Examen DSA + Design Patterns – Intermédiaire (Questions + Réponses + Solutions Java & Python)

DSA – Structures de données & Algorithmes (avec code)

Q1. Implémenter une pile avec support getMin() en O(1).

Réponse : Utiliser une 2e pile des minimums (ou stocker (val,minCourant)).

Java :

```
class MinStack {
  private java.util.Deque<Integer> st = new java.util.ArrayDeque<>();
  private java.util.Deque<Integer> mins = new java.util.ArrayDeque<>();
  void push(int x){ st.push(x); mins.push(mins.isEmpty()? x : Math.min(x, mins.peek())); }
  int pop(){ mins.pop(); return st.pop(); }
  int min(){ return mins.peek(); }
}
```

Python :

```
class MinStack:
    def __init__(self):
        self.st=[]; self.mins=[]
    def push(self,x):
        self.st.append(x)
        self.mins.append(x if not self.mins else min(x,self.mins[-1]))
    def pop(self):
        self.mins.pop()
        return self.st.pop()
    def get_min(self):
        return self.mins[-1]
```

Q2. Détecter si une liste chaînée a un cycle.

Réponse : Floyd (tortue-lièvre) O(n) temps, O(1) espace.

Java :

```
boolean hasCycle(ListNode head){
  ListNode slow=head, fast=head;
  while(fast!=null && fast.next!=null){
    slow=slow.next; fast=fast.next.next;
    if(slow==fast) return true;
  }
  return false;
}
```

Python :

```
def has_cycle(head):
    slow=fast=head
    while fast and fast.next:
        slow=slow.next
        fast=fast.next.next
        if slow is fast:
            return True
    return False
```

Q3. Trouver le plus court chemin dans un graphe non pondéré.

Réponse : BFS depuis la source; garder parent/dist.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q4. Calculer la médiane d'un flux de nombres.

Réponse : Deux heaps: maxHeap (bas) et minHeap (haut). Rééquilibrer.

Java :

```
// Median of stream
java.util.PriorityQueue<Integer> low = new java.util.PriorityQueue<>(java.util.Collections.reverseOrder());
java.util.PriorityQueue<Integer> high = new java.util.PriorityQueue<>();
void add(int x){
  if(low.isEmpty() || x<=low.peek()) low.add(x); else high.add(x);
  if(low.size()>high.size()+1) high.add(low.poll());
  if(high.size()>low.size()) low.add(high.poll());
}
double median(){ return low.size()==high.size()? (low.peek()+high.peek())/2.0 : low.peek(); }
```

Python :

```
import heapq
low=[]   # max-heap via negatives
high=[]  # min-heap
def add(x):
    if not low or x <= -low[0]:
        heapq.heappush(low, -x)
    else:
        heapq.heappush(high, x)
    if len(low) > len(high)+1:
        heapq.heappush(high, -heapq.heappop(low))
    if len(high) > len(low):
        heapq.heappush(low, -heapq.heappop(high))
def median():
    if len(low)==len(high):
        return (-low[0]+high[0])/2
    return -low[0]
```

Q5. Top-K éléments les plus fréquents.

Réponse : HashMap counts + heap taille K (ou bucket sort).

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q6. Two Sum (indices) en O(n).

Réponse : HashMap valeur→index.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q7. Longest Substring Without Repeating Characters.

Réponse : Sliding window + map dernier index.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q8. Fusionner k listes triées.

Réponse : Min-heap des têtes; O(N log k).

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q9. Détecter un intervalle qui chevauche (merge intervals).

Réponse : Trier par start, fusionner en parcourant.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q10. Binary search : conditions de boucle correctes.

Réponse : Invariants low/high, mid; attention overflow; renvoyer insertion point.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q11. Implémenter une pile avec support getMin() en O(1).

Réponse : Utiliser une 2e pile des minimums (ou stocker (val,minCourant)).

Java :

```
class MinStack {
  private java.util.Deque<Integer> st = new java.util.ArrayDeque<>();
  private java.util.Deque<Integer> mins = new java.util.ArrayDeque<>();
  void push(int x){ st.push(x); mins.push(mins.isEmpty()? x : Math.min(x, mins.peek())); }
  int pop(){ mins.pop(); return st.pop(); }
  int min(){ return mins.peek(); }
}
```

Python :

```
class MinStack:
    def __init__(self):
        self.st=[]; self.mins=[]
    def push(self,x):
        self.st.append(x)
        self.mins.append(x if not self.mins else min(x,self.mins[-1]))
    def pop(self):
        self.mins.pop()
        return self.st.pop()
```

```
    def get_min(self):
        return self.mins[-1]
```

Q12. Détecter si une liste chaînée a un cycle.

