrates strong understanding of circular and singly linked lists; effectively explains node management and operations. |
| Correctness of Scheduling Algorithm (CLO2) | No submission/ Plagiarism | Algorithm fails to work correctly, or dynamic insertion is not properly handled. | Algorithm implemented but contains significant errors . | Scheduling works with minor inefficiencies or fails to handle some edge cases during dynamic insertion. | Scheduling algorithm works correctly and handles dynamic process insertion seamlessly; handles all edge cases |
| Correctness of Prime Evaluation (CLO2) | No submission/ Plagiarism | Prime checking is incomplete or does not work for large numbers; division method fails. | Basic primality check with significant issues | Prime check works but may miss some edge cases; division across nodes is implemented but inefficient or with minor bugs. | Primality test correctly identifies primes and non-primes using linked lists; division method across nodes works efficiently. |
| Efficient Use of Data Structures (CLO1) | No submission/ Plagiarism | Linked list structure is not used effectively, leading to high inefficiency or memory wastage. | Basic use of linked lists but poor memory management or significant redundancies. | Linked lists are used effectively but may contain some inefficiencies | Efficient use of circular and singly linked lists with minimal redundancy; properly manages memory and dynamic processes. |
| Code Structure and Documentation | No submission/ Plagiarism | Code is poorly written, difficult to understand, or lacks any documentation. | Code is difficult to follow, poorly organized, or with minimal documentation. | Code is readable but lacks some structure or documentation. | Code is well-organized, readable, and well-documented; appropriate use of functions and modularity. |
| Version Control | No submission/ Plagiarism | Inconsistent or minimal use of version control | Basic use of version control with some structure, but lacking in meaningful commit messages or regular usage. | Regular use of version control with meaningful commit messages but could improve in frequency or structuring of commits. | Effective use of version control with regular, well-structured commits, meaningful commit messages, and appropriate branching/tagging practices. |
| Optional Task | Optional Task in problem 1 was not attempted | Optional Task in problem 1 was attempted but not completed | Optional Task in problem 1 was completed |  |  |