{% note danger no-icon flat %}
{% btn 'https://labuladong.online/',labuladong的算法笔记,fa fa-share ,larger outline%}
Start: 2025.03.01
End: 2025.03.31
Plan: AM 9:00 - 11:30
Check:

章节	(一)	(二)	(三)	(四)	(五)	(六)	(七)
基础	03.01	03.01	03.01	03.01	03.01	03.01	03.01
	03.02	03.02	03.02	\	\	\	\
第零章	03.04	03.05	03.06	03.07	03.08	03.09	03.09
	03.10	03.11	03.12	03.13	03.13	03.14	\
第一章	03.15	04.08	04.09				04.09
	04.10	\	\	\	\	\	\
第二章			\	\	\	\	\
第三章			04.11			04.11	\
第四章			\	\	\	\	\

前言：标准模板库 STL
- 数据结构
  1. vector
  2. stack
  3. queue
  4. deque
  5. unordered_set
  6. unordered_map
  7. priority_queue
  8. string
  9、list
- 算法与心得
  1. algorithm
  2. cmath
  3. 必备实现
  4. 心得操作与失误总结

基础：数据结构及排序
- (一) 数组（静态、动态）
- (二) 链表（单、双）
- (三) 变种：环形数组、跳表
- (四) 队列、栈、双端队列
- (五) 哈希表、哈希集合、加强哈希表
- (六) 二叉树
  1. 满二叉树
  2. 完全二叉树
  3. 平衡二叉树

4. 二叉搜索树
5. 递归遍历(DFS)
6. 层序遍历(BFS)
(七) 多叉树
1. 递归遍历（DFS）
2. 层序遍历（BFS）
(八) 二叉树变种
1. 二叉搜索树
2. 红黑树
3. Tire(字典树/前缀树)
4. 二叉堆
5. 线段树
(九) 图论
1. 图结构
2. 图的遍历
3. 并查集
(十) 十大排序
1. 选择排序
2. 冒泡排序（解决稳定）
3. 插入排序（逆向提高效率）
4. 希尔排序（突破n^2）
5. 快速排序（二叉树前序位置）
6. 归并排序（二叉树后序位置）
7. 堆排序（运用二叉堆）
8. 计数排序（新原理）
9. 桶排序（博采众长）
10. 基数排序(Radix Sort)
第零章、核心刷题框架汇总
(零) 万剑归宗
1. 数据结构的存储
2. 数据结构的操作
3. 算法的本质
4. 穷举的难点
(一) 双指针（链表）
1、合并两个有序链表
2、单链表的分解
3、合并 k 个有序链表
4、单链表的倒数第 k 个节点
5、单链表的中点
6、判断链表是否包含环
7、两个链表是否相交
(二) 双指针（数组）
1、原地修改
2、滑动窗口
3、二分查找
4、n 数之和
5、反转数组
6、回文串判断
(三) 滑动窗口
1、框架概述
2、最小覆盖子串
3、字符串排列

4、找所有字母异位词
5、最长无重复子串
(四) 二分搜索
1、代码框架
2、寻找一个数（基本）
3、寻找左边界
4、寻找右边界
(五) 递归（一个视角+两种思维）
1、从树的角度理解递归
2、编写递归的两种思维模式
(六) 动态规划
1、斐波那契数列
2、凑零钱
(七) 回溯(DFS)
1、全排列
(八) BFS
1、算法框架
2、滑动谜题
3、解开密码锁的最少次数
4、双向 BFS 优化
(九) 二叉树系列
1、二叉树的重要性、深入理解前中后序
2、两种解题思路
3、后序位置的特殊之处
4、以树的视角看动归/回溯/DFS算法的区别和联系
5、层序遍历
(十) 排列、组合、子集（回溯）
1、子集（元素无重不可复选）
2、组合（元素无重不可复选）
3、排列（元素无重不可复选）
4、子集/组合（元素有重不可复选）
5、排列（元素有重不可复选）
6、子集/组合（元素无重可复选）
7、排列（元素无重可复选）
8、最后总结
(十一) 贪心算法
1、贪心选择性质
2、例题实战
3、贪心算法的解题步骤
(十二) 分治算法
1、分治思想
2、分治算法
3、无效的分治
4、有效的分治
5、总结
(十三) 时空复杂度分析
1、复杂度反推解题
2、编码失误导致异常
3、Big O 表示法
4、算法分析
5、最后总结
第一章、经典数据结构算法

(一) 链表
1. 82. 删除排序链表中的重复元素 II
2. 264. 丑数 II
3. 有序矩阵中第 K 小的元素
4. 查找和最小的 K 对数字
5. 两数相加
6. 两数相加 II
7、方法：花式反转链表
8、方法：回文链表判断
(二) 数组
1、双指针的七道题
2、二维数组的花式遍历
3、团灭 nSum 问题
4、前缀和数组
5、差分数组
6、滑动窗口延伸：Rabin Karp 字符匹配算法
7、二分搜索思维
8、带权随机选择算法
9、田忌赛马决策算法
(三) 队列、栈
1、队列与栈的互相实现
2、单调栈
3、单调队列
(四) 二叉树
(五) 二叉树强化
(六) 二叉树拓展
(七) 经典数据结构设计
1、LRU 算法
2、LFU 算法
3、随机集合、黑名单随机数
4、TreeMap 与 TreeSet
5、基本线段树
6、动态线段树
7、懒更新线段树
8、Tire 树
9、朋友圈时间线算法
10、考场座位分配算法
11、计算器的设计
12、两个二叉堆实现中位数算法
13、数组去重问题
(八) 图
1、有向图：环检测与拓扑排序
2、名流问题
3、二分图判定
4、Union-Find 并查集
5、Kruskal MST最小生成树（并查集+排序贪心）
6、Prim MST最小生成树（BFS+切分贪心）
7、Dijkstra 单源最短路径
第二章、经典暴力搜索算法
(一) DFS、回溯
(二) BFS
第三章、经典动态规划算法

(一) 基本技巧
(二) 子序列类型
(三) 背包类型
1、0-1 背包
2、子集背包
3、完全背包
4、目标和
(四) 游戏类型
(五) 贪心类型
1、解题框架
2、老司机加油算法
3、区间调度问题
4、扫描线技巧-安排会议室
5、视频剪辑
第四章、其他常见算法技巧
(一) 数学运算技巧
(二) 经典面试题

{% endnote %}

前言：标准模板库 STL

数据结构

1. vector

```cpp
#include <vector>
vector<int> a;
vector<int> a(n);
vector<int> a(n, 0);
vector<vector<int> b;
vector<vector<int> b(n, vector<int>(5, 0));

a.resize(m);
a.resize(m, 0);
a.assign(n, 0);

a[0] = 5;
a.push_back(666);
a.insert(a.begin() + 2, e); // 插入为第三个元素
a.insert(a.end(), c.begin(), c.end());  // a+c
a.erase(a.begin() + 2);     // 删除第三个元素
a.clear();

a.size();
a.empty();

a.front();
a.back();
```

2. stack

```
#include <stack>
stack<int> stk;

stk.push(4);
stk.pop();

stk.top();

stk.empty();
stk.size();
```

3. queue

```
#include <queue>
queue<int> Q;

Q.push(4);
Q.pop();

Q.front();
Q.back();

Q.empty();
Q.size();
```

4. deque

```
#include <deque>
deque<int> DQ;

DQ.push_front(e);
DQ.push_back(e);
DQ.pop_front();
DQ.pop_back();

DQ.front();
DQ.back();

DQ.empty();
DQ.size();
DQ.clear();
```

5. unordered_set

```cpp
#include <unordered_set>
unorder_set<int> Table;

Table.insert(e);
Table.erase(e);
Table.emplace(e);   // 比 insert 更高效

Table.count(e);
Table.find(e);  // 不存在则 Table.end()


Table.empty();
Table.size();
Table.clear();

for(auto &x: Table)  cout << x << endl;
```

6. unordered_map

```cpp
#include <unordered_map>
unordered_map<string, int> Map;

Map["apple"] = 1;
Map.insert({"banana", 2});
Map.erase("apple");
Map.emplace("cherry", 3);

int e = Map["apple"];

Map.count("apple");
Map.find("apple");  // 不存在返回 Map.end()

Map.empty();
Map.size();
Map.clear();

// 遍历
for(auto &x: Map)  cout << x.first << x.second << endl;    // apple  1
```

7. priority_queue

```cpp
#include <priority_queue>
// 大根堆
priority_queue<int> maxHeap;
priority_queue<int, vector<int>, less<int>> maxHeap;
// 小根堆
priority_queue<int, vector<int>, greater<int>> minHeap;

maxHeap.push(5);
```

```
maxHeap.pop();

maxHeap.top();

maxHeap.empty();
maxHeap.size();

// 自定义类型
struct DT{
    int no;
    string name;
    DT(int n, string na): no(n), name(na) {}
};
struct Cmp{
    bool operator()(const DT &a, const DT &b) const{
        return a.no < b.no;     // no 越大，则 DT 越大
    }
};

priority_queue<DT, vector<DT>, Cmp> maxHeap;
maxHeap.push(DT(3, "Mary"));
maxHeap.push(DT(1, "Smith"));
maxHeap.push(DT(2, "John"));

DT std = maxHeap.top();  // 3-Mary
std.no;
std.name;

maxHeap.pop();


// 优先级队列，最小堆
auto cmp = [](ListNode* a, ListNode* b) { return a->val > b->val; };
priority_queue<ListNode*, vector<ListNode*>, decltype(cmp)> minHeap;
```

8. string

```
#include <string>

string s1;
s1 = "apple";
string s2("hello");
string s3(s2);      // "hello"
string s4(5, 'A');  // "AAAAA"

s = s1 + s2;
s += "app";

s1.find('p');       // 1
s1.find("le");      // 3
s1.find("banana");  // string::npos
```

```cpp

s.size();
s.length();     // 一样效果，两个返回值都是 size_t 类型，可以转 int

char ch = s[0];
char ch = s.at(3);
s[5] = 'k';

s2.substr(2, 3);    // "llo"

str.insert(5, "Insert");    // 在指定位置插入字符串
str.erase(5, 6);            // 从指定位置开始删除指定长度的字符

string s5 = "abcdefg";
s5.replace(2, 3, "AAAAA");   // abAAAAAfg

bool r1 = (s1 == s2);    // 比较 true false
int r2 = s1.compare(s2); // 比较 -1, 0, 1 字典序比较

for(int i=0; i<s.size(); i++)  cout << s[i];
for(char &c : s)  cout << c;

string numS = "1234";
int numI = numS.stoi(numS);
double numD = numS.stod(numS);
int ai = 34;
string aiS = to_string(ai);

reverse(s.begin(), s.end());    // 反转
sort(s.begin(), s.end());       // 字典序

s.empty();
s.clear();
```

9、list

```cpp
// 双向链表容器，它允许在常数时间内进行任意位置的插入和删除操作
#include <list>

list<int> myList;
list<int> l2(n);    // n 个元素，全 0
list<int> l3(n, 8); // n 个元素，全 8
list<int> l4 = {1, 2, 3};
list<int> l5(l4);

myList.push_back(10);    // 在末尾添加元素
myList.push_front(20);   // 在开头添加元素
myList.insert(++myList.begin(), 30); // 在第二个位置插入30

myList.pop_back();       // 删除末尾元素
myList.pop_front();      // 删除开头元素
```

```cpp
myList.erase(myList.begin()); // 删除第一个元素
myList.remove(10);        // 删除所有值为10的元素
myList.clear();           // 清空list

int first = myList.front(); // 第一个元素
int last = myList.back();   // 最后一个元素

int size = myList.size();
bool isEmpty = myList.empty();

for (auto it = myList.begin(); it != myList.end(); ++it) {
    cout << *it << " ";
}

for (int val : myList) {
    cout << val << " ";
}

// 正向迭代器
for (list<int>::iterator it = myList.begin(); it != myList.end(); ++it) {
    cout << *it << " ";
}
// 反向迭代器
for (list<int>::reverse_iterator rit = myList.rbegin(); rit != myList.rend(); ++rit)
{
    cout << *rit << " ";
}

// 合并两个已排序的list
list<int> list1 = {1, 3, 5};
list<int> list2 = {2, 4, 6};
list1.merge(list2); // list1变为1,2,3,4,5,6, list2为空

// 排序
myList.sort(); // 默认升序
myList.sort(greater<int>()); // 降序

// 去重（必须先排序）
myList.unique();

// 反转
myList.reverse();

// 拼接（转移元素）
list<int> otherList = {7, 8, 9};
myList.splice(myList.end(), otherList); // 将otherList的元素移到myList末尾
```

算法与心得

1. algorithm

```cpp
#include <algorithm>

// 交换
swap(vec[0], vec[1]);

// 排序
sort(vec.begin(), vec.end());

// 最值
min(a, b);
max(a, b);
min({a, b, c, d});
max({a, b, c, d, e});
min_element(vec.begin(), vec.end());
max_element(vec.begin(), vec.end());

// 累积
#include <numeric>
int sum = accumulate(vec.begin(), vec.end(), 0);

// 二分查找（有序）
bool r = binary_search(vec.begin(), vec.end(), 3);

// 查找
auto it = find(vec.begin(), vec.end(), 3);
if (it != vec.end()) cout << *it << (it-vec.begin())  << endl;

// 出现次数
int cnt = count(vec.begin(), vec.end(), 3);

// 修改
fill(vec.begin(), vec.end(), 0); // 全部填充为 0
replace(vec.begin(), vec.end(), 3, 99); // 将 3 替换为 99

// 拷贝
vector<int> vec2(vec.size());
copy(vec.begin(), vec.end(), vec2.begin());

// 反转
reverse(vec.begin(), vec.end());
```

