**12.19测验考试**

**完成情况**

**💡平均分：225（本次题目较难，大家不要过度焦虑**）

**题目详解**

**#239. 乘积最大**

**【解题思路】**

输入处理：读入数字 **n** 和乘号个数 **k**，并将数字 **n** 分解为数组 **a** 中的每一位。

递归函数 ：这个函数接受一个参数 **m**，表示还需要放置的乘号数量。它递归地尝试所有可能的乘号位置，并计算每种情况的乘积。如果找到一个乘积比当前最大乘积 **best** 更大的情况，就更新 **best**。

计算乘积：在递归函数中，使用了 **pow** 函数来计算乘积，将每个分割后的数字片段转换成整数。

穷举乘号位置：函数通过修改 **used** 数组来记录乘号的位置，并通过递归调用 **zhao** 来穷举所有可能的位置。

输出结果：最后，打印出最大的乘积 **best**。

示例代码

#include <stdio.h>

#include <string.h>

int n, k, len;

int a[100], best = 0;

char numStr[100];

// 计算字符串s从下标start到end（包括end）的子串的整数值

int substringToInt(int start, int end) {

int value = 0;

for (int i = start; i <= end; i++) {

value = value \* 10 + a[i];

}

return value;

}

// 递归函数来穷举乘号位置

void findMaxProduct(int pos, int remainingK, int currentProduct) {

if (remainingK == 0) {

int product = currentProduct \* substringToInt(pos, len - 1);

if (product > best) {

best = product;

}

return;

}

for (int i = pos; i < len - remainingK; i++) {

int partialProduct = substringToInt(pos, i);

findMaxProduct(i + 1, remainingK - 1, currentProduct \* partialProduct);

}

}

int main() {

scanf("%s %d", numStr, &k);

len = strlen(numStr);

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

a[i] = numStr[i] - '0';

}

findMaxProduct(0, k, 1);

printf("%d", best);

return 0;

}

**# 862. 3个物品背包问题**

【解题思路】经典的背包问题的解决方案，使用深度优先搜索（DFS）算法。

变量定义：**v[i]** 和 **w[i]** 分别存储第 **i** 个物品的价值和重量。**w\_limit** 是背包的重量限制。**vm** 和 **wm** 分别表示当前选择的物品的总价值和总重量。**mx** 存储最大价值。**ans** 和 **trueans** 分别存储当前选择和最优选择。

深度优先搜索（DFS）：**dfs** 函数递归地遍历每个物品，考虑是否选择该物品（通过 **i** 取 0 或 1 表示）。如果选择当前物品不会导致总重量超过限制，则更新当前总价值和总重量，然后递归地考虑下一个物品。

更新最优解：当考虑完所有物品后（即 **x == 4**），如果当前总价值大于已知的最大价值 **mx**，则更新 **mx** 和 **trueans**。

示例代码

#include <stdio.h>

int ans[4], v[4], w[4], w\_limit, vm, wm, mx = -9999, trueans[4];

void dfs(int x) {

    int i;

    if (x == 4) {

        if (vm > mx) {

            mx = vm;

            for (i = 1; i <= 3; i++) {

                trueans[i] = ans[i];

            }

        }

        return;

    }

    for (i = 1; i >= 0; i--) {

        int tv = vm, tw = wm;

        if (wm + i \* w[x] <= w\_limit) {

            vm += i \* v[x];

            wm += i \* w[x];

            ans[x] = i;

            dfs(x + 1);

            ans[x] = 0;

            vm = tv;

            wm = tw;

        }

    }

    return;

}

int main() {

    int i;

    for (i = 1; i <= 3; i++)

        scanf("%d", &v[i]);

    for (i = 1; i <= 3; i++)

        scanf("%d", &w[i]);

    scanf("%d", &w\_limit);

    dfs(1);

    for (i = 1; i <= 3; i++)

        printf("%d ", trueans[i]);

    return 0;