Réponse : Floyd (tortue-lièvre) O(n) temps, O(1) espace.

Java :

```
boolean hasCycle(ListNode head){
  ListNode slow=head, fast=head;
  while(fast!=null && fast.next!=null){
    slow=slow.next; fast=fast.next.next;
    if(slow==fast) return true;
  }
  return false;
}
```

Python :

```
def has_cycle(head):
    slow=fast=head
    while fast and fast.next:
        slow=slow.next
        fast=fast.next.next
        if slow is fast:
            return True
    return False
```

Q13. Trouver le plus court chemin dans un graphe non pondéré.

Réponse : BFS depuis la source; garder parent/dist.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q14. Calculer la médiane d'un flux de nombres.

Réponse : Deux heaps: maxHeap (bas) et minHeap (haut). Rééquilibrer.

Java :

```
// Median of stream
java.util.PriorityQueue<Integer> low = new java.util.PriorityQueue<>(java.util.Collections.reverseOrder());
java.util.PriorityQueue<Integer> high = new java.util.PriorityQueue<>();
void add(int x){
  if(low.isEmpty() || x<=low.peek()) low.add(x); else high.add(x);
  if(low.size()>high.size()+1) high.add(low.poll());
  if(high.size()>low.size()) low.add(high.poll());
}
double median(){ return low.size()==high.size()? (low.peek()+high.peek())/2.0 : low.peek(); }
```

Python :

```
import heapq
low=[]   # max-heap via negatives
high=[]  # min-heap
def add(x):
    if not low or x <= -low[0]:
        heapq.heappush(low, -x)
    else:
        heapq.heappush(high, x)
    if len(low) > len(high)+1:
        heapq.heappush(high, -heapq.heappop(low))
```

```
    if len(high) > len(low):
        heapq.heappush(low, -heapq.heappop(high))
def median():
    if len(low)==len(high):
        return (-low[0]+high[0])/2
    return -low[0]
```

Q15. Top-K éléments les plus fréquents.

Réponse : HashMap counts + heap taille K (ou bucket sort).

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q16. Two Sum (indices) en O(n).

Réponse : HashMap valeur→index.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q17. Longest Substring Without Repeating Characters.

Réponse : Sliding window + map dernier index.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q18. Fusionner k listes triées.

Réponse : Min-heap des têtes; O(N log k).

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q19. Détecter un intervalle qui chevauche (merge intervals).

Réponse : Trier par start, fusionner en parcourant.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q20. Binary search : conditions de boucle correctes.

Réponse : Invariants low/high, mid; attention overflow; renvoyer insertion point.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q21. Implémenter une pile avec support getMin() en O(1).

Réponse : Utiliser une 2e pile des minimums (ou stocker (val,minCourant)).

Java :

```
class MinStack {
  private java.util.Deque<Integer> st = new java.util.ArrayDeque<>();
  private java.util.Deque<Integer> mins = new java.util.ArrayDeque<>();
  void push(int x){ st.push(x); mins.push(mins.isEmpty()? x : Math.min(x, mins.peek())); }
  int pop(){ mins.pop(); return st.pop(); }
  int min(){ return mins.peek(); }
}
```

Python :

```
class MinStack:
    def __init__(self):
        self.st=[]; self.mins=[]
    def push(self,x):
        self.st.append(x)
        self.mins.append(x if not self.mins else min(x,self.mins[-1]))
    def pop(self):
        self.mins.pop()
        return self.st.pop()
    def get_min(self):
        return self.mins[-1]
```

Q22. Détecter si une liste chaînée a un cycle.

Réponse : Floyd (tortue-lièvre) O(n) temps, O(1) espace.

Java :

```
boolean hasCycle(ListNode head){
  ListNode slow=head, fast=head;
  while(fast!=null && fast.next!=null){
    slow=slow.next; fast=fast.next.next;
    if(slow==fast) return true;
  }
  return false;
}
```

Python :

```
def has_cycle(head):
    slow=fast=head
    while fast and fast.next:
        slow=slow.next
        fast=fast.next.next
        if slow is fast:
            return True
    return False
```

Q23. Trouver le plus court chemin dans un graphe non pondéré.

Réponse : BFS depuis la source; garder parent/dist.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q24. Calculer la médiane d'un flux de nombres.

Réponse : Deux heaps: maxHeap (bas) et minHeap (haut). Rééquilibrer.

