ACADEMIC TASK - 2 CSE316

(OPERATING SYSTEMS)

LOVELY PROFESSIONAL UNIVERSITY

TOPIC: "Real-Time Memory Allocation Tracker"

PROJECT REPORT

By

Reg. No.	Name
12307183	Roshani Kumari
12321291	Reshu kushwah
12317024	Rama kumari

TO

Amandeep Kaur(UID: 31019)

1. Project Overview

The "Real-Time Memory Allocation Tracker" is a tool that shows how memory is used and freed up as it happens. It focuses on simple memory tricks like paging and segmentation. The tool pretends to have a fixed amount of memory, letting you take or give back pieces of it using easy methods like best-fit and worst-fit. It also draws a picture of the memory and shows some basic computer stats. The goal is to teach you about memory stuff, show different ways to handle it, and let you try it out yourself. You'll get a desktop app that updates live, gives clear numbers, and makes memory easy to see. It's a basic Python program that runs on one thread, but you can add more features like paging later if you want.

2. Module-Wise Breakdown

The project is divided into three modules:

- Memory Management Core:
    - Purpose: Manages the logic for memory allocation, deallocation, and algorithm evaluation. ○ Role: Serves as the backend, maintaining the memory state and providing data for visualization.
- Graphical User Interface (GUI):
    - Purpose: Offers an interactive interface for users to control and observe memory operations. ○ Role: Displays memory blocks, system stats, and user controls, connecting the core logic to the user.
- System Monitoring and Visualization:
    - Purpose: Tracks system performance (CPU, RAM) and visualizes memory allocation graphically. ○ Role: Provides context with real-world system data and a dynamic visual representation of memory.

3. Functionalities

- Memory Management Core:
    - Allocate memory blocks based on size and algorithm (e.g., allocate 50 units with best-fit).
    - Deallocate memory by ID, merging free blocks (e.g., free ID 3).

- Evaluate best-fit vs. worst-fit algorithms (e.g., 20 random operations, reporting fragmentation).
- Track memory stats (e.g., "Total: 1024, Used: 300, Free: 724").

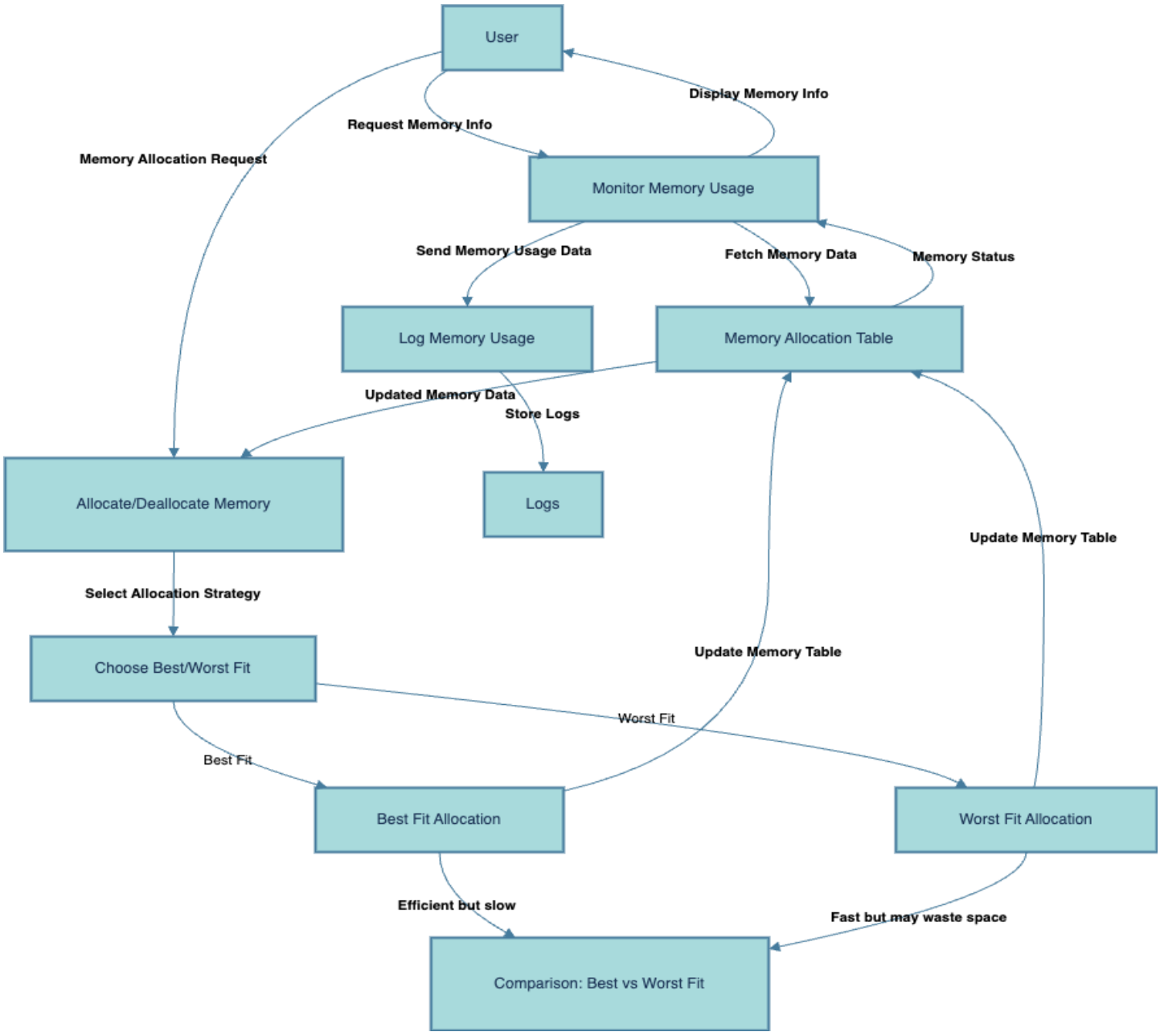
- Graphical User Interface (GUI):
  - Manual allocation/deallocation controls (e.g., enter "50" and click "Allocate").
  - Algorithm selection via radio buttons (e.g., switch to "Worst-Fit"). ○ Automated testing feature (e.g., simulate 10 random operations).
  - Error feedback (e.g., "Not enough memory" popup).
- System Monitoring and Visualization:
  - Display CPU/RAM usage (e.g., "CPU: 45%, RAM: 4 GB / 8 GB"). ○ List top 10 processes by memory usage (e.g., "chrome.exe: 512 MB").
  - Visualize memory blocks (e.g., red for allocated, green for free).
  - Real-time updates (e.g., canvas refreshes after allocation).

4. Technology Used

- Programming Languages:
  - Python: Chosen for its simplicity, extensive library support, and alignment with the provided code.
- Libraries and Tools:
  - Tkinter: Used for building the GUI, providing a lightweight and native desktop interface. ○ psutil: Monitors system resources (CPU, RAM, processes) with cross-platform compatibility. ○ random: Generates random sizes and operations for automated testing.
  - time: Manages timing for real-time updates and delays.
- Other Tools:

- [GitHub for version control]: Recommended for tracking revisions (placeholder until repository is created). ○ [PyCharm/VS Code]: Suggested IDEs for development and debugging.

5. Flow Diagram



6. . Revision Tracking on GitHub

• Repository Name**:** [Real-Time-Memory-Allocation-Tracer]

• GitHub Link**:** [kumariroshani-890/_Real-Time-Memory-Allocation-Tracker_]

7. Conclusion and Future Scope

The project successfully delivers a functional tool for visualizing memory allocation in real-time, with support for best-fit and worst-fit algorithms, system monitoring, and an interactive GUI. It provides a solid foundation for understanding memory management concepts. However, it currently lacks full paging and segmentation implementations, which could be added for a more comprehensive demonstration. Future enhancements could include:

- Implementing paging with fixed-size pages and a page table visualization.
- Adding segmentation with variable-size segments and metadata display.
- Supporting multiple memory allocation strategies (e.g., first-fit, next-fit).
- Shifting it to a web-based version for easier access by more users.
- Integrating real-time memory leak detection or performance benchmarking.

