# 1. Quoit Design

#include <iostream>

#include <vector>

#include <cmath>

#include <algorithm>

#include <iomanip>

using namespace std;

struct Point {

double x, y;

};

struct Circle {

Point center;

double radius;

};

// 计算两点之间的距离

double distance(const Point& p1, const Point& p2) {

return sqrt((p1.x - p2.x) \* (p1.x - p2.x) + (p1.y - p2.y) \* (p1.y - p2.y));

}

// 检查点是否在圆内

bool isInside(const Point& p, const Circle& c) {

return distance(p, c.center) < c.radius;

}

// 寻找一组点的最小外接圆

Circle findMinEnclosingCircle(vector<Point>& points) {

if (points.size() == 0) {

return {{0, 0}, 0};

} else if (points.size() == 1) {

return {points[0], 0};

} else if (points.size() == 2) {

Point center = {(points[0].x + points[1].x) / 2, (points[0].y + points[1].y) / 2};

double radius = distance(points[0], points[1]) / 2;

return {center, radius};

}

random\_shuffle(points.begin(), points.end());

Circle minCircle = {{0, 0}, 0};

minCircle.center = {(points[0].x + points[1].x) / 2, (points[0].y + points[1].y) / 2};

minCircle.radius = distance(points[0], points[1]) / 2;

for (int i = 2; i < points.size(); ++i) {

if (!isInside(points[i], minCircle)) {

minCircle.center = {(points[0].x + points[i].x) / 2, (points[0].y + points[i].y) / 2};

minCircle.radius = distance(points[0], points[i]) / 2;

vector<Point> remainingPoints(points.begin(), points.begin() + i);

minCircle = findMinEnclosingCircle(remainingPoints);

}

}

return minCircle;

}

int main() {

int N;

while (true) {

cin >> N;

if (N == 0) {

break; // 当 N 为 0 时终止程序

}

vector<Point> points(N);

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

cin >> points[i].x >> points[i].y;

}

Circle result = findMinEnclosingCircle(points);

cout << fixed << setprecision(2) << result.radius << endl;

}

return 0;

}

使用 findMinEnclosingCircle 函数找到每组点的最小外接圆，并以两位小数的精度打印圆的半径

**5.Uniform Generator**

#include <iostream>

#include <set>

using namespace std;

// 计算 (a^b) % m

int modpow(int a, int b, int m) {

int result = 1;

while (b > 0) {

if (b % 2 == 1) {

result = (result \* a) % m;

}

a = (a \* a) % m;

b /= 2;

}

return result;

}

int main() {

int step, mod;

while (cin >> step >> mod) {

cout << right << setw(10) << step << right << setw(10) << mod;

set<int> visited;

bool isUniform = true;

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

int curr = (i == 0) ? 0 : (modpow(curr, step, mod) + step) % mod;

if (visited.find(curr) != visited.end()) {

isUniform = false;

break;

}

visited.insert(curr);

}

if (isUniform && visited.size() == mod) {

cout << left << setw(5) << "Good Choice" << endl;

} else {

cout << left << setw(5) << "Bad Choice" << endl;

}

cout << endl;

}

return 0;

}

定义了一个辅助函数modpow来计算 (a^b) % m。然后，在主函数中循环读取输入的STEP和MOD值。对于每个测试组合，我们使用一个集合来跟踪已经生成的伪随机数，并使用一个布尔变量isUniform来跟踪是否生成了所有的数字。然后，使用分治法生成一系列的伪随机数，并将它们插入到集合中。如果我们发现重复的数字，或者集合的大小小于MOD，则isUniform被设置为false。最后根据isUniform和集合的大小输出结果。

**6.Max Sum**

#include <iostream>

#include <vector>

using namespace std;

// 分治法求解最大子序列和

int maxSubSeqSum(vector<int>& nums, int start, int end) {

if (start == end) {

return nums[start];

}

int mid = (start + end) / 2;

int leftMax = maxSubSeqSum(nums, start, mid);

int rightMax = maxSubSeqSum(nums, mid + 1, end);

int leftBorderMax = nums[mid];

int leftSum = leftBorderMax;

for (int i = mid - 1; i >= start; i--) {

leftSum += nums[i];

leftBorderMax = max(leftSum, leftBorderMax);

