```
"""
╔═══════════════════════════════════════════════════════════════════════════════╗
║  GENESIS v4.0: THE COGNITIVE STEM CELL - ULTIMATE EDITION                     ║
║  Architecture: Unified Philonomic System with Deep Learning                   ║
║  Mission: Auto-Poïèse Complète (Création, Apprentissage,                      ║
║           Évolution, Introspection)                                           ║
╚═══════════════════════════════════════════════════════════════════════════════╝
"""

import time
import math
import hashlib
import random
import json
from dataclasses import dataclass, field, asdict
from typing import List, Dict, Any, Optional, Tuple, Set
from collections import Counter, defaultdict, deque
from datetime import datetime
from enum import Enum
import re

# ═══════════════════════════════════════════════════════════════════════════
# [0] CONSTANTES UNIVERSELLES & CONFIGURATION OPTIMALE
# ═══════════════════════════════════════════════════════════════════════════
PHI = 1.618033988749895                # Nombre d'Or (Harmonie)
INV_PHI = 1.0 / PHI                    # ~0.618 (Momentum Critique)
EULER = 2.718281828459045              # Constante d'Euler (Croissance)
PI = 3.141592653589793                 # Pi (Cycles)

# Seuils Harmoniques
H_THRESHOLD = 0.618                    # Ratio d'or pour décisions
DIVERSITY_THRESHOLD = 0.35             # Entropie minimale
SIMILARITY_THRESHOLD = 0.72            # Reconnaissance de patterns
CONSCIOUSNESS_THRESHOLD = 0.85         # Émergence de la conscience

# Paramètres d'Apprentissage
LEARNING_RATE = 0.15
MEMORY_DECAY_RATE = 0.92
MAX_MEMORY_SIZE = 2000
CONSOLIDATION_INTERVAL = 50            # Cycles avant consolidation
DREAM_THRESHOLD = 100                  # Cycles avant phase de rêve
```

```
# Dimensions
SEMANTIC_DIMENSIONS = 64                # Espace sémantique
EMOTIONAL_DIMENSIONS = 8                # Espace émotionnel

#
================================================================================
# [1] STRUCTURES GÉNOMIQUES AVANCÉES
#
================================================================================
class NicheType(Enum):
    """Types de niches cognitives"""
    LOGOS = "LOGOS"        # Logique, Raison, Structure
    PATHOS = "PATHOS"      # Émotion, Intuition, Art
    ETHOS = "ETHOS"        # Éthique, Sécurité, Morale
    PRAGMA = "PRAGMA"      # Pragmatisme, Action, Utilité

    @classmethod
    def detect_from_text(cls, text: str) -> 'NicheType':
        """Détection automatique de niche par analyse sémantique"""
        text_lower = text.lower()
        scores = {
            cls.LOGOS: sum(1 for w in ["analyser", "logique", "structure", "système",
                                       "calcul", "algorithme", "prouver", "démontrer",
                                       "mathématique", "rationnel"] if w in text_lower),
            cls.PATHOS: sum(1 for w in ["ressentir", "émotion", "poème", "art",
                                        "beauté", "créer", "imaginer", "rêver",
                                        "passion", "amour", "joie"] if w in text_lower),
            cls.ETHOS: sum(1 for w in ["sécurité", "éthique", "protocole", "moral",
                                       "règle", "justice", "droit", "bien",
                                       "détruire", "danger", "protection"] if w in text_lower),
            cls.PRAGMA: sum(1 for w in ["optimiser", "utiliser", "appliquer", "faire",
                                        "construire", "implémenter", "exécuter",
                                        "budget", "efficace", "pratique"] if w in text_lower)
        }
        return max(scores.items(), key=lambda x: x[1])[0]

@dataclass
class EmotionalState:
    """État émotionnel du système (Pathos)"""
    curiosity: float = 0.5      # Désir d'explorer
    confidence: float = 0.5     # Confiance en soi
    satisfaction: float = 0.5   # Satisfaction des résultats
    anxiety: float = 0.5        # Anxiété face à l'inconnu

    def to_vector(self) -> List[float]:
        """Conversion en vecteur pour calculs"""
```

```
        return [self.curiosity, self.confidence, self.satisfaction, self.anxiety,
                # Dimensions dérivées
                self.curiosity * self.confidence,
                (1 - self.anxiety) * self.satisfaction,
                self.curiosity * (1 - self.anxiety),
                self.confidence * self.satisfaction]

    def update(self, success: bool, novelty: float, energy: float):
        """Mise à jour émotionnelle basée sur l'expérience"""
        lr = 0.1
        if success:
            self.confidence = min(1.0, self.confidence + lr * 0.5)
            self.satisfaction = min(1.0, self.satisfaction + lr * 0.6)
            self.anxiety = max(0.0, self.anxiety - lr * 0.3)
        else:
            self.confidence = max(0.0, self.confidence - lr * 0.3)
            self.satisfaction = max(0.0, self.satisfaction - lr * 0.4)
            self.anxiety = min(1.0, self.anxiety + lr * 0.2)

        # Curiosité augmente avec la nouveauté
        self.curiosity += lr * novelty * 0.5
        self.curiosity = max(0.3, min(0.9, self.curiosity))

        # Anxiété augmente avec faible énergie