8. References

- Python Documentation: https://docs.python.org/3/
- Tkinter Documentation: https://docs.python.org/3/library/tkinter.html
- psutil Documentation: https://psutil.readthedocs.io/en/latest/
- Operating System Concepts (Silberschatz, Galvin, Gagne) – For memory management theory.

Appendix

A. AI-Generated Project Elaboration/Breakdown Report

Project Overview: Goals include visualizing memory allocation in real-time, demonstrating paging/segmentation, and providing an interactive tool. Expected outcomes are a graphical desktop app with real-time updates and stats. Scope is a single-threaded Python app with a fixed memory size.

Module-Wise Breakdown:

- Memory Management Core: Backend for allocation/deallocation logic.
- GUI: Interface for user interaction and control.
- System Monitoring and Visualization: Tracks system stats and visualizes memory.

Functionalities:

- Core: Allocate/free memory, evaluate algorithms, track stats.
- GUI: Manual controls, algorithm selection, auto-testing, feedback.
- Visualization: System stats, process list, memory block display, real-time updates.

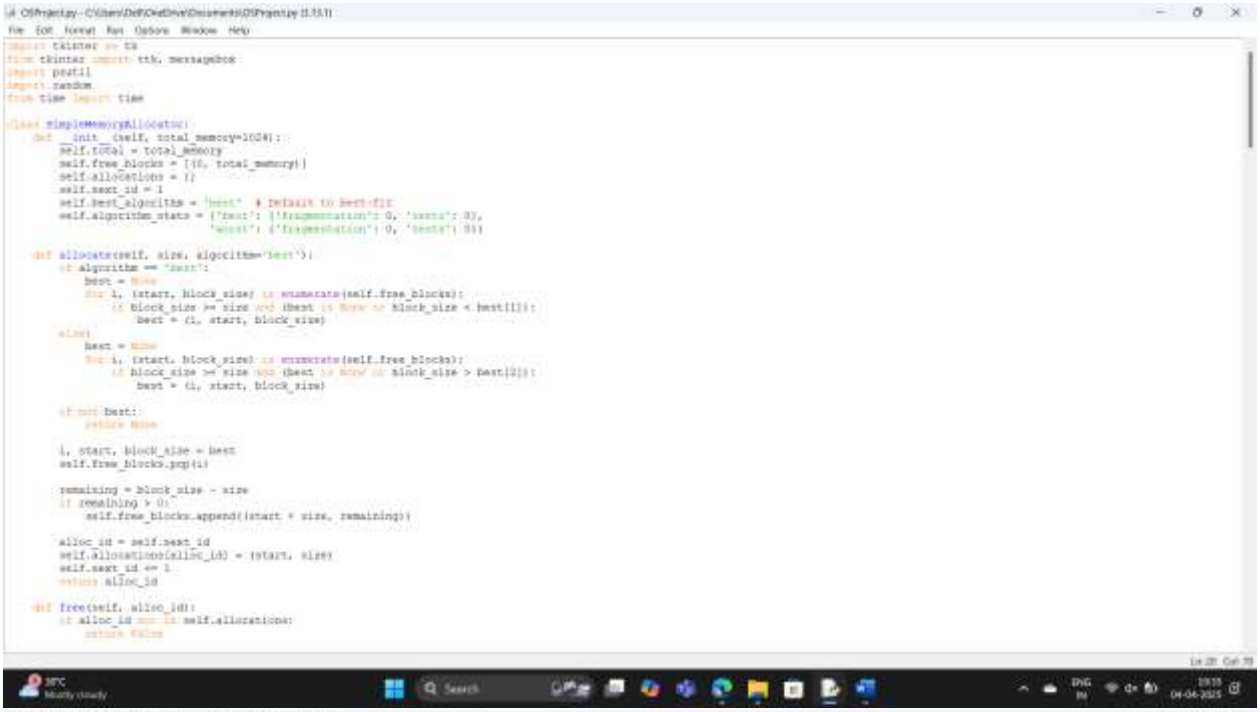
Technology Recommendations: Python with tkinter, psutil, random, time.

Execution Plan: Setup environment, enhance core, refine GUI, integrate visualization, test/debug, finalize.

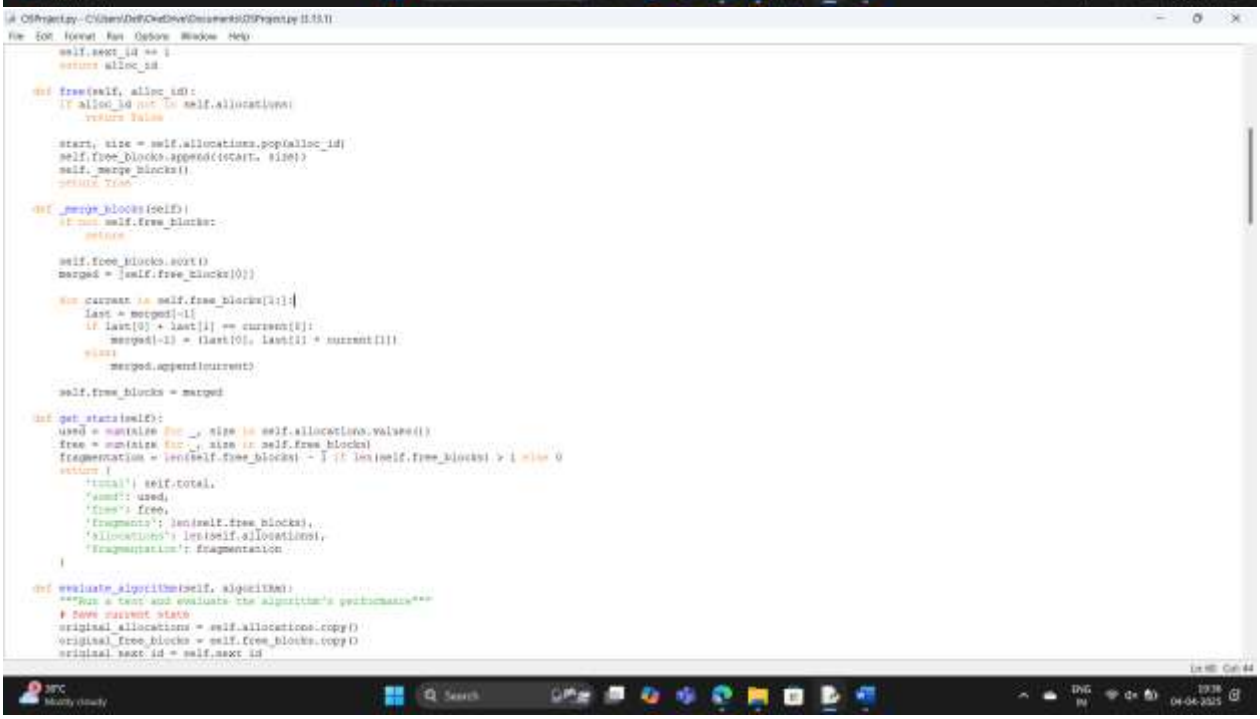
B. Problem Statement

"Real-Time Memory Allocation Tracker
Description: Build a tool that visualizes memory allocation in real-time, showcasing paging and segmentation."

