**ATM SERVICE**

This solution uses a set to store unique ATM IDs and a priority queue to store service requests. The priority queue automatically orders the requests by priority level, with lower numbers indicating higher priority. The solution then iterates through the priority queue and adds unique service requests to a list while ignoring duplicate ATM IDs. Finally, the list is sorted by region number in ascending order and printed out in the appropriate order.

import java.util.\*;

public class ServiceOrder {

public static void main(String[] args) {

// create a set to store unique ATM IDs

Set<Integer> uniqueIds = new HashSet<>();

// create a priority queue to store service requests

PriorityQueue<ServiceRequest> serviceRequests = new PriorityQueue<>();

// add sample service requests

serviceRequests.add(new ServiceRequest(1, 5, 2)); // ATM ID 1, region 5, low cash priority

serviceRequests.add(new ServiceRequest(2, 3, 1)); // ATM ID 2, region 3, emergency priority

serviceRequests.add(new ServiceRequest(3, 2, 3)); // ATM ID 3, region 2, standard priority

serviceRequests.add(new ServiceRequest(4, 1, 2)); // ATM ID 4, region 1, low cash priority

serviceRequests.add(new ServiceRequest(5, 4, 3)); // ATM ID 5, region 4, standard priority

serviceRequests.add(new ServiceRequest(1, 5, 2)); // duplicate request for ATM ID 1

// iterate through the service requests and add them to the unique IDs set

List<ServiceRequest> uniqueServiceRequests = new ArrayList<>();

while (!serviceRequests.isEmpty()) {

ServiceRequest request = serviceRequests.poll();

if (!uniqueIds.contains(request.getAtmId())) {

uniqueIds.add(request.getAtmId());

uniqueServiceRequests.add(request);

}

}

// sort the unique service requests by region number in ascending order

Collections.sort(uniqueServiceRequests, new Comparator<ServiceRequest>() {

@Override

public int compare(ServiceRequest r1, ServiceRequest r2) {

return Integer.compare(r1.getRegion(), r2.getRegion());

}

});

// print the unique service requests in the appropriate order

for (ServiceRequest request : uniqueServiceRequests) {

System.out.println(request);

}

}

}

class ServiceRequest implements Comparable<ServiceRequest> {

private int atmId;

private int region;

private int priority;

public ServiceRequest(int atmId, int region, int priority) {

this.atmId = atmId;

this.region = region;

this.priority = priority;

}

public int getAtmId() {

return atmId;

}

public int getRegion() {

return region;

}

public int getPriority() {

return priority;

}

@Override

public int compareTo(ServiceRequest other) {

// compare service requests by priority level, then by region number

int result = Integer.compare(this.getPriority(), other.getPriority());

if (result == 0) {

result = Integer.compare(this.getRegion(), other.getRegion());

}

return result;

}

@Override

public String toString() {

return "ATM ID: " + atmId + ", Region: " + region + ", Priority: " + priority;

}

}

**ONLINE GAME**

This algorithm uses the Clan class to represent a clan, and implements the Comparable interface to allow clans to be sorted based on their points, size, and ID. It then reads in the input values, sorts the clans in decreasing order of points, and divides them into groups according to the given rules. Finally, it outputs the clan IDs in the order in which they should enter the event.

import java.util.\*;

class Clan implements Comparable<Clan> {

int id;

int size;

int points;

Clan(int id, int size, int points) {

this.id = id;

this.size = size;

this.points = points;

}

@Override

public int compareTo(Clan other) {

if (this.points != other.points) {

return other.points - this.points;

} else if (this.size != other.size) {

return this.size - other.size;

} else {

return this.id - other.id;

}

}

}

public class EventPlanner {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int m = scanner.nextInt();

int n = scanner.nextInt();

Clan[] clans = new Clan[n];

for (int i = 0; i < n; i++) {

int size = scanner.nextInt();

int points = scanner.nextInt();

clans[i] = new Clan(i + 1, size, points);

}

Arrays.sort(clans);

int groupSize = Math.min(m, clans[0].size);

int groups = (n - 1) / (m / groupSize) + 1;

List<Integer>[] groupMembers = new List[groups];

for (int i = 0; i < groups; i++) {

groupMembers[i] = new ArrayList<Integer>();

}

int groupIndex = 0;

int groupPoints = 0;

for (Clan clan : clans) {

if (groupPoints + clan.points > m \* groupSize) {

groupIndex++;

groupPoints = 0;

groupSize = Math.min(m, clan.size);

}

groupPoints += clan.points;

groupMembers[groupIndex].add(clan.id);

}

for (int i = 0; i < groups; i++) {

System.out.println(String.join(" ", groupMembers[i].stream().map(Object::toString).toArray(String[]::new)));

}

scanner.close();

}

}

**TRANSACTIONS**

This solution processes transactions by iterating over each transaction, updating the balance of the corresponding account, and then creating a list of accounts sorted by account ID. The Account class represents an account and keeps track of its balance, number of credits, and number of debits. The Transaction class represents a single transaction with an account ID and an amount.

The solution uses a HashMap to store account balances for fast lookup, and a custom Comparator to sort the list of accounts by account ID.

import java.util.\*;

public class TransactionProcessor {

private Map<String, Integer> accountBalances = new HashMap<>(); // to store account balances

public List<Account> processTransactions(List<Transaction> transactions) {

// calculate account balances

for (Transaction t : transactions) {

String accountId = t.getAccountId();

int amount = t.getAmount();

if (!accountBalances.containsKey(accountId)) {

accountBalances.put(accountId, 0);

}

int balance = accountBalances.get(accountId);

balance += amount;

accountBalances.put(accountId, balance);

}

// create list of accounts sorted by account ID

List<Account> accounts = new ArrayList<>();

for (String accountId : accountBalances.keySet()) {

int balance = accountBalances.get(accountId);

Account account = new Account(accountId, balance);

accounts.add(account);

}

Collections.sort(accounts, new AccountComparator());

return accounts;

}

private static class AccountComparator implements Comparator<Account> {

public int compare(Account a1, Account a2) {

return a1.getAccountId().compareTo(a2.getAccountId());

}

}

}

class Account {

private String accountId;

private int balance;

private int numCredits;

private int numDebits;

public Account(String accountId, int balance) {

this.accountId = accountId;

this.balance = balance;

this.numCredits = balance >= 0 ? 1 : 0; // initial credit or debit counts

this.numDebits = balance < 0 ? 1 : 0;

}

public String getAccountId() {

return accountId;

}

public int getBalance() {

return balance;

}

public int getNumCredits() {

return numCredits;

}

public int getNumDebits() {

return numDebits;

}

public void addCredit() {

numCredits++;

}

public void addDebit() {

numDebits++;

}

}

class Transaction {

private String accountId;

private int amount;

public Transaction(String accountId, int amount) {

this.accountId = accountId;

this.amount = amount;

}

public String getAccountId() {

return accountId;

}

public int getAmount() {

return amount;

}

}