```
string s;
getline(cin, s);
// 输出读入的字符串
cout << s << endl;

getchar();

vector<string> path(n, string(n, '.')); // 初始化方法
```

数组

二分查找

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int numSize = nums.size();
        int left = 0, right = numSize - 1;

        while(left <= right) {
            int pos = (right + left) / 2;

            if(nums[pos] == target) {
                return pos;
            }
            else if(nums[pos] > target) {
                right = pos - 1; // 正确更新 right
            }
            else {
                left = pos + 1; // 正确更新 left
            }
        }

        return -1; // 如果没有找到目标，则返回 -1
    }
};
```

移除元素

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int k = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] != val) {
                nums[k] = nums[i];
                k++;
            }
        }
        return k; // 数组中不含val元素的新长度
    }
};
```

有序数组的平方——双指针法

```
class Solution {
public:
    vector<int> sortedSquares(vector<int>& nums) {
        int left = 0, right = nums.size() - 1, k = nums.size() - 1;
        vector<int> result(nums.size(), 0);
        while(left <= right) {
            if(nums[left] * nums[left] < nums[right] * nums[right]) {
                result[k] = nums[right] * nums[right];
                k--;
                right--;
            }
            else if(nums[left] * nums[left] > nums[right] * nums[right]){
                result[k] = nums[left] * nums[left];
                k--;
                left++;
            }
            else {
                result[k] = nums[left] * nums[left];
                k--;
                if(k >=0 ){ // 最后一个会出现问题
                    result[k] = nums[left] * nums[left];
                    k--;
                }
                left++;
                right--;
            }
        }
        return result;
    }
};
```

长度最小的子数组——滑动窗口

```
class Solution {
public:
```

```
    int minSubArrayLen(int target, vector<int>& nums) {
        int left = 0, sum = 0;
        int cnt = INT_MAX; // 使用 INT_MAX 进行初始化
        for (int right = 0; right < nums.size(); right++) {
            sum += nums[right];
            while (sum >= target) { // 检查当前总和是否达到或超过目标值
                cnt = min(cnt, right - left + 1); // 更新 cnt 为当前 cnt 或当前子数组
长度的较小者
                sum -= nums[left++]; // 通过移除最左侧元素减少 sum，并增加 left
            }
        }
        return cnt == INT_MAX ? 0 : cnt; // 如果 cnt 未更新，则返回 0，否则返回 cnt
    }
};
```

螺旋矩阵

```
class Solution {
public:
    vector<vector<int>> generateMatrix(int n) {
        vector<vector<int>> res(n, vector<int>(n, 0));
        int cnt = 1, start = 0;
        for(int i = 0; i < (n+1) / 2; i++) {
            for(int j = start; j < n - 1 - start; j++) {
                res[start][j] = cnt++;
            }
            for(int j = start; j < n - 1 - start; j++) {
                res[j][n - 1 - start] = cnt++;
            }
            for(int j = start; j < n - 1 - start; j++) {
                res[n - 1 - start][n - j - 1] = cnt++;
            }
            for(int j = start; j < n - 1 - start; j++) {
                res[n - j - 1][start] = cnt++;
            }
            start++;
        }
        if(n % 2) res[start-1][start-1] = cnt;
        return res;
    }
};
```

哈希表

基础知识

常见的三种哈希结构

当我们想使用哈希法来解决问题的时候，我们一般会选择如下三种数据结构。

- 数组
- set （集合）
- map(映射)

这里数组就没啥可说的了，我们来看一下set。

在C++中，set 和 map 分别提供以下三种数据结构，其底层实现以及优劣如下表所示：

集合	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::set	红黑树	有序	否	否	O(log n)	O(log n)
std::multiset	红黑树	有序	是	否	O(logn)	O(logn)
std::unordered_set	哈希表	无序	否	否	O(1)	O(1)

std::unordered_set底层实现为哈希表，std::set 和std::multiset 的底层实现是红黑树，红黑树是一种平衡二叉搜索树，所以key值是有序的，但key不可以修改，改动key值会导致整棵树的错乱，所以只能删除和增加。

映射	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::map	红黑树	key有序	key不可重复	key不可修改	O(logn)	O(logn)
std::multimap	红黑树	key有序	key可重复	key不可修改	O(log n)	O(log n)
std::unordered_map	哈希表	key无序	key不可重复	key不可修改	O(1)	O(1)

std::unordered_map 底层实现为哈希表，std::map 和std::multimap 的底层实现是红黑树。同理，std::map 和std::multimap 的key也是有序的（这个问题也经常作为面试题，考察对语言容器底层的理解）。

当我们要使用集合来解决哈希问题的时候，优先使用unordered_set，因为它的查询和增删效率是最优的，如果需要集合是有序的，那么就用set，如果要求不仅有序还要有重复数据的话，那么就用multiset。

那么再来看一下map ，在map 是一个key value 的数据结构，map中，对key是有限制，对value没有限制的，因为key的存储方式使用红黑树实现的。

其他语言例如：java里的HashMap ，TreeMap 都是一样的原理。可以灵活贯通。

虽然std::set、std::multiset 的底层实现是红黑树，不是哈希表，std::set、std::multiset 使用红黑树来索引和存储，不过给我们的使用方式，还是哈希法的使用方式，即key和value。所以使用这些数据结构来解决映射问题的方法，我们依然称之为哈希法。 map也是一样的道理。

这里在说一下，一些C++的经典书籍上 例如STL源码剖析，说到了hash_set hash_map，这个与unordered_set，unordered_map又有什么关系呢？

实际上功能都是一样一样的， 但是unordered_set在C++11的时候被引入标准库了，而hash_set并没有，所以建议还是使用unordered_set比较好，这就好比一个是官方认证的，hash_set，hash_map 是C++11标准之前民间高手自发造的轮子。

有效的字母异位词

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        int m[26];
        memset(m, 0, sizeof(m));
        for(int i = 0; i < s.size(); i++) {
            m[s[i]-'a']++;
        }
        for(int i = 0; i < t.size(); i++) {
            m[t[i]-'a']--;
        }
        for(int i = 0;i < 26; i++) {
            if(m[i] != 0) {
                return false;
            }
        }
        return true;
    }
};
```

两个数组的交集 —— `unordered_set`

```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_set<int> res, num;
        vector<int> ret;
        for(int i : nums1) num.insert(i);
        for(int i : nums2) {
            auto index = num.find(i);
            if(index != num.end()) { // if(num.find(i) != num.end())
                res.insert(i);
            }
        }
        for(int i : res) ret.push_back(i);
        return ret;
    }
};
```

`unordered_set`也可以遍历

快乐数

```
class Solution {
public:
    int happy(int n) {
        int sum = 0;
        while (n) {
            sum += (n % 10) * (n % 10);
            n /= 10;
        }
```

```
        return sum;
    }

    bool isHappy(int n) {
        unordered_set<int> res;
        while (1) {
            n = happy(n);
            if (res.find(n) == res.end()) {
                res.insert(n);
                if (n == 1)
                    return true;
            }
            else {
                return false;
            }
        }
    }
};
```

这道题目看上去貌似一道数学问题，其实并不是！

题目中说了会 **无限循环**，那么也就是说**求和的过程中，sum会重复出现，这对解题很重要！**

正如：关于哈希表，你该了解这些！中所说，**当我们遇到了要快速判断一个元素是否出现集合里的时候，就要考虑哈希法了。**

所以这道题目使用哈希法，来判断这个sum是否重复出现，如果重复了就是return false， 否则一直找到sum为1为止。

判断sum是否重复出现就可以使用unordered_set。

还有一个难点就是求和的过程，如果对取数值各个位上的单数操作不熟悉的话，做这道题也会比较艰难。

类中函数写法

两数之和 —— `unordered_map`

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        std::unordered_map <int,int> map;
        for(int i = 0; i < nums.size(); i++) {
            // 遍历当前元素，并在map中寻找是否有匹配的key
            auto iter = map.find(target - nums[i]); // O(1)时间即可找到
            if(iter != map.end()) {
                return {iter->second, i};
            }
            // 如果没找到匹配对，就把访问过的元素和下标加入到map中
            map.insert(pair<int, int>(nums[i], i));
        }
        return {};
    }
};
```

四数相加II

```
class Solution {
public:
    int fourSumCount(vector<int>& nums1, vector<int>& nums2, vector<int>& nums3,
vector<int>& nums4) {
        unordered_map<int,int> m;
        int cnt = 0;
        for(int a : nums1) {
            for(int b : nums2) {
                m[a+b]++;
            }
        }
        for(int c : nums3) {
            for(int d : nums4) {
                if(m.find(-(c+d)) != m.end()) {
                    //cnt++; 错误的 可能有不止一种
                    cnt += m[-(c+d)];
                }
            }
        }
        return cnt;
    }
};
```

在C++中，当你声明一个`unordered_map`对象但不显式地初始化它时，它将自动调用默认构造函数进行初始化。对于`unordered_map<int, int>`这种类型的映射，所有的键值对默认都是初始化为零。这意味着当你尝试访问一个不存在的键时，`unordered_map`会自动插入这个键，并将对应的值初始化为0（对于整型`int`）。

所以，在你提供的代码中，当执行`umap[a + b]++;`操作时，如果`umap`中不存在键`a + b`，它会自动创建这个键，并将其值设为0，然后立即增加1。因此，`unordered_map`的这一特性使得你不需要事先初始化映射中的每个可能的键值对。

赎金信

```
class Solution {
public:
    bool canConstruct(string ransomNote, string magazine) {
        unordered_map<char,int> m;
        for(char ch : magazine) {
            m[ch]++;
        }
        for(char ch : ransomNote) {
           m[ch]--;
        }
        for(auto iter : m) {
            if(iter.second < 0) {
                return false;
            }
        }
        return true;
    }
```

```
};
```