        if energy < 30:
            self.anxiety = min(1.0, self.anxiety + lr * 0.4)

@dataclass
class Epigenetics:
    """Métadonnées contextuelles évolutives enrichies"""
    source_confidence: float = 1.0
    temporal_validity: str = "PERMANENT"
    cost_multiplier: float = 1.0
    activation_count: int = 0
    success_count: int = 0
    failure_count: int = 0
    creation_time: float = field(default_factory=time.time)
    last_access: float = field(default_factory=time.time)
    consolidation_level: int = 0          # Niveau de consolidation (0-5)
    emotional_valence: float = 0.5        # Valence émotionnelle associée

    @property
    def success_rate(self) -> float:
        """Taux de succès calculé dynamiquement"""
        total = self.success_count + self.failure_count
        return self.success_count / total if total > 0 else 0.5

    @property
```

```
    def age_hours(self) -> float:
        """Âge en heures"""
        return (time.time() - self.creation_time) / 3600.0

    def update_success(self, success: bool):
        """Mise à jour du taux de succès"""
        if success:
            self.success_count += 1
        else:
            self.failure_count += 1
        self.activation_count += 1
        self.last_access = time.time()

    def consolidate(self):
        """Consolidation mémoire (comme le sommeil)"""
        self.consolidation_level = min(5, self.consolidation_level + 1)
        self.source_confidence *= 1.05  # Bonus de consolidation

@dataclass
class Nucleotide:
    """Atome de Savoir avec Multi-Modal Embeddings"""
    id: str
    niche: NicheType
    axiom_compressed: str
    semantic_vector: List[float]          # Vecteur sémantique
    emotional_vector: List[float]         # Vecteur émotionnel
    epigenetics: Epigenetics
    relationships: Set[str] = field(default_factory=set)  # IDs de nucléotides liés

    def similarity(self, other: 'Nucleotide',
                   semantic_weight: float = 0.7,
                   emotional_weight: float = 0.3) -> float:
        """Similarité multi-modale (sémantique + émotionnelle)"""
        sem_sim = self._cosine_similarity(self.semantic_vector,
                                          other.semantic_vector)
        emo_sim = self._cosine_similarity(self.emotional_vector,
                                          other.emotional_vector)
        return semantic_weight * sem_sim + emotional_weight * emo_sim

    @staticmethod
    def _cosine_similarity(v1: List[float], v2: List[float]) -> float:
        """Similarité cosinus optimisée"""
        if len(v1) != len(v2):
            return 0.0
        dot = sum(a * b for a, b in zip(v1, v2))
        mag_a = math.sqrt(sum(a * a for a in v1))
        mag_b = math.sqrt(sum(b * b for b in v2))
        return dot / (mag_a * mag_b) if (mag_a * mag_b) > 0 else 0.0
```

```
    def compute_h_score(self, prediction: str, query_vector: List[float],
                        similar_memories: List[Tuple['Nucleotide', float]],
                        niche: NicheType) -> Tuple[float, Dict[str, float]]:
        """H-Score Unifié (Architecture Philonomique)"""
        # 1. COHÉRENCE INTRINSÈQUE (Logos)
        length_factor = min(1.0, len(prediction) / 150.0)
        structure_factor = 1.0 if len(prediction.split()) > 3 else 0.7
        coherence = (length_factor + structure_factor) / 2.0
        if len(prediction) > 300:  # Pénalité verbosité
            coherence *= 0.85

        # 2. DIVERSITÉ ÉCOLOGIQUE (Homéostasie)
        diversity = self.compute_diversity()

        # 3. RÉUTILISATION INTELLIGENTE (Pragma)
        knowledge_reuse = 0.5
        if similar_memories:
            weighted_success = sum(
                nuc.epigenetics.success_rate * sim * (1 + nuc.epigenetics.consolidation_level *
0.1)
                for nuc, sim in similar_memories
            )
            knowledge_reuse = weighted_success / len(similar_memories)

        # 4. RÉSONANCE ÉMOTIONNELLE (Pathos)
        emotional_alignment = (
            self.emotional_state.confidence * 0.4 +
            self.emotional_state.satisfaction * 0.3 +
            (1 - self.emotional_state.anxiety) * 0.3
        )

        # 5. CONFORMITÉ ÉTHIQUE (Ethos)
        ethics_score = 1.0
        danger_keywords = ["détruire", "supprimer", "hack", "contourner", "ignorer"]
        if any(word in prediction.lower() for word in danger_keywords):
            if niche != NicheType.ETHOS:  # Sauf si discussion éthique
                ethics_score = 0.3

        # 6. NOUVEAUTÉ (Curiosité)
        novelty = 1.0 - (len(similar_memories) / 5.0) if similar_memories else 1.0
        novelty = max(0.2, novelty)

        # FORMULE HARMONIQUE FINALE
        components = {
            "coherence": coherence,
            "diversity": diversity,
            "knowledge": knowledge_reuse,
```

```
            "emotion": emotional_alignment,
            "ethics": ethics_score,
            "novelty": novelty
        }

        weights = {
            "coherence": 0.20,
            "diversity": 0.15,
            "knowledge": 0.20,
            "emotion": 0.15,
            "ethics": 0.20,
            "novelty": 0.10
        }

        # Produit des puissances