2. cmath

```cpp
#include <cmath>

// 常用函数
sqrt(x);    // 平方根
cbrt(x);    // 立方根

// 绝对值（整数、浮点数、长整数）
abs(a);
```

```
fabs(f);
labs(l);

// 取整（向上取整、向下取整、四舍五入）
ceil(a);
floor(a);
round(a);

// 指数
exp(k);     // e^k
exp2(k);    // 2^k
pow(a, k);  // a^k

// 对数
log(x);     // 自然对数（e 底）
log10(x);   // 常用对数（10 底）
log2(x);    // 对数（2 底）

// 三角函数
sin(x);
cos(x);
tan(x);

// 反三角函数
asin(x);
acos(x);
atan(x);

// 双曲函数
sinh(x);
cosh(x);
tanh(x);
```

3. 必备实现

```
// 最大公约数
// gcd(a, b) = gcd(b, a%b) 直到 b=0时，答案为a
int gcd(int a, int b) {
    a = abs(a);
    b = abs(b);
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// 最小公倍数
// lcm(a, b) = a * b / gcd(a, b)
int lcm(int a, int b) {
```

```
    if (a == 0 || b == 0)
        return 0;
    a = abs(a);
    b = abs(b);
    return a * b / gcd(a, b);
}
```

4. 心得操作与失误总结

```
// 保证下一个检索元素不同
for(int i=0; i<n; i++){
    ....
    ....
    while(i+1<n && nums[i]==nums[i+1])  i++;
}


// 如若是需要 if 的逻辑，不要自作聪明写 =。
// 下面的两个方式结果在遍历后会不一样（会被覆盖）
if(color[i]==color[j])  flag = false;   // 方式 1
flag = color[i] != color[j];            // 方式 2（会在遍历时被覆盖）


// 使用栈和队列时，取值后面记得写 pop ！！！
// 如果发现超时，很可能是你压根没 pop 一直卡死
Q.front();  Q.pop();
S.top();  S.pop();


// 当你发现能执行，题目也没写错，那么问题可能是你把int 写成了 bool （因为这个是能操作的，不会报错）
// 主要是在图：邻接表用 int，邻接矩阵用 bool。或者都用 int 也行
vector<vector<int>> G1;     // 邻接表
vector<vector<bool>> G2;    // 邻接矩阵
vector<vector<int>> G3;     // 邻接矩阵+权重


// 对自己设计的数据结构排序
// 设计一个三元组的数组，按照第三个元素排序
vector<vector<int>> MD;
MD.push_back({1, 2, 88});
MD.push_back({2, 2, 66});
MD.push_back({5, 1, 99});
sort(MD.begin(), MD.end(), [](const vector<int>& A, const vector<int>& B){
    return A[2] < B[2];
});


// 注意 优先队列不是 front 而是 top（因为本质是二叉堆，是树结构）
Q.front();  // 队列
S.top();    // 栈
PQ.top();   // 优先队列
```

```cpp


// 对于使用 cmp
// 1. sort 里面希望从小到大排序则用：
auto cmp = [](const vector<int>& a, const vector<int>& b){
    return a[2] < b[2];
}
sort(vc.begin(), vc.end(), cmp);

// 2. 最小堆使用：
auto cmp = [](const vector<int>& a, const vector<int>& B){
    return A[2] > B[2];
}
priority_quque<vector<int>, vector<vector<int>>, decltype(cmp)> PQ;


// 对于动态规划初始化，看情况，别乱覆盖了
// 错误（区别在 dp[0][0]会被覆盖）
for(int i=0; i<=n; i++)  dp[i][0] = true;
for(int j=0; j<=m; j++)  dp[0][j] = false;
// 正确
for(int i=0; i<=n; i++)  dp[i][0] = true;
for(int j=1; j<=m; j++)  dp[0][j] = false;


// 多变量状态的哈希方法：字符串哈希表
unordered_map<string, int> memo;    // 备忘录
string key = to_string(i) + ',' + to_string(remain);
memo[key] = 88;


// 动态规划备忘录，你别光用不记录啊！
// 错误：
string cur = to_string(i) + ',' + to_string(remain);
if(memo.find(cur) != memo.end())
    return memo[cur];
return dp(nums, i+1, remain-nums[i]) + dp(nums, i+1, remain+nums[i]);
// 正确：
string cur = to_string(i) + ',' + to_string(remain);
if(memo.find(cur) != memo.end())
    return memo[cur];
int res = dp(nums, i+1, remain-nums[i]) + dp(nums, i+1, remain+nums[i]);
memo[cur] = res;
return res;


// 动态规划，多想一想子问题如何慢慢推出完整问题，一般是选前/后 i 个量、达到 j 能否完成/完成的方法数。
// 贪心问题，多画图看看：一般是某个 change 量的折线图，然后看最低点。
// 贪心思路的本质，如果找不到重复计算，那就通过问题中一些隐藏较深的规律，来减少冗余计算。
// 一般是刚好可以把第二层的循环去掉，也就是说：
// 仔细思考，会发现第二层的循环失败的位置j，会刚好成为第一层循环的i的下一个位置。
```

```
// 这样子就会使得两层循环实际上是一层循环！！！

```

基础：数据结构及排序

(一) 数组（静态、动态）

动态数组底层还是静态数组，只是自动帮我们进行**数组空间的扩缩容**，并把增删查改操作进行了封装，让我们使用起来更方便而已。

(二) 链表（单、双）

略

(三) 变种：环形数组、跳表

1. 环形数组：
    - 环形数组技巧利用求模（余数）运算，将普通数组变成逻辑上的环形数组，可以让我们用 $O(1)$ 的时间在数组头部增删元素。
    - 核心原理：环形数组的关键在于，它维护了两个指针 `start` 和 `end`，`start` 指向第一个有效元素的索引，`end` 指向最后一个有效元素的下一个位置索引。
    - 理论上，你可以随意设计区间的开闭，但一般设计为**左闭右开**区间是最方便处理的。因为这样初始化 `start = end = 0` 时，区间 `[0, 0)` 中没有元素，但只要让 `end` 向右移动（扩大）一位，区间 `[0, 1)` 就包含一个元素 `0` 了。
2. 跳表
   一条普通的单链表长这样：

```
index  0  1  2  3  4  5  6  7  8  9
node   a->b->c->d->e->f->g->h->i->j
```

如果我们想查询索引为 `7` 的元素是什么，只能从索引 `0` 头结点开始往后遍历，直到遍历到索引 `7`，找到目标节点 `h`。

而跳表则是这样的：

```
indexLevel   0-----------------------8-----10
indexLevel   0-----------4-----------8-----10
indexLevel   0-----2-----4-----6-----8-----10
indexLevel   0--1--2--3--4--5--6--7--8--9--10
nodeLevel    a->b->c->d->e->f->g->h->i->j->k
```

此时，如果我们想查询索引为 `7` 的元素，可以从最高层索引开始一层一层地往下找，这个搜索过程中，会经过 $\log N$ 层索引，在每层索引中移动的次数不会超过 2 次（因为上层索引区间在下一层被分为两半），所以跳表的查询时间复杂度是 $O(\log N)$。

(四) 队列、栈、双端队列

1. 其实队列和栈都是 `操作受限` 的数据结构。
2. 队列只能在一端插入元素，另一端删除元素；
   栈只能在某一端插入和删除元素。
   而双端队列的队头和队尾都可以插入或删除元素。

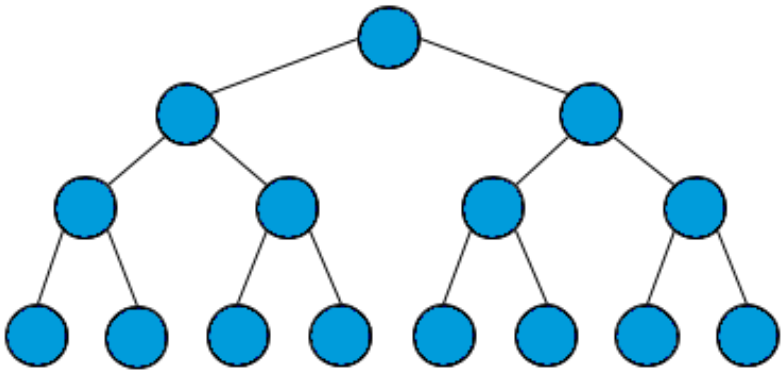
(五) 哈希表、哈希集合、加强哈希表

略

(六) 二叉树

1. 满二叉树

直接看图比较直观，满二叉树就是每一层节点都是满的，整棵树像一个正三角形：



满二叉树有个优势，就是它的节点个数很好算。假设深度为 h，那么总节点数就是 $2^h - 1$。

2. 完全二叉树

完全二叉树是指，二叉树的每一层的节点都紧凑靠左排列，且除了最后一层，其他每层都必须是满的：

不难发现，满二叉树其实是一种特殊的完全二叉树。

完全二叉树的特点：由于它的节点紧凑排列，如果从左到右从上到下对它的每个节点编号，那么父子节点的索引存在明显的规律。

这个特点在讲到 二叉堆核心原理 和 线段树核心原理 时会用到：**完全二叉树可以用数组来存储，不需要真的构建链式节点**。

完全二叉树还有个比较难发觉的性质：**完全二叉树的左右子树也是完全二叉树**。

或者更准确地说应该是：**完全二叉树的左右子树中，至少有一棵是满二叉树**。

3. 平衡二叉树

平衡二叉树（Balanced Binary Tree）是一种特殊的二叉树，**它的每个节点的左右子树的高度差不超过 1**。

要注意是每个节点，而不仅仅是根节点。

假设平衡二叉树中共有 N 个节点，那么平衡二叉树的高度是 $O(logN)$。

这是非常重要的性质，后面讲到的 `红黑树` 和 `线段树` 会利用平衡二叉树的这个性质，保证算法的高效性。

4. 二叉搜索树

二叉搜索树（Binary Search Tree，简称 BST）是一种很常见的二叉树，它的定义是：对于树中的每个节点，其**左子树的每个节点**的值都要小于这个节点的值，**右子树的每个节点**的值都要大于这个节点的值。你可以简单记为 **左小右大**。

BST 是非常常用的数据结构。因为左小右大的特性，可以让我们在 BST 中快速找到某个节点，或者找到某个范围内的所有节点，这是 BST 的优势所在。

5. 递归遍历(DFS)

```cpp
// 基本的二叉树节点
class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// 二叉树的遍历框架
void traverse(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    // 前序位置
    traverse(root->left);
    // 中序位置
    traverse(root->right);
    // 后序位置
}
```

> 1. 这个**递归遍历节点顺序是固定的**，务必记住这个顺序，否则你肯定玩不转二叉树结构。
> 2. 二叉树有前/中/后序三种遍历，会得到三种不同顺序的结果。为啥你这里说递归遍历节点的顺序是固定的呢?
>
> - 递归遍历的顺序，即 `traverse` 函数访问节点的顺序确实是固定的。正如上面那个可视化面板，root 指针在树上移动的顺序是固定的。
> - 前中后序遍历的结果不同，原因是因为你把操作效果代码写在了不同位置，所以产生了不同的效果。
>
> 3. BST 的中序遍历结构有序。

6. 层序遍历(BFS)

二叉树的层序遍历，顾名思义，就是一层一层地遍历二叉树。

这个遍历方式需要借助队列来实现，而且根据不同的需求，主要有三种不同的写法，下面一一列举。

（1）写法一

- 优点：简单
- 缺点：无法知道当前是第几层

```
void levelOrderTraverse(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    std::queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        TreeNode* cur = q.front();
        q.pop();
        // 访问 cur 节点
        std::cout << cur->val << std::endl;

        // 把 cur 的左右子节点加入队列
        if (cur->left != nullptr) {
            q.push(cur->left);
        }
        if (cur->right != nullptr) {
            q.push(cur->right);
        }
    }
}
```

> 知道节点的层数是个常见的需求，比方说让你收集每一层的节点，或者计算二叉树的最小深度等等。
>
> 所以这种写法虽然简单，但用的不多，下面介绍的写法会更常见一些。

（2）写法二

- 优点
- 缺点：

```
void levelOrderTraverse(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    queue<TreeNode*> q;
    q.push(root);
    // 记录当前遍历到的层数（根节点视为第 1 层）
    int depth = 1;

    while (!q.empty()) {
        int sz = q.size();
        for (int i = 0; i < sz; i++) {
            TreeNode* cur = q.front();
```

```
            q.pop();
            // 访问 cur 节点，同时知道它所在的层数
            cout << "depth = " << depth << ", val = " << cur->val << endl;

            // 把 cur 的左右子节点加入队列
            if (cur->left != nullptr) {
                q.push(cur->left);
            }
            if (cur->right != nullptr) {
                q.push(cur->right);
            }
        }
        depth++;
    }
}
```

这个变量 `i` 记录的是节点 `cur` 是当前层的第几个，大部分算法题中都不会用到这个变量。

但是注意队列的长度 `sz` 一定要**在循环开始前保存**下来，因为在循环过程中队列的长度是会变化的，不能直接用 `q.size()` 作为循环条件。

这种写法就可以记录下来每个节点所在的层数，可以解决诸如二叉树最小深度这样的问题，是我们最常用的层序遍历写法。

（3）写法三

既然写法二是最常见的，为啥还有个写法三呢？因为要给后面的进阶内容做铺垫。

回顾写法二，我们每向下遍历一层，就给 `depth` 加 `1`，可以理解为每条树枝的权重是 `1`，二叉树中每个节点的深度，其实就是从根节点到这个节点的路径权重和，且同一层的所有节点，路径权重和都是相同的。

那么假设，如果每条树枝的权重和可以是任意值，现在让你层序遍历整棵树，打印每个节点的路径权重和，你会怎么做？

写法三就是为了解决这个问题，在写法一的基础上添加一个 `State` 类，让每个节点自己负责维护自己的路径权重和，代码如下：

```
class State {
public:
    TreeNode* node;
    int depth;

    State(TreeNode* node, int depth) : node(node), depth(depth) {}
};

void levelOrderTraverse(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    queue<State> q;
    // 根节点的路径权重和是 1
    q.push(State(root, 1));