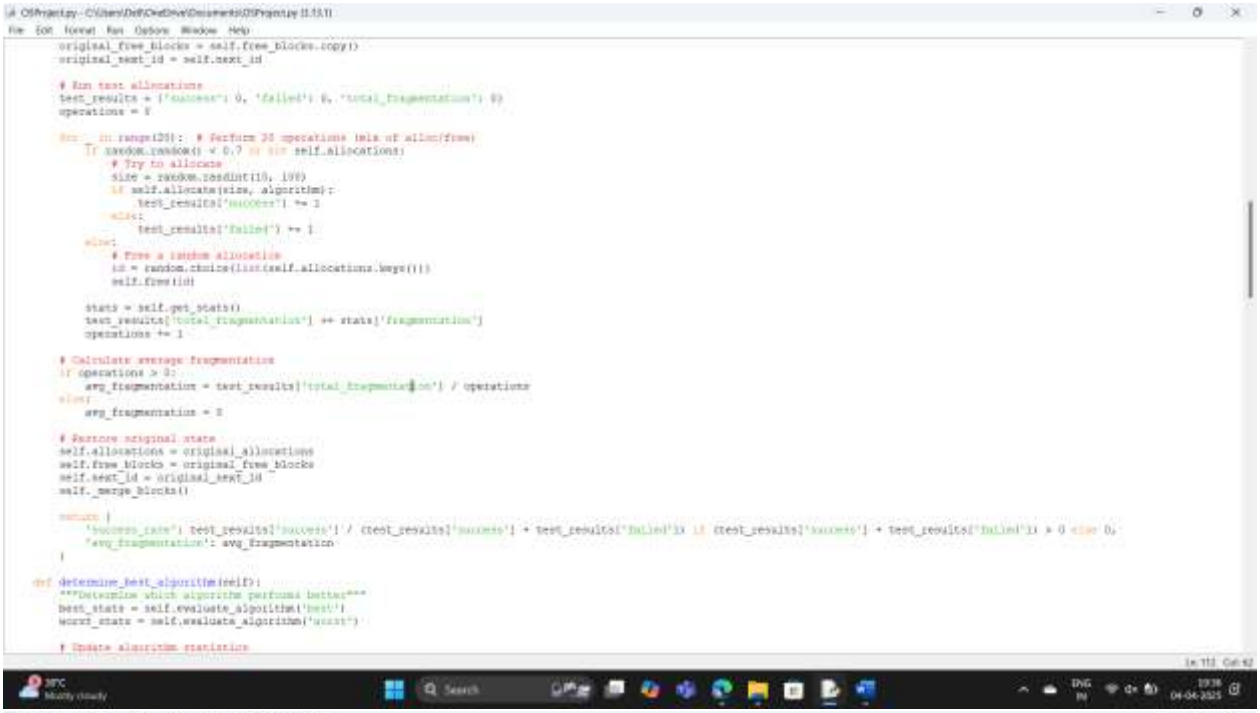
C. Solution/Code



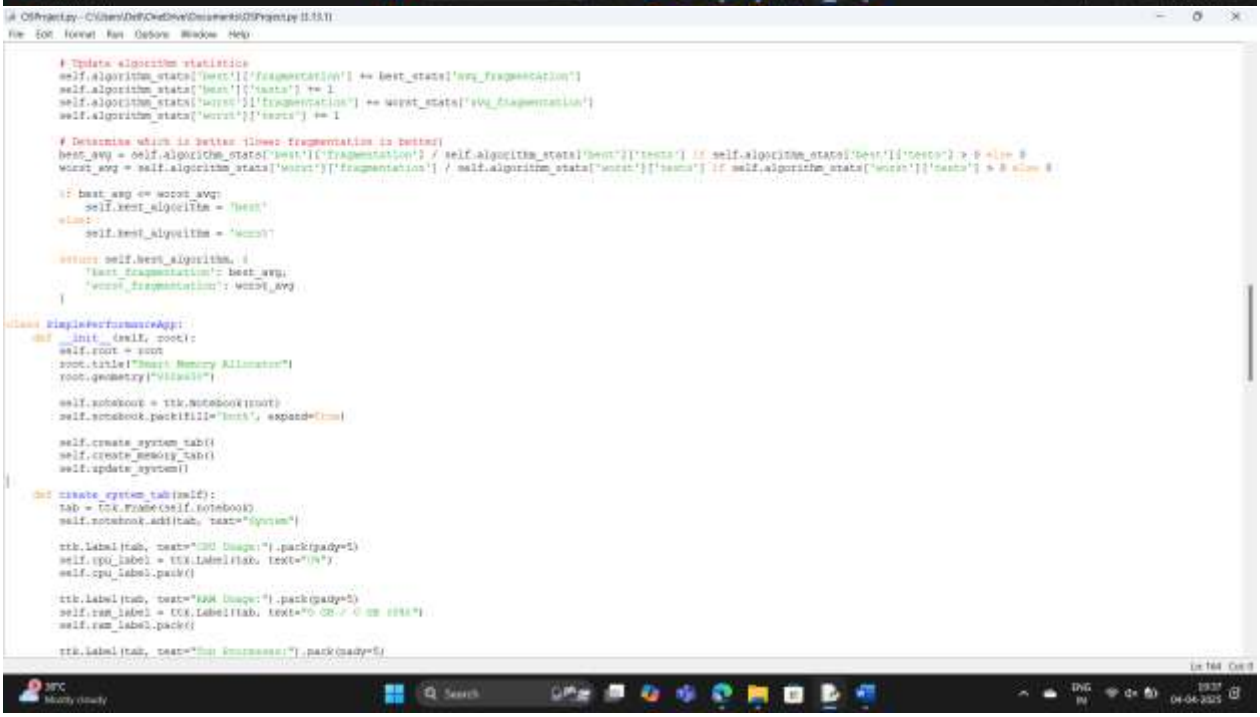
```
File  Edit  Format  Run  Options  Window  Help
import tkinter as tk
from tkinter import ttk, messagebox
import psutil
import random
from time import time

class SimpleMemoryAllocator:
    def __init__(self, total_memory=1024):
        self.total = total_memory
        self.free_blocks = [(0, total_memory)]
        self.allocations = {}
        self.next_id = 1
        self.best_algorithm = 'best'  # Default to best-fit
        self.algorithm_stats = {'best': {'fragmentation': 0, 'tests': 0},
                                'worst': {'fragmentation': 0, 'tests': 0}}

    def allocate(self, size, algorithm='best'):
        if algorithm == 'best':
            best = None
            for i, (start, block_size) in enumerate(self.free_blocks):
                if block_size >= size and (best is None or block_size < best[1]):
                    best = (i, start, block_size)
        else:
            best = None
            for i, (start, block_size) in enumerate(self.free_blocks):
                if block_size >= size and (best is None or block_size > best[2]):
                    best = (i, start, block_size)

        if not best:
            return None

        i, start, block_size = best
        self.free_blocks.pop(i)

        remaining = block_size - size
        if remaining > 0:
            self.free_blocks.append((start + size, remaining))

        alloc_id = self.next_id
        self.allocations[alloc_id] = (start, size)
        self.next_id += 1
        return alloc_id

    def free(self, alloc_id):
        if alloc_id not in self.allocations:
            return False
```