}

int rightBorderMax = nums[mid + 1];

int rightSum = rightBorderMax;

for (int i = mid + 2; i <= end; i++) {

rightSum += nums[i];

rightBorderMax = max(rightSum, rightBorderMax);

}

return max(max(leftMax, rightMax), leftBorderMax + rightBorderMax);

}

int main() {

int T;

cin >> T;

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

int N;

cin >> N;

vector<int> nums(N);

for (int j = 0; j < N; j++) {

cin >> nums[j];

}

cout << "Case " << i+1 << ":" << endl;

int maxSum = maxSubSeqSum(nums, 0, N-1);

int start = 0;

int end = 0;

int sum = 0;

int tempStart = 0;

for (int j = 0; j < N; j++) {

sum += nums[j];

if (sum > maxSum) {

maxSum = sum;

start = tempStart;

end = j;

}

if (sum < 0) {

sum = 0;

tempStart = j + 1;

}

}

cout << maxSum << " " << start+1 << " " << end+1 << endl;

cout << endl;

}

return 0;

}

先定义了一个辅助函数maxSubSeqSum，该函数以递归的方式将序列划分为左右两个部分，并返回三种情况中的最大值：左半部分的最大子序列和、右半部分的最大子序列和或跨越中心点的最大子序列和。

在主函数中，我们首先读取测试用例的数量T，然后循环处理每个测试用例。对于每个测试用例，我们首先读取序列的长度N和具体的数字。然后，我们调用maxSubSeqSum函数来计算最大子序列和。接下来，我们使用一个for循环来计算最大子序列和的起始位置和结束位置。我们将变量sum初始化为0，并使用一个临时变量tempStart来记录当前子序列的起始位置。如果sum大于maxSum，则更新maxSum、start和end的值；如果sum小于0，则将sum重置为0，并将tempStart设置为当前位置加1。

**7.Eddy's digital Roots**

#include <iostream>

using namespace std;

// 计算一个正整数的数字根

int digitalRoot(int n) {

if (n < 10) {

return n;

}

int sum = 0;

while (n != 0) {

sum += n % 10;

n /= 10;

}

return digitalRoot(sum);

}

int main() {

int n;

cin >> n;

while (n != 0) {

int result = 1;

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

result \*= n;

}

cout << digitalRoot(result) << endl;

cin >> n;

}

return 0;

}

该程序首先定义了一个辅助函数digitalRoot，用于计算一个正整数的数字根。如果输入的数字小于10，则直接返回该数字。否则，我们将对该数字的各个位数进行求和，然后递归调用digitalRoot函数继续计算。

在主函数中，我们首先读取一个正整数n，然后使用一个while循环来不断读取下一个n并处理。对于每个n，我们先将result初始化为1，然后使用一个for循环将n自乘n次得到结果。最后，我们调用digitalRoot函数来计算结果的数字根，并输出。

**8.Island of Logic**

#include <iostream>

#include <vector>

#include <algorithm>

#include <cstring>

using namespace std;

// 定义居民类型的枚举

enum InhabitantType {

Divine,

Human,

Evil,

Lying

};

// 居民结构体

struct Inhabitant {

char name; // 名字

InhabitantType type; // 类型

bool isDay; // 是否是白天

};

// 根据居民名称查找居民结构体

Inhabitant\* findInhabitant(char name, vector<Inhabitant>& inhabitants) {

for (int i = 0; i < inhabitants.size(); i++) {

if (inhabitants[i].name == name) {

return &inhabitants[i];

}

}

return nullptr;

}

// 判断是否存在矛盾的陈述

bool hasContradiction(vector<Inhabitant>& inhabitants, bool isDay) {

for (int i = 0; i < inhabitants.size(); i++) {

Inhabitant& speaker = inhabitants[i];

// 如果说谎者在白天说了真话，或者在晚上说了谎话，就产生了矛盾

if ((speaker.type == Lying && speaker.isDay && isDay) ||

(speaker.type == Lying && !speaker.isDay && !isDay)) {

return true;

