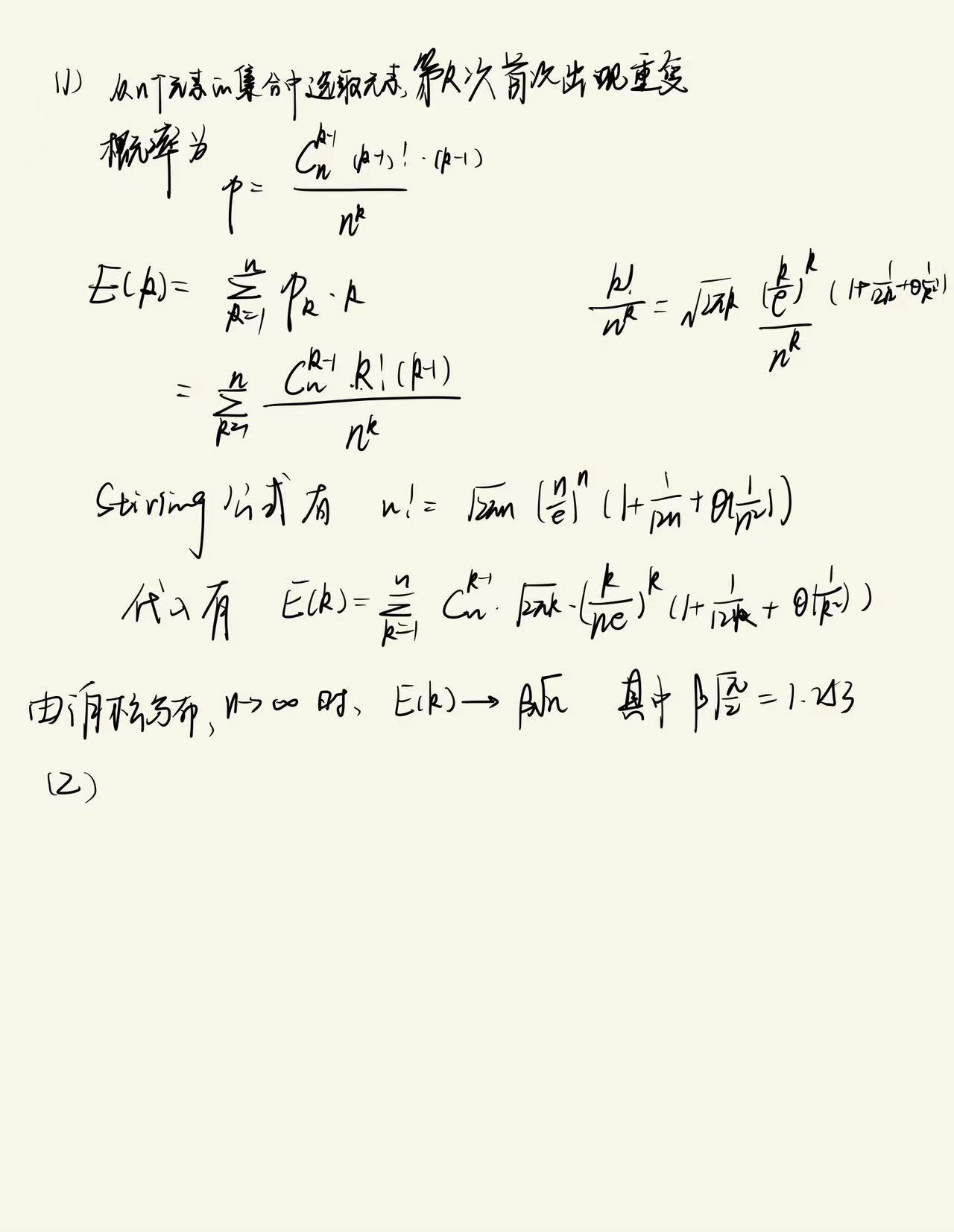
==============================作业7==============================

算法分析题7-4

思路：



代码实现：

(2)

int estimateSetSize(const std::vector<int>& setX) {

    unordered\_set<int> selected;

    int k = 0; // 记录选择的次数

    while (true) {

        // 从集合X中随机选择一个元素

        int index = rand() % setX.size();

        int element = setX[index];

        // 检查元素是否已经被选择过

        if (selected.find(element) != selected.end()) {

            // 如果元素已经被选择过，结束循环

            break;

        }

        // 将元素添加到已选择的集合中

        selected.insert(element);

        k++;