}

#include <stdio.h>

int main() {

int a, b, c, d, e, f, n;

int maxBenefit = 0, minCostWeight = 9999; // 假设初始成本权重非常大

int bestI = 0, bestJ = 0, bestT = 0; // 最佳选择

for (int i = 0; i <= 1; i++) {

for (int j = 0; j <= 1; j++) {

for (int t = 0; t <= 1; t++) {

int cost = i \* d + j \* e + t \* f;

if (cost <= n) {

int benefit = i \* a + j \* b + t \* c;

int costWeight = i \* 1 + j \* 2 + t \* 3;

if (benefit > maxBenefit || (benefit == maxBenefit && costWeight < minCostWeight)) {

maxBenefit = benefit;

minCostWeight = costWeight;

bestI = i;

bestJ = j;

bestT = t;

}

}

}

}

}

printf("%d %d %d", bestI, bestJ, bestT);

return 0;

}

**##597. matrix**

【解题思路】

主要思路

递归搜索：**move** 函数递归地搜索所有可能的路径。每次递归都考虑从当前位置出发，向上、下、左、右四个方向移动，每个方向移动的步数从 1 到 **matrix[x][y]**。

更新最短路径：如果到达目的地 **(n-1,n-1)**，则更新最小步数 **\*p**。在每次递归中，如果找到了一条路径的步数小于当前记录的最小步数，也更新最小步数。

避免重复访问：为了防止无限循环，每次从一个单元格出发时，会将其值暂时设置为 0，避免重复访问。visited 矩阵来存储每个单元格的最小访问步数，并且用数组来表示方向。

示例代码

#include <stdio.h>

#include <stdlib.h>

int move(int \*\*matrix, int \*\*visited, int x, int y, int n, int done, int \*p) {

if (done >= \*p || visited[x][y] <= done) {

return 1000;

}

visited[x][y] = done;

if (x == n - 1 && y == n - 1) {

\*p = done;

return 0;

}

int dx[4] = {1, -1, 0, 0};

int dy[4] = {0, 0, 1, -1};

int min = 1000;

for (int dir = 0; dir < 4; dir++) {

for (int i = 1; i <= matrix[x][y]; i++) {

int newX = x + dx[dir] \* i;

int newY = y + dy[dir] \* i;

int tmp = matrix[x][y];

matrix[x][y] = 0;

int step = 1 + move(matrix, visited, newX, newY, n, done + 1, p);

if (step < min) {

min = step;

}

matrix[x][y] = tmp;

}

}

}

return min;

}

int main() {

int n, step, leading = 1000;

scanf("%d", &n);

int \*\*matrix = (int\*\*)malloc(n \* sizeof(int\*));

int \*\*visited = (int\*\*)malloc(n \* sizeof(int\*));

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

matrix[i] = (int\*)malloc(n \* sizeof(int));

visited[i] = (int\*)malloc(n \* sizeof(int));

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

scanf("%d", &matrix[i][j]);

visited[i][j] = 1000;

}

}

step = move(matrix, visited, 0, 0, n, 0, &leading);

printf("%d", step);

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

free(matrix[i]);

free(visited[i]);

}

free(matrix);

free(visited);

return 0;

}

**# 227. 连连看**

【解题思路】

主要思路

初始化和输入：代码首先读取网格的尺寸（宽度 **w** 和高度 **h**），然后读取网格本身。之后读取路径查询的数量 **n** 以及每个查询的起点和终点坐标。

深度优先搜索（DFS）：**dfs** 函数用于遍历网格，尝试从起点到终点的所有路径。它递归地尝试网格中的每个点，直到找到终点或探索完所有可能的路径。

方向和状态：使用 **drct** 数组表示移动的四个方向（上、下、左、右）。**p** 数组用于标记网格中的点是否已经被访问过，以避免重复访问。**c** 结构体数组用于存储每个点在两个方向（水平和垂直）上的最少转弯次数。

