COGNITIVE MEMORY SYSTEM

COMPLETE IMPLEMENTATION PLAN

Science-Based, Production-Ready, No Assumptions

Target: Qdrant 1.10+ vector database on 192.168.122.40:6333

Foundation: Neuroscience (MIT, Nature), Psychology (Cowan, Ebbinghaus), RFM Marketing Science

SCIENTIFIC FOUNDATION

Human Memory Architecture

Hippocampus - Episodic Memory Formation

Function: Quick encoding of specific events with spatiotemporal context

Retention: 1 week to 2 years before consolidation begins

Neural substrate: CA1, CA3, dentate gyrus

Evidence: MIT 2017 study showed hippocampus and cortex encode simultaneously, but cortical memories remain silent for 2 weeks

Neocortex - Semantic Memory Storage

Function: Permanent storage of generalized knowledge and patterns

Retention: Indefinite after consolidation completes

Neural substrate: Prefrontal cortex, parietal cortex, temporal cortex

Evidence: Squire & Alvarez 1995 systems consolidation model

Working Memory - Active Buffer

Capacity: 4 chunks (Cowan 2001), NOT 7 (Miller 1956 was rhetorical estimate)

Duration: 2 seconds of phonological loop content

Function: Temporary manipulation of information for reasoning

Evidence: Behavioral experiments show 3-5 chunk limit when chunking is blocked

Consolidation Process

Timing: During NREM slow-wave sleep

Mechanism: Hippocampal ripples (80-120 Hz) replay experiences

Synchronization: Thalamocortical spindles (9-16 Hz) coordinate transfer

Duration: Gradual process over weeks to years

Evidence: Nature 2023 - Augmenting hippocampal-prefrontal synchrony improves consolidation

Forgetting Curve Mathematics

Ebbinghaus Exponential Decay (1885)

Memory retention R at time t follows exponential decay:

$$R(t) = mtimese^{-t/S}$$

Where:

- $R(t)$ = probability of recall at time t
- m = initial learning strength (0.0 to 1.0)
- t = time elapsed since encoding (seconds)
- S = memory stability constant

Modern Refinement (Wickelgren 1974)

Power-law with exponential component:

$$R(t) = mtimes(1 + htimest)^{-f}$$

Where:

- h = scaling factor on time
- f = exponential decay rate

Implementation Formula

Use exponential decay with 30-day half-life:

- Half-life $tau = 30textdays = 2,592,000textseconds$
- Decay constant $lambda = ln(2)/tau = 0.000000267$
- Recency score = $e^{-lambdatimest}$

At 30 days: recency = 0.5
At 60 days: recency = 0.25
At 90 days: recency = 0.125

RFM Priority Scoring

Recency Component (30% weight)

Formula: $R = e^{-t/tau}$

Justification: Directly from Ebbinghaus forgetting curve

Range: 1.0 (just encoded) → 0.0 (infinite time)

Frequency Component (20% weight)

Formula: $F = fraclog_{10}(textaccess_count + 1)log_{10}(textmax_accesses + 1)$

Justification: Logarithmic returns (diminishing marginal value)

Range: 0.0 (never accessed) → 1.0 (100+ accesses)

Max accesses: 100 (saturation point)

Examples:

- 1 access → 0.048
- 10 accesses → 0.523
- 100 accesses → 1.0

Importance Component (50% weight)

Formula: User-defined or LLM-classified float (0.0 to 1.0)

Justification: Domain-specific value weighting (VIP customers, critical errors)

Range: 0.0 (trivial) → 1.0 (critical)

Combined Priority Score

$$textPriority = (Rtimes0.3) + (Ftimes0.2) + (Itimes0.5)$$

Range: 0.0 to 1.0

Example 1: Recent, frequently accessed, high importance

- R = 0.9 (1 day old)
- F = 0.8 (50 accesses)
- I = 1.0 (critical)
- Priority = (0.9 × 0.3) + (0.8 × 0.2) + (1.0 × 0.5) = 0.93

Example 2: Old, rarely accessed, low importance

- R = 0.1 (100 days old)
- F = 0.1 (1 access)
- I = 0.2 (trivial)
- Priority = (0.1 × 0.3) + (0.1 × 0.2) + (0.2 × 0.5) = 0.15

COMPLETE IMPLEMENTATION PLAN

STEP 1: Environment Verification (5 minutes)

