二

1

算法思路：

1）初始化一个空集合 U

2）从集合 W 中选择一个子集 S 放入 U

3）递归地在剩余的集合 W' = W - {S} 中继续选择子集放入 U，确保新加入的子集与已有的子集都不相交

4）当 U 的并集等于 A 时，将 U 加入结果集

5）回溯到上一步，选择 W 中的下一个子集，并重复步骤 3）和 4）

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;

// 判断两个集合是否相交
bool isIntersect(const set<int>& set1, const set<int>& set2) {
    for (int elem : set1) {
        if (set2.count(elem) > 0) {
            return true;
        }
    }
    return false;
}

// 回溯算法主体
void backtrack(const vector<set<int> >& W, set<int>& current, set<set<int> >& result, const set<int>& A) {
    // 如果当前集合的并集等于 A，则将其加入结果集
    if (current == A) {
        result.insert(current);
        return;
    }
```

```cpp

    // 从集合 W 中选择一个子集放入当前集合
    for (const set<int>& subset : W) {
        if (!isIntersect(current, subset)) {
            current.insert(subset.begin(), subset.end());

            // 递归调用
            backtrack(W, current, result, A);

            // 回溯，移除最后一个加入的子集
            for (int elem : subset) {
                current.erase(elem);
            }
        }
    }
}

// 主函数
set<set<int> > findSubsets(const set<int>& A, const vector<set<int> >& W) {
    set<set<int> > result;
    set<int> current;

    // 调用回溯算法
    backtrack(W, current, result, A);

    return result;
}

int main() {
    set<int> A;
    A.insert(1);
    A.insert(2);
    A.insert(3);
    vector<set<int> > W;
    set<int> subset1, subset2, subset3, subset4;
    subset1.insert(1);
    subset2.insert(2);
    subset2.insert(3);
    subset3.insert(1);
    subset3.insert(2);
```

```
    subset4.insert(3);

    W.push_back(subset1);
    W.push_back(subset2);
    W.push_back(subset3);
    W.push_back(subset4);

    set<set<int> > subsets = findSubsets(A, W);

    // 输出结果
    for (auto subset : subsets) {
        cout << "{";
        for (auto elem : subset) {
            cout << elem << " ";
        }
        cout << "}" << endl;
    }

    return 0;
}
```

2

算法思路，使用回溯算法，依次对`12`个数位进行从`1`到`4`的枚举，引进`row[4][4]`，`col[4][4]`，`row[i][j]`表示第`i`行数字`j+1`是否能使用，若为`0`则不能使用，为`1`则可以使用；`col[4][4]`类似

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
const int n = 12;
int line[12] = {0};
int row[4][4] = {{0,0,0,0},
                 {1,1,1,1},
                 {1,1,1,1},
                 {1,1,1,1}
};

```

```cpp
int col[4][4] = {{0,1,1,1},
                 {1,0,1,1},
                 {1,1,0,1},
                 {1,1,1,0}
};

void dfs(int u){
    if(u == n){
        for(int i = 0; i < n; i++){
            cout << line[i] << " ";
        }
        cout << endl;
        return;
    }

    int x = u / 4 + 1, y = u % 4;
    for(int i = 1; i <= 4; i++){
        if(row[x][i - 1] != 0 && col[y][i - 1] != 0){
            row[x][i - 1] = 0;
            col[y][i - 1] = 0;
            line[u] = i;
            dfs(u + 1);
            row[x][i - 1] = 1;
            col[y][i - 1] = 1;
            line[u] = 0;
        }
    }

}

int main() {
    dfs(0);
    return 0;
}
```

