CodeSignal Assessment - Complete Practice Exam

Time Limit: 90 minutes total Language: Python only

Overview

This practice exam simulates the actual CodeSignal assessment. You'll implement a task management system across 4 progressive levels:

- **Level 1** (~20 min): Basic CRUD operations
- **Level 2** (~25 min): Dependencies and priorities
- **Level 3** (~30 min): Users, assignments, and deadlines
- **Level 4** (~40 min): Projects, templates, and time tracking

Strategy:

- Focus on passing tests, not code quality
- Run tests early and often - they define the true requirements
- Use simple data structures - no need for optimization
- Progress through all levels quickly rather than perfecting one level

---

Level 1: Basic Task Management

Task: Implement basic CRUD operations for tasks

```
# Task Management System - Level 1
# Time Limit: ~20 minutes for this level

from abc import ABC, abstractmethod
from typing import Dict, List, Optional
from dataclasses import dataclass
from enum import Enum

class TaskStatus(Enum):
    """Enumeration for task status values."""
    TODO = "todo"
    IN_PROGRESS = "in_progress"
    DONE = "done"

@dataclass
class Task:
    """Represents a single task in the system."""
    id: int
```

```
    title: str
    description: str
    status: TaskStatus

class TaskManagerInterface(ABC):
    """Interface for the task management system.

    This defines the contract that your TaskManager class must implement.
    Copy these method signatures to your solution and implement the bodies.
    """

    @abstractmethod
    def create_task(self, title: str, description: str) -> int:
        """Create a new task with the given title and description.

        The task should start with status TaskStatus.TODO.
        Task IDs should be unique integers starting from 1.

        Args:
            title: The task title (non-empty string)
            description: The task description (can be empty string)

        Returns:
            The unique ID of the created task

        Raises:
            ValueError: If title is empty or None
        """
        pass

    @abstractmethod
    def get_task(self, task_id: int) -> Optional[Task]:
        """Retrieve a task by its ID.

        Args:
            task_id: The ID of the task to retrieve

        Returns:
            The Task object if found, None otherwise
        """
        pass

    @abstractmethod
    def get_all_tasks(self) -> List[Task]:
        """Get all tasks in the system.

        Returns:
            List of all Task objects, ordered by creation time (oldest first)
        """
        pass
```

```
    @abstractmethod
    def update_task_status(self, task_id: int, status: TaskStatus) -> bool:
        """Update the status of an existing task.

        Args:
            task_id: The ID of the task to update
            status: The new status for the task

        Returns:
            True if the task was found and updated, False otherwise
        """
        pass

    @abstractmethod
    def delete_task(self, task_id: int) -> bool:
        """Delete a task from the system.

        Args:
            task_id: The ID of the task to delete

        Returns:
            True if the task was found and deleted, False otherwise
        """
        pass

    @abstractmethod
    def get_tasks_by_status(self, status: TaskStatus) -> List[Task]:
        """Get all tasks with a specific status.

        Args:
            status: The status to filter by

        Returns:
            List of Task objects with the specified status, ordered by creation time
        """
        pass

# ============================================================================
# IMPLEMENT YOUR SOLUTION BELOW
# ============================================================================

class TaskManager(TaskManagerInterface):
    """Your implementation of the TaskManager.

    Implement all methods from TaskManagerInterface.
```

```
    Remember: focus on passing tests, not code quality!
    """

    def __init__(self):
        # Initialize your data structures here
        pass

    def create_task(self, title: str, description: str) -> int:
        # TODO: Implement this method
        pass

    def get_task(self, task_id: int) -> Optional[Task]:
        # TODO: Implement this method
        pass

    def get_all_tasks(self) -> List[Task]:
        # TODO: Implement this method
        pass