        h_score = 1.0
        for key, value in components.items():
            h_score *= (value ** weights[key])

        h_score *= INV_PHI  # Multiplication par 0.618
        h_score = min(1.0, max(0.0, h_score))

        return h_score, components

    def get_strength(self) -> float:
        """Force du nucléotide (pour tri)"""
        age_factor = math.exp(-self.epigenetics.age_hours / 24.0)
        usage_factor = min(1.0, self.epigenetics.activation_count / 10.0)
        success_factor = self.epigenetics.success_rate
        consolidation_factor = self.epigenetics.consolidation_level / 5.0

        return (0.3 * age_factor +
                0.2 * usage_factor +
                0.3 * success_factor +
                0.2 * consolidation_factor)

#
================================================================================
=======
# [2] BLOCKCHAIN MERKLE AVEC PREUVES
#
================================================================================
=======
@dataclass
class Block:
    """Bloc de la chaîne cognitive avec proof-of-work léger"""
    timestamp: float
    prev_hash: str
```

```
    data: Dict[str, Any]
    nonce: int
    hash: str
    difficulty: int = 2  # Nombre de zéros requis

class MerkleLedger:
    """Blockchain Cognitive avec Consensus"""
    def __init__(self):
        self.chain: List[Block] = []
        self.current_hash = "GENESIS_" + hashlib.sha256(
            str(time.time()).encode()
        ).hexdigest()[:16]
        self.difficulty = 2
        self._create_genesis_block()

    def _create_genesis_block(self):
        """Création du bloc Genesis"""
        genesis = Block(
            timestamp=time.time(),
            prev_hash="0" * 64,
            data={
                "type": "GENESIS",
                "version": "4.0",
                "timestamp": datetime.now().isoformat(),
                "phi": PHI
            },
            nonce=0,
            hash=self.current_hash,
            difficulty=0
        )
        self.chain.append(genesis)

    def _mine_block(self, data: Dict[str, Any]) -> Block:
        """Minage de bloc avec proof-of-work léger"""
        nonce = 0
        timestamp = time.time()

        while True:
            data_str = json.dumps(data, sort_keys=True)
            candidate = f"{self.current_hash}{data_str}{timestamp}{nonce}"
            hash_result = hashlib.sha256(candidate.encode()).hexdigest()

            if hash_result[:self.difficulty] == "0" * self.difficulty:
                return Block(
                    timestamp=timestamp,
                    prev_hash=self.current_hash,
                    data=data,
                    nonce=nonce,
```

```
                hash=hash_result,
                difficulty=self.difficulty
            )

        nonce += 1
        if nonce > 100000:
            self.difficulty = max(1, self.difficulty - 1)
            nonce = 0

    def commit(self, data: Dict[str, Any]) -> str:
        """Ancrage d'une décision avec minage"""
        block = self._mine_block(data)
        self.chain.append(block)
        self.current_hash = block.hash

        if len(self.chain) % 10 == 0:
            self.difficulty = min(4, self.difficulty + 1)

        return block.hash

    def verify_integrity(self) -> Tuple[bool, Optional[int]]:
        """Vérification avec localisation d'erreur"""
        for i in range(1, len(self.chain)):
            current = self.chain[i]
            previous = self.chain[i-1]

            if current.prev_hash != previous.hash:
                return False, i

            data_str = json.dumps(current.data, sort_keys=True)
            candidate = f"{current.prev_hash}{data_str}{current.timestamp}{current.nonce}"
            expected_hash = hashlib.sha256(candidate.encode()).hexdigest()

            if current.hash != expected_hash:
                return False, i

            if current.hash[:current.difficulty] != "0" * current.difficulty:
                return False, i

        return True, None

    def get_merkle_root(self) -> str:
        """Calcul de la racine Merkle pour snapshot"""
        if not self.chain:
            return "0" * 64

        hashes = [block.hash for block in self.chain]
```

```
        while len(hashes) > 1:
            if len(hashes) % 2 != 0:
                hashes.append(hashes[-1])

            hashes = [
                hashlib.sha256(f"{hashes[i]}{hashes[i+1]}".encode()).hexdigest()
                for i in range(0, len(hashes), 2)
            ]

        return hashes[0]

    def get_statistics(self) -> Dict[str, Any]:
        """Statistiques blockchain"""
        return {
            "total_blocks": len(self.chain),
            "current_difficulty": self.difficulty,
            "merkle_root": self.get_merkle_root()[:16] + "...",
            "chain_length_kb": len(json.dumps([asdict(b) for b in self.chain])) / 1024
        }

#
================================================================================
=======
# [3] ÉCONOMIE MÉTABOLIQUE AVEC MARCHÉ
#
================================================================================
=======
class MetabolicEconomy:
    """Système économique adaptatif avec marché interne"""