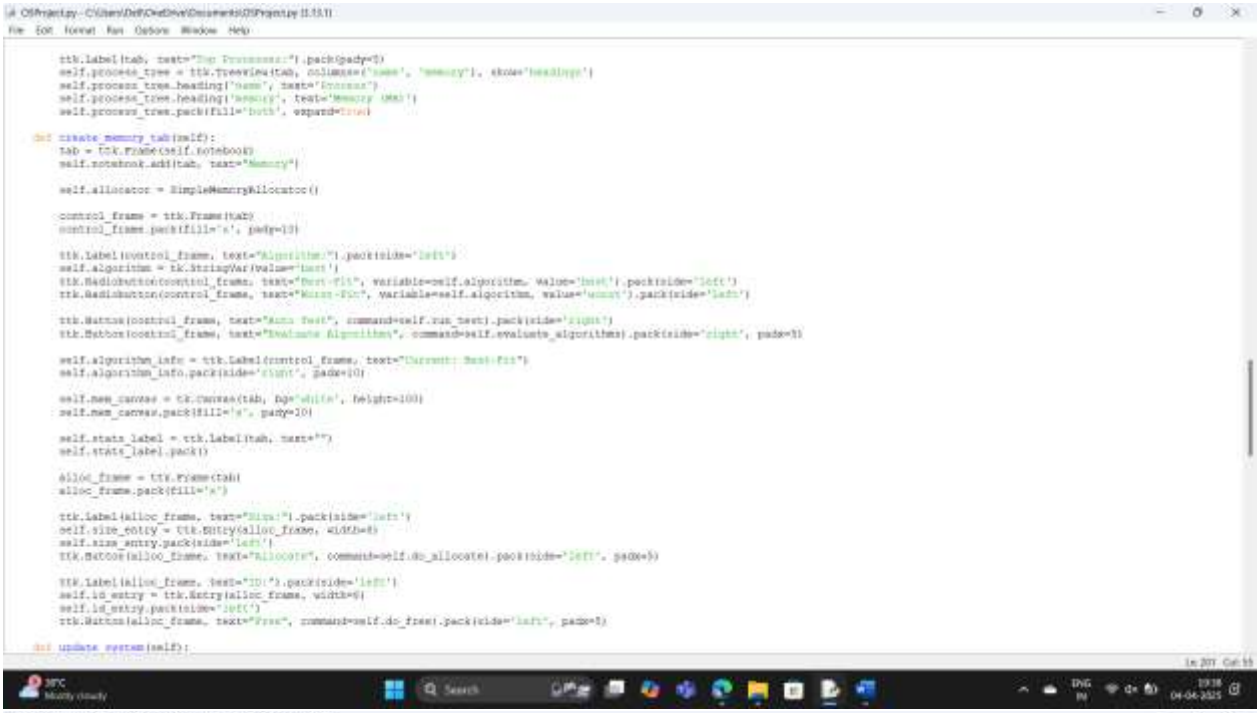
```
File  Edit  Format  Run  Options  Window  Help
        self.next_id += 1
        return alloc_id

    def free(self, alloc_id):
        if alloc_id not in self.allocations:
            return False

        start, size = self.allocations.pop(alloc_id)
        self.free_blocks.append((start, size))
        self._merge_blocks()
        return True

    def _merge_blocks(self):
        if not self.free_blocks:
            return

        self.free_blocks.sort()
        merged = [self.free_blocks[0]]

        for current in self.free_blocks[1:]:
            last = merged[-1]
            if last[0] + last[1] == current[0]:
                merged[-1] = (last[0], last[1] + current[1])
            else:
                merged.append(current)

        self.free_blocks = merged

    def get_stats(self):
        used = sum(size for _, size in self.allocations.values())
        free = sum(size for _, size in self.free_blocks)
        fragmentation = len(self.free_blocks) - 1 if len(self.free_blocks) > 1 else 0
        return {
            'total': self.total,
            'used': used,
            'free': free,
            'fragments': len(self.free_blocks),
            'allocations': len(self.allocations),
            'fragmentation': fragmentation
        }

    def evaluate_algorithm(self, algorithm):
        """Run a test and evaluate the algorithm's performance"""
        # Save current state
        original_allocations = self.allocations.copy()
        original_free_blocks = self.free_blocks.copy()
        original_next_id = self.next_id
```



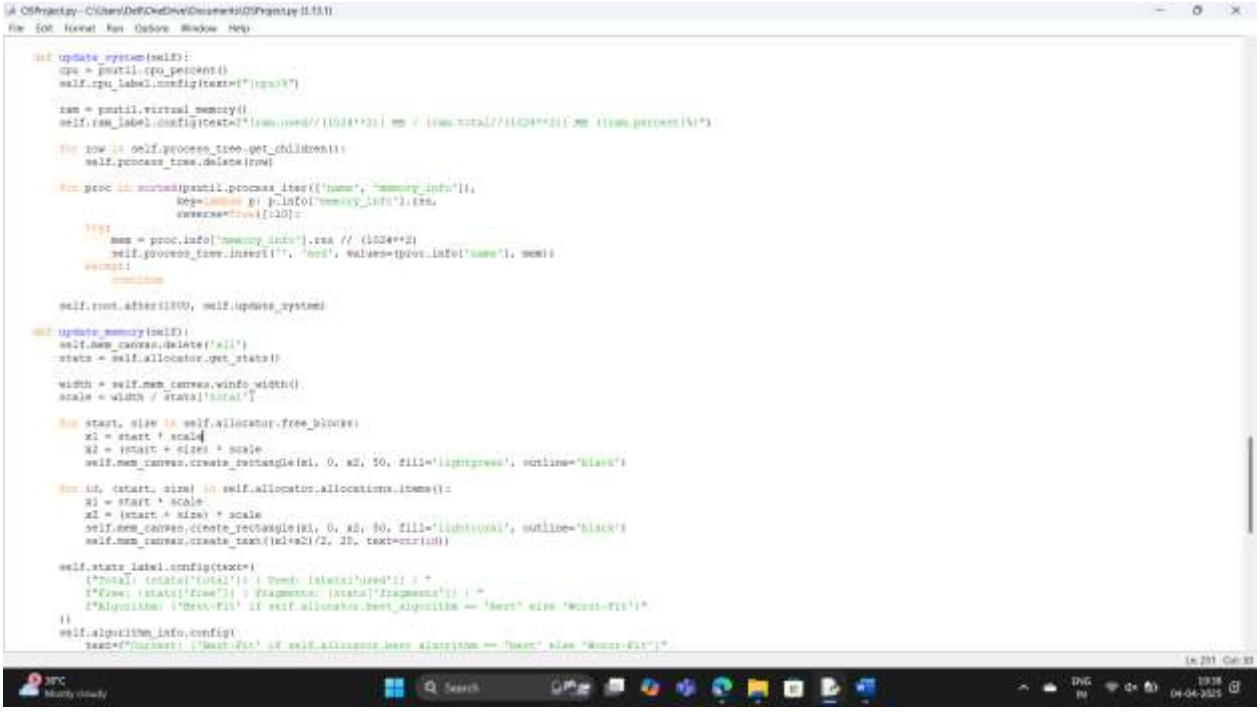
```
        original_free_blocks = self.free_blocks.copy()
        original_next_id = self.next_id

        # Run test allocations
        test_results = {'success': 0, 'failed': 0, 'total_fragmentation': 0}
        operations = 0

        for _ in range(20):  # Perform 20 operations (mix of alloc/free)
            if random.random() < 0.7 or not self.allocations:
                # Try to allocate
                size = random.randint(10, 100)
                if self.allocate(size, algorithm):
                    test_results['success'] += 1
                else:
                    test_results['failed'] += 1
            else:
                # Free a random allocation
                id = random.choice(list(self.allocations.keys()))
                self.free(id)

            stats = self.get_stats()
            test_results['total_fragmentation'] += stats['fragmentation']
            operations += 1

        # Calculate average fragmentation
        if operations > 0:
            avg_fragmentation = test_results['total_fragmentation'] / operations
        else:
            avg_fragmentation = 0

        # Restore original state
        self.allocations = original_allocations
        self.free_blocks = original_free_blocks
        self.next_id = original_next_id
        self._merge_blocks()

        return {
            'success_rate': test_results['success'] / (test_results['success'] + test_results['failed']) if (test_results['success'] + test_results['failed']) > 0 else 0,
            'avg_fragmentation': avg_fragmentation
        }

    def determine_best_algorithm(self):
        """Determine which algorithm performs better"""
        best_stats = self.evaluate_algorithm('best')
        worst_stats = self.evaluate_algorithm('worst')

        # Update algorithm statistics
```