Java :

```
// Median of stream
java.util.PriorityQueue<Integer> low = new java.util.PriorityQueue<>(java.util.Collections.reverseOrder());
java.util.PriorityQueue<Integer> high = new java.util.PriorityQueue<>();
void add(int x){
  if(low.isEmpty() || x<=low.peek()) low.add(x); else high.add(x);
  if(low.size()>high.size()+1) high.add(low.poll());
  if(high.size()>low.size()) low.add(high.poll());
}
double median(){ return low.size()==high.size()? (low.peek()+high.peek())/2.0 : low.peek(); }
```

Python :

```
import heapq
low=[]   # max-heap via negatives
high=[]  # min-heap
def add(x):
    if not low or x <= -low[0]:
        heapq.heappush(low, -x)
    else:
        heapq.heappush(high, x)
    if len(low) > len(high)+1:
        heapq.heappush(high, -heapq.heappop(low))
    if len(high) > len(low):
        heapq.heappush(low, -heapq.heappop(high))
def median():
    if len(low)==len(high):
        return (-low[0]+high[0])/2
    return -low[0]
```

Q25. Top-K éléments les plus fréquents.

Réponse : HashMap counts + heap taille K (ou bucket sort).

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q26. Two Sum (indices) en O(n).

Réponse : HashMap valeur→index.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q27. Longest Substring Without Repeating Characters.

Réponse : Sliding window + map dernier index.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q28. Fusionner k listes triées.

Réponse : Min-heap des têtes; O(N log k).

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q29. Détecter un intervalle qui chevauche (merge intervals).

Réponse : Trier par start, fusionner en parcourant.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

Q30. Binary search : conditions de boucle correctes.

Réponse : Invariants low/high, mid; attention overflow; renvoyer insertion point.

Java :

```
// Voir approche dans la réponse; implémentation standard.
```

Python :

```
# Implémentation standard (voir réponse).
```

DP – Design Patterns (avec code)

Q31. Appliquer Strategy pour calcul de frais (fees) selon type de compte.

Réponse : Interface FeeStrategy; implémentations; injection dans service.

Java :

```
interface FeeStrategy { java.math.BigDecimal fee(Transaction t); }
class RetailFee implements FeeStrategy { public java.math.BigDecimal fee(Transaction t){ return t.amount().
class VipFee implements FeeStrategy { public java.math.BigDecimal fee(Transaction t){ return java.math.BigD
```

Python :

```
from dataclasses import dataclass
@dataclass
class Tx: amount: float
class FeeStrategy:
    def fee(self, tx: Tx) -> float: raise NotImplementedError
class RetailFee(FeeStrategy):
    def fee(self, tx): return tx.amount*0.01
class VipFee(FeeStrategy):
    def fee(self, tx): return 0.0
```

Q32. Utiliser Observer pour notifications après transaction.

Réponse : Publisher émet event; subscribers (Email/SMS/Fraud) reçoivent.

Java :

```
interface TxListener { void onPosted(Transfer tx); }
class EventBus {
  private final java.util.List<TxListener> ls = new java.util.ArrayList<>();
  void subscribe(TxListener l){ ls.add(l); }
  void publish(Transfer tx){ for(var l: ls) l.onPosted(tx); }
}
```

Python :

```
class EventBus:
    def __init__(self): self.subs=[]
    def subscribe(self, fn): self.subs.append(fn)
    def publish(self, tx):
        for fn in self.subs: fn(tx)
```

Q33. Factory Method pour créer des comptes (chequing/savings).

Réponse : Creator expose createAccount(); sous-classes créent produit.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q34. Abstract Factory pour famille 'Carte + Plafond + Règles'.

Réponse : Factory crée Card + LimitPolicy + FeePolicy compatibles.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q35. Decorator pour ajouter logging/métriques autour d’un service.

Réponse : Wrapper implémente même interface et délègue avec ajout.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q36. Adapter pour intégrer un provider de paiement externe.

Réponse : Adapter traduit calls internes vers API tiers.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q37. Command pour encapsuler 'Virement' avec retry/queue.

Réponse : Command.execute() + serialize; invoker gère retries.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q38. Template Method pour pipeline KYC.

Réponse : Classe abstraite définit étapes; sous-classes spécialisent.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q39. Builder pour construire un objet Transaction riche.

Réponse : Builder valide champs, construit immutable Transaction.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q40. Circuit Breaker : state machine.

Réponse : Closed→Open→Half-open; compter échecs; timeouts.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q41. Appliquer Strategy pour calcul de frais (fees) selon type de compte.

Réponse : Interface FeeStrategy; implémentations; injection dans service.

Java :

```
interface FeeStrategy { java.math.BigDecimal fee(Transaction t); }
class RetailFee implements FeeStrategy { public java.math.BigDecimal fee(Transaction t){ return t.amount().
class VipFee implements FeeStrategy { public java.math.BigDecimal fee(Transaction t){ return java.math.BigD
```

Python :

```
from dataclasses import dataclass
@dataclass
class Tx: amount: float
class FeeStrategy:
    def fee(self, tx: Tx) -> float: raise NotImplementedError
class RetailFee(FeeStrategy):
    def fee(self, tx): return tx.amount*0.01
class VipFee(FeeStrategy):
    def fee(self, tx): return 0.0
```

Q42. Utiliser Observer pour notifications après transaction.