}

// 如果说实话的人在说自己是邪恶的，或者在说其他人是神圣的，就产生了矛盾

if ((speaker.type != Lying && speaker.isDay && isDay && speaker.type == Evil) ||

(speaker.type != Lying && !speaker.isDay && !isDay && speaker.type == Divine)) {

return true;

}

// 如果说谎者对说实话的人的描述是错误的，就产生了矛盾

if (speaker.type == Lying) {

Inhabitant\* describedInhabitant = findInhabitant(speaker.name, inhabitants);

if (describedInhabitant != nullptr && describedInhabitant->type != Lying &&

((describedInhabitant->type == Divine && speaker.type != Divine) ||

(describedInhabitant->type != Divine && speaker.type == Divine))) {

return true;

}

}

}

return false;

}

int main() {

int caseNum = 1; // 标记对话的编号

int n;

cin >> n;

while (n != 0) {

vector<Inhabitant> inhabitants; // 存储居民信息

// 读取对话

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

char name;

string statement;

cin >> name;

getline(cin, statement);

Inhabitant inhabitant;

inhabitant.name = name;

if (statement.find("divine") != string::npos) {

inhabitant.type = Divine;

} else if (statement.find("human") != string::npos) {

inhabitant.type = Human;

} else if (statement.find("evil") != string::npos) {

inhabitant.type = Evil;

} else if (statement.find("lying") != string::npos) {

inhabitant.type = Lying;

}

if (statement.find("day") != string::npos) {

inhabitant.isDay = true;

} else {

inhabitant.isDay = false;

}

inhabitants.push\_back(inhabitant);

}

cout << "Conversation #" << caseNum << endl;

// 排除不可能的情况

if (hasContradiction(inhabitants, true) || hasContradiction(inhabitants, false)) {

cout << "This is impossible." << endl;

} else {

bool factsDeducible = false;

sort(inhabitants.begin(), inhabitants.end(), [](const Inhabitant& a, const Inhabitant& b){

return a.name < b.name;

});

// 输出推断出的事实

for (int i = 0; i < inhabitants.size(); i++) {

if (inhabitants[i].type != Lying) {

cout << inhabitants[i].name << " is ";

switch(inhabitants[i].type) {

case Divine:

cout << "divine";

break;

case Human:

cout << "human";

break;

case Evil:

cout << "evil";

break;

}

cout << "." << endl;

factsDeducible = true;

}

}

// 输出白天或晚上的事实

if (!hasContradiction(inhabitants, true)) {

cout << "It is day." << endl;

factsDeducible = true;

}

if (!hasContradiction(inhabitants, false)) {

cout << "It is night." << endl;

factsDeducible = true;

}

if (!factsDeducible) {

cout << "No facts are deducible." << endl;

}

}

cout << endl;

cincaseNum

这个变量是用来标记对话的编号。在每个测试用例开始时，我们将其初始化为1。

struct Inhabitant

这是一个结构体，用于存储居民的信息。它包含三个成员变量：

name：居民的名字，类型为字符。

type：居民的类型，可以是神圣（Divine）、人类（Human）、邪恶（Evil）或说谎（Lying）。这是一个枚举类型。

isDay：表示当前是否是白天的布尔值。

findInhabitant(char name, vector<Inhabitant>& inhabitants)

这是一个辅助函数，用于根据居民的名字在居民列表中查找对应的居民结构体。它返回指向该居民结构体的指针，如果未找到则返回nullptr。

hasContradiction(vector<Inhabitant>& inhabitants, bool isDay)