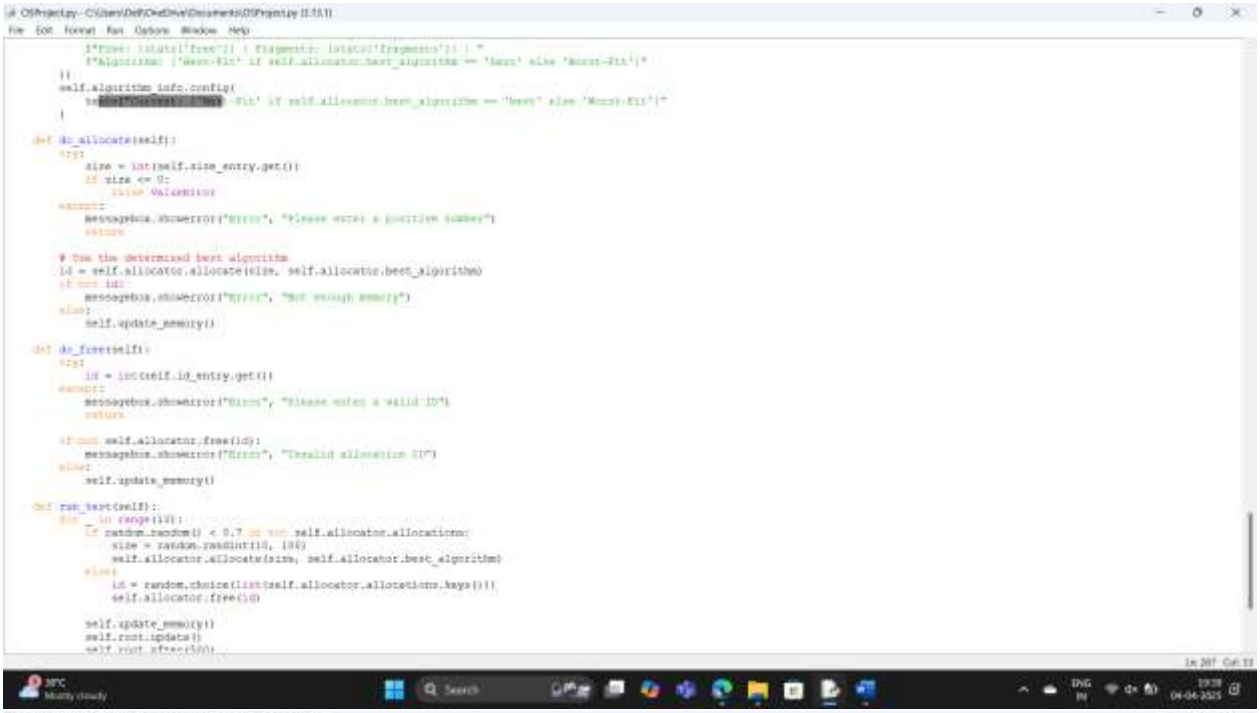
```
        # Update algorithm statistics
        self.algorithm_stats['best']['fragmentation'] += best_stats['avg_fragmentation']
        self.algorithm_stats['best']['tests'] += 1
        self.algorithm_stats['worst']['fragmentation'] += worst_stats['avg_fragmentation']
        self.algorithm_stats['worst']['tests'] += 1

        # Determine which is better (lower fragmentation is better)
        best_avg = self.algorithm_stats['best']['fragmentation'] / self.algorithm_stats['best']['tests'] if self.algorithm_stats['best']['tests'] > 0 else 0
        worst_avg = self.algorithm_stats['worst']['fragmentation'] / self.algorithm_stats['worst']['tests'] if self.algorithm_stats['worst']['tests'] > 0 else 0

        if best_avg <= worst_avg:
            self.best_algorithm = 'best'
        else:
            self.best_algorithm = 'worst'

        return self.best_algorithm, {
            'best_fragmentation': best_avg,
            'worst_fragmentation': worst_avg
        }

class SimplePerformanceApp:
    def __init__(self, root):
        self.root = root
        root.title("Smart Memory Allocator")
        root.geometry("1152x720")

        self.notebook = ttk.Notebook(root)
        self.notebook.pack(fill='both', expand=True)

        self.create_system_tab()
        self.create_memory_tab()
        self.update_system()

    def create_system_tab(self):
        tab = ttk.Frame(self.notebook)
        self.notebook.add(tab, text="System")

        ttk.Label(tab, text="CPU Usage:").pack(pady=5)
        self.cpu_label = ttk.Label(tab, text="0%")
        self.cpu_label.pack()

        ttk.Label(tab, text="RAM Usage:").pack(pady=5)
        self.ram_label = ttk.Label(tab, text="0 GB / 0 GB (0%)")
        self.ram_label.pack()

        ttk.Label(tab, text="Top Processes:").pack(pady=5)
```



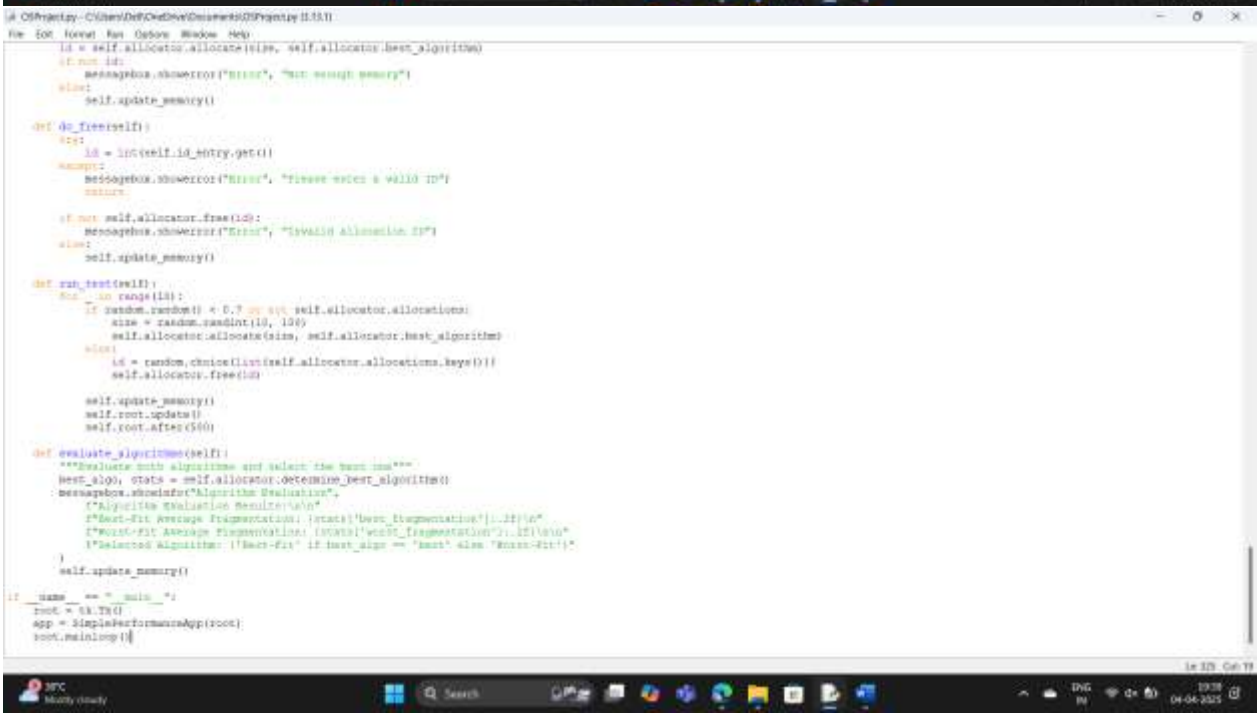
```
        ttk.Label(tab, text="Top Processes:").pack(pady=5)
        self.process_tree = ttk.Treeview(tab, columns=('name', 'memory'), show='headings')
        self.process_tree.heading('name', text='Process')
        self.process_tree.heading('memory', text='Memory (MB)')
        self.process_tree.pack(fill='both', expand=True)

    def create_memory_tab(self):
        tab = ttk.Frame(self.notebook)
        self.notebook.add(tab, text="Memory")

        self.allocator = SimpleMemoryAllocator()

        control_frame = ttk.Frame(tab)
        control_frame.pack(fill='x', pady=10)

        ttk.Label(control_frame, text="Algorithm:").pack(side='left')
        self.algorithm = tk.StringVar(value='best')
        ttk.Radiobutton(control_frame, text="Best-Fit", variable=self.algorithm, value='best').pack(side='left')
        ttk.Radiobutton(control_frame, text="Worst-Fit", variable=self.algorithm, value='worst').pack(side='left')

        ttk.Button(control_frame, text="Auto Test", command=self.run_test).pack(side='right')
        ttk.Button(control_frame, text="Evaluate Algorithms", command=self.evaluate_algorithms).pack(side='right', padx=5)

        self.algorithm_info = ttk.Label(control_frame, text="Current: Best-Fit")
        self.algorithm_info.pack(side='right', padx=10)

        self.mem_canvas = tk.Canvas(tab, bg='white', height=100)
        self.mem_canvas.pack(fill='x', pady=10)

        self.stats_label = ttk.Label(tab, text="")
        self.stats_label.pack()

        alloc_frame = ttk.Frame(tab)
        alloc_frame.pack(fill='x')

        ttk.Label(alloc_frame, text="Size:").pack(side='left')
        self.size_entry = ttk.Entry(alloc_frame, width=8)
        self.size_entry.pack(side='left')
        ttk.Button(alloc_frame, text="Allocate", command=self.do_allocate).pack(side='left', padx=5)

        ttk.Label(alloc_frame, text="ID:").pack(side='left')
        self.id_entry = ttk.Entry(alloc_frame, width=6)
        self.id_entry.pack(side='left')
        ttk.Button(alloc_frame, text="Free", command=self.do_free).pack(side='left', padx=5)

    def update_system(self):
```



```
    def update_system(self):
        cpu = psutil.cpu_percent()
        self.cpu_label.config(text=f"CPU: {cpu}%")

        ram = psutil.virtual_memory()
        self.ram_label.config(text=f"RAM: {ram.used//(1024**2)} MB / {ram.total//(1024**2)} MB ({ram.percent}%)")

        for row in self.process_tree.get_children():
            self.process_tree.delete(row)

        for proc in sorted(psutil.process_iter(['name', 'memory_info']),
                           key=lambda p: p.info['memory_info'].rss,
                           reverse=True)[:10]:
            try:
                mem = proc.info['memory_info'].rss // (1024**2)
                self.process_tree.insert('', 'end', values=(proc.info['name'], mem))
            except:
                continue

        self.root.after(1000, self.update_system)

    def update_memory(self):
        self.mem_canvas.delete('all')
        stats = self.allocator.get_stats()

        width = self.mem_canvas.winfo_width()
        scale = width / stats['total']

        for start, size in self.allocator.free_blocks:
            x1 = start * scale
            x2 = (start + size) * scale
            self.mem_canvas.create_rectangle(x1, 0, x2, 50, fill='lightgreen', outline='black')

        for id, (start, size) in self.allocator.allocations.items():
            x1 = start * scale
            x2 = (start + size) * scale
            self.mem_canvas.create_rectangle(x1, 0, x2, 50, fill='lightcoral', outline='black')
            self.mem_canvas.create_text((x1+x2)/2, 25, text=str(id))

        self.stats_label.config(text=
            f"Total: {stats['total']} | Used: {stats['used']} | "
            f"Free: {stats['free']} | Fragments: {stats['fragments']} | "
            f"Algorithm: {'Best-Fit' if self.allocator.best_algorithm == 'best' else 'Worst-Fit'}"
        )
        self.algorithm_info.config(
            text=f"Current: {'Best-Fit' if self.allocator.best_algorithm == 'best' else 'Worst-Fit'}"
```