    def update_task_status(self, task_id: int, status: TaskStatus) -> bool:
        # TODO: Implement this method
        pass

    def delete_task(self, task_id: int) -> bool:
        # TODO: Implement this method
        pass

    def get_tasks_by_status(self, status: TaskStatus) -> List[Task]:
        # TODO: Implement this method
        pass

#
==========================================================================
=
# TEST CASES - DO NOT MODIFY
#
==========================================================================
=

def test_level_1():
    """Test cases for Level 1. All must pass to proceed to Level 2."""

    # Test 1: Basic task creation
    manager = TaskManager()
    task_id = manager.create_task("Buy groceries", "Milk, eggs, bread")
    assert task_id == 1, f"Expected task ID 1, got {task_id}"

    # Test 2: Task retrieval
    task = manager.get_task(1)
    assert task is not None, "Task should exist"
```

```
    assert task.id == 1, f"Expected task ID 1, got {task.id}"
    assert task.title == "Buy groceries", f"Expected title 'Buy groceries', got '{task.title}'"
    assert task.description == "Milk, eggs, bread", f"Expected description 'Milk, eggs, bread', got
'{task.description}'"
    assert task.status == TaskStatus.TODO, f"Expected status TODO, got {task.status}"

    # Test 3: Multiple task creation with incrementing IDs
    task_id_2 = manager.create_task("Walk dog", "Take Max to the park")
    task_id_3 = manager.create_task("Finish report", "")
    assert task_id_2 == 2, f"Expected task ID 2, got {task_id_2}"
    assert task_id_3 == 3, f"Expected task ID 3, got {task_id_3}"

    # Test 4: Get all tasks
    all_tasks = manager.get_all_tasks()
    assert len(all_tasks) == 3, f"Expected 3 tasks, got {len(all_tasks)}"
    assert all_tasks[0].id == 1, "Tasks should be ordered by creation time"
    assert all_tasks[1].id == 2, "Tasks should be ordered by creation time"
    assert all_tasks[2].id == 3, "Tasks should be ordered by creation time"

    # Test 5: Status update
    success = manager.update_task_status(2, TaskStatus.IN_PROGRESS)
    assert success == True, "Should return True when updating existing task"

    updated_task = manager.get_task(2)
    assert updated_task.status == TaskStatus.IN_PROGRESS, f"Expected IN_PROGRESS, got
{updated_task.status}"

    # Test 6: Status update on non-existent task
    success = manager.update_task_status(999, TaskStatus.DONE)
    assert success == False, "Should return False when updating non-existent task"

    # Test 7: Get tasks by status
    todo_tasks = manager.get_tasks_by_status(TaskStatus.TODO)
    assert len(todo_tasks) == 2, f"Expected 2 TODO tasks, got {len(todo_tasks)}"

    in_progress_tasks = manager.get_tasks_by_status(TaskStatus.IN_PROGRESS)
    assert len(in_progress_tasks) == 1, f"Expected 1 IN_PROGRESS task, got
{len(in_progress_tasks)}"
    assert in_progress_tasks[0].id == 2, "Should be task ID 2"

    done_tasks = manager.get_tasks_by_status(TaskStatus.DONE)
    assert len(done_tasks) == 0, f"Expected 0 DONE tasks, got {len(done_tasks)}"

    # Test 8: Task deletion
    success = manager.delete_task(1)
    assert success == True, "Should return True when deleting existing task"

    deleted_task = manager.get_task(1)
    assert deleted_task is None, "Deleted task should not be found"
```

```
    # Test 9: Delete non-existent task
    success = manager.delete_task(999)
    assert success == False, "Should return False when deleting non-existent task"

    # Test 10: Verify state after deletion
    remaining_tasks = manager.get_all_tasks()
    assert len(remaining_tasks) == 2, f"Expected 2 remaining tasks, got {len(remaining_tasks)}"
    assert remaining_tasks[0].id == 2, "Should have task ID 2"
    assert remaining_tasks[1].id == 3, "Should have task ID 3"

    # Test 11: Empty title validation
    try:
        manager.create_task("", "Some description")
        assert False, "Should raise ValueError for empty title"
    except ValueError:
        pass  # Expected

    try:
        manager.create_task(None, "Some description")
        assert False, "Should raise ValueError for None title"
    except ValueError:
        pass  # Expected

    # Test 12: Get non-existent task
    non_existent = manager.get_task(999)
    assert non_existent is None, "Non-existent task should return None"

    print("✅ All Level 1 tests passed!")

if __name__ == "__main__":
    test_level_1()
```