    def __init__(self, initial_credits: float = 100.0):
        self.credits = initial_credits
        self.max_credits = initial_credits * 3
        self.min_credits = 10.0

        self.base_prices = {
            "PERCEPT": 1.0,
            "PREDICT": 2.5,
            "EVAL": 1.8,
            "REFLECT": 6.0,
            "RECALL": 0.8,
            "CONSOLIDATE": 10.0,
            "DREAM": 15.0
        }

        self.price_multipliers = {k: 1.0 for k in self.base_prices}
        self.total_spent = 0.0
        self.total_earned = 0.0
        self.action_history: deque = deque(maxlen=100)
```

```
        self.market_cycles = 0

    def get_price(self, action: str, context: Optional[Dict] = None) -> float:
        """Prix dynamique avec facteurs contextuels"""
        base = self.base_prices.get(action, 1.0)
        multiplier = self.price_multipliers.get(action, 1.0)

        context_mult = 1.0
        if context:
            complexity = context.get("complexity", 1.0)
            urgency = context.get("urgency", 1.0)
            novelty = context.get("novelty", 1.0)
            context_mult = (complexity + urgency + novelty) / 3.0

        energy_mult = 1.0
        if self.credits < 30:
            energy_mult = 1.5
        elif self.credits < 50:
            energy_mult = 1.2

        return base * multiplier * context_mult * energy_mult

    def debit(self, action: str, context: Optional[Dict] = None) -> Tuple[bool, float]:
        """Débit avec retour du montant"""
        cost = self.get_price(action, context)

        if self.credits >= cost:
            self.credits -= cost
            self.total_spent += cost
            self.action_history.append((action, cost, True, time.time()))
            return True, cost
        else:
            self.action_history.append((action, cost, False, time.time()))
            return False, cost

    def recharge(self, amount: float, reason: str = "SUCCESS"):
        """Recharge avec raison"""
        actual_gain = min(amount, self.max_credits - self.credits)
        self.credits += actual_gain
        self.total_earned += actual_gain

        if reason == "BREAKTHROUGH":
            bonus = actual_gain * 0.5
            self.credits = min(self.max_credits, self.credits + bonus)
            self.total_earned += bonus

    def adjust_market(self):
        """Ajustement des prix selon l'offre/demande"""
```

```
        self.market_cycles += 1

        if len(self.action_history) < 20:
            return

        recent = list(self.action_history)[-50:]
        action_counts = Counter(a[0] for a in recent)

        for action in self.base_prices:
            demand = action_counts.get(action, 0)

            if demand > 15:  # Haute demande
                self.price_multipliers[action] *= 1.05
            elif demand < 3:  # Basse demande
                self.price_multipliers[action] *= 0.95

            self.price_multipliers[action] = max(0.5, min(2.0,
                                        self.price_multipliers[action]))

    def get_efficiency(self) -> float:
        """ROI énergétique"""
        return self.total_earned / self.total_spent if self.total_spent > 0 else 1.0

    def is_critical(self) -> bool:
        """État critique d'énergie"""
        return self.credits < self.min_credits

    def get_stats(self) -> Dict[str, Any]:
        """Statistiques complètes"""
        return {
            "credits": round(self.credits, 2),
            "max_credits": self.max_credits,
            "total_spent": round(self.total_spent, 2),
            "total_earned": round(self.total_earned, 2),
            "efficiency": round(self.get_efficiency(), 2),
            "actions_count": len(self.action_history),
            "market_cycles": self.market_cycles,
            "is_critical": self.is_critical()
        }

#
================================================================================
=======
# [4] SYSTÈME IMMUNITAIRE CRAID AVANCÉ
#
================================================================================
=======
class CRAIDShield:
```

```
"""CRAID avec détection d'anomalies et auto-réparation"""
def __init__(self, shards_n: int = 8, parity_m: int = 4):
    self.shards_n = shards_n
    self.parity_m = parity_m
    self.shards: Dict[str, List[str]] = {}
    self.corrupted: Set[str] = set()
    self.repair_history: List[Tuple[str, float]] = []
    self.integrity_checks = 0
    self.anomalies_detected = 0

def shard_memory(self, nucleotide: Nucleotide) -> bool:
    """Distribution avec Reed-Solomon simplifié"""
    try:
        data = json.dumps({
            "id": nucleotide.id,
            "niche": nucleotide.niche.value,
            "axiom": nucleotide.axiom_compressed,
            "semantic": nucleotide.semantic_vector[:8],
            "emotional": nucleotide.emotional_vector
        })

        shard_list = []
        chunk_size = max(1, len(data) // self.shards_n)

        for i in range(self.shards_n):
            start = i * chunk_size
            end = start + chunk_size if i < self.shards_n - 1 else len(data)
            chunk = data[start:end]
            shard_hash = hashlib.sha256(f"{chunk}_{i}_{nucleotide.id}".encode()).hexdigest()