```

```
    while (!q.empty()) {
        State cur = q.front();
        q.pop();
        // 访问 cur 节点，同时知道它的路径权重和
        cout << "depth = " << cur.depth << ", val = " << cur.node->val << endl;

        // 把 cur 的左右子节点加入队列
        if (cur.node->left != nullptr) {
            q.push(State(cur.node->left, cur.depth + 1));
        }
        if (cur.node->right != nullptr) {
            q.push(State(cur.node->right, cur.depth + 1));
        }
    }
}
```

这样每个节点都有了自己的 `depth` 变量，是最灵活的，可以满足所有 BFS 算法的需求。

其实你很快就会学到，这种边带有权重的场景属于图结构算法，在之后的 BFS 算法习题集 和 dijkstra 算法 中，才会用到这种写法。

> 二叉树的遍历方式只有上面两种，也许有其他的写法，但都是表现形式上的差异，本质上不可能跳出上面两种遍历方式。
>
> **DFS 算法在寻找所有路径的问题中更常用。**
>
> **而 BFS 算法在寻找最短路径的问题中更常用。**
>
> 1. 为什么 BFS 常用来寻找最短路径?
>
> - 对于 DFS，你能不能在不遍历完整棵树的情况下，提前结束算法?
> 不可以，因为你必须确切的知道每条树枝的深度（根节点到叶子节点的距离），才能找到最小的那个。
> 即，**DFS 遍历当然也可以用来寻找最短路径，但必须遍历完所有节点才能得到最短路径。**
> - 对于 BFS，**由于 BFS 逐层遍历的逻辑，第一次遇到目标节点时，所经过的路径就是最短路径，算法可能并不需要遍历完所有节点就能提前结束。**
> - 从时间复杂度的角度来看，两种算法在最坏情况下都会遍历所有节点，时间复杂度都是 $O(N)$，但在一般情况下，显然 BFS 算法的实际效率会更高。所以在**寻找最短路径的问题中，BFS 算法是首选。**
>
> 2. 为什么 DFS 常用来寻找所有路径?
>
> - 你想啊，就以二叉树为例，如果要用 BFS 算法来寻找所有路径（根节点到每个叶子节点都是一条路径），队列里面就不能只放节点了，而需要使用第三种写法，新建一个 `State` 类，**把当前节点以及到达当前节点的路径都存进去，这样才能正确维护每个节点的路径**，最终计算出所有路径。
> - 而使用 DFS 算法就简单了，**它本就是一条树枝一条树枝从左往右遍历的，每条树枝就是一条路径，所以 DFS 算法天然适合寻找所有路径。**

(七) 多叉树

二叉树的节点长这样，每个节点有两个子节点。多叉树的节点长这样，每个节点有任意个子节点。就这点区别，其他没了。

```cpp
// 二叉树
class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int v) : val(v), left(nullptr), right(nullptr) {}
};

// 多叉树
class Node {
public:
    int val;
    vector<Node*> children;
};
```

> 森林：森林就是多个多叉树的集合。一棵多叉树其实也是一个特殊的森林。
>
> 在并查集算法中，我们会同时持有多棵多叉树的根节点，那么这些根节点的集合就是一个森林。

1. 递归遍历（DFS）

唯一的区别是，多叉树没有了中序位置，因为可能有多个节点嘛，所谓的中序位置也就没什么意义了。

```cpp
// 二叉树的遍历框架
void traverse(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    // 前序位置
    traverse(root->left);
    // 中序位置
    traverse(root->right);
    // 后序位置
}

// N 叉树的遍历框架
void traverse(Node* root) {
    if (root == nullptr) {
        return;
    }
    // 前序位置
    for (Node* child : root->children) {
        traverse(child);
    }
    // 后序位置
}
```

2. 层序遍历（BFS）

多叉树的层序遍历和 二叉树的层序遍历 一样，都是用队列来实现，无非就是把二叉树的左右子节点换成了多叉树的所有子节点。所以多叉树的层序遍历也有三种写法，下面一一列举。

```cpp
// 写法一：简单层序遍历
void levelOrderTraverse(Node* root) {
    if (root == nullptr) {
        return;
    }
    std::queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* cur = q.front();
        q.pop();
        // 访问 cur 节点
        std::cout << cur->val << std::endl;

        // 把 cur 的所有子节点加入队列
        for (Node* child : cur->children) {
            q.push(child);
        }
    }
}

// 写法二：记录节点深度
#include <iostream>
#include <queue>
#include <vector>

void levelOrderTraverse(Node* root) {
    if (root == nullptr) {
        return;
    }
    std::queue<Node*> q;
    q.push(root);
    // 记录当前遍历到的层数（根节点视为第 1 层）
    int depth = 1;

    while (!q.empty()) {
        int sz = q.size();
        for (int i = 0; i < sz; i++) {
            Node* cur = q.front();
            q.pop();
            // 访问 cur 节点，同时知道它所在的层数
            std::cout << "depth = " << depth << ", val = " << cur->val << std::endl;

            for (Node* child : cur->children) {
                q.push(child);
            }
        }
        depth++;
    }
}
```

```cpp

// 写法三：适配不同权重边
class State {
public:
    Node* node;
    int depth;

    State(Node* node, int depth) : node(node), depth(depth) {}
};

void levelOrderTraverse(Node* root) {
    if (root == nullptr) {
        return;
    }
    std::queue<State> q;
    // 记录当前遍历到的层数（根节点视为第 1 层）
    q.push(State(root, 1));

    while (!q.empty()) {
        State state = q.front();
        q.pop();
        Node* cur = state.node;
        int depth = state.depth;
        // 访问 cur 节点，同时知道它所在的层数
        std::cout << "depth = " << depth << ", val = " << cur->val << std::endl;

        for (Node* child : cur->children) {
            q.push(State(child, depth + 1));
        }
    }
}
```

。

(八) 二叉树变种

1. 二叉搜索树

- 二叉搜索树是特殊的 二叉树结构，其主要的实际应用是 `TreeMap` 和 `TreeSet`。
- 对于树中的每个节点，其**左子树的每个节点**的值都要小于这个节点的值，**右子树的每个节点**的值都要大于这个节点的值。
- 这个**左小右大**的特性，可以让我们在 BST 中快速找到某个节点，或者找到某个范围内的所有节点，这是 BST 的优势所在。
- 理想的时间复杂度是树的高度 $O(logN)$，而普通的二叉树遍历函数则需要 $O(N)$ 的时间遍历所有节点。
- 增删改的时间复杂度也是 $O(logN)$。

- 前文说复杂度是树的高度 $O(logN)$（N为节点总数），这是有前提的，即**二叉搜索树要是平衡的，也就是左右子树的高度差不能太大。**
  如果搜索树不平衡，比如这种极端情况，所有节点都只有右子树，没有左子树。这样二叉搜索树其实退化成了一条单链表，树的高度等于节点数 $O(N)$，这种情况下，即便这棵树符合 BST 的定义，但是性能就退化成了链表的性能，复杂度全部变成了 $O(N)$。
- **二叉搜索树的性能取决于树的高度，树的高度取决于树的平衡性。**
- 大家熟知的红黑树就是一类自平衡的二叉搜索树，它的平衡性能非常好，但是实现起来比较复杂，这就是完美所需付出的代价。

```
    7
   / \
  4   9
 / \   \
1   5   10
```

TreeMap/TreeSet 实现原理

它和前文介绍的 哈希表 `HashMap` 的结构是类似的，都是**存储键值对**的，`HashMap` 底层把键值对存储在一个 `table` 数组里面，而 `TreeMap` 底层把**键值对存储在一棵二叉搜索树的节点**里面。

至于 `TreeSet`，它和 `TreeMap` 的关系正如哈希表 `HashMap` 和哈希集合 `HashSet` 的关系一样，说白了就是 `TreeMap` 的简单封装，所以下面主要讲解 `TreeMap` 的实现原理。

```cpp
// 普通 TreeNode
class TreeNode {
public:
 int val;
 TreeNode* left;
 TreeNode* right;
 TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// 更改得到 TreeMap
// 大写 K 为键的类型，大写 V 为值的类型
template <typename K, typename V>
class TreeNode {
public:
 K key;
 V value;
 TreeNode<K, V>* left;
 TreeNode<K, V>* right;
 TreeNode(K key, V value) : key(key), value(value), left(nullptr),
right(nullptr) {}
};
```

- `get` 方法其实就类似上面可视化面板中查找目标节点的操作，根据目标 `key` 和当前节点的 `key` 比较，决定往左走还是往右走，可以一次性排除掉一半的节点，复杂度是 $O(\log N)$。
- 至于 `put`，`remove`，`containsKey` 方法，其实也是要先利用 `get` 方法找到目标键所在的节点，然后做一些指针操作，复杂度都是 $O(\log N)$。

> - `keys` 方法返回所有键，且结果有序。可以利用 BST 的中序遍历结果有序的特性。

2. 红黑树

- 红黑树是自平衡的二叉搜索树，它的树高在任何时候都能保持在 $O(\log N)$（完美平衡），这样就能保证增删查改的时间复杂度都是 $O(\log N)$。

3. Tire(字典树/前缀树)

- Trie 树本质上就是一棵从二叉树衍生出来的多叉树。
- Trie 树就是 **多叉树结构** 的延伸，是一种针对字符串进行特殊优化的数据结构。
- Trie 树在处理字符串相关操作时有诸多优势，比如节省公共字符串前缀的内存空间、方便处理前缀操作、支持通配符匹配等。
- 这里要特别注意，`TrieNode` 节点本身只存储 `val` 字段，并没有一个字段来存储字符，字符是通过子节点在父节点的 `children` 数组中的索引确定的。形象理解就是，Trie 树用**树枝**存储字符串（键），用**节点**存储字符串（键）对应的数据（值）。所以我在图中把字符标在树枝，键对应的值 `val` 标在节点上：

```
// Trie 树节点实现
template<typename V>
struct TrieNode {
    V val = NULL;
    TrieNode<V>* children[256] = { NULL };
};
```

这个 `val` 字段存储键对应的值，`children` 数组存储指向子节点的指针。但是和之前的普通多叉树节点不同，`TrieNode` 中 `children` 数组的索引是有意义的，代表键中的一个字符。

> 比如在实际做题时，题目说了只包含字符 `a-z`，那么你可以把大小改成 `26`；或者你不想用字符索引来映射，直接用哈希表 `HashMap<Character, TrieNode>` 也可以，都是一样的效果。

4. 二叉堆

- 二叉堆是一种能够动态排序的数据结构，是二叉树结构的延伸。
- 两个核心操作：下沉`sink`和上浮`swim`。
- 主要应用：优先队列`priority_queue`、堆排序`heap_sort`。

> 能够动态排序的数据结构，只有两个：1. 优先队列（底层采用二叉堆实现）2. 二叉搜索树
> 二叉搜索树的功能更广，但是优先队列的 API 代码更简单，一般采用优先队列。

- 你可以认为二叉堆是一种特殊的二叉树，这棵二叉树上的任意节点的值，都必须大于等于（或小于等于）其左右子树**所有**节点的值。如果是大于等于，我们称之为`大顶堆`，如果是小于等于，我们称之为`小顶堆`。
- 二叉堆可以辅助我们快速找到最大值或最小值。
- 二叉堆还有个性质：**一个二叉堆的左右子堆（子树）也是一个二叉堆。**
- 应用 1：优先队列
    - 自动排序是有代价的，注意优先级队列 API 的时间复杂度，**增删元素**的复杂度是 $O(\log N)$其中N是当前二叉堆中的元素个数。

- 标准队列是先进先出的顺序，而二叉堆可以理解为一种**会自动排序的队列**，所以叫做优先级队列感觉也挺贴切的。当然，你也一定要明白，虽然它的 API 像队列，但**它的底层原理和二叉树有关，和队列没啥关系**。

> - 以小顶堆为例，向小顶堆中**插入**新元素遵循两个步骤：
> 1、先把新元素追加到二叉树底层的最右侧，保持完全二叉树的结构。此时该元素的父节点可能比它大，不满足小顶堆的性质。
> 2、为了恢复小顶堆的性质，需要将这个新元素不断上浮（swim），直到它的父节点比它小为止，或者到达根节点。此时整个二叉树就满足小顶堆的性质了。
> - 以小顶堆为例，**删除**小顶堆的堆顶元素遵循两个步骤：
> 1、先把堆顶元素删除，把二叉树底层的最右侧元素摘除并移动到堆顶，保持完全二叉树的结构。此时堆顶元素可能比它的子节点大，不满足小顶堆的性质。
> 2、为了恢复小顶堆的性质，需要将这个新的堆顶元素不断下沉（sink），直到它比它的子节点小为止，或者到达叶子节点。此时整个二叉树就满足小顶堆的性质了。
> - **在数组上模拟二叉树：**
> 正常情况下你如何拿到二叉树的底层最右侧节点？你需要层序遍历或递归遍历二叉树，时间复杂度是 $O(N)$，进而导致 `push` 和 `pop` 方法的时间复杂度退化到 $O(N)$，这显然是不可接受的。如果用数组来模拟二叉树，就可以完美解决这个问题，在 $O(1)$ 时间内找到二叉树的**底层最右侧节点**。
> - 完全二叉树是关键：
> 想要用数组模拟二叉树，前提是这个二叉树**必须是完全二叉树**。
> 直接在数组的末尾追加元素，就相当于在完全二叉树的最后一层从左到右依次填充元素；
> 数组中最后一个元素，就是完全二叉树的底层最右侧的元素，完美契合我们实现二叉堆的场景。
>
> 在这个数组中，索引 0 空着不用，就可以根据任意节点的索引计算出父节点或左右子节点的索引：
>
> ```
> // 父节点的索引
> int parent(int node) {
>   return node / 2;
> }
> // 左子节点的索引
> int left(int node) {
>     return node * 2;
> }
> // 右子节点的索引
> int right(int node) {
>     return node * 2 + 1;
> }
> ```
>
> 其实从 0 开始也是可以的，稍微改一改计算公式就行了。