Level 2: Task Dependencies and Priorities

Task: Add task dependencies, priorities, and smart status management

```
# Task Management System - Level 2
# Time Limit: ~25 minutes for this level
# NOTE: All Level 1 functionality must continue to work!

from abc import ABC, abstractmethod
from typing import Dict, List, Optional, Set
from dataclasses import dataclass, field
from enum import Enum

class TaskStatus(Enum):
    """Enumeration for task status values."""
```

```
    TODO = "todo"
    IN_PROGRESS = "in_progress"
    DONE = "done"
    BLOCKED = "blocked"  # NEW: Task cannot start due to dependencies

class TaskPriority(Enum):
    """Enumeration for task priority levels."""
    LOW = 1
    MEDIUM = 2
    HIGH = 3
    URGENT = 4

@dataclass
class Task:
    """Represents a single task in the system."""
    id: int
    title: str
    description: str
    status: TaskStatus
    priority: TaskPriority = TaskPriority.MEDIUM  # NEW: Default priority
    dependencies: Set[int] = field(default_factory=set)  # NEW: Task IDs this task depends on

class TaskManagerInterface(ABC):
    """Interface for the task management system - Level 2.

    ALL Level 1 methods must continue to work exactly as before.
    NEW methods are added for Level 2 functionality.
    """

    # ========== LEVEL 1 METHODS (MUST STILL WORK) ==========

    @abstractmethod
    def create_task(self, title: str, description: str) -> int:
        """Create a new task with the given title and description.

        The task should start with status TaskStatus.TODO and priority TaskPriority.MEDIUM.
        Task IDs should be unique integers starting from 1.

        Args:
            title: The task title (non-empty string)
            description: The task description (can be empty string)

        Returns:
            The unique ID of the created task

        Raises:
            ValueError: If title is empty or None
        """
        pass
```

```
@abstractmethod
def get_task(self, task_id: int) -> Optional[Task]:
    """Retrieve a task by its ID."""
    pass

@abstractmethod
def get_all_tasks(self) -> List[Task]:
    """Get all tasks in the system, ordered by creation time (oldest first)."""
    pass

@abstractmethod
def update_task_status(self, task_id: int, status: TaskStatus) -> bool:
    """Update the status of an existing task."""
    pass

@abstractmethod
def delete_task(self, task_id: int) -> bool:
    """Delete a task from the system."""
    pass

@abstractmethod
def get_tasks_by_status(self, status: TaskStatus) -> List[Task]:
    """Get all tasks with a specific status, ordered by creation time."""
    pass

# ========== NEW LEVEL 2 METHODS ==========

@abstractmethod
def create_task_with_priority(self, title: str, description: str, priority: TaskPriority) -> int:
    """Create a new task with specified priority.

    Args:
        title: The task title (non-empty string)
        description: The task description (can be empty string)
        priority: The priority level for the task

    Returns:
        The unique ID of the created task

    Raises:
        ValueError: If title is empty or None
    """
    pass

@abstractmethod
def update_task_priority(self, task_id: int, priority: TaskPriority) -> bool:
    """Update the priority of an existing task.

    Args:
        task_id: The ID of the task to update
```

```
        priority: The new priority for the task

    Returns:
        True if the task was found and updated, False otherwise
    """
    pass

@abstractmethod
def add_dependency(self, task_id: int, depends_on_task_id: int) -> bool:
    """Add a dependency relationship between tasks.

    Task with task_id cannot be started until depends_on_task_id is DONE.

    Args:
        task_id: The ID of the task that has the dependency
        depends_on_task_id: The ID of the task that must be completed first

    Returns:
        True if dependency was added successfully, False otherwise

    Rules:
        - Both tasks must exist
        - Cannot create circular dependencies
        - Cannot depend on yourself
        - Dependency can only be added if target task is not DONE
    """
    pass

@abstractmethod
def remove_dependency(self, task_id: int, depends_on_task_id: int) -> bool:
    """Remove a dependency relationship between tasks.

    Args:
        task_id: The ID of the task to remove dependency from
        depends_on_task_id: The ID of the dependency to remove

    Returns:
        True if dependency was removed, False if it didn't exist
    """
    pass

@abstractmethod
def get_available_tasks(self) -> List[Task]:
    """Get all tasks that can be started (no blocking dependencies).

