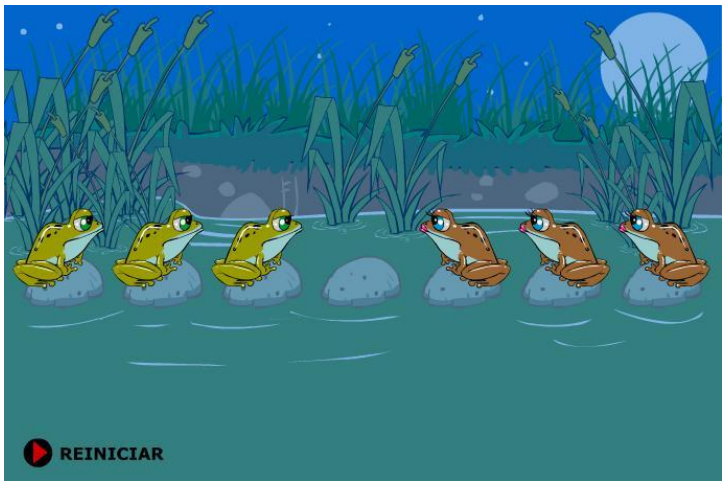
frog_leap 问题



算法思路：

状态表示：用一个字符数组来表示青蛙的位置状态，例如用 'L' 表示左边的青蛙，'R' 表示右边的青蛙，'_' 表示空位。初始状态是若干个 'L' 、一个或多个 '_' 、若干个 'R' ，目标状态是若干个 'R' 、一个或多个 '_' 、若干个 'L' 。

解空间：解空间是所有可能的青蛙移动序列。可以构建一棵解空间树，树的每一层表示一次移动操作。每个节点表示当前的青蛙位置状态，节点的分支表示不同的移动选择（左青蛙右移、左青蛙跳过右青蛙、右青蛙左移、右青蛙跳过左青蛙）。

约束条件：左青蛙只能向右移动，右青蛙只能向左移动。

移动时，目标位置必须为空位或者目标位置上是对方阵营的青蛙且下一个位置为空位（即可以跳过对方青蛙）。

记录已经尝试过的状态，避免重复搜索。如果当前状态已经在之前的搜索中出现过，则该分支可以剪枝。

搜索策略：采用深度优先搜索（DFS）策略遍历解空间树。从初始状态开始，依次尝试各种合法的移动操作，直到达到目标状态。记录达到目标状态的移动序列，比较不同序列的长度，找到最短的移动序列。

算法分析题 5

5-6

设 G 是一个有 n 个顶点的有向图，从顶点 i 发出的边的最小费用记为 $\min(i)$。

（1）证明图 G 的所有前缀为 $x[1:i]$的旅行售货员回路的费用至少为：

$$\sum_{j=2}^{i} a(x_{j-1},x_j)+\sum_{j=i}^{n}\min(x_j)$$

式中，$a(u,v)$是边(u,v)的费用。

（2）利用上述结论设计一个高效的上界函数，重写旅行售货员问题的回溯法，并与主教材中的算法进行比较。

（1）证明

设 T 是图 G 的一条旅行售货员回路，其顶点序列为 x1,x2,...,xn,x1 ，其中 x[1:i]是该回路的前缀。

对于旅行售货员回路 T，其费用 $C(T)=\sum_{j=2}^{n} a(x_{j-1},x_j)+a(x_n,x_1)$。

已知从顶点 k 发出的边的最小费用记为 min(k)。
考虑前缀 x[1:i]，这部分回路的费用为$\sum_{j=2}^{i}a(x_{j-1},x_j)$。
对于回路中剩余的顶点 $x_i,x_{i+1},...,x_n,x_1$，从每个顶点 x_j（j>= i）出发的边的费用最小为 $min(x_j)$。
那么，从顶点 xi 开始后续部分回路的费用至少是从 xi 出发的最小费用边，再加上从后续每个顶点出发的最小费用边，即至少为$\sum_{j=i}^{n} min(x_j)$。