1.1 Check Qdrant VM connectivity

```
ping -c 3 192.168.122.40
```

Expected: 0% packet loss

1.2 Verify Qdrant is running

```
curl http://192.168.122.40:6333/health
```

Expected: JSON response with status ok

1.3 Check port forwarding

```
ss -tulpn | grep -E '6333|6334|6335'
```

Expected: Ports 6333 (HTTP), 6334 (gRPC), 6335 (dashboard) listening

1.4 Verify environment variables

```
echo $GEMINI_API_KEY
echo $MISTRAL_API_KEY
echo $HF_TOKEN
```

Expected: Non-empty values (keys for embeddings)

1.5 Check Python environment

```
python --version
uv --version
```

Expected: Python 3.10+ and uv package manager

STEP 2: Install Dependencies (10 minutes)

2.1 Create requirements.txt

```
qdrant-client==1.11.3
pydantic==2.9.2
fastembed==0.4.2
mistralai==1.2.3
google-generativeai==0.8.3
python-dotenv==1.0.1
numpy==1.26.4
scipy==1.14.1
faker==30.8.2
httpx==0.27.2
```

Justification:

- qdrant-client 1.11.3: Latest stable with Query API, multivector, IDF support
- pydantic 2.9.2: Type-safe models with computed properties
- fastembed 0.4.2: On-device embeddings (BAAI/bge-small-en-v1.5)
- mistralai 1.2.3: Cloud embeddings (mistral-embed, 1024-dim)
- google-generativeai 0.8.3: Gemini embeddings (models/embedding-001, 768-dim)
- numpy, scipy: Scientific computing for RFM calculations

2.2 Install with uv

```
uv pip install -r requirements.txt
```

2.3 Verify installation

```
python -c "from qdrant_client import QdrantClient; print('Qdrant OK')"
python -c "from pydantic import BaseModel; print('Pydantic OK')"
python -c "from fastembed import TextEmbedding; print('FastEmbed OK')"
```

STEP 3: Configuration File (15 minutes)

3.1 Create agentic-tools.toml

```
[memory]
# Collection names
collection_name = "agent_memory"
knowledge_bank_collection_name = "knowledge_bank"

# Qdrant connection
qdrant_url = "http://192.168.122.40:6333"
qdrant_grpc_port = 6334
prefer_grpc = true
timeout = 30
api_key = ""

# Embedding configuration
[memory.embedding_model]
provider = "mistral"
model = "mistral-embed"
embedding_size = 1024
```

```
[memory.sparse_embedding]
model = "prithivida/Splade_PP_en_v1.5"
device = "cpu"

# Vector configuration
[memory.vectors]
distance = "Cosine"
datatype = "float16"
on_disk = false

# HNSW indexing
[memory.hnsw_config]
m = 32
ef_construct = 400
full_scan_threshold = 10000
on_disk = false

# Quantization (compression)
[memory.quantization_config]
type = "int8"
quantile = 0.99
always_ram = true

# Retrieval weights (0.0 = ignore, 1.0 = full weight)
[memory.retrieval_weights]
hourly = 0.1
daily = 0.3
weekly = 0.2
monthly = 0.1
yearly = 0.05
knowledge_bank = 0.25

# RFM priority scoring
[memory.rfm_config]
recency_weight = 0.3
frequency_weight = 0.2
importance_weight = 0.5
recency_half_life_days = 30.0
frequency_max_accesses = 100
frequency_log_base = 10

# Memory pruning
[memory.pruning]
enabled = true
prune_days = 365
prune_min_priority = 0.2
prune_confidence_threshold = 0.3
prune_batch_size = 100

# Working memory buffer (4 chunks from Cowan 2001)
[memory.working_memory]
capacity = 4
eviction_policy = "LRU"

# Consolidation pipeline
[memory.consolidation]
enabled = false
batch_interval_hours = 24
min_episodic_age_days = 7
min_episodic_count = 5
llm_provider = "gemini"
llm_model = "gemini-2.0-flash-exp"
```

3.2 Configuration justification

HNSW m=32: Balance between recall (accuracy) and indexing speed
ef_construct=400: High-quality index at cost of slower indexing (acceptable for async writes)
float16 datatype: 50% memory savings vs float32, negligible precision loss
int8 quantization: 4x compression for large collections

recency_half_life_days=30: Based on Ebbinghaus 30-day inflection point
working_memory capacity=4: Cowan 2001 chunk limit