    A task is available if:
    - Status is TODO
    - All its dependencies are DONE

    Returns:
```

```
            List of available Task objects, ordered by priority (URGENT first)
            then by creation time for same priority
        """
        pass

    @abstractmethod
    def get_blocked_tasks(self) -> List[Task]:
        """Get all tasks that are blocked by dependencies.

        A task is blocked if:
        - Status is TODO or BLOCKED
        - Has at least one dependency that is not DONE

        Returns:
            List of blocked Task objects, ordered by creation time
        """
        pass

    @abstractmethod
    def auto_update_blocked_status(self) -> int:
        """Automatically update task statuses based on dependencies.

        Rules:
        - If a TODO task has incomplete dependencies, set to BLOCKED
        - If a BLOCKED task has all dependencies complete, set to TODO
        - Don't change IN_PROGRESS or DONE tasks

        Returns:
            Number of tasks whose status was changed
        """
        pass

#
==========================================================================
=
# IMPLEMENT YOUR SOLUTION BELOW
#
==========================================================================
=

class TaskManager(TaskManagerInterface):
    """Your implementation of the TaskManager.

    ALL Level 1 functionality must continue working!
    Add new Level 2 functionality while maintaining backward compatibility.
    """

    def __init__(self):
        # Initialize your data structures here
        # HINT: You'll need to handle dependencies and track creation order
```

```
    pass

# ========== LEVEL 1 METHODS (KEEP THESE WORKING) ==========

def create_task(self, title: str, description: str) -> int:
    # TODO: Update to set default priority
    pass

def get_task(self, task_id: int) -> Optional[Task]:
    # TODO: Implement this method
    pass

def get_all_tasks(self) -> List[Task]:
    # TODO: Implement this method
    pass

def update_task_status(self, task_id: int, status: TaskStatus) -> bool:
    # TODO: Implement this method
    pass

def delete_task(self, task_id: int) -> bool:
    # TODO: Also need to remove this task from other tasks' dependencies
    pass

def get_tasks_by_status(self, status: TaskStatus) -> List[Task]:
    # TODO: Implement this method
    pass

# ========== NEW LEVEL 2 METHODS ==========

def create_task_with_priority(self, title: str, description: str, priority: TaskPriority) -> int:
    # TODO: Implement this method
    pass

def update_task_priority(self, task_id: int, priority: TaskPriority) -> bool:
    # TODO: Implement this method
    pass

def add_dependency(self, task_id: int, depends_on_task_id: int) -> bool:
    # TODO: Implement with cycle detection
    pass

def remove_dependency(self, task_id: int, depends_on_task_id: int) -> bool:
    # TODO: Implement this method
    pass

def get_available_tasks(self) -> List[Task]:
    # TODO: Filter TODO tasks with no blocking dependencies, sort by priority
    pass
```

```
    def get_blocked_tasks(self) -> List[Task]:
        # TODO: Find tasks blocked by incomplete dependencies
        pass

    def auto_update_blocked_status(self) -> int:
        # TODO: Update BLOCKED/TODO status based on dependencies
        pass

#
==========================================================================
=
# TEST CASES - DO NOT MODIFY
#
==========================================================================
=

def test_level_2():
    """Test cases for Level 2. All must pass to proceed to Level 3."""

    print("Testing Level 2 functionality...")

    # Test 1: Level 1 functionality still works
    manager = TaskManager()
    task1_id = manager.create_task("Task 1", "First task")
    task2_id = manager.create_task("Task 2", "Second task")

    task1 = manager.get_task(task1_id)
    assert task1.priority == TaskPriority.MEDIUM, f"Default priority should be MEDIUM, got
{task1.priority}"
    assert len(task1.dependencies) == 0, "New task should have no dependencies"

    # Test 2: Create task with custom priority
    task3_id = manager.create_task_with_priority("Urgent task", "Very important",
TaskPriority.URGENT)
    task3 = manager.get_task(task3_id)
    assert task3.priority == TaskPriority.URGENT, f"Expected URGENT priority, got {task3.priority}"

    # Test 3: Update task priority
    success = manager.update_task_priority(task1_id, TaskPriority.HIGH)
    assert success == True, "Should successfully update priority"

    updated_task1 = manager.get_task(task1_id)
    assert updated_task1.priority == TaskPriority.HIGH, f"Expected HIGH priority, got