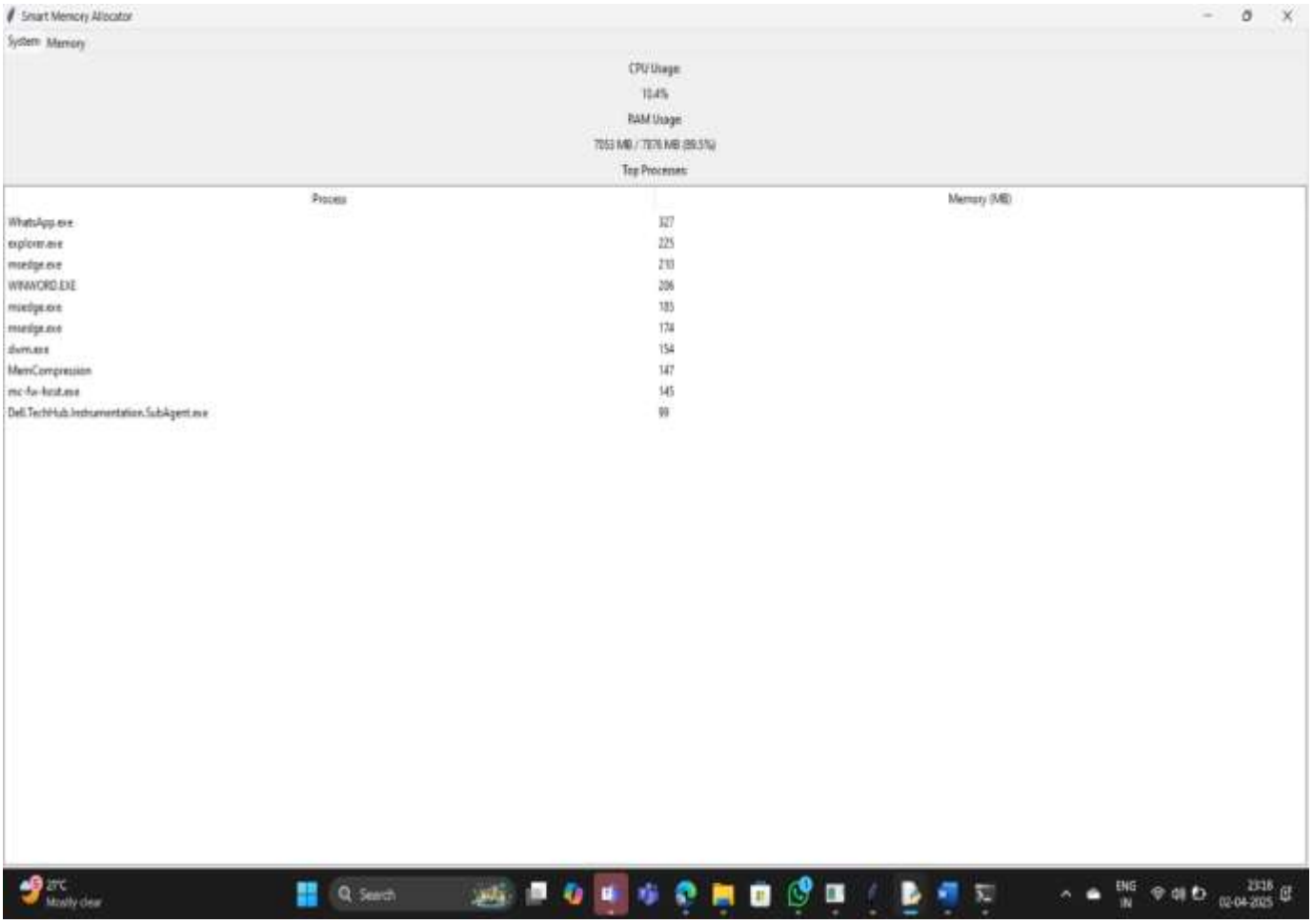


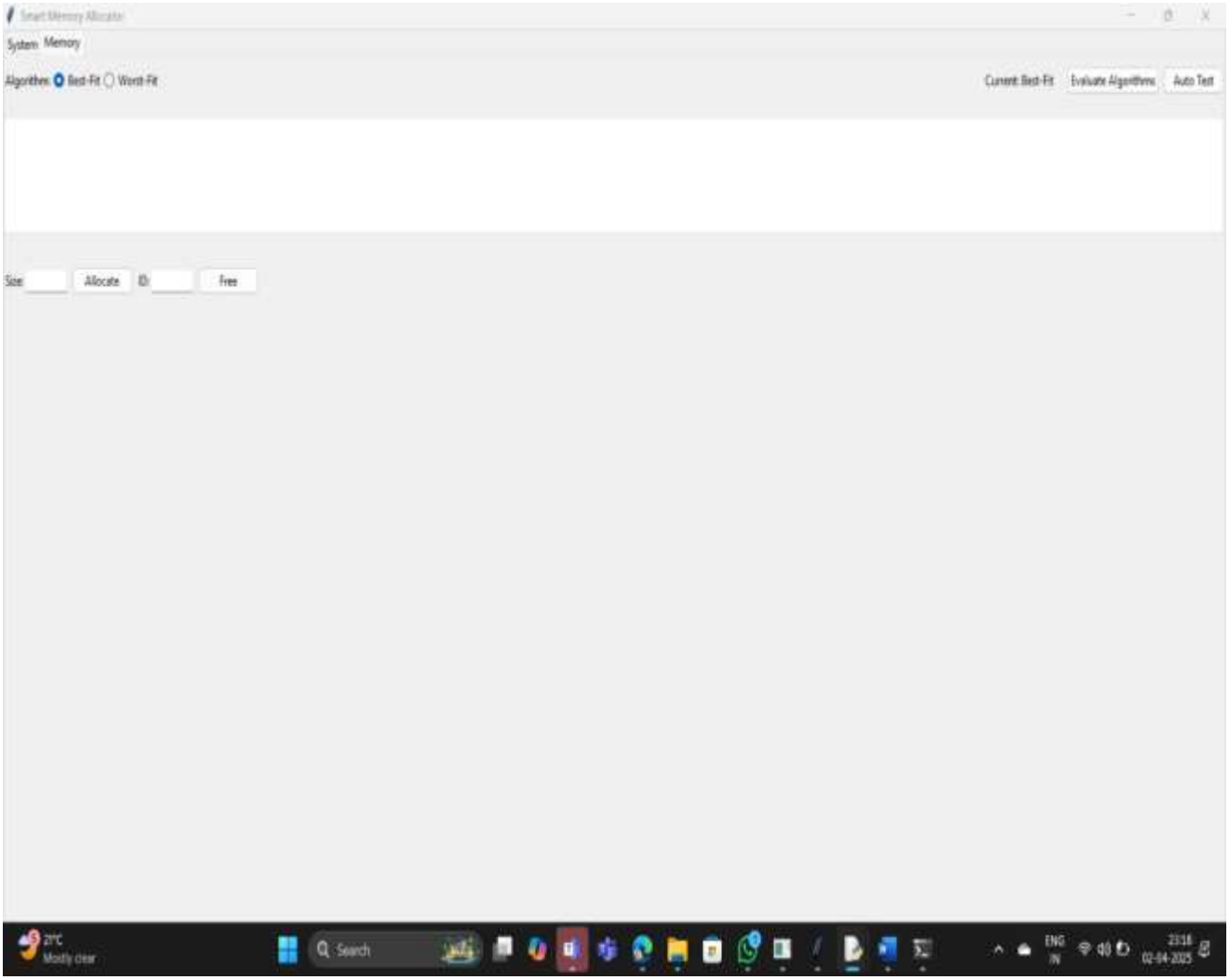
```
        f"Free: {stats['free']} | Fragments: {stats['fragments']} | "
        f"Algorithm: {'Best-Fit' if self.allocator.best_algorithm == 'best' else 'Worst-Fit'}"
    )
    self.algorithm_info.config(
        text="Best-Fit" if self.allocator.best_algorithm == "best" else "Worst-Fit"
    )

def do_allocate(self):
    try:
        size = int(self.size_entry.get())
        if size <= 0:
            raise ValueError
    except:
        messagebox.showerror("Error", "Please enter a positive number")
        return

    # Use the determined best algorithm
    id = self.allocator.allocate(size, self.allocator.best_algorithm)
    if not id:
        messagebox.showerror("Error", "Not enough memory")
    else:
        self.update_memory()

def do_free(self):
    try:
        id = int(self.id_entry.get())
    except:
        messagebox.showerror("Error", "Please enter a valid ID")
        return

    if not self.allocator.free(id):
        messagebox.showerror("Error", "Invalid allocation ID")
    else:
        self.update_memory()

def run_test(self):
    for _ in range(10):
        if random.random() < 0.7 or not self.allocator.allocations:
            size = random.randint(10, 100)
            self.allocator.allocate(size, self.allocator.best_algorithm)
        else:
            id = random.choice(list(self.allocator.allocations.keys()))
            self.allocator.free(id)

        self.update_memory()
        self.root.update()
        self.root.after(500)
```