三数之和

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> result;
        sort(nums.begin(), nums.end());
        // 找出a + b + c = 0
        // a = nums[i], b = nums[left], c = nums[right]
        for (int i = 0; i < nums.size(); i++) {
            // 排序之后如果第一个元素已经大于零，那么无论如何组合都不可能凑成三元组，直接返回结果就可以了
            if (nums[i] > 0) {
                return result;
            }
            // 错误去重a方法，将会漏掉-1,-1,2 这种情况
            /*
            if (nums[i] == nums[i + 1]) {
                continue;
            }
            */
            // 正确去重a方法
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            int left = i + 1;
            int right = nums.size() - 1;
            while (right > left) {
                // 去重复逻辑如果放在这里，0，0，0 的情况，可能直接导致 right<=left 了，从而漏掉了 0,0,0 这种三元组
                /*
                while (right > left && nums[right] == nums[right - 1]) right--;
                while (right > left && nums[left] == nums[left + 1]) left++;
                */
                if (nums[i] + nums[left] + nums[right] > 0) right--;
                else if (nums[i] + nums[left] + nums[right] < 0) left++;
                else {
                    result.push_back(vector<int>{nums[i], nums[left], nums[right]});
                    // 去重逻辑应该放在找到一个三元组之后，对b 和 c去重
                    while (right > left && nums[right] == nums[right - 1]) right--;
                    while (right > left && nums[left] == nums[left + 1]) left++;

                    // 找到答案时，双指针同时收缩
                    right--;
                    left++;
                }
            }

        }
        return result;
    }
```

```
};
```

双指针法

sort(nums.begin(), nums.end());

sort(a,a+10) 自定义cmp函数

四数之和

```
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> res;
        if(nums.size() < 4) return res;
        for(int i = 0; i < nums.size() - 3; i++) {
            if(nums[i] > target && nums[i] >= 0) break;
            if(i > 0 && nums[i] == nums[i-1]) continue;
            for(int j = i + 1; j < nums.size() -2; j++) {
                if(nums[i] + nums[j] > target && nums[i] + nums[j] >= 0) break;
                if(j > i + 1 && nums[j] == nums[j-1]) continue;
                int left = j + 1, right = nums.size() - 1;
                while(left < right) {
                    if((long)nums[i] + nums[j] + nums[left] + nums[right] >
target) {
                        right--;
                    }
                    else if((long)nums[i] + nums[j] + nums[left] + nums[right] <
target) {
                        left++;
                    }
                    else {
                        res.push_back({nums[i], nums[j], nums[left],
nums[right]});
                        while(right > left && nums[left] == nums[left + 1])
left++;
                        while(right > left && nums[right] == nums[right - 1])
right--;
                        right--;
                        left++;
                    }
                }
            }
        }
        return res;
    }
};
```

长度不足直接返回

字符串

反转字符串

```
class Solution {
public:
    void reverseString(vector<char>& s) {
        int cnt = (s.size() + 1) / 2;
        for(int i = 0; i < cnt; i++) {
            int left = i, right = s.size() - 1 -left;
            char x = s[left];
            s[left] = s[right];
            s[right] = x;
        }
    }
};
```

```
void reverseString(vector<char>& s) {
    for (int i = 0, j = s.size() - 1; i < s.size()/2; i++, j--) {
        swap(s[i],s[j]);
    }
}
```

`reverse(a, a+5);`

`reverse(str.begin(), str.end());`

反转字符串II

```
class Solution {
public:
    string reverseStr(string s, int k) {
        for (int i = 0; i < s.size(); i += (2 * k)) {
            // 1. 每隔 2k 个字符的前 k 个字符进行反转
            // 2. 剩余字符小于 2k 但大于或等于 k 个，则反转前 k 个字符
            if (i + k <= s.size()) {
                reverse(s.begin() + i, s.begin() + i + k );
            } else {
                // 3. 剩余字符少于 k 个，则将剩余字符全部反转。
                reverse(s.begin() + i, s.end());
            }
        }
        return s;
    }
};
```

`reverse`函数

翻转字符串里的单词

```
class Solution {
public:
```

```
    string reverseWords(string s) {
        stringstream ss;
        ss<<s;
        stack<string> stack;
        string str;
        while(ss>>str) {
            stack.push(str);
        }
        ss.clear();
        ss.str(""); //如果想清空 stringstream，必须使用 sstream.str(""); 方式
        while(!stack.empty()) {
            str = stack.top();
            ss<<str;
            stack.pop();
            if(!stack.empty()) {
                ss<<" ";
            }
        }
        return ss.str();
    }
};
```

stack.pop不返回元素（pop）

stringstream用法

字符串截取

右旋字符串

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    string s;
    cin >> n >> s;
    int len = s.size();
    cout << (s + s).substr(len - n, len) << '\n';
    return 0;
}
```

重复的子字符串 —— `string.find()`

```
class Solution {
public:
    void getNext(string& s, int next[]) {
        next[0] = 0; // 初始化
        int j = 0;
        for(int i = 1; i < s.size(); i++) {
            while(j > 0 && s[i] != s[j]) { // 是j>0而不是>=
                j = next[j-1];
            }
            if(s[i] == s[j]) {
```

```
                j++;
            }
            next[i] = j;
        }
    }
    bool repeatedSubstringPattern(string s) {
        if (s.size() == 0) {
            return false;
        }
        int len = s.size();
        int next[len];
        getNext(s, next);
        if(next[len - 1] != 0 && len % (len - next[len-1]) == 0) return true; //
next[len-1] == 0那么肯定满足后面条件
        else return false;
    }
};
```

KMP算法

find()函数返回出现的下标

栈和队列

有效的括号

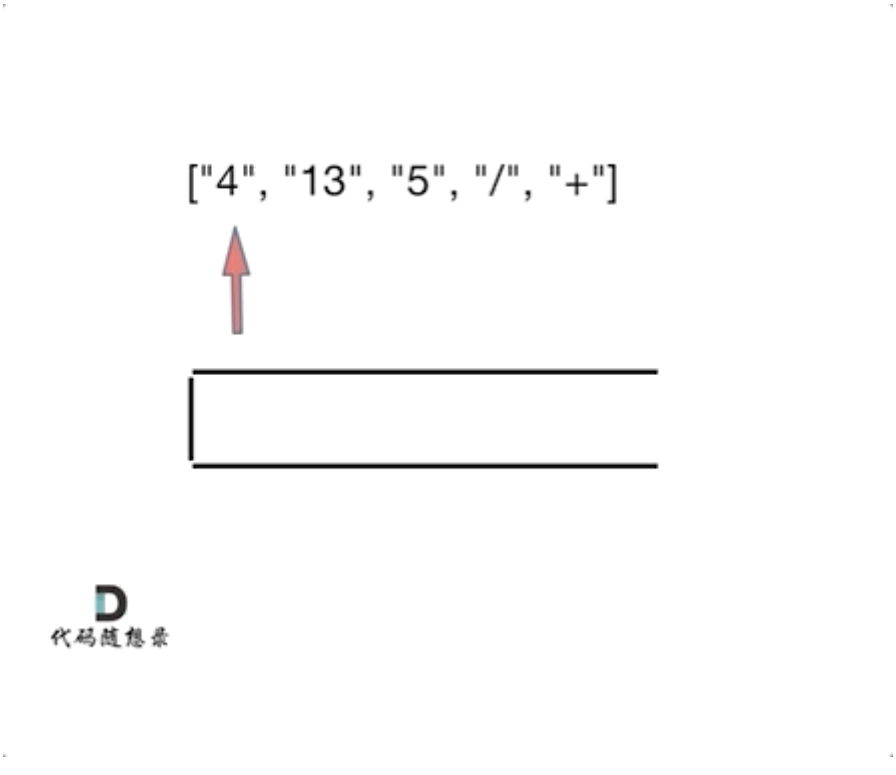
```
class Solution {
public:
    bool isValid(string s) {
        if (s.size() % 2 != 0) return false;
        stack<char> stack;
        for(int i = 0; i < s.length(); i++) {
            char ch = s[i];
            if(ch == '(' || ch == '[' || ch == '{') {
                stack.push(ch);
            }
            else {
                if(!stack.empty()){
                    if(ch == ')') {
                        if(stack.top() == '(') stack.pop();
                        else return false;
                    }
                    else if(ch == ']') {
                        if(stack.top() == '[') stack.pop();
                        else return false;
                    }
                    else {
                        if(stack.top() == '{') stack.pop();
                        else return false;
                    }
                }
                else return false;
            }
        }
        if(stack.empty()) return true;
```

```
        else return false;
    }
};
```

在匹配左括号的时候，右括号先入栈，就只需要比较当前元素和栈顶相不相等就可以了，比左括号先入栈代码实现要简单的多了！

```
class Solution {
public:
    bool isValid(string s) {
        if (s.size() % 2 != 0) return false; // 如果s的长度为奇数，一定不符合要求
        stack<char> st;
        for (int i = 0; i < s.size(); i++) {
            if (s[i] == '(') st.push(')');
            else if (s[i] == '{') st.push('}');
            else if (s[i] == '[') st.push(']');
            // 第三种情况：遍历字符串匹配的过程中，栈已经为空了，没有匹配的字符了，说明右括号没有找到对应的左括号 return false
            // 第二种情况：遍历字符串匹配的过程中，发现栈里没有我们要匹配的字符。所以return false
            else if (st.empty() || st.top() != s[i]) return false;
            else st.pop(); // st.top() 与 s[i]相等，栈弹出元素
        }
        // 第一种情况：此时我们已经遍历完了字符串，但是栈不为空，说明有相应的左括号没有右括号来匹配，所以return false，否则就return true
        return st.empty();
    }
};
```

逆波兰表达式求值



示例 1：

- 输入: ["2", "1", "+", "3", " * "]