所以，图 G 的所有前缀为 x[1:i]的旅行售货员回路的费用至少为$\sum_{j=2}^{i}a(x_{j-1},x_j)+\sum_{j=i}^{n} min(xj)$。
（2）
上界函数设计 根据（1）中的结论，可以设计上界函数 $ub(x[1:i])=\sum_{j=2}^{i}a(x_{j-1},x_j)+\sum_{j=i}^{n}min(xj)$。
这个上界函数表示以 x[1:i]为前缀的旅行售货员回路费用的下界估计。在回溯过程中，如果当前扩展节点的费用加上从该节点出发后续部分的最小费用估计（由上界函数计算）已经大于当前已知的最小回路费用（初始化为一个较大值，在搜索过程中更新），则可以对该节点进行剪枝，不再继续扩展其分支。
改写后的回溯算法如下：

```
const int INF = numeric_limits<int>::max();int n;
vector<vector<int>> graph;
vector<bool> visited;
vector<int> path;
vector<int> bestPath;int minCost = INF;
// 计算从顶点 v 出发的最小边费用 int minEdgeCost(int v) {
    int minCost = INF;
    for (int i = 0; i < n; ++i) {
        if (graph[v][i] > 0 && graph[v][i] < minCost) {
            minCost = graph[v][i];
        }
    }
    return minCost;}
// 计算上界函数值 int upperBound(const vector<int>& partialPath) {
    int cost = 0;
    for (size_t i = 1; i < partialPath.size(); ++i) {
        cost += graph[partialPath[i - 1]][partialPath[i]];
    }
    for (size_t i = partialPath.size(); i < n; ++i) {
        cost += minEdgeCost(partialPath.back());
    }
    return cost;}
// 回溯函数 void backtrack(int depth) {
    if (depth == n) {
        if (graph[path.back()][path[0]] > 0) {
            int currentCost = 0;
            for (size_t i = 1; i < path.size(); ++i) {
                currentCost += graph[path[i - 1]][path[i]];
            }
```

```
                currentCost += graph[path.back()][path[0]];
                if (currentCost < minCost) {
                    minCost = currentCost;
                    bestPath = path;
                }
            }
            return;
        }
        for (int i = 1; i < n; ++i) {
            if (!visited[i]) {
                path[depth] = i;
                visited[i] = true;
                int ub = upperBound(path);
                if (ub < minCost) {
                    backtrack(depth + 1);
                }
                visited[i] = false;
            }
        }}
// 解决旅行售货员问题 void travelingSalesman() {
        visited.assign(n, false);
        path.assign(n, 0);
        path[0] = 0;
        visited[0] = true;
        backtrack(1);
        if (minCost != INF) {
            cout << "最小费用: " << minCost << endl;
            cout << "路径: ";
            for (int node : bestPath) {
                cout << node << " ";
            }
            cout << endl;
        } else {
            cout << "没有找到合法路径" << endl;
        }}
```

算法实现题 5

5-1

问题描述：子集和问题的一个实例为<S, c>。其中，$S=\{x_1, x_2, \cdots, x_n\}$是一个正整数的集合，$c$ 是一个正整数。子集和问题判定是否存在 S 的一个子集 $S1$，使得 $\sum_{x \in S1} x = c$。试设计一个解子集和问题的回溯法。

算法设计：对于给定的正整数的集合 $S=\{x_1, x_2, \cdots, x_n\}$和正整数 c，计算 S 的一个子集 S_1，使得 $\sum_{x \in S_1} x = c$。

数据输入：由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 c，n 表示 S 的大小，c 是子集和的目标值。接下来的 1 行中，有 n 个正整数，表示集合 S 中的元素。

结果输出：将子集和问题的解输出到文件 output.txt。当问题无解时，输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
5 10	2 2 6
2 2 6 5 4	