#include <stdio.h>

char mp[77][77];

int w, h, n, x1, y\_, x2, y2, ans[100], mn;

struct ps {

    int nx;

    int ny;

};

ps c[77][77];

int drct[4][2] = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}}, p[77][77];

void dfs(int cnt, int dr, int x, int y) {

    int i, j;

    //printf("%d %d %d\n", x, y, cnt);

    if (x == x2 && y == y2) {

        if (cnt < mn)

            mn = cnt;

        return;

    }

    //printf("%d %d %d\n", x, y, cnt);

    for (i = 0; i <= 3; i++) {

        int xx = x + drct[i][0], yy = y + drct[i][1];

         && mp[xx][yy] == 'X')) {

            p[xx][yy] = 1;

            if (i <= 1) {

                if (dr == 0 && cnt + 1 < c[xx][yy].nx) {

                    c[xx][yy].nx = cnt + 1;

                    dfs(cnt + 1, 1, xx, yy);

                }

                if (dr == 1 && cnt < c[xx][yy].nx) {

                    c[xx][yy].nx = cnt;

                    dfs(cnt, 1, xx, yy);

                }

            }

            if (i > 1) {

                if (dr == 0 && cnt < c[xx][yy].ny) {

                    c[xx][yy].ny = cnt;

                    dfs(cnt, 0, xx, yy);

                }

                if (dr == 1 && cnt + 1 < c[xx][yy].ny) {

                    c[xx][yy].ny = cnt + 1;

                    dfs(cnt + 1, 0, xx, yy);

                }

            }

            p[xx][yy] = 0;

        }

    }

    return;

}

int main() {

    int i, j;

    char a;

    scanf("%d%d", &w, &h);

    a = getchar();

    for (i = 1; i <= h; i++) {

        for (j = 1; j <= w; j++) //i=1,i+1%2

            mp[i][j] = getchar();

        a = getchar();

    }

    scanf("%d", &n);

    for (i = 1; i <= n; i++) {

        scanf("%d%d%d%d", &y\_, &x1, &y2, &x2);

        mn = 1000;

        for (int u = 0; u <= h + 1; u++)

            for (int v = 0; v <= w + 1; v++) {

                c[u][v].nx = 1000;

                c[u][v].ny = 1000;

            }

        for (j = 0; j <= 3; j++) {

            int xx = x1 + drct[j][0], yy = y\_ + drct[j][1];

            p[xx][yy] = 1;

                c[xx][yy].nx = 1;

                dfs(1, 1, xx, yy);

            }

                c[xx][yy].ny = 1;

                dfs(1, 0, xx, yy);

            }

            p[xx][yy] = 0;

        }

        ans[i] = mn;

    }

    for (i = 1; i <= n; i++) {

        if (ans[i] > 10)

            printf("impossible\n");

        else

            printf("%d\n", ans[i]);

    }

    return 0;

}

**#785. 平面最近点对**

解题思路

数据结构：使用 **Point** 结构体存储点的 **x**、**y** 坐标。**point** 数组存储所有点，**mpt** 数组用于分治算法中的辅助存储。

排序和比较函数：

距离计算函数 **dis**：计算两点间的欧氏距离的平方。

分治算法 **Closest\_Pair**：

如果区间只有一个点或两个点，直接计算距离。

否则，将点集划分为左右两部分，分别递归计算左右子集的最近点对距离。

从两部分中选择较小的距离作为当前最近距离 d

然后对距离中线，由中点 mid一定范围内的点进行线性扫描，更新最近距离。

示例代码

#include <iostream>

#include <cstdio>

#include <cstring>

#include <cmath>

#include <algorithm>

using namespace std;

const double inf = 1e20;

const int maxn = 1000000;

struct Point{ // 结构体定义点的坐标

double x, y;

}point[maxn];

int n, mpt[maxn];