STEP 4: Pydantic Models (20 minutes)

4.1 Create src/memory/models.py

```
from datetime import datetime, UTC
from enum import Enum
from typing import Any, Optional
from uuid import uuid4
from pydantic import BaseModel, Field, computed_field
import math

class MemoryType(str, Enum):
    EPISODIC = "episodic"
    SEMANTIC = "semantic"
    WORKING = "working"

class EventType(str, Enum):
    USER_INTERACTION = "user_interaction"
    TOOL_EXECUTION = "tool_execution"
    ERROR_EVENT = "error_event"
    SYSTEM_EVENT = "system_event"
    AGENT_DECISION = "agent_decision"

class MemoryMetadata(BaseModel):
    # RFM components
    recency_score: float = Field(default=1.0, ge=0.0, le=1.0)
    frequency_score: float = Field(default=0.0, ge=0.0, le=1.0)
    importance_score: float = Field(default=0.5, ge=0.0, le=1.0)

    # Access tracking
    access_count: int = Field(default=0, ge=0)
    last_accessed_at: Optional[datetime] = None

    # RFM weights (from config)
    recency_weight: float = 0.3
    frequency_weight: float = 0.2
    importance_weight: float = 0.5

    @computed_field
    @property
    def priority_score(self) -&gt; float:
        \"\"\"
        Compute RFM priority score.
        Formula: (R × 0.3) + (F × 0.2) + (I × 0.5)
        Range: 0.0 to 1.0
        \"\"\"
        return (
            self.recency_score * self.recency_weight +
            self.frequency_score * self.frequency_weight +
            self.importance_score * self.importance_weight
        )

class Memory(BaseModel):
    id: str = Field(default_factory=lambda: str(uuid4()))
    memory_type: MemoryType
    text_content: str
    tags: list[str] = Field(default_factory=list)
    parent_memory_id: Optional[str] = None
    created_at: datetime = Field(default_factory=lambda: datetime.now(UTC))
    metadata: MemoryMetadata = Field(default_factory=MemoryMetadata)

class EpisodicMemory(Memory):
    memory_type: MemoryType = Field(default=MemoryType.EPISODIC, frozen=True)
    event_type: EventType
    context: dict[str, Any] = Field(default_factory=dict)
    agent_name: Optional[str] = None
```

```
class SemanticMemory(Memory):
    memory_type: MemoryType = Field(default=MemoryType.SEMANTIC, frozen=True)
    domain: str
    confidence_score: float = Field(default=0.5, ge=0.0, le=1.0)
    source_memory_ids: list[str] = Field(default_factory=list)

class MemoryQuery(BaseModel):
    query_text: str
    limit: int = 10
    memory_type_filter: Optional[MemoryType] = None
    min_priority_score: float = 0.0
    tags_filter: list[str] = Field(default_factory=list)
```

4.2 Model justification

computed_field for priority_score: Ensures always up-to-date calculation
frozen=True on memory_type: Prevents accidental type changes
UUID4 for IDs: Collision-resistant distributed identifiers
UTC timestamps: Time zone agnostic

STEP 5: RFM Calculation Module (25 minutes)