{updated_task1.priority}"

    # Test 4: Add valid dependency
    success = manager.add_dependency(task2_id, task1_id)  # task2 depends on task1
    assert success == True, "Should successfully add dependency"

    task2 = manager.get_task(task2_id)
```

```
    assert task1_id in task2.dependencies, f"Task {task1_id} should be in dependencies of task {task2_id}"

    # Test 5: Prevent self-dependency
    success = manager.add_dependency(task1_id, task1_id)
    assert success == False, "Should not allow self-dependency"

    # Test 6: Prevent circular dependency
    success = manager.add_dependency(task1_id, task2_id)  # Would create cycle: task1 -> task2 -> task1
    assert success == False, "Should not allow circular dependency"

    # Test 7: Add dependency to non-existent task
    success = manager.add_dependency(999, task1_id)
    assert success == False, "Should fail when task doesn't exist"

    success = manager.add_dependency(task1_id, 999)
    assert success == False, "Should fail when dependency target doesn't exist"

    # Test 8: Get available tasks (no blocking dependencies)
    available = manager.get_available_tasks()
    available_ids = [t.id for t in available]
    assert task1_id in available_ids, "Task 1 should be available (no dependencies)"
    assert task2_id not in available_ids, "Task 2 should not be available (depends on task 1)"
    assert task3_id in available_ids, "Task 3 should be available (no dependencies)"

    # Available tasks should be sorted by priority (URGENT first)
    assert available[0].id == task3_id, "URGENT task should be first"

    # Test 9: Get blocked tasks
    blocked = manager.get_blocked_tasks()
    blocked_ids = [t.id for t in blocked]
    assert task2_id in blocked_ids, "Task 2 should be blocked"
    assert task1_id not in blocked_ids, "Task 1 should not be blocked"
    assert task3_id not in blocked_ids, "Task 3 should not be blocked"

    # Test 10: Auto-update blocked status
    changes = manager.auto_update_blocked_status()
    assert changes == 1, f"Should update 1 task status, got {changes}"

    updated_task2 = manager.get_task(task2_id)
    assert updated_task2.status == TaskStatus.BLOCKED, f"Task 2 should be BLOCKED, got {updated_task2.status}"

    # Test 11: Complete dependency and check unblocking
    manager.update_task_status(task1_id, TaskStatus.DONE)
    changes = manager.auto_update_blocked_status()
    assert changes == 1, f"Should unblock 1 task, got {changes}"

    unblocked_task2 = manager.get_task(task2_id)
```

```
    assert unblocked_task2.status == TaskStatus.TODO, f"Task 2 should be TODO after dependency completed, got {unblocked_task2.status}"

    # Test 12: Remove dependency
    success = manager.remove_dependency(task2_id, task1_id)
    assert success == True, "Should successfully remove dependency"

    task2_after_removal = manager.get_task(task2_id)
    assert task1_id not in task2_after_removal.dependencies, "Dependency should be removed"

    # Test 13: Remove non-existent dependency
    success = manager.remove_dependency(task2_id, task1_id)
    assert success == False, "Should return False when removing non-existent dependency"

    # Test 14: Delete task removes it from other dependencies
    task4_id = manager.create_task("Task 4", "Fourth task")
    manager.add_dependency(task4_id, task3_id)

    # Verify dependency exists
    task4 = manager.get_task(task4_id)
    assert task3_id in task4.dependencies, "Dependency should exist before deletion"

    # Delete the dependency target
    manager.delete_task(task3_id)

    # Verify dependency is removed
    task4_after_deletion = manager.get_task(task4_id)
    assert task3_id not in task4_after_deletion.dependencies, "Dependency should be removed when target is deleted"

    # Test 15: Complex dependency chain
    task5_id = manager.create_task_with_priority("Task 5", "Fifth task", TaskPriority.LOW)
    task6_id = manager.create_task_with_priority("Task 6", "Sixth task", TaskPriority.HIGH)

    # Create chain: task6 -> task5 -> task4
    manager.add_dependency(task6_id, task5_id)
    manager.add_dependency(task5_id, task4_id)

    # Only task4 should be available initially
    available_after_chain = manager.get_available_tasks()
    available_ids_after_chain = [t.id for t in available_after_chain]
    assert task4_id in available_ids_after_chain, "Task 4 should be available"
    assert task5_id not in available_ids_after_chain, "Task 5 should be blocked"
    assert task6_id not in available_ids_after_chain, "Task 6 should be blocked"

    print("✅ All Level 2 tests passed!")

if __name__ == "__main__":
    test_level_2()
```