完整实现：

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

```

```cpp
class SimpleMinPQ {
    // 底层使用数组实现二叉堆
    vector<int> heap;

    // 堆中元素的数量
    int size;

    // 父节点的索引
    static int parent(int node) {
        return (node - 1) / 2;
    }

    // 左子节点的索引
    static int left(int node) {
        return node * 2 + 1;
    }

    // 右子节点的索引
    static int right(int node) {
        return node * 2 + 2;
    }

    // 上浮操作，时间复杂度是树高 O(logN)
    void swim(int node) {
        while (node > 0 && heap[parent(node)] > heap[node]) {
            swap(heap[parent(node)], heap[node]);
            node = parent(node);
        }
    }

    // 下沉操作，时间复杂度是树高 O(logN)
    void sink(int node) {
        while (left(node) < size || right(node) < size) {
            // 比较自己和左右子节点，看看谁最小
            int min = node;
            if (left(node) < size && heap[left(node)] < heap[min]) {
                min = left(node);
            }
            if (right(node) < size && heap[right(node)] < heap[min]) {
                min = right(node);
            }
            if (min == node) {
                break;
            }
            // 如果左右子节点中有比自己小的，就交换
            swap(heap[node], heap[min]);
            node = min;
        }
    }

public:
    // 构造函数，初始化容量
```

```cpp
    SimpleMinPQ(int capacity) {
        heap.resize(capacity);
        size = 0;
    }

    // 返回堆的元素数量
    int getSize() const {
        return size;
    }

    // 查，返回堆顶元素，时间复杂度 O(1)
    int peek() {
        return heap[0];
    }

    // 增，向堆中插入一个元素，时间复杂度 O(logN)
    void push(int x) {
        // 把新元素追加到最后
        heap[size] = x;
        // 然后上浮到正确位置
        swim(size);
        size++;
    }

    // 删，删除堆顶元素，时间复杂度 O(logN)
    int pop() {
        int res = heap[0];
        // 把堆底元素放到堆顶
        heap[0] = heap[size - 1];
        size--;
        // 然后下沉到正确位置
        sink(0);
        return res;
    }
};

int main() {
    SimpleMinPQ pq(5);
    pq.push(3);
    pq.push(2);
    pq.push(1);
    pq.push(5);
    pq.push(4);

    cout << pq.pop() << endl; // 1
    cout << pq.pop() << endl; // 2
    cout << pq.pop() << endl; // 3
    cout << pq.pop() << endl; // 4
    cout << pq.pop() << endl; // 5

    return 0;
}
```

- 应用 2：堆排序
  - 它的原理特别简单，就相当于把一个乱序的数组都 `push` 到一个二叉堆（优先级队列）里面，然后再一个个 `pop` 出来，就得到了一个有序的数组。

```cpp
// 堆排序伪码，对 arr 原地排序
// 时间复杂度 O(NlogN)，空间复杂度 O(N)
vector<int> heapSort(vector<int>& arr) {
    vector<int> res(arr.size());
    MyPriorityQueue pq;
    for (int x : arr)
        pq.push(x);
    // 元素出堆的顺序是有序的
    for (int i = 0; i < arr.size(); i++)
        res[i] = pq.pop();
    return res;
}
```

> 当然，正常的堆排序算法的代码并不依赖优先级队列，且空间复杂度是 $O(1)$。那是因为它把 `push` 和 `pop` 的代码逻辑展开了，再加上直接在数组上原地建堆，这样就不需要额外的空间了。
> 刚才不是说二叉堆是一种特殊的二叉树吗？怎么可能不使用额外的空间复杂度，直接在数组上原地创建二叉树呢？
> **二叉树**是一种**逻辑概念**，并不是说只有 `TreeNode` 类构造出来的那个结构才是二叉树，其实数组也可以抽象成一棵树，一切分治穷举的思想都可以抽象成一棵树，递归函数的那个递归栈也可以理解成一棵树。

5. 线段树

- 线段树是二叉树结构的衍生，用于高效解决区间查询和动态修改的问题。
  - 区间查询：$O(\log n)$
  - 动态修改单个元素：$O(\log n)$
- 这棵二叉树的叶子节点是数组中的元素，非叶子节点就是索引区间（线段）的汇总信息。
- 希望对整个区间进行查询，同时支持动态修改元素的场景，是线段树结构的应用场景。

(九) 图论

1. 图结构

- 图结构就是 **多叉树结构** 的延伸。图结构逻辑上由若干节点（`Vertex`）和边（`Edge`）构成，我们一般用邻接表、邻接矩阵等方式来存储图。
- 在树结构中，只允许父节点指向子节点，不存在子节点指向父节点的情况，子节点之间也不会互相链接；而图中没有那么多限制，节点之间可以相互指向，形成复杂的网络结构。
- 图的逻辑结构:一幅图是由节点 (Vertex) 和边 (Edge) 构成的，逻辑结构如下：

```cpp
// 图节点的逻辑结构
class Vertex {
public:
    int id;
    std::vector<Vertex*> neighbors;
};
// 邻接表
// graph[x] 存储 x 的所有邻居节点
vector<vector<int>> graph;
// 邻接矩阵
// matrix[x][y] 记录 x 是否有一条指向 y 的边
vector<vector<bool>> matrix;

// 对比
// 基本的 N 叉树节点
class TreeNode {
public:
    int val;
    std::vector<TreeNode*> children;
};
```

- **节点类型不是 int 怎么办?**
  上述讲解中，我默认图节点是一个从 0 开始的整数，所以才能存储到邻接表和邻接矩阵中，通过索引访问。
  但实际问题中，图节点可能是其他类型，比如字符串、自定义类等，那应该怎么存储呢？很简单，你再额外使用一个**哈希表**，把实际节点和整数 id 映射起来，然后就可以用邻接表和邻接矩阵存储整数 id 了。
  后面的讲解及习题中，我都会默认图节点是整数 id。
- 对于一幅有 V 个节点，E 条边的图，邻接表的空间复杂度是 $O(V+E)$（稀疏图），而邻接矩阵的空间复杂度是 $O(V^2)$（稠密图）。
  在后面的图算法和习题中，大多都是稀疏图，所以你会看到**邻接表**的使用更多一些。
  邻接矩阵的最大优势在于，矩阵是一个强有力的数学工具，图的一些隐晦性质可以借助精妙的矩阵运算展现出来。不过本文不准备引入数学内容，所以有兴趣的读者可以自行搜索学习。这也是为什么**一定要把图节点类型转换成整数 id 的原因**，不然的话你怎么用矩阵运算呢?

完整实现（邻接表）：

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

// 加权有向图的通用实现（邻接表）
class WeightedDigraph {
public:
    // 存储相邻节点及边的权重
    struct Edge {
```

```
        int to;
        int weight;

        Edge(int to, int weight) {
            this->to = to;
            this->weight = weight;
        }
    };

private:
    // 邻接表，graph[v] 存储节点 v 的所有邻居节点及对应权重
    vector<vector<Edge>> graph;

public:
    WeightedDigraph(int n) {
        // 我们这里简单起见，建图时要传入节点总数，这其实可以优化
        // 比如把 graph 设置为 Map<Integer, List<Edge>>，就可以动态添加新节点了
        graph = vector<vector<Edge>>(n);
    }

    // 增，添加一条带权重的有向边，复杂度 O(1)
    void addEdge(int from, int to, int weight) {
        graph[from].emplace_back(to, weight);
    }

    // 删，删除一条有向边，复杂度 O(V)
    void removeEdge(int from, int to) {
        for (auto it = graph[from].begin(); it != graph[from].end(); ++it) {
            if (it->to == to) {
                graph[from].erase(it);
                break;
            }
        }
    }

    // 查，判断两个节点是否相邻，复杂度 O(V)
    bool hasEdge(int from, int to) {
        for (const auto& e : graph[from]) {
            if (e.to == to) {
                return true;
            }
        }
        return false;
    }

    // 查，返回一条边的权重，复杂度 O(V)
    int weight(int from, int to) {
        for (const auto& e : graph[from]) {
            if (e.to == to) {
                return e.weight;
            }
        }
```

```
        throw invalid_argument("No such edge");
    }

    // 查，返回某个节点的所有邻居节点，复杂度 O(1)
    const vector<Edge>& neighbors(int v) {
        return graph[v];
    }
};

int main() {
    WeightedDigraph graph(3);
    graph.addEdge(0, 1, 1);
    graph.addEdge(1, 2, 2);
    graph.addEdge(2, 0, 3);
    graph.addEdge(2, 1, 4);

    cout << boolalpha << graph.hasEdge(0, 1) << endl; // true
    cout << boolalpha << graph.hasEdge(1, 0) << endl; // false

    for (const auto& edge : graph.neighbors(2)) {
        cout << "2 -> " << edge.to << ", wight: " << edge.weight << endl;
    }
    // 2 -> 0, wight: 3
    // 2 -> 1, wight: 4

    graph.removeEdge(0, 1);
    cout << boolalpha << graph.hasEdge(0, 1) << endl; // false

    return 0;
}
```

完整实现（邻接矩阵）：

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

// 加权有向图的通用实现（邻接矩阵）
class WeightedDigraph {
public:
    // 存储相邻节点及边的权重
    struct Edge {
        int to;
        int weight;

        Edge(int to, int weight) : to(to), weight(weight) {}
    };

    WeightedDigraph(int n) {
        matrix = std::vector<std::vector<int>>(n, std::vector<int>(n, 0));
    }

```

```
    // 增，添加一条带权重的有向边，复杂度 O(1)
    void addEdge(int from, int to, int weight) {
        matrix[from][to] = weight;
    }

    // 删，删除一条有向边，复杂度 O(1)
    void removeEdge(int from, int to) {
        matrix[from][to] = 0;
    }

    // 查，判断两个节点是否相邻，复杂度 O(1)
    bool hasEdge(int from, int to) {
        return matrix[from][to] != 0;
    }

    // 查，返回一条边的权重，复杂度 O(1)
    int weight(int from, int to) {
        return matrix[from][to];
    }

    // 查，返回某个节点的所有邻居节点，复杂度 O(V)
    std::vector<Edge> neighbors(int v) {
        std::vector<Edge> res;
        for (int i = 0; i < matrix[v].size(); i++) {
            if (matrix[v][i] > 0) {
                res.push_back(Edge(i, matrix[v][i]));
            }
        }
        return res;
    }

private:
    // 邻接矩阵，matrix[from][to] 存储从节点 from 到节点 to 的边的权重
    // 0 表示没有连接
    std::vector<std::vector<int>> matrix;
};

int main() {
    WeightedDigraph graph(3);
    graph.addEdge(0, 1, 1);
    graph.addEdge(1, 2, 2);
    graph.addEdge(2, 0, 3);
    graph.addEdge(2, 1, 4);

    std::cout << std::boolalpha;
    std::cout << graph.hasEdge(0, 1) << std::endl; // true
    std::cout << graph.hasEdge(1, 0) << std::endl; // false

    for (const auto& edge : graph.neighbors(2)) {
        std::cout << "2 -> " << edge.to << ", weight: " << edge.weight << std::endl;
    }
    // 2 -> 0, weight: 3
```

```
    // 2 -> 1, weight: 4

    graph.removeEdge(0, 1);
    std::cout << graph.hasEdge(0, 1) << std::endl; // false

    return 0;
}
```

2. 图的遍历

- 图的遍历就是 多叉树遍历 的延伸，主要遍历方式还是**深度优先搜索（DFS）**和**广度优先搜索（BFS）**。
- 唯一的区别是，树结构中不存在环，而**图结构中可能存在环**，所以我们需要**标记遍历过的节点**，避免遍历函数在环中死循环。
- 遍历图的「节点」和「路径」略有不同，遍历「**节点**」时，需要 visited 数组在**前序位置**标记节点；遍历图的所有「**路径**」时，需要 onPath 数组在**前序位置**标记节点，在**后序位置**撤销标记。

1. **深度优先搜索（DFS）**
   （1）遍历所有节点（visited 数组）

```
// 多叉树节点
class Node {
public:
    int val;
    std::vector<Node*> children;
};

// 多叉树的遍历框架
void traverse(Node* root) {
    // base case
    if (root == nullptr) {
        return;
    }
    // 前序位置
    std::cout << "visit " << root->val << std::endl;
    for (auto child : root->children) {
        traverse(child);
    }
    // 后序位置
}

// 图节点
class Vertex {
public:
    int id;
    std::vector<Vertex*> neighbors;
};

// 图的遍历框架
// 需要一个 visited 数组记录被遍历过的节点
// 避免走回头路陷入死循环
```

```cpp
void traverse(Vertex* s, std::vector<bool>& visited) {
    // base case
    if (s == nullptr) {
        return;
    }
    if (visited[s->id]) {
        // 防止死循环
        return;
    }
    // 前序位置
    visited[s->id] = true;
    std::cout << "visit " << s->id << std::endl;
    for (auto neighbor : s->neighbors) {
        traverse(neighbor);
    }
    // 后序位置
}
```

(2)遍历所有路径（onPath 数组）

前文遍历节点的代码中，visited 数组的职责是保证每个节点只会被访问一次。而对于图结构来说，要想遍历所有路径，可能会多次访问同一个节点，这是关键的区别。