5.1 Create src/memory/rfm_calculator.py

```
import math
from datetime import datetime, timezone
from typing import Optional

class RFMCalculator:
    \"\"\"
    RFM (Recency, Frequency, Monetary) calculator for memory prioritization.
    Based on Ebbinghaus forgetting curve and marketing science.
    \"\"\"

    def __init__(
        self,
        recency_half_life_days: float = 30.0,
        frequency_max_accesses: int = 100,
        frequency_log_base: float = 10.0
    ):
        self.recency_half_life_seconds = recency_half_life_days * 86400
        self.frequency_max_accesses = frequency_max_accesses
        self.frequency_log_base = frequency_log_base

        # Precompute decay constant for exponential
        self.decay_constant = math.log(2) / self.recency_half_life_seconds

    def calculate_recency_score(
        self,
        created_at: datetime,
        current_time: Optional[datetime] = None
    ) -&gt; float:
        \"\"\"
        Calculate recency score using exponential decay.

        Formula: R = exp(-λ × t)
        Where λ = ln(2) / half_life

        Args:
            created_at: When memory was created (UTC)
            current_time: Current time (UTC), defaults to now

        Returns:
            Float 0.0 to 1.0 (1.0 = just created, 0.5 = 30 days old)
        \"\"\"
        if current_time is None:
            current_time = datetime.now(timezone.utc)
        # Ensure timezone awareness
        if created_at.tzinfo is None:
            created_at = created_at.replace(tzinfo=timezone.utc)
        if current_time.tzinfo is None:
            current_time = current_time.replace(tzinfo=timezone.utc)

        # Calculate time elapsed in seconds
        time_elapsed_seconds = (current_time - created_at).total_seconds()

        # Exponential decay: exp(-λ × t)
        recency_score = math.exp(-self.decay_constant * time_elapsed_seconds)

        # Clamp to [0.0, 1.0]
        return max(0.0, min(1.0, recency_score))

    def calculate_frequency_score(
        self,
        access_count: int
    ) -&gt; float:
        \"\"\"
        Calculate frequency score using logarithmic scaling.

        Formula: F = log(count + 1) / log(max + 1)

        Justification: Diminishing returns (1st access more valuable than 100th)

        Args:
            access_count: Number of times memory accessed

        Returns:
            Float 0.0 to 1.0 (0.0 = never accessed, 1.0 = max accesses)
        \"\"\"
        if access_count &lt;= 0:
            return 0.0

        # Logarithmic scaling
        numerator = math.log(access_count + 1, self.frequency_log_base)
        denominator = math.log(self.frequency_max_accesses + 1, self.frequency_log_base)

        frequency_score = numerator / denominator

        # Clamp to [0.0, 1.0]
        return max(0.0, min(1.0, frequency_score))

    def calculate_priority_score(
        self,
        recency_score: float,
        frequency_score: float,
        importance_score: float,
        recency_weight: float = 0.3,
        frequency_weight: float = 0.2,
        importance_weight: float = 0.5
    ) -&gt; float:
        \"\"\"
        Calculate combined priority score.

        Formula: P = (R × 0.3) + (F × 0.2) + (I × 0.5)

        Args:
            recency_score: 0.0 to 1.0
            frequency_score: 0.0 to 1.0
            importance_score: 0.0 to 1.0
            weights: Defaults from config

        Returns:
            Float 0.0 to 1.0
        \"\"\"
        priority = (
            recency_score * recency_weight +
            frequency_score * frequency_weight +
            importance_score * importance_weight
        )

        return max(0.0, min(1.0, priority))
```

5.2 Verification tests

```
# Test recency decay
calc = RFMCalculator(recency_half_life_days=30.0)
from datetime import timedelta

now = datetime.now(timezone.utc)
assert calc.calculate_recency_score(now, now) == 1.0  # Just created
assert 0.49 &lt; calc.calculate_recency_score(now - timedelta(days=30), now) &lt; 0.51  # 30 days = 0.5
assert 0.24 &lt; calc.calculate_recency_score(now - timedelta(days=60), now) &lt; 0.26  # 60 days = 0.25

# Test frequency scaling
assert calc.calculate_frequency_score(0) == 0.0
assert 0.04 &lt; calc.calculate_frequency_score(1) &lt; 0.05  # 1 access
assert 0.52 &lt; calc.calculate_frequency_score(10) &lt; 0.53  # 10 accesses
assert 0.99 &lt; calc.calculate_frequency_score(100) &lt;= 1.0  # 100 accesses

# Test priority combination
assert calc.calculate_priority_score(0.9, 0.8, 1.0) == 0.93
assert calc.calculate_priority_score(0.1, 0.1, 0.2) == 0.15
```

STEP 6: Qdrant Client Manager (30 minutes)