Level 3: Task Assignment and Deadlines

Task: Add user management, task assignments, and due date tracking

```
# Task Management System - Level 3
# Time Limit: ~30 minutes for this level
# NOTE: All Level 1 and Level 2 functionality must continue to work!

from abc import ABC, abstractmethod
from typing import Dict, List, Optional, Set
from dataclasses import dataclass, field
from enum import Enum
from datetime import datetime, date

class TaskStatus(Enum):
    """Enumeration for task status values."""
    TODO = "todo"
    IN_PROGRESS = "in_progress"
    DONE = "done"
    BLOCKED = "blocked"
    OVERDUE = "overdue"  # NEW: Task is past due date

class TaskPriority(Enum):
    """Enumeration for task priority levels."""
    LOW = 1
    MEDIUM = 2
    HIGH = 3
    URGENT = 4

@dataclass
class User:
    """Represents a user in the system."""
    id: int
    name: str
    email: str

@dataclass
class Task:
    """Represents a single task in the system."""
    id: int
    title: str
    description: str
    status: TaskStatus
    priority: TaskPriority = TaskPriority.MEDIUM
    dependencies: Set[int] = field(default_factory=set)
    assignee_id: Optional[int] = None  # NEW: User ID of person assigned to task
    due_date: Optional[date] = None  # NEW: When task is due
```

```
    created_date: date = field(default_factory=date.today)  # NEW: When task was created

class TaskManagerInterface(ABC):
    """Interface for the task management system - Level 3.

    ALL Level 1 and Level 2 methods must continue to work exactly as before.
    NEW methods are added for Level 3 functionality.
    """

    # ========== LEVEL 1 & 2 METHODS (MUST STILL WORK) ==========
    # [Previous methods abbreviated for space - include all from Level 2]

    # ========== NEW LEVEL 3 METHODS ==========

    @abstractmethod
    def create_user(self, name: str, email: str) -> int:
        """Create a new user in the system.

        Args:
            name: The user's display name (non-empty string)
            email: The user's email address (non-empty string)

        Returns:
            The unique ID of the created user

        Raises:
            ValueError: If name or email is empty or None
        """
        pass

    @abstractmethod
    def get_user(self, user_id: int) -> Optional[User]:
        """Retrieve a user by their ID.

        Args:
            user_id: The ID of the user to retrieve

        Returns:
            The User object if found, None otherwise
        """
        pass

    @abstractmethod
    def assign_task(self, task_id: int, user_id: int) -> bool:
        """Assign a task to a user.

        Args:
            task_id: The ID of the task to assign
            user_id: The ID of the user to assign the task to
```

```
    Returns:
        True if assignment was successful, False otherwise

    Rules:
        - Both task and user must exist
        - Cannot assign DONE tasks
    """
    pass

@abstractmethod
def unassign_task(self, task_id: int) -> bool:
    """Remove assignment from a task.

    Args:
        task_id: The ID of the task to unassign

    Returns:
        True if task was unassigned, False if task doesn't exist
    """
    pass

@abstractmethod
def get_user_tasks(self, user_id: int) -> List[Task]:
    """Get all tasks assigned to a specific user.

    Args:
        user_id: The ID of the user

    Returns:
        List of Task objects assigned to the user, ordered by due date (earliest first),
        then by priority (URGENT first), then by creation time
    """
    pass

@abstractmethod
def set_due_date(self, task_id: int, due_date: date) -> bool:
    """Set the due date for a task.

    Args:
        task_id: The ID of the task
        due_date: The due date for the task

    Returns:
        True if due date was set, False if task doesn't exist
    """
    pass

@abstractmethod
def get_overdue_tasks(self) -> List[Task]:
    """Get all tasks that are past their due date.
```

```
    A task is overdue if:
    - Has a due date
    - Due date is before today
    - Status is not DONE

    Returns:
        List of overdue Task objects, ordered by how overdue they are (most overdue first)
    """
    pass

@abstractmethod
def update_overdue_status(self) -> int:
    """Automatically update task statuses to OVERDUE if past due date.

    Rules:
    - If a task has a due date before today and status is not DONE, set to OVERDUE
    - Don't change DONE tasks

    Returns:
        Number of tasks whose status was changed to OVERDUE
    """
    pass

@abstractmethod
def get_tasks_due_soon(self, days: int) -> List[Task]:
    """Get tasks that are due within the specified number of days.

    Args:
        days: Number of days to look ahead

    Returns:
        List of Task objects due within 'days' days, ordered by due date (earliest first)
    """
    pass

@abstractmethod
def bulk_assign_tasks(self, task_ids: List[int], user_id: int) -> int:
    """Assign multiple tasks to a user at once.

    Args:
        task_ids: List of task IDs to assign
        user_id: The ID of the user to assign tasks to

    Returns:
        Number of tasks successfully assigned
    """
    pass

@abstractmethod
```

```
    def get_task_summary_by_user(self) -> Dict[int, Dict[str, int]]:
        """Get a summary of task counts by user and status.

        Returns:
            Dictionary mapping user_id to a dictionary of status counts.
            Example: {
                1: {"TODO": 3, "IN_PROGRESS": 1, "DONE": 5, "BLOCKED": 0, "OVERDUE": 1},
                2: {"TODO": 1, "IN_PROGRESS": 2, "DONE": 3, "BLOCKED": 1, "OVERDUE": 0}
            }
            Include users with 0 tasks as empty dictionaries.
        """
        pass

#
==========================================================================
=
# IMPLEMENT YOUR SOLUTION BELOW - BUILD ON YOUR LEVEL 2 IMPLEMENTATION
#
==========================================================================
=

class TaskManager(TaskManagerInterface):
    """Your implementation of the TaskManager.

    ALL Level 1 and Level 2 functionality must continue working!
    Add new Level 3 functionality while maintaining backward compatibility.
    """

    def __init__(self):
        # Your existing data structures from Levels 1-2
        # Add new data structures for users
        pass

    # [Include all your Level 1 & 2 methods here]

    # ========== NEW LEVEL 3 METHODS - IMPLEMENT THESE ==========

    def create_user(self, name: str, email: str) -> int:
        # TODO: Implement user creation
        pass

    def get_user(self, user_id: int) -> Optional[User]:
        # TODO: Implement user retrieval
        pass

    def assign_task(self, task_id: int, user_id: int) -> bool:
        # TODO: Implement task assignment
        pass

    def unassign_task(self, task_id: int) -> bool:
```

```
        # TODO: Implement task unassignment
        pass

    def get_user_tasks(self, user_id: int) -> List[Task]:
        # TODO: Get tasks for a specific user, sorted by due date, priority, creation time
        pass

    def set_due_date(self, task_id: int, due_date: date) -> bool:
        # TODO: Set due date for a task
        pass

    def get_overdue_tasks(self) -> List[Task]:
        # TODO: Get tasks past their due date
        pass

    def update_overdue_status(self) -> int:
        # TODO: Update tasks to OVERDUE status if past due
        pass

    def get_tasks_due_soon(self, days: int) -> List[Task]:
        # TODO: Get tasks due within specified days
        pass

    def bulk_assign_tasks(self, task_ids: List[int], user_id: int) -> int:
        # TODO: Assign multiple tasks to one user
        pass

    def get_task_summary_by_user(self) -> Dict[int, Dict[str, int]]:
        # TODO: Return task count summary by user and status
        pass

# [Include Level 3 test cases - similar structure to Level 2]
```