```cpp
// 下面的算法代码可以遍历图的所有路径，寻找从 src 到 dest 的所有路径

// onPath 和 path 记录当前递归路径上的节点
vector<bool> onPath(graph.size());
list<int> path;

void traverse(Graph& graph, int src, int dest) {
    // base case
    if (src < 0 || src >= graph.size()) {
        return;
    }
    if (onPath[src]) {
        // 防止死循环（成环）
        return;
    }
    // 前序位置
    onPath[src] = true;
    path.push_back(src);
    if (src == dest) {
        cout << "find path: ";
        for (int node : path) {
            cout << node << " ";
        }
        cout << endl;
    }
    for (const Edge& e : graph.neighbors(src)) {
        traverse(graph, e.to, dest);
    }
    // 后序位置
```

```
    path.pop_back();
    onPath[src] = false;
}
```

(3)同时使用 visited 和 onPath 数组
遍历所有路径的算法复杂度较高，大部分情况下我们可能并不需要穷举完所有路径，而是仅需要找到某一条符合条件的路径。这种场景下，我们可能会借助 visited 数组进行**剪枝**，提前排除一些不符合条件的路径，从而降低复杂度。

(4)完全不用 visited 和 onPath 数组
visited 和 onPath 主要的作用就是处理成环的情况，避免死循环。那如果**题目告诉你输入的图结构不包含环**，那么你就不需要考虑成环的情况了。

2. **广度优先搜索（BFS）**

- 图结构的广度优先搜索其实就是 多叉树的**层序遍历**，无非就是加了一个 `visited` 数组来**避免重复遍历节点**。
- 理论上 BFS 遍历也需要区分遍历所有「节点」和遍历所有「路径」，但是实际上 BFS 算法一般只用来寻找那条**最短路径**，不会用来求所有路径。

(1)写法一：不记录遍历步数

```
// 多叉树的层序遍历
void levelOrderTraverse(Node* root) {
    if (root == nullptr) {
        return;
    }

    std::queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* cur = q.front();
        q.pop();
        // 访问 cur 节点
        std::cout << cur->val << std::endl;

        // 把 cur 的所有子节点加入队列
        for (Node* child : cur->children) {
            q.push(child);
        }
    }
}


// 图结构的 BFS 遍历，从节点 s 开始进行 BFS
void bfs(const Graph& graph, int s) {
    std::vector<bool> visited(graph.size(), false);
    std::queue<int> q;
    q.push(s);
    visited[s] = true;
```

```
30
31      while (!q.empty()) {
32          int cur = q.front();
33          q.pop();
34          std::cout << "visit " << cur << std::endl;
35          for (const Edge& e : graph.neighbors(cur)) {
36              if (!visited[e.to]) {
37                  q.push(e.to);
38                  visited[e.to] = true;
39              }
40          }
41      }
42  }
```

(2)写法二：记录遍历步数

```
1   // 多叉树的层序遍历
2   void levelOrderTraverse(Node* root) {
3       if (root == nullptr) {
4           return;
5       }
6       queue<Node*> q;
7       q.push(root);
8       // 记录当前遍历到的层数（根节点视为第 1 层）
9       int depth = 1;
10
11      while (!q.empty()) {
12          int sz = q.size();
13          for (int i = 0; i < sz; i++) {
14              Node* cur = q.front();
15              q.pop();
16              // 访问 cur 节点，同时知道它所在的层数
17              cout << "depth = " << depth << ", val = " << cur->val << endl;
18
19              for (Node* child : cur->children) {
20                  q.push(child);
21              }
22          }
23          depth++;
24      }
25  }
26
27
28  // 从 s 开始 BFS 遍历图的所有节点，且记录遍历的步数
29  void bfs(const Graph& graph, int s) {
30      vector<bool> visited(graph.size(), false);
31      queue<int> q;
32      q.push(s);
33      visited[s] = true;
34      // 记录从 s 开始走到当前节点的步数
35      int step = 0;
```

```
    while (!q.empty()) {
        int sz = q.size();
        for (int i = 0; i < sz; i++) {
            int cur = q.front();
            q.pop();
            cout << "visit " << cur << " at step " << step << endl;
            for (const Edge& e : graph.neighbors(cur)) {
                if (!visited[e.to]) {
                    q.push(e.to);
                    visited[e.to] = true;
                }
            }
        }
        step++;
    }
}
```

(3)写法三：适配不同权重边

```
// 多叉树的层序遍历
// 每个节点自行维护 State 类，记录深度等信息
class State {
public:
    Node* node;
    int depth;

    State(Node* node, int depth) : node(node), depth(depth) {}
};

void levelOrderTraverse(Node* root) {
    if (root == nullptr) {
        return;
    }
    queue<State> q;
    // 记录当前遍历到的层数（根节点视为第 1 层）
    q.push(State(root, 1));

    while (!q.empty()) {
        State state = q.front();
        q.pop();
        Node* cur = state.node;
        int depth = state.depth;
        // 访问 cur 节点，同时知道它所在的层数
        cout << "depth = " << depth << ", val = " << cur->val << endl;

        for (Node* child : cur->children) {
            q.push(State(child, depth + 1));
        }
    }
}

```

```cpp

// 图结构的 BFS 遍历，从节点 s 开始进行 BFS，且记录路径的权重和
// 每个节点自行维护 State 类，记录从 s 走来的权重和
class State {
public:
    // 当前节点 ID
    int node;
    // 从起点 s 到当前节点的权重和
    int weight;

    State(int node, int weight) : node(node), weight(weight) {}
};

void bfs(const Graph& graph, int s) {
    vector<bool> visited(graph.size(), false);
    queue<State> q;

    q.push(State(s, 0));
    visited[s] = true;

    while (!q.empty()) {
        State state = q.front();
        q.pop();
        int cur = state.node;
        int weight = state.weight;
        cout << "visit " << cur << " with path weight " << weight << endl;
        for (const Edge& e : graph.neighbors(cur)) {
            if (!visited[e.to]) {
                q.push(State(e.to, weight + e.weight));
                visited[e.to] = true;
            }
        }
    }
}
```

3. 并查集

- 并查集 Union Find 是一种二叉树结构的衍生，用于高效解决**无向图的连通性**问题。
- 查询两个节点是否连通：$O(1)$
- 合并两个连通分量：$O(1)$
- 查询连通分量的个数：$O(1)$

> 你单纯去查邻接表/邻接矩阵，只能判断两个节点是否**直接**相连，而无法处理这种**传递**的连通关系。

- 如果我们想办法把**同一个连通分量的节点**都放到**同一棵树**中，把**这棵树的根节点**作为这个连通分量的代表，那么我们就可以高效实现上面的操作了。
- 并查集底层其实是一片**森林（若干棵多叉树）**，每棵树代表一个**连通分量**：
    - `connected(p, q)`：只需要判断 p 和 q 所在的多叉树的**根节点**，若**相同**，则 p 和 q 在同一棵树中，即连通，否则不连通。

- `count()`：只需要统计一下总共有**多少棵树**，即可得到连通分量的数量。
- `union(p, q)`：只需要将 p 节点所在的这棵树的根节点，接入到 q 节点所在的这棵树的根节点下面，即可完成连接操作。注意这里并不是 p, q 两个节点的合并，而是**两棵树根节点的合并**。因为 p, q 一旦连通，那么他们所属的连通分量就合并成了同一个更大的连通分量。
  综上，并查集中**每个节点其实不在乎自己的子节点是谁，只在乎自己的根节点是谁**，所以一个并查集节点类似于下面这样：

> 所以**并查集算法最终的目标**，就是要**尽可能降低树的高度**，如果能保持树高为常数，那么上述方法的复杂度就都是 $O(1)$ 了。

（1）权重数组的优化效果
在仔细观察即可发现，使得树高线性增长的原因是，每次 `union` 操作都是将节点个数较多的树接到了节点个数较少的树下面，这就很容易让树高增加，很不明智。一种优化思路是引入一个权重数组，记录以每个节点的为根的树的节点个数，然后在 union 方法中，总是将节点个数较少的树接到节点个数较多的树下面，这样可以保证树尽可能平衡，树高也就不会线性增长。

（2）路径压缩的优化效果
路径压缩的效果，一旦触发，无论树枝的高度是多少，都会被直接压缩为 2，路径压缩的
均摊复杂度 是 $O(1)$，这样就可以保证 union, connected, find 方法的时间复杂度都是常数级别 $O(1)$。

(十) 十大排序

- 排序算法的关键指标
  1. 时空复杂度
  2. 排序稳定性
  3. 是否原地排序

1. 选择排序

- 选择排序是最简单朴素的排序算法，但是时间复杂度较高，且不是稳定排序。其他基础排序算法都是基于选择排序的优化。
- 每次都去遍历选择最小的元素。
- 分析：
  1. 选择排序算法是个不稳定排序算法，因为每次都要交换最小元素和当前元素的位置，这样可能会改变相同元素的相对位置。
  2. 选择排序的时间复杂度和初始数据的有序度完全没有关系，即便输入的是一个已经有序的数组，选择排序的时间复杂度依然是 $O(n^2)$。
  3. 选择排序的时间复杂度是 $O(n^2)$，具体的操作次数大概是 $n^2/2$ 次，常规的优化思路无法降低时间复杂度。

```
1  void sort(vector<int>& nums) {
2      int n = nums.size();
3      // sortedIndex 是一个分割线
4      // 索引 < sortedIndex 的元素都是已排序的
```

```
    // 索引 >= sortedIndex 的元素都是未排序的
    // 初始化为 0，表示整个数组都是未排序的
    int sortedIndex = 0;
    while (sortedIndex < n) {
        // 找到未排序部分 [sortedIndex, n) 中的最小值
        int minIndex = sortedIndex;
        for (int i = sortedIndex + 1; i < n; i++) {
            if (nums[i] < nums[minIndex]) {
                minIndex = i;
            }
        }
        // 交换最小值和 sortedIndex 处的元素
        int tmp = nums[sortedIndex];
        nums[sortedIndex] = nums[minIndex];
        nums[minIndex] = tmp;

        // sortedIndex 后移一位
        sortedIndex++;
    }
}
```

2. 冒泡排序（解决稳定）

- 冒泡算法是对 选择排序 的一种优化，通过交换 `nums[sortedIndex]` 右侧的逆序对完成排序，是一种**稳定**排序算法。
- 优化的方向就在这里，你不要图省事儿直接把本次查找的最小的直接一次交换到前面；而应该一步一步交换，慢慢到最前面。
- 这个算法的名字叫做冒泡排序，因为它的执行过程就像从数组尾部向头部冒出水泡，每次都会将最小值顶到正确的位置。
- 如果一次交换操作都没有进行，说明数组已经有序，可以提前终止算法。
- 唯一的遗憾是，时间复杂度依然是 $O(n^2)$，并没有降低。

```
// 对选择排序进行第二波优化，获得稳定性的同时避免额外的 for 循环
// 这个算法有另一个名字，叫做冒泡排序
void sort(int[] nums) {
    int n = nums.length;
    int sortedIndex = 0;
    while (sortedIndex < n) {
        // 寻找 nums[sortedIndex..] 中的最小值
        // 同时将这个最小值逐步移动到 nums[sortedIndex] 的位置
        for (int i = n - 1; i > sortedIndex; i--) {
            if (nums[i] < nums[i - 1]) {
                // swap(nums[i], nums[i - 1])
                int tmp = nums[i];
                nums[i] = nums[i - 1];
                nums[i - 1] = tmp;
            }
        }
        sortedIndex++;
```

```
    }
}
```

3. 插入排序（逆向提高效率）

- 插入排序是基于 选择排序 的一种优化，将 `nums[sortedIndex]` 插入到左侧的有序数组中。对于有序度较高的数组，插入排序的效率比较高。
- 选择排序和冒泡排序是在 `nums[sortedIndex..]` 中找到最小值，然后将其插入到 `nums[sortedIndex]` 的位置。
- 那么我们能不能反过来想，在 `nums[0..sortedIndex-1]` 这个部分有序的数组中，找到 `nums[sortedIndex]` 应该插入的位置插入。
- 这个算法的名字叫做插入排序，它的执行过程就像是打扑克牌时，将新抓到的牌插入到手中已经排好序的牌中。
- 插入排序的**空间复杂度**是 $O(1)$，是**原地**排序算法。**时间复杂度**是 $O(n^2)$，具体的操作次数和选择排序类似，是一个等差数列求和，大约是 $n^2/2$ 次。
- 插入排序是一种**稳定**排序，因为只有在 `nums[i] < nums[i - 1]` 的情况下才会交换元素，所以相同元素的相对位置不会发生改变。
- 初始有序度越高，效率越高。插入排序的综合性能应该要高于冒泡排序。

```
// 对选择排序进一步优化，向左侧有序数组中插入元素
// 这个算法有另一个名字，叫做插入排序
void sort(int[] nums) {
    int n = nums.length;
    // 维护 [0, sortedIndex) 是有序数组
    int sortedIndex = 0;
    while (sortedIndex < n) {
        // 将 nums[sortedIndex] 插入到有序数组 [0, sortedIndex) 中
        for (int i = sortedIndex; i > 0; i--) {
            if (nums[i] < nums[i - 1]) {
                // swap(nums[i], nums[i - 1])
                int tmp = nums[i];
                nums[i] = nums[i - 1];
                nums[i - 1] = tmp;
            } else {
                break;
            }
        }
        sortedIndex++;
    }
}
```

4. 希尔排序（突破n^2）

- 希尔排序是基于 插入排序 的简单改进，通过预处理增加数组的局部有序性，突破了插入排序的 $O(N^2)$ 时间复杂度。

- 希尔排序是**不稳定**排序。这个比较容易理解吧，当 `h` 大于 1 时进行的排序操作，就可能打乱相同元素的相对位置了。
- 空间复杂度是 $O(1)$，是**原地**排序算法。
- 希尔排序的时间复杂度是小于 $O(N^2)$ 的。

> h 有序数组：一个数组是 h 有序的，是指这个数组中任意间隔为 `h`（或者说间隔元素的个数为 `h-1`）的元素都是有序的。
> **当一个数组完成排序的时候，其实就是 1 有序数组。**