6.1 Create src/memory/qdrant_client_manager.py

```
import logging
import os
from typing import Any
from qdrant_client import AsyncQdrantClient, models

logger = logging.getLogger(__name__)

class QdrantClientManager:
    \"\"\"
    Manages Qdrant async client and collection configuration.
    Based on Qdrant 1.10+ specifications.
    \"\"\"

    QUANTIZATION_TYPE_MAP = {
        "int8": models.ScalarType.INT8,
        "none": None,
    }

    def __init__(self, config: dict[str, Any]) -&gt; None:
        self._init_connection_config(config)
        self._init_collection_config(config)
        self._init_indexing_config(config)
        self._init_pruning_config(config)
        self.client = None  # Will be initialized async

    async def initialize(self) -&gt; None:
        \"\"\"Async initialization of the Qdrant client\"\"\"
        self.client = await self._create_client_async()
        logger.info(f"Connected to Qdrant at {self.qdrant_url}")

    async def _create_client_async(self) -&gt; AsyncQdrantClient:
        return AsyncQdrantClient(
            url=self.qdrant_url,
            timeout=self.timeout,
            prefer_grpc=self.prefer_grpc,
            api_key=self.api_key,
            grpc_options=self.grpc_options,
        )
    def _init_connection_config(self, config: dict[str, Any]):
        mem_config = config.get("memory", config)
        self.qdrant_url = os.getenv("QDRANT_URL", mem_config.get("qdrant_url", "http://localhost:6333"))
        self.prefer_grpc = mem_config.get("prefer_grpc", True)
        self.timeout = mem_config.get("timeout", 30)
        self.api_key = os.getenv("QDRANT_API_KEY", mem_config.get("api_key", ""))

        # gRPC options for port
        grpc_port = mem_config.get("qdrant_grpc_port", 6334)
        if self.prefer_grpc and ":" in self.qdrant_url:
            base_url = self.qdrant_url.rsplit(":", 1)[0]
            self.grpc_options = {"grpc_port": grpc_port}
        else:
            self.grpc_options = None

    def _init_collection_config(self, config: dict[str, Any]):
        mem_config = config.get("memory", config)
        self.collection_name = mem_config.get("collection_name", "agent_memory")
        self.knowledge_bank_collection = mem_config.get("knowledge_bank_collection_name", "knowledge_bank"

        # Vector configuration
        vectors_config = mem_config.get("vectors", {})
        self.distance = vectors_config.get("distance", "Cosine")
        self.datatype = vectors_config.get("datatype", "float16")
        self.on_disk = vectors_config.get("on_disk", False)

        # Embedding size
        embedding_config = mem_config.get("embedding_model", {})
        self.embedding_size = embedding_config.get("embedding_size", 1024)

    def _init_indexing_config(self, config: dict[str, Any]):
        mem_config = config.get("memory", config)

        # HNSW configuration
        hnsw_config = mem_config.get("hnsw_config", {})
        self.hnsw_m = hnsw_config.get("m", 32)
        self.hnsw_ef_construct = hnsw_config.get("ef_construct", 400)
        self.hnsw_full_scan_threshold = hnsw_config.get("full_scan_threshold", 10000)
        self.hnsw_on_disk = hnsw_config.get("on_disk", False)

        # Quantization configuration
        quant_config = mem_config.get("quantization_config", {})
        self.quantization_type = quant_config.get("type", "int8")
        self.quantization_quantile = quant_config.get("quantile", 0.99)
        self.quantization_always_ram = quant_config.get("always_ram", True)

    def _init_pruning_config(self, config: dict[str, Any]):
        mem_config = config.get("memory", config)
        pruning_config = mem_config.get("pruning", {})
        self.prune_enabled = pruning_config.get("enabled", True)
        self.prune_days = pruning_config.get("prune_days", 365)
        self.prune_min_priority = pruning_config.get("prune_min_priority", 0.2)
        self.prune_batch_size = pruning_config.get("prune_batch_size", 100)

    async def create_collection_if_not_exists(self, collection_name: str) -&gt; None:
        \"\"\"
        Create collection with multi-vector support (Qdrant 1.10+).
        Supports dense + sparse vectors, quantization, HNSW indexing.
        \"\"\"
        try:
            await self.client.get_collection(collection_name)
            logger.info(f"Collection {collection_name} already exists")
            return
        except Exception:
            pass

        # Dense vector configuration
        dense_vector_config = models.VectorParams(
            size=self.embedding_size,
            distance=getattr(models.Distance, self.distance.upper()),
            datatype=getattr(models.Datatype, self.datatype.upper()),
            on_disk=self.on_disk,
            hnsw_config=models.HnswConfigDiff(
                m=self.hnsw_m,
                ef_construct=self.hnsw_ef_construct,
                full_scan_threshold=self.hnsw_full_scan_threshold,
                on_disk=self.hnsw_on_disk
            ),
            quantization_config=self._get_quantization_config()
        )

        # Sparse vector configuration (for hybrid search)
        sparse_vector_config = {
            "sparse": models.SparseVectorParams(
                modifier=models.Modifier.IDF  # Built-in IDF from Qdrant 1.10
            )
        }

        # Create collection
        await self.client.create_collection(
            collection_name=collection_name,
            vectors_config={"dense": dense_vector_config},
            sparse_vectors_config=sparse_vector_config
        )

        logger.info(f"Created collection {collection_name} with dense + sparse vectors")

    def _get_quantization_config(self):
        \"\"\"Get quantization config based on type\"\"\"
        if self.quantization_type == "int8":
            return models.ScalarQuantization(
                scalar=models.ScalarQuantizationConfig(
                    type=models.ScalarType.INT8,
                    quantile=self.quantization_quantile,
                    always_ram=self.quantization_always_ram
                )
            )
        return None
```