- 输出: 9
- 解释: 该算式转化为常见的中缀算术表达式为：((2 + 1) * 3) = 9

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        // 力扣修改了后台测试数据，需要用longlong
        stack<long long> st;
        for (int i = 0; i < tokens.size(); i++) {
            if (tokens[i] == "+" || tokens[i] == "-" || tokens[i] == "*" ||
tokens[i] == "/") {
                long long num1 = st.top();
                st.pop();
                long long num2 = st.top();
                st.pop();
                if (tokens[i] == "+") st.push(num2 + num1);
                if (tokens[i] == "-") st.push(num2 - num1);
                if (tokens[i] == "*") st.push(num2 * num1);
                if (tokens[i] == "/") st.push(num2 / num1);
            } else {
                st.push(stoll(tokens[i]));
            }
        }

        int result = st.top();
        st.pop(); // 把栈里最后一个元素弹出（其实不弹出也没事）
        return result;
    }
};
```

`stoll(tokens[i])`：

- `stoll`：这是 C++ 标准库函数 `std::stoll`，用于将字符串转换为长整型（`long long`）。`stoll` 的全称是 `string to long long`。（`stoi`）

long long类型

num1 num2减除顺序

滑动窗口最大值 —— `multiset`

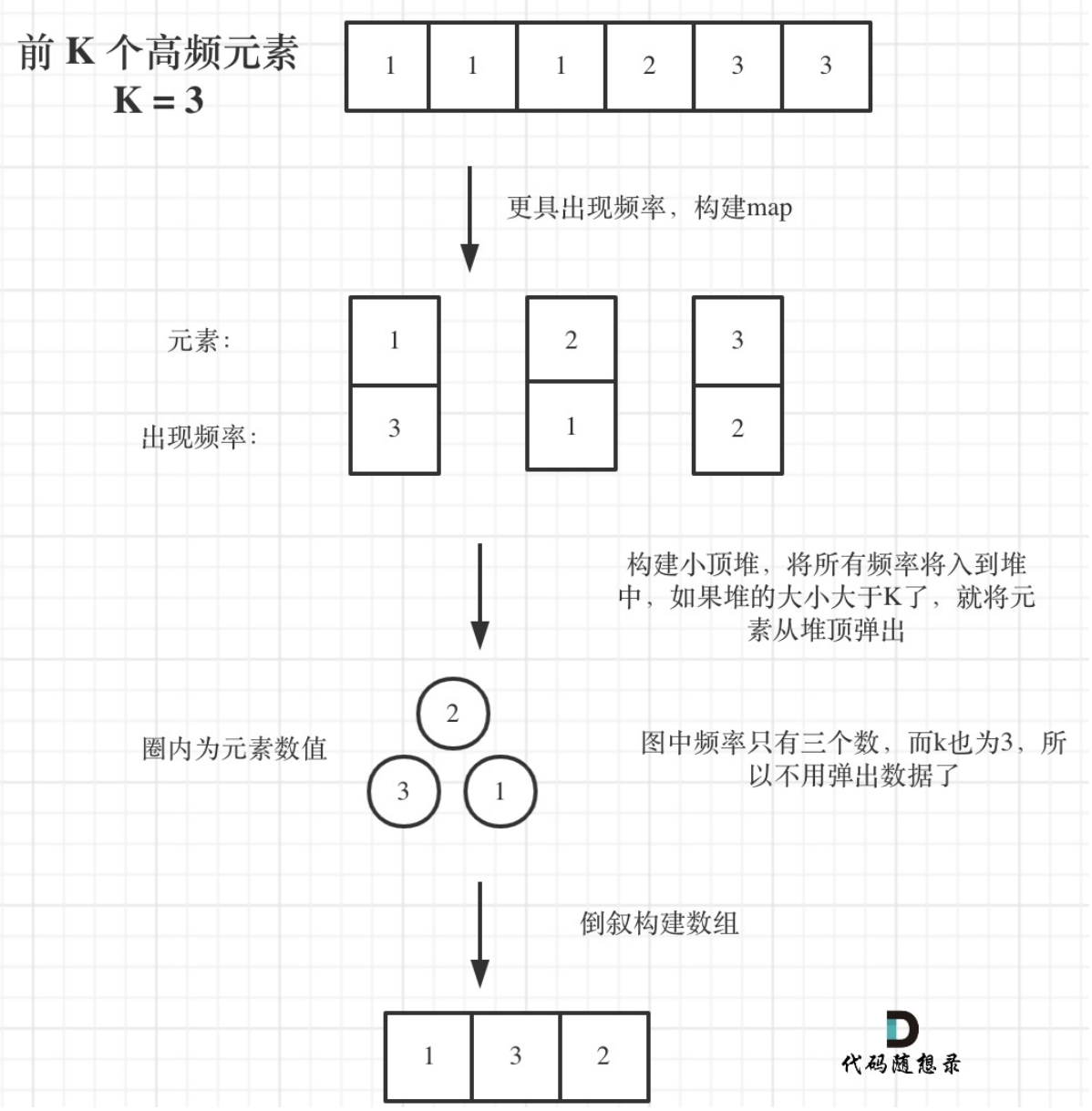
每次窗口移动的时候，调用que.pop(滑动窗口中移除元素的数值)，que.push(滑动窗口添加元素的数值)，然后que.front()就返回我们要的最大值。

```cpp
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        multiset<int> st;
        vector<int> ans;
        for (int i = 0; i < nums.size(); i++) {
            if (i >= k) st.erase(st.find(nums[i - k])); // 去除滑动窗口第一个元素
            st.insert(nums[i]);
            if (i >= k - 1) ans.push_back(*st.rbegin()); // 将滑动窗口最大值入栈
        }
        return ans;
    }
};
```

`multiset`

`*st.rbegin()`

前 K 个高频元素 —— `priority_queue` `unordered_map`



大家对这个比较运算在建堆时是如何应用的，为什么左大于右就会建立小顶堆，反而建立大顶堆比较困惑。

确实 例如我们在写快排的cmp函数的时候，`return left>right` 就是从大到小，`return left<right` 就是从小到大。

优先级队列的定义正好反过来了，可能和优先级队列的源码实现有关（我没有仔细研究），我估计是底层实现上优先队列队首指向后面，队尾指向最前面的缘故！

```
class Solution {
public:
    struct comparison {
        bool operator()(pair<int,int> p1, pair<int,int> p2) {
            return p1.second > p2.second;
        }
    };

    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int,int> m;
        for(int i : nums) {
            m[i]++;
        }
        priority_queue<pair<int, int>, vector<pair<int, int>>, comparison>
pri_que;

        for(auto p : m) {
            pri_que.push(p);
            if(pri_que.size() > k) pri_que.pop();
        }
        vector<int> ret;
        ret.resize(k);
        for (int i = k - 1; i >= 0; i--) {
            ret[i] = pri_que.top().first;
            pri_que.pop();
        }
        return ret;
    }
};
```

类中类（比较函数）

最大堆最小堆建立方法

二叉树

翻转二叉树

这里帮助大家确定下来递归算法的三个要素。**每次写递归，都按照这三要素来写，可以保证大家写出正确的递归算法！**

1. **确定递归函数的参数和返回值：** 确定哪些参数是递归的过程中需要处理的，那么就在递归函数里加上这个参数， 并且还要明确每次递归的返回值是什么进而确定递归函数的返回类型。

2. **确定终止条件：** 写完了递归算法, 运行的时候，经常会遇到栈溢出的错误，就是没写终止条件或者终止条件写的不对，操作系统也是用一个栈的结构来保存每一层递归的信息，如果递归没有终止，操作系统的内存栈必然就会溢出。
3. **确定单层递归的逻辑：** 确定每一层递归需要处理的信息。在这里也就会重复调用自己来实现递归的过程。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (root == NULL) return root;
        swap(root->left,root->right);
        invertTree(root->left);         // 左
        invertTree(root->right);        // 右
        return root;
    }
};
```

回溯法

回溯算法模板框架如下：

```
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
    }
}
```

组合

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
```

```
    void traceback(int n, int k, int deepth) {
        if(path.size() == k) {
            res.push_back(path);
            return;
        }
        for(int i = deepth; i <= n; i++) { // 所以是<=
            path.push_back(i);
            traceback(n, k, i+1);
            path.pop_back(); // 复原
        }
    }

    vector<vector<int>> combine(int n, int k) {
        traceback(n, k, 1); //要求从1开始
        return res;
    }
};
```

组合总和III

每个数字最多使用一次

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void traceback(int k, int n, int sum, int deepth) {
        if(path.size() == k) {
            if(sum == n) {
                res.push_back(path);
                return;
            }
        }
        for(int i = deepth; i <= 9; i++) {
            if(sum < n) {
                sum += i;
                path.push_back(i);
                traceback(k, n, sum, i+1);
                path.pop_back();
                sum -= i;
            }
        }
    }
    vector<vector<int>> combinationSum3(int k, int n) {
        traceback(k, n, 0, 1);
        return res;
    }
};
```