```
nums:
[1, 2, 4, 3, 5, 7, 8, 6, 10, 9, 12, 11]
 ^--------^--------^---------^
    ^--------^--------^---------^
       ^--------^--------^----------^

 1--------3--------8---------9
    2--------5--------6---------12
        4--------7--------10---------11
```

```java
// 希尔排序，对 h 有序数组进行插入排序
// 逐渐缩小 h，最后 h=1 时，完成整个数组的排序
void sort(int[] nums) {
    int n = nums.length;
    // 我们使用的生成函数是 2^(k-1)
    // 即 h = 1, 2, 4, 8, 16...
    int h = 1;
    while (h < n / 2) {
        h = 2 * h;
    }

    // 改动一，把插入排序的主要逻辑套在 h 的 while 循环中
    while (h >= 1) {
        // 改动二，sortedIndex 初始化为 h，而不是 1
        int sortedIndex = h;
        while (sortedIndex < n) {
            // 改动三，把比较和交换元素的步长设置为 h，而不是相邻元素
            for (int i = sortedIndex; i >= h; i -= h) {
                if (nums[i] < nums[i - h]) {
                    // swap(nums[i], nums[i - h])
                    int tmp = nums[i];
                    nums[i] = nums[i - h];
                    nums[i - h] = tmp;
                } else {
                    break;
                }
            }
            sortedIndex++;
        }

        // 按照递增函数的规则，缩小 h
```

```
        h /= 2;
    }
}
```

> 希尔排序的性能和**递增函数的选择**有很大关系，上面的代码中我们使用的递增函数是 $2^k - 1$，因为这是最简单的，但这并不最优的选择。比方说《算法 4》 中给的递增函数是 $(3^k - 1)/2$，即 `1,4,13,40,121,364...`
>
> ```
> // 把生成函数换成 (3^k - 1) / 2
> // 即 h = 1, 4, 13, 40, 121, 364...
> int h = 1;
> while (h < n / 3) {
>     h = 3 * h + 1;
> }
> ....
> // 按照递增函数的规则，缩小 h
> h /= 3;
> ....
> ```

5. 快速排序（二叉树前序位置）

- 快速排序的核心思路需要结合 **二叉树的前序遍历** 来理解：在二叉树遍历的前序位置将一个元素排好位置，然后递归地将剩下的元素排好位置。
- 快速排序的思路是：先把一个元素排好序，然后去把剩下的元素排好序。
    1. 任意选择一个元素作为切分元素 pivot（一般选择第一个元素）
    2. 对数组中的元素进行若干交换操作，将小于 pivot 的元素放到 pivot 的左边，大于 pivot 的元素放到 pivot 的右边（换句话说，**其实就是将 pivot 这一个元素排好序**）
    3. **递归的去把 pivot 左右两侧的其他元素排好序**。
- 本质是二叉树的前序遍历，在前序位置将 `nums[p]` 排好序，然后递归排序左右元素
- 其中 `partition` 函数的实现是快速排序的核心，即遍历 `nums[lo..hi]`，将切分点元素 `pivot` 放到正确的位置，并返回该位置的索引 `p`。
- 每层元素总和仍然是 $O(n)$，树高是 $O(\log n)$，所以总的时间复杂度是 $O(n \log n)$。
- 快速排序不需要额外的辅助空间，是原地排序算法。递归遍历二叉树时，递归函数的堆栈深度为树的高度，所以空间复杂度是 $O(\log n)$。
- 快速排序是**不稳定**排序算法，因为在 `partition` 函数中，不会考虑相同元素的相对位置。

```java
void sort(int nums[], int lo, int hi) {
    if (lo >= hi) {
        return;
    }
    // 对 nums[lo..hi] 进行切分，将 nums[p] 排好序
    // 使得 nums[lo..p-1] <= nums[p] < nums[p+1..hi]
    int p = partition(nums, lo, hi);
    // 去左右子数组进行切分
    sort(nums, lo, p - 1);
    sort(nums, p + 1, hi);
}
```

6. 归并排序（二叉树后序位置）

- 归并排序的核心思路需要结合 **二叉树的后序遍历** 来理解：先利用递归把左右两半子数组排好序，然后在二叉树的后序位置**合并**两个有序数组。
- 归并排序的稳定性取决于 `merge` 函数的实现，需要用到 **双指针技巧**，是**稳定**排序。
- 每层元素总和仍然是 $O(n)$，树高是 $O(\log n)$，所以总的时间复杂度是 $O(n \log n)$。
- **不是原地**排序。归并排序的 `merge` 函数需要一个额外的数组来辅助进行有序数组的合并操作，消耗 $O(n)$ 的空间。

```java
// 定义：排序 nums[lo..hi]
void sort(int[] nums, int lo, int hi) {
    if (lo == hi) {
        return;
    }
    int mid = (lo + hi) / 2;
    // 利用定义，排序 nums[lo..mid]
    sort(nums, lo, mid);
    // 利用定义，排序 nums[mid+1..hi]
    sort(nums, mid + 1, hi);

    // 此时两部分子数组已经被排好序
    // 合并两个有序数组，使 nums[lo..hi] 有序
    merge(nums, lo, mid, hi);
}
```

7. 堆排序（运用二叉堆）

- 堆排序是从 **二叉堆结构** 衍生出来的排序算法，复杂度为 $O(N \log N)$。
- 堆排序主要分两步，第一步是在待排序数组上**原地创建二叉堆**（Heapify），然后进行**原地排序**（Sort）。
- 堆排序是一种**不稳定**的排序算法，因为二叉堆本质上是把数组结构抽象成了二叉树结构，在二叉树逻辑结构上的元素交换操作映射回数组上，无法顾及相同元素的相对位置。
- 分析：
    1. 二叉堆（优先级队列）底层是用**数组**实现的，但是**逻辑上是一棵完全二叉树**，主要依靠 上浮`swim`, 下沉 `sink` 方法来维护堆的性质。

2. 优先级队列可以分为**小顶堆**和**大顶堆**，小顶堆会将整个堆中最小的元素维护在堆顶，大顶堆会将整个堆中最大的元素维护在堆顶。
3. 优先级队列**插入**元素时：首先把元素追加到二叉堆**底部**，然后调用 `swim` 方法把该元素**上浮**到合适的位置，时间复杂度是 $O(\log N)$。
4. 优先级队列**删除**堆顶元素时：首先把堆底的**最后一个元素交换到堆顶**作为新的堆顶元素，然后调用 `sink` 方法把这个新的堆顶元素**下沉**到合适的位置，时间复杂度是 $O(\log N)$。

- 那么最简单的堆排序算法思路就是直接利用优先级队列，把所有元素塞到优先级队列里面，然后再取出来，就完成排序了。
- 优先级队列的 `push`，`pop` 方法的复杂度都是 $O(\log N)$，所以整个排序的时间复杂度是 $O(N \log N)$。

```
// 直接利用优先级队列对数组从小到大排序
void sort(int[] nums) {
    // 创建一个从小到大排序元素的小顶堆
    SimpleMinPQ pq = new SimpleMinPQ(nums.length);
    // 先把所有元素插入到优先级队列中
    for (int num : nums) {
        // push 操作会自动构建二叉堆，时间复杂度为 O(logN)
        pq.push(num);
    }
    // 再把所有元素取出来，就是从小到大排序的结果
    for (int i = 0; i < nums.length; i++) {
        // pop 操作从堆顶弹出二叉堆堆中最小的元素，时间复杂度为 O(logN)
        nums[i] = pq.pop();
    }
}
```

- 堆排序的两个关键步骤：
  1. 原地建堆（Heapify）：直接把待排序数组原地变成一个二叉堆。
  2. 排序（Sort）：将元素不断地从堆中取出，最终得到有序的结果。
- 时间复杂度，假设 `nums` 的元素个数为 `N`：
  1. 第一步建堆的过程中，`swim` 方法的时间复杂度是 $O(\log N)$，算法对每个元素调用一次 `swim` 方法，所以总时间复杂度是 $O(N \log N)$。
  2. 第二步排序的过程中，每次 `sink` 方法的时间复杂度是 $O(\log N)$，算法对每个元素调用一次 `sink` 方法，所以总时间复杂度是 $O(N \log N)$。
     综上，整个堆排序的时间复杂度是 $O(N \log N)$。
- 空间复杂度是 $O(1)$。

```
// 小顶堆的上浮操作，时间复杂度是树高 O(logN)
void minHeapSwim(std::vector<int>& heap, int node) {
    while (node > 0 && heap[parent(node)] > heap[node]) {
        swap(heap, parent(node), node);
        node = parent(node);
    }
}

```

```cpp
// 小顶堆的下沉操作，时间复杂度是树高 O(logN)
void minHeapSink(std::vector<int>& heap, int node, int size) {
    while (left(node) < size || right(node) < size) {
        // 比较自己和左右子节点，看看谁最小
        int min = node;
        if (left(node) < size && heap[left(node)] < heap[min]) {
            min = left(node);
        }
        if (right(node) < size && heap[right(node)] < heap[min]) {
            min = right(node);
        }
        if (min == node) {
            break;
        }
        // 如果左右子节点中有比自己小的，就交换
        swap(heap, node, min);
        node = min;
    }
}

// 大顶堆的上浮操作
void maxHeapSwim(std::vector<int>& heap, int node) {
    while (node > 0 && heap[parent(node)] < heap[node]) {
        swap(heap, parent(node), node);
        node = parent(node);
    }
}

// 大顶堆的下沉操作
void maxHeapSink(std::vector<int>& heap, int node, int size) {
    while (left(node) < size || right(node) < size) {
        // 小顶堆和大顶堆的唯一区别就在这里，比较逻辑相反
        // 比较自己和左右子节点，看看谁最大
        int max = node;
        if (left(node) < size && heap[left(node)] > heap[max]) {
            max = left(node);
        }
        if (right(node) < size && heap[right(node)] > heap[max]) {
            max = right(node);
        }
        if (max == node) {
            break;
        }
        swap(heap, node, max);
        node = max;
    }
}

// 父节点的索引
int parent(int node) {
    return (node - 1) / 2;
}
```

```cpp

// 左子节点的索引
int left(int node) {
    return node * 2 + 1;
}

// 右子节点的索引
int right(int node) {
    return node * 2 + 2;
}

// 交换数组中两个元素的位置
void swap(std::vector<int>& heap, int i, int j) {
    int temp = heap[i];
    heap[i] = heap[j];
    heap[j] = temp;
}

// 将输入的数组元素从小到大排序
void sort(std::vector<int>& nums) {
    // 第一步，原地建堆，注意这里创建的是大顶堆
    // 只要从左往右对每个元素调用 swim 方法，就可以原地建堆
    for (int i = 0; i < nums.size(); i++) {
        maxHeapSwim(nums, i);
    }

    // 第二步，排序
    // 现在整个数组已经是一个大顶了，直接模拟删除堆顶元素的过程即可
    int heapSize = nums.size();
    while (heapSize > 0) {
        // 从堆顶删除元素，放到堆的后面
        swap(nums, 0, heapSize - 1);
        heapSize--;
        // 恢复堆的性质
        maxHeapSink(nums, 0, heapSize);
        // 现在 nums[0..heapSize) 是一个大顶堆，nums[heapSize..) 是有序元素
    }
}
```

- 直接实现堆排序

```cpp
// 将输入的数组元素从小到大排序
void sort(int[] nums) {
    // 第一步，原地建堆，注意这里创建的是大顶堆
    // 只要从左往右对每个元素调用 swim 方法，就可以原地建堆
    for (int i = 0; i < nums.length; i++) {
        maxHeapSwim(nums, i);
    }

    // 第二步，排序
    // 现在整个数组已经是一个大顶了，直接模拟删除堆顶元素的过程即可
```

```
    int heapSize = nums.length;
    while (heapSize > 0) {
        // 从堆顶删除元素，放到堆的后面
        swap(nums, 0, heapSize - 1);
        heapSize--;
        // 恢复堆的性质
        maxHeapSink(nums, 0, heapSize);
        // 现在 nums[0..heapSize) 是一个大顶堆，nums[heapSize..) 是有序元素
    }
}

// maxHeapSink, maxHeapSwim 等函数代码见上文
```

- 再优化：**对于一个二叉堆来说，其左右子堆（子树）也是一个二叉堆。**

```
// 将输入的数组元素从小到大排序
void sort(vector<int>& nums) {
    // 第一步，原地建堆，注意这里创建的是大顶堆
    // 从最后一个非叶子节点开始，依次下沉，合并二叉堆
    int n = nums.size();
    for (int i =  n / 2 - 1; i >= 0; i--) {
        maxHeapSink(nums, i, n);
    }