思路：

解空间：对于给定的包含 n 个正整数的集合 S={x_1,x_2,...,x_n}，子集和问题的解空间是所有可能的子集组合。可以用一棵高度为 n + 1 的子集树来表示解空间。树的第 i 层（1<= i<= n）表示是否选择集合中的第 i 个元素，每个节点有两个分支，左分支表示选择该元素，右分支表示不选择该元素 。

约束条件：在构建子集的过程中，需要记录当前子集中元素的和。设当前已选元素的和为 sum，若 sum > c，则该分支不符合要求，进行剪枝；若 sum = c，则找到了一个满足条件的子集，记录结果。

搜索策略：从根节点开始，按照深度优先搜索（DFS）的策略遍历子集树。先递归进入左子树（选择当前元素），然后递归进入右子树（不选择当前元素）。

代码：

```
vector<int> result;
vector<int> current;
int target;

// 回溯函数
void backtrack(vector<int>& nums, int start, int sum) {
    if (sum == target) {
        result = current;
        return;
    }
    if (start >= nums.size() || sum > target) {
        return;
    }
    // 选择当前元素
    current.push_back(nums[start]);
    backtrack(nums, start + 1, sum + nums[start]);
    current.pop_back();
    // 不选择当前元素
    backtrack(nums, start + 1, sum);
}
```

```
vector<int> subsetSum(vector<int>& nums, int c) {
    target = c;
    backtrack(nums, 0, 0);
    if (result.empty()) {
        return { -1 }; // 表示无解
    }
    return result;
}
```

5-3

问题描述：设某一机器由 n 个部件组成，每种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量，c_{ij} 是相应的价格。试设计一个算法，给出总价格不超过 c 的最小重量机器设计。

算法设计：对于给定的机器部件重量和机器部件价格，计算总价格不超过 d 的最小重量机器设计。

数据输入：由文件 input.txt 给出输入数据。第一行有 3 个正整数 n、m 和 d。接下来的 $2n$ 行，每行 n 个数。前 n 行是 c，后 n 行是 w。

结果输出：将计算的最小重量及每个部件的供应商输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 3 4	4
1 2 3	1 3 1

思路：

解空间：这是一个组合优化问题，解空间是所有可能的部件供应商选择组合。可以用一棵高度为 n + 1 的树来表示解空间，树的第 i 层（1<= i<= n）表示为第 i 个部件选择供应商，每个节点有 m 个分支，分别对应从 m 个不同供应商中选择一个。

约束条件：在构建组合的过程中，需要记录当前选择的部件的总价格和总重量。设当前总价格为 totalCost，总重量为 totalWeight 。若 totalCost > d（总价格超过预算），则该分支不符合要求，进行剪枝；若 totalCost<= d 且当前总重量小于已记录的最小重量（初始化为一个较大值），则更新最小重量和对应的部件供应商选择方案。

搜索策略：从根节点开始，按照深度优先搜索（DFS）的策略遍历这棵解空间树。即先对一个部件的所有供应商选择分支进行深度探索，再进入下一个部件的供应商选择探索。

代码：

```
// 全局变量
int n, m, d;
vector<vector<int>> cost;
vector<vector<int>> weight;
vector<int> currentSolution;
vector<int> bestSolution;
int minWeight = numeric_limits<int>::max();
```

```
// 回溯函数
void backtrack(int componentIndex, int totalCost, int totalWeight) {
    if (componentIndex == n) {
        if (totalCost <= d && totalWeight < minWeight) {
            minWeight = totalWeight;
            bestSolution = currentSolution;
        }
        return;
    }
    for (int supplierIndex = 0; supplierIndex < m; ++supplierIndex) {
        currentSolution[componentIndex] = supplierIndex + 1;
        int newCost = totalCost + cost[componentIndex][supplierIndex];
        int newWeight = totalWeight +
weight[componentIndex][supplierIndex];
        if (newCost <= d) {
            backtrack(componentIndex + 1, newCost, newWeight);
        }
    }
}

void findMinWeightDesign() {
    currentSolution.resize(n);
    bestSolution.resize(n);
    backtrack(0, 0, 0);
    if (minWeight != numeric_limits<int>::max()) {
        cout << minWeight << endl;
        for (int supplier : bestSolution) {
            cout << supplier << " ";
        }
    } else {
        cout << "No Solution!" << endl;
    }
}
```