Level 4: Projects, Templates, and Time Tracking

Task: Add project management, task templates, and comprehensive reporting

```
# Task Management System - Level 4 (FINAL LEVEL)
# Time Limit: ~40 minutes for this level
# NOTE: All previous level functionality must continue to work!

from abc import ABC, abstractmethod
from typing import Dict, List, Optional, Set, Union
from dataclasses import dataclass, field
from enum import Enum
from datetime import datetime, date, timedelta
import copy
```

```
# [Include all previous enums and dataclasses, plus new ones:]

@dataclass
class Project:
    """Represents a project containing multiple tasks."""
    id: int
    name: str
    description: str
    owner_id: int  # User ID of project owner
    created_date: date = field(default_factory=date.today)

@dataclass
class TaskTemplate:
    """Template for creating similar tasks."""
    id: int
    name: str
    title_template: str  # Can contain {project_name} placeholders
    description_template: str
    default_priority: TaskPriority
    estimated_hours: Optional[float] = None

@dataclass
class TimeEntry:
    """Represents time logged on a task."""
    task_id: int
    user_id: int
    hours: float
    date: date
    description: str

@dataclass
class Task:
    """Updated Task with new Level 4 fields."""
    id: int
    title: str
    description: str
    status: TaskStatus
    priority: TaskPriority = TaskPriority.MEDIUM
    dependencies: Set[int] = field(default_factory=set)
    assignee_id: Optional[int] = None
    due_date: Optional[date] = None
    created_date: date = field(default_factory=date.today)
    project_id: Optional[int] = None  # NEW: Project this task belongs to
    estimated_hours: Optional[float] = None  # NEW: Estimated time to complete
    tags: Set[str] = field(default_factory=set)  # NEW: Tags for categorization

class TaskManagerInterface(ABC):
    """Interface for the task management system - Level 4 (FINAL).
```

```
ALL previous level methods must continue to work exactly as before.
NEW methods are added for Level 4 functionality.
"""

# [Include all previous methods]

# ========== NEW LEVEL 4 METHODS ==========

@abstractmethod
def create_project(self, name: str, description: str, owner_id: int) -> int:
    """Create a new project."""
    pass

@abstractmethod
def get_project(self, project_id: int) -> Optional[Project]:
    """Retrieve a project by its ID."""
    pass

@abstractmethod
def assign_task_to_project(self, task_id: int, project_id: int) -> bool:
    """Assign a task to a project."""
    pass

@abstractmethod
def get_project_tasks(self, project_id: int) -> List[Task]:
    """Get all tasks in a project, ordered by priority then due date."""
    pass

@abstractmethod
def create_task_template(self, name: str, title_template: str,
                         description_template: str, default_priority: TaskPriority,
                         estimated_hours: Optional[float] = None) -> int:
    """Create a reusable task template."""
    pass

@abstractmethod
def create_task_from_template(self, template_id: int, project_id: Optional[int] = None) -> int:
    """Create a new task from a template."""
    pass

@abstractmethod
def add_task_tags(self, task_id: int, tags: List[str]) -> bool:
    """Add tags to a task."""
    pass

@abstractmethod
def remove_task_tags(self, task_id: int, tags: List[str]) -> bool:
    """Remove tags from a task."""
    pass
```

```
@abstractmethod
def get_tasks_by_tags(self, tags: List[str], match_all: bool = True) -> List[Task]:
    """Get tasks that have specific tags."""
    pass

@abstractmethod
def log_time(self, task_id: int, user_id: int, hours: float, description: str,
             log_date: Optional[date] = None) -> bool:
    """Log time spent on a task."""
    pass

@abstractmethod
def get_task_time_entries(self, task_id: int) -> List[TimeEntry]:
    """Get all time entries for a task, ordered by date (newest first)."""
    pass

@abstractmethod
def get_user_time_entries(self, user_id: int, start_date: Optional[date] = None,
                          end_date: Optional[date] = None) -> List[TimeEntry]:
    """Get time entries for a user within a date range."""
    pass

@abstractmethod
def get_total_time_spent(self, task_id: int) -> float:
    """Get total hours logged on a task."""
    pass

@abstractmethod
def search_tasks(self, query: str, project_id: Optional[int] = None,