    }

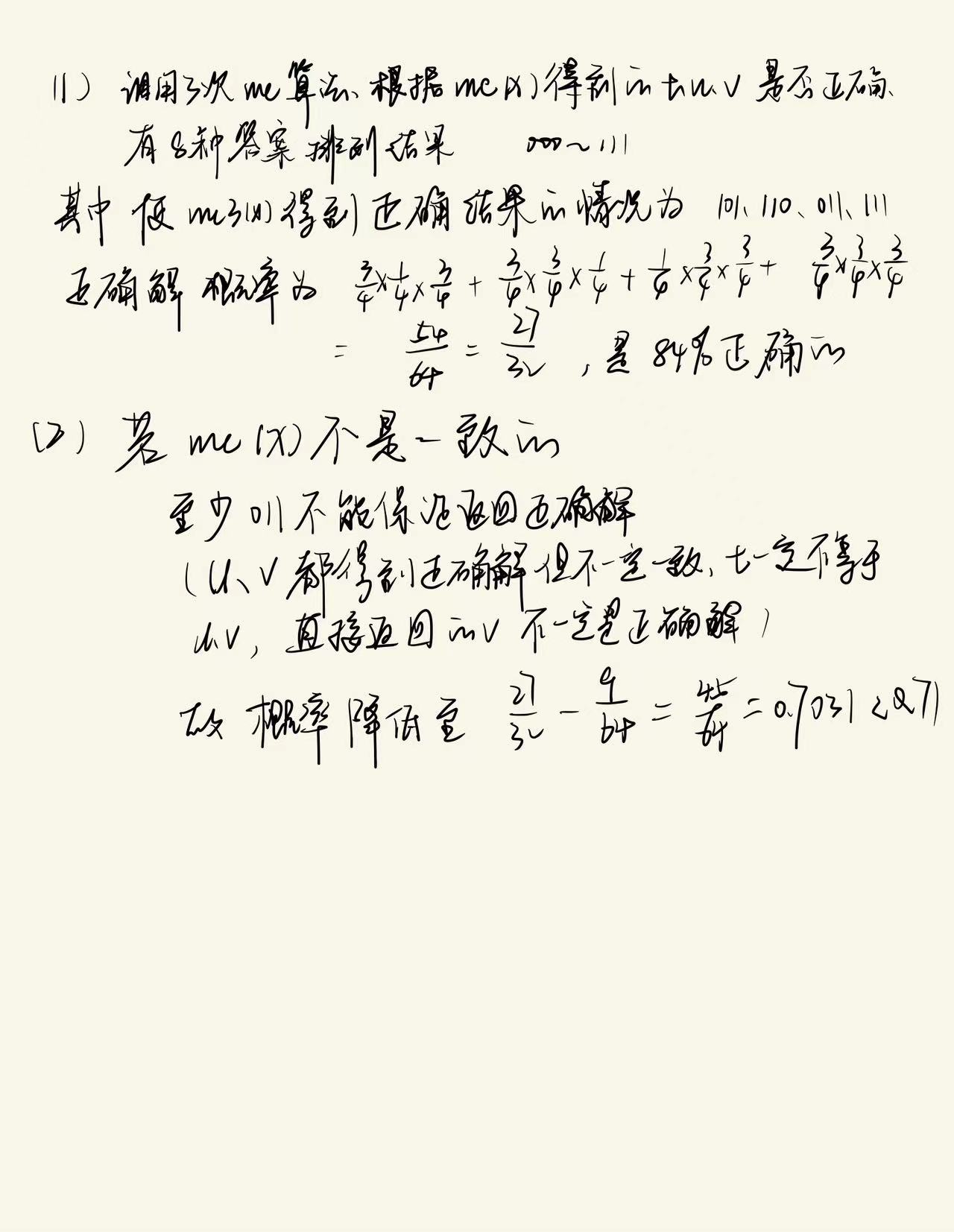
    // 使用观察到的选择次数k来估计集合X的大小

    double estimatedSize = 2 \* k \* k / M\_PI;

    return static\_cast<int>(std::round(estimatedSize));

}

算法分析题7-12



算法分析题7-14

实现：

bool LV(x){

    while(true){

        if(A(x))

        return true;

        if(!B(x))

        return false;

    }

}

算法实现题7-3

思路：

确认集合S和T的大小。如果它们的大小不相等，那么它们一定不相等，算法结束。如果大小相等，随机选择集合S中的若干（例如k次）元素。对于每次随机选择的元素，检查它是否也存在于集合T中。如果在k次随机选择中，每次选中的元素S中的元素都能在T中找到，则认为集合S和T有很高的概率相等；如果有一次选择中S的元素在T中找不到，则认为S和T不相等。

代码实现:

bool MC(const vector<int>& S, const vector<int>& T, int k) {

    if (S.size() != T.size()) return false; // 尺寸不相等，集合一定不相等

    srand(time(0)); // 初始化随机数生成器

    for (int i = 0; i < k; ++i) {

        int randIndex = rand() % S.size(); // 随机选择索引

        int elem = S[randIndex]; // 从集合S中随机选择一个元素

        if (find(T.begin(), T.end(), elem) == T.end()) {

            // 如果元素在集合T中找不到，返回false

            return false;

        }

    }

    // 在k次随机测试中，S中的元素都能在T中找到，认为S和T相等

    return true;

}

算法实现题7-4

思路：蒙特卡洛算法的关键在于利用随机性。在这个问题中，我们可以通过随机生成向量v，然后判断是否等于v来检验A和B是否互为逆矩阵。如果A和B互为逆矩阵，那么理论上应该对于所有v都成立。但在蒙特卡洛算法中，我们只随机检验一部分v，从而降低运算复杂度到O(n^2)。

代码实现:

bool areInverses(const vector<vector<int>>& A, const vector<vector<int>>& B, int n) {

    srand(time(0));

    for (int i = 0; i < 10; ++i) { // 进行10次检验以提高准确性

        vector<int> v = generateRandomVector(n);

        vector<int> Bv = matrixVectorMultiply(B, v);

        vector<int> ABv = matrixVectorMultiply(A, Bv);

        // 检验ABv是否等于v

        for (int j = 0; j < n; ++j) {

            if (ABv[j] - v[j] >1e-4) {

                return false; // 如果不接近，A和B不是互逆的

            }

        }

    }

    return true; // 如果在所有检验中ABv都等于v，那么很可能A和B是互逆的

==============================作业6==============================

算法分析题6-6

不能，在当前扩展结点数团顶点的上界应该是cn+n-i+1，如果条件改成cn+n-i>bestn，则上界为cn+n-i，上界不正确，最终不能保证算法正确性。

算法实现题6-1

思路:

设置一个全局变量bestd来存储目前找到的最短长度，并用一个数组bestx来存储对应的排列。这里的bestd初始化为一个较大的值，表示尚未找到有效解。将问题表示为一个状态空间树，其中每个节点表示一个部分解，即电路板的一个排列。

界限函数: 计算给定部分解的界限。如果当前部分解已经不能产生比当前bestd更好的解，则剪枝。

从当前部分解出发，通过交换元素生成新的部分解。采用深度优先搜索遍历状态空间树。每次遍历到叶子节点时，计算当前解的长度，并更新bestd和bestx。

代码实现：

#include<bits/stdc++.h>

using namespace std;

int n, m;

int bestx[10]; *// 最终的最优解排列顺序*

int B[10][10]; *// 电路板在连接块中的排列，是一个二维数组*

int x[10]; *// 当前的排列*

int bestd = *INT\_MAX*; *// 最优解，初始化为最大整数*

*// 计算当前排列最小长度*

int *len*() {

    int maxLen = 0;

    for (int i = 1; i <= m; i++) {

        int minPos = n + 1, maxPos = 0;

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

            if (B[x*[*j*]*][i] > 0) {

                minPos = *min*(minPos, j);

                maxPos = *max*(maxPos, j);

            }

        }

        maxLen = *max*(maxLen, maxPos - minPos);

    }

    return maxLen;

}

*// 交换i和j位置的值*

void *swap*(int& *a*, int& *b*) {

    int temp = *a*;

*a* = *b*;

*b* = temp;

}

*// 分支界限法核心函数*

void *backtrack*(int *i*) {

    if (*i* > n) { *// 如果到达末尾*

        int curLen = *len*(); *// 计算当前排列最小长度*

        if (curLen < bestd) { *// 如果比最优解还要好，则更新*

            bestd = curLen;

            for (int j = 1; j <= n; j++) bestx*[*j*]* = x*[*j*]*;

        }

    } else {

        for (int j = *i*; j <= n; j++) {

*swap*(x*[i]*, x*[*j*]*);

            if (*len*() < bestd) { *// 剪枝条件*

*backtrack*(*i* + 1); *// 继续搜索*

            }

*swap*(x*[i]*, x*[*j*]*); *// 回溯*

        }

    }

}

int *main*() {

    cin *>>* n *>>* m;

    for (int i = 1; i <= n; i++) {

        x*[*i*]* = i; *// 初始化排列*

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

            cin *>>* B[i][j]; *// 输入电路板的二维数组排列*

        }

    }

*backtrack*(1); *// 开始搜索*

    cout *<<* bestd *<<* *endl*; *// 输出最小长度*

    for (int i = 1; i <= n; i++) cout *<<* bestx*[*i*]* *<<* " "; *// 输出最优排列*

    cout *<<* *endl*;

    return 0;

}

算法实现题6-2

思路：

创建一个优先队列来存储待处理的节点。每个节点表示图覆盖的一个状态，包括当前的顶点集合、其权重和以及一个标记数组来表示哪些顶点或边已被覆盖。

通过分支，对每个顶点决定是否加入覆盖集合。这样从根节点到叶节点的每一条路径都代表了一种可能的顶点覆盖。

根据节点的总权重（使用堆结点中的cn字段）来确定优先级，保证总是首先探索权重较小的顶点组合。

在扩展节点之前，检查当前的顶点集合是否已经覆盖了所有的边。如果已经覆盖，则不需要进一步扩展这个节点。如果当前节点的权重已经超过了已知的最优解的权重，也可以停止扩展这个节点。当找到一个覆盖所有边且权重小于当前最优解的顶点集合时，更新最优解。

代码实现：

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

struct *Node* {

*vector*<int> coverStatus; *// 每个顶点的覆盖状态*

    int level; *// 当前处理到的顶点*

    int weightSum; *// 当前覆盖的总权重*

};

*// 自定义比较函数，优先权重小的*

auto cmp = [](*Node* *left*, *Node* *right*) { return *left*.weightSum > *right*.weightSum; };

*priority\_queue*<*Node*, *vector*<*Node*>, decltype(cmp)> *pq*(*cmp*);

*vector*<*vector*<int>> graph; *// 邻接矩阵表示的图*

*vector*<int> vertexWeights; *// 顶点权重*

*vector*<int> bestCover; *// 最佳覆盖*

int minWeightSum = *INT\_MAX*; *// 最小覆盖权重*

bool *isCovered*(const *vector*<int>& *coverStatus*) {

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

        if (*coverStatus[*i*]* == 0) return false;

    }

    return true;

}

void *branchAndBound*(int *n*) {

*Node* initialNode;

    initialNode.coverStatus.*resize*(*n*, 0);

    initialNode.level = 0;

    initialNode.weightSum = 0;

    pq.*push*(initialNode);

    while (!pq.*empty*()) {

*Node* current = pq.*top*();

        pq.*pop*();

        if (current.level == *n*) {

            if (*isCovered*(current.coverStatus) && current.weightSum < minWeightSum) {

                minWeightSum = current.weightSum;

                bestCover *=* current.coverStatus;

