Oracle Coding Practice Workbook – With Complexity Cheat Sheet

1. LRU Cache

Design a cache with get/put operations in O(1). Evict the least recently used item when capacity is exceeded.
Approach: Use a HashMap for O(1) lookups and a Doubly Linked List to maintain usage order. On access, move the node to the front. Evict from the tail when over capacity.

```
#include <unordered_map>
using namespace std;

struct Node {
    int key, value;
    Node *prev, *next;
    Node(int k, int v): key(k), value(v), prev(nullptr), next(nullptr) {}
};

class LRUCache {
    int capacity;
    unordered_map<int, Node*> cache;
    Node *head, *tail;

    void remove(Node* node) {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }
    void insert(Node* node) {
        node->next = head->next;
        head->next->prev = node;
        head->next = node;
        node->prev = head;
    }

public:
    LRUCache(int cap): capacity(cap) {
        head = new Node(0,0);
        tail = new Node(0,0);
        head->next = tail;
        tail->prev = head;
    }

    int get(int key) {
        if (cache.find(key) == cache.end()) return -1;
        Node* node = cache[key];
        remove(node);
        insert(node);
        return node->value;
    }

    void put(int key, int value) {
        if (cache.find(key) != cache.end()) {
            remove(cache[key]);
        }
        Node* node = new Node(key, value);
        cache[key] = node;
        insert(node);

        if (cache.size() > capacity) {
            Node* lru = tail->prev;
            remove(lru);
            cache.erase(lru->key);
            delete lru;
```

```
        }
    }
};
```

2. Top K Frequent Elements

Given an array of integers, return the k most frequent elements.
Approach: Count frequencies with HashMap. Use a min-heap to maintain the top k frequent elements.

```
#include <vector>
#include <unordered_map>
#include <queue>
using namespace std;

vector<int> topKFrequent(vector<int>& nums, int k) {
    unordered_map<int,int> freq;
    for (int n : nums) freq[n]++;

    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> minHeap;

    for (auto& [num, count] : freq) {
        minHeap.push({count, num});
        if (minHeap.size() > k) minHeap.pop();
    }

    vector<int> result;
    while (!minHeap.empty()) {
        result.push_back(minHeap.top().second);
        minHeap.pop();
    }
    return result;
}
```

3. Rotate Matrix (90 degrees)

Rotate an N x N matrix by 90 degrees clockwise, in place.
Approach: First transpose the matrix, then reverse each row.

```
#include <vector>
using namespace std;

void rotate(vector<vector<int>>& matrix) {
    int n = matrix.size();
    // transpose
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            swap(matrix[i][j], matrix[j][i]);
        }
    }
    // reverse each row
    for (int i = 0; i < n; i++) {
        reverse(matrix[i].begin(), matrix[i].end());
    }
}
```

4. Concurrency: Print FooBar Alternately

Two threads alternately print Foo and Bar.
Approach: Use mutex + condition_variable to synchronize between two threads.

```
#include <iostream>
#include <thread>
#include <mutex>
```

```
#include <condition_variable>
using namespace std;

class FooBar {
private:
    int n;
    mutex mtx;
    condition_variable cv;
    bool fooTurn;

public:
    FooBar(int n) {
        this->n = n;
        fooTurn = true;
    }

    void foo() {
        for (int i = 0; i < n; i++) {
            unique_lock<mutex> lock(mtx);
            cv.wait(lock, [&]{ return fooTurn; });
            cout << "Foo";
            fooTurn = false;
            cv.notify_all();
        }
    }

    void bar() {
        for (int i = 0; i < n; i++) {
            unique_lock<mutex> lock(mtx);
            cv.wait(lock, [&]{ return !fooTurn; });
            cout << "Bar\n";
            fooTurn = true;
            cv.notify_all();
        }
    }
};
```

5. Rate Limiter

Implement a class RateLimiter that allows X requests per user per Y seconds.
Approach: Use sliding window or token bucket. Here we use fixed window counter with a queue of timestamps per user.

```
#include <unordered_map>
#include <queue>
#include <chrono>
using namespace std;
using namespace std::chrono;

class RateLimiter {
    int maxRequests;
    int windowSec;
    unordered_map<int, queue<long long>> userRequests;

public:
    RateLimiter(int maxReq, int window) : maxRequests(maxReq), windowSec(window) {}

    bool allowRequest(int userId) {
        auto now = duration_cast<seconds>(system_clock::now().time_since_epoch()).count();
        auto& q = userRequests[userId];

        while (!q.empty() && now - q.front() >= windowSec) {
            q.pop();
        }

        if (q.size() < maxRequests) {
            q.push(now);
            return true;
        }
```

```
            return false;
        }
    };
```

Complexity Cheat Sheet

LRU Cache: Time: O(1) for get/put | Space: O(capacity)

Top K Frequent Elements: Time: O(N log K) | Space: O(N)

Rotate Matrix: Time: O(N^2) | Space: O(1)

Concurrency: FooBar Alternately: Time: O(N) per thread | Space: O(1)

Rate Limiter: Time: O(1) amortized per request | Space: O(users * requests)