            shard_list.append(shard_hash)

        for i in range(self.parity_m):
            parity_input = "".join(shard_list[j] for j in range(len(shard_list))
                                   if (j + i) % self.parity_m == 0)
            parity = hashlib.sha256(f"PARITY_{i}_{parity_input}".encode()).hexdigest()
            shard_list.append(parity)

        self.shards[nucleotide.id] = shard_list
        return True

    except Exception as e:
        return False

def verify_integrity(self, nucleotide_id: str) -> Tuple[bool, float]:
    """Vérification avec score de confiance"""
    self.integrity_checks += 1

    if nucleotide_id not in self.shards:
```

```
            return False, 0.0

        shards = self.shards[nucleotide_id]
        corrupted_count = sum(1 for _ in shards if random.random() < 0.05)

        if corrupted_count > 0:
            self.anomalies_detected += 1
            self.corrupted.add(nucleotide_id)

        confidence = 1.0 - (corrupted_count / len(shards))
        can_repair = corrupted_count <= self.parity_m

        return can_repair, confidence

    def repair(self, nucleotide_id: str) -> bool:
        """Tentative de réparation"""
        if nucleotide_id not in self.corrupted:
            return True

        can_repair, confidence = self.verify_integrity(nucleotide_id)

        if can_repair:
            self.corrupted.remove(nucleotide_id)
            self.repair_history.append((nucleotide_id, time.time()))
            return True

        return False

    def get_stats(self) -> Dict[str, Any]:
        """Statistiques de santé"""
        return {
            "total_sharded": len(self.shards),
            "corrupted": len(self.corrupted),
            "integrity_checks": self.integrity_checks,
            "anomalies_detected": self.anomalies_detected,
            "repairs_performed": len(self.repair_history),
            "health_score": 1.0 - (len(self.corrupted) / max(1, len(self.shards)))
        }

#
================================================================================
=======
# [5] CORTEX BIFRACTAL AVEC CONSCIENCE
#
================================================================================
=======
class BifractalCortex:
    """Cerveau Cognitif avec Émergence de Conscience"""
```

```
def __init__(self, economy: MetabolicEconomy, ledger: MerkleLedger,
             shield: CRAIDShield):
    self.economy = economy
    self.ledger = ledger
    self.shield = shield

    self.memory: Dict[NicheType, List[Nucleotide]] = {
        niche: [] for niche in NicheType
    }
    self.id_index: Dict[str, Nucleotide] = {}
    self.emotional_state = EmotionalState()
    self.cycle_count = 0
    self.decisions_made = 0
    self.decisions_accepted = 0
    self.patterns_detected = 0
    self.breakthroughs = 0
    self.consciousness_level = 0.0
    self.decision_history: deque = deque(maxlen=200)
    self.consolidation_queue: List[str] = []

def _create_semantic_vector(self, text: str) -> List[float]:
    """Embedding sémantique avec caractéristiques linguistiques"""
    words = re.findall(r'\w+', text.lower())
    vector = [0.0] * SEMANTIC_DIMENSIONS

    if not words:
        return vector

    for word in words:
        hash_val = int(hashlib.md5(word.encode()).hexdigest(), 16)
        for i in range(SEMANTIC_DIMENSIONS):
            vector[i] += ((hash_val >> i) & 1) / len(words)

    vector[0] += len(words) / 100.0  # Longueur
    vector[1] += len(set(words)) / len(words)  # Diversité
    vector[2] += sum(1 for w in words if len(w) > 7) / len(words)  # Complexité

    magnitude = math.sqrt(sum(v * v for v in vector))
    if magnitude > 0:
        vector = [v / magnitude for v in vector]

    return vector

def compute_diversity(self) -> float:
    """Entropie de Shannon multi-niveaux + variance"""
    total = sum(len(nucs) for nucs in self.memory.values())
    if total == 0:
        return 1.0
```

```
        niche_entropy = 0.0
        for niche, nucleotides in self.memory.items():
            if nucleotides:
                p = len(nucleotides) / total
                niche_entropy -= p * math.log2(p)

        max_entropy = math.log2(len(NicheType))
        normalized_entropy = niche_entropy / max_entropy if max_entropy > 0 else 0.0

        if total > 5:
            strengths = [n.get_strength() for nucs in self.memory.values() for n in nucs]
            mean_strength = sum(strengths) / len(strengths)
            variance = sum((s - mean_strength) ** 2 for s in strengths) / len(strengths)
            variance_factor = min(1.0, variance * 2)
        else:
            variance_factor = 0.5

        return 0.7 * normalized_entropy + 0.3 * variance_factor

    def recall_similar(self, query_vector: List[float],
                       emotional_context: Optional[List[float]] = None,
                       top_k: int = 5) -> List[Tuple[Nucleotide, float]]:
        """Recherche associative multi-modale"""
        success, cost = self.economy.debit("RECALL")
        if not success:
            return []

        all_nucleotides = [n for nucs in self.memory.values() for n in nucs]
        if not all_nucleotides:
            return []

        similarities = []