电话号码的字母组合

```
class Solution {
public:
    const string strMap[10] = {
```

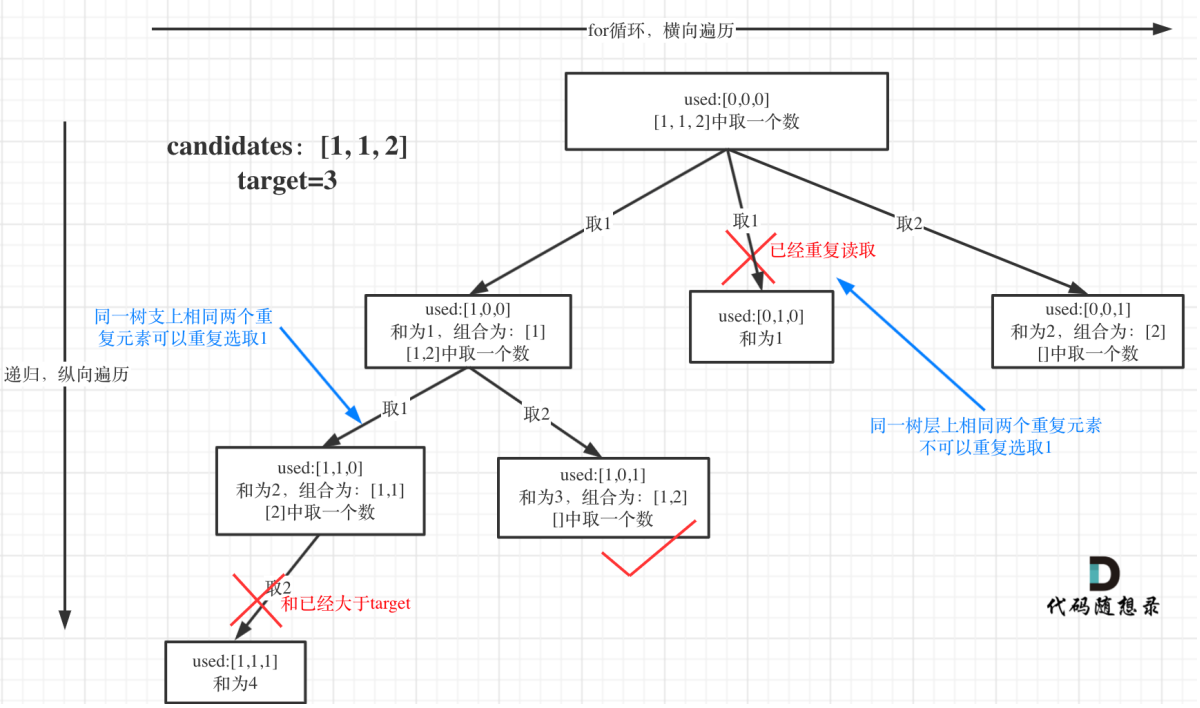
```
        "", // 0
        "", // 1
        "abc", // 2
        "def", // 3
        "ghi", // 4
        "jkl", // 5
        "mno", // 6
        "pqrs", // 7
        "tuv", // 8
        "wxyz", // 9
    };
    vector<string> res;
    string s;
    void tracebak(string digits, int deepth) {
        if(deepth == digits.size()) {
            res.push_back(s);
            return;
        }
        int num = digits[deepth] - '0';
        string str = strMap[num];
        for(int i = 0; i < str.size(); i++) {
            s.push_back(str[i]);
            tracebak(digits, deepth + 1);
            s.pop_back();
        }
    }

    vector<string> letterCombinations(string digits) {
        if(digits.size() == 0) return res;
        tracebak(digits, 0);
        return res;
    }
};
```

组合总和II

```
class Solution {
public:
    vector<int> path;
    vector<vector<int>> ret;
    void backtrace(vector<int>& candidates, int target, int sum, int deepth) {
        if (sum > target)
            return;
        else if (sum == target) {
            ret.push_back(path);
            return;
        } else {
            for (int i = deepth; i < candidates.size() && sum + candidates[i] <=
target; i++) { // 加了剪枝
                if (i > deepth && candidates[i] == candidates[i - 1])
                    continue; // 同一层树不能选两个相同的candidates
                sum += candidates[i];
                path.push_back(candidates[i]);
                backtrace(candidates, target, sum, i + 1);
                path.pop_back();
```

```
                sum -= candidates[i];
            }
        }
    }
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        sort(candidates.begin(), candidates.end());
        backtrace(candidates, target, 0, 0);
        return ret;
    }
};
```



分割回文串

```
class Solution {
public:
    vector<vector<string>> res;
    vector<string> path;
    bool ishuiwen (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;
    }
    void backtrace(string s, int deepth) {
        if(deepth >= s.size()) {
            res.push_back(path);
            return;
        }
        for(int i = deepth; i < s.size(); i++) {
            if(ishuiwen(s, deepth, i) == 1) {
                string str = s.substr(deepth, i - deepth + 1);
                path.push_back(str);
                backtrace(s, i + 1);
                path.pop_back();
            }
```

```
            else {
                //return;
                continue;
            }
        }
    }
    vector<vector<string>> partition(string s) {
        backtrace(s, 0);
        return res;
    }
};
```

`substr`用法 `s.substr(deepth, i - deepth + 1);` 起点 长度

`continue`和`return`

复原IP地址

```
class Solution {
public:
    vector<string> res;
    vector<string> restoreIpAddresses(string s) {
        int len = s.length();
        string s1, s2, s3, s4;
        if (len > 12) return res; // 不加会报错 std::out_of_range what():  stoi
        for (int i = 1; i < len - 2; i++) {
            for (int j = i + 1; j < len - 1; j++) {
                for (int k = j + 1; k < len; k++) {
                    s1 = s.substr(0, i);
                    s2 = s.substr(i, j - i);
                    s3 = s.substr(j, k - j);
                    s4 = s.substr(k, len - k);
                    int n1 = stoi(s1), n2 = stoi(s2), n3 = stoi(s3),
                        n4 = stoi(s4);
                    if ((s1.length() > 1 && s1[0] == '0') ||
                        (s2.length() > 1 && s2[0] == '0') ||
                        (s3.length() > 1 && s3[0] == '0') ||
                        (s4.length() > 1 && s4[0] == '0'))
                        continue;
                    if (n1 > 255 || n2 > 255 || n3 > 255 || n4 > 255)
                        continue;
                    else {
                        string str;
                        str.append(s1);
                        str.append(".");
                        str.append(s2);
                        str.append(".");
                        str.append(s3);
                        str.append(".");
                        str.append(s4);
                        res.push_back(str);
                    }
                }
            }
        }
```

```
        return res;
    }
};
```

`stoi`函数

`string`字符串拼接

子集、子集II

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void backtrack(vector<int>& nums, int deepth) {
        res.push_back(path);
        if (deepth >= nums.size()) {
            // res.push_back(path);
            return;
        }
        for (int i = deepth; i < nums.size(); i++) {
            if (i > deepth && nums[i] == nums[i - 1]) {
                continue;
            }
            path.push_back(nums[i]);
            backtrack(nums, i + 1);
            path.pop_back();
        }
    }
    vector<vector<int>> subsets(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        backtrack(nums, 0);
        return res;
    }
};
```

要清楚子集问题和组合问题、分割问题的的区别：

- **子集是收集树形结构中树的所有节点的结果**（所以`res.push_back(path);`在开头）
- **而组合问题、分割问题是收集树形结构中叶子节点的结果**（最底下）

非递减子序列

```
class Solution {
public:
    vector<int> path;
    vector<vector<int>> res;
    void backtrack(vector<int>& nums, int deepth) {
        if(path.size() > 1) {
            res.push_back(path);
            //return; 注意这里不要加return，要取树上的节点
        }
        unordered_set<int> used; // 不 sort 则加 unordered_set
        for(int i = deepth; i < nums.size(); i++) {
```

```
            if(i > deepth && used.find(nums[i]) != used.end()) {
                continue;
            }
            if(!path.empty() && nums[i] < path.back()) continue;
            used.insert(nums[i]);
            path.push_back(nums[i]);
            backtrack(nums, i + 1);
            path.pop_back();
        }
    }
    vector<vector<int>> findSubsequences(vector<int>& nums) {
        backtrack(nums, 0);
        return res;
    }
};
```

N皇后

```
class Solution {
public:
    vector<vector<string>> res;
    bool isValid(int row, int col, int n, vector<string>& path) {
        for(int i = 0; i < row; i++) {
            if(path[i][col] == 'Q') return false;
        }
        // 检查 45度角是否有皇后 斜下
        for (int i = row - 1, j = col - 1; i >=0 && j >= 0; i--, j--) {
            if (path[i][j] == 'Q') {
                return false;
            }
        }
        // 检查 135度角是否有皇后 斜上
        for(int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
            if (path[i][j] == 'Q') {
                return false;
            }
        }
        return true;
    }
    void backtrack(int n, int row, vector<string>& path) { // row为行 col为列
        if(row == n) {
            res.push_back(path);
            return;
        }
        for(int col = 0; col < n; col++) {
            if(isValid(row, col, n, path)) {
                path[row][col] = 'Q';
                backtrack(n, row + 1, path);
                path[row][col] = '.';
            }
        }
    }
    vector<vector<string>> solveNQueens(int n) {
        vector<string> path(n, string(n, '.')); // 初始化方法
        backtrack(n, 0, path);
```

```
        return res;
    }
};
```

`vector<string> path(n, string(n, '.'));` // 初始化方法

总结

回溯算法能解决如下问题：

- 组合问题：N个数里面按一定规则找出k个数的集合
- 排列问题：N个数按一定规则全排列，有几种排列方式
- 切割问题：一个字符串按一定规则有几种切割方式
- 子集问题：一个N个数的集合里有多少符合条件的子集
- 棋盘问题：N皇后，解数独等等

贪心算法

跳跃游戏

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        if(nums.size() == 1) return true;
        int cover = 0;
        for(int i = 0; i < nums.size() - 1; i++) {
            cover = max(i + nums[i], cover);
            if(cover < i + 1) return false;
        }
        if(cover >= nums.size() - 1) return true;
        else return false;
    }
};
```

跳跃游戏 II

```
class Solution {
public:
    int jump(vector<int>& nums) {
        if(nums.size() == 1) return 0;
        int cnt = 0, curcover = 0, nextcover = 0; // curcover为跳cnt步可到达的最大值
        for(int i = 0; i < nums.size() - 1; i++) {
            nextcover = max(i + nums[i], nextcover);
            if(curcover == i) {
                cnt++;
                curcover = nextcover;
                if(curcover >= nums.size() - 1) {
                    return cnt;
                }
            }
        }
        return -1; // 出错
```

```
    }
};
```

