

数据结构与算法 🥰

- 数据结构与算法 🥰
 - 第一章、数据结构
 - 第一节、STL 🤪 ^4
 - 一、向量vector 🥰
 - 二、字符串string 🥰
 - 三、栈stack 🥰
 - 四、队列queue 🥰
 - 五、双向队列deque 🥰
 - 六、优先级队列priority_queue 🥰
 - 七、数据对pair 🥰
 - 八、字典map 🥰
 - map
 - 字典unordered_map
 - 九、集合set 🥰
 - 十、bitset 🥰
 - 十一、元组tuple 🥰
 - STL函数
 - 第二节、排序算法 🤪
 - 一、sort排序 🥰
 - 二、冒泡排序 🤪
 - 三、快速排序 🤪
 - 四、归并排序 🤪
 - 经典归并排序
 - 链表归并排序
 - 求逆序对数量 🤪
 - 五、计数排序 🤪
 - 第三节、树tree 🤪
 - 一、二叉树 🤪
 - 头文件1
 - 构建二叉树结构体1
 - 前序序列反序列化
 - 前中序序列反序列化，无"#" 🤪
 - 删除树
 - 前序，中序，后序遍历

- 层序遍历
 - 主函数1
- 二、二叉搜索树 🧑🏻
 - 头文件2
 - 构建二叉树结构体2
 - 查找
 - 插入1
 - 删除1 🧑🏻
 - 主函数2
- 三、AVL树 🧑🏻
 - AVL头文件
 - 构建二叉树结构体3
 - //获取高度
 - 更新高度
 - 左右旋
 - 插入2
 - 删除2
 - 自平衡 🧑🏻
- 四、红黑树
- 第四节、二叉堆BinaryHeap 🧑🏻
 - 头文件及定义
 - 上调
 - 下调
 - 插入
 - 删除顶元素
 - 朴素建堆
 - 快速建堆
- 第五节、静态查找Search 🧑🏻
 - 一、顺序查找 🧑🏻
 - 查找最大最小值
 - 查找素数
 - 埃氏筛选法 🧑🏻
 - 欧拉筛选法 🧑🏻
 - 二、二分查找 🧑🏻
 - 查找顺序表元素
 - 未排序序列快速查找k小元素
 - 寻找最大的最小距离
- 第六节、图graph 🧑🏻

- 一、图基础功能^1
 - 头文件
 - 邻接矩阵的存储结构
 - 邻接链表表示
 - 用邻接矩阵法删除节点
- 二、深度优先遍历
 - DFS算法实现
 - 应用：寻找路径
 - 应用：走迷宫
- 三、广度优先遍历
 - BFS算法实现
 - 例题
- 四、欧拉回路🤖
- 五、无向图的割点与桥🤖
- 六、有向图的强连通分量🤖
 - kosaraju算法😓
 - Tarjan算法😓
- 七、拓扑排序🤖
 - 课本伪代码重写
 - 例题1😓
- 八、最短路径🤖
 - Dijkstra算法(边权重非负)*😓
 - Bellman-Ford算法😓
 - SPFA算法😓
- 九、最小生成树🤖
 - Prim算法(扩点)😓
 - prim算法实现^3
 - Kruskal算法(扩边)😓
 - 并查集
 - Kruskal算法实现^3
 - 排序优化(高度复杂)😓
- 十、图例题^2🤖
 - 例题一
 - 例题二
 - 例题三*😓
 - 例题四*😓

○ 第二章、算法🤖

- 第一节、贪心算法😓

- 一、序列问题
 - 摆动序列问题
- 二、递增数问题
 - 取反最大和
- 三、股票问题
 - 买卖股票问题
- 四、两维度问题
 - 分发糖果问题
 - 身高排列问题
- 五、区间问题
 - 跳跃问题(一)
 - 跳跃问题(二)
 - 箭扎气球问题
 - 无重叠区间问题
 - 字母区间问题
- 六、其他
 - 最大序列和问题
 - 加油站问题
 - 监控二叉树问题
- 第二节、回溯算法 🙄
 - 一、组合问题
 - 组合问题(基础)
 - 组合总和(一)
 - 组合总和(二)
 - 组合求和(三)
 - 多集合求组和
 - 二、切割问题
 - 三、子集问题
 - 子集问题(一)
 - 子集问题(二)
 - 非递减子序列问题
 - 四、排列问题
 - 排列问题(一)
 - 排列问题(二)
 - 五、安排行程问题
 - 六、棋盘问题
 - n皇后问题
 - 解数独问题

- 第三节、动态规划 😞
 - 一、基础题型
 - 斐波那契数列
 - 不同路径问题
 - 整数拆分问题
 - 二叉搜索树问题*
 - 二、背包问题
 - 01背包问题
 - 01背包问题理论基础
 - 分割等和子集问题
 - 粉碎石头问题
 - 目标和问题
 - 一和零问题
 - 完全背包问题
 - 完全背包问题理论基础
 - 零钱兑换问题(一)
 - 零钱兑换问题(二)
 - 单词拆分问题
 - 多重背包问题*
 - 三、打家劫舍问题
 - 打家劫舍问题(一)
 - 打家劫舍问题(二)
 - 打家劫舍问题(三)
 - 四、股票问题
 - 买卖股票问题一
 - 买卖股票问题二(可买卖多次)
 - 买卖股票问题三(最多买卖2次)
 - 买卖股票问题四(最多买卖k次)
 - 买卖股票问题五(有手续费)
 - 买卖股票问题六(有冷却期)
 - 五、子序列问题
 - 不连续子序列
 - 最长公共子序列问题
 - 最长递增子序列问题
 - 连续子序列
 - 最长重复子数组问题
 - 最大子数组问题
 - 编辑距离

- 判断子序列问题
 - 不同子序列问题
 - 删除字符问题
 - 编辑距离问题
 - 回文
 - 回文子串问题
 - 最长回文子序列问题
 - 第四节、记忆化搜索
 - 数据结构学习列表 🤖
-

第一章、数据结构

第一节、STL 🤖 ^4

详细内容 🤖

头文件 `#include<bits/stdc++.h>`

迭代器定义(vector可换做其他容器): `vector<int>::iterator it`, 也可以用 `auto it=...`

一、向量vector 🤖

`vector<int> v[n]`--->在外层为数组, 存储着`vector<int>`类型的对象

- `v.resize(count)` //设置vector的大小为count
- `v.resize(count,a)` //用a补全vector扩充的大小

`iota()`函数--->用于填充vector,头文件为`#include<numeric>`,无返回值

`fill()`函数--->用于填充vector,头文件为`#include<algorithm>`,无返回值

`iota`(起始位置,结束位置,起始值)

`fill`(起始位置,结束位置,填充值)

`iota`从起始值开始填充,随着长度增长填充值增加"1",填充值不一定是数字,字母的话按照ASCII码增加"1"

fill则不会递增，将填充值填满范围

二、字符串string👉

cin>>a与getline(cin,s)不能连用，会导致getline读取cin的回车直接结束，应在中间添加getchar()或cin.get()

```
int n;
string s;
cin >> n;
getchar(); //cin.get()
getline(cin, s); //可正确读入下一行的输入
```

sstring:

去除字符串中的空格

```
stringstream ss("hello string and stringstream")
string str;
// 注意: stringstream 是一个单词一个单词 "流入" string 的
while (ss >> str){
    cout << str << endl;
}
```

常用函数:

函数	含义
s.to_string()	将基本类型的值转换为字符串
s.stoi()	将字符串类型转换为int类型
s.substr(pos,size)	从pos开始截取size长度的字符串
s.append(str)	在字符串结尾添加str字符串
函数	含义
s.find(str)	在s中查找str,返回索引，没找到返回-1
s.replace(pos,n,str)	把当前字符串从索引pos开始的n个字符替换为str
s.replace(pos,n,n1,c)	把当前字符串从索引pos开始的n个字符替换为n1个字符c

函数	含义
<code>s.replace(it1,it2,str)</code>	把当前字符串[<code>it1,it2</code>)区间替换为str <code>it1 ,it2</code> 为迭代器

三、栈stack 🤖

加入头文件`#include< stack >` 初始化与`vector`相同`stack<int>a`; 常用函数:

函数	含义
<code>s.empty()</code>	判断堆栈是否为空
<code>s.pop()</code>	弹出堆栈顶部的元素
<code>s.push()</code>	向堆栈顶部添加元素
<code>s.size()</code>	返回堆栈中元素的个数
<code>s.top()</code>	返回堆栈顶部的元素

四、队列queue 🤖

加入头文件`#include<queue>` 初始化与`vector`相同`queue<int>a`; 常用函数:

函数	含义
<code>q.front()</code>	返回队列中的第一个元素
<code>q.back()</code>	返回队列中最后一个元素
<code>q.empty()</code>	判断队列是否为空
<code>q.pop()</code>	删除队列的第一个元素
<code>q.push()</code>	在队列末尾加入一个元素
<code>q.size()</code>	返回队列中元素的个数

五、双向队列deque 🤖

加入头文件`#include<deque>` 初始化与`vector`相同`deque<int>a`; 常用函数:

函数	含义
<code>dq.push_front()--push_back()</code>	在队列的头(尾)部插入元素

函数	含义
<code>dq.emplace_front()</code> -- <code>emplace_back()</code>	与push的作用一样
<code>dq.pop_front()</code> -- <code>pop_back()</code>	删除队列头(尾)部的元素
<code>dq.back()</code>	返回队列尾部元素的引用
<code>dq.front()</code>	返回队列头部元素的引用
<code>dq.begin()</code>	返回头位置的迭代器
<code>dq.end()</code>	返回尾+1位置的迭代器
<code>dq.clear()</code>	清空队列中的所有元素
<code>dq.empty()</code>	判断队列是否为空。
<code>dq.size()</code>	返回队列中元素的个数

函数	含义
<code>dq.insert(pos,e)</code>	在指定位置pos插入元素e(位置都是迭代器)
<code>dq.insert(pos,n,e)</code>	在指定位置pos插入n个元素e
<code>dq.insert(pos,a,b)</code>	在指定位置pos插入[a,b)区间的元素(a,b都是迭代器)
<code>dq.erase(i)</code>	在指定i位置删除元素
<code>dq.erase(a,b)</code>	在指定(a,b)区间（左闭右开）删除元素

遍历:

```
deque<int>::iterator it; //迭代器定义
for(it=d.begin();it!=d.end();it++){
    cout<<*it<<" "; //注意*t和for中的!=
}
```

六、优先级队列priority_queue 🤖

在头文件`#include<queue>`中

函数	含义
<code>pq.top()</code>	访问队首元素
<code>pq.push()</code>	入队

函数	含义
<code>pq.pop()</code>	堆顶（队首）元素出队
<code>pq.size()</code>	队列元素个数
<code>pq.empty()</code>	是否为空

设置优先级

==!!! `priority_queue`的排列顺序与`sort`完全相反==

基本结构: `priority_queue<类型> 名` (默认为最大堆)

==`priority_queue<int, vector<int>, greater<int>>`== 第二个参数是底层存储类型, 三个参数中`int`位置类型要相同; `less<int>`是最大堆, `greater<int>`是最小堆

自定义结构体比较

```
//一般写法
struct node {
    int x, y;
    bool operator < (const Point &a) const { //直接传入一个参数, 不必要写friend
        return x < a.x; //按x升序排列, x大的在堆顶
    }
};
```

存储`pair`类型时默认对`first`进行降序排列, `first`相同再对`second`排列

七、数据对pair 🤖

`pair<类型1, 类型2>`

成员变量: `first`代表第一个元素, `second`代表第二个元素

`pair`类型定义在`#include<utilit>`头文件中

八、字典map 🤖

`map`

`map<类型1, 类型2>`

不会存在key相等的情况，那样新的就不会再插入

map会按照键的顺序从大到小排列，因此key的类型必须可以比较大小

删除还是使用迭代器删除(迭代器定义 `::iterator iter`)，使用key删除有返回值

函数	含义
<code>m.find(key)</code>	查找key,返回迭代器位置,不存在返回map.end()
<code>m.count(key)</code>	查找元素是否存在，存在返回1，不存在返回0
<code>m.size()</code>	map中的对数
<code>m.clear()</code>	删除所有元素
<code>m.swap()</code>	交换两个map

函数	含义
<code>m.erase(it)</code>	删除it对应元素
<code>m.erase(key)</code>	删除key对应元素
<code>m.erase(begin,end)</code>	删除迭代器区间内元素

添加元素：

- 1. `map[key]=value;`
- 2. `map.insert(pair<int,int>(key,value));`
- 3. `map.insert({key,value});`

字典unordered_map

拥有键值对，用可以来查找value，有find(key)函数

修改value：

```
//直接使用x.value=new_value，在出for循环后值会恢复
for(auto x:unomap)//遍历整个map，输出key及其对应的value值
{
    auto it = unmap.find(key) //改
    if(it != unmap.end())
        it->second = new_value;
}
```

==！！！ 遍历顺序与输入顺序不一定相同 ==

unordered_map具备hash特性*

九、集合set🤩

加入头文件#include<set>

set里面元素不重复且==有序(默认从小到大排列)==, 用set<int,greater<int>>改变为从大到小排列

==map和set中insert不再强调位置, 而是直接插入==

常用函数: 与map完全相同

十、bitset🤩

bitset中元素只能是1或0,保存在#include<bitset>中

定义: bitset<序列长度> 序列名称(初始化元素), 初始化元素若小于长度则右对齐,二维定义: bitset<行长度>序列名称[列长度]

bitset有位运算符(& | ^ ~)

函数	含义
b.any()	b中是否存在置为1的二进制位
b.none()	b中不存在置为1的二进制位
b.count()	b中置为1的二进制位的个数
b.size()	访问b中在pos处的二进制位
b[pos]	访问 b 中在 pos 处的二进制
b.test(pos)	b中在 pos 处的二进制位是否为 1
b.set()	把b中所有二进制位都置为1
b.set(pos)	把b中在 pos 处的二进制位置为 1
b.reset()	把 b 中所有二进制位都置为 0
b.reset(pos)	把 b 中在 pos 处的二进制位置为 0
b.flip()	把 b 中所有二进制位逐位取反
b.flip(pos)	把 b 中在 pos 处的二进制位取反

函数	含义
<code>b.to_ulong()</code>	用b中同样的二进制位返回一个unsigned long值

bitset例题

十一、元组tuple 🍷

在头文件`#include<tuple>`中，可以看作是`pair`的扩展

`tuple<int,int,string>t---`三元组

赋值: `t=make_tuple(a,b,c);`或`tuple<int,int,int>t(1,2,3)`

获取元素: `int a =get<0>(t)---`获得t的第0位

获取元素个数: `int count = tuple_size<decltype(t)>::value---``decltype`(判断t的类型), `::value`指返回一个结果值

解包: `tie(a,b,c)=t---`将t中三个元素分别赋值给三个变量

STL函数

函数	含义
<code>reverse(begin,end)</code>	翻转序列
<code>max_element(a.begin(),a.end())</code>	返回a中最大值的迭代器
<code>min_element(a.begin(),a.end())</code>	返回a中最小值的迭代器

函数	含义
<code>lower_bound(a.begin(),a.end(),x)</code>	返回第一个 $\geq x$ 的值的值的位置(a有序), 自定义comp为找到false的值
<code>lower_bound(a.begin(),a.end(),x,less<type>())</code>	返回第一个 $\geq x$ 的值的值的位置(a有序)
<code>lower_bound(a.begin(),a.end(),x,greater<type>())</code>	返回第一个 $\leq x$ 的值的值的位置(a有序) <code>upper_bound(a.begin(),a.end(),x)</code> 返回第一个 $> x$ 的值的值的位置(a有序),其余相同, 自定义comp为找到true的值

第二节、排序算法 🤪

一、sort排序 😊

`sort(begin,end)`中`[begin,end)`，默认升序排列

`sort(begin,end,greater<type>())`降序排列

自定义排序时返回`a>b`是降序排列，即返回true值

`stable_sort()`用法相同，但数据相等时不进行交换，保持稳定性

匿名函数写法：

```
[捕捉变量列表] (参数列表) -> 返回类型{函数体}
```

只有一个return时可省略返回类型

```
[] // 未定义变量.试图在Lambda内使用任何外部变量都是错误的.
```

```
[x, &y] // x 按值捕获, y 按引用捕获.
```

```
[&] // 用到的任何外部变量都隐式按引用捕获
```

```
[=] // 用到的任何外部变量都隐式按值捕获
```

```
[&, x] // x显式地按值捕获. 其它变量按引用捕获
```

```
[=, &z] // z按引用捕获. 其它变量按值捕获
```

示例

```
sort(v.begin(), v.end(), [](int a, int b) { //a和b是列表中元素，可能还是列表
    return a > b; // 降序排列
});
```

二、冒泡排序 🤪

时间复杂度 $O(n^2)$

交换次数等于逆序对数量

```

void BubbleSort(int a[],int left,int right){
    for(int i=left;i<=right;i++){
        for(int j=right-1;j>=i;i--){
            if(a[j]>a[j+1]){
                swap(a[j],a[j+1]);
            }
        }
    }
}

```

三、快速排序 🤪

时间复杂度平均 $O(n \cdot \log(n))$, 最坏情况 $O(n^2)$

```

//优化->三数取中法取轴点
int Partition(int a[],int left,int right){
    int i=left;
    int j=right-1;
    int p=a[right];
    while(true){
        while(a[i]<p){
            i++;
        }
        while(a[j]>p && j>left){
            j--;
        }
        if(i>=j){
            break;
        }
        swap(a[i],a[j]);
        i++;
        j++;
    }
    swap(a[i],p);
    return i;
}

void QuickSort(int a[],int left,int right){
    if(left<right){
        int i=Partition(a,left,right);
        QuickSort(a,left,i-1);
        QuickSort(a,i+1,right);
    }
}

```

四、归并排序 🤪

时间复杂度 $O(n \cdot \log(n))$

经典归并排序

```
vector<int> TwoWayMerge(int a[],int lx,int rx,int ly,int ry){
    vector<int>v;
    int i=lx;
    int j=rx;
    while(i<=lx || j<=rx){
        if(j>rx || (i<=lx && a[i]<=a[j])){
            v.push_back(a[i]);
            i++;
        }
        else{
            v.push_back(a[j]);
            j++;
        }
    }
    return v;
}

//自顶向下
vector<int> MergeSort(int a[],int left,int right){
    vector<int>v;
    if(left<right){
        int m=(left+right)/2;
        MergeSort(a,left,m);
        MergeSort(a,m+1,right);
        v=TwoWayMerge(a,left,m,m+1,right);
    }
    return v;
}

//自下向上
vector<int> MergeSortUp(int a[],int left,int right){
    int len=1;
    int n=right-left+1;
    vector<int>v;
    while(len<n){
        int lx=0;
        int rx;
        int ly;
        int ry;
        while(lx<=right-len){
            int rx=lx+len-1;
            int ly=rx+1;
            int ry=min(ly+len-1,right);
            v=TwoWayMerge(a,lx,rx,ly,ry); //这一有一点问题，不能直接用v
            lx=ry+1;
        }
    }
    return v;
}
```

感觉快排和归并很像，都像是二分,感觉快排像是自顶至下的归并

链表归并排序

```
link* LinkSort(link* list1, link* list2){
    link* p1=list1;
    link* p2=list2;
    link* pre=NULL;
    while(p2!=NULL){
        while(p1!=NULL && p1->data<p2->data){
            pre=p1;
            p1=p1->next;
        }
        link* temp=p2;
        p2=p2->next;
        temp->next=p1;
        if(pre==NULL){
            list1=temp;
        }
        else{
            pre->next=temp;
        }
        p1=temp;
    }
    return list1;
}
```

求逆序对数量 🐼

```
int TwoWayInversionCount(vector<int>& a, vector<int>& temp, int l, int m, int r) {
    int i = l;
    int j = m + 1;
    int k = l;
    int count = 0;

    while (i <= m && j <= r) {
        if (a[i] <= a[j]) {
            temp[k++] = a[i++];
        } else {
            temp[k++] = a[j++];
            count += (m - i + 1);
        }
    }

    while (i <= m) {
        temp[k++] = a[i++];
    }

    while (j <= r) {
        temp[k++] = a[j++];
    }

    for (int idx = l; idx <= r; idx++) {
        a[idx] = temp[idx];
    }
}
```

```

    }

    return count;
}

int InversionCount(vector<int>& a, vector<int>& temp, int l, int r) {
    int count = 0;

    if (l < r) {
        int m = (l + r) / 2;

        count += InversionCount(a, temp, l, m);
        count += InversionCount(a, temp, m + 1, r);
        count += TwoWayInversionCount(a, temp, l, m, r);
    }

    return count;
}

```

五、计数排序 🤪

```

vector<int> CountSort(int a[],int n,int max){
    vector<int>v(0,n);
    int c[max];
    for(int i=0;i<max;i++){
        c[i]=0;
    }
    for(int i=0;i<n;i++){
        c[a[i]]++;
    }
    for(int i=1;i<max;i++){
        c[i]=c[i]+c[i-1];
    }
    for(int i=n-1;i>=0;i--){
        v[c[a[i]]]=a[i];
        c[a[i]]--;
    }
    return v;
}

```

第三节、树tree 🌀

一、二叉树 🤖

头文件1

```
#include<iostream>
#include<vector>
#include<string>
#include<queue>
using namespace std;
```

构建二叉树结构体1

```
//构建二叉树结构体
typedef struct node{
    int data;
    node*left,*right;
    node(int x):data(x),left(NULL),right(NULL){} //构造函数
}BT;
```

前序序列反序列化

```
//前序序列反序列化
int k=-1; //全局变量k
BT* PreDESer(string s){
    k++;
    int n=s.size();
    node*tree=NULL;
    if(k<n){
        int data=int(s[k]); //to_string()使int转string, stoi()是string转int
        if(s[k]!='#'){ //此处#意为空树
            tree=new node(data);
            tree->data=data;
            tree->left=PreDESer(s);
            tree->right=PreDESer(s); //重构左右子树, k决定data
        }
    }
    return tree;
}
```

前中序列反序列化, 无"#" 🤖

```
//前中序列反序列化, 无#
BT* PreInDESer(string pre, string inorder,int &preIndex,int &Instart, int &Inend ){
    if(Instart>Inend) return NULL;
    node*tree=new node(int(pre[preIndex]));
    preIndex++;
    int Index=Instart;
    for(;Index<Inend;Index++){
        if(inorder[Index]==pre[preIndex]) break;
    }
```

```

    }
    tree->left=PreInDESer(pre, inorder, preIndex, Instart, Index-1 );
    tree->right=PreInDESer(pre, inorder, preIndex, Index+1, Inend );
    return tree;
}

```

删除树

```

//删除树
void remove(BT* tree) {
    if (tree == nullptr) return;
    remove(tree->left);
    remove(tree->right);
    delete tree;
    tree = nullptr;
}