5-6

问题描述：设 S 是正整数集合。S 是一个无和集，当且仅当 $x, y \in S$ 蕴含 $x+y \notin S$。对于任意正整数 k，如果可将 $\{1, 2, \cdots, k\}$ 划分为 n 个无和子集 $S_1, S_2, \cdots, S_n$，则称正整数 k 是 n 可分的。记 $F(n)=\max\{k \mid k$ 是 n 可分的$\}$。试设计一个算法，对任意给定的 n，计算 $F(n)$的值。

算法设计：对任意给定的 n，计算 $F(n)$的值。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n。

结果输出：将计算的 $F(n)$的值以及 $\{1, 2, \cdots, F(n)\}$的一个 n 划分输出到文件 output.txt。文件的第 1 行是 $F(n)$的值。接下来的 n 行，每行是一个无和子集 S_i。

输入文件示例	输出文件示例
input.txt	output.txt
2	8
	1 2 4 8
	3 5 6 7

思路：

解空间：解空间是将正整数集合{1, 2, ..., k}划分为 n 个无和子集的所有可能划分方式。可以构建一棵解空间树，树的每一层代表向一个子集中添加元素的过程。树的深度从 1 到 k，表示处理数字 1 到 k。在每一层，有 n 个分支，分别表示将当前数字放入 n 个子集中的某一个。

约束条件：在向子集中添加元素时，需要判断加入元素后该子集是否仍为无和集。即对于子集中任意两个元素 x、y，其和 x + y 不在该子集中。如果不满足这个条件，则该分支不符合要求，进行剪枝。

搜索策略：采用深度优先搜索（DFS）策略遍历解空间树。从根节点开始，先尝试将数字放入第一个子集，然后递归处理下一个数字；如果不满足条件，再回溯尝试将数字放入下一个子集，以此类推。不断增大 k，直到找到最大的满足 n 划分的 k 值，即 F(n) 。

代码：

```
vector<vector<int>> subsets;
int n;

// 检查集合是否为无和集
bool isSumFree(const vector<int>& subset) {
    unordered_set<int> sumSet;
    for (size_t i = 0; i < subset.size(); ++i) {
        for (size_t j = i + 1; j < subset.size(); ++j) {
            int sum = subset[i] + subset[j];
            if (sumSet.find(sum) != sumSet.end()) {
                return false;
            }
            sumSet.insert(sum);
        }
    }
    return true;
}

// 回溯函数
bool backtrack(int num) {
```

```
    if (num == 0) {
        return true;
    }
    for (int i = 0; i < n; ++i) {
        subsets[i].push_back(num);
        if (isSumFree(subsets[i]) && backtrack(num - 1)) {
            return true;
        }
        subsets[i].pop_back();
    }
    return false;
}

// 计算 F(n)并得到划分
void findFAndPartition() {
    int k = 1;
    while (true) {
        subsets.assign(n, vector<int>());
        if (backtrack(k)) {
            k++;
        } else {
            break;
        }
    }
    cout << k - 1 << endl;
    for (const auto& subset : subsets) {
        for (int num : subset) {
            cout << num << " ";
        }
        cout << endl;
    }
}
```

5-13

问题描述：设有 n 件工作分配给 n 个人。将工作 i 分配给第 j 个人所需的费用为 c_{ij}。试设计一个算法，为每个人都分配 1 件不同的工作，并使总费用达到最小。

算法设计：设计一个算法，对于给定的工作费用，计算最佳工作分配方案，使总费用达到最小。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（$1 \leqslant n \leqslant 20$）。接下来的 n 行，每行 n 个数，表示工作费用。

结果输出：将计算的最小总费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	9
10 2 3	
2 3 4	
3 4 5	