                 assignee_id: Optional[int] = None, status: Optional[TaskStatus] = None) -> List[Task]:
    """Search tasks by title/description text with optional filters."""
    pass

@abstractmethod
def get_project_progress(self, project_id: int) -> Dict[str, Union[int, float]]:
    """Get progress statistics for a project."""
    pass

@abstractmethod
def bulk_update_task_status(self, task_ids: List[int], status: TaskStatus) -> int:
    """Update status for multiple tasks at once."""
    pass

@abstractmethod
def get_productivity_report(self, user_id: int, days: int) -> Dict[str, Union[int, float]]:
    """Generate productivity report for a user over the last N days."""
    pass
```

```
#
==========================================================================
=
# IMPLEMENT YOUR SOLUTION BELOW - BUILD ON YOUR LEVEL 3 IMPLEMENTATION
#
==========================================================================
=

class TaskManager(TaskManagerInterface):
    """Your implementation of the TaskManager - Level 4 (FINAL).

    All previous functionality is preserved. Implement the new Level 4 methods.
    """

    def __init__(self):
        # Your existing data structures from previous levels
        # Add new Level 4 data structures for projects, templates, time entries
        pass

    # [Include ALL your methods from Levels 1-3]

    # ========== NEW LEVEL 4 METHODS - IMPLEMENT THESE ==========

    def create_project(self, name: str, description: str, owner_id: int) -> int:
        # TODO: Implement project creation
        pass

    def create_task_template(self, name: str, title_template: str,
                             description_template: str, default_priority: TaskPriority,
                             estimated_hours: Optional[float] = None) -> int:
        # TODO: Create reusable task template
        pass

    def create_task_from_template(self, template_id: int, project_id: Optional[int] = None) -> int:
        # TODO: Create task from template, handle {project_name} substitution
        pass

    def add_task_tags(self, task_id: int, tags: List[str]) -> bool:
        # TODO: Add tags to task
        pass

    def log_time(self, task_id: int, user_id: int, hours: float, description: str,
                 log_date: Optional[date] = None) -> bool:
        # TODO: Log time entry for task
        pass

    def search_tasks(self, query: str, project_id: Optional[int] = None,
                     assignee_id: Optional[int] = None, status: Optional[TaskStatus] = None) -> List[Task]:
        # TODO: Search tasks with filters, prioritize exact title matches
        pass
```

```
    def get_project_progress(self, project_id: int) -> Dict[str, Union[int, float]]:
        # TODO: Calculate project completion statistics
        pass

    def get_productivity_report(self, user_id: int, days: int) -> Dict[str, Union[int, float]]:
        # TODO: Generate productivity metrics for user
        pass

    # [Implement all other Level 4 methods]

# [Include comprehensive Level 4 test cases]
```

Tips for Success

Level-by-Level Strategy:

1. **Level 1**: Focus on basic data structures (dictionary for tasks, counter for IDs)
2. **Level 2**: Add cycle detection for dependencies (use BFS/DFS), priority-based sorting
3. **Level 3**: Add user management, date arithmetic, complex multi-key sorting
4. **Level 4**: String templating, advanced search with ranking, statistical calculations

Common Pitfalls:

- Breaking earlier functionality when adding new features
- Forgetting to update Task dataclass fields between levels
- Cycle detection bugs (infinite loops)
- Incorrect sorting logic (especially multi-key sorts)
- Date boundary conditions
- Template string substitution edge cases

Test-Driven Development:

- Run tests immediately after implementing each method
- The tests define the exact requirements - trust them over documentation
- If a test fails, read it carefully to understand what's expected
- Don't overthink edge cases that aren't tested

Time Management:

- Spend no more than the suggested time per level
- If stuck on one method, move to the next and return later
- Prioritize breadth over depth - get all levels working partially rather than perfecting one level

Remember: This is about speed and correctness under pressure. Sacrifice code quality for passing tests!

Good luck! 🚀