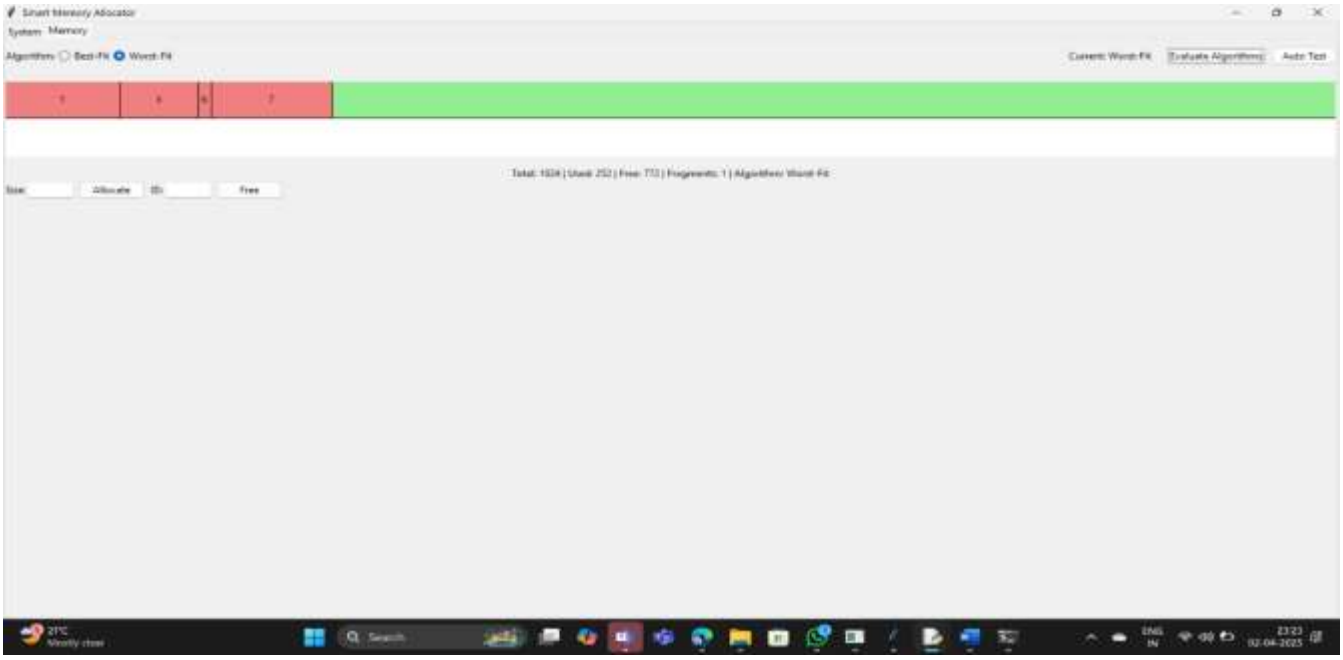


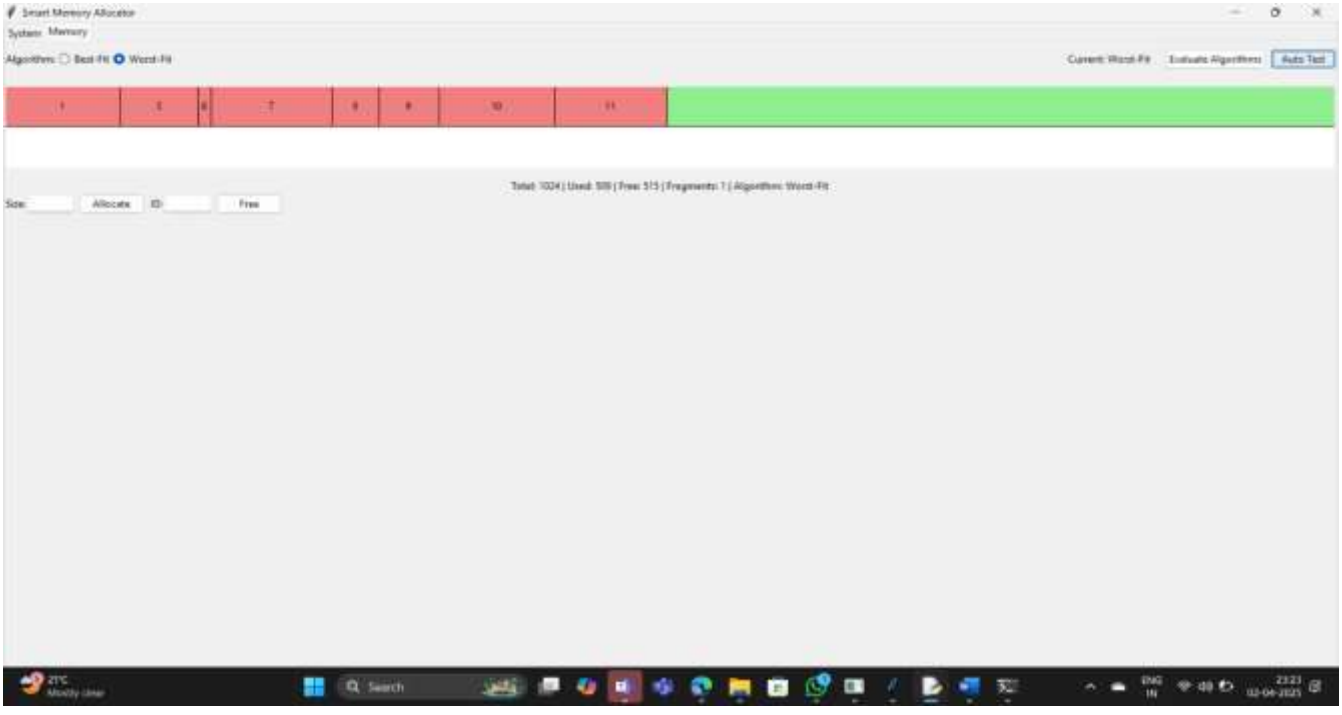
```
        id = self.allocator.allocate(size, self.allocator.best_algorithm)
        if not id:
            messagebox.showerror("Error", "Not enough memory")
        else:
            self.update_memory()

    def do_free(self):
        try:
            id = int(self.id_entry.get())
        except:
            messagebox.showerror("Error", "Please enter a valid ID")
            return

        if not self.allocator.free(id):
            messagebox.showerror("Error", "Invalid allocation ID")
        else:
            self.update_memory()

    def run_test(self):
        for _ in range(10):
            if random.random() < 0.7 or not self.allocator.allocations:
                size = random.randint(10, 100)
                self.allocator.allocate(size, self.allocator.best_algorithm)
            else:
                id = random.choice(list(self.allocator.allocations.keys()))
                self.allocator.free(id)

            self.update_memory()
            self.root.update()
            self.root.after(500)

    def evaluate_algorithms(self):
        """Evaluate both algorithms and select the best one"""
        best_algo, stats = self.allocator.determine_best_algorithm()
        messagebox.showinfo("Algorithm Evaluation",