Réponse : Publisher émet event; subscribers (Email/SMS/Fraud) reçoivent.

Java :

```
interface TxListener { void onPosted(Transfer tx); }
class EventBus {
  private final java.util.List<TxListener> ls = new java.util.ArrayList<>();
  void subscribe(TxListener l){ ls.add(l); }
  void publish(Transfer tx){ for(var l: ls) l.onPosted(tx); }
}
```

Python :

```
class EventBus:
    def __init__(self): self.subs=[]
    def subscribe(self, fn): self.subs.append(fn)
    def publish(self, tx):
        for fn in self.subs: fn(tx)
```

Q43. Factory Method pour créer des comptes (chequing/savings).

Réponse : Creator expose createAccount(); sous-classes créent produit.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q44. Abstract Factory pour famille 'Carte + Plafond + Règles'.

Réponse : Factory crée Card + LimitPolicy + FeePolicy compatibles.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q45. Decorator pour ajouter logging/métriques autour d’un service.

Réponse : Wrapper implémente même interface et délègue avec ajout.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q46. Adapter pour intégrer un provider de paiement externe.

Réponse : Adapter traduit calls internes vers API tiers.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q47. Command pour encapsuler 'Virement' avec retry/queue.

Réponse : Command.execute() + serialize; invoker gère retries.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q48. Template Method pour pipeline KYC.

Réponse : Classe abstraite définit étapes; sous-classes spécialisent.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q49. Builder pour construire un objet Transaction riche.

Réponse : Builder valide champs, construit immutable Transaction.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q50. Circuit Breaker : state machine.

Réponse : Closed→Open→Half-open; compter échecs; timeouts.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q51. Appliquer Strategy pour calcul de frais (fees) selon type de compte.

Réponse : Interface FeeStrategy; implémentations; injection dans service.

Java :

```
interface FeeStrategy { java.math.BigDecimal fee(Transaction t); }
class RetailFee implements FeeStrategy { public java.math.BigDecimal fee(Transaction t){ return t.amount().
class VipFee implements FeeStrategy { public java.math.BigDecimal fee(Transaction t){ return java.math.BigD
```

Python :

```
from dataclasses import dataclass
@dataclass
class Tx: amount: float
class FeeStrategy:
    def fee(self, tx: Tx) -> float: raise NotImplementedError
class RetailFee(FeeStrategy):
    def fee(self, tx): return tx.amount*0.01
class VipFee(FeeStrategy):
    def fee(self, tx): return 0.0
```

Q52. Utiliser Observer pour notifications après transaction.

Réponse : Publisher émet event; subscribers (Email/SMS/Fraud) reçoivent.

Java :

```
interface TxListener { void onPosted(Transfer tx); }
class EventBus {
  private final java.util.List<TxListener> ls = new java.util.ArrayList<>();
  void subscribe(TxListener l){ ls.add(l); }
  void publish(Transfer tx){ for(var l: ls) l.onPosted(tx); }
}
```

Python :

```
class EventBus:
    def __init__(self): self.subs=[]
    def subscribe(self, fn): self.subs.append(fn)
    def publish(self, tx):
        for fn in self.subs: fn(tx)
```

Q53. Factory Method pour créer des comptes (chequing/savings).

Réponse : Creator expose createAccount(); sous-classes créent produit.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q54. Abstract Factory pour famille 'Carte + Plafond + Règles'.

Réponse : Factory crée Card + LimitPolicy + FeePolicy compatibles.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q55. Decorator pour ajouter logging/métriques autour d’un service.

Réponse : Wrapper implémente même interface et délègue avec ajout.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q56. Adapter pour intégrer un provider de paiement externe.

Réponse : Adapter traduit calls internes vers API tiers.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q57. Command pour encapsuler 'Virement' avec retry/queue.

Réponse : Command.execute() + serialize; invoker gère retries.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q58. Template Method pour pipeline KYC.

Réponse : Classe abstraite définit étapes; sous-classes spécialisent.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q59. Builder pour construire un objet Transaction riche.

Réponse : Builder valide champs, construit immutable Transaction.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```

Q60. Circuit Breaker : state machine.

Réponse : Closed→Open→Half-open; compter échecs; timeouts.

Java :

```
// Exemple de code: voir réponse (pattern appliqué).
```

Python :

```
# Exemple de code: voir réponse (pattern appliqué).
```