这是一个辅助函数，用于判断对话中是否存在矛盾的陈述。它接受一个存储居民信息的向量和一个表示白天或晚上的布尔值作为输入。

在函数内部，它检查每个居民的陈述，并根据规则判断是否存在矛盾。

如果发现任何矛盾，函数将返回true；否则，返回false。

主函数

主函数从输入读取对话的数量n，并在n不为0时执行以下操作：

创建一个容器vector<Inhabitant>，用于存储居民信息。

读取n个陈述，并将每个陈述的信息存储在居民结构体中，然后将该结构体添加到容器中。

输出对话的编号（caseNum）。

检查是否存在矛盾的陈述。如果存在矛盾，则输出"This is impossible."。

否则，对居民信息进行排序，并按照规定的格式输出推断出的事实。

输出白天或晚上的事实（如果可以推断出来）。

如果无法推断出任何事实，则输出"No facts are deducible."。

这就是这个程序的主要逻辑和功能。它通过检查陈述的规则和推理逻辑来推断出可能的事实，并根据规定的格式输出结果。

**9.Picture**

定义一个结构体来表示矩形，包含左下角和右上角的坐标。

创建一个递归函数calculatePerimeter(vector<Rectangle>& rectangles, int left, int right)来计算给定矩形列表中[left, right]范围内的矩形的周长。

在该函数中，如果left > right，说明没有矩形需要处理，返回0。

如果left == right，说明只有一个矩形，直接计算并返回其周长。

否则，将[left, right]范围内的矩形分成两部分，即[left, mid]和[mid+1, right]。

递归调用calculatePerimeter(rectangles, left, mid)和calculatePerimeter(rectangles, mid+1, right)来计算这两部分的周长。

然后，计算左边部分的最右边界和右边部分的最左边界，并根据它们之间的距离来计算两部分之间的连接线段的长度。

最后，将左、右部分的周长以及连接线段的长度相加，并返回结果。

#include <iostream>

#include <vector>

using namespace std;

struct Rectangle {

int x1, y1, x2, y2;

};

int calculatePerimeter(vector<Rectangle>& rectangles, int left, int right) {

if (left > right) {

return 0;

} else if (left == right) {

Rectangle& rect = rectangles[left];

return 2 \* (rect.x2 - rect.x1 + rect.y2 - rect.y1);

} else {

int mid = left + (right - left) / 2;

int perimeterLeft = calculatePerimeter(rectangles, left, mid);

int perimeterRight = calculatePerimeter(rectangles, mid + 1, right);

int maxRight = rectangles[mid].x2;

int minLeft = rectangles[mid + 1].x1;

int connectingLine = 0;

if (maxRight < minLeft) {

connectingLine = minLeft - maxRight;

}

return perimeterLeft + perimeterRight + connectingLine;

}

}

int main() {

int numRectangles;

cin >> numRectangles;

vector<Rectangle> rectangles(numRectangles);

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

cin >> rectangles[i].x1 >> rectangles[i].y1 >> rectangles[i].x2 >> rectangles[i].y2;

}

cout << calculatePerimeter(rectangles, 0, numRectangles - 1) << endl;

return 0;

}

**10.汉诺塔III**

将N-1个盘子从最左边移动到中间杆；

将第N个盘子从最左边移动到最右边；

将N-1个盘子从中间杆移动到最右边。

然而，在这个变种的问题中，我们不能直接从最左(右)边移到最右(左)边，也不能将大盘放到小盘的上面。因此，我们需要修改以上步骤。

具体的解决方案如下：

如果N=1，那么只需将盘子从最左边移到最右边，移动次数为2。

对于N>1的情况，我们可以将问题分解为三个子问题： a. 将N-1个盘子从最左边移动到中间杆，移动次数为A； b. 将第N个盘子从最左边移动到中间杆，移动次数为1； c. 将N-1个盘子从中间杆移动到最右边，移动次数为B。

在子问题a和子问题c中，我们需要将盘子从一个杆移到另一个杆，但是不能直接从最左(右)边移到最右(左)边。因此，我们需要交换中间杆和最右(左)边的位置。

子问题的解可以通过递归来求解。

#include <iostream>

long long hanoi(int N, char from, char to, char aux) {

if (N == 1) {

// 只需将盘子从from移动到to，移动次数为2

return 2;

} else {

// 分治解法

long long A = hanoi(N-1, from, aux, to); // 移动N-1个盘子从from到aux

long long B = 1; // 移动第N个盘子从from到to

long long C = hanoi(N-1, aux, to, from); // 移动N-1个盘子从aux到to

// 返回总移动次数

return A + B + C;

}

}

int main() {

int N;

while (std::cin >> N) {

long long moves = hanoi(N, 'A', 'B', 'C');

std::cout << moves << std::endl;

}

return 0;

}