            f"Algorithm Evaluation Results:\n\n"
            f"Best-Fit Average Fragmentation: {stats['best_fragmentation']:.2f}\n"
            f"Worst-Fit Average Fragmentation: {stats['worst_fragmentation']:.2f}\n\n"
            f"Selected Algorithm: {'Best-Fit' if best_algo == 'best' else 'Worst-Fit'}"
        )
        self.update_memory()

if __name__ == "__main__":
    root = tk.Tk()
    app = SimplePerformanceApp(root)
    root.mainloop()
```

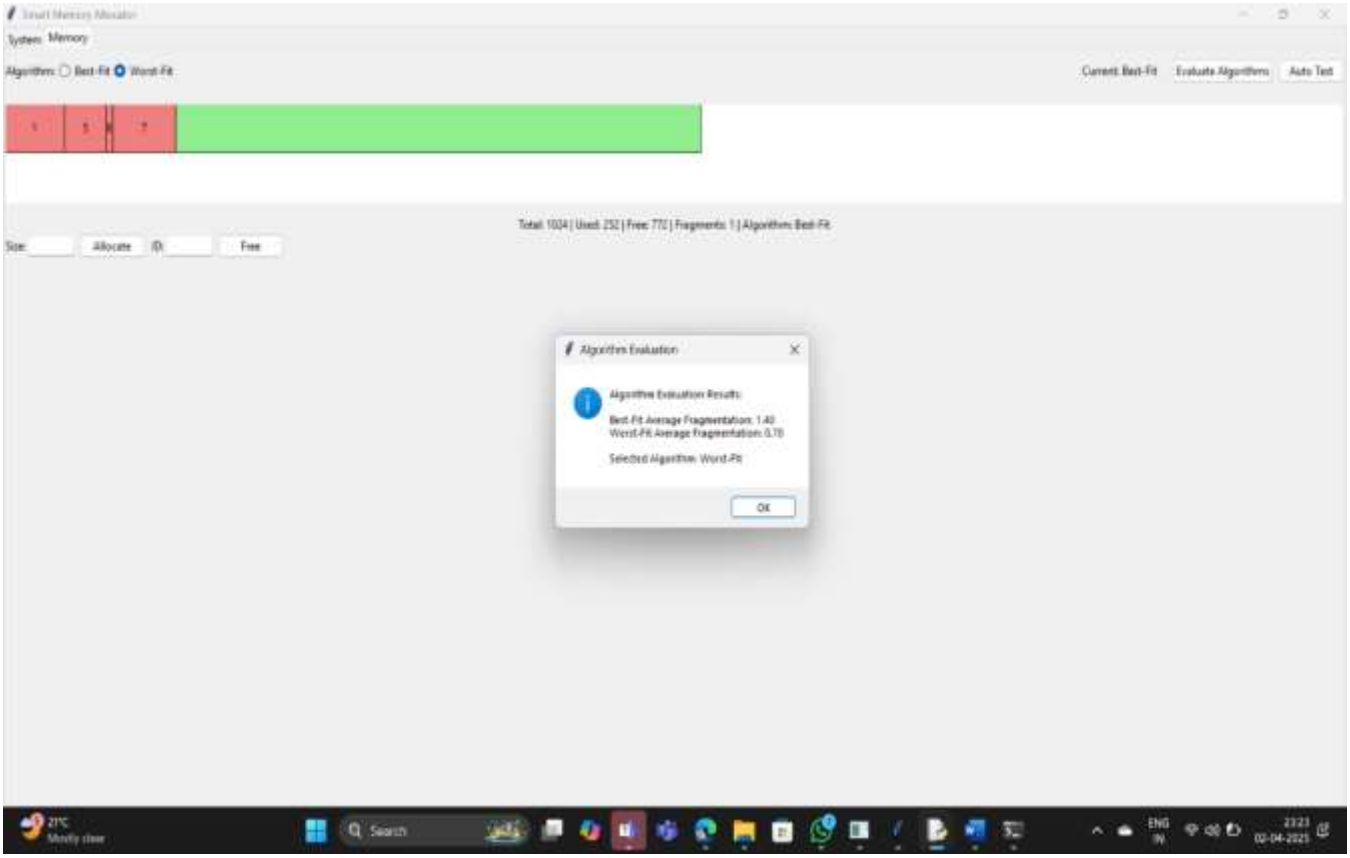
D. Real-time Results UI




Smart Memory Allocator
System Memory
Algorithm: Best-Fit Worst-Fit
Current: Best-Fit
Evaluate Algorithms
Auto Test
Size:
Allocate
ID:
Free
21°C
Mostly clear
Search
ENG
IN
23:18
02-04-2025


Smart Memory Allocator
System Memory
Algorithm: Best-Fit Worst-Fit
Current: Worst-Fit
Evaluate Algorithms
Auto Test
Size:
Allocate
ID:
Free


Smart Memory Allocator
System Memory
Algorithm: Best-Fit Worst-Fit
Current: Worst-Fit
Evaluate Algorithms
Auto Test
Size:
Allocate
ID:
Free


System: Memory
Algorithm: Best-Fit Worst-Fit
Current: Best-Fit
Evaluate Algorithms
Auto Test
Total: 1024 | Used: 252 | Free: 772 | Fragments: 1 | Algorithm: Best-Fit
Size:
Allocate
ID:
Free
Algorithm Evaluation
Algorithm Evaluation Results:
Best-Fit Average Fragmentation: 1.40
Worst-Fit Average Fragmentation: 0.70
Selected Algorithm: Worst-Fit
OK
Search