//以x为基准排序按 x 坐标排序，若 x 相同则按 y 排序

bool cmpxy(const Point& a, const Point& b){

if (a.x != b.x)

return a.x < b.x;

return a.y < b.y;

}

// 按 y 坐标排序

bool cmpy(const int& a, const int& b){

return point[a].y < point[b].y;

}

double min(double a, double b){

return a < b ? a : b;

}

// 计算两点间距离的平方

double dis(int i, int j){

return (point[i].x - point[j].x)\*(point[i].x - point[j].x) + (point[i].y - point[j].y)\*(point[i].y - point[j].y);

}

// 分治算法求解最近点对问题

double Closest\_Pair(int left, int right){

double d = inf;

if (left == right)

return d;

if (left + 1 == right)

return dis(left, right);

int mid = (left + right) >> 1;

double d1 = Closest\_Pair(left, mid);

double d2 = Closest\_Pair(mid + 1, right);

d = min(d1, d2);

int i, j, k = 0;

//分离出宽度为d的区间

for (i = left; i <= right; i++){

if (pow((point[mid].x - point[i].x) , 2) <= d)

mpt[k++] = i;

}

sort(mpt, mpt + k, cmpy);

//线性扫描

for (i = 0; i < k; i++){

for (j = i + 1; j < k && pow(point[mpt[j]].y - point[mpt[i]].y , 2) < d ; j++){

double d3 = dis(mpt[i], mpt[j]);

if (d > d3)d = d3;

}

}

return d;

}

// 读入点集，排序并计算最近点对距离

int main(){

scanf("%d", &n);

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

scanf("%lf %lf", &point[i].x, &point[i].y);

sort(point, point + n, cmpxy);

printf("%.0lf", Closest\_Pair(0, n - 1));

return 0;

}

**12.12 课后题**

🎉平均分：290+

**题目详解**

**#202. 分书问题**

【解题思路】

输入处理：读取每个人对书的兴趣。每行一个数字 **x**，表示第 **i** 个人对每本书的兴趣（用二进制位表示）。

递归分配书籍：通过递归函数 **book** 对每个人分配书籍。

如果所有人都分配了书则记录当前分配方案。

否则，遍历所有书籍，为当前的人分配尚未分配且他感兴趣的书。

输出结果：如果有分配方案，则打印方案数量和具体的分配方案；如果没有，打印 "NO"。

示例代码

#include <stdio.h>

int n, a[10][10] = {0}, count = 0, visit[10] = {0}, result[5000][10] = {0}, take[10] = {0};

//给第i个人分书

void book(int i) {

    if (i == n) {

        for (int j = 0; j < n; j++)

            result[count][j] = take[j];//中间变量将存储分配方案传递给result

        count++;

        return;

    }

    for (int j = 0; j < n; j++)

        if (a[i][j] == 1 && visit[j] == 0) {

            //标记

            visit[j] = 1;

            take[i] = j + 1;//中间变量暂时存储分配方案

            //递归

            book(i + 1);

            //恢复

            visit[j] = 0;

        }

}

int main(void) {

    scanf("%d", &n);

    for (int i = 0; i < n; i++) {

        int x;

        scanf("%d", &x);

        for (int k = n - 1; k >= 0; k--) {

            a[i][k] = x % 10;

            x = x / 10;

        }

    }

    book(0);

    if (count) {

        printf("%d\n", count);

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

            for (int k = 0; k < n; k++)

                printf("B%d", result[i][k]);

            printf("\n");

        }

    } else

        printf("NO");

    return 0;

}

**#594. find**

【解题思路】

搜索函数 **search**：用于在矩阵的一个子区域中搜索给定的值 **s**。它首先检查边界情况，然后在矩阵的四个角尝试找到目标值。如果在角上没找到，它使用二分查找在行和列上定位可能包含目标值的更小的子矩阵，并递归地在这个子矩阵中搜索。