            }

        } else {

*// 尝试包含当前顶点*

*Node* includeNode = current;

            includeNode.level++;

            includeNode.weightSum += vertexWeights*[*current.level*]*;

            includeNode.coverStatus*[*current.level*]* = 1; *// 包含*

            if (includeNode.weightSum < minWeightSum) { *// 剪枝*

                pq.*push*(includeNode);

            }

*// 尝试不包含当前顶点*

*Node* excludeNode = current;

            excludeNode.level++;

            if (excludeNode.weightSum < minWeightSum) { *// 剪枝*

                pq.*push*(excludeNode);

            }

        }

    }

}

int *main*() {

    int n, m; *// 顶点数和边数*

    cin *>>* n *>>* m;

    graph.*resize*(n, *vector*<int>(n, 0));

    vertexWeights.*resize*(n);

    for (int i = 0; i < n; ++i) {

        cin *>>* vertexWeights*[*i*]*; *// 输入顶点权重*

    }

    for (int i = 0; i < m; ++i) {

        int u, v;

        cin *>>* u *>>* v; *// 输入边*

        graph*[*u*][*v*]* = graph*[*v*][*u*]* = 1; *// 无向图*

    }

*branchAndBound*(n);

    cout *<<* "Minimum Weight of Vertex Cover: " *<<* minWeightSum *<<* *endl*;

    cout *<<* "Vertices in the Cover (1-based index): ";

    for (int i = 0; i < n; ++i) {

        if (bestCover*[*i*]*) cout *<<* i+1 *<<* " ";

    }

    cout *<<* *endl*;

    return 0;

}

算法实现题6-5

思路:

从第一个男运动员开始，尝试为他找到一个配对的女运动员。对于当前尝试配对的男运动员，遍历所有女运动员。如果当前女运动员未被配对，则计算将这对男女运动员配对后的总竞赛优势，并继续尝试为下一个男运动员配对。

在每一步分配过程中，计算当前的总竞赛优势加上剩余未配对男女运动员可能达到的最大竞赛优势。如果这个可能的最大总优势小于当前已找到的最大总优势，则不需要继续探索这个分支。

如果为当前男运动员找不到合适的配对，则回溯到上一个男运动员，改变他的配对，尝试其他可能的配对方式。当所有男运动员都找到了配对的女运动员，并且无法通过界限条件剪枝，这时的配对方式就是一个可能的最佳配对。比较所有找到的最佳配对，选出总竞赛优势最大的一个。

代码实现：

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

*// 计算当前匹配的总竞赛优势*

int *calculateAdvantage*(const *vector*<*vector*<int>>& *P*, const *vector*<*vector*<int>>& *Q*, const *vector*<int>& *match*) {

    int totalAdvantage = 0;

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

        if (*match[*i*]* != -1) {

            totalAdvantage += *P[*i*][match[*i*]]* \* *Q[match[*i*]][*i*]*;

        }

    }

    return totalAdvantage;

}

void *findBestMatch*(int *k*, *vector*<int>& *match*, *vector*<bool>& *used*, const *vector*<*vector*<int>>& *P*, const *vector*<*vector*<int>>& *Q*, int& *maxAdvantage*, *vector*<int>& *bestMatch*) {

    if (*k* == *match*.*size*()) {

        int currentAdvantage = *calculateAdvantage*(*P*, *Q*, *match*);

        if (currentAdvantage > *maxAdvantage*) {

*maxAdvantage* = currentAdvantage;

*bestMatch* *=* *match*;

        }

        return;

    }

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

        if (!*used[*i*]*) {

*match[k]* = i;

*used[*i*]* *=* true;

*findBestMatch*(*k* + 1, *match*, *used*, *P*, *Q*, *maxAdvantage*, *bestMatch*);

*used[*i*]* *=* false;

        }

    }

}

int *main*() {

    int n;

    cin *>>* n;

*vector*<*vector*<int>> *P*(n, *vector*<int>(n)), *Q*(n, *vector*<int>(n));

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            cin *>>* P*[*i*][*j*]*;

        }

    }

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            cin *>>* Q*[*i*][*j*]*;

        }

    }

*vector*<int> *match*(n, -1), bestMatch;

*vector*<bool> *used*(n, false);

    int maxAdvantage = 0;

*findBestMatch*(0, match, used, P, Q, maxAdvantage, bestMatch);

    cout *<<* "Maximum Advantage: " *<<* maxAdvantage *<<* *endl*;

    cout *<<* "Best Match (Male -> Female): ";

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

        cout *<<* i *<<* " -> " *<<* bestMatch*[*i*]* *<<* ", ";

    }

    cout *<<* *endl*;

    return 0;

}

算法实现题6-10

思路:

可以用一个二维数组来表示陈列馆，其中的每个元素代表一个陈列室，值为0表示未被覆盖，值为1表示已覆盖。创建一个优先队列，其中的元素是当前的状态，例如已经放置的警卫机器人数和未被覆盖的陈列室数。队列按照放置的警卫机器人数进行优化，即数目较少的状态具有更高的优先级。在每一步，考虑放置一个新的警卫机器人。遍历每个未被覆盖的陈列室，并尝试将一个警卫机器人放置在那里。每放置一个警卫机器人，更新状态并将其加入优先队列。

如果当前状态下，即使在剩余的所有陈列室都放置警卫机器人，其数量也超过了当前已找到的最优解，则不再向队列中添加该状态。取出优先队列中的下一个状态，重复分支和限界的过程。

当优先队列为空或者找到一个状态，其中所有陈列室都被覆盖，且使用的警卫机器人数目为当前已知最少时，算法终止。

代码实现：

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

struct *State* {

*vector*<*vector*<int>> gallery;

    int robots;

    int uncovered;