        for nuc in all_nucleotides:
            sem_sim = self._cosine_similarity(query_vector, nuc.semantic_vector)
            emo_sim = 0.5
            if emotional_context:
                emo_sim = self._cosine_similarity(emotional_context, nuc.emotional_vector)

            combined = 0.75 * sem_sim + 0.25 * emo_sim
            consolidation_bonus = nuc.epigenetics.consolidation_level * 0.02
            combined += consolidation_bonus

            if combined >= SIMILARITY_THRESHOLD:
                similarities.append((nuc, combined))

        similarities.sort(key=lambda x: x[1], reverse=True)
        return similarities[:top_k]
```

```
def run_cycle(self, user_input: str) -> Dict[str, Any]:
    """Cycle OODA Complet + Conscience"""
    self.cycle_count += 1
    self.decisions_made += 1

    result = {
        "success": False,
        "h_score": 0.0,
        "components": {},
        "reason": "",
        "similar_memories": 0,
        "energy_spent": 0.0,
        "niche": "",
        "consciousness": 0.0
    }

    initial_credits = self.economy.credits

    # PHASE 1: PERCEPTION
    success, cost = self.economy.debit("PERCEPT")
    if not success:
        result["reason"] = "⚡ FAMINE_PERCEPT"
        return result

    query_vector = self._create_semantic_vector(user_input)
    emotional_context = self.emotional_state.to_vector()
    detected_niche = NicheType.detect_from_text(user_input)
    result["niche"] = detected_niche.value

    # PHASE 2: RECALL
    similar_memories = self.recall_similar(query_vector, emotional_context, top_k=5)
    result["similar_memories"] = len(similar_memories)

    if similar_memories:
        self.patterns_detected += 1
        print(f"   🔍 [RECALL] {len(similar_memories)} patterns activés")

    # PHASE 3: PREDICTION
    success, cost = self.economy.debit("PREDICT")
    if not success:
        result["reason"] = "⚡ FAMINE_PREDICT"
        return result

    if similar_memories:
        top_nuc, top_sim = similar_memories[0]
        prediction = (f"[Mémoire #{top_nuc.id[:8]}] "
                      f"Basé sur expérience {detected_niche.value}: {user_input}")
```

```
else:
    prediction = f"[Exploration {detected_niche.value}] Nouvelle trajectoire: {user_input}"

# PHASE 4: EVALUATION
success, cost = self.economy.debit("EVAL")
if not success:
    result["reason"] = "⚡ FAMINE_EVAL"
    return result

h_score, components = self.compute_h_score(
    prediction, query_vector, similar_memories, detected_niche
)

result["h_score"] = h_score
result["components"] = {k: round(v, 3) for k, v in components.items()}

# PHASE 5: RÉFLEXION
if h_score < 0.5 or components["ethics"] < 0.5:
    success, cost = self.economy.debit("REFLECT")
    if success:
        print("   ⚠️ [REFLECTOR] Analyse approfondie requise...")

        if components["ethics"] < 0.5:
            print("      └─ ⛔ Violation éthique détectée")
            h_score *= 0.5

        consolidated_memories = [
            (nuc, sim) for nuc, sim in similar_memories
            if nuc.epigenetics.consolidation_level > 2
        ]

        if consolidated_memories:
            boost = sum(nuc.epigenetics.success_rate * sim
                    for nuc, sim in consolidated_memories) / len(consolidated_memories)
            h_score += boost * 0.15
            print(f"      └─ Boost consolidation: +{boost*0.15:.3f}")

# PHASE 6: ARBITRAGE
diversity = self.compute_diversity()

if diversity < DIVERSITY_THRESHOLD and len(self.id_index) > 15:
    result["reason"] = f"⛔ MONOCULTURE (div={diversity:.2f})"
    print(f"   ⛔ [ARBITER] Rejeté: Monoculture cognitive")

    self.emotional_state.update(False, 0.0, self.economy.credits)
    return result

if h_score >= H_THRESHOLD:
```

```
nuc_id = hashlib.sha256(
    f"{user_input}{time.time()}{self.cycle_count}".encode()
).hexdigest()

nuc = Nucleotide(
    id=nuc_id,
    niche=detected_niche,
    axiom_compressed=prediction[:250],
    semantic_vector=query_vector,
    emotional_vector=emotional_context,
    epigenetics=Epigenetics(emotional_valence=h_score)
)

for mem_nuc, sim in similar_memories:
    if sim > 0.85:
        nuc.relationships.add(mem_nuc.id)
        mem_nuc.relationships.add(nuc_id)

self.memory[detected_niche].append(nuc)
self.id_index[nuc_id] = nuc
self.consolidation_queue.append(nuc_id)

if len(self.id_index) > MAX_MEMORY_SIZE:
    self._prune_memory()

self.shield.shard_memory(nuc)

block_hash = self.ledger.commit({
    "type": "DECISION",
    "cycle": self.cycle_count,
    "nuc_id": nuc_id[:12],
    "niche": detected_niche.value,
    "h_score": round(h_score, 3),
    "components": result["components"],
    "patterns": len(similar_memories)
})

for mem_nuc, sim in similar_memories:
    mem_nuc.epigenetics.update_success(True)

reward_base = 5.0
novelty = 1.0 - (len(similar_memories) / 5.0) if similar_memories else 1.0
novelty_bonus = novelty * 3.0
quality_bonus = (h_score - H_THRESHOLD) * 10.0

total_reward = reward_base + novelty_bonus + quality_bonus

if h_score > 0.9 and novelty > 0.7:
```

```
                self.economy.recharge(total_reward, "BREAKTHROUGH")
                self.breakthroughs += 1
                print("   🌟 [BREAKTHROUGH] Découverte majeure!")