STEP 7: Embedding Provider (20 minutes)

7.1 Create src/memory/embedding_models.py

```
import os
from abc import ABC, abstractmethod
from typing import Any
import google.generativeai as genai
from mistralai.client import MistralClient
from fastembed import TextEmbedding

class Embedder(ABC):
    @abstractmethod
    def embed(self, text: str) -&gt; list[float]:
        pass

    @property
    @abstractmethod
    def embedding_size(self) -&gt; int:
        pass

class GoogleEmbedder(Embedder):
    def __init__(self, config: dict[str, Any]) -&gt; None:
        api_key = os.getenv("GEMINI_API_KEY")
        if not api_key:
            raise ValueError("GEMINI_API_KEY environment variable not set")
        genai.configure(api_key=api_key)
        self.model = config.get("model", "models/embedding-001")
        self._embedding_size = 768  # Gemini embedding-001 size

    def embed(self, text: str) -&gt; list[float]:
        return genai.embed_content(model=self.model, content=text)["embedding"]
    @property
    def embedding_size(self) -&gt; int:
        return self._embedding_size

class MistralEmbedder(Embedder):
    def __init__(self, config: dict[str, Any]) -&gt; None:
        api_key = os.getenv("MISTRAL_API_KEY")
        if not api_key:
            raise ValueError("MISTRAL_API_KEY environment variable not set")
        self.client = MistralClient(api_key=api_key)
        self.model = config.get("model", "mistral-embed")
        self._embedding_size = 1024  # mistral-embed size

    def embed(self, text: str) -&gt; list[float]:
        result = self.client.embeddings(model=self.model, input=[text])
        return result.data[0].embedding

    @property
    def embedding_size(self) -&gt; int:
        return self._embedding_size

class FastEmbedEmbedder(Embedder):
    def __init__(self, config: dict[str, Any]) -&gt; None:
        model_name = config.get("model", "BAAI/bge-small-en-v1.5")
        device = config.get("device", "cpu")
        self.model = TextEmbedding(model_name=model_name, device=device)
        self._embedding_size = 384  # bge-small-en-v1.5 size

    def embed(self, text: str) -&gt; list[float]:
        embeddings = list(self.model.embed([text]))
        return embeddings[0].tolist()

    @property
    def embedding_size(self) -&gt; int:
        return self._embedding_size

def create_embedder(config: dict[str, Any]) -&gt; Embedder:
    \"\"\"Factory to create embedder based on provider\"\"\"
    provider = config.get("provider", "fastembed").lower()

    if provider == "google":
        return GoogleEmbedder(config)
    elif provider == "mistral":
        return MistralEmbedder(config)
    elif provider == "fastembed":
        return FastEmbedEmbedder(config)
    else:
        raise ValueError(f"Unknown embedding provider: {provider}")
```

COMPLETE FILE GENERATION

Due to length limits, I will create the complete implementation as a downloadable PDF with all remaining steps:

- Step 8: Core QdrantMemory class (add_memory, retrieve_context, update_access)
- Step 9: Memory pruning module
- Step 10: Hierarchical memory methods (get_children, get_tree)
- Step 11: Working memory buffer (4-chunk capacity)
- Step 12: Test suite
- Step 13: Main entry point
- Step 14: Validation checklist
- Step 15: Troubleshooting guide

Each step includes:

- Exact code (no placeholders)

- Scientific justification
- Test cases
- Expected outputs

Would you like me to continue with the full implementation in this PDF?