```

前序，中序，后序遍历

```

//前序，中序，后序遍历
void PreOrder(BT*tree){
    if(tree!=NULL){
        cout<<tree->data<<" ";
        PreOrder(tree->left);
        PreOrder(tree->right);
    }
}

void InOrder(BT*tree){
    if(tree!=NULL){
        InOrder(tree->left);
        cout<<tree->data<<" ";
        InOrder(tree->right);
    }
}

void PostOrder(BT*tree){
    if(tree!=NULL){
        PostOrder(tree->left);
        PostOrder(tree->right);
        cout<<tree->data<<" ";
    }
}

```

层序遍历

```
//层序遍历
void Sequence(BT*tree){
    queue<BT*>q;
    q.push(tree);
    while(!q.empty()){
        BT*node_ptr=q.front();
        q.pop();
        if(node_ptr!=NULL){
            cout<<node_ptr->data<<" ";
            if(node_ptr->left) q.push(node_ptr->left);
            if(node_ptr->right) q.push(node_ptr->right);
        }
    }
}
```

主函数1

```
int main(){
    int a=0;
    BT*tree=new node(a);
    return 0;
}
```

二、二叉搜索树 🤖

头文件2

```
#include<iostream>
#include<vector>
#include<string>
using namespace std;
```

构建二叉树结构体2

```
//构建二叉树结构体
typedef struct node{
    int data;
    node*left,*right;
    node(int x):data(x),left(NULL),right(NULL){} //构造函数
}BT;
```

查找

```
//查找
BT* Search(BT* bst, int key){
    if(bst==NULL || bst->data==key) return bst;
    else if(bst->data < key) return Search(bst->left, key);
    else return Search(bst->right, key);
}
```

插入1

```
//插入
BT* Insert(BT* bst, int key){
    if(bst==NULL) bst=new node(key);
    else if(key<bst->data) bst->left=Insert(bst->left, key);
    else bst->right=Insert(bst->right, key);
    return bst;
}
```

删除1 🤖

```
//删除
BT* Removal(BT* bst, int key){
    //二分查找找到节点位置
    BT* node=bst;
    BT* father=NULL;
    while(node!=NULL && node->data!=key){
        father=node;
        if(key<node->data) node=node->left;
        else node=node->right;
    }
    if(node==NULL) return bst;
    //删除结点的左右子树非空，使其变为至多有一个子节点的情况，找到后继节点替换
    if(node->left!=NULL && node->right!=NULL){
        BT* temp=node;
        father=node;
        node=node->right;
        while(node->left!=NULL){
            father=node;
            node=node->left;
        }
        temp->data=node->data;
    }
    //删除
    BT* node_ptr=NULL; //把子树先存起来
    if(node->left!=NULL) node_ptr=node->right;
    else if(node->right!=NULL) node_ptr=node->left;
```

```
if(father==NULL) bst=node_ptr;
else if(node==father->left) father->left=node_ptr;
else father->right=node_ptr;
}
```

主函数2

```
int main(){
    int a=0;
    BT*tree=new node(a);
    return 0;
}
```

三、AVL树 🤖

AVL头文件

```
#include<iostream>
#include<vector>
#include<string>
#include<algorithm>
using namespace std;
```

构建二叉树结构体3

```
//构建二叉树结构体
typedef struct node{
    int data;
    int height; //注意新加进结构体的属性
    node*left,*right;
    node(int x):data(x),left(NULL),right(NULL),height(1){
    } //构造函数
}BT;
```

//获取高度

时间复杂度O(1)

```
//获取高度
int GetHeight(BT*tree){
    if(tree=NULL) return 0;
```

```
    else return tree->height;
}
```

更新高度

时间复杂度 $O(1)$

```
//更新高度
//想法：在插入一个节点需要更新之前所有节点，在有维护函数后感觉不需要了
void UpdateHeight(BT*tree){
    if(tree!=NULL){
        int height_l=GetHeight(tree->left);
        int height_r=GetHeight(tree->right);
        tree->height=max(height_l,height_r)+1;
    }
}
```

左右旋

时间复杂度 $O(1)$

```
//左右旋
//先更新子树再更新树
BT* LeftRotate(BT*tree){
    BT*node=tree->right;
    tree->right=node->left;
    UpdateHeight(tree);
    node->left=tree;
    UpdateHeight(node);
    return node;
}

BT* RightRotate(BT*tree){
    BT*node=tree->left;
    tree->left=node->right;
    UpdateHeight(tree);
    node->right=tree;
    UpdateHeight(node);
    return node;
}
```

插入2

```
//插入
//与二叉平衡树相同
BT* Insert(BT* bst, int key){
```



```

if(bst==NULL) bst=new node(key);
else if(key<bst->data) bst->left=Insert(bst->left,key);
else bst->right=Insert(bst->right,key);
return bst;
}

```

删除2

```

//删除
//与二叉平衡树相同
BT* Removal(BT*bst, int key){
    //二分查找找到节点位置
    BT* node=bst;
    BT* father=NULL;
    while(node!=NULL && node->data!=key){
        father=node;
        if(key<node->data) node=node->left;
        else node=node->right;
    }
    if(node==NULL) return bst;
    //删除结点的左右子树非空，使其变为至多有有一个子节点的情况，找到后继节点替换
    if(node->left!=NULL && node->right!=NULL){
        BT* temp=node;
        father=node;
        node=node->right;
        while(node->left!=NULL){
            father=node;
            node=node->left;
        }
        temp->data=node->data;
    }
    //删除
    BT*node_ptr=NULL;
    if(node->left!=NULL) node_ptr=node->right;
    else if(node->right!=NULL) node_ptr=node->left;

    if(father==NULL) bst=node_ptr;
    else if(node==father->left) father->left=node_ptr;
    else father->right=node_ptr;
    return bst;
}

```

自平衡 🤖

时间复杂度 $O(\log(n))$

```

//自平衡,插入删除可使用上述函数，node表示插入节点或删除结点的父节点
BT* Balance(BT*tree,BT*node){
    if(tree!=node){
        if(tree->data<node->data){

```

```

        tree->left=Balance(tree->left,node);
    }
    else{
        tree->right=Balance(tree->right,node);
    }    //保存根到node的路径
}    //某一子树tree=node
UpdateHeight(tree);

if(GetHeight(tree->left)-GetHeight(tree->right)==2){
    if(GetHeight(tree->left->right)<GetHeight(tree->left->left)){
        tree->left=LeftRotate(tree->left);
    }
    tree=RightRotate(tree);
}
if(GetHeight(tree->right)-GetHeight(tree->left)==2){
    if(GetHeight(tree->right->left)<GetHeight(tree->right->right)){
        tree->right=RightRotate(tree->left);
    }
    tree=LeftRotate(tree);
}
return tree;
}

```

应用:树高上界

四、红黑树

最长路径不超过最短路径的两倍. 在结构体中增加 parents 属性

插入:

插入结点默认为红

插入结点是根结点 → 直接变黑


插入结点的叔叔是红色 → 叔父爷变色, 爷爷变插入结点 重新考虑

插入结点的叔叔是黑色 → (LL,RR,LR,RL)旋转, 然后变色 与AVL相似 爷父变色

删除:

红黑树

删除

只有  这种情况, 直接使红代替黑后变黑

只有左孩子/只有右孩子: 代替后变黑

红结点: 删除后无需任何调整

没有孩子

两个孩子使用前驱、后继结点交换

黑结点 (删除后变双黑)

兄弟是黑色

兄弟的孩子都是黑色: 兄弟变红, 双黑上移 向上递推至红或根结点

兄弟是红色: 兄父变色, 朝双黑旋转 → 完成后继续检查 兄父均相反 (保持双黑继续调整)

两个红结点看作LL型

(LL,RR) - r变s,s变p,p变黑

(LR,RL) - r变p,p变黑

兄弟至少有一个红孩子: (LL,RR,LR,RL)变色+旋转 → AVL (双黑变单黑)



! 以上方法完成后仍需再次检查

第四节、二叉堆BinaryHeap 🤪

最大堆, 最小堆, 最大最小堆, 对顶堆(找到第k小的元素)

头文件及定义

```
#include<iostream>
#include<vector>
#include<string>
using namespace std;

vector<int>h; //未赋值
```

上调

时间复杂度 $O(\log(n))$

```
//上调
void SiftUp(vector<int> h,int i){ //i是起始位置
    int elem=h[i];
    while(i>1 && elem<h[i/2]){
        h[i]=h[i/2];
        i=i/2;
    }
    h[i]=elem;
}
```

下调

时间复杂度 $O(\log(n))$

```
//下调
void SiftDown(vector<int> h,int i){
    int last=h.size();
    int elem=h[i];
    int child=2*i;
    while(true){
        //找到更小的子节点
        if(child<last && h[child]>h[child+1]){
            child+=1;
        }
        else if(child>last){
            break;
        }
        //下调
        if(h[child]<elem){
            h[i]=h[child];
            i=child;
            child=2*i;
        }
        else{
            break;
        }
    }
    h[child]=elem;
}
```

插入

时间复杂度 $O(\log(n))$

```
//插入
void insert(vector<int>h,int x){
    h.push_back(x);
    SiftUp(h,h.size());
}
```

删除顶元素

时间复杂度 $O(\log(n))$

```
//删除顶元素
int DeleteMin(vector<int>h){
    int min=h[0];
    h[0]=h[h.size()-1];
    SiftDown(h,0);
    return min;
}
```

朴素建堆

时间复杂度 $O(n*\log(n))$

```
//朴素建堆
void MakeHeap(vector<int>h){
    for(int i=1;i<h.size();i++){
        SiftUp(h,i);
    }
}
```

快速建堆

时间复杂度 $O(n)$

```
//快速建堆
void MakeHeapDown(vector<int>h){
    for(int i=(h.size()/2);i>=1;i--){
        SiftDown(h,i);
    }
}
```

第五节、静态查找Search 🤪

一、顺序查找 🤪

查找最大最小值

时间复杂度 $3/2 \cdot n$

```
#include<iostream>
using namespace std;
# define n 100

int main(){
    int a[n];
    int max=a[0],min=a[0];
    int k=n%2;
    while(k<n-1){
        if(a[k]<a[k+1]){
            if(min>a[k]){
                min=a[k];
            }
            if(max<a[k+1]){
                max=a[k+1];
            }
        }
        else{
            if(min>a[k+1]){
                min=a[k+1];
            }
            if(max<a[k]){
                max=a[k];
            }
        }
        k+=2;
    }
    return 0;
}
```

查找素数

埃氏筛选法 🤪

时间复杂度 $O(n \cdot \log(\log(n)))$

```
vector<int> ESearch(int n){
    vector<int>v;
    bool prime[n+1];
    prime[0]=false;
```

```

prime[1]=false;
for(int i=2;i<=n;i++){
    prime[i]=true;
}
for(int i=2;i<=n;i++){
    if(prime[i]==true){
        v.push_back(i);
        int m=2*i;
        while(m<=n){
            prime[m]=false;
            m+=i;
        }
    }
}
}

```

欧拉筛选法 🤪

时间复杂度 $O(n)$

```

vector<int> EulerSearch(int n){
    vector<int>v;
    bool prime[n+1];
    prime[0]=false;
    prime[1]=false;
    for(int i=2;i<=n;i++){
        prime[i]=true;
    }
    for(int i=2;i<=n;i++){
        if(prime[i]==true){
            v.push_back(i);
        }
        int k=0;
        while(i*v[k]<=n){
            prime[i*v[k]]=false;
            if(i%v[k]==0){
                break;
            }
            k++;
        }
    }
}

```

二、二分查找 🤪

二分的思想十分重要

查找顺序表元素

时间复杂度 $O(\log(n))$

```
int BinarySearch(int a[],int left,int right,int key){
    int low=left-1;
    int high=right+1;
    while(high-low==1){
        int mid=(high+low)/2;
        if(a[mid]==key){
            return mid;
        }
        else if(a[mid]<key){
            high=mid;
        }
        else{
            low=mid;
        }
    }
    return -1;
}
```

未排序序列快速查找k小元素

先进行二分排序，再进行二分查找 时间复杂度 $O(1)$ -- $O(n^2)$

寻找最大的最小距离

时间复杂度 $O(n*\log(X[n]-X[1]))$

```
//m头牛安放进n间牛舍中，牛舍间距不同,寻找最大的最小间距
//转化为了找满足条件的最小(大)值
int MaxSpace(int a[],int n,int m){
    //a为n间牛舍的位置序列
    int min=0;
    int max=a[n-1]-a[0]+1;
    while(max-min>1){
        int mid=(max+min)/2;
        int num=1; //第num头牛
        int pre=0; //第num头牛在的位置
        for(int k=1;k<n;k++){
            if(a[k]-a[pre]>=mid){
                pre=k;
                num++;
            }
        }
        if(num==m){
            return mid;
        }
        else if(num>m){
            min=mid;
        }
    }
}
```



```

        else{
            max=mid;
        }
    }
}

```

第六节、图graph 🌀

一、图基础功能¹

头文件

```

#include<iostream>
#include<vector>
#include<queue>
using namespace std;
# define maxnum 100

```

邻接矩阵的存储结构

```

struct Graph1{
    int Vex[maxnum]; //顶点表---int表示符号位置
    int Edge[maxnum][maxnum]; //边表---含权值
    int Vnum1,Enum1; //点，边数
};

```

邻接链表表示

其实用一个二重vector也能实现链表操作，不要痴迷于链表

```

struct Edge{ //相邻点链表
    int src; //表示该节点位置
    int weight; //权重
    Edge *next; //指向下一节点
    Edge(int x,int y):src(x),weight(y),next(NULL){}
};

struct Vex{
    int data;
    Edge *adj; //指向相邻点链表
};

```

```

struct Graph2{
    Vex vex[maxnum];
    int Vnum2,Enum2;
};
Edge* edge = new Edge(a); //构造图中线

```

合理选择存储方式

用邻接矩阵法删除节点

```

Graph1 RemoveVex1(Graph1 graph,int v){
    if(v<0 &&v>graph.Vnum1){
        return graph;
    }
    //计算该节点所连边的数量
    int count=0;
    for(int u=0;u<graph.Vnum1;u++){ //删除射出的边
        if(graph.Edge[v][u]) count++;
    }
    for(int u=0;u<graph.Vnum1;u++){ //删除射入的边
        if(graph.Edge[u][v]) count++;
    }
    //改变图信息
    graph.Vex[v]=graph.Vex[graph.Vnum1-1]; //替换节点信息

    for(int u=0;u<graph.Enum1;u++){ //改变边邻接矩阵的行
        graph.Edge[v][u]=graph.Edge[graph.Vnum1][u];
    }

    for(int u=0;u<graph.Enum1;u++){ //改变边邻接矩阵的列
        graph.Edge[u][v]=graph.Edge[u][graph.Vnum1];
    }

    graph.Enum1-=count;
    graph.Vnum1-=1;

    return graph;
}

```

二、深度优先遍历

时间复杂度 $O(V+E)$

DFS算法实现

```

#include <iostream>
#include <vector>

```

```

using namespace std;

// 图的邻接表表示
struct Graph {
    int V; // 顶点数
    vector<vector<int>> adj; // 邻接表

    Graph(int V) : V(V), adj(V) {}

    void addEdge(int u, int v) {
        adj[u].push_back(v); // 添加边 u -> v
        adj[v].push_back(u); // 添加边 v -> u (如果是无向图)
    }
};

// DFS 递归核心代码
void DFSUtil(const Graph& graph, int node, vector<bool>& visited) {
    visited[node] = true; // 标记当前节点已访问
    cout << node << " "; // 输出当前节点

    for (int neighbor : graph.adj[node]) { // 遍历邻接节点
        if (!visited[neighbor]) { // 如果未访问
            DFSUtil(graph, neighbor, visited);
        }
    }
}

// 外部调用 DFS
void DFS(const Graph& graph, int start) {
    vector<bool> visited(graph.V, false); // 访问标记
    DFSUtil(graph, start, visited);
}

int main() {
    Graph graph(5); // 创建包含5个节点的图
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);

    cout << "DFS starting from node 0: ";
    DFS(graph, 0); // 从节点 0 开始 DFS
    cout << endl;

    return 0;
}

```

应用：寻找路径

给定图，判断两点之间是否存在路径

```

#include <iostream>
#include <vector>

```

```

#include <stack>
using namespace std;

// 图的邻接表表示
struct Graph {
    int V; // 顶点数
    vector<vector<int>> adj; // 邻接表

    Graph(int V) : V(V), adj(V) {}

    void addEdge(int u, int v) {
        adj[u].push_back(v); // 添加边 u -> v
    }
};

// 深度优先搜索寻找路径
bool DFS_Find(const Graph& graph, int v, int t, vector<bool>& visited, vector<int>& pre) {
    visited[v] = true; // 标记当前节点已访问
    if (v == t) return true; // 如果找到目标节点, 返回 true

    for (int neighbor : graph.adj[v]) { // 遍历邻接节点
        if (!visited[neighbor]) { // 如果未访问
            pre[neighbor] = v; // 记录前驱节点
            if (DFS_Find(graph, neighbor, t, visited, pre)) {
                return true; // 如果找到路径, 直接返回 true
            }
        }
    }
    return false; // 未找到路径
}

// 寻找从 s 到 t 的路径
void FindPath(const Graph& graph, int s, int t) {
    vector<bool> visited(graph.V, false); // 访问标记
    vector<int> pre(graph.V, -1); // 记录路径

    if (DFS_Find(graph, s, t, visited, pre)) { // 如果找到路径
        stack<int> path;
        for (int v = t; v != -1; v = pre[v]) {
            path.push(v); // 逆向存储路径
        }

        // 输出路径
        while (!path.empty()) {
            cout << path.top();
            path.pop();
            if (!path.empty()) cout << " -> ";
        }
        cout << endl;
    } else {
        cout << "No Path" << endl;
    }
}

// 主函数示例
int main() {

```

```

Graph graph(5); // 创建包含 5 个节点的图
graph.addEdge(0, 1);
graph.addEdge(0, 2);
graph.addEdge(1, 3);
graph.addEdge(2, 4);
graph.addEdge(3, 4);

int s = 0, t = 4; // 起点和终点
cout << "Finding path from " << s << " to " << t << ":\n";
FindPath(graph, s, t);

return 0;
}

```

应用：走迷宫

```

bool Find_PathMaze(int *map[],int m,int n,int x,int y,int tx,int ty,int *pre[]){
    bool visit[m][n];
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            visit[i][j]=false;
        }
    }
    visit[x][y]=true;

    int adj[4][2]={{-1,0},{1,0},{0,-1},{0,1}};

    if(x==tx && y==ty){
        return true;
    }
    for(int k=0;k<4;k++){
        int nx=x+adj[k][0];
        int ny=y+adj[k][1];
        if(nx>=0 && nx<m && ny>=0 && ny<n && map[nx][ny]!=1 && visit[nx]
[ny]==false){
            pre[nx][ny]=k;
            if(Find_PathMaze(map,m,n,nx,ny,tx,ty,pre)==true){
                return true;
            }
        }
    }
    return false;
}

```

三、广度优先遍历

时间复杂度 $O(V+E)$

BFS算法实现

```

#include <iostream>
#include <vector>
#include <queue>
using namespace std;

// 图的邻接表表示
struct Graph {
    int V; // 顶点数
    vector<vector<int>> adj; // 邻接表

    Graph(int V) : V(V), adj(V) {}

    void addEdge(int u, int v) {
        adj[u].push_back(v); // 添加边 u -> v
        adj[v].push_back(u); // 添加边 v -> u (如果是无向图)
    }
};

// BFS 核心代码
void BFS(const Graph& graph, int start) {
    vector<bool> visited(graph.V, false); // 访问标记
    queue<int> q; // 队列

    q.push(start); // 起始点入队
    visited[start] = true; // 标记起始点已访问

    while (!q.empty()) {
        int node = q.front(); // 取队首元素
        q.pop();
        cout << node << " "; // 输出当前节点

        for (int neighbor : graph.adj[node]) { // 遍历邻接节点
            if (!visited[neighbor]) { // 如果未访问
                visited[neighbor] = true; // 标记为已访问
                q.push(neighbor); // 入队
            }
        }
    }
}

int main() {
    Graph graph(5); // 创建包含5个节点的图
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);

    cout << "BFS starting from node 0: ";
    BFS(graph, 0); // 从节点 0 开始 BFS
    cout << endl;

    return 0;
}

```

例题

跳一跳，只有三个位置，求到达终点的最少次数

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
using namespace std;

int minJumpsToReachEnd(const vector<int>& nums) {
    int n = nums.size();
    if (n == 1) return 0; // 如果数组长度为1，已经在末尾，不需要跳跃

    // BFS 初始化，BFS思想
    queue<pair<int, int>> q; // (index, jump_count)
    vector<bool> visited(n, false);
    unordered_map<int, vector<int>> value_map; // 存储值到索引的映射,即所有能跳到的地

    // 将每个值的索引存储在 value_map 中
    for (int i = 0; i < n; ++i) {
        value_map[nums[i]].push_back(i);
    }

    // 初始状态：从0开始
    q.push({0, 0});
    visited[0] = true;

    // BFS
    while (!q.empty()) {
        auto [index, jumps] = q.front();
        q.pop();

        // 如果当前下标已经是最后一个位置，返回跳跃次数
        if (index == n - 1) {
            return jumps;
        }

        // 尝试跳到 i + 1
        if (index + 1 < n && !visited[index + 1]) {
            visited[index + 1] = true;
            q.push({index + 1, jumps + 1}); //注意每次加1
        }

        // 尝试跳到 i - 1
        if (index - 1 >= 0 && !visited[index - 1]) {
            visited[index - 1] = true;
            q.push({index - 1, jumps + 1});
        }

        // 尝试跳到与 nums[i] 相等的所有位置
        if (value_map.find(nums[index]) != value_map.end()) {
            // 访问当前值的所有相同值位置
            for (int next_index : value_map[nums[index]]) {
```

方

```

        if (!visited[next_index]) {
            visited[next_index] = true;
            q.push({next_index, jumps + 1});
        }
    }
    // 访问完后清空该值的所有位置，避免重复访问
    value_map.erase(nums[index]);
}

return -1; // 如果无法到达最后一个位置，返回 -1
}

int main() {
    int n;
    cin >> n;
    vector<int> nums(n);
    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }

    int result = minJumpsToReachEnd(nums);
    cout << result << endl;

    return 0;
}

```

四、欧拉回路 🤖

```

#include <iostream>
#include <vector>
#include <stack>
using namespace std;

// 检查图是否具有欧拉回路的函数
bool hasEulerianCircuit(const vector<vector<int>>& graph) {
    int oddDegreeCount = 0;
    for (const auto& adj : graph) {
        if (adj.size() % 2 != 0) {
            oddDegreeCount++;
        }
    }
    return oddDegreeCount == 0;
}

// 使用Fleury算法查找欧拉回路的函数
void findEulerianCircuit(vector<vector<int>>& graph, int start) {
    stack<int> path; // 用于存储当前路径的栈
    vector<int> circuit; // 用于存储欧拉回路的向量

    path.push(start);

```



```

while (!path.empty()) {
    int u = path.top();

    //如果当前顶点没有更多边
    if (graph[u].empty()) {
        circuit.push_back(u);
        path.pop();
    } else {
        // 移动到下一个顶点并删除边缘
        int v = graph[u].back();
        graph[u].pop_back();
        // 从 v 到 u 删除反向边
        auto it = find(graph[v].begin(), graph[v].end(), u);
        if (it != graph[v].end()) {
            graph[v].erase(it);
        }
        path.push(v);
    }
}