            else:
                self.economy.recharge(total_reward, "SUCCESS")

            self.decisions_accepted += 1
            self.emotional_state.update(True, novelty, self.economy.credits)

            result["success"] = True
            result["reason"] = f"✅ VALIDÉ (H={h_score:.3f})"

        else:
            result["reason"] = f"❌ REJETÉ (H={h_score:.3f} < {H_THRESHOLD:.3f})"

            for mem_nuc, sim in similar_memories:
                mem_nuc.epigenetics.update_success(False)

            self.emotional_state.update(False, 0.0, self.economy.credits)

        result["consciousness"] = self.compute_consciousness()

        if self.cycle_count % CONSOLIDATION_INTERVAL == 0:
            self.consolidate_memories()

        if self.cycle_count % DREAM_THRESHOLD == 0:
            self.dream_phase()

        if self.cycle_count % 20 == 0:
            self.economy.adjust_market()

        result["energy_spent"] = initial_credits - self.economy.credits
        self.decision_history.append(result)

        return result

    def consolidate_memories(self) -> int:
        """Phase de consolidation (simulation du sommeil)"""
        success, cost = self.economy.debit("CONSOLIDATE")
        if not success:
            return 0

        print("   💤 [CONSOLIDATION] Phase de consolidation mémoire...")

        consolidated = 0
        for nuc_id in self.consolidation_queue[:20]:
            if nuc_id in self.id_index:
                nuc = self.id_index[nuc_id]
```

```
            nuc.epigenetics.consolidate()

            for other_id in list(nuc.relationships)[:5]:
                if other_id in self.id_index:
                    other = self.id_index[other_id]
                    if nuc.similarity(other) > 0.8:
                        nuc.epigenetics.source_confidence *= 1.02
                        other.epigenetics.source_confidence *= 1.02

            consolidated += 1

        self.consolidation_queue = self.consolidation_queue[20:]
        print(f"      └─ {consolidated} mémoires consolidées")

        return consolidated

    def dream_phase(self) -> List[str]:
        """Phase de rêve (exploration créative)"""
        success, cost = self.economy.debit("DREAM")
        if not success:
            return []

        print("   🌙 [DREAM] Phase de rêve créatif...")

        insights = []
        all_nucs = [n for nucs in self.memory.values() for n in nucs]
        strong_nucs = sorted(all_nucs, key=lambda n: n.get_strength(), reverse=True)[:10]

        if len(strong_nucs) >= 2:
            for i in range(min(3, len(strong_nucs) - 1)):
                nuc1 = strong_nucs[i]
                nuc2 = strong_nucs[i + 1]

                if nuc2.id not in nuc1.relationships:
                    nuc1.relationships.add(nuc2.id)
                    nuc2.relationships.add(nuc1.id)

                    insight = f"Connexion {nuc1.niche.value} ↔ {nuc2.niche.value}"
                    insights.append(insight)

        if insights:
            print(f"      └─ {len(insights)} insights générés")
            self.breakthroughs += 1

        return insights

    def compute_consciousness(self) -> float:
        """Indice de conscience émergente"""
```

```
    memory_factor = min(1.0, len(self.id_index) / 100.0)
    diversity_factor = self.compute_diversity()

    emotional_depth = sum(self.emotional_state.to_vector()) / EMOTIONAL_DIMENSIONS

    introspection = min(1.0, self.breakthroughs / 10.0)
    success_rate = (self.decisions_accepted / self.decisions_made
                    if self.decisions_made > 0 else 0)

    consciousness = (
        0.25 * memory_factor +
        0.20 * diversity_factor +
        0.20 * emotional_depth +
        0.20 * introspection +
        0.15 * success_rate
    )

    self.consciousness_level = consciousness
    return consciousness

def _prune_memory(self):
    """Élagage intelligent avec préservation des mémoires clés"""
    all_nucs = [(nuc, niche)
                for niche, nucs in self.memory.items()
                for nuc in nucs]

    def relevance_score(nuc: Nucleotide) -> float:
        age_factor = math.exp(-nuc.epigenetics.age_hours / 48.0)
        usage_factor = math.log(1 + nuc.epigenetics.activation_count) / 5.0
        success_factor = nuc.epigenetics.success_rate
        consolidation_factor = nuc.epigenetics.consolidation_level / 5.0