示例代码

#include <stdio.h>

int mp[1001][1001], ansx[10001], ansy[10001], n, k;

void search(int x1, int x2, int y1, int y2, int target, int cnt) {

if (x2 < x1 || y2 < y1) {

ansx[cnt] = -1;

return;

}

if (x2 == x1 && y2 == y1) {

if (mp[x1][y1] != target) {

ansx[cnt] = -1;

} else {

ansx[cnt] = x1;

ansy[cnt] = y1;

}

return;

}

// 检查四个方向

int corners[4][2] = {{x1, y1}, {x2, y2}, {x1, y2}, {x2, y1}};

for (int i = 0; i < 4; i++) {

if (mp[corners[i][0]][corners[i][1]] == target) {

ansx[cnt] = corners[i][0];

ansy[cnt] = corners[i][1];

return;

}

}

// 二分查找

int mid, left, right;

left = x1, right = x2;

while (left < right) {

mid = (left + right) / 2;

if (mp[mid][y2] >= target) right = mid;

else left = mid + 1;

}

int newX1 = left;

left = x1, right = x2;

while (left < right) {

mid = (left + right + 1) / 2;

if (mp[mid][y1] <= target) left = mid;

else right = mid - 1;

}

int newX2 = left;

left = y1, right = y2;

while (left < right) {

mid = (left + right) / 2;

if (mp[x2][mid] >= target) right = mid;

else left = mid + 1;

}

int newY1 = left;

left = y1, right = y2;

while (left < right) {

mid = (left + right + 1) / 2;

if (mp[x1][mid] <= target) left = mid;

else right = mid - 1;

}

int newY2 = left;

search(newX1, newX2, newY1, newY2, target, cnt);

}

int main() {

scanf("%d", &n);

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

for (int j = 0; j < n; j++)

scanf("%d", &mp[i][j]);

scanf("%d", &k);

for (int i = 1; i <= k; i++) {

int target;

scanf("%d", &target);

if (target > mp[n - 1][n - 1] || target < mp[0][0])

ansx[i] = -1;

else

search(0, n - 1, 0, n - 1, target, i);

}

for (int i = 1; i <= k; i++) {

if (ansx[i] == -1)

printf("-1\n");

else

printf("%d %d\n", ansx[i], ansy[i]);

}

return 0;

}

**#546. 约瑟夫环**

【解题思路】

**递归函数 taotai**：

如果只剩下一个对象，则输出这个对象的编号并结束。

根据淘汰的位置 ，来更新数组。如果 x 等于数组长度 n 或不等于 1，需要调整数组使得淘汰后的对象排在数组的前面

递归调用 taotai函数，处理减少一个对象后的新数组。

示例代码

include <stdio.h>

#include <math.h>

void taotai(int a[], int x, int n) {

    if (n == 1) {

        printf("%d", a[0]);

        return;

    }

    int m = x;

    int b[n];

    if (x == n) {

        taotai(a, x, n - 1);

    } else if (x == 1) {

        for (int i = 0; i < n; i++) {

            a[i] = a[i + 1];

        }

        taotai(a, x, n - 1);

    } else if (x < n && x != 1) {

        for (int i = 0; i < n; i++) {

            b[i] = a[i];

        }

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

            a[i] = b[m];

            m++;

        }

        m = 0;

        for (int i = n - x ; i < n - 1; i++) {

            a[i] = b[m];

            m++;

        }

        taotai(a, x, n - 1);

    } else {

        int t = x % n;

        for (int i = 0; i < n; i++) {

            b[i] = a[i];

        }

        m = t;

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

            a[i] = b[m];

            m++;

        }

        m = 0;

        for (int i = n - t ; i < n - 1; i++) {

            a[i] = b[m];

            m++;

        }

        taotai(a, x, n - 1);

    }

}

int main() {

    int p, q;

    scanf("%d%d", &p, &q);

    int a[p];

    for (int i = 0; i < p; i++) {

        a[i] = i + 1;

    }

    taotai(a, q, p);

}