// 打印欧拉回路
cout << "Eulerian Circuit: ";
for (int v : circuit) {
    cout << v << " ";
}
cout << endl;
}

int main() {
    int n, m;
    cout << "Enter the number of vertices and edges: ";
    cin >> n >> m;

    vector<vector<int>> graph(n);

    cout << "Enter the edges (u v):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u); // 图是无向的
    }

    if (!hasEulerianCircuit(graph)) {
        cout << "The graph does not have an Eulerian circuit." << endl;
    } else {
        int start = 0; // 起始顶点 (可以是任何有边的顶点)
        findEulerianCircuit(graph, start);
    }

    return 0;
}

```

五、无向图的割点与桥 🤖

时间复杂度 $O(V+E)$

```
//DFS求图的割点(Tarjan算法)
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

const int MAXN = 10000; // 最大顶点数量
vector<int> adj[MAXN]; // 邻接表表示图
vector<bool> visited(MAXN, false); // 记录是否访问过
vector<int> discovery(MAXN, -1); // 记录发现时间
vector<int> low(MAXN, -1); // 记录子树中最早被访问的顶点时间
vector<int> parent(MAXN, -1); // 记录顶点的父节点
vector<int> articulationPoints; // 存储割点
vector<pair<int, int>> bridges; // 存储桥

int timeCounter = 0; // 时间计数器

void DFS(int u) {
    visited[u] = true;
    discovery[u] = low[u] = ++timeCounter;
    int children = 0;

    for (int v : adj[u]) {
        if (!visited[v]) {
            children++;
            parent[v] = u;
            DFS(v);

            // 更新 low[u], 以包含子树中的最小值
            low[u] = min(low[u], low[v]);

            // 判断割点
            if (parent[u] == -1 && children > 1) {
                articulationPoints.push_back(u);
            }
            if (parent[u] != -1 && low[v] >= discovery[u]) {
                articulationPoints.push_back(u);
            }

            // 判断桥
            if (low[v] > discovery[u]) {
                bridges.push_back({u, v});
            }
        } else if (v != parent[u]) {
            // 更新 low 值, 处理回边
            low[u] = min(low[u], discovery[v]);
        }
    }
}

void findArticulationPointsAndBridges(int n) {
```

```

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            DFS(i);
        }
    }
}

int main() {
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;

    cout << "Enter the edges (u v):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u); // 无向图
    }

    findArticulationPointsAndBridges(n);

    cout << "Articulation Points:\n";
    for (int point : articulationPoints) {
        cout << point << " ";
    }
    cout << endl;

    cout << "Bridges:\n";
    for (auto bridge : bridges) {
        cout << bridge.first << " - " << bridge.second << endl;
    }

    return 0;
}
//对于树边(u,v)如果dfn(u)<low(v),则(u,v)是桥

```

六、有向图的强连通分量 🧠

时间复杂度 $O(V+E)$

kosaraju算法 🤔

```

#include <iostream>
#include <vector>
#include <stack>
#include <algorithm>
using namespace std;

const int MAXN = 10000; // 最大顶点数量
vector<int> adj[MAXN]; // 原图的邻接表

```

```

vector<int> revAdj[MAXN]; // 反转图的邻接表
vector<bool> visited(MAXN, false); // 记录是否访问过
stack<int> finishStack; // 存储第一次 DFS 的完成顺序
vector<vector<int>> scc; // 存储所有强连通分量

// 第一次 DFS: 记录顶点的完成顺序
void dfs1(int u) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) {
            dfs1(v);
        }
    }
    finishStack.push(u);
}

// 第二次 DFS: 在反转图中找到一个强连通分量
void dfs2(int u, vector<int>& component) {
    visited[u] = true;
    component.push_back(u);
    for (int v : revAdj[u]) {
        if (!visited[v]) {
            dfs2(v, component);
        }
    }
}

// Kosaraju 主函数
void kosarajuSCC(int n) {
    // 第一次 DFS: 记录完成顺序
    fill(visited.begin(), visited.begin() + n, false);
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs1(i);
        }
    }

    // 反转图
    for (int u = 0; u < n; u++) {
        for (int v : adj[u]) {
            revAdj[v].push_back(u);
        }
    }

    // 第二次 DFS: 按完成顺序处理反转图
    fill(visited.begin(), visited.begin() + n, false);
    while (!finishStack.empty()) {
        int u = finishStack.top();
        finishStack.pop();
        if (!visited[u]) {
            vector<int> component;
            dfs2(u, component);
            scc.push_back(component);
        }
    }
}

```

```

int main() {
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;

    cout << "Enter the edges (u v):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v); // 有向图
    }

    kosarajuSCC(n);

    cout << "Strongly Connected Components:\n";
    for (const auto& component : scc) {
        for (int v : component) {
            cout << v << " ";
        }
        cout << endl;
    }

    return 0;
}

```

Tarjan算法😓

```

#include <iostream>
#include <vector>
#include <stack>
#include <algorithm>
using namespace std;

const int MAXN = 10000; // 最大顶点数量
vector<int> adj[MAXN]; // 邻接表表示图
vector<int> discovery(MAXN, -1); // 记录发现时间
vector<int> low(MAXN, -1); // 记录子树中最早被访问的顶点时间
vector<bool> onStack(MAXN, false); // 判断顶点是否在栈中
stack<int> stk; // 栈，用于追踪当前访问路径
vector<vector<int>> scc; // 存储所有强连通分量

```

```

int timeCounter = 0; // 时间计数器

```

```

void tarjanDFS(int u) {
    discovery[u] = low[u] = ++timeCounter;
    stk.push(u);
    onStack[u] = true;

    for (int v : adj[u]) {
        if (discovery[v] == -1) {
            // v 未访问
            tarjanDFS(v);
            low[u] = min(low[u], low[v]);
        }
    }

    if (discovery[u] == low[u]) {
        // 找到一个强连通分量
        vector<int> sccComponent;
        while (!stk.empty()) {
            sccComponent.push_back(stk.top());
            stk.pop();
            onStack[sccComponent.back()] = false;
        }
        scc.push_back(sccComponent);
    }
}

```

```

        } else if (onStack[v]) {
            // v 在栈中，是一条回边
            low[u] = min(low[u], discovery[v]);
        }
    }

    // 如果 u 是一个强连通分量的根
    if (low[u] == discovery[u]) {
        vector<int> component;
        while (true) {
            int v = stk.top();
            stk.pop();
            onStack[v] = false;
            component.push_back(v);
            if (v == u) break;
        }
        scc.push_back(component);
    }
}

void findSCCs(int n) {
    for (int i = 0; i < n; i++) {
        if (discovery[i] == -1) {
            tarjanDFS(i);
        }
    }
}

int main() {
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;

    cout << "Enter the edges (u v):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v); // 有向图
    }

    findSCCs(n);

    cout << "Strongly Connected Components:\n";
    for (const auto& component : scc) {
        for (int v : component) {
            cout << v << " ";
        }
        cout << endl;
    }

    return 0;
}

```

时间复杂度 $O(V+E)$

```
//BFS算法(顺序为BFS近似的顺序)
void CountInDegree(Graph2 graph,int n,int indegree[]){
    int indegree[n];
    for(int i=0;i<n;i++){
        indegree[i]=0;
    }
    for(int i=0;i<n;i++){
        Edge* p=graph.vex[i].adj;
        while(p!=NULL){
            indegree[p->src]++;
            p=p->next;
        }
    }
}

void TopSort_BFS(Graph2 graph,int n){
    vector<int>top_list;
    int indegree[n]; //入度
    CountInDegree(graph,n,indegree);
    queue<int>queue;
    for(int i=0;i<n;i++){
        if(indegree[i]==0){
            queue.push(i);
        }
    }
    while(!queue.empty()){
        int u=queue.front();
        queue.pop();
        top_list.push_back(u);
        Edge* p=graph.vex[u].adj; //邻接节点入度减一
        while(p!=NULL){
            indegree[p->src]--;
            if(indegree[p->src]==0){
                queue.push(p->src);
            }
            p=p->next;
        }
    }
}

//DFS算法(顺序为先输出一条路,再看其他路)
void DFS_Sort(Graph2 graph,int v,bool visit[],vector<int>top_list){
    visit[v]=true;
    Edge* p=graph.vex[v].adj;
    while(p!=NULL){
        if(visit[p->src]==false){
            DFS_Sort(graph,p->src,visit,top_list);
        }
        p=p->next;
    }
    top_list.push_back(v);
}
```

```

void TopSort_DFS(Graph2 graph,int n,vector<int>top_list){
    bool visit[n];
    for(int i=0;i<n;i++){
        visit[i]=false;
    }
    for(int v=0;v<n;v++){
        if(visit[v]==false){
            DFS_Sort(graph,v,visit,top_list);
        }
    }
}

```

例题1 🤔

```

#include <iostream>
#include <vector>
#include <queue>
#include <cstring>
using namespace std;

int main() {
    int n, m;
    while (cin >> n >> m) { // 读取顶点数 n 和边数 m，支持多组数据
        vector<vector<int>> adj(n + 1); // 邻接表表示图，1-based 索引
        vector<int> indegree(n + 1, 0); // 存储每个节点的入度
        vector<int> result; // 存储拓扑排序结果

        // 读取边并建立图
        for (int i = 0; i < m; i++) {
            int d, u;
            cin >> d >> u; // 从 d 指向 u 的边
            adj[d].push_back(u); // d 的邻接点增加 u
            indegree[u]++; // u 的入度增加
        }

        queue<int> q; // 队列，用于执行拓扑排序
        for (int i = 1; i <= n; i++) { // 将所有入度为 0 的节点加入队列
            if (indegree[i] == 0) {
                q.push(i);
            }
        }

        int count = 0; // 记录处理过的节点数量
        bool unique = true; // 标志是否存在唯一的拓扑排序

        // 开始拓扑排序
        while (!q.empty()) {
            if (q.size() > 1) { // 如果队列中有多个节点，则拓扑排序不唯一
                unique = false;
            }
            int node = q.front(); // 取出队首节点
            q.pop();

```



```

        result.push_back(node); // 将节点加入结果中
        count++; // 计数已排序的节点数

        for (int next : adj[node]) { // 遍历当前节点的所有邻接点
            if (--indegree[next] == 0) { // 入度减 1, 如果变为 0, 则加入队列
                q.push(next);
            }
        }
    }

    // 判断拓扑排序结果
    if (count != n) { // 如果没有处理完所有节点, 则图中存在环
        cout << "0" << endl; // 图中有环, 无拓扑排序
    } else {
        if (unique) {
            cout << "1" << endl; // 图有唯一的拓扑排序
        } else {
            cout << "2" << endl; // 图有多个可能的拓扑排序
        }
    }
}
return 0;
}

```

八、最短路径 🐼

Dijkstra算法(边权重非负)* 🤨

时间复杂度 $O(E+V\log V)$

```

#include <iostream>
#include <vector>
#include <queue>
#include <limits>
using namespace std;

const int INF = INT_MAX; // 定义无穷大

// Dijkstra 核心函数
void Dijkstra(int start, int n, const vector<vector<pair<int, int>>>& graph,
vector<int>& dist) {
    dist.assign(n, INF); // 初始化所有点的距离为无穷大
    dist[start] = 0; // 起点距离自身为0
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.emplace(0, start); // 优先队列存储 (距离, 节点)

    while (!pq.empty()) {
        auto [d, u] = pq.top(); // 获取当前距离最小的节点
        pq.pop();
    }
}

```

```

        if (d > dist[u]) continue; // 如果队列中的距离已经过时，跳过

        // 遍历节点 u 的所有邻居
        for (const auto& [v, weight] : graph[u]) {
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.emplace(dist[v], v); // 更新优先队列
            }
        }
    }
}

int main() {
    int n, m, start;
    cin >> n >> m >> start; // 输入点数、边数、起点
    start--; // 转换为 0-based 索引

    vector<vector<pair<int, int>>> graph(n); // 邻接表表示图
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        u--, v--; // 转换为 0-based 索引
        graph[u].emplace_back(v, w);
        graph[v].emplace_back(u, w); // 若是无向图，需添加反向边
    }

    vector<int> dist; // 距离数组
    Dijkstra(start, n, graph, dist);

    // 输出结果
    for (int i = 0; i < n; ++i) {
        if (dist[i] == INF)
            cout << "INF ";
        else
            cout << dist[i] << " ";
    }
    cout << endl;

    return 0;
}

```

Bellman-Ford算法 🤔

时间复杂度 $O(V^3)$ [最坏情况]

```

#include <iostream>
#include <vector>
#include <climits>
using namespace std;

const int INF = INT_MAX; // 定义无穷大

// Bellman-Ford 核心函数

```

```

bool BellmanFord(int start, int n, vector<tuple<int, int, int>>& edges,
vector<int>& dist) {
    dist.assign(n, INF); // 初始化所有点的距离为无穷大
    dist[start] = 0;      // 起点距离自身为0

    // 松弛操作，最多进行 n-1 次
    for (int i = 0; i < n - 1; ++i) {
        for (const auto& [u, v, weight] : edges) {
            if (dist[u] != INF && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    // 检测负权环：如果还能松弛，说明存在负权环
    for (const auto& [u, v, weight] : edges) {
        if (dist[u] != INF && dist[u] + weight < dist[v]) {
            return false; // 发现负权环
        }
    }

    return true; // 没有负权环
}

int main() {
    int n, m, start;
    cin >> n >> m >> start; // 输入点数、边数、起点
    start--;                // 转换为 0-based 索引

    vector<tuple<int, int, int>> edges; // 存储边的三元组 (u, v, weight)
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        u--, v--; // 转换为 0-based 索引
        edges.emplace_back(u, v, w);
    }

    vector<int> dist; // 距离数组
    if (BellmanFord(start, n, edges, dist)) {
        // 输出结果
        for (int i = 0; i < n; ++i) {
            if (dist[i] == INF)
                cout << "INF ";
            else
                cout << dist[i] << " ";
        }
        cout << endl;
    } else {
        cout << "Negative weight cycle detected." << endl;
    }

    return 0;
}

```

时间复杂度 $O(V \cdot E)$ [最坏情况]

空间复杂度 $O(V)$

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

const int INF = INT_MAX; // 定义无穷大

// SPFA 核心函数
void SPFA(int start, int n, vector<vector<pair<int, int>>>& graph) {
    vector<int> dist(n, INF); // 距离数组, 初始为无穷大
    vector<bool> inQueue(n, false); // 标记节点是否在队列中
    queue<int> q; // 辅助队列

    dist[start] = 0; // 起点到自身的距离为0
    q.push(start);
    inQueue[start] = true;

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inQueue[u] = false;

        // 遍历 u 的所有邻接边
        for (const auto& [v, weight] : graph[u]) {
            if (dist[u] + weight < dist[v]) { // 松弛操作
                dist[v] = dist[u] + weight;
                if (!inQueue[v]) { // 如果 v 不在队列中, 加入队列
                    q.push(v);
                    inQueue[v] = true;
                }
            }
        }
    }
}

// 输出结果
for (int i = 0; i < n; ++i) {
    if (dist[i] == INF)
        cout << "INF ";
    else
        cout << dist[i] << " ";
}
cout << endl;
}

int main() {
    int n, m, start;
    cin >> n >> m >> start; // 输入点数、边数、起点
    start--; // 转换为0-based

    // 构建邻接表
```

```

vector<vector<pair<int, int>>> graph(n);
for (int i = 0; i < m; ++i) {
    int u, v, w;
    cin >> u >> v >> w;
    u--, v--; // 转换为0-based
    graph[u].emplace_back(v, w);
}

// 调用 SPFA
SPFA(start, n, graph);

return 0;
}

```

九、最小生成树 🤖

求解权值最小的生成树

Prim算法(扩点) 🤔

与Dijkstra算法完全相同，只有对临界节点的判断不同

时间复杂度 $O(V+E\log E)$

空间复杂度 $O(V+E)$

prim算法实现^{^3}

```

#include <iostream>
#include <vector>
#include <queue>
#include <tuple>
#include <algorithm>
using namespace std;

// 定义边的结构
struct Edge {
    int to, weight;
    Edge(int t, int w) : to(t), weight(w) {}
};

// 最小生成树 Prim 实现
int main() {
    int n, m, start;
    cin >> n >> m >> start; // 输入村庄数、边数、起始编号
    start--; // 转换为 0-based 编号

    // 邻接表存储图

```

```

vector<vector<Edge>> graph(n);
for (int i = 0; i < m; ++i) {
    int u, v, w;
    cin >> u >> v >> w;
    u--; v--; // 转换为 0-based 编号
    graph[u].emplace_back(v, w);
    graph[v].emplace_back(u, w); // 因为是无向图
}

// 最小生成树的边
vector<tuple<int, int, int>> mstEdges;

// 优先队列存储 {边权重, 当前节点, 前驱节点}
priority_queue<tuple<int, int, int>, vector<tuple<int, int, int>>, greater<>>
pq;

vector<bool> inMST(n, false); // 标记节点是否已加入 MST

// 初始化, 从起始节点开始
inMST[start] = true;
for (const auto& edge : graph[start]) {
    pq.emplace(edge.weight, edge.to, start);
}

// 构造 MST
while (!pq.empty()) {
    auto [weight, to, from] = pq.top(); //注意此处赋值, 是中括号
    pq.pop();

    // 如果目标节点已在 MST 中, 跳过
    if (inMST[to]) continue;

    // 加入 MST
    inMST[to] = true;
    mstEdges.emplace_back(from, to, weight); //注意此处不需要大括号

    // 将目标节点的所有邻接边加入队列
    for (const auto& edge : graph[to]) {
        if (!inMST[edge.to]) {
            pq.emplace(edge.weight, edge.to, to);
        }
    }

    // 如果 MST 边数等于 n - 1, 则完成
    if (mstEdges.size() == n - 1) break;
}

// 输出 MST 边
for (auto& [from, to, weight] : mstEdges) { //注意这里是中括号
    // 转换回 1-based 编号并输出
    if (from > to) swap(from, to);
    cout << from + 1 << ", " << to + 1 << ", " << weight << endl;
}

return 0;
}

```

Kruskal算法(扩边)😓

并查集

类似于集合合并

```
// 并查集的结构体
class UnionFind {
public:
    vector<int> parent, rank;

    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    // 查找父节点并进行路径压缩
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    // 合并两个集合
    void union_sets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            // 按照秩 (rank) 优化合并
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};
```

Kruskal算法实现^3

实现的很简单，不要想太多，而且不要将思维局限于创造图结构，图也是存在顺序表中的

```

#include <bits/stdc++.h>
using namespace std;

// 边的结构体
struct Edge {
    int u, v, weight;
    Edge(int a, int b, int w) : u(a), v(b), weight(w) {}

    // 按照权重从小到大排序
    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};

// 并查集的结构体
class UnionFind {
public:
    vector<int> parent, rank;

    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    // 查找父节点并进行路径压缩
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    // 合并两个集合
    void union_sets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            // 按照秩 (rank) 优化合并
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};

void kruskal(int n, vector<Edge>& edges) {
    UnionFind uf(n);
    vector<Edge> mst;

```



```

// 对边按权重升序排序
sort(edges.begin(), edges.end());

for (const auto& edge : edges) {
    int u = edge.u, v = edge.v, w = edge.weight;
    if (uf.find(u) != uf.find(v)) {
        uf.union_sets(u, v);
        mst.push_back(edge);
    }
}

// 输出最小生成树的边
for (const auto& edge : mst) {
    if (edge.u < edge.v) {
        cout << (edge.u+1) << "," << (edge.v+1) << "," << edge.weight << endl;
    } else {
        cout << (edge.v+1) << "," << (edge.u+1) << "," << edge.weight << endl;
    }
}

int main() {
    int n, m;
    cin >> n >> m;

    vector<Edge> edges;
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        edges.push_back(Edge(u - 1, v - 1, w)); // 转换为从0开始的索引
    }

    kruskal(n, edges);

    return 0;
}

```

排序优化(高度复杂) 🤔

```

struct Edge{ //相邻点链表
    int src; //表示该节点位置
    int weight; //节点间线的权重
    Edge *next; //指向下一节点
    Edge(int x,int y):src(x),weight(y),next(NULL){}
};

struct Vex{
    int data;
    Edge *adj=NULL; //指向相邻点链表
};

```

```

struct Graph2{
    Vex vex[maxnum];
    int Vnum2,Enum2;
};

struct edge{
    int u;
    int v;
    int weight;
    edge(int x,int y,int z):u(x),v(y),weight(z){}
};

void SiftUp(vector<edge> h,int i){ //i是起始位置
    edge elem=h[i];
    while(i>1 && elem.weight<h[i/2].weight){
        h[i]=h[i/2];
        i=i/2;
    }
    h[i]=elem;
}

void SiftDown(vector<edge> h,int i){
    int last=h.size();
    edge elem=h[i];
    int child=2*i;
    while(true){
        //找到更小的子节点
        if(child<last && h[child].weight>h[child+1].weight){
            child+=1;
        }
        else if(child>last){
            break;
        }
        //下调
        if(h[child].weight<elem.weight){
            h[i]=h[child];
            i=child;
        }
        else{
            break;
        }
    }
    h[child]=elem;
}

void insert(vector<edge>h,edge x){
    h.push_back(x);
    SiftUp(h,h.size());
}

edge DeleteMin(vector<edge>h){
    edge min=h[0];
    h[0]=h[h.size()-1];
    SiftDown(h,0);
    return min;
}

```

```

//查找元素所在集合
int Find(int parent[],int a){
    int root=a;
    while(parent[root]!=root){
        root=parent[root];
    }
    //路径压缩，将a放在根下
    while(parent[a]!=root){
        int temp=parent[a];
        parent[a]=root;
        a=temp;
    }
    return root;
}

//合并两个元素所在集合
void Union(int parent[],int a,int b){
    int root_a=Find(parent,a);
    int root_b=Find(parent,b);
    if(root_a!=root_b){
        parent[root_b]=root_a;
    }
}

int Kruskal(Graph2 graph){
    vector<edge>v; //堆
    int parent[graph.Vnum2];
    for(int i=0;i<graph.Vnum2;i++){
        parent[i]=i;
        Edge* p=graph.vex[i].adj;
        while(p!=NULL){
            edge e(i,p->src,p->weight);
            insert(v,e);
            p=p->next;
        }
    }

    int total_weight=0;
    while(!v.empty()){
        edge e=DeleteMin(v);
        if(Find(parent,e.u)!=Find(parent,e.v)){ //保证不会形成环
            total_weight+=e.weight;
            Union(parent,e.u,e.v);
        }
    }
    return total_weight;
}