用最少数量的箭引爆气球 —— 自定义排序

```
class Solution {
public:
    static bool cmp(const vector<int>& a, const vector<int>& b) {
        return a[1] < b[1];
    }
    int findMinArrowShots(vector<vector<int>>& points) {
        sort(points.begin(), points.end(), cmp);
        int cnt = 1, right = points[0][1];
        for(int i = 1; i < points.size(); i++) {
            if(right >= points[i][0]) {
                continue;
            }
            else {
                cnt++;
                right = points[i][1];
            }
        }
        return cnt;
    }
};
```

无重叠区间

```
class Solution {
public:
    static bool cmp(vector<int>& a1, vector<int>& a2) {
        if(a1[1] != a2[1]) return a1[1] < a2[1];
        else return a1[0] < a2[0];
    }
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end(), cmp);
        int right = intervals[0][1], cnt = 0;
        for(int i = 1; i < intervals.size(); i++) {
            if(right > intervals[i][0]) {
                cnt++;
            }
            else {
                right = intervals[i][1];
            }
        }
        return cnt;
    }
};
```

划分字母区间

```
class Solution {
public:
```

```
    vector<int> partitionLabels(string s) {
        int a[26];
        vector<int> ret;
        memset(a, 0 ,sizeof(a));
        for(int i = 0; i < s.length(); i++) {
            a[s[i] - 'a'] = i;
        }
        int minnum = a[s[0] - 'a'], left = 0;
        for(int i = 0; i < s.length(); i++) {
            minnum = max(minnum, a[s[i] - 'a']);
            if(minnum == i) {
                ret.push_back(minnum - left + 1);
                left = minnum + 1;
                if(i < s.length() - 1) minnum = a[s[i + 1] -'a'];
            }
        }
        return ret;
    }
};
```

最大子序和

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxVal = nums[0], res = nums[0];
        if(nums.size() == 1) return res;
        for(int i = 1; i < nums.size(); i++) {
            if(maxVal > 0) {
                maxVal += nums[i];
                res = max(res, maxVal);
            }
            else {
                maxVal = nums[i];
                res = max(res, maxVal);
            }
        }
        return res;
    }
};
```

加油站

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        for(int i = 0; i < gas.size(); i++) {
            int leftgas = gas[i];
            for(int j = 0; j < gas.size(); j++) {
                if(j == gas.size() - 1 && leftgas - cost[(i + j) % gas.size()] >=
0) return i;
                if(leftgas - cost[(i + j) % gas.size()] <= 0) {
                    break;
                }
```

```
                else {
                    leftgas = leftgas - cost[(i + j) % gas.size()] + gas[(i + j +
1) % gas.size()];
                }

            }
        }
        return -1;
    }
};
```

到最后一个加油站油可以为0

非最后一个加油站即使不需要油，没油也不能走

暴力法

监控二叉树

```
class Solution {
public:
    int res = 0;
    int trave(TreeNode* cur) {
        if(cur == nullptr) return 2;
        int left = trave(cur->left);    // 左
        int right = trave(cur->right);  // 右
        if (left == 2 && right == 2) return 0;
        if (left == 0 || right == 0) {
            res++;
            return 1;
        }
        if (left == 1 || right == 1) return 2;
        return -1;
    }
    int minCameraCover(TreeNode* root) {
        if (trave(root) == 0) { // root 无覆盖
            res++;
        }
        return res;
    }
};
```

从叶节点向根节点寻找

递归

单调栈

每日温度

```
class Solution {
public:
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        vector<int> res(temperatures.size(), 0);
```

```
        stack<int> st;
        for(int i = 0; i < temperatures.size(); i++) {
            if(st.empty()) {
                st.push(i);
            }
            else {
                if(temperatures[i] <= temperatures[st.top()]) {
                    st.push(i);
                }
                else {
                    while(!st.empty() && temperatures[i] >
temperatures[st.top()]) { // 一直用i匹配，可能是很多个的最大值
                        res[st.top()] = i - st.top();
                        st.pop();
                    }
                    st.push(i);
                }
            }
        }
        return res;
    }
};
```

接雨水

```
class Solution {
public:
    int trap(vector<int>& height) {
        int sum = 0;
        if(height.size() <= 2) return 0;
        int size = height.size();
        vector<int> maxLeft(size, 0);
        vector<int> maxRight(size, 0);
        maxLeft[0] = height[0];
        for(int i = 1; i < size; i++) {
            maxLeft[i] = max(height[i], maxLeft[i - 1]);
        }
        maxRight[size - 1] = height[size - 1];
        for(int i = size - 2; i >= 0; i--) {
            maxRight[i] = max(height[i], maxRight[i + 1]);
        }
        for(int i = 1; i < size; i++) {
            int cnt = min(maxLeft[i], maxRight[i]) - height[i]; // 主要代码
            if(cnt > 0) sum += cnt;
        }
        return sum;
    }
};
```

双指针法

图论

理论知识

dfs的代码框架：

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

    for (选择：本节点所连接的其他节点) {
        处理节点;
        dfs(图，选择的节点); // 递归
        回溯，撤销处理结果
    }
}
```

广搜的搜索方式就适合于解决两个点之间的最短路径问题。

因为广搜是从起点出发，以起始点为中心一圈一圈进行搜索，一旦遇到终点，记录之前走过的节点就是一条最短路。

当然，也有一些问题是广搜 和 深搜都可以解决的，例如岛屿问题，**这类问题的特征就是不涉及具体的遍历方式，只要能把相邻且相同属性的节点标记上就行。**（我们会在具体题目讲解中详细来说）

仅仅需要一个容器，能保存我们要遍历过的元素就可以，**那么用队列，还是用栈，甚至用数组，都是可以的。**

用队列的话，就是保证每一圈都是一个方向去转，例如统一顺时针或者逆时针。

因为队列是先进先出，加入元素和弹出元素的顺序是没有改变的。

如果用栈的话，就是第一圈顺时针遍历，第二圈逆时针遍历，第三圈有顺时针遍历。

因为栈是先进后出，加入元素和弹出元素的顺序改变了。

那么广搜需要注意 转圈搜索的顺序吗？ 不需要！

所以用队列，还是用栈都是可以的，但大家都习惯用队列了，**所以下面的讲解用我也用队列来讲，只不过要给大家说清楚，并不是非要用队列，用栈也可以。**

岛屿数量

深搜：

```
class Solution {
private:
    int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1}; // 四个方向
    void dfs(vector<vector<char>>& grid, vector<vector<bool>>& visited, int x,
int y) {
        for (int i = 0; i < 4; i++) {
            int nextx = x + dir[i][0]; // 每一次都是重新定义计算 所以不需要撤销
            int nexty = y + dir[i][1];
```

```
            if (nextx < 0 || nextx >= grid.size() || nexty < 0 || nexty >=
grid[0].size()) continue;  // 越界了，直接跳过
            if (!visited[nextx][nexty] && grid[nextx][nexty] == '1') { // 没有访问
过的 同时 是陆地的

                visited[nextx][nexty] = true;
                dfs(grid, visited, nextx, nexty);
            }
        }
    }
public:
    int numIslands(vector<vector<char>>& grid) {
        int n = grid.size(), m = grid[0].size();
        vector<vector<bool>> visited = vector<vector<bool>>(n, vector<bool>(m,
false));

        int result = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (!visited[i][j] && grid[i][j] == '1') {
                    visited[i][j] = true;
                    result++; // 遇到没访问过的陆地，+1
                    dfs(grid, visited, i, j); // 将与其链接的陆地都标记上 true
                }
            }
        }
        return result;
    }
};
```

广搜:

```
class Solution {
public:
    vector<vector<bool>> visited;
    int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1}; // 四个方向
    int numIslands(vector<vector<char>>& grid) {
        int n = grid.size(), m = grid[0].size();
        visited = vector<vector<bool>>(n, vector<bool>(m, false));
        int result = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (!visited[i][j] && grid[i][j] == '1') {
                    //visited[i][j] = true;
                    result++; // 遇到没访问过的陆地，+1
                    bfs(grid, i, j); // 将与其链接的陆地都标记上 true
                }
            }
        }
        return result;
    }
    void bfs(vector<vector<char>>& grid, int x, int y) {
        queue<pair<int, int>> q;
        q.push({x, y});
        visited[x][y] = true; // 只要加入立刻标记
```

```
        while(!q.empty()) {
            pair<int, int> cur = q.front();
            q.pop();
            int curx = cur.first;
            int cury = cur.second;
            visited[curx][cury] = true; // 从队列中取出在标记走过
            for (int i = 0; i < 4; i++) {
                int nextx = curx + dir[i][0];
                int nexty = cury + dir[i][1];
                if (nextx < 0 || nextx >= grid.size() || nexty < 0 || nexty >=
grid[0].size()) continue;  // 越界了，直接跳过
                if (!visited[nextx][nexty] && grid[nextx][nexty] == '1') {
                    q.push({nextx, nexty});
                    visited[nextx][nexty] = true; // 只要加入立刻标记
                }
            }
        }
    }
};
```

最大人工岛

```
class Solution {
public:
    int dir[4][2] = {0, 1, 1, 0, -1, 0, 0, -1};
    int cnt;
    void dfs(vector<vector<int>>& grid, int x, int y, int mark) {
        if (grid[x][y] != 1) return;
        grid[x][y] = mark;
        cnt++;
        for (int i = 0; i < 4; i++) {
            int nextx = x + dir[i][0];
            int nexty = y + dir[i][1];
            if (nextx < 0 || nextx >= grid.size() || nexty < 0 || nexty >=
grid[0].size())
                continue; // 越界了，直接跳过
            dfs(grid, nextx, nexty, mark);
        }
    }
    int largestIsland(vector<vector<int>>& grid) {
        int n = grid.size(), m = grid[0].size();
        unordered_map<int, int> gridNum;
        int mark = 2;
        bool isAllGrid = true; // 标记是否整个地图都是陆地
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (grid[i][j] == 0)
                    isAllGrid = false;
                if (grid[i][j] == 1) {
                    cnt = 0;
                    dfs(grid, i, j, mark);
                    gridNum[mark] = cnt;
                    mark++;
                }
            }
```