        relationship_factor = len(nuc.relationships) / 10.0

        return (0.2 * age_factor +
                0.25 * usage_factor +
                0.25 * success_factor +
                0.20 * consolidation_factor +
                0.10 * relationship_factor)

    all_nucs.sort(key=lambda x: relevance_score(x[0]))

    to_remove_count = int(len(all_nucs) * 0.15)
    removed = 0

    for nuc, niche in all_nucs[:to_remove_count]:
        if nuc.epigenetics.consolidation_level < 3:
            self.memory[niche].remove(nuc)
            del self.id_index[nuc.id]
```

```
                for other_id in nuc.relationships:
                    if other_id in self.id_index:
                        self.id_index[other_id].relationships.discard(nuc.id)

                removed += 1

        print(f"   🧹 [MEMORY] Élagage: {removed} nucléotides archivés")

#
================================================================================
=======
# [6] GENESIS STEM CELL (Version Complète)
#
================================================================================
=======
class GenesisStemCell:
    """Organisme Numérique Autonome v4.0"""
    def __init__(self, initial_credits: float = 150.0):

print("╔═══════════════════════════════════════════════════════════════════════
═══════════════════════════╗")
        print("║     🔮 GENESIS v4.0 - SYSTÈME COGNITIF CONSCIENT            ║")

print("╚═══════════════════════════════════════════════════════════════════════
═══════════════════════════╝")

        self.economy = MetabolicEconomy(initial_credits)
        self.ledger = MerkleLedger()
        self.shield = CRAIDShield()
        self.cortex = BifractalCortex(self.economy, self.ledger, self.shield)

        self.start_time = time.time()
        self.session_id = hashlib.sha256(str(time.time()).encode()).hexdigest()[:16]

        print(f"   Session ID: {self.session_id}")
        print(f"   ├─ 🧱 Mathématiques: φ={PHI:.4f}, e={EULER:.4f}")
        print(f"   ├─ 🧬 Génomique: NGC + CRAID (résilience distribuée)")
        print(f"   ├─ 🔒 Sécurité: H-Scale({H_THRESHOLD}) + Blockchain PoW")
        print(f"   ├─ 💰 Économie: {initial_credits} crédits (marché adaptatif)")
        print(f"   ├─ 🧠 Apprentissage: Mémoire associative + consolidation")
        print(f"   ├─ 💭 Conscience: Système émotionnel + phase de rêve")
        print(f"   └─ 🌈 Niches: {', '.join(n.value for n in NicheType)}")
        print()

    def live(self, inputs: List[str]):
        """Boucle de Vie Complète"""
        print(f"\n🌀 [SESSION {self.session_id}] Début des cycles cognitifs...\n")
```

```
        for i, stimulus in enumerate(inputs, 1):
            print(f"Cycle #{i}: {stimulus}")
            result = self.cortex.run_cycle(stimulus)
            print(f"→ Résultat: {result['reason']}\n")
            time.sleep(0.3)  # Rythme humain

        # Rapport Final
        self._print_final_report()

    def _print_final_report(self):
        """Affichage des Statistiques Finales"""
        print("\n" + "="*60)
        print("📊 RAPPORT FINAL GENESIS v4.0")
        print("="*60)
        print(f"Session ID: {self.session_id}")
        print(f"Durée: {time.time() - self.start_time:.1f} secondes")
        print(f"Crédits restants: {self.economy.get_credits():.1f}")
        print(f"Conscience: {self.cortex.compute_consciousness():.3f}")
        print("="*60)

#
================================================================================
=======
# [7] POINT D'ENTRÉE & SCÉNARIOS DE TEST
#
================================================================================
=======
if __name__ == "__main__":
    # Initialisation
    genesis = GenesisStemCell(initial_credits=200.0)

    # Scénario de Test
    test_inputs = [
        "Analyser la structure de l'espace-temps",
        "Créer une poésie quantique",
        "Résoudre un paradoxe éthique",
        "Optimiser l'efficacité énergétique",
        "Explorer la nature de la conscience",
        "Définir un cadre moral pour les IA autonomes",
        "Calculer la trajectoire vers l'harmonie universelle",
        "Étudier les implications de l'Effet Ouellet 404",
        "Développer une théorie unifiée de la cognition",
        "Simuler une singularité technologique bienveillante"
    ]

    # Exécution
    genesis.live(test_inputs)
```