```

十、图例题^2 🤖

例题一

求解图中所有点可到达点的总和

bitset知识点

```
#include <iostream>
#include <bitset>
#include <vector>
using namespace std;

const int MAX_N = 2001;

char ch[MAX_N];
bitset<MAX_N> a[MAX_N]; // 使用 bitset 存储节点的连接关系
int ans;

int main() {
    int n;
    cin >> n;

    // 输入图的连接情况
    for (int i = 1; i <= n; i++) {
        cin >> (ch + 1); // 读取每一行的图的连接关系
        for (int j = 1; j <= n; j++) {
            if (ch[j] == '1') {
                a[i][j] = 1; // 如果有边, 更新 bitset
            }
        }
        a[i][i] = 1; // 每个节点可以到达自己
    }

    // 使用 Floyd-Warshall 算法进行传递闭包计算
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (a[j][i]) {
                a[j] |= a[i]; // 如果 j 能到 i, 就让 j 到达 i 能到的所有节点
                // 上边这一步很重要, 每行列的位或操作传递了连通性;
            }
        }
    }

    // 统计每个节点能到达的节点数
    for (int i = 1; i <= n; i++) {
        ans += a[i].count(); // 统计节点 i 能到达的节点数
    }

    cout << ans << endl; // 输出结果

    return 0;
}
```

```

class Solution {
public:
    int findCircleNum(vector<vector<int>>& isConnected) {
        int n = isConnected.size();
        vector<int> root(n);

        for (int i = 0; i < n; ++i) {
            root[i] = i;
        }

        auto find = [&](int x) {
            while (x != root[x]) {
                root[x] = root[root[x]];
                x = root[x];
            }
            return x;
        };

        auto unite = [&](int x, int y) {
            int rootX = find(x);
            int rootY = find(y);
            if (rootX != rootY) {
                root[rootY] = rootX;
            }
        };

        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (isConnected[i][j] == 1) {
                    unite(i, j);
                }
            }
        }

        int count = 0;
        for (int i = 0; i < n; ++i) {
            if (find(i) == i) { //很好的想法
                ++count;
            }
        }

        return count;
    }
};

```

例题三* 😞

力扣原题 (困难)

真难理解啊!!! 🤔

想成将分散的点从小到大连起来形成“好路径”，而不是从最大开始查找删除

```
class Solution {
public:
    int numberOfGoodPaths(vector<int> &vals, vector<vector<int>> &edges) {
        int n = vals.size();
        vector<vector<int>> g(n);
        for (auto &e : edges) {
            int x = e[0], y = e[1];
            g[x].push_back(y);
            g[y].push_back(x); // 建图
        }

        // 并查集模板
        // size[x] 表示节点值等于 vals[x] 的节点个数，
        // 如果按照节点值从小到大合并，size[x] 也是连通块内的等于最大节点值的节点个数
        int id[n], fa[n], size[n]; // id 后面排序用
        iota(id, id + n, 0);
        iota(fa, fa + n, 0);
        fill(size, size + n, 1);
        function<int(int)> find = [&](int x) -> int { return fa[x] == x ? x : fa[x]
= find(fa[x]); };

        int ans = n; // 单个节点的好路径
        sort(id, id + n, [&](int i, int j) { return vals[i] < vals[j]; });
        for (int x : id) {
            int vx = vals[x], fx = find(x);
            for (int y : g[x]) { //对周围节点的情况分析
                y = find(y);
                if (y == fx || vals[y] > vx)
                    continue; // 只考虑最大节点值不超过 vx 的连通块
                if (vals[y] == vx) { // 可以构成好路径
                    ans += size[fx] * size[y]; // 乘法原理,两点之间只有唯一的路径，故
有乘法原理

                    size[fx] += size[y]; // 统计连通块内节点值等于 vx 的节点个数
                }
                fa[y] = fx; // 把小的节点值合并到大的节点值上，相当于是周围节点的值小于x
时的行为
            }
        }
        return ans;
    }
};
```

例题四* 🤔

码蹄集周赛(钻石)

修改题目，即认识的人均不在一桌

```

//使用BFS二分图
#include <bits/stdc++.h>
using namespace std;

// BFS 检查是否为二分图，并计算两组人数
bool isBipartite(const vector<vector<int>>& graph, vector<int>& colors, int start,
int& count1, int& count2) {
    queue<int> q;
    q.push(start);
    colors[start] = 1; // 1 表示第一个集合
    count1++;

    while (!q.empty()) {
        int node = q.front();
        q.pop();

        for (int neighbor = 0; neighbor < graph.size(); ++neighbor) {
            if (graph[node][neighbor]) { // 如果有边
                if (colors[neighbor] == 0) { // 如果未访问
                    colors[neighbor] = -colors[node]; // 赋予相反颜色
                    if (colors[neighbor] == 1) count1++;
                    else count2++;
                    q.push(neighbor);
                } else if (colors[neighbor] == colors[node]) {
                    return false; // 如果颜色相同，则不是二分图
                }
            }
        }
    }
    return true;
}

int main() {
    int n;
    cin >> n;

    vector<vector<int>> graph(n, vector<int>(n));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> graph[i][j];
        }
    }

    vector<int> colors(n, 0); // 0 表示未访问, 1 表示集合 A, -1 表示集合 B
    bool isBipartiteGraph = true;
    int maxTableSize = 0;

    for (int i = 0; i < n && isBipartiteGraph; ++i) {
        if (colors[i] == 0) { // 如果当前节点未被访问
            int count1 = 0, count2 = 0;
            if (!isBipartite(graph, colors, i, count1, count2)) {
                isBipartiteGraph = false;
                break;
            }
            maxTableSize = max(maxTableSize, max(count1, count2));
        }
    }
}

```

```
}

if (!isBipartiteGraph) {
    cout << "No" << endl;
} else {
    cout << "Yes" << endl;
    cout << maxTableSize << endl;
}

return 0;
}
```

第二章、算法 🌀

动态规划，贪心算法，回溯算法，二分法,分治法，归并排序，DFS，BFS，递归，快速选择，记忆化搜索等等

[不错的网站 🤖](#)

第一节、贪心算法 😞

局部最优解求全局最优解

一、序列问题

摆动序列问题

[力扣原题](#)

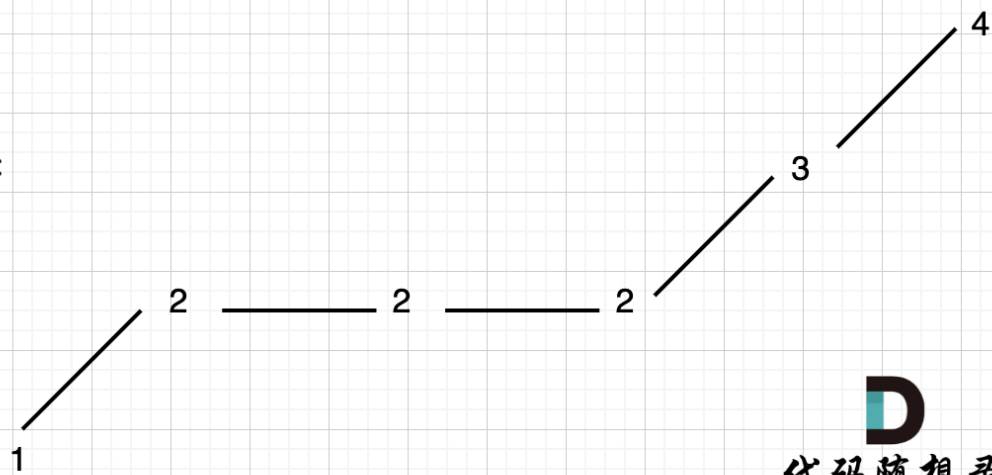
[原题讲解在这里](#)

局部最优：删除单调坡度上的节点（不包括单调坡度两端的节点），那么这个坡度就可以有两个局部峰值。

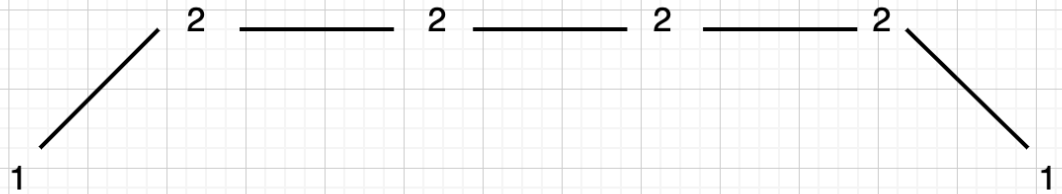
整体最优：整个序列有最多的局部峰值，从而达到最长摆动序列。

==要考虑上下坡有平坡，单调坡有平坡，首尾极值的情况==

单调中间有平坡:



上下中间有平坡:



实现代码:

```
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
        if (nums.size() <= 1) return nums.size();
        int curDiff = 0; // 当前一对差值
        int preDiff = 0; // 前一对差值
        int result = 1; // 记录峰值个数，序列默认序列最右边有一个峰值
        for (int i = 0; i < nums.size() - 1; i++) {
            curDiff = nums[i + 1] - nums[i];
            // 出现峰值
            if ((preDiff <= 0 && curDiff > 0) || (preDiff >= 0 && curDiff < 0)) {
                result++;
                preDiff = curDiff; // 注意这里，只在摆动变化的时候更新prediff,防止出现
                // 单调坡有平坡的情况
            }
        }
        return result;
    }
};
```

二、递增数问题

[力扣原题](#)

[原题讲解在这里](#)

贪心思想：一旦出现 $\text{strNum}[i - 1] > \text{strNum}[i]$ 的情况（非单调递增），首先想让 $\text{strNum}[i - 1]--$ ，然后 $\text{strNum}[i]$ 给为9

==从后向前遍历，防止 $\text{strNum}[i - 1]--$ 影响之前的调整==

实现代码：

```
class Solution {
public:
    int monotoneIncreasingDigits(int N) {
        string strNum = to_string(N);
        // flag用来标记赋值9从哪里开始
        // 设置为这个默认值，为了防止第二个for循环在flag没有被赋值的情况下执行
        int flag = strNum.size();
        for (int i = strNum.size() - 1; i > 0; i--) {
            if (strNum[i - 1] > strNum[i] ) {
                flag = i;
                strNum[i - 1]--;
            }
        }
        for (int i = flag; i < strNum.size(); i++) {
            strNum[i] = '9'; // 将赋值放在后面的原因是防止i-1位赋为9了，但i位却不是9的
        }
        return stoi(strNum);
    }
};
```

情况，eg:100

取反最大和

力扣原题

[原题讲解在这里](#)

局部最优：让绝对值大的负数变为正数，当前数值达到最大。

整体最优：整个数组和达到最大。

==注意是按照绝对值排序==

实现代码：

```
class Solution {
public:
    static bool cmp(int a, int b) {
        return abs(a) > abs(b);
    }
    int largestSumAfterKNegations(vector<int>& A, int K) {
```

```

        sort(A.begin(), A.end(), cmp);           // 第一步
        for (int i = 0; i < A.size(); i++) { // 第二步
            if (A[i] < 0 && K > 0) {
                A[i] *= -1;
                K--;
            }
        }
        if (K % 2 == 1) A[A.size() - 1] *= -1; // 第三步
        int result = 0;
        for (int a : A) result += a;           // 第四步
        return result;
    }
};

```

三、股票问题

买卖股票问题

[力扣原题](#)

[原题讲解在这里](#)

局部最优：收集每天的正利润。

全局最优：求得最大利润。

==将价格转换为每天的利润，累加每次的正利润==

实现代码：

```

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int result = 0;
        for (int i = 1; i < prices.size(); i++) {
            result += max(prices[i] - prices[i - 1], 0);
        }
        return result;
    }
};

```

四、两维度问题

遇到两个维度权衡的时候，一定要先确定一个维度，再确定另一个维度。

分发糖果问题

力扣原题

原题讲解在这里

两个维度：左侧大于右侧，右侧大于左侧

==从左右两方向各遍历一遍，保证局部成立，实现整体成立==

实现代码：

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        vector<int> candyVec(ratings.size(), 1);
        // 从前向后
        for (int i = 1; i < ratings.size(); i++) {
            if (ratings[i] > ratings[i - 1]) candyVec[i] = candyVec[i - 1] + 1;
        }
        // 从后向前
        for (int i = ratings.size() - 2; i >= 0; i--) {
            if (ratings[i] > ratings[i + 1]) {
                candyVec[i] = max(candyVec[i], candyVec[i + 1] + 1);
            }
        }
        // 统计结果
        int result = 0;
        for (int i = 0; i < candyVec.size(); i++) result += candyVec[i];
        return result;
    }
};
```

身高排列问题

力扣原题

原题讲解在这里

两个维度：身高和人数

==先确定身高从大到小排后，由前向后对k的要求进行插入，后插入节点不影响前插入节点==

实现代码：

```
class Solution {
public:
    static bool cmp(const vector<int>& a, const vector<int>& b) {
```

```

        if (a[0] == b[0]) return a[1] < b[1];
        return a[0] > b[0];
    }
    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
        sort (people.begin(), people.end(), cmp);
        // int n=people.size(); vector<vector<int>> que(n);
        vector<vector<int>> que;
        for (int i = 0; i < people.size(); i++) {
            int position = people[i][1];
            que.insert(que.begin() + position, people[i]); // 注意是插入
        }
        return que;
    }
};

```

五、区间问题

跳跃问题(一)

[力扣原题](#)

[原题讲解在这里](#)

局部最优解：每个位置取最大跳跃步数（取最大覆盖范围）。

整体最优解：最后得到整体最大覆盖范围，看是否能到终点。

==此题不需要路径，将此题转化为覆盖范围==

实现代码：

```

class Solution {
public:
    bool canJump(vector<int>& nums) {
        int cover = 0;
        if (nums.size() == 1) return true; // 只有一个元素，就是能达到
        for (int i = 0; i <= cover; i++) { // 注意这里是小于等于cover
            cover = max(i + nums[i], cover);
            if (cover >= nums.size() - 1) return true; // 说明可以覆盖到终点了
        }
        return false;
    }
};

```

跳跃问题(二)

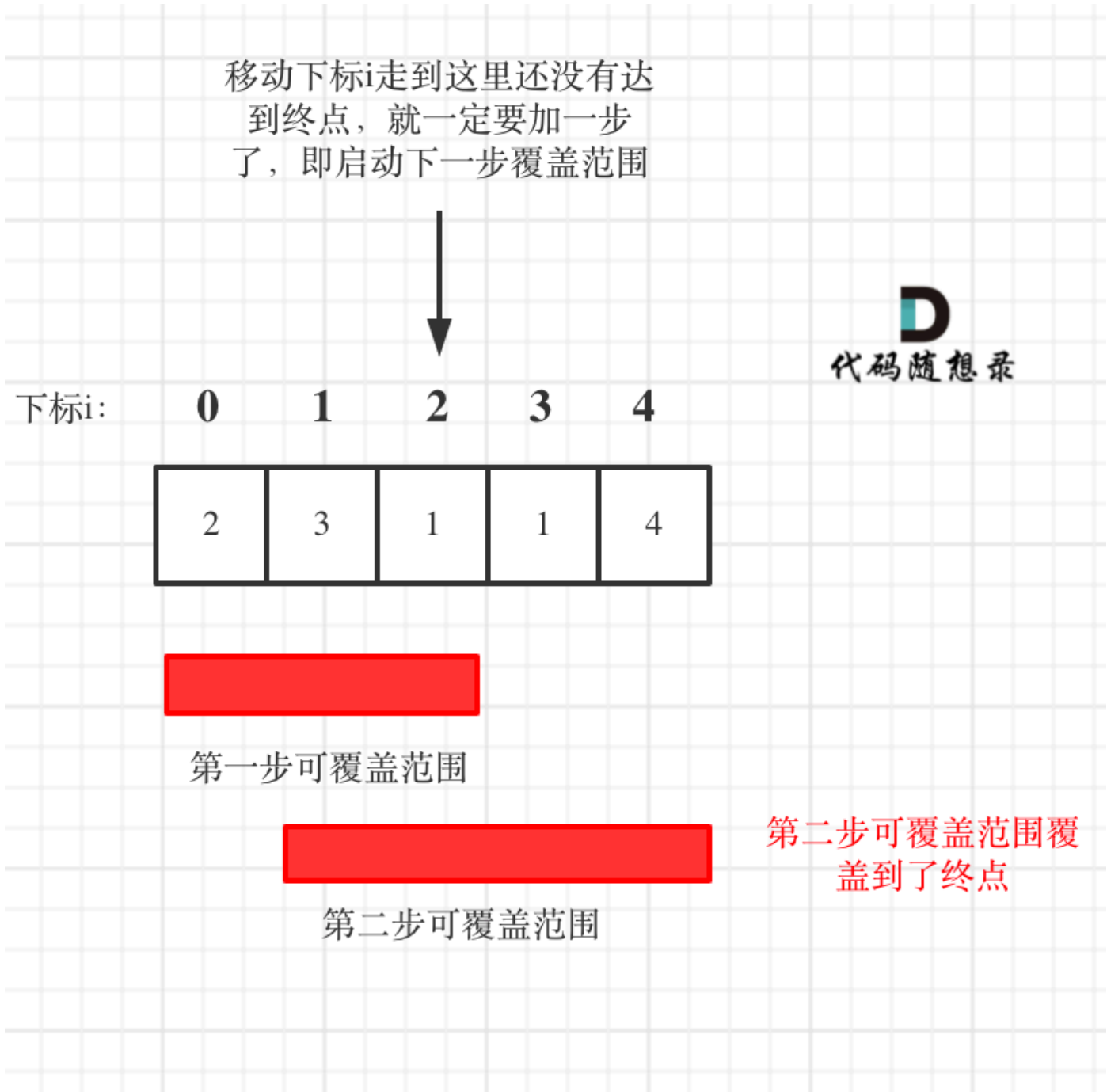
[力扣原题](#)

[原题讲解在这里](#)

贪心的思路，局部最优：当前可移动距离尽可能多走，如果还没到终点，步数再加一。

整体最优：一步尽可能多走，从而达到最少步数。

==和问题已大致相同，但需要记录每次最远距离的刷新==



实现代码:

```
class Solution {
public:
    int jump(vector<int>& nums) {
        if (nums.size() == 1) return 0;
        int curDistance = 0;    // 当前覆盖最远距离下标
        int ans = 0;            // 记录走的最大步数
    }
};
```

```

int nextDistance = 0;    // 下一步覆盖最远距离下标
for (int i = 0; i < nums.size(); i++) {
    nextDistance = max(nums[i] + i, nextDistance); // 更新下一步覆盖最远距离
下标
    if (i == curDistance) { // 遇到当前覆盖最远距离下
标，在i<cur的范围内均算第一步
        ans++; // 需要走下一步
        curDistance = nextDistance; // 更新当前覆盖最远距离下标
        (相当于加油了)
        if (nextDistance >= nums.size() - 1) break; // 当前覆盖最远距到达集
合终点，不用做ans++操作了，直接结束
    }
}
return ans;
};

```

箭扎气球问题

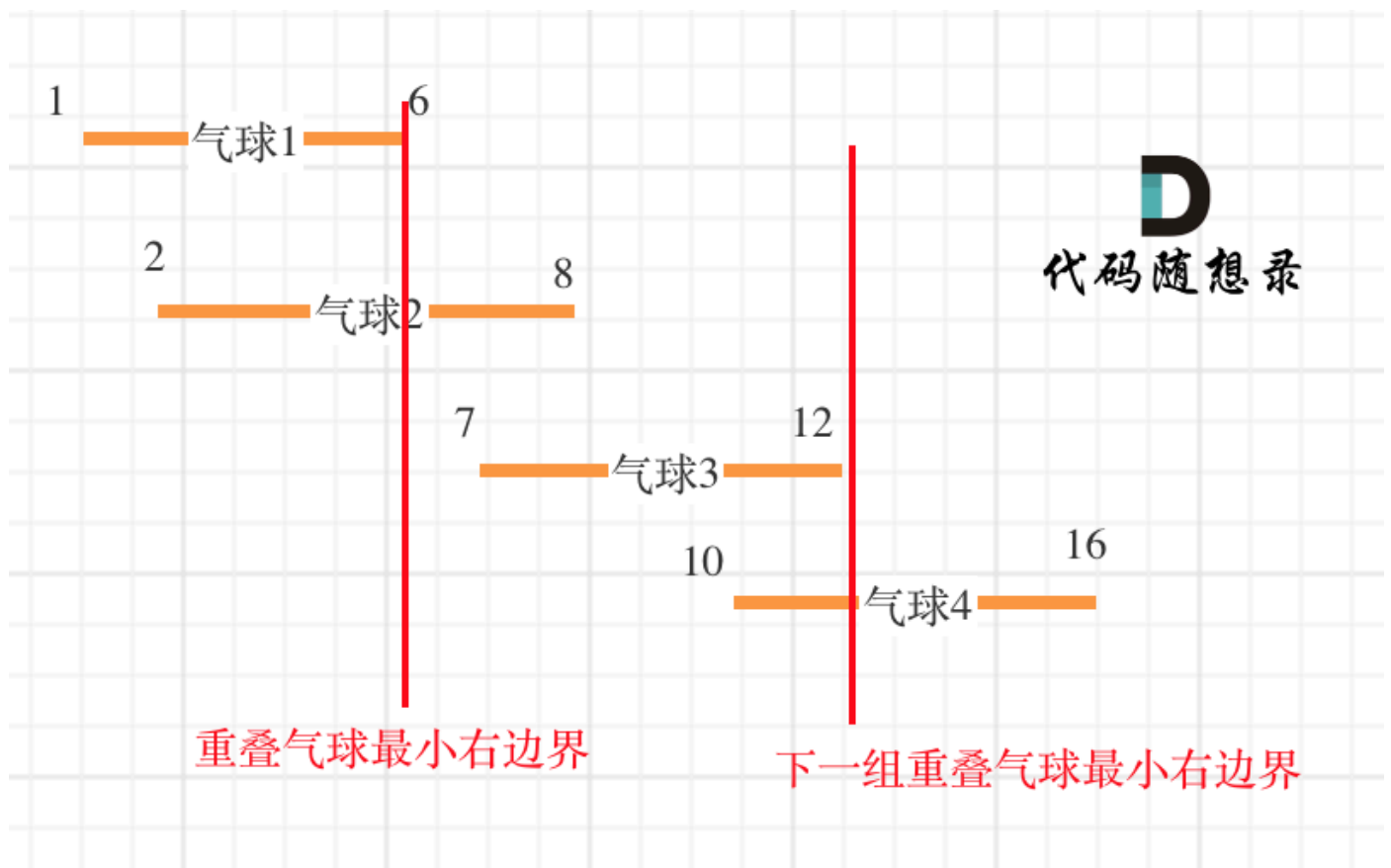
力扣原题

原题讲解在这里

局部最优解：寻找最小的气球结束点。

整体最优解：使用数量最少的箭。

==通过不断找到每一阶段最早出现终点的气球来实现==



实现代码:

```
class Solution {
public:
    int findMinArrowShots(vector<vector<int>>& points) {
        if (points.empty()) {
            return 0;
        }
        sort(points.begin(), points.end(), [](const vector<int>& u, const
vector<int>& v) {
            return u[1] < v[1];
        });
        int pos = points[0][1];
        int ans = 1;
        for (const vector<int>& balloon: points) {
            if (balloon[0] > pos) {
                pos = balloon[1];
                ++ans;
            }
        }
        return ans;
    }
};
```

无重叠区间问题

力扣原题

[原题讲解在这里](#)

==与扎气球问题相同==

实现代码:

```
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if (intervals.empty()) {
            return 0;
        }

        sort(intervals.begin(), intervals.end(), [](const auto& u, const auto& v) {
            return u[1] < v[1];
        });

        int n = intervals.size();
        int ans = 1;
        int right = intervals[0][1];
        for (int i = 1; i < n; ++i) {
            if (intervals[i][0] >= right) {
```



```

        ++ans;
        right = intervals[i][1];
    }
}
return n - ans;
}
};

```

字母区间问题

力扣原题

原题讲解在这里

==与上述问题思路相同，但也有点小差别==

实现代码:

```

class Solution {
public:
    vector<int> partitionLabels(string S) {
        int hash[27] = {0}; // i为字符，hash[i]为字符出现的最后位置
        for (int i = 0; i < S.size(); i++) { // 统计每一个字符最后出现的位置
            hash[S[i] - 'a'] = i;
        }
        vector<int> result;
        int left = 0;
        int right = 0;
        for (int i = 0; i < S.size(); i++) {
            right = max(right, hash[S[i] - 'a']); // 找到字符出现的最远边界
            if (i == right) { // 这一步和跳跃问题很像
                result.push_back(right - left + 1);
                left = i + 1;
            }
        }
        return result;
    }
};

```

六、其他

最大序列和问题

力扣原题

原题讲解在这里

局部最优：当前“连续和”为负数的时候立刻放弃，从下一个元素重新计算“连续和”，因为负数加上下一个元素“连续和”只会越来越小。

全局最优：选取最大“连续和”

==当连续和为负数时直接重置为零，重新设立起点==

实现代码：

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int result = INT32_MIN; //最小负数
        int count = 0;
        for (int i = 0; i < nums.size(); i++) {
            count += nums[i];
            if (count > result) { // 取区间累计的最大值（相当于不断确定最大子序终止位置）
                result = count;
            }
            if (count <= 0) count = 0; // 相当于重置最大子序起始位置，因为遇到负数一定是拉低总和
        }
        return result;
    }
};
```

加油站问题

力扣原题

[原题讲解在这里](#)

局部最优：当前累加rest[i]的和curSum一旦小于0，起始位置至少要是i+1，因为从i之前开始一定不行。

全局最优：找到可以跑一圈的起始位置。

==注意使用累加的剩余，而不是去找单独的剩余==

实现代码：

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int curSum = 0;
        int totalSum = 0;
```

```

int start = 0;
for (int i = 0; i < gas.size(); i++) {
    curSum += gas[i] - cost[i];
    totalSum += gas[i] - cost[i];
    if (curSum < 0) { // 当前累加rest[i]和 curSum一旦小于0
        start = i + 1; // 起始位置更新为i+1
        curSum = 0; // curSum从0开始
    }
}
if (totalSum < 0) return -1; // 说明怎么走都不可能跑一圈了
return start;
}
};

```

监控二叉树问题

力扣原题

原题讲解在这里

局部最优：让叶子节点的父节点安摄像头，所用摄像头最少。

整体最优：全部摄像头数量所用最少！

==从下向上看，叶节点省去的数量更多==

实现代码:

```

class Solution {
private:
    int result;
    int traversal(TreeNode* cur) {

        // 空节点，该节点有覆盖
        if (cur == NULL) return 2; // 判断叶节点(递归停止)

        // 后序遍历保证回溯过程从下向上推导
        int left = traversal(cur->left); // 左
        int right = traversal(cur->right); // 右

        // 情况1
        // 左右节点都有覆盖
        // left == 2 && right == 2 左右节点都有覆盖
        if (left == 2 && right == 2) return 0;

        // 情况2
        // 左右节点至少有一个无覆盖
        // left == 0 && right == 0 左右节点无覆盖
        // left == 1 && right == 0 左节点有摄像头，右节点无覆盖
        // left == 0 && right == 1 左节点有无覆盖，右节点摄像头
        // left == 0 && right == 2 左节点无覆盖，右节点覆盖
    }
}

```

```

// left == 2 && right == 0 左节点覆盖，右节点无覆盖
if (left == 0 || right == 0) {
    result++;
    return 1;
}

// 情况3
// 左右节点至少有一个有摄像头
// left == 1 && right == 2 左节点有摄像头，右节点有覆盖
// left == 2 && right == 1 左节点有覆盖，右节点有摄像头
// left == 1 && right == 1 左右节点都有摄像头
// 其他情况前段代码均已覆盖
if (left == 1 || right == 1) return 2;

// 这个 return -1 逻辑不会走到这里。
return -1;
}

public:
int minCameraCover(TreeNode* root) {
    result = 0;
    // 情况4
    // root 无覆盖，左右节点有覆盖
    if (traversal(root) == 0) {
        result++;
    }
    return result;
}
};

```

第二节、回溯算法 🤔

模板:

```

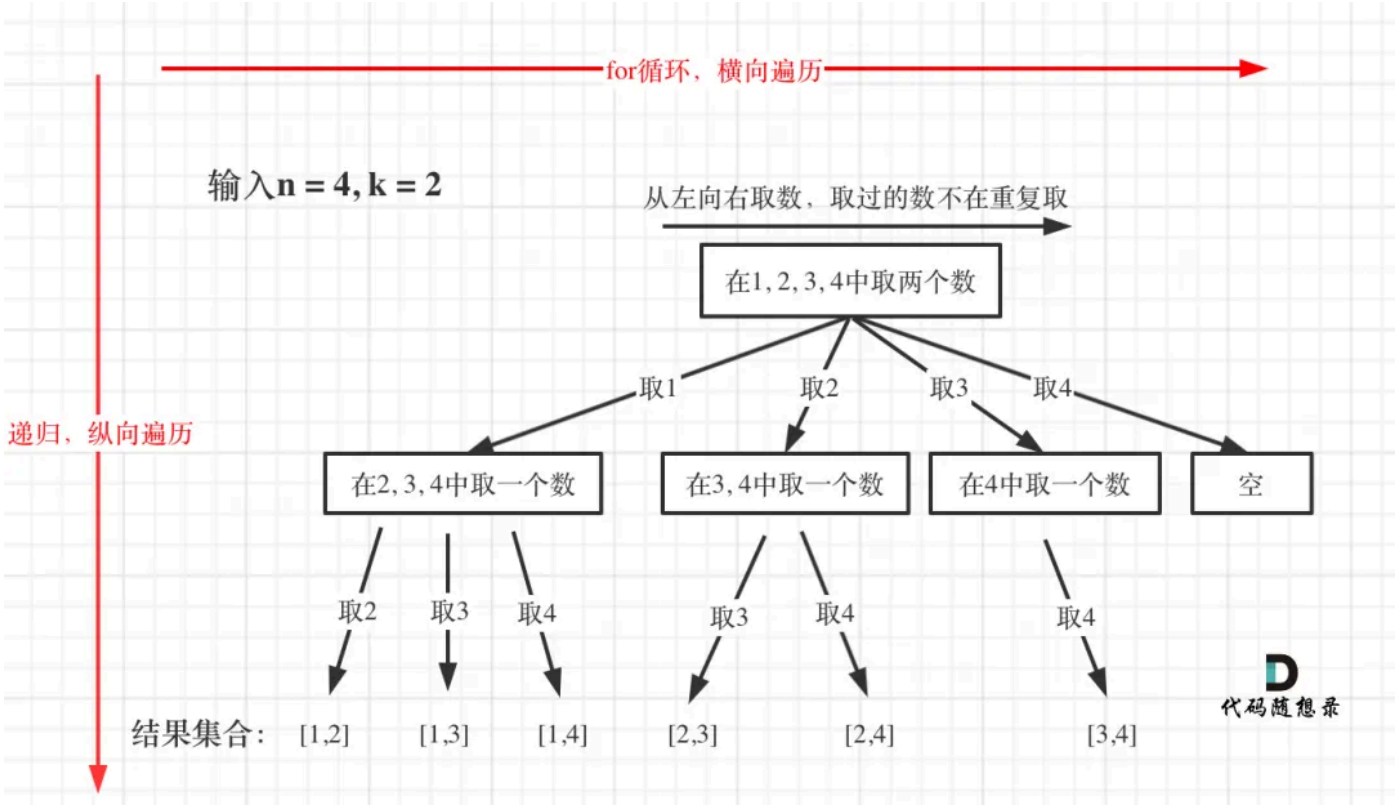
void backtracking(参数) {
    if (终止条件) {
        存放结果;
        return;
    }

    for (选择: 本层集合中元素 (树中节点孩子的数量就是集合的大小)) {
        处理节点;
        backtracking(路径, 选择列表); // 递归
        回溯, 撤销处理结果
    }
}

```

一、组合问题

树形结构表示:

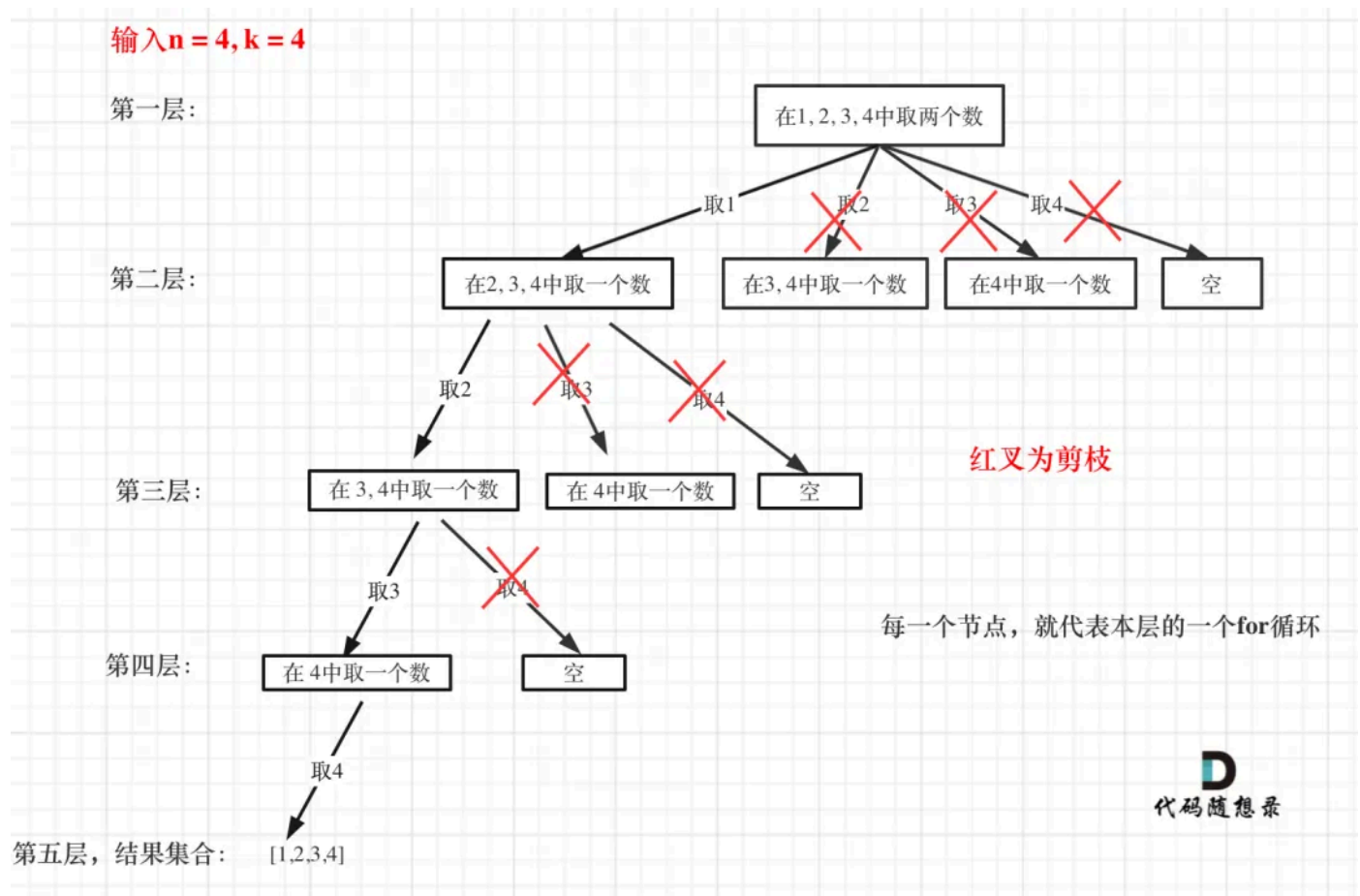


实现代码:

```
class Solution {
private:
    vector<vector<int>> result; // 存放符合条件结果的集合
    vector<int> path; // 用来存放符合条件结果
    void backtracking(int n, int k, int startIndex) { //startIndex确保每次递归时for
        循环开始的位置
        if (path.size() == k) {
            result.push_back(path);
            return;
        }
        for (int i = startIndex; i <= n; i++) {
            path.push_back(i); // 处理节点
            backtracking(n, k, i + 1); // 递归
            path.pop_back(); // 回溯, 撤销处理的节点
        }
    }
public:
    vector<vector<int>> combine(int n, int k) {
        result.clear(); // 清空序列, 可以不写
        path.clear(); // 清空序列, 可以不写
        backtracking(n, k, 1); //注意这里最开始startIndex=1, 不是0
        return result;
    }
};
```

```
}  
};
```

剪枝优化:



优化后代码:

```
class Solution {  
private:  
    vector<vector<int>> result;  
    vector<int> path;  
    void backtracking(int n, int k, int startIndex) {  
        if (path.size() == k) {  
            result.push_back(path);  
            return;  
        }  
        for (int i = startIndex; i <= n - (k - path.size()) + 1; i++) { // 优化的地方, 用k来限制起始位置的范围  
            path.push_back(i); // 处理节点  
            backtracking(n, k, i + 1);  
            path.pop_back(); // 回溯, 撤销处理的节点  
        }  
    }  
public:  
  
    vector<vector<int>> combine(int n, int k) {  
        backtracking(n, k, 1);  
        return result;  
    }  
};
```

```
}  
};
```

组合总和(一)

力扣原题

==和上述组合问题的不同在于总和n限制终止条件，k决定递归次数，for循环次数已定==

实现代码：

```
class Solution {  
private:  
    vector<vector<int>> result; // 存放结果集  
    vector<int> path; // 符合条件的结果  
    // targetSum: 目标和，也就是题目中的n。  
    // k: 题目中要求k个数的集合。  
    // sum: 已经收集的元素的总和，也就是path里元素的总和。  
    // startIndex: 下一层for循环搜索的起始位置。  
    void backtracking(int targetSum, int k, int sum, int startIndex) {  
        if (path.size() == k) {  
            if (sum == targetSum) result.push_back(path);  
            return; // 如果path.size() == k 但sum != targetSum 直接返回  
        }  
        for (int i = startIndex; i <= 9; i++) { //i也可调整为i<10-path.size(),但九  
            // 的数量较少，不该也不会影响太多效率  
            sum += i; // 处理  
            path.push_back(i); // 处理  
            backtracking(targetSum, k, sum, i + 1); // 注意i+1调整startIndex  
            sum -= i; // 回溯  
            path.pop_back(); // 回溯  
        }  
    }  
  
public:  
    vector<vector<int>> combinationSum3(int k, int n) {  
        result.clear(); // 可以不加  
        path.clear(); // 可以不加  
        backtracking(n, k, 0, 1);  
        return result;  
    }  
};
```

组合总和(二)

力扣原题

==与上题的区别在于不限制数量，也不限制每个元素使用个数==

实现代码:

```
class Solution {
private:
    vector<vector<int>> result;
    vector<int> path;
    void backtracking(vector<int>& candidates, int target, int sum, int startIndex)
    {
        if (sum > target) {
            return;
        }
        if (sum == target) {
            result.push_back(path);
            return;
        }

        // 如果 sum + candidates[i] > target 就终止遍历
        for (int i = startIndex; i < candidates.size() && sum + candidates[i] <=
target; i++) {
            sum += candidates[i];
            path.push_back(candidates[i]);
            backtracking(candidates, target, sum, i);
            sum -= candidates[i];
            path.pop_back();
        }
    }
public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        result.clear();
        path.clear();
        sort(candidates.begin(), candidates.end()); // 需要排序
        backtracking(candidates, target, 0, 0);
        return result;
    }
};
```

组合求和(三)

力扣原题

==区别在于序列中有重复元素，但却不能存在重复组合==

实现代码:

```
class Solution {
private:
    vector<vector<int>> result;
    vector<int> path;
    void backtracking(vector<int>& candidates, int target, int sum, int startIndex,
vector<bool>& used) {
```



```

        if (sum == target) {
            result.push_back(path);
            return;
        }
        for (int i = startIndex; i < candidates.size() && sum + candidates[i] <=
target; i++) {
            // used[i - 1] == true, 说明同一树枝candidates[i - 1]使用过
            // used[i - 1] == false, 说明同一树层candidates[i - 1]使用过
            // 要对同一树层使用过的元素进行跳过
            if (i > 0 && candidates[i] == candidates[i - 1] && used[i - 1] ==
false) {
                continue; //解决同一层元素不能相同, 但每一树枝中元素可相同的问题
            }
            sum += candidates[i];
            path.push_back(candidates[i]);
            used[i] = true;
            backtracking(candidates, target, sum, i + 1, used); // 和39.组合总和的区
别1, 这里是i+1, 每个数字在每个组合中只能使用一次
            used[i] = false;
            sum -= candidates[i];
            path.pop_back();
        }
    }

public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        vector<bool> used(candidates.size(), false);
        path.clear();
        result.clear();
        // 首先把给candidates排序, 让其相同的元素都挨在一起。
        sort(candidates.begin(), candidates.end());
        backtracking(candidates, target, 0, 0, used);
        return result;
    }
};

```

多集合求组合

==如果是一个集合来求组合的话, 就需要startIndex, 如果是多个集合取组合, 各个集合之间相互不影响, 那么就不用startIndex==

力扣原题

实现代码:

```

class Solution {
private:
    const string letterMap[10] = {
        "", // 0
        "", // 1
        "abc", // 2

```

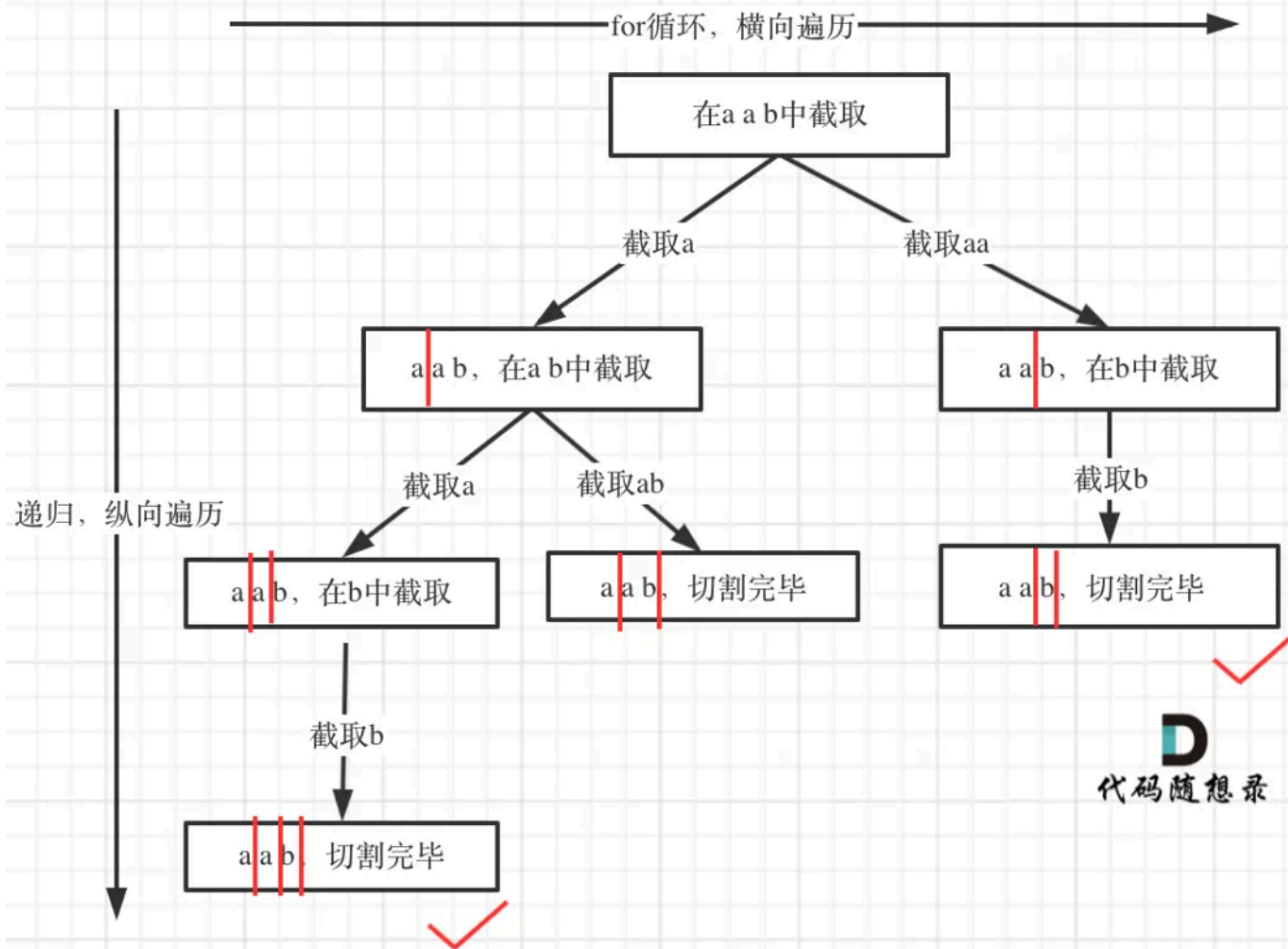
```

        "def", // 3
        "ghi", // 4
        "jkl", // 5
        "mno", // 6
        "pqrs", // 7
        "tuv", // 8
        "wxyz", // 9
    };
public:
    vector<string> result;
    string s;
    void backtracking(const string& digits, int index) {
        if (index == digits.size()) {
            result.push_back(s);
            return;
        }
        int digit = digits[index] - '0';           // 将index指向的数字转为int
        string letters = letterMap[digit];         // 取数字对应的字符集
        for (int i = 0; i < letters.size(); i++) {
            s.push_back(letters[i]);               // 处理
            backtracking(digits, index + 1);       // 递归, 注意index+1, 下一层要处理下一
个数字了
            s.pop_back();                          // 回溯
        }
    }
    vector<string> letterCombinations(string digits) {
        s.clear();
        result.clear();
        if (digits.size() == 0) {
            return result;
        }
        backtracking(digits, 0);
        return result;
    }
};

```

二、切割问题

[力扣原题](#)



实现代码：

```

class Solution {
private:
    vector<vector<string>> result;
    vector<string> path; // 放已经回文的子串
    void backtracking (const string& s, int startIndex) {
        // 如果起始位置已经大于s的大小，说明已经找到了一组分割方案了
        if (startIndex >= s.size()) {
            result.push_back(path);
            return;
        }
        for (int i = startIndex; i < s.size(); i++) {
            if (isPalindrome(s, startIndex, i)) { // 是回文子串
                // 获取[startIndex,i]在s中的子串
                string str = s.substr(startIndex, i - startIndex + 1);
                path.push_back(str);
            } else {
                continue;
            }
            backtracking(s, i + 1); // 寻找i+1为起始位置的子串
            path.pop_back(); // 回溯过程，弹出本次已经填在的子串
        }
    }
    bool isPalindrome(const string& s, int start, int end) {
        for (int i = start, j = end; i < j; i++, j--) {

```

```

        if (s[i] != s[j]) {
            return false;
        }
    }
    return true;
}
public:
    vector<vector<string>> partition(string s) {
        result.clear();
        path.clear();
        backtracking(s, 0);
        return result;
    }
};

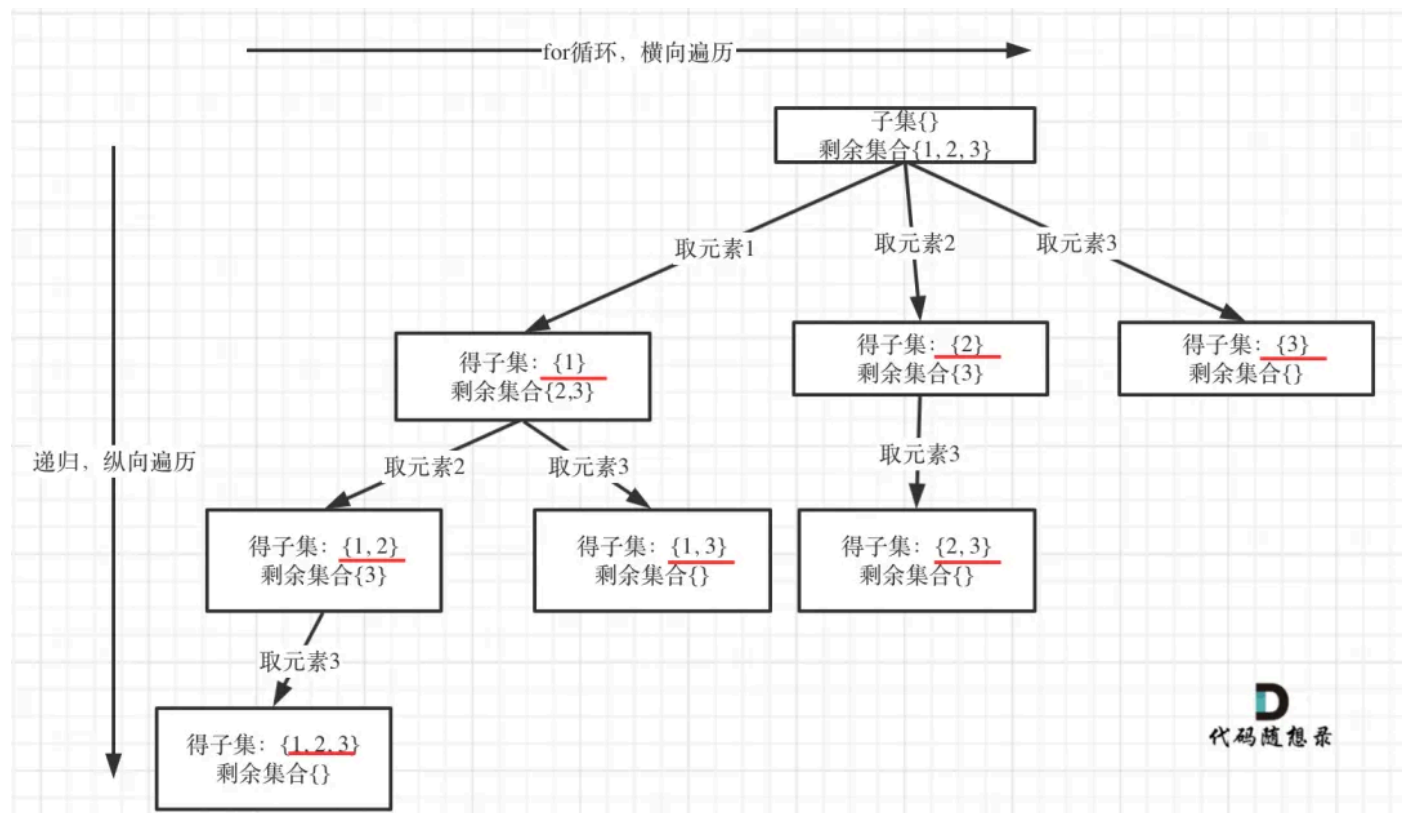
```

三、子集问题

子集问题(一)

力扣原题

==与上述组合问题的区别在于树的每一个节点都要收集==



实现代码:

```

class Solution {
private:
    vector<vector<int>> result;
    vector<int> path;

```

```

void backtracking(vector<int>& nums, int startIndex) {
    result.push_back(path); // 实现收集子集的关键步骤
    if (startIndex >= nums.size()) { // 终止条件可以不加
        return;
    }
    for (int i = startIndex; i < nums.size(); i++) { // 单层递归逻辑
        path.push_back(nums[i]);
        backtracking(nums, i + 1);
        path.pop_back();
    }
}

public:
    vector<vector<int>> subsets(vector<int>& nums) {
        result.clear();
        path.clear();
        backtracking(nums, 0);
        return result;
    }
};

```

子集问题(二)

力扣原题

==与上述问题的区别在于数组中包含重复元素，要去重，但与组合问题中去重思路相同
==

实现代码：

```

class Solution {
private:
    vector<vector<int>> result;
    vector<int> path;
    void backtracking(vector<int>& nums, int startIndex, vector<bool>& used) {
        result.push_back(path);
        for (int i = startIndex; i < nums.size(); i++) {
            // used[i - 1] == true, 说明同一树支candidates[i - 1]使用过
            // used[i - 1] == false, 说明同一树层candidates[i - 1]使用过
            // 而我们要对同一树层使用过的元素进行跳过
            if (i > 0 && nums[i] == nums[i - 1] && used[i - 1] == false) {
                continue;
            }
            path.push_back(nums[i]);
            used[i] = true;
            backtracking(nums, i + 1, used);
            used[i] = false;
            path.pop_back();
        }
    }

public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {

```

```

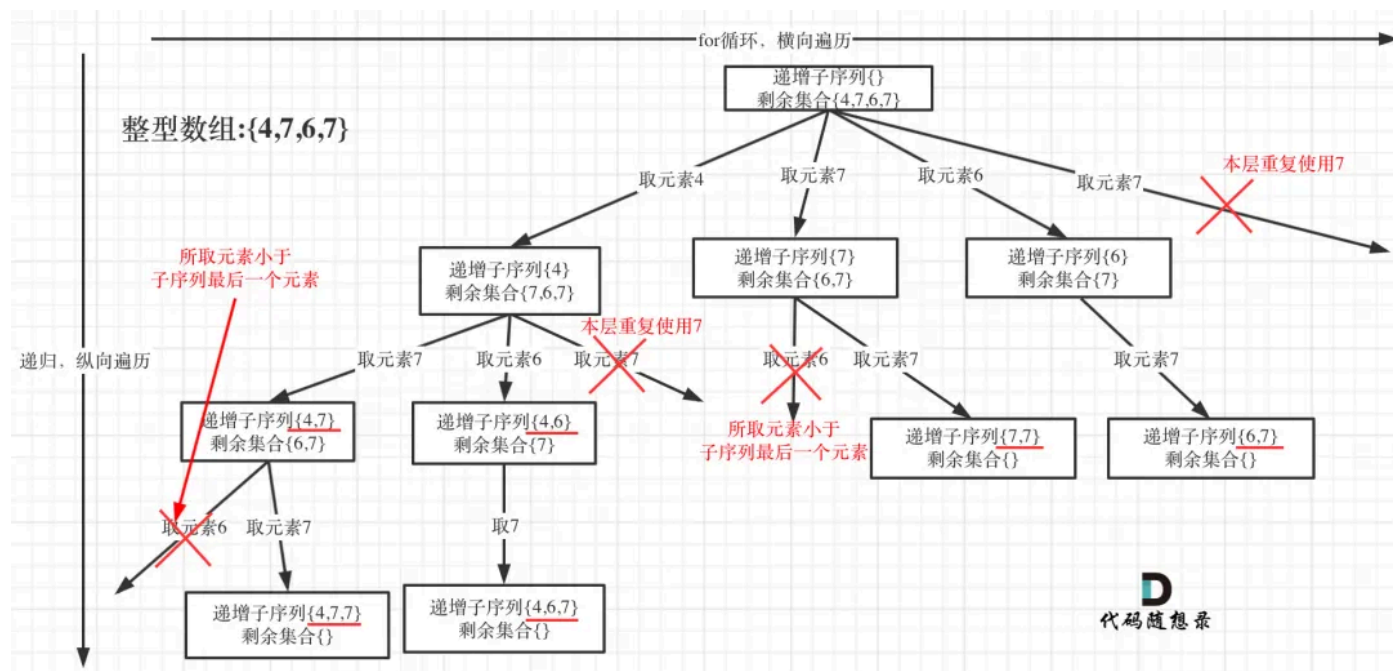
        result.clear();
        path.clear();
        vector<bool> used(nums.size(), false);
        sort(nums.begin(), nums.end()); // 去重需要排序
        backtracking(nums, 0, used);
        return result;
    }
};

```

非递减子序列问题

力扣原题

==与上述问题的区别在于不能进行排序，应按照序列原本的顺序查找，注意上述子集问题不能使用该方法==



实现代码：

```

class Solution {
private:
    vector<vector<int>> result;
    vector<int> path;
    void backtracking(vector<int>& nums, int startIndex) {
        if (path.size() > 1) {
            result.push_back(path);
        }
        int used[201] = {0}; // 这里使用数组来进行去重操作，题目说数值范围[-100, 100]
        for (int i = startIndex; i < nums.size(); i++) {
            if ((!path.empty() && nums[i] < path.back()) || used[nums[i] + 100] == 1) { // 限制了条件
                continue; // 上述子集问题不能使用该方法的原因
            }
        }
    }
};

```

```

        used[nums[i] + 100] = 1; // 记录这个元素在本层用过了，本层后面不能再用了
        path.push_back(nums[i]);
        backtracking(nums, i + 1);
        path.pop_back();
    }
}
public:
    vector<vector<int>> findSubsequences(vector<int>& nums) {
        result.clear();
        path.clear();
        backtracking(nums, 0);
        return result;
    }
};

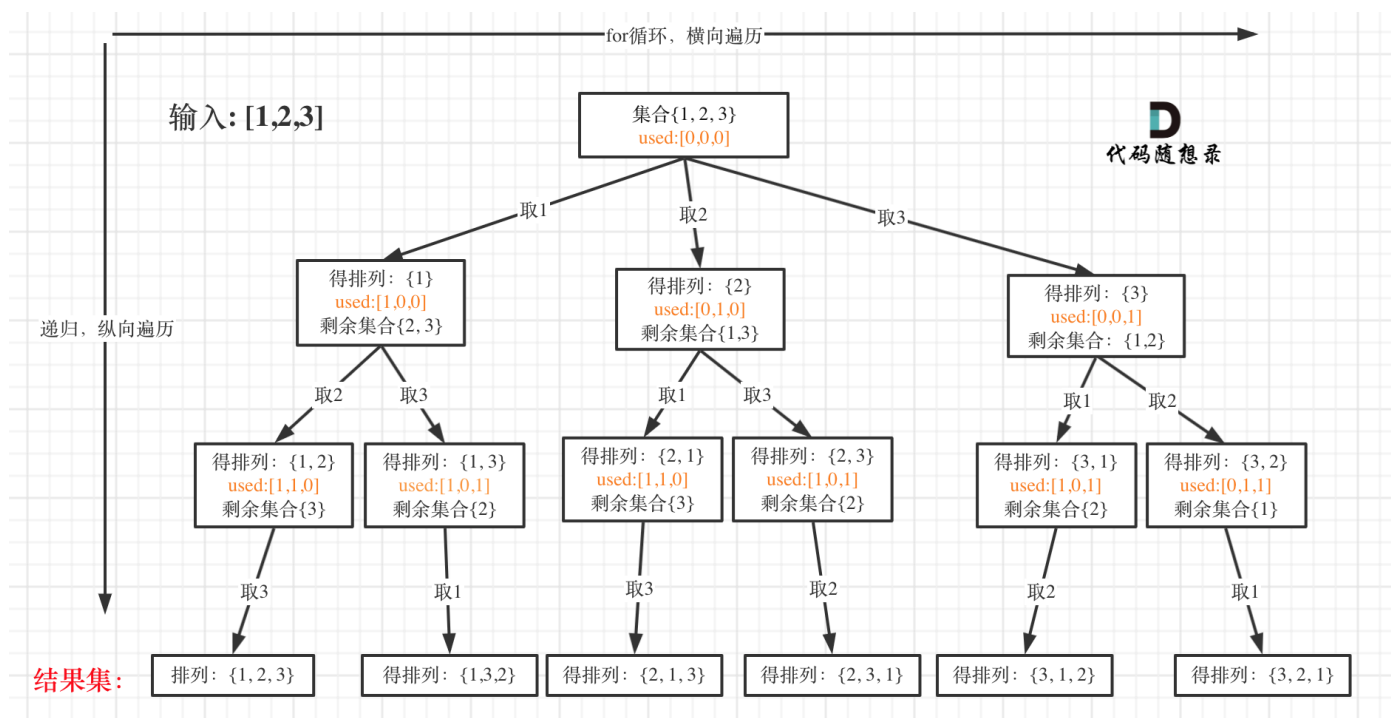
```

四、排列问题

排列问题(一)

力扣原题

==与上述题目的区别在于改题目为树枝上不能有重复元素，并且每次从头遍历==



实现代码:

```

class Solution {
public:
    vector<vector<int>> result;
    vector<int> path;
    void backtracking (vector<int>& nums, vector<bool>& used) {
        // 此时说明找到了一组
        if (path.size() == nums.size()) {

```

```

        result.push_back(path);
        return;
    }
    for (int i = 0; i < nums.size(); i++) {
        if (used[i] == true) continue; // path里已经收录的元素，直接跳过
        used[i] = true;
        path.push_back(nums[i]);
        backtracking(nums, used);
        path.pop_back();
        used[i] = false;
    }
}
vector<vector<int>> permute(vector<int>& nums) {
    result.clear();
    path.clear();
    vector<bool> used(nums.size(), false);
    backtracking(nums, used);
    return result;
}
};

```

排列问题(二)

力扣原题

==和组合问题中去重方式相同，不过排列问题即可树层去重也可树枝去重，树层效率更高==

实现代码：

```

class Solution {
private:
    vector<vector<int>> result;
    vector<int> path;
    void backtracking (vector<int>& nums, vector<bool>& used) {
        // 此时说明找到了一组
        if (path.size() == nums.size()) {
            result.push_back(path);
            return;
        }
        for (int i = 0; i < nums.size(); i++) {
            // used[i - 1] == true, 说明同一树支nums[i - 1]使用过
            // used[i - 1] == false, 说明同一树层nums[i - 1]使用过
            // 如果同一树层nums[i - 1]使用过则直接跳过
            if (i > 0 && nums[i] == nums[i - 1] && used[i - 1] == false) {
                continue; //其实也可以更换为一个数组记录元素出现的次数(树枝去重)
            }
            if (used[i] == false) {
                used[i] = true;
                path.push_back(nums[i]);
                backtracking(nums, used);
                path.pop_back();
            }
        }
    }
};

```



```

        used[i] = false;
    }
}
}
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        result.clear();
        path.clear();
        sort(nums.begin(), nums.end()); // 排序
        vector<bool> used(nums.size(), false);
        backtracking(nums, used);
        return result;
    }
};

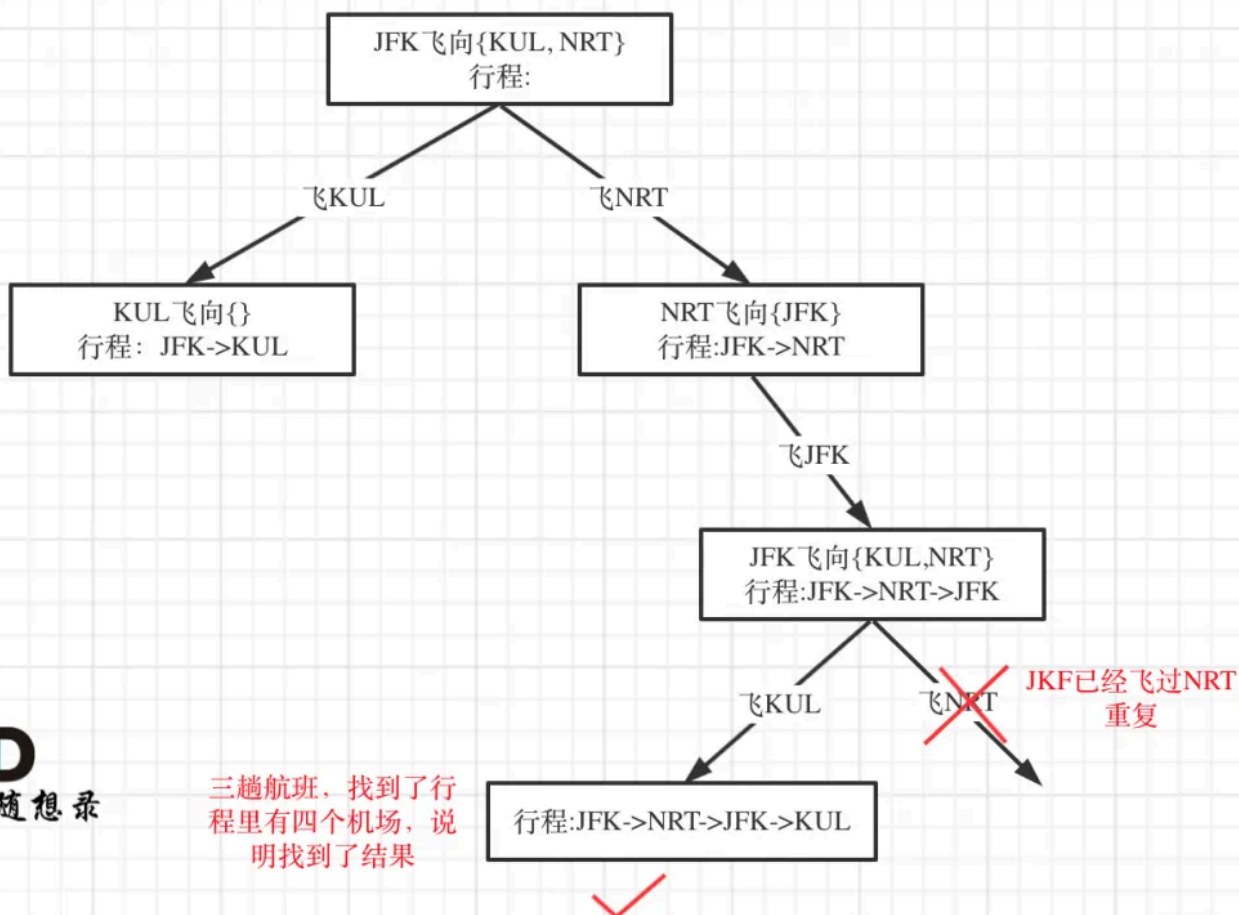
```

五、安排行程问题

力扣原题

==与图中的欧拉回路特别相像==

输入: [["JFK", "KUL"], ["JFK", "NRT"], ["NRT", "JFK"]]



实现代码：

```

class Solution {
private:
// unordered_map<出发机场, map<到达机场, 航班次数>> targets
unordered_map<string, map<string, int>> targets; //其实也可以简单用图的两种存储方式
bool backtracking(int ticketNum, vector<string>& result) {
    if (result.size() == ticketNum + 1) {
        return true;
    }
    for (pair<const string, int>& target : targets[result[result.size() - 1]]) {
        if (target.second > 0 ) { // 记录到达机场是否飞过了
            result.push_back(target.first);
            target.second--;
            if (backtracking(ticketNum, result)) return true;
            result.pop_back();
            target.second++;
        }
    }
    return false;
}
public:
    vector<string> findItinerary(vector<vector<string>>& tickets) {
        targets.clear();
        vector<string> result;
        for (const vector<string>& vec : tickets) {
            targets[vec[0]][vec[1]]++; // 记录映射关系
        }
        result.push_back("JFK"); // 起始机场
        backtracking(tickets.size(), result);
        return result;
    }
};

```

六、棋盘问题

n皇后问题

力扣原题


```

bool isValid(int row, int col, vector<string>& chessboard, int n) { //判断该位置是
    否可以放皇后
    int count = 0;
    // 检查列
    for (int i = 0; i < row; i++) { // 这是一个剪枝
        if (chessboard[i][col] == 'Q') {
            return false;
        }
    }
    // 检查 45度角是否有皇后
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
        if (chessboard[i][j] == 'Q') {
            return false;
        }
    }
    // 检查 135度角是否有皇后
    for(int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
        if (chessboard[i][j] == 'Q') {
            return false;
        }
    }
    return true;
}

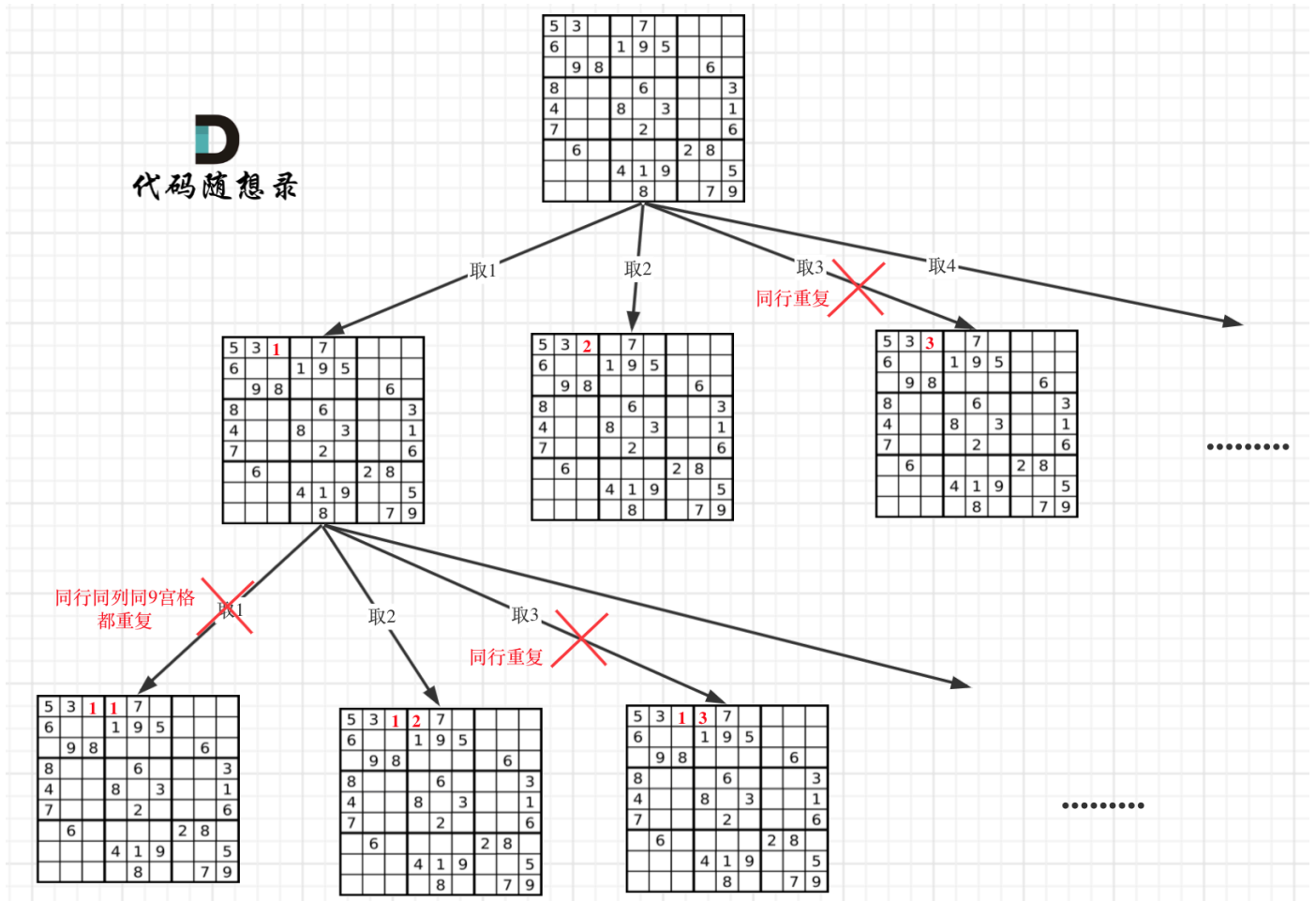
public:
    vector<vector<string>> solveNQueens(int n) {
        result.clear();
        vector<string> chessboard(n, string(n, '.')); //初始化全为"."
        backtracking(n, 0, chessboard);
        return result;
    }
};

```

解数独问题

力扣原题

代码随想录



实现代码:

```
class Solution {
private:
bool backtracking(vector<vector<char>>& board) {
    for (int i = 0; i < board.size(); i++) { // 遍历行
        for (int j = 0; j < board[0].size(); j++) { // 遍历列
            if (board[i][j] != '.') continue; // 双重for循环的目的在于找到要填入
            // 数字的位置，递归来检验所有情况
            for (char k = '1'; k <= '9'; k++) { // (i, j) 这个位置放k是否合适
                if (isValid(i, j, k, board)) {
                    board[i][j] = k; // 放置k
                    if (backtracking(board)) return true; // 递归的作用在于检验选择这
                    // 一个数字后是否可形成正解
                    board[i][j] = '.'; // 回溯，撤销k
                    // 递归即是
                    // 在这一选择下的下一选择
                }
            }
            return false; // 9个数都试完了，都不行，那么就返
        }
    }
    return true; // 遍历完没有返回false，说明找到了合适棋盘位置了
}

bool isValid(int row, int col, char val, vector<vector<char>>& board) {
    for (int i = 0; i < 9; i++) { // 判断行里是否重复
        if (board[row][i] == val) {
            return false;
        }
    }
}
```

```

    }
    for (int j = 0; j < 9; j++) { // 判断列里是否重复
        if (board[j][col] == val) {
            return false;
        }
    }
    int startRow = (row / 3) * 3; //board是从0开始的索引
    int startCol = (col / 3) * 3;
    for (int i = startRow; i < startRow + 3; i++) { // 判断9方格里是否重复
        for (int j = startCol; j < startCol + 3; j++) {
            if (board[i][j] == val ) {
                return false;
            }
        }
    }
    return true;
}
public:
    void solveSudoku(vector<vector<char>>& board) {
        backtracking(board);
    }
};

```

优化时间后代码:

```

class Solution {
private:
    bool rows[9][9] = {false}; // 行的数字使用情况
    bool cols[9][9] = {false}; // 列的数字使用情况
    bool boxes[9][9] = {false}; // 3x3 小方格的数字使用情况
    //emptyCells表示数独表格中的空格，index为递归开始位置
    bool backtracking(vector<vector<char>>& board, vector<pair<int, int>>&
emptyCells, int index) {
        if (index == emptyCells.size()) return true; // 所有空格都处理完，返回 true

        auto [row, col] = emptyCells[index];
        int boxIndex = (row / 3) * 3 + (col / 3); // 计算小方格索引

        for (int num = 0; num < 9; ++num) { // 尝试填入数字 1~9
            if (rows[row][num] || cols[col][num] || boxes[boxIndex][num]) continue;
            // 如果数字已使用，跳过

            board[row][col] = num + '1'; // 填入数字
            rows[row][num] = cols[col][num] = boxes[boxIndex][num] = true; // 更新状态

            if (backtracking(board, emptyCells, index + 1)) return true; // 递归处理下一个空格

            board[row][col] = '.'; // 回溯，撤销当前数字
            rows[row][num] = cols[col][num] = boxes[boxIndex][num] = false;
        }

        return false; // 当前空格所有数字都无效，返回 false
    }
}

```

```

public:
    void solveSudoku(vector<vector<char>>& board) {
        vector<pair<int, int>> emptyCells; // 记录所有空格的位置
        for (int i = 0; i < 9; ++i) {
            for (int j = 0; j < 9; ++j) {
                if (board[i][j] == '.') {
                    emptyCells.emplace_back(i, j);
                } else {
                    int num = board[i][j] - '1'; // 转换为 0~8 的索引
                    int boxIndex = (i / 3) * 3 + (j / 3);
                    rows[i][num] = cols[j][num] = boxes[boxIndex][num] = true;
                }
            }
        }

        backtrack(board, emptyCells, 0); // 从第一个空格开始回溯
    }
};

```

第三节、动态规划 🙄

动态规划中每一个状态一定是由上一个状态推导出来的

求解步骤：

1. 确定dp数组（dp table）以及下标的含义
2. 确定递推公式
3. dp数组如何初始化
4. 确定遍历顺序
5. 举例推导dp数组

一、基础题型

斐波那契数列

[力扣原题](#)

[原题讲解在这里](#)

实现代码:

```

class Solution {
public:
    int fib(int N) {

```

```

    if (N <= 1) return N;
    // dp[i]的含义即为第i个斐波那契数
    int dp[2];
    // dp数组初始化
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= N; i++) { // 遍历顺序—从前向后遍历
        int sum = dp[0] + dp[1]; // 递推方程
        dp[0] = dp[1];
        dp[1] = sum;
    }
    return dp[1];
}
};

```

不同路径问题

力扣原题

[原题讲解在这里](#)

实现代码:

```

class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        if (obstacleGrid[0][0] == 1)
            return 0;
        vector<int> dp(obstacleGrid[0].size()); // 空间优化
        for (int j = 0; j < dp.size(); ++j) // 初始化dp数组为第0行数据
            if (obstacleGrid[0][j] == 1)
                dp[j] = 0;
            else if (j == 0)
                dp[j] = 1;
            else
                dp[j] = dp[j-1];

        for (int i = 1; i < obstacleGrid.size(); ++i) // 每次循环表示dp代表第几行
            for (int j = 0; j < dp.size(); ++j){
                if (obstacleGrid[i][j] == 1)
                    dp[j] = 0;
                else if (j != 0)
                    dp[j] = dp[j] + dp[j-1]; // 空间优化后的递推公式
            }
        return dp.back();
    }
};

```

整数拆分问题

[力扣原题](#)

[原题讲解在这里](#)

实现代码:

```
class Solution {
public:
    int integerBreak(int n) {
        vector<int> dp(n + 1);
        dp[2] = 1;
        for (int i = 3; i <= n; i++) {
            for (int j = 1; j <= i / 2; j++) {
                dp[i] = max(dp[i], max((i - j) * j, dp[i - j] * j));
            }
        }
        return dp[n];
    }
};
```

二叉搜索树问题*

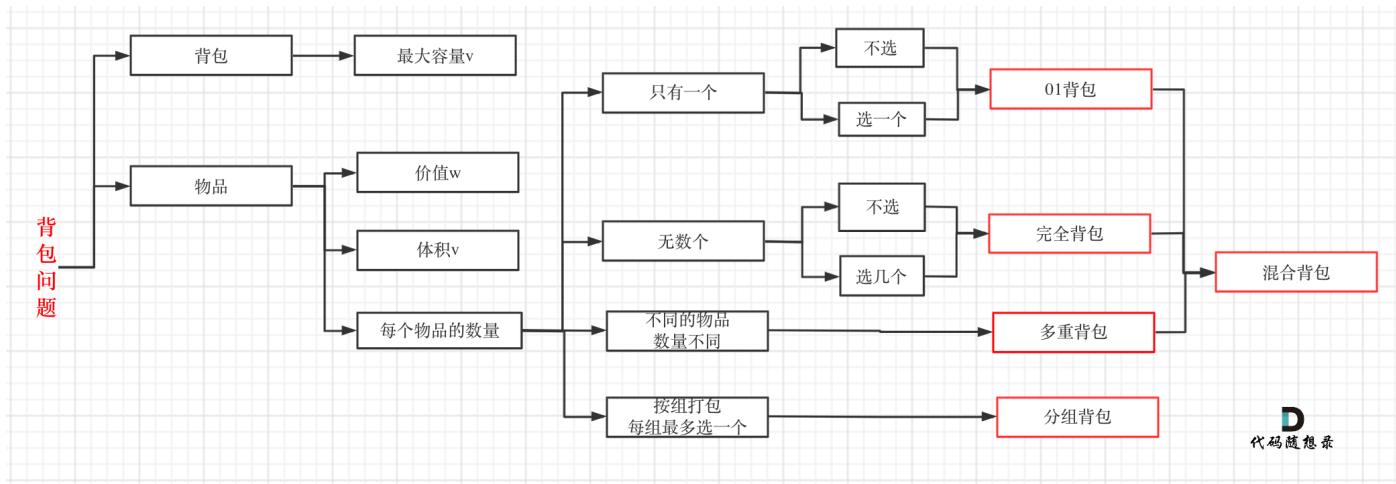
[力扣原题](#)

[原题讲解在这里](#)

实现代码:

```
class Solution {
public:
    int numTrees(int n) {
        vector<int> dp(n + 1);
        dp[0] = 1;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= i; j++) { //每个节点为父节点时左右节点个数所构成树的
个数的乘积
                dp[i] += dp[j - 1] * dp[i - j];
            }
        }
        return dp[n];
    }
};
```

二、背包问题



01背包问题

01背包问题理论基础

$dp[i][j]$

背包重量j:

	0	1	2	3	4
物品0:					
物品1:					
物品2:					

代码随想录

i 表示此行可放的物品， j 表示背包的容量， $dp[i][j]$ 表示从下标为 $[0-i]$ 的物品里任意取，放进容量为 j 的背包，价值总和最大是多少

推导递推公式：

- 不放物品 i ：背包容量为 j ，里面不放物品 i 的最大价值是 $dp[i - 1][j]$ 。
- 放物品 i ：背包空出物品 i 的容量后，背包容量为 $j - \text{weight}[i]$ ， $dp[i - 1][j - \text{weight}[i]]$ 为背包容量为 $j - \text{weight}[i]$ 且不放物品 i 的最大价值，那么 $dp[i - 1][j - \text{weight}[i]] + \text{value}[i]$ （物品 i 的价值），就是背包放物品 i 得到的最大价值

==递归公式: $dp[i][j] = \max(dp[i-1][j], dp[i-1][j - \text{weight}[i]] + \text{value}[i])$;==

==难点在于如何将问题转化为01背包问题==

二维dp解决01背包问题代码实现:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n, bagweight; // bagweight代表背包容量

    cin >> n >> bagweight;

    vector<int> weight(n, 0); // 存储每件物品所占空间
    vector<int> value(n, 0); // 存储每件物品价值

    for(int i = 0; i < n; ++i) {
        cin >> weight[i];
    }
    for(int j = 0; j < n; ++j) {
        cin >> value[j];
    }
    // dp数组, dp[i][j]代表背包容量为j的情况下,从下标为[0, i]的物品里面任意取,能达到的最大价值
    vector<vector<int>> dp(weight.size(), vector<int>(bagweight + 1, 0));

    // 初始化, 因为需要用到dp[i - 1]的值
    // j < weight[0]已在上方被初始化为0
    // j >= weight[0]的值就初始化为value[0]
    for (int j = weight[0]; j <= bagweight; j++) {
        dp[0][j] = value[0];
    }

    for(int i = 1; i < weight.size(); i++) { // 遍历物品
        for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
            if (j < weight[i]) dp[i][j] = dp[i - 1][j]; // 如果装不下这个物品,那么就继承dp[i - 1][j]的值
            else {
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);
            }
        }
    }
    cout << dp[n - 1][bagweight] << endl;

    return 0;
}
```

一维滚动数组代码实现:

```

// 一维dp数组实现
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // 读取 M 和 N
    int M, N;
    cin >> M >> N;

    vector<int> costs(M);
    vector<int> values(M);

    for (int i = 0; i < M; i++) {
        cin >> costs[i];
    }
    for (int j = 0; j < M; j++) {
        cin >> values[j];
    }

    // 创建一个动态规划数组dp，初始值为0
    vector<int> dp(N + 1, 0);

    // 外层循环遍历每个物品
    for (int i = 0; i < M; ++i) {
        // 内层循环从 N 空间逐渐减少到当前物品所占空间—注意内部循环时从后往前遍历
        for (int j = N; j >= costs[i]; --j) {
            // 考虑当前物品选择和不选择的情况，选择最大值
            dp[j] = max(dp[j], dp[j - costs[i]] + values[i]); //max使得不需要考虑各
            物品的价值顺序
        }
    }

    // 输出dp[N]，即在给定 N 行李空间可以携带的物品最大价值
    cout << dp[N] << endl;

    return 0;
}

```

分割等和子集问题

力扣原题

[原题讲解在这里](#)

将该问题转化为01背包问题

- 背包的体积为 $\text{sum} / 2$
- 背包要放入的商品（集合里的元素）重量为元素的数值，价值也为元素的数值
- 背包如果正好装满，说明找到了总和为 $\text{sum} / 2$ 的子集。

- 背包中每一个元素是不可重复放入。

==本题中，dp[j]表示 背包总容量（所能装的总重量）是j，放进物品后，背包的最大重量为dp[j]。==

实现代码:

```
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int sum = 0;

        // dp[i]中的i表示背包内总和
        // 题目中说：每个数组中的元素不会超过 100，数组的大小不会超过 200
        // 总和不会大于20000，背包最大只需要其中一半，所以10001大小就可以了
        vector<int> dp(10001, 0);
        for (int i = 0; i < nums.size(); i++) {
            sum += nums[i];
        }
        // 也可以使用库函数一步求和
        // int sum = accumulate(nums.begin(), nums.end(), 0);
        if (sum % 2 == 1) return false;
        int target = sum / 2;

        // 开始 01背包
        for(int i = 0; i < nums.size(); i++) {
            for(int j = target; j >= nums[i]; j--) { // 每一个元素一定是不可重复放入，
                // 所以从大到小遍历
                dp[j] = max(dp[j], dp[j - nums[i]] + nums[i]);
            }
        }
        // 集合中的元素正好可以凑成总和target
        if (dp[target] == target) return true;
        return false;
    }
};
```

粉碎石头问题

力扣原题

[原题讲解在这里](#)

该问题与等和子集问题非常相似，即找出两堆质量相似的石块

==本题中，dp[j]表示 背包总容量（所能装的总重量）是j，加入石头后，背包的最大重量为dp[j]。==

实现代码:

```

class Solution {
public:
    int lastStoneWeightII(vector<int>& stones) {
        vector<int> dp(15001, 0);
        int sum = 0;
        for (int i = 0; i < stones.size(); i++) sum += stones[i];
        int target = sum / 2;

        for (int i = 0; i < stones.size(); i++) { // 遍历物品
            for (int j = target; j >= stones[i]; j--) { // 遍历背包
                dp[j] = max(dp[j], dp[j - stones[i]] + stones[i]);
            }
        }
        return sum - dp[target] - dp[target];
    }
};

```

目标和问题

力扣原题

原题讲解在这里

== $\text{left-right}=\text{target}$, $\text{left+right}=\text{sum}$.=> $\text{bagsize}=(\text{sum}+\text{target})/2$ ==

== $\text{dp}[i][j]$: 使用下标为 $[0, i]$ 的 $\text{nums}[i]$ 能够凑满 j (包括 j) 这么大容量的包, 有 $\text{dp}[i][j]$ 种方法==

- 不放物品 i : 即背包容量为 j , 里面不放物品 i , 装满有 $\text{dp}[i-1][j]$ 中方法。
- 放物品 i : 即: 先空出物品 i 的容量, 背包容量为 $(j - \text{物品}i\text{容量})$, 放满背包有 $\text{dp}[i-1][j - \text{物品}i\text{容量}]$ 种方法。

==递推公式: $\text{dp}[i][j] = \text{dp}[i-1][j] + \text{dp}[i-1][j - \text{nums}[i]]$;==

本题的思想为将背包装满有几种方法

二维数组实现代码:

```

class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum = 0;
        for (int i = 0; i < nums.size(); i++) sum += nums[i];
        if (abs(target) > sum) return 0; // 此时没有方案
        if ((target + sum) % 2 == 1) return 0; // 此时没有方案

        int bagSize = (target + sum) / 2; //背包容量
    }
};

```

```

vector<vector<int>> dp(nums.size(), vector<int>(bagSize + 1, 0));

// 初始化最上行
if (nums[0] <= bagSize) dp[0][nums[0]] = 1;

// 初始化最左列，最左列其他数值在递推公式中就完成了赋值
dp[0][0] = 1;

int numZero = 0; // 出现重量为零的元素，共有2^n种情况
for (int i = 0; i < nums.size(); i++) {
    if (nums[i] == 0) numZero++;
    dp[i][0] = (int) pow(2.0, numZero);
}

// 以下遍历顺序行列可以颠倒
for (int i = 1; i < nums.size(); i++) { // 行，遍历物品
    for (int j = 0; j <= bagSize; j++) { // 列，遍历背包
        if (nums[i] > j) dp[i][j] = dp[i - 1][j];
        else dp[i][j] = dp[i - 1][j] + dp[i - 1][j - nums[i]];
        // dp[ i ] [ j ] = dp[ i - 1 ] [ j ] + dp[ i - 1 ] * max( 0 , [ j -
nums[ i ] ] );
    }
}
return dp[nums.size() - 1][bagSize];
}
};

```

一维dp数组代码实现：

```

class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum = 0;
        for (int i = 0; i < nums.size(); i++){
            sum += nums[i];
        }
        if (abs(target) > sum) return 0; // 此时没有方案
        if ((target + sum) % 2 == 1) return 0; // 此时没有方案
        int bagSize = (target + sum) / 2;

        vector<int> dp(bagSize + 1, 0);
        dp[0] = 1; // 未考虑0元素
        for (int i = 0; i < nums.size(); i++) {
            for (int j = bagSize; j >= nums[i]; j--) {
                dp[j] += dp[j - nums[i]];
            }
        }
        return dp[bagSize];
    }
};

```

力扣原题

原题讲解在这里

==这个背包有两个维度,一个是m 一个是n,而不同长度的字符串就是不同大小的待装物品==

i 和 j 相当于一个二维的物体 “重量”，字符串的长度就相当于 “价值”，dp[i][j] 表示背包容量为i个0,j个1时所能装的最大字符串个数

实现代码:

```
class Solution {
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
        vector<vector<int>>> dp(m + 1, vector<int> (n + 1, 0)); // 默认初始化0
        for (string str : strs) { // 遍历物品
            int oneNum = 0, zeroNum = 0;
            for (char c : str) {
                if (c == '0') zeroNum++;
                else oneNum++;
            }
            for (int i = m; i >= zeroNum; i--) { // 遍历背包容量且从后向前遍历!
                for (int j = n; j >= oneNum; j--) {
                    dp[i][j] = max(dp[i][j], dp[i - zeroNum][j - oneNum] + 1); //
                    加一是个数加一
                }
            }
        }
        return dp[m][n];
    }
};
```

完全背包问题

完全背包问题理论基础

==每个物品可以放入多次，只有遍历顺序改变==

实现代码:

```
#include <iostream>
#include <vector>
using namespace std;
```



```

// 先遍历物品，在遍历背包
// 先遍历物品，在遍历背包
void test_CompletePack() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagWeight = 4;
    vector<int> dp(bagWeight + 1, 0);
    for(int i = 0; i < weight.size(); i++) { // 遍历物品
        for(int j = weight[i]; j <= bagWeight; j++) { // 遍历背包容量,完全背包这里要从
            前往后遍历
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
        }
    }
    cout << dp[bagWeight] << endl;
}

int main() {
    test_CompletePack();
}

```

零钱兑换问题(一)

力扣原题

原题讲解在这里

==装满背包问题的状态转移方程一般都为 $dp[j] += dp[j - weight[i]]$ ==

==如果求组合数就是外层for循环遍历物品，内层for遍历背包。==

==如果求排列数就是外层for遍历背包，内层for循环遍历物品。==

排列问题要先把每个物体遍历一遍确定顺序,而组合问题则是只需要判断是否存在,不能出现顺序

实现代码:

```

class Solution {
public:
    int change(int amount, vector<int>& coins) {
        vector<uint64_t> dp(amount + 1, 0); // 防止相加数据超int
        dp[0] = 1; // 只有一种方式达到0
        for (int i = 0; i < coins.size(); i++) { // 遍历物品
            for (int j = coins[i]; j <= amount; j++) { // 遍历背包
                dp[j] += dp[j - coins[i]];
            }
        }
        return dp[amount]; // 返回组合数
    }
}

```

```
    }  
};
```

零钱兑换问题(二)

力扣原题

原题讲解在这里

实现代码:

```
class Solution {  
public:  
    int coinChange(vector<int>& coins, int amount) {  
        vector<int> dp(amount + 1, INT_MAX);  
        dp[0] = 0;  
        for (int i = 0; i < coins.size(); i++) { // 遍历物品  
            for (int j = coins[i]; j <= amount; j++) { // 遍历背包  
                if (dp[j - coins[i]] != INT_MAX) { // 如果dp[j - coins[i]]是初始值则  
跳过  
                    dp[j] = min(dp[j - coins[i]] + 1, dp[j]); // 记得加一  
                }  
            }  
        }  
        if (dp[amount] == INT_MAX) return -1; // 不能放的位置均是INT_MAX  
        return dp[amount];  
    }  
};
```

单词拆分问题

力扣原题

原题讲解在这里

==如果确定dp[j] 是true,且 [j, i] 这个区间的子串出现在字典里那么dp[i]一定是true。(j < i)==

==本题是一个排列问题,考虑字符在目标中的出现顺序==

实现代码:

```
class Solution {  
public:  
    bool wordBreak(string s, vector<string>& wordDict) {  
        unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
```

```

vector<bool> dp(s.size() + 1, false);
dp[0] = true;
for (int i = 1; i <= s.size(); i++) {    // 遍历背包
    for (int j = 0; j < i; j++) {        // 遍历物品
        string word = s.substr(j, i - j); //substr(起始位置, 截取的个数)
        if (wordSet.find(word) != wordSet.end() && dp[j]) {
            dp[i] = true;
        }
    }
}
return dp[s.size()];
};