    // 合并完成，现在整个数组已经是一个大顶堆

    // 第二步，排序，和刚才的代码一样
    int heapSize = n;
    while (heapSize > 0) {
        // 从堆顶删除元素，放到堆的后面
        swap(nums, 0, heapSize - 1);
        heapSize--;
        // 恢复堆的性质
        maxHeapSink(nums, 0, heapSize);
        // 现在 nums[0..heapSize) 是一个大顶堆，nums[heapSize..) 是有序元素
    }
}
```

8. 计数排序（新原理）

- 计数排序的原理比较简单：统计每种元素出现的次数，进而推算出每个元素在排序后数组中的索引位置，最终完成排序。
- 在它的算法思想中同时看到前面讲的 **归并排序** 和 **计数排序** 的影子。
- 计数排序的时间和空间复杂度都是 $O(n + max - min)$，其中n是待排序数组长度，$max - min$是待排序数组的元素范围。
- 处理负数和自定义类型：简单映射技巧

- 非比较排序：计数排序都不需要比较元素的大小，代码中不包含 `if (nums[i] > nums[j])` 这样的比较逻辑,那么到底是什么让他能够完成排序呢？答案是，它依靠数组**索引的有序性**，所以不用对元素进行比较。（也因此，不能使用哈希表作为 `count` 计数数组。
- 计数排序以及后面介绍到的其他非比较排序算法，在特定场景下的时间复杂度是**线性**的$O(n)$，性能会显著高于通用排序算法。

9. 桶排序（博采众长）

- 桶排序算法的核心思想分三步：
  1. 将待排序数组中的元素使用**映射函数**分配到若干个「**桶**」中。
  2. 对**每个桶**中的元素进行**排序**。
  3. 最后将这些排好序的**桶进行合并**，得到排序结果。
- 求模的方法不可行，我们需要用**除法向下取整**的方式来将元素**分配到桶**中。
- 对于桶合并可以有多种排序方法：使用插入排序的桶排序、使用递归的桶排序。
- 桶排序的稳定性主要取决于对**每个桶的排序**算法。
- 空间复杂度是 $O(n+k)$，均摊时间复杂度是 $O(n+k)$，最坏时间复杂度是 $O(n^2)$。

10. 基数排序(Radix Sort)

- 基数排序是 **计数排序** 算法的扩展，它的主要思路是对待排序元素的每一位依次进行计数排序。由于计数排序是稳定的，所以对每一位完成计数排序后，所有元素就完成了排序。
- 基数排序的原理很简单，比方说输入的数组都是三位数 `nums = [329, 457, 839, 439, 720, 355, 350]`，我们先按照**个位**数排序，然后按照**十位**数排序，然后按照**百位**数排序，最终就完成了整个数组的排序。这里面的关键在于，对每一位的排序都必须是稳定排序，否则最终结果就不对了。
- 使用什么稳定排序比较好？计数排序，复杂度$O(n+10)$
- 位数不同怎么办？前缀补 0
- 有负数？正数分数分开排序
- 采用 LSD 算法（Least Significant Digit first，最低位优先）。对应的是 MSD（Most Significant Digit first，最高位优先）
- 对于 LSD 基数排序，由于对每一位的排序都是稳定的，所以最终的排序结果也是**稳定**的。
- 假设待排序数组长度为 n，最大元素的位数为 m，LSD 计数排序本质上就是执行了 m 次计数排序。前文分析过，计数排序的时空复杂度都是 $O(n+max-min)$，在十进制整数的基数排序的场景中，$max-min$ 的值是常数 `10`，可以忽略，所以每次计数排序的时空复杂度都是 $O(n)$。因此，LSD 基数排序的时间复杂度是 $O(mn)$，空间复杂度是 $O(n)$。

```cpp
// 基数排序
void radixSortLSD(std::vector<int>& nums) {
    int min = INT_MAX;
    for (int num : nums) {
        min = std::min(min, num);
    }

    // 根据最小元素，将所有元素转化为从零开始的非负数
```

```cpp
    int offset = -min;
    for (int i = 0; i < nums.size(); i++) {
        nums[i] += offset;
    }

    int max = INT_MIN;
    for (int num : nums) {
        max = std::max(max, num);
    }

    // 计算最大元素的位数
    int maxLen = 0;
    while (max > 0) {
        max /= 10;
        maxLen++;
    }

    // 从低位到高位，依次对每一位进行计数排序
    for (int k = 0; k < maxLen; k++) {
        countSort(nums, k);
    }

    // 将所有元素转化回原来的值
    for (int i = 0; i < nums.size(); i++) {
        nums[i] -= offset;
    }
}

// 基数排序使用的计数排序算法函数
// 已经确保 nums 中的元素都是非负数
// k 是当前需要排序的位数
void countSort(std::vector<int>& nums, int k) {
    // 基数排序中每一位十进制数的取值范围是 0~9
    std::vector<int> count(10, 0);

    // 对每个元素的第 k 位进行计数
    for (int num : nums) {
        int digit = (num / static_cast<int>(std::pow(10, k))) % 10;
        count[digit]++;
    }

    for (int i = 1; i < count.size(); i++) {
        count[i] += count[i - 1];
    }

    // 按照第 k 位的值对元素进行排序
    std::vector<int> sorted(nums.size());
    for (int i = nums.size() - 1; i >= 0; i--) {
        int digit = (nums[i] / static_cast<int>(std::pow(10, k))) % 10;
        sorted[count[digit] - 1] = nums[i];
        count[digit]--;
    }
```

```
61
62      // 把排序结果复制回原数组
63      for (int i = 0; i < nums.size(); i++) {
64          nums[i] = sorted[i];
65      }
66  }
```

第零章、核心刷题框架汇总

(零) 万剑归宗

{% note danger no-icon flat %}

几句话总结一切数据结构和算法：

1. 种种数据结构，皆为**数组**（顺序存储）和**链表**（链式存储）的变换。
2. 数据结构的关键点在于**遍历和访问**，即增删查改等基本操作。
3. 种种算法，皆为**穷举**。
4. 穷举的关键点在于**无遗漏和无冗余**。熟练掌握算法框架，可以做到无遗漏；充分利用信息，可以做到无冗余。

{% btn 'https://labuladong.online/algo/essential-technique/algorithm-summary/#%E6%9C%80%E5%90%8E%E6%80%BB%E7%BB%93',鞭辟入里：原文链接,fa fa-share ,larger outline%}
{% endnote %}

1. 数据结构的存储

- **队列、栈** 这两种数据结构既可以使用链表也可以使用数组实现。用数组实现，就要处理扩容缩容的问题；用链表实现，没有这个问题，但需要更多的内存空间存储节点指针。
- **图结构** 的两种存储方式，邻接表就是链表，邻接矩阵就是二维数组。邻接矩阵判断连通性迅速，并可以进行矩阵运算解决一些问题，但是如果图比较稀疏的话很耗费空间。邻接表比较节省空间，但是很多操作的效率上肯定比不过邻接矩阵。
- **哈希表** 就是通过散列函数把键映射到一个大数组里。而且对于解决散列冲突的方法，拉链法 需要链表特性，操作简单，但需要额外的空间存储指针；线性探查法 需要数组特性，以便连续寻址，不需要指针的存储空间，但操作稍微复杂些。
- **树结构**，用数组实现就是「堆」，因为「堆」是一个完全二叉树，用数组存储不需要节点指针，操作也比较简单，经典应用有 二叉堆；用链表实现就是很常见的那种「树」，因为不一定是完全二叉树，所以不适合用数组存储。为此，在这种链表「树」结构之上，又衍生出各种巧妙的设计，比如 二叉搜索树、AVL 树、红黑树、区间树、B 树等等，以应对不同的问题。

综上，数据结构种类很多，甚至你也可以发明自己的数据结构，但是底层存储无非数组或者链表，二者的优缺点如下：

数组 由于是紧凑连续存储，可以随机访问，通过索引快速找到对应元素，而且相对节约存储空间。但正因为连续存储，内存空间必须一次性分配够，所以说数组如果要扩容，需要重新分配一块更大的空间，再把数据全部复制过去，时间复杂度 O(N)；而且你如果想在数组中间进行插入和删除，每次必须搬移后面的所有数据以保持连续，时间复杂度 $O(N)$。

链表 因为元素不连续，而是靠指针指向下一个元素的位置，所以不存在数组的扩容问题；如果知道某一元素的前驱和后驱，操作指针即可删除该元素或者插入新元素，时间复杂度 $O(1)$。但是正因为存储空间不连续，你无法根据一个索引算出对应元素的地址，所以不能随机访问；而且由于每个元素必须存储指向前后元素位置的指针，会消耗相对更多的储存空间。

2. 数据结构的操作

- **线性**就是 `for/while` 迭代为代表，**非线性**就是 `递归` 为代表。

1. **数组**遍历框架，典型的线性迭代结构：

```
void traverse(vector<int>& arr) {
    for (int i = 0; i < arr.size(); i++) {
        // 迭代访问 arr[i]
    }
}
```

1. **链表**遍历框架，兼具迭代和递归结构：

```
// 基本的单链表节点
class ListNode {
    public:
        int val;
        ListNode* next;
};

void traverse(ListNode* head) {
    for (ListNode* p = head; p != nullptr; p = p->next) {
        // 迭代访问 p->val
    }
}

void traverse(ListNode* head) {
    // 递归访问 head->val
    traverse(head->next);
}
```

3. **二叉树**遍历框架，典型的非线性递归遍历结构：

```
// 基本的二叉树节点
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
};

void traverse(TreeNode* root) {
    traverse(root->left);
    traverse(root->right);
}
```

二叉树框架可以扩展为 N 叉树的遍历框架:

```cpp
// 基本的 N 叉树节点
class TreeNode {
public:
    int val;
    vector<TreeNode*> children;
};

void traverse(TreeNode* root) {
    for (TreeNode* child : root->children)
        traverse(child);
}
```

4. **图**：N 叉树的遍历又可以扩展为图的遍历，因为**图就是好几 N 叉棵树的结合体**。
   图是可能出现环的？这个很好办，用个布尔数组 `visited` 做标记就行了。

3. 算法的本质

- 算法的本质就是「**穷举**」。

4. 穷举的难点

- 穷举有两个关键难点：**无遗漏**、**无冗余**。
- 当你看到一道算法题，可以从这两个维度去思考：
  1. 如何穷举？即无遗漏地穷举所有可能解。
  2. 如何聪明地穷举？即避免所有冗余的计算，消耗尽可能少的资源求出答案。
- 后续会有的系列：
  1. 数组/单链表系列算法：单指针、双指针、二分搜索、滑动窗口、前缀和、差分数组。
  2. 二叉树系列算法：动态规划、回溯（DFS）、层序（BFS）、分治。附加技巧剪枝、备忘录。
  3. 图系列算法：二叉树算法的延续。

(一) 双指针（链表）

> 技巧目录：
> 1、合并两个有序链表
> 2、链表的分解
> 3、合并 k 个有序链表
> 4、寻找单链表的倒数第 k 个节点
> 5、寻找单链表的中点
> 6、判断单链表是否包含环并找出环起点
> 7、判断两个单链表是否相交并找出交点

1、合并两个有序链表

链接：合并两个有序链表

```cpp
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        // 虚拟头结点
        ListNode dummy(-1), *p = &dummy;
        ListNode *p1 = l1, *p2 = l2;

        while (p1 != nullptr && p2 != nullptr) {
            // 比较 p1 和 p2 两个指针
            // 将值较小的的节点接到 p 指针
            if (p1->val > p2->val) {
                p->next = p2;
                p2 = p2->next;
            } else {
                p->next = p1;
                p1 = p1->next;
            }
            // p 指针不断前进
            p = p->next;
        }

        // 合并后 l1 和 l2 最多只有一个还未被合并完，我们直接将链表末尾指向未合并完的链表即可
        if (p1 != nullptr)  p->next = p1;
        if (p2 != nullptr)  p->next = p2;

        return dummy.next;
    }
};
```

> 什么时候需要用虚拟头结点？当你需要**创造一条新链表**的时候，可以使用**虚拟头结点**简化边界情况的处理。

2、单链表的分解

链接：分割链表

```cpp
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        // 存放小于 x 的链表的虚拟头结点
        ListNode* dummy1 = new ListNode(-1);
        // 存放大于等于 x 的链表的虚拟头结点
        ListNode* dummy2 = new ListNode(-1);
        // p1, p2 指针负责生成结果链表
        ListNode* p1 = dummy1, *p2 = dummy2;
        // p 负责遍历原链表，类似合并两个有序链表的逻辑
        // 这里是将一个链表分解成两个链表
        ListNode* p = head;
```

```
        while (p != nullptr) {
            if (p->val >= x) {
                p2->next = p;
                p2 = p2->next;
            } else {
                p1->next = p;
                p1 = p1->next;
            }
            // 不能直接让 p 指针前进，
            // p = p->next
            // 断开原链表中的每个节点的 next 指针
            ListNode* temp = p->next;
            p->next = nullptr;
            p = temp;
        }
        // 连接两个链表
        p1->next = dummy2->next;

        return dummy1->next;
    }
};
```

3、合并 k 个有序链表

链接：[合并k个有序链表]

- 合并 k 个有序链表的逻辑类似合并两个有序链表，难点在于，如何快速得到 k 个节点中的最小节点，接到结果链表上？
  这里我们就要用到**优先级队列**这种数据结构，把链表节点放入一个**最小堆**，就可以**每次获得 k 个节点中的最小节点**。
- 优先队列 pq 中的元素个数最多是 k，所以一次 `push` 或者 `pop` 方法的时间复杂度是 $O(\log k)$；所有的链表节点都会被加入和弹出 pq，所以算法整体的时间复杂度是 $O(N \log k)$其中k是链表的条数，N是这些链表的节点总数。

```
class Solution {
public:
    // 关键优化：使用优先队列快速找到 k 个头节点的最小值
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if(lists.empty())  return NULL;
        // 准备虚拟头节点
        ListNode* dumy = new ListNode();
        ListNode* p = dumy;

        // 小根堆
        auto cmp = [](ListNode* a, ListNode* b){ return a->val > b->val; };
        priority_queue<ListNode*, vector<ListNode*>, decltype(cmp)> heap;
        // 初始存入 k 个的头节点
        for(ListNode* t: lists)  if(t != NULL)  heap.push(t);

        // 开始连接
```

```
        while(!heap.empty()){
            // 最小的先连接
            ListNode* minp = heap.top();
            heap.pop();
            p->next = minp;
            p = p->next;

            // 待选节点存入一个新的
            if(minp->next != NULL)  heap.push(minp->next);
        }

        // 最终结果
        p->next = NULL;
        return dummy->next;
    }
};
```

4、单链表的倒数第 k 个节点

- 参考：类似题

那你可能说，假设链表有 `n` 个节点，倒数第 `k` 个节点就是正数第 `n - k + 1` 个节点，不也是一个 `for` 循环的事儿吗?

这个解法需要遍历两次链表才能得到出倒数第 `k` 个节点。

能不能**只遍历一次**链表，就算出倒数第 k 个节点?