```
        }
        if (isAllGrid) return n * m;
        int result = 0;                  // 记录最后结果
        unordered_set<int> visitedGrid; // 标记访问过的岛屿
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                int count = 1;       // 记录连接之后的岛屿数量
                visitedGrid.clear(); // 每次使用时，清空
                if (grid[i][j] == 0) {
                    for (int k = 0; k < 4; k++) {
                        int neari = i + dir[k][1]; // 计算相邻坐标
                        int nearj = j + dir[k][0];
                        if (neari < 0 || neari >= grid.size() || nearj < 0 ||
nearj >= grid[0].size())
                            continue;
                        if (visitedGrid.count(grid[neari][nearj]))
                            continue; // 添加过的岛屿不要重复添加
                        // 把相邻四面的岛屿数量加起来
                        count += gridNum[grid[neari][nearj]];
                        visitedGrid.insert(grid[neari][nearj]); // 标记该岛屿已经添
加过
                    }
                }
                result = max(result, count);
            }
        }
        return result;
    }
};
```

单词接龙

求起点和终点的最短路径长度，**这里无向图求最短路，广搜最为合适，广搜只要搜到了终点，那么一定是最短的路径。**因为广搜就是以起点中心向四周扩散的搜索。

本题如果用深搜，会比较麻烦，要在到达终点的不同路径中选则一条最短路。 而广搜只要达到终点，一定是最短路。

另外需要有一个注意点：

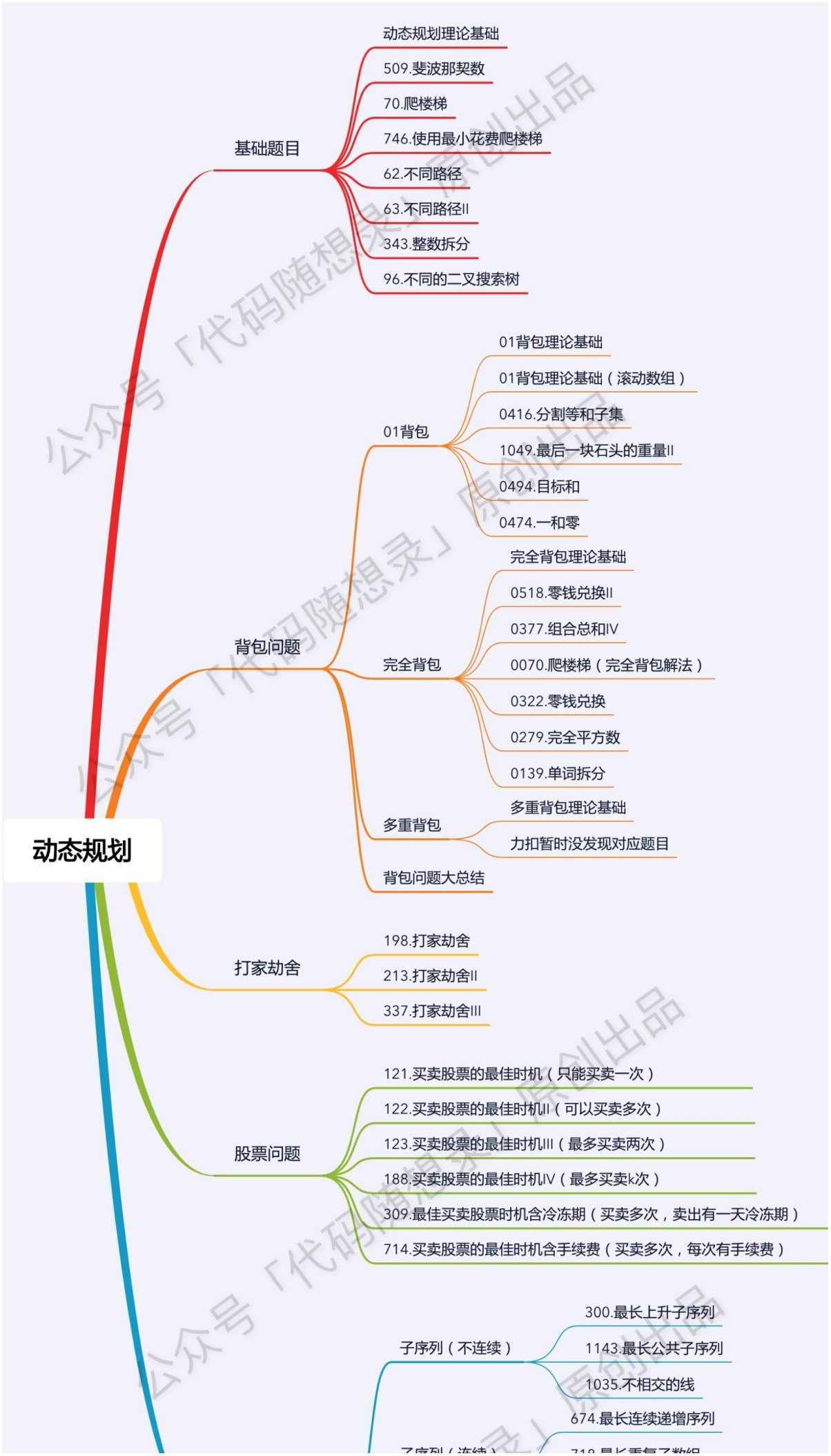
- 本题是一个无向图，需要用标记位，标记着节点是否走过，否则就会死循环！
- 本题给出集合是数组型的，可以转成set结构，查找更快一些

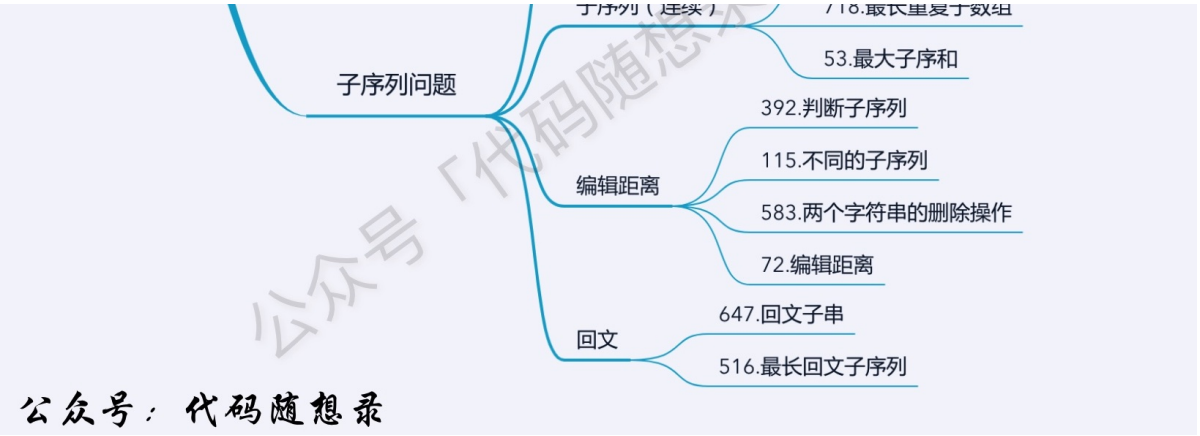
```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, vector<string>& wordList)
{
        // 将vector转成unordered_set，提高查询速度
        unordered_set<string> wordSet(wordList.begin(), wordList.end());
        // 如果endWord没有在wordSet出现，直接返回0
        if (wordSet.find(endWord) == wordSet.end()) return 0;
        // 记录word是否访问过
        unordered_map<string, int> visitMap; // <word, 查询到这个word路径长度>
        // 初始化队列
        queue<string> que;
```

```
        que.push(beginWord);
        // 初始化visitMap
        visitMap.insert(pair<string, int>(beginWord, 1));

        while(!que.empty()) {
            string word = que.front();
            que.pop();
            int path = visitMap[word]; // 这个word的路径长度
            for (int i = 0; i < word.size(); i++) {
                string newWord = word; // 用一个新单词替换word，因为每次置换一个字母
                for (int j = 0 ; j < 26; j++) {
                    newWord[i] = j + 'a';
                    if (newWord == endWord) return path + 1; // 找到了end，返回
path+1
                    // wordSet出现了newWord，并且newWord没有被访问过
                    if (wordSet.find(newWord) != wordSet.end()
                            && visitMap.find(newWord) == visitMap.end()) {
                        // 添加访问信息
                        visitMap.insert(pair<string, int>(newWord, path + 1));
                        que.push(newWord);
                    }
                }
            }
        }
        return 0;
    }
};
```

动态规划


动态规划
基础题目
动态规划理论基础
509.斐波那契数
70.爬楼梯
746.使用最小花费爬楼梯
62.不同路径
63.不同路径II
343.整数拆分
96.不同的二叉搜索树
背包问题
01背包
01背包理论基础
01背包理论基础（滚动数组）
0416.分割等和子集
1049.最后一块石头的重量II
0494.目标和
0474.一和零
完全背包
完全背包理论基础
0518.零钱兑换II
0377.组合总和IV
0070.爬楼梯（完全背包解法）
0322.零钱兑换
0279.完全平方数
0139.单词拆分
多重背包
多重背包理论基础
力扣暂时没发现对应题目
背包问题大总结
打家劫舍
198.打家劫舍
213.打家劫舍II
337.打家劫舍III
股票问题
121.买卖股票的最佳时机（只能买卖一次）
122.买卖股票的最佳时机II（可以买卖多次）
123.买卖股票的最佳时机III（最多买卖两次）
188.买卖股票的最佳时机IV（最多买卖k次）
309.最佳买卖股票时机含冷冻期（买卖多次，卖出有一天冷冻期）
714.买卖股票的最佳时机含手续费（买卖多次，每次有手续费）
子序列（不连续）
300.最长上升子序列
1143.最长公共子序列
1035.不相交的线
674.最长连续递增序列



携带研究材料

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

int main() {
    int m, n; cin >> m >> n;
    int space[m], value[m];
    for(int i = 0; i < m; i++) {
        cin >> space[i];
    }
    for(int i = 0; i < m; i++) {
        cin >> value[i];
    }
    int dp[n + 1];
    memset(dp, 0, sizeof(dp));
    for(int i = 0; i < m; i++) { // 外层循环遍历每个类型的研究材料
        for (int j = n; j >= space[i]; --j) { // 内层循环从 N 空间逐渐减少到当前研究材料所占空间
            // 考虑当前研究材料选择和不选择的情况，选择最大值
            dp[j] = max(dp[j], dp[j - space[i]] + value[i]);
        }
    }
    cout << dp[n] << endl;
    return 0;
}
```