*// 优先队列需要比较函数来确定优先级*

    bool *operator<*(const *State*& *other*) const {

        return robots > *other*.robots;

    }

};

*// 检查新状态是否有效（是否所有陈列室都被覆盖）*

bool *isValidState*(const *State*& *state*, int *m*, int *n*) {

    for (int i = 0; i < *m*; i++) {

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

            if (*state*.gallery*[*i*][*j*]* == 0) return false;

        }

    }

    return true;

}

*// 更新陈列室状态*

void *placeRobot*(*State*& *state*, int *row*, int *col*, int *m*, int *n*) {

*state*.robots++;

*state*.gallery*[row][col]* = 1;

    if (*row* - 1 >= 0) *state*.gallery*[row* - 1*][col]* = 1;

    if (*row* + 1 < *m*) *state*.gallery*[row* + 1*][col]* = 1;

    if (*col* - 1 >= 0) *state*.gallery*[row][col* - 1*]* = 1;

    if (*col* + 1 < *n*) *state*.gallery*[row][col* + 1*]* = 1;

}

int *findMinimumRobots*(int *m*, int *n*) {

*priority\_queue*<*State*> pq;

*State* initial = {*vector*<*vector*<int>>(*m*, *vector*<int>(*n*, 0)), 0, *m* \* *n*};

    pq.*push*(initial);

    int minimumRobots = *m* \* *n*; *// 最坏情况下，每个陈列室都需要一个警卫机器人*

    while (!pq.*empty*()) {

*State* current = pq.*top*();

        pq.*pop*();

        if (current.robots >= minimumRobots) continue; *// 限界条件*

*// 尝试在每个未覆盖的陈列室放置警卫机器人*

        for (int i = 0; i < *m*; i++) {

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

                if (current.gallery*[*i*][*j*]* == 0) {

*State* newState = current;

*placeRobot*(newState, i, j, *m*, *n*);

                    if (*isValidState*(newState, *m*, *n*)) {

                        minimumRobots = *min*(minimumRobots, newState.robots);

                    } else {

                        pq.*push*(newState);

                    }

                }

            }

        }

    }

    return minimumRobots;

}

int *main*() {

    int m, n;

    cin *>>* m *>>* n;

    cout *<<* "Minimum robots required: " *<<* *findMinimumRobots*(m, n) *<<* *endl*;

    return 0;

}

==============================作业5==============================

算法分析题5-3

思路：

首先初始化全局变量maxVal为0，来记录遍历过程中找到的最大价值，以及一个bestPicked数组来记录达到最大价值时选择的物品组合。knapsack函数通过递归考虑每个物品选或不选的情况，并更新全局变量。最后，主函数中调用knapsack函数并打印出最大价值和对应的物品组合。

代码实现：

#include <iostream>

#include <vector>

using namespace std;

// 全局变量，用于记录最大价值及其对应的物品组合

int maxVal = 0;

vector<int> bestPicked;

vector<int> picked;

// 参数说明：

// i: 当前考虑的物品索引

// curWeight: 当前总重量

// curValue: 当前总价值

// weights: 各物品的重量

// values: 各物品的价值

// W: 背包最大承重

void knapsack(int i, int curWeight, int curValue, const vector<int>& weights, const vector<int>& values, int W) {

    int n = weights.size();

    if(i == n) { // 边界条件，所有物品都考虑完了

        if(curValue > maxVal) { // 发现更好的解

            maxVal = curValue;

            bestPicked = picked; // 记录最优解的物品组合

        }

        return;

    }

    // 不选当前物品

    knapsack(i + 1, curWeight, curValue, weights, values, W);

    // 选当前物品，前提是加上这个物品不超过背包的最大承重

    if(curWeight + weights[i] <= W) {

        picked[i] = 1; // 标记选择了这个物品

        knapsack(i + 1, curWeight + weights[i], curValue + values[i], weights, values, W);

        picked[i] = 0; // 回溯，撤销选择这个物品的操作

    }

}

int main() {

    vector<int> weights = {2, 3, 4, 5}; // 物品的重量

    vector<int> values = {3, 4, 5, 6}; // 物品的价值

    int W = 8; // 背包的最大承重

    picked.resize(weights.size(), 0);

    knapsack(0, 0, 0, weights, values, W);

    cout << "Max Value: " << maxVal << endl;

    cout << "Items picked: ";

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

        if(bestPicked[i]) {

            cout << "(" << weights[i] << ", " << values[i] << ") ";

        }

    }

    cout << endl;

    return 0;

}

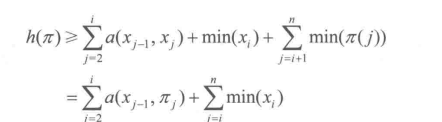
算法分析题5-6

思路：

1. 前缀为x[1:i]的旅行售货员回路任一旅行售货员回路可以表示成一个排列，可以计算出回路的费用为



可以比较得到



从而可以计算出图G所有前缀为x[1:i]的旅行售货员回路的费用至少为



1. 由上述结论，只需要对图G进行简单的遍历，计算出min(i)求和的值，

上界函数中限界条件用此值来计算即可

代码实现：

#include <iostream>

#include <vector>

#include <limits>

using namespace std;

const int INF = *numeric\_limits*<int>::*max*();

int *tsp*(const *vector*<*vector*<int>>& *graph*, *vector*<bool>& *visited*, int *currentPos*, int *n*, int *count*, int *cost*, int& *ans*) {