思路：

解空间：这是一个排列组合问题，解空间是 n 个人对 n 件工作的所有排列方式，即 n 个元素的全排列。可以用一棵高度为 n 的排列树来表示解空间。树的第 i 层（1<= i<= n）表示为第 i 个人分配工作，每个节点对应一种工作分配的选择。

约束条件：在回溯过程中，需要记录已经分配过的工作（避免重复分配），并且记录当前分配方案的总费用。设当前已分配 k 个人的工作，总费用为 currentCost，若 currentCost 已经大于当前已知的最小总费用（初始化为一个较大值），则该分支不符合要求，进行剪枝。

搜索策略：采用深度优先搜索（DFS）策略遍历排列树。从根节点开始，依次尝试为每个人分配不同的工作，当为第 n 个人分配完工作后，检查总费用是否为最小，若为最小则更新最小总费用。

代码：

```
int n;
vector<vector<int>> cost;
vector<bool> used;
int minCost = INF;

// 回溯函数
void backtrack(int person, int currentCost) {
    if (person == n) {
        minCost = min(minCost, currentCost);
        return;
    }
    for (int job = 0; job < n; ++job) {
        if (!used[job]) {
            used[job] = true;
            backtrack(person + 1, currentCost + cost[person][job]);
            used[job] = false;
        }
    }
}

int minCostAssignment() {
```

```
    used.assign(n, false);
    backtrack(0, 0);
    return minCost;
}
```

5-16

问题描述：给定 n 个正整数和 4 个运算符+、−、*、/，且运算符无优先级，如 2+3×5=25。对于任意给定的整数 m，试设计一个算法，用以上给出的 n 个数和 4 个运算符，产生整数 m，且用的运算次数最少。给出的 n 个数中每个数最多只能用 1 次，但每种运算符可以任意使用。

算法设计：对于给定的 n 个正整数，设计一个算法，用最少的无优先级运算次数产生整数 m。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m。第 2 行是给定的用于运算的 n 个正整数。

结果输出：将计算的产生整数 m 的最少无优先级运算次数以及最优无优先级运算表达式输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 25	2
5 2 3 6 7	2+3*5

思路：

解空间：解空间是由 n 个给定正整数通过+、-、\times、\div 这 4 种运算符进行组合运算的所有可能表达式。可以用一棵解空间树来表示，树的每一层代表一次运算。树的根节点表示初始状态（即给定的 n 个数），每一层的节点表示对上一层某个中间结果进行一次运算后的状态。每个节点有 4 \times C_{n}^{2}个分支（4 种运算符，每次从 n 个数中选 2 个数进行运算），表示不同的运算选择。

约束条件：在回溯过程中，要记录已经使用过的数字（每个数最多用一次），并且记录当前运算得到的中间结果。同时，记录当前的运算次数。若当前运算次数已经大于当前已知的最少运算次数（初始化为一个较大值），则该分支不符合要求，进行剪枝。另外，除法运算要求除数不为 0，如果出现除数为 0 的情况，该分支也不符合要求。

搜索策略：采用深度优先搜索（DFS）策略遍历解空间树。从根节点开始，依次尝试不同的两个数和不同的运算符进行运算，不断得到新的中间结果并继续向下搜索，当得到目标整数 m 时，更新最少运算次数和对应的最优表达式。

代码：

```
int n, target;
vector<int> numbers;
int minOps = numeric_limits<int>::max();
string bestExpression;

// 检查是否能通过运算得到目标值，返回运算后的结果
int operate(int a, int b, char op) {
```

```
    switch (op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return b != 0 ? a / b : 0;
    }
    return 0;
}

// 回溯函数
void backtrack(vector<int> remaining, string expression, int ops) {
    if (remaining.size() == 1) {
        if (remaining[0] == target && ops < minOps) {
            minOps = ops;
            bestExpression = expression;
        }
        return;
    }
    if (ops >= minOps) {
        return;
    }
    for (size_t i = 0; i < remaining.size(); ++i) {
        for (size_t j = i + 1; j < remaining.size(); ++j) {
            for (char op : {'+', '-', '*', '/'}) {
                vector<int> newRemaining = remaining;
                int result = operate(newRemaining[i], newRemaining[j], op);
                newRemaining.erase(newRemaining.begin() + j);
                newRemaining.erase(newRemaining.begin() + i);
                newRemaining.push_back(result);
                string newExpression = expression;
                if (!expression.empty()) {
                    newExpression += " ";
                }
                newExpression += to_string(remaining[i]) + op +
to_string(remaining[j]);
                backtrack(newRemaining, newExpression, ops + 1);
            }
        }
    }
}

void findMinOpsExpression() {
    backtrack(numbers, "", 0);
    if (minOps != numeric_limits<int>::max()) {
```

```
        cout << minOps << endl;
        cout << bestExpression << endl;
    } else {
        cout << "No Solution!" << endl;
    }
}
```