```
class Solution {
public:
    // 找到倒数第k个节点
    ListNode* findFromEnd(ListNode* head, int k){
        // p1先走 k 步
        ListNode* p1 = head;
        for(int i=0; i<k; i++)  p1 = p1->next;

        // p1, p2共同走 n-k 步
        ListNode* p2 = head;
        while(p1 != NULL){
            p1 = p1->next;
            p2 = p2->next;
        }

        return p2;
    }

    ListNode* removeNthFromEnd(ListNode* head, int n) {
        // 虚拟头节点
        ListNode* dummy = new ListNode();
        dummy->next = head;

```

```
        // 删除倒数第k个，要找到倒数第k+1个
        ListNode* p = findFromEnd(dummy, n+1);
        p->next = p->next->next;

        return dummy->next;
    }
};
```

无论遍历一次链表和遍历两次链表的时间复杂度都是 $O(N)$，但上述这个算法更有**技巧性**。

> 我们又使用了虚拟头结点的技巧，也是为了防止出现空指针的情况，比如说链表总共有 5 个节点，题目就让你删除倒数第 5 个节点，也就是第一个节点，那按照算法逻辑，应该首先找到倒数第 6 个节点。但第一个节点前面已经没有节点了，这就会出错。

5、单链表的中点

参考：中点

我们让两个指针 `slow` 和 `fast` 分别指向链表头结点 `head`。每当慢指针 `slow` 前进一步，快指针 `fast` 就前进两步，这样，当 `fast` 走到链表末尾时，`slow` 就指向了链表中点。

```
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        // 快慢指针初始化指向 head
        ListNode* slow = head;
        ListNode* fast = head;
        // 快指针走到末尾时停止
        while (fast != nullptr && fast->next != nullptr) {
            // 慢指针走一步，快指针走两步
            slow = slow->next;
            fast = fast->next->next;
        }
        // 慢指针指向中点
        return slow;
    }
};
```

6、判断链表是否包含环

参考：判断环

- 判断链表是否包含环属于经典问题了，解决方案也是用快慢指针：
  每当慢指针 `slow` 前进一步，快指针 `fast` 就前进两步。
  - 如果 `fast` 最终能正常走到链表末尾，说明链表中没有环；
  - 如果 `fast` 走着走着竟然和 `slow` 相遇了，那肯定是 `fast` 在链表中转圈了，说明链表中含有环。
- 如果链表中含有环，如何计算这个环的起点?

- `fast` 一定比 `slow` 多走了 `k` 步，这多走的 `k` 步其实就是 `fast` 指针在环里转圈圈，所以 `k` 的值就是**环长度**的「整数倍」。
- 假设相遇点距环的起点的距离为 `m`，那么结合上图的 `slow` 指针，环的起点距头结点 `head` 的距离为 `k - m`，也就是说如果从 `head` 前进 `k - m` 步就能到达环起点。
- 巧的是，如果从相遇点继续前进 `k - m` 步，也恰好到达环起点。因为结合上图的 `fast` 指针，从相遇点开始走 `k` 步可以转回到相遇点，那走 `k - m` 步肯定就走到环起点了

- 所以，只要我们把快慢指针中的任一个重新指向 `head`，然后两个指针同速前进，`k - m` 步后一定会相遇，相遇之处就是环的起点了。

```cpp
class Solution {
public:
    // 快慢指针，两次相遇即可
    ListNode *detectCycle(ListNode *head) {
        ListNode* slow = head;
        ListNode* fast = head;
        // 第一次相遇：快慢
        while(fast != NULL && fast->next != NULL){
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast)  break;    // 相遇
        }
        // 无环
        if(fast == NULL || fast->next == NULL)  return NULL;

        // 第二次相遇：一起走
        slow = head;
        while(slow != fast){
            slow = slow->next;
            fast = fast->next;
        }
        // 入环点
        return slow;
    }
};
```

7、两个链表是否相交

参考：相交

- 这个题直接的想法可能是用 `HashSet` 记录**一个链表的所有节点**，然后和**另一条链表对比**，但这就需要额外的空间。
- 如果不用额外的空间，只使用两个指针，你如何做呢？难点在于，由于两条链表的长度可能不同，两条链表之间的节点无法对应。

- 如果用两个指针 `p1` 和 `p2` 分别在两条链表上前进，并不能同时走到公共节点，也就无法得到相交节点 `c1`。解决这个问题的关键是，**通过某些方式，让 `p1` 和 `p2` 能够同时到达相交节点 `c1`**。所以，我们可以让 `p1` 遍历完链表 A 之后开始遍历链表 B，让 `p2` 遍历完链表 B 之后开始遍历链表 A，这样相当于「逻辑上」两条链表接在了一起。如果这样进行拼接，就可以让 p1 和 p2 同时进入公共部分，也就是同时到达相交节点 c1：

```
class Solution {
public:
    // 拼接思想
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        ListNode *p1 = headA, *p2 = headB;
        while(p1 != p2){
            if(p1 != NULL)  p1 = p1->next;
            else  p1 = headB;
            if(p2 != NULL)  p2 = p2->next;
            else p2 = headA;
        }
        return p1;
    }
};
```

(二) 双指针（数组）

> 一、快慢指针技巧
>
> 1. 原地修改
> 2. 滑动窗口
>
> 二、左右指针的常用算法
>
> 3. 二分查找
> 4. n 数之和
> 5. 反转数组
> 6. 回文串判断

1、原地修改

参考：[删除有序数组中的重复项]

- 数组问题中比较常见的快慢指针技巧，是让你原地修改数组。
- 由于数组已经排序，所以重复的元素一定连在一起，找出它们并不难。但如果每找到一个重复元素就立即原地删除它，由于数组中删除元素涉及**数据搬移**，整个时间复杂度是会达到 $O(N^2)$。
- 们让慢指针 `slow` 走在后面，快指针 `fast` 走在前面探路，找到一个不重复的元素就赋值给 `slow` 并让 `slow` 前进一步。这样，就保证了 `nums[0..slow]` 都是无重复的元素，当 `fast` 指针遍历完整个数组 `nums` 后，`nums[0..slow]` 就是整个数组去重之后的结果。

```
class Solution {
public:
```

```
    // 快慢指针
    int removeDuplicates(vector<int>& nums) {
        int n = nums.size();
        if(n == 0)  return 0;
        // fast 探路，有不重复的数字，交给 slow 填
        int slow = 0, fast = 0;
        while(fast < n){
            if(nums[fast] != nums[slow])
                nums[++slow] = nums[fast];
            fast++;
        }
        // 最终 0..slow 就是答案
        return slow + 1;
    }
};
```

> 再简单扩展一下，看看力扣第 83 题「删除排序链表中的重复元素」，如果给你一个有序的单链表，如何去重呢？其实和数组去重是一模一样的，唯一的区别是把数组赋值操作变成操作指针而已。

```
class Solution {
public:
    // 快慢指针
    ListNode* deleteDuplicates(ListNode* head) {
        if(head == NULL)  return NULL;
        // fast 探路，slow 填写
        ListNode *slow = head, *fast = head;
        while(fast != NULL){
            if(fast->val != slow->val){
                slow->next = fast;
                slow = slow->next;
            }
            fast = fast->next;
        }
        // 注意切断
        slow->next = NULL;
        return head;
    }
};
```

- 除了让你在有序数组/链表中去重，题目还可能让你对数组中的某些元素进行「**原地删除**」。
- 参考:移除元素
- 题目要求我们把 `nums` 中所有值为 `val` 的元素原地删除，依然需要使用快慢指针技巧：
  如果 `fast` 遇到值为 `val` 的元素，则直接跳过，否则就赋值给 `slow` 指针，并让 `slow` 前进一步。

```cpp
class Solution {
public:
    // 快慢指针
    int removeElement(vector<int>& nums, int val) {
        int fast = 0, slow = 0;
        while(fast < nums.size()){
            if(nums[fast] != val)
                nums[slow++] = nums[fast];
            fast++;
        }
        return slow;
    }
};
```

- 给你输入一个数组 `nums`，请你原地修改，将数组中的所有值为 `0` 的元素移到数组末尾。
- 参考：移动零

```cpp
class Solution {
public:
    // 快慢指针
    void moveZeroes(vector<int>& nums) {
        int fast = 0, slow = 0;
        while(fast < nums.size()){
            if(nums[fast] != 0)
                nums[slow++] = nums[fast];
            fast++;
        }
        while(slow < nums.size())
            nums[slow++] = 0;
    }
};
```

到这里，原地修改数组的这些题目就已经差不多了。

2、滑动窗口

- 数组中另一大类快慢指针的题目就是「滑动窗口算法」。在下文 【（三）滑动窗口】给出了滑动窗口的代码框架：

```
// 滑动窗口伪代码
int left = 0, right = 0;
while(right < nums.size()){
    // 增大窗口
    window.push_back(nums[right]);
    right++;

    // 缩小窗口
    while(window needs shrink){
        window.pop_front(); // nums[left]
        left++;
    }
}
```

- 具体的题目本文就不重复了，这里只强调滑动窗口算法的快慢指针特性：
  `left` 指针在后，`right` 指针在前，两个指针中间的部分就是「**窗口**」，算法通过扩大和缩小「**窗口**」来解决某些问题。

3、二分查找

- 在另一篇文章 【二分查找框架详解】 中有详细探讨二分搜索代码的细节问题，这里只写最简单的二分算法，旨在突出它的双指针特性：

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

int binarySearch(vector<int>& nums, int target) {
    // 一左一右两个指针相向而行
    int left = 0, right = nums.size() - 1;
    while(left <= right) {
        int mid = (right + left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else if (nums[mid] > target)
            right = mid - 1;
    }
    return -1;
}
```

4、n 数之和

- 参考：两数之和 II
- 给你一个下标从 `1` 开始的整数数组 `numbers` ，该数组已按 **非递减顺序排列** ，请你从数组中找出满足相加之和等于目标数 `target` 的两个数。如果设这两个数分别是 `numbers[index1]` 和 `numbers[index2]` ，则 `1 <= index1 < index2 <= numbers.length` 。

- 只要数组有序，就应该想到双指针技巧。这道题的解法有点类似二分查找，通过调节 left 和 right 就可以调整 sum 的大小：

```
class Solution {
public:
    // 左右指针
    vector<int> twoSum(vector<int>& numbers, int target) {
        int left = 0, right = numbers.size()-1;
        while(left < right){
            int sum = numbers[left] + numbers[right];
            // 完成
            if(sum == target)  return vector<int>{left+1, right+1}; // 题目索引从1开始
            // 让 sum 小一点
            else if(sum > target)  right--;
            // 让 sum 大一点
            else  left++;
        }

        return vector<int>{-1, -1};
    }
};
```

- 在另一篇文章 【一个函数秒杀所有nSum问题】 中也运用类似的左右指针技巧给出了 nSum 问题的一种通用思路，本质上利用的也是双指针技巧。

5、反转数组

- 一般编程语言都会提供 `reverse` 函数，其实这个函数的原理非常简单，力扣第 344 题「反转字符串」就是类似的需求，让你反转一个 char[] 类型的字符数组，我们直接看代码吧：
- 参考：反转字符串

```
class Solution {
public:
    // 左右指针
    void reverseString(vector<char>& s) {
        int left = 0, right = s.size()-1;
        while(left < right){
            char t = s[left];
            s[left] = s[right];
            s[right] = t;
            left++, right--;
        }
    }
};
```

- 关于数组翻转的更多进阶问题，可以参见 【二维数组的花式遍历】 。

6、回文串判断

- 判断很简单

```
bool isPalindrome(string s){
    int left  = 0, right = s.size()-1;
    while(left < right){
        if(s[left] != s[right])  return false;
        left++, right--;
    }
    return true;
}
```

- 提升一点难度，给你一个字符串，让你用双指针技巧从中找出**最长的回文串**?
- 参考：[最长回文子串]

```
class Solution {
public:
    // 找到以 l, r 为中心的最长子回文串(若l=r，则说明以一个字符为中心)
    string palindrome(string& s, int l, int r){
        // 注意越界
        while(l>=0 && r<s.size() && s[l]==s[r])
            l--, r++;
        // 返回
        return s.substr(l+1, r-l-1);    // 注意，退出循环时，l,r 都扩大了
    }

    string longestPalindrome(string s) {
        string ans = "";
        for(int i=0; i<s.size(); i++){
            // 中心扩散
            string s1 = palindrome(s, i, i);
            string s2 = palindrome(s, i, i+1);
            // 更新答案
            ans = s1.size() > ans.size() ? s1 : ans;
            ans = s2.size() > ans.size() ? s2 : ans;
        }
        return ans;
    }
};
```

> 你应该能发现最长回文子串使用的左右指针和之前题目的左右指针有一些不同：之前的左右指针都是从两端向中间相向而行，而回文子串问题则是让左右指针从中心向两端扩展。不过这种情况也就回文串这类问题会遇到，所以我也把它归为左右指针了。

(三) 滑动窗口

- 滑动窗口可以归为快慢双指针，一快一慢两个指针前后相随，中间的部分就是窗口。
- **滑动窗口算法技巧主要用来解决子数组问题，比如让你寻找符合某个条件的最长/最短子数组。**

1、框架概述

- 如果用暴力解的话，你需要嵌套 `for` 循环这样穷举所有子数组，时间复杂度是 $O(N^2)$：

```
for (int i = 0; i < nums.size(); i++) {
    for (int j = i; j < nums.size(); j++) {
        // nums[i, j] 是一个子数组
    }
}
```

- 滑动窗口算法技巧的思路也不难，就是维护一个窗口，不断滑动，然后更新答案，该算法的大致逻辑如下：

```
// 滑