dp[j]为 容量为j的背包所背的最大价值

分割等和子集

```
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int sum = 0;
        for(int i : nums) {
            sum += i;
        }
        if(sum % 2) return false;
        int target = sum / 2;
        int dp[10001];
        memset(dp, 0 ,sizeof(dp));
```

```
        for(int i = 0; i < nums.size(); i++) {
            for(int j = target; j >= nums[i]; j--) {
                dp[j] = max(dp[j], dp[j - nums[i]] + nums[i]);
            }
        }
        if(dp[target] == target) return true;
        else return false;
    }
};
```

最后一块石头的重量II

本题其实就是尽量让石头分成重量相同的两堆，相撞之后剩下的石头最小，**这样就化解成01背包问题了。**

```
class Solution {
public:
    int lastStoneWeightII(vector<int>& stones) {
        int sum = 0;
        for(int i : stones) {
            sum += i;
        }
        int target = sum / 2;
        int dp[target + 1];
        memset(dp, 0 ,sizeof(dp));
        for(int i = 0; i < stones.size(); i++) {
            for(int j = target; j >= stones[i]; j--) {
                dp[j] = max(dp[j], dp[j - stones[i]] + stones[i]);
            }
        }
        return abs(2 * dp[target] - sum);
    }
};
```

目标和

如何转化为01背包问题呢。

假设加法的总和为x，那么减法对应的总和就是sum - x。

所以我们要求的是 x - (sum - x) = target

x = (target + sum) / 2

此时问题就转化为，装满容量为x的背包，有几种方法。

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum = 0;
        for(int i : nums) {
            sum += i;
        }
        sum += target;
        if(sum % 2 || sum < 0) return 0;
```

```
        sum /= 2;
        int dp[2001];
        memset(dp, 0, sizeof(dp));
        dp[0] = 1;
        for(int i = 0; i < nums.size(); i++) {
            for(int j = sum; j >= nums[i]; j--) { // sum - j
                dp[j] += dp[j - nums[i]];
            }
        }
        return dp[sum];
    }
};
```

一和零

```
class Solution {
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
        int dp[m + 1][n + 1];
        memset(dp, 0, sizeof(dp));
        for(int i = 0; i < strs.size(); i++) { // 外层for循环遍历物品
            int numof0 = 0, numof1 = 0;
            for(int j = 0; j < strs[i].size(); j++) {
                if(strs[i][j] == '0') numof0++;
                else numof1++;
            }
            for(int u = m; u >= numof0; u--) { // 倒序遍历
                for(int v = n; v >= numof1; v--) {
                    dp[u][v] = max(dp[u][v], dp[u - numof0][v - numof1] + 1);
                }
            }
        }
        return dp[m][n];
    }
};
```

零钱兑换II

完全背包 -> 允许元素重复 -> 从前向后遍历

```
class Solution { // 完全背包
public:
    int change(int amount, vector<int>& coins) {
        int dp[amount + 1];
        memset(dp, 0, sizeof(dp));
        dp[0] = 1;
        for(int i = 0; i < coins.size(); i++) {
            for (int j = coins[i]; j <= amount; j++) {
                dp[j] += dp[j - coins[i]];
            }
        }
        return dp[amount];
    }
```

```
};
```

组合总和 IV

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。

```
class Solution { // 排列数
public:
    int combinationSum4(vector<int>& nums, int target) {
        int dp[target + 1];
        memset(dp, 0 ,sizeof(dp));
        dp[0] = 1;
         for (int i = 0; i <= target; i++) { // 遍历背包
            for (int j = 0; j < nums.size(); j++) { // 遍历物品
                if (i - nums[j] >= 0 && dp[i] < INT_MAX - dp[i - nums[j]]) {
                    dp[i] += dp[i - nums[j]];
                }
            }
        }
        return dp[target];
    }
};
```

爬楼梯（进阶版）

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

int main() {
    int n, m; cin >> n >> m;
    int dp[n + 1];
    memset(dp, 0, sizeof(dp));
    dp[0] = 1;
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= m; j++) {
            if(i >= j) dp[i] += dp[i - j];
        }
    }
    cout << dp[n] << endl;
    return 0;
}
```

零钱兑换

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> dp(amount + 1, INT_MAX / 2);
        dp[0] = 0;
        for(int i = 0; i < coins.size(); i++) {
            for(int j = coins[i]; j <= amount; j++) {
```

```
                dp[j] = min(dp[j], dp[j - coins[i]] + 1); // 可能会导致溢出
            }
        }
        if(dp[amount] == INT_MAX / 2) return -1;
        return dp[amount];
    }
};
```

初始值为 `INT_MAX / 2`

完全平方数

```
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1, INT_MAX);
        dp[0] = 0;
        for(int i = 1; i <= n; i++) {
            for(int j = 1; j * j <= i; j++) {
                dp[i] = min(dp[i], dp[i - j * j] + 1);
            }
        }
        return dp[n];
    }
};
```

单词拆分

相当于求 排列数

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> word(wordDict.begin(), wordDict.end());
        vector<bool> dp(s.size() + 1, false);
        dp[0] = true;
        for(int i = 1; i <= s.size(); i++) {
            for(int j = 0; j < i; j++) {
                string str = s.substr(j, i - j);
                if(dp[j] && word.find(str) != word.end()) {
                    dp[i] = true;
                }
            }
        }
        return dp[s.size()];
    }
};
```

背包问题总结

回顾一下01背包的核心代码

```
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = bagWeight; j >= weight[i]; j--) { // 遍历背包容量
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
```

我们知道01背包内嵌的循环是从大到小遍历，为了保证每个物品仅被添加一次。

而完全背包的物品是可以添加多次的，所以要从小到大去遍历，即：

```
// 先遍历物品，再遍历背包
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
        dp[j] = max(dp[j], dp[j - weight[i]] + value[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。

背包递推公式：

问能否能装满背包（或者最多装多少）：dp[j] = max(dp[j], dp[j - nums[i]] + nums[i]);，对应题目如下：

- 动态规划：416.分割等和子集(opens new window)
- 动态规划：1049.最后一块石头的重量 II(opens new window)

问装满背包有几种方法：dp[j] += dp[j - nums[i]]，对应题目如下：

- 动态规划：494.目标和(opens new window)
- 动态规划：518. 零钱兑换 II(opens new window)
- 动态规划：377.组合总和IV(opens new window)
- 动态规划：70. 爬楼梯进阶版（完全背包）(opens new window)

问背包装满最大价值：dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);，对应题目如下：

- 动态规划：474.一和零(opens new window)

问装满背包所有物品的最小个数：dp[j] = min(dp[j - coins[i]] + 1, dp[j]);，对应题目如下：

- 动态规划：322.零钱兑换(opens new window)
- 动态规划：279.完全平方数

背包问题

01背包

理论基础
- 初始化背包容量为0的列及只取一号物品的行
- 对于容量小于物品容量的情况单独处理
- 背包容量大于本行物体容量再使用递归公式
- 遍历顺序需保证左上角，因此先行后列或者先列后行都可以

滚动一维数组解决01背包
- 初始化为0
- 递归公式有所变化
- 对背包容量需倒序遍历，防止一个物品被多次使用
- 遍历应先遍历物品内部遍历背包容量

分割等和子集
- 价值与重量都是一个值
- 背包容量应该是从0至数组和的一半
- 当有最大价值与最大背包容量恰好一致那么就说明可分解
- 可以在遍历过程中增加一个判断，当判断到此时的最大价值已经等于目标值直接返回，这样可以进一步节省时间

```
for (int i=0;i<nums.size();i++){
        for (int j=sum;j>=nums[i];j--){
            dp[j]=max(dp[j],dp[j-nums[i]]+nums[i]);
            if (dp[j]==sum) return true;
        }
    }
```

- 时间复杂度O(n^2) 空间复杂度O(n)

最后一块石头的重量
- 基本跟分割子集一样，尽可能分成两个和相近的子集，返回两个子集的差
- 时间复杂度O(m*n),m为石头总重量的一半，n为石头块数
- 空间复杂度O(m)

目标和
- 我觉得最开始减法和应该是x-sum,然后后边x+(x-sum)=S，不过最终都一样，x=(S+sum)/2
- 动态表里每个元素的值代表有多少种方法
- 动态方程为dp[j]+=dp[j-nums[i]]　在求装满背包的方法数时就用这个状态方程
- 其他与常规背包问题一致

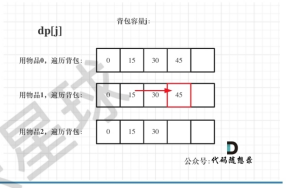
0和1
- 背包维度是二维的
- 需要先遍历物品，每个物品维护一个二维数组，二维数组不断更新，直到所有物品遍历完成
- 动态方程是靠当前物品与前一个状态的值加一得出的　dp[i][j]=max(dp[i][j],dp[i-zeroNum][j-oneNum]+1);

完全背包

理论基础
- 完全背包与01背包的区别只有物品是无数的，因此只需要更改对背包容量的遍历方式即可，也就是不使用倒序遍历，而是使用正序遍历