5-20

问题描述：原始部落 byteland 中的居民们为了争夺有限的资源，经常发生冲突。几乎每个居民都有他的仇敌。部落酋长为了组织一支保卫部落的队伍，希望从部落的居民中选出最多的居民入伍，并保证队伍中任何 2 个人都不是仇敌。

算法设计：给定 byteland 部落中居民间的仇敌关系，计算组成部落卫队的最佳方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，表示 byteland 部落中有 n 个居民，居民间有 m 个仇敌关系。居民编号为 $1, 2, \cdots, n$。接下来的 m 行中，每行有 2 个正整数 u 和 v，表示居民 u 与居民 v 是仇敌。

结果输出：将计算的部落卫队的最佳组建方案输出到文件 output.txt。文件的第 1 行是部落卫队的人数；第 2 行是卫队组成 x_i（$1 \leqslant i \leqslant n$）。$x_i$=0 表示居民 i 不在卫队中，x_i=1 表示居民在卫队中。

输入文件示例	输出文件示例
input.txt	output.txt
7 10	3

思路：

解空间：解空间是所有可能的居民选择组合，可表示为一棵子集树。树的高度为 n + 1，第 i 层（1<= i<= n）表示是否选择编号为 i 的居民加入队伍。每个节点有两个分支，左分支表示选择该居民，右分支表示不选择该居民。

约束条件：在选择居民加入队伍的过程中，要检查新加入的居民与队伍中已有的居民是否为仇敌关系。设当前已选居民集合为 team，若新加入居民 j 与 team 中任意居民是仇敌，则该分支不符合要求，进行剪枝。同时记录当前队伍人数，若当前队伍人数大于已记录的最大队伍人数（初始化为 0），则更新最大队伍人数和对应的队伍组成方案。

搜索策略：采用深度优先搜索（DFS）策略遍历子集树。从根节点开始，先递归进入左子树（选择当前居民），再递归进入右子树（不选择当前居民）。

代码：

```
int n, m;
unordered_map<int, vector<int>> enemyMap;
vector<int> currentTeam;
vector<int> bestTeam;
int maxSize = 0;

// 检查居民能否加入当前队伍
bool canAdd(int resident) {
```

```
    for (int member : currentTeam) {
        auto it = enemyMap.find(member);
        if (it != enemyMap.end()) {
            for (int enemy : it->second) {
                if (enemy == resident) {
                    return false;
                }
            }
        }
    }
    return true;
}

// 回溯函数
void backtrack(int residentIndex) {
    if (residentIndex == n + 1) {
        if (currentTeam.size() > maxSize) {
            maxSize = currentTeam.size();
            bestTeam = currentTeam;
        }
        return;
    }
    // 尝试加入当前居民
    if (canAdd(residentIndex)) {
        currentTeam.push_back(residentIndex);
        backtrack(residentIndex + 1);
        currentTeam.pop_back();
    }
    // 不加入当前居民
    backtrack(residentIndex + 1);
}

void findBestTeam() {
    backtrack(1);
    cout << maxSize << endl;
    vector<int> result(n, 0);
    for (int member : bestTeam) {
        result[member - 1] = 1;
    }
    for (int num : result) {
        cout << num;
    }
}
```