*// 所有的节点都已经访问过*

    if (*count* == *n* && *graph[currentPos][*0*]*) {

*ans* = *min*(*ans*, *cost* + *graph[currentPos][*0*]*);

        return *ans*;

    }

    for (int i = 0; i < *n*; i++) {

        if (!*visited[*i*]* && *graph[currentPos][*i*]*) {

*// 标记为已访问*

*visited[*i*]* *=* true;

*tsp*(*graph*, *visited*, i, *n*, *count* + 1, *cost* + *graph[currentPos][*i*]*, *ans*);

*// 标记为未访问，回溯*

*visited[*i*]* *=* false;

        }

    }

    return *ans*;

}

int *main*() {

*// 图的表示，graph[i][j]表示从i到j的成本*

*vector*<std::*vector*<int>> graph = {

        {0, 10, 15, 20},

        {10, 0, 35, 25},

        {15, 35, 0, 30},

        {20, 25, 30, 0}

    };

*// 标记访问过的节点*

*vector*<bool> *visited*(graph.*size*());

    visited*[*0*]* *=* true; *// 假设从节点0开始*

    int ans = INF;

*// 计算最小旅行成本*

*tsp*(graph, visited, 0, graph.*size*(), 1, 0, ans);

    cout *<<* "Minimum cost of traveling salesman tour: " *<<* ans *<<* *endl*;

    return 0;

}

==============================作业4==============================

算法实现题4-1

思路：希望使用尽可能少的会场，考虑用贪心算法，先按区间左端点排序，然后从左到右遍历一遍所有时间区间，如果当前区间与上一个区间有交集，说明是不相容的，否则是相容的，可以设置一个start和end来标记当前重合的区间段的起始和末尾，同时用一个cnt判断当前这段重合区间已经涉及几个区间的交集，如果新的区间的左端点比end大，则更新start和end，否则cnt++,同时更新start和end。

贪心选择性质证明：按左端点排序，假设前k个舞会活动排列是最优解，那么此时加上下一个区间后的前k+1个舞会活动排列也是最优解，因为若前k+1个舞会活动需要的会场不是最优解，那么说明前k个舞会活动排列需要的会场不是最优解，这与我们的假设矛盾，所以可以保证这样做每一步都是最优的，从而实现全局最优。

int *minMeetingRooms*(*vector*<*Interval*>& *intervals*) {

    if (*intervals*.*empty*()) return 0;

*// 按会议开始时间排序*

*sort*(*intervals*.*begin*(), *intervals*.*end*(), compareInterval);

*// 初始化会场数量为1，因为至少需要一个会场*

    int count = 1;

*// 初始化当前重合区间段的起始和末尾*

    int start = *intervals[*0*]*.start;

    int end = *intervals[*0*]*.end;

*// 遍历所有区间*

    for (*size\_t* i = 1; i < *intervals*.*size*(); ++i) {

*// 如果当前区间与上一个区间有交集，增加会场数量*

        if (*intervals[*i*]*.start < end) {

            count++;

        }

*// 更新当前重合区间段的起始和末尾*

        start = *intervals[*i*]*.start;

        end = *max*(end, *intervals[*i*]*.end);

    }

    return count;

}

时间复杂度O(nlogn)为排序的时间复杂度，遍历区间仅需O(n)

算法实现题4-2

思路:

多个序列合并时，每两个合并会产生m+n-1次比较，所以在合并的过程中总有序列会被多次合并比较，这些序列应该优先选长度较小的序列，这样可以让所需的总比较次数最少，反之最差的合并顺序就应该优先选长度较大的序列。总体思路类似哈夫曼树的构建，最终需要的总比较次数即哈夫曼树每个非叶结点权值和。

贪心选择性质证明：

设有k个序列，需要进行k-1次合并才能将它们合并成一个序列。我们从这些文件中选取两个最小的序列（设它们的大小为x和y，且x <= y），合并成一个新的序列，其大小为x + y。

假设存在一个最优解，其中x和y不是在第一步就合并的，我们可以将这个解中x和y的合并提前到第一步，而不影响其他合并的成本。因为x和y是最小的两个序列，提前合并它们不会增加后续合并的成本。

这个操作后，我们得到了一个新的问题实例，但是序列的总数减少了一个。依此类推，我们可以应用这个贪心策略，每次都合并当前最小的两个序列。这个过程会重复n-1次，直到所有的序列都合并在一起。

因此，我们可以得出结论，每一步中，选择两个最小的序列进行合并的贪心策略，可以产生最优合并问题的全局最优解。

int *optimalMerge*(*vector*<int>& *sequences*) {

*// 使用C++的优先队列实现最小堆，用于存储序列的长度*

    priority\_queue<int, vector<int>, greater<int>> *minHeap*(sequences.*begin*(), sequences.*end*());

    int totalCost = 0; *// 总成本初始化为0*

    while (minHeap.*size*() > 1) { *// 当堆中有超过一个序列时进行合并*

        int seq1 = minHeap.*top*();

        minHeap.*pop*();

        int seq2 = minHeap.*top*();

        minHeap.*pop*();*// 取出两个最小的序列*

        int mergedSeq = seq1 + seq2; *// 合并序列，并计算成本*

        totalCost += mergedSeq;

        minHeap.*push*(mergedSeq); *// 将合并后的序列长度放回最小堆*

    }

    return totalCost;

}

算法实现题4-6

思路:

要让所有顾客的平均等待时间最少，则顾客应该尽快获得服务，按贪心思想，应该让需要服务时间短的客户优先获得服务，这样才能尽快轮到其他客户。所以需要对客户服务时间service从小到大排序，然后设置一个剩余客户数client和已等待时间sum，服务完每一个客户,client--,sum+=client\*service[i]。

贪心选择性质证明：

假设贪心选择得到的客户服务处理序列是最优的，那么用反证法，假设存在最优解，且与贪心策略产生的解不同：即存在至少一对顾客，其中按服务时间排序的贪心解中顾客i在顾客j之前，但在某个最优解中顾客j在顾客i之前，同时假设服务时间为t[i] <= t[j]。

在最优解中交换顾客i和顾客j的顺序。由于t[i] <= t[j]，交换后顾客j的等待时间不会增加，而顾客i及其后面的顾客的等待时间会减少。

交换顾客i和顾客j后，总等待时间会减少或保持不变（因为t[i] <= t[j]），这意味着新顺序至少和之前的解一样好。但这与我们的假设矛盾，即存在一个不同的最优解。

由于交换可以减少总等待时间，我们可以不断进行这样的交换，直到达到贪心解的顺序，而且交换不会使解变得更差。因此，按照服务时间的非降序排列是最优的。

int *calculateMinimumAverageWaitTime*(*vector*<int>& *serviceTimes*) {

*sort*(*serviceTimes*.*begin*(), *serviceTimes*.*end*(), compareServiceTimes);

    int sum = 0; *// 已等待时间的累计*

    int totalWaitTime = 0; *// 所有顾客的等待时间总和*

    int clients = *serviceTimes*.*size*(); *// 剩余顾客数*

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

*// 为当前顾客服务后，剩余的顾客将等待当前服务时间*

        totalWaitTime += sum;

        sum += *serviceTimes[*i*]*; *// 更新已等待时间*

    }

    return totalWaitTime;

}

时间复杂度O(nlogn)遍历一遍客户服务时间仅需O(n)

算法实现题4-11

思路:可以把正整数a的各位数视作在平面直角坐标系中表示，横轴为第几位，纵轴为该位的数字，这样可以从中看到最大值，按顺序逐个剔除，因为如果不剔除这个最大值，剔除最大值两侧的较小值，那么产生的数总会比剔除最大值更大，所以按这样的顺序剔除满k个数后，剩余的数按原次序拼成的就是最优解。

贪心选择性质：

在每一步中，都寻找并删除第一个使得其后一个数字变小的数字。这意味在数字的某个位置找到了一个局部峰值，并将其删除以使结果最小化。

假设存在一个更优的解决方案，即在某一步骤中，应该删除一个不同的数字。但是，因为总是选择第一个局部峰值进行删除，删除任意其他数字将会导致一个更大的结果，因为我们会留下一个较大的局部峰值。

通过反证法，假设贪心选择没有产生全局最优解。这意味着至少有一个数字可以被删除以获得一个更小的数。但是，根据贪心策略，我们总是删除当前能够产生最小数的数字，因此不存在一个可以产生更小数的删除选项。

*string* *removeKdigits*(*string* *num*, int *k*) {

*// 如果k等于数字的长度，则删除所有数字*

    if(*num*.*length*() == *k*) return "0";

    while(*k* > 0) {

        int n = *num*.*length*();

        int i = 0;

*// 找到第一个局部最大值*

        while(i+1<n && *num[*i*]*<=*num[*i+1*]*) {

            i++;

        }

*// 删除这个局部最大值*

*num*.*erase*(i, 1);

*k*--;

    }

*// 删除前导0*

    int start = 0;

    while(start < *num*.*length*() && *num[*start*]* == '0') {

        start++;

    }

*num*.*erase*(0, start);

    return *num*.*empty*() ? "0" : *num*;

}

时间复杂度为O(n)

==============================作业3==============================

算法分析题3-1

数组b[i]代表以a[i]为结尾的最长单调子序列的长度,易知该问题具有最优子结构性质，可以通过动态规划得到最优解

for(int i = 0;i<n;i++)b[i] = 1;

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

    for(int j = 0;j<i;j++){

        if(a[j]<a[i])b[i] = *max*(b[i],b[j]+1);

    }

}

遍历b数组得到最优解

int res = 0;

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

    res = *max*(res,b[i]);

}

经过两重循环，易知该算法时间复杂度为O(n^2)

算法分析题3-2

数组b[i]代表长度为i的单调递增子序列的最小结尾元素大小，易知该问题具有最优子结构性质，因为算法要求时间复杂度为nlogn，所以利用b[i]具有的单调性，如果a[i]大于当前最长单调递增子序列的最小结尾元素，那么可以更新出更长的单调递增子序列，并以a[i]作为最小结尾元素，否则可以在0-i-1中找a[i]小于的第一个数来更新b[x]（x为该数下标）

关于b[i]单调性的证明，可以假设存在b[i-1]>b[i]，且结尾元素分别为a[i-1]、a[i]，那么a[i-1]>a[i],但是由数组的定义可以知道，b[i-1]应该是长度为i-1的最长单调子序列的最小结尾元素，那么必须是min(a[i-1],a[i])，则有b[i-1] = a[i],与b[i-1]>b[i] =a[i]矛盾，所以b[i]总是具有单调性

for(int i = 0;i<n;i++)b[i] = 0x3f3f3f3f;

b[1] = a[0];

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

    if(a[i]>=b[k]){

        b[++k] = a[i];

    }

    else{

        b[*binaryserach*(i,k)] = a[i];

    }

}

int *binarysearch*(int *i*,int *k*){

    if(a[i]<b[1])return 1;

    for(int l = 1,r = k;l!=r-1;){

        int mid = (l+r)/2;

        if(b[mid]>a[i])r = mid;

        else l = mid;

    }

    return r;

}

算法分析题3-4

二维01背包问题本质也是一个特殊的整数规划问题，可以通过三重数组来存储背包状态，通过背包容量、容积进行状态转移计算

设背包容量为j，容积为k，在前i个物品中选得到的最优值是m[i][j][k]

容易证明该问题具有最优子结构性质，同时该问题的状态转移方程为

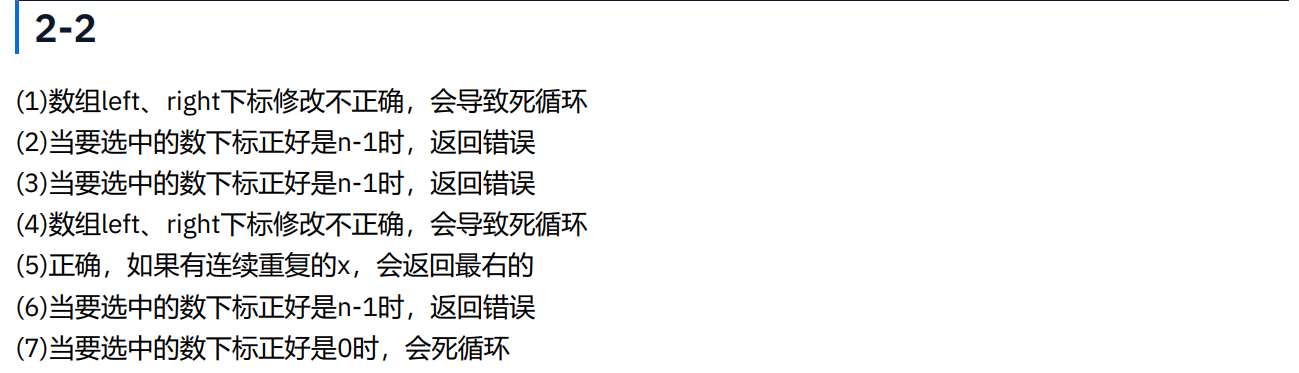
M[i][j][k] = max(m[i-1][j][k],m[i-1][j-w[i]][k-b[i]]+v[i])

通过三重循环填表，可以得到最终最优解即 m[n][m][s]

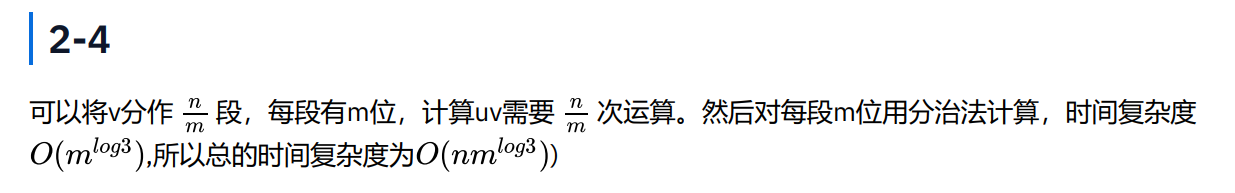
其中n为物品种类，m为容量大小，s为容积大小

==============================作业2==============================

算法分析题2-2



算法分析题2-4



算法实现题2-9

2-9

要实现在O(1)的空间复杂度和最坏情况下O(n)的时间复杂度内完成数组合并排序，可以通过循环右移合并：

用一个shiftRight函数用于将数组中s到e之间的元素向右移动k个位置。这个函数通过循环移动来实现，每次都将区间的最后一个元素保存在临时变量temp中，然后将其他元素向右移动一个位置，并最终将temp放到区间的开始位置。

再通过merge函数实现合并两个有序子数组。创建变量i和j分别是第一个和第二个子数组的起始索引。算法在每一步中：

通过二分搜索找到第一个子数组中元素a[i]在第二个子数组中应该插入的位置pos。

通过shiftRight函数将i到pos之间的元素向右移动，使得a[i]可以放到正确的位置。

更新指针i和j到新的位置，继续下一轮的比较和移动。

算法复杂度分析

空间复杂度：O(1)，因为除了输入数组之外，没有使用额外的数组或者数据结构。

时间复杂度：O(n)

算法实现题2-15

对于偶数个选手，通过递归地将问题分解成处理一半选手的子问题，然后用一个copy函数将已经生成的小规模轮赛表复制和扩展成完整的轮赛表。对于奇数个选手，先添加一个虚拟选手使选手数变为偶数，然后按照偶数选手的处理方式生成轮赛表，最后通过copyodd函数特殊处理轮空的情况，确保所有选手都能参与比赛。

时间复杂度分析：

由于每次递归调用tournament函数时，选手数量减半（假设没有奇数情况的额外处理），递归的深度是O(log n)，其中n是开始时的选手数量。因此，总的时间复杂度是每层递归的时间复杂度之和，即O(m^2) + O((m/2)^2) + O((m/4)^2) + ... + O(1)，这是一个几何级数，其和可以近似为O(n^2)。

空间复杂度分析：

空间复杂度主要取决于存储轮赛表所需的空间。轮赛表是一个二维数组，其大小为n\*n，因此空间复杂度为O(n^2)。

==============================作业1==============================