> 2 1 4 3 3 4 1 2 4 3 2 1
> 2 1 4 3 3 4 2 1 4 3 1 2
> 2 1 4 3 4 3 1 2 3 4 2 1
> 2 1 4 3 4 3 2 1 3 4 1 2
> 2 3 4 1 3 4 1 2 4 1 2 3

```
2 3 4 1 4 1 2 3 3 4 1 2
2 4 1 3 3 1 4 2 4 3 2 1
2 4 1 3 4 3 2 1 3 1 4 2
3 1 4 2 2 4 1 3 4 3 2 1
3 1 4 2 4 3 2 1 2 4 1 3
3 4 1 2 2 1 4 3 4 3 2 1
3 4 1 2 2 3 4 1 4 1 2 3
3 4 1 2 4 1 2 3 2 3 4 1
3 4 1 2 4 3 2 1 2 1 4 3
3 4 2 1 2 1 4 3 4 3 1 2
3 4 2 1 4 3 1 2 2 1 4 3
4 1 2 3 2 3 4 1 3 4 1 2
4 1 2 3 3 4 1 2 2 3 4 1
4 3 1 2 2 1 4 3 3 4 2 1
4 3 1 2 3 4 2 1 2 1 4 3
4 3 2 1 2 1 4 3 3 4 1 2
4 3 2 1 2 4 1 3 3 1 4 2
4 3 2 1 3 1 4 2 2 4 1 3
4 3 2 1 3 4 1 2 2 1 4 3
```

3

采用优先队列分支限界法：

```cpp
#include <iostream>
#include <queue>

using namespace std;

const int MAX_TASKS = 5;

// 问题表示
int nTasks = 4;  // 任务数
int taskCost[MAX_TASKS][MAX_TASKS] = {
    {0},
    {0, 9, 2, 7, 8},
    {0, 6, 4, 3, 7},
    {0, 5, 8, 1, 8},
```

```cpp
    {0, 7, 6, 9, 4}
};

int bestAssignment[MAX_TASKS]; // 最优分配方案
int minTotalCost = 0x3f3f3f3f; // 最小成本
int totalNodes = 1;            // 结点个数累计

struct Node {
    int no;                       // 结点编号
    int person;                   // 人员编号
    int assignment[MAX_TASKS];    // assignment[i]为人员i分配的任务编号
    bool allocated[MAX_TASKS];    // allocated[i]=true表示任务i已经分配
    int cost;                     // 已经分配任务所需要的成本
    int lb;                       // 下界

    bool operator<(const Node &other) const // 重载<关系函数
    {
        return lb > other.lb;
    }
};

void calculateLowerBound(Node &node) {
    int minSum = 0;
    for (int i = node.person + 1; i <= nTasks; i++) {
        int minTaskCost = 0x3f3f3f3f;
        for (int j = 1; j <= nTasks; j++)
            if (!node.allocated[j] && taskCost[i][j] < minTaskCost) // 寻找每一列的最小值
                minTaskCost = taskCost[i][j];
        minSum += minTaskCost;
    }
    node.lb = node.cost + minSum;
}

void bfs() {
    int j;
    Node current, next;
    priority_queue<Node> nodeQueue;
    memset(current.assignment, 0, sizeof(current.assignment));
    memset(current.allocated, 0, sizeof(current.allocated));

```

```cpp
    current.person = 0;
    current.cost = 0;
    calculateLowerBound(current);
    current.no = totalNodes++;
    nodeQueue.push(current);

    while (!nodeQueue.empty()) {
        current = nodeQueue.top();
        nodeQueue.pop();
        if (current.person == nTasks) {
            if (current.cost < minTotalCost) {
                minTotalCost = current.cost;
                for (j = 1; j <= nTasks; j++)
                    bestAssignment[j] = current.assignment[j];
            }
        }

        next.person = current.person + 1;
        for (j = 1; j <= nTasks; j++) {
            if (current.allocated[j])
                continue;
            for (int i = 1; i <= nTasks; i++)
                next.assignment[i] = current.assignment[i];
            next.assignment[next.person] = j;
            for (int i = 1; i <= nTasks; i++)
                next.allocated[i] = current.allocated[i];
            next.allocated[j] = true;
            next.cost = current.cost + taskCost[next.person][j];
            calculateLowerBound(next);
            next.no = totalNodes++;
            if (next.lb <= minTotalCost)
                nodeQueue.push(next);
        }
    }
}

int main() {
    bfs();
    cout << "最小成本为： " << minTotalCost << endl;
    cout << "最优分配方案为： " << endl;
    for (int i = 1; i <= nTasks; i++)
```

```
        cout << "人员" << i << "分配任务" << bestAssignment[i] << endl;
    return 0;
}
```

> 最小成本为：13
> 最优分配方案为:
> 人员1分配任务2
> 人员2分配任务1
> 人员3分配任务3
> 人员4分配任务4