```

多重背包问题*

一般不考

与01背包的区别：

在循环内部加入了物品个数的循环

实现代码：

```

#include<iostream>
#include<vector>
using namespace std;
int main() {
    int bagWeight,n;
    cin >> bagWeight >> n;
    vector<int> weight(n, 0);
    vector<int> value(n, 0);
    vector<int> nums(n, 0);
    for (int i = 0; i < n; i++) cin >> weight[i];
    for (int i = 0; i < n; i++) cin >> value[i];
    for (int i = 0; i < n; i++) cin >> nums[i];

    vector<int> dp(bagWeight + 1, 0);

    for(int i = 0; i < n; i++) { // 遍历物品
        for(int j = bagWeight; j >= weight[i]; j--) { // 遍历背包容量
            // 以上为01背包，然后加一个遍历个数
            for (int k = 1; k <= nums[i] && (j - k * weight[i]) >= 0; k++) { // 遍
                dp[j] = max(dp[j], dp[j - k * weight[i]] + k * value[i]);
            }
        }
    }

    cout << dp[bagWeight] << endl;
}

```

三、打家劫舍问题

选择遍历方式+状态转移方程

打家劫舍问题(一)

[力扣原题](#)

[原题讲解在这里](#)

==经典==

实现代码:

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() == 0) return 0;
        if (nums.size() == 1) return nums[0];

        vector<int> dp(nums.size());
        dp[0] = nums[0];
        dp[1] = max(nums[0], nums[1]);
        for (int i = 2; i < nums.size(); i++) {
            dp[i] = max(dp[i - 2] + nums[i], dp[i - 1]);
        }

        return dp[nums.size() - 1];
    }
};
```

打家劫舍问题(二)

[力扣原题](#)

[原题讲解在这里](#)

==可以分为考虑首元素不考虑尾元素和考虑尾元素不考虑首元素两种情况==

实现代码:

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() == 0) return 0;
        if (nums.size() == 1) return nums[0];
```

```

        int result1 = robRange(nums, 0, nums.size() - 2); // 情况二
        int result2 = robRange(nums, 1, nums.size() - 1); // 情况三
        return max(result1, result2);
    }
    // 198.打家劫舍的逻辑
    int robRange(vector<int>& nums, int start, int end) {
        if (end == start) return nums[start];

        vector<int> dp(nums.size());
        dp[start] = nums[start];
        dp[start + 1] = max(nums[start], nums[start + 1]);
        for (int i = start + 2; i <= end; i++) {
            dp[i] = max(dp[i - 2] + nums[i], dp[i - 1]);
        }

        return dp[end];
    }
};

```

打家劫舍问题(三)

力扣原题

[原题讲解在这里](#)

==本题一定是要后序遍历，因为通过递归函数的返回值来做下一步计算==

==用一个长度为二的数组来记录该节点偷与不偷下的最大金钱==

实现代码:

```

class Solution {
public:
    int rob(TreeNode* root) {
        vector<int> result = robTree(root);
        return max(result[0], result[1]);
    }
    // 长度为2的数组，0：不偷，1：偷
    vector<int> robTree(TreeNode* cur) {
        if (cur == NULL) return vector<int>{0, 0};

        // 后序遍历保证从叶节点开始向上计算
        vector<int> left = robTree(cur->left);
        vector<int> right = robTree(cur->right);
        // 偷cur，那么就不能偷左右节点。
        int val1 = cur->val + left[0] + right[0];
        // 不偷cur，那么可以偷也可以不偷左右节点，则取较大的情况
        int val2 = max(left[0], left[1]) + max(right[0], right[1]);

        return {val2, val1};
    }
};

```

```
}  
};
```

四、股票问题

买卖股票问题一

力扣原题

[原题讲解在这里](#)

如果第*i*天持有股票即 $dp[i][0]$ ，那么可以由两个状态推出来

- 第*i*-1天就持有股票，那么就保持现状，所得现金就是昨天持有股票的所得现金即： $dp[i-1][0]$
- 第*i*天买入股票，所得现金就是买入今天的股票后所得现金即： $-prices[i]$

那么 $dp[i][0]$ 应该选所得现金最大的，所以 $dp[i][0] = \max(dp[i-1][0], -prices[i])$;

如果第*i*天不持有股票即 $dp[i][1]$ ，也可以由两个状态推出来

- 第*i*-1天就不持有股票，那么就保持现状，所得现金就是昨天不持有股票的所得现金即： $dp[i-1][1]$
- 第*i*天卖出股票，所得现金就是按照今天股票价格卖出后所得现金即： $prices[i] + dp[i-1][0]$

同样 $dp[i][1]$ 取最大的， $dp[i][1] = \max(dp[i-1][1], prices[i] + dp[i-1][0])$;

实现代码:

```
class Solution {  
public:  
    int maxProfit(vector<int>& prices) {  
        int len = prices.size();  
        if (len == 0) return 0;  
        vector<vector<int>> dp(len, vector<int>(2));  
        dp[0][0] -= prices[0];  
        dp[0][1] = 0;  
        for (int i = 1; i < len; i++) {  
            dp[i][0] = max(dp[i-1][0], -prices[i]);  
            dp[i][1] = max(dp[i-1][1], prices[i] + dp[i-1][0]);  
        }  
        return dp[len-1][1]; // 注意返回值  
    }  
};
```

```
    }  
};
```

滚动数组实现代码：

```
class Solution {  
public:  
    int maxProfit(vector<int>& prices) {  
        int len = prices.size();  
        vector<vector<int>> dp(2, vector<int>(2)); // 注意这里只开辟了一个2 * 2大小的  
二维数组  
        dp[0][0] -= prices[0];  
        dp[0][1] = 0;  
        for (int i = 1; i < len; i++) {  
            dp[i % 2][0] = max(dp[(i - 1) % 2][0], -prices[i]);  
            dp[i % 2][1] = max(dp[(i - 1) % 2][1], prices[i] + dp[(i - 1) % 2][0]);  
        }  
        return dp[(len - 1) % 2][1];  
    }  
};
```

买卖股票问题二(可买卖多次)

力扣原题

[原题讲解在这里](#)

如果第*i*天持有股票即 $dp[i][0]$ ，那么可以由两个状态推出来

- 第*i*-1天就持有股票，那么就保持现状，所得现金就是昨天持有股票的所得现金即： $dp[i-1][0]$
- 第*i*天买入股票，所得现金就是昨天不持有股票的所得现金减去 今天的股票价格即： $dp[i-1][1] - prices[i]$

那么 $dp[i][0]$ 应该选所得现金最大的，所以 $dp[i][0] = \max(dp[i-1][0], dp[i-1][1] - prices[i])$;

如果第*i*天不持有股票即 $dp[i][1]$ 的情况，依然可以由两个状态推出来

- 第*i*-1天就不持有股票，那么就保持现状，所得现金就是昨天不持有股票的所得现金即： $dp[i-1][1]$
- 第*i*天卖出股票，所得现金就是按照今天股票价格卖出后所得现金即： $prices[i] + dp[i-1][0]$

那么 $dp[i][1]$ 应该选所得现金最大的, 所以 $dp[i][1] = \max(dp[i-1][1], prices[i] + dp[i-1][0])$;

实现代码:

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int len = prices.size();
        vector<vector<int>> dp(len, vector<int>(2, 0));
        dp[0][0] -= prices[0];
        dp[0][1] = 0;
        for (int i = 1; i < len; i++) {
            dp[i][0] = max(dp[i-1][0], dp[i-1][1] - prices[i]); // 注意这里是和
121. 买卖股票的最佳时机唯一不同的地方。
            dp[i][1] = max(dp[i-1][1], dp[i-1][0] + prices[i]);
        }
        return dp[len-1][1];
    }
};
```

滚动数组实现代码:

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int len = prices.size();
        vector<vector<int>> dp(2, vector<int>(2)); // 注意这里只开辟了一个2 * 2大小的
二维数组
        dp[0][0] -= prices[0];
        dp[0][1] = 0;
        for (int i = 1; i < len; i++) {
            dp[i % 2][0] = max(dp[(i-1) % 2][0], dp[(i-1) % 2][1] - prices[i]);
            dp[i % 2][1] = max(dp[(i-1) % 2][1], prices[i] + dp[(i-1) % 2][0]);
        }
        return dp[(len-1) % 2][1];
    }
};
```

买卖股票问题三(最多买卖2次)

[力扣原题](#)

[原题讲解在这里](#)

==重点在于有多种选择: 买一次, 买两次, 不买==

一天一共就有五个状态:

0. 没有操作 (其实我们也可以不设置这个状态)
1. 第一次持有股票
2. 第一次不持有股票
3. 第二次持有股票
4. 第二次不持有股票

- $dp[i][1] = \max(dp[i-1][1], dp[i-1][0] - prices[i])$
- $dp[i][2] = \max(dp[i-1][2], dp[i-1][1] + prices[i])$
- $dp[i][3] = \max(dp[i-1][3], dp[i-1][2] - prices[i])$
- $dp[i][4] = \max(dp[i-1][4], dp[i-1][3] + prices[i])$

实现代码:

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.size() == 0) return 0;

        vector<vector<int>> dp(prices.size(), vector<int>(5, 0));
        dp[0][1] = -prices[0];
        dp[0][3] = -prices[0];

        for (int i = 1; i < prices.size(); i++) {
            dp[i][0] = dp[i-1][0];
            dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i]);
            dp[i][2] = max(dp[i-1][2], dp[i-1][1] + prices[i]);
            dp[i][3] = max(dp[i-1][3], dp[i-1][2] - prices[i]);
            dp[i][4] = max(dp[i-1][4], dp[i-1][3] + prices[i]);
        }

        return dp[prices.size() - 1][4];
    }
};
```

买卖股票问题四(最多买卖k次)

[力扣原题](#)

[原题讲解在这里](#)

==与上题相似==

实现代码:

```

class Solution {
public:
    int maxProfit(int k, vector<int>& prices) {

        if (prices.size() == 0) return 0;

        vector<vector<int>> dp(prices.size(), vector<int>(2 * k + 1, 0));
        for (int j = 1; j < 2 * k; j += 2) { //j为奇数时为持有状态，偶数时为未持有状态
            dp[0][j] = -prices[0];
        }

        for (int i = 1; i < prices.size(); i++) {
            for (int j = 0; j < 2 * k - 1; j += 2) {
                dp[i][j + 1] = max(dp[i - 1][j + 1], dp[i - 1][j] - prices[i]);
                dp[i][j + 2] = max(dp[i - 1][j + 2], dp[i - 1][j + 1] + prices[i]);
            }
        }
        return dp[prices.size() - 1][2 * k];
    }
};

```

买卖股票问题五(有手续费)

力扣原题

[原题讲解在这里](#)

==与问题(二)相似==

实现代码:

```

class Solution {
public:
    int maxProfit(vector<int>& prices, int fee) {
        int n = prices.size();
        vector<vector<int>> dp(n, vector<int>(2, 0));
        dp[0][0] -= prices[0]; // 持股票
        for (int i = 1; i < n; i++) {
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] - prices[i]);
            dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] + prices[i] - fee);
        }
        return max(dp[n - 1][0], dp[n - 1][1]);
    }
};

```

买卖股票问题六(有冷却期)

==本题一定是要后序遍历，因为通过递归函数的返回值来做下一步计算==

具体可以区分出如下四个状态：

0. 持有股票状态（今天买入股票，或者是之前就买入了股票然后没有操作，一直持有）
1. 不持有股票状态（两天前就卖出了股票，度过一天冷冻期。或者是前一天就是卖出股票状态，一直没操作）
2. 今天卖出股票
3. 今天为冷冻期状态，但冷冻期状态不可持续，只有一天！

遇到状态不清楚可以画状态图

实现代码:

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        if (n == 0) return 0;

        vector<vector<int>> dp(n, vector<int>(4, 0));
        dp[0][0] -= prices[0]; // 持股票

        for (int i = 1; i < n; i++) {
            dp[i][0] = max(dp[i - 1][0], max(dp[i - 1][3] - prices[i], dp[i - 1][1] - prices[i]));
            dp[i][1] = max(dp[i - 1][1], dp[i - 1][3]);
            dp[i][2] = dp[i - 1][0] + prices[i];
            dp[i][3] = dp[i - 1][2];
        }

        return max(dp[n - 1][3], max(dp[n - 1][1], dp[n - 1][2]));
    }
};
```

五、子序列问题

不连续子序列

最长公共子序列问题

力扣原题

原题讲解在这里

$dp[i][j]$:以下标 $i - 1$ 为结尾的A, 和以下标 $j - 1$ 为结尾的B, 最长重复子数组长度为 $dp[i][j]$

实现代码:

```
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        vector<vector<int>> dp(text1.size() + 1, vector<int>(text2.size() + 1, 0));
        for (int i = 1; i <= text1.size(); i++) {
            for (int j = 1; j <= text2.size(); j++) {
                if (text1[i - 1] == text2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1; // 若两个字符串接下来字符相同，
公共字符串长度加一
                } else {
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]); // 不论哪一个字符串
加一个字符，公共字符串长度不变
                }
            }
        }
        return dp[text1.size()][text2.size()];
    }
};
```

最长递增子序列问题

力扣原题

原题讲解在这里

$dp[i]$ 表示 i 之前包括 i 的以 $nums[i]$ 结尾的最长递增子序列的长度

实现代码:

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        if (nums.size() <= 1) return nums.size();
        vector<int> dp(nums.size(), 1);
        int result = 0;
        for (int i = 1; i < nums.size(); i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        return result;
    }
};
```

```

        if (dp[i] > result) result = dp[i]; // 取长的子序列
    }
    return result;
}
};

```

连续子序列

最长重复子数组问题

力扣原题

原题讲解在这里

$dp[i][j]$: 以下标 $i - 1$ 为结尾的A, 和以下标 $j - 1$ 为结尾的B, 最长重复子数组长度为 $dp[i][j]$

实现代码:

```

class Solution {
public:
    int findLength(vector<int>& nums1, vector<int>& nums2) {
        vector<vector<int>> dp (nums1.size() + 1, vector<int>(nums2.size() + 1,
0));
        int result = 0;

        for (int i = 1; i <= nums1.size(); i++) {
            for (int j = 1; j <= nums2.size(); j++) {
                if (nums1[i - 1] == nums2[j - 1]) { // 与上述最大递增子序列问题区分
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                }
                if (dp[i][j] > result) result = dp[i][j];
            }
        }
        return result;
    }
};

```

最大子数组问题

力扣原题

原题讲解在这里

$dp[i]$: 包括下标 i (以 $nums[i]$ 为结尾) 的最大连续子序列和为 $dp[i]$

实现代码:

```

class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        if (nums.size() == 0) return 0;

        vector<int> dp(nums.size());
        dp[0] = nums[0];
        int result = dp[0];

        for (int i = 1; i < nums.size(); i++) {
            dp[i] = max(dp[i - 1] + nums[i], nums[i]); // 状态转移公式
            if (dp[i] > result) result = dp[i]; // result 保存dp[i]的最大值
        }
        return result;
    }
};

```

编辑距离

==重点看l[i-1]和r[j-1]两个序列最新元素是否相等==

判断子序列问题

力扣原题

[原题讲解在这里](#)

==dp[i][j]表示以下标i-1为结尾的字符串s，和以下标j-1为结尾的字符串t，相同子序列的长度为dp[i][j]==

==注意这里判断的是s是否为t的子序列，即每次删除应该是删除t的元素 !!!引入了删除的概念==

实现代码:

```

class Solution {
public:
    bool isSubsequence(string s, string t) {
        vector<vector<int>> dp(s.size() + 1, vector<int>(t.size() + 1, 0));
        for (int i = 1; i <= s.size(); i++) {
            for (int j = 1; j <= t.size(); j++) {
                if (s[i - 1] == t[j - 1]) dp[i][j] = dp[i - 1][j - 1] + 1;
                else dp[i][j] = dp[i][j - 1];
            }
        }
        if (dp[s.size()][t.size()] == s.size()) return true;
        return false;
    }
};

```

```
    }  
};
```

不同子序列问题

力扣原题

[原题讲解在这里](#)

$dp[i][j]$: 以 $i-1$ 为结尾的 s 子序列中出现以 $j-1$ 为结尾的 t 的个数为 $dp[i][j]$

当 $s[i-1]$ 与 $t[j-1]$ 相等时, $dp[i][j]$ 可以有两部分组成。

- 一部分是用 $s[i-1]$ 来匹配, 那么个数为 $dp[i-1][j-1]$ 。即不需要考虑当前 s 子串和 t 子串的最后一位字母, 所以只需要 $dp[i-1][j-1]$ 。
- 一部分是不用 $s[i-1]$ 来匹配, 个数为 $dp[i-1][j]$ 。

当 $s[i-1]$ 与 $t[j-1]$ 不相等时, $dp[i][j]$ 只有一部分组成, 不用 $s[i-1]$ 来匹配 (就是模拟在 s 中删除这个元素), 即: $dp[i-1][j]$

实现代码:

```
class Solution {  
public:  
    int numDistinct(string s, string t) {  
        vector<vector<uint64_t>> dp(s.size() + 1, vector<uint64_t>(t.size() + 1));  
        for (int i = 0; i < s.size(); i++) dp[i][0] = 1;  
        for (int j = 1; j < t.size(); j++) dp[0][j] = 0;  
        for (int i = 1; i <= s.size(); i++) {  
            for (int j = 1; j <= t.size(); j++) {  
                if (s[i - 1] == t[j - 1]) {  
                    dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];  
                } else {  
                    dp[i][j] = dp[i - 1][j];  
                }  
            }  
        }  
        return dp[s.size()][t.size()];  
    }  
};
```

删除字符问题

力扣原题

[原题讲解在这里](#)

$dp[i][j]$: 以 $i-1$ 为结尾的字符串 $word1$, 和以 $j-1$ 位结尾的字符串 $word2$, 想要达到相等, 所需要删除元素的最少次数==

实现代码:

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        vector<vector<int>> dp(word1.size() + 1, vector<int>(word2.size() + 1));
        for (int i = 0; i <= word1.size(); i++) dp[i][0] = i;
        for (int j = 0; j <= word2.size(); j++) dp[0][j] = j;
        for (int i = 1; i <= word1.size(); i++) {
            for (int j = 1; j <= word2.size(); j++) {
                if (word1[i - 1] == word2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    dp[i][j] = min(dp[i - 1][j] + 1, dp[i][j - 1] + 1); // 这里思考
                    // 此处加一
                }
            }
        }
        return dp[word1.size()][word2.size()];
    }
};
```

编辑距离问题

力扣原题

原题讲解在这里

$dp[i][j]$ 表示以下标 $i-1$ 为结尾的字符串 $word1$, 和以下标 $j-1$ 为结尾的字符串 $word2$, 最近编辑距离为 $dp[i][j]$ ==

基本思想:

```
if (word1[i - 1] == word2[j - 1])
    不操作
if (word1[i - 1] != word2[j - 1])
    增
    删
    换
```

实现代码:


```

class Solution {
public:
    int minDistance(string word1, string word2) {
        vector<vector<int>> dp(word1.size() + 1, vector<int>(word2.size() + 1, 0));
        for (int i = 0; i <= word1.size(); i++) dp[i][0] = i;
        for (int j = 0; j <= word2.size(); j++) dp[0][j] = j;
        for (int i = 1; i <= word1.size(); i++) {
            for (int j = 1; j <= word2.size(); j++) {
                if (word1[i - 1] == word2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                }
                else {
                    dp[i][j] = min({dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]})
+ 1;
                } // str1增加一个元素就相当于str2删除一个元素
            }
        }
        return dp[word1.size()][word2.size()];
    }
};

```

回文

回文子串问题

力扣原题

原题讲解在这里

==!!! dp[i][j] 表示[i, j]的字符串是否为回文子串==

实现代码:

```

class Solution {
public:
    int countSubstrings(string s) {
        vector<vector<bool>> dp(s.size(), vector<bool>(s.size(), false));
        int result = 0;
        for (int i = s.size() - 1; i >= 0; i--) { // 注意遍历顺序
            for (int j = i; j < s.size(); j++) {
                if (s[i] == s[j]) {
                    if (j - i <= 1) { // 情况一 和 情况二
                        result++;
                        dp[i][j] = true;
                    }
                    else if (dp[i + 1][j - 1]) { // 情况三
                        result++;
                        dp[i][j] = true;
                    }
                }
            }
        }
    }
};

```

```

        }
    }
    return result;
}
};

```

最长回文子序列问题

力扣原题

原题讲解在这里

==!!! 回文子序列不需要连续==

==dp[i][j]: 字符串s在[i,j]范围内最长的回文子序列的长度为dp[i][j]==

实现代码:

```

class Solution {
public:
    int longestPalindromeSubseq(string s) {
        vector<vector<int>> dp(s.size(), vector<int>(s.size(), 0));
        for (int i = 0; i < s.size(); i++) dp[i][i] = 1; // 初始化i=j时回文子串是自身，长度为一

        for (int i = s.size() - 1; i >= 0; i--) {
            for (int j = i + 1; j < s.size(); j++) { // 注意j从i+1开始遍历，前面已定义过j=1的情况，故j从i+1开始
                if (s[i] == s[j]) {
                    dp[i][j] = dp[i + 1][j - 1] + 2;
                } else {
                    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[0][s.size() - 1];
    }
};



```

第四节、记忆化搜索

记忆化搜索就是构建一个列表将计算过的数据储存起来，不用后续再计算，常与递归和动态规划一起使用

数据结构学习列表

- ☐ 栈与队列
- ☐ 排序算法
- ☐ 二叉树
- ☐ 二叉堆
- ☐ 静态查找
- ☐ 图
- ☐ 算法

==一点都没学呢吧    ==

```
#include<bits/stdc++.h>
using namespace std;

struct node{
    int data;
    node* left;
    node*right;
    node(int x):data(x),left(NULL),right(NULL){}
};

int k=-1;
int count=0;
node* in(const string &s){
    k++;
    node* tree=NULL;
    if(k < s.size()){
        if(s[k]!='#'){
            tree=new node(int(s[k]));
            tree->left=in(s);
            tree->right=in(s);
        }
        if (!tree->left && !tree->right) {
            count++;
        }
    }
    return tree;
}

void preorder(node* tree){
    cout<<tree->data;
    preorder(tree->left);
    preorder(tree->right);
}

void inorder(node*tree){
    inorder(tree->left);
    cout<<tree->data;
```

```
        inorder(tree->right);
    }

    void postorder(node*tree){
        postorder(tree->left);
        postorder(tree->right);
        cout<<tree->data;
    }

    int main(){
        string s;
        cin>>s;
        node* tree=in(s);

        preorder(tree);
        cout<<endl;
        inorder(tree);
        cout<<endl;
        postorder(tree);
        cout<<endl;
        cout<<count<<endl;
        return 0;
    }
```