```
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
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);

    }
}
```



- 另外对完全背包问题物品与背包的遍历顺序可以随意调换，两个for的嵌套顺序那种都可以
- 可以自己写一下完全背包的二维数组形式

```
for(int i = 0; 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];
        else dp[i][j] = max(dp[i-1][j], dp[i ][j - weight[i]] + value[i]);
    }
}
```

零钱兑换
- 该问题要求组合数，也就是说不区分顺序，因此对嵌套有要求，只能先遍历物品，不能先遍历容量，先遍历容量的话，会导致一个组合被计算多次，如[1,3]和[3,1]
- 求组合数的动态方程为　dp[j]+=dp[j-coins[i]]

组合总和
- 该问题的解是有顺序的，所以需要更改遍历顺序，应该把背包容量放在嵌套的外边，物品种类放在嵌套的里边
- 另外存在过程的某两个数相加大于int的最大值，但是题干说了组合数在int范围内，因此对于那个大于范围的动态数组值选择不处理

爬楼梯
- 求上到n阶楼梯，每步可走1-m层，求有多少种方法
- 是一个完全背包问题，每个步长看作物品种类可以多次使用，目标值就是n,解需要考虑顺序，因此是外背包容量，内物品种类。
- 另外需注意嵌套循环是从1开始的，一位不存在0阶

零钱兑换2
- 各种硬币数量无限，求凑成总钱数的最小硬币数
- 动态数组代表总钱数j的最少硬币数
- 递推公式：dp[j] = min(dp[j - coins[i]] + 1, dp[j])
- 初始化dp[0]=0
- 遍历顺序无所谓

完全平方数
- 给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。
- 背包容量是n，物品种类是若干平方数，求组成和的个数最少
- 动态方程：dp[j] = min(dp[j - i * i] + 1, dp[j]);

单词拆分
- 有些没看懂动态公式！

多重背包
- 多个物品种类，每个物品种类都有不同数量个，因此可以把一个种类物品的数量拆开成每一个物品，这样就变成了01背包

	重量	价值	数量
物品0	1	15	1
物品1	3	20	2

	重量	价值	数量
物品0	1	15	1
物品0	1	15	1
物品1	3	20	1
物品1	3	20	1

物品0	1	15	2
物品1	3	20	3
物品2	4	30	2

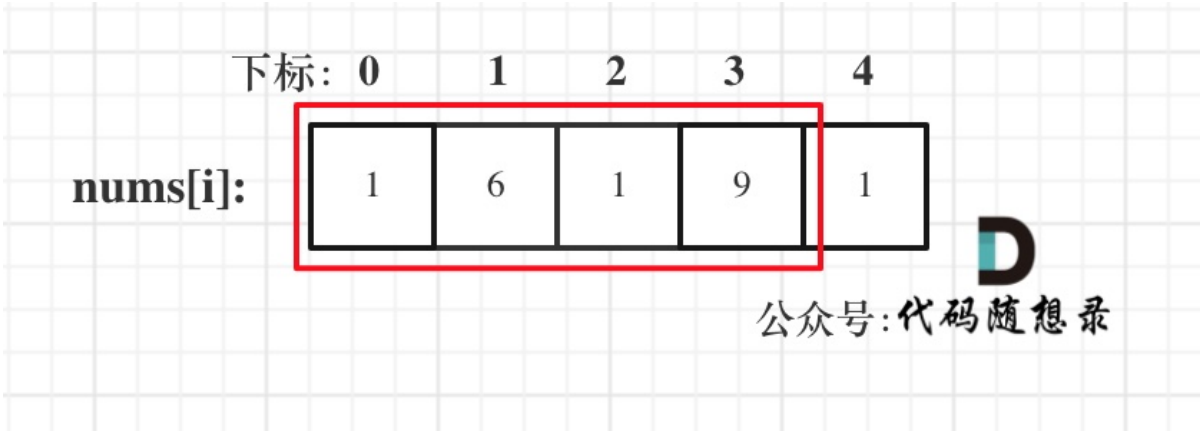
物品1	3	20	1
物品2	4	30	1
物品2	4	30	1

打家劫舍

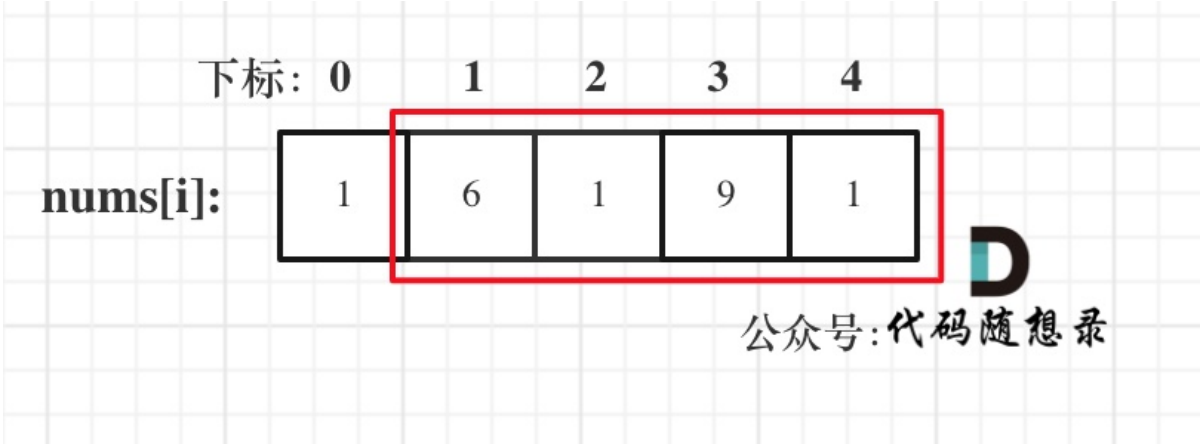
```
class Solution {
public:
    int rob(vector<int>& nums) {
        int dp[nums.size() + 1];
        memset(dp, 0, sizeof(dp));
        if(nums.size() == 1) return nums[0];
        else if(nums.size() == 2) return max(nums[0], nums[1]);
        dp[1] = nums[0];
        dp[2] = max(nums[0], nums[1]);
        for(int i = 2; i < nums.size(); i++) {
            dp[i + 1] = max(dp[i - 1] + nums[i], dp[i]);
        }
        return dp[nums.size()];
    }
};
```

打家劫舍II

- 情况一：考虑包含首元素，不包含尾元素



- 情况二：考虑包含尾元素，不包含首元素




```
class Solution {
public:
```

```
    int robRange(vector<int>& nums) {
        int dp[nums.size() + 1];
        memset(dp, 0, sizeof(dp));
        if(nums.size() == 1) return nums[0];
        else if(nums.size() == 2) return max(nums[0], nums[1]);
        dp[1] = nums[0];
        dp[2] = max(nums[0], nums[1]);
        for(int i = 2; i < nums.size(); i++) {
            dp[i + 1] = max(dp[i - 1] + nums[i], dp[i]);
        }
        return dp[nums.size()];
    }

    int rob(vector<int>& nums) {
        if(nums.size() == 1) return nums[0];
        else if(nums.size() == 2) return max(nums[0], nums[1]);
        vector<int> nums1 = nums;
        vector<int> nums2 = nums;
        nums1.pop_back();
        nums2.erase(nums2.begin());
        return max(robRange(nums1), robRange(nums2));
    }
};
```

删除元素 erase

买卖股票的最佳时机

直接贪心即可

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

买卖股票的最佳时机II

dp数组的含义：

- `dp[i][0]` 表示第i天持有股票所得现金。
- `dp[i][1]` 表示第i天不持有股票所得最多现金

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]`

再来看看如果第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]`

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int len = prices.size();
        int dp[len][2];
        memset(dp, 0, sizeof(dp));
        dp[0][0] -= prices[0]; // 第一天持有股票即花 prices[0] 购买股票
        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]);
            dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] + prices[i]);
        }
        return dp[len - 1][1];
    }
};
```

最长递增子序列

```
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); // 遍历所有比
nums[i]小的nums[j]
            }
            if (dp[i] > result) result = dp[i]; // 取长的子序列
        }
        return result;
    }
};
```

最长重复子数组

```
class Solution {
public:
    int findLength(vector<int>& nums1, vector<int>& nums2) {
        int dp[nums1.size() + 1][nums2.size() + 1];
        memset(dp, 0, sizeof(dp));
        int res = 0;
```

```
        for(int i = 1; i <= nums1.size(); i++) {
            for(int j = 1; j <= nums2.size(); j++) {
                if(nums1[i - 1] == nums2[j - 1]) { // 不相等肯定没相等大
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                }
                if(dp[i][j] > res) res = dp[i][j];
            }
        }
        return res;
    }
};
```

最长公共子序列

```
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;
                } else {
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[text1.size()][text2.size()];
    }
};
```

不同的子序列

`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]`

```
class Solution {
public:
    int numDistinct(string s, string t) {
        uint64_t dp[s.length() + 1][t.length() + 1];
        memset(dp, 0, sizeof(dp));
        for(int i = 0; i <= s.length(); i++) {
            dp[i][0] = 1;
        }
        for(int i = 1; i <= s.length(); i++) {
            for(int j = 1; j <= t.length(); 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.length()][t.length()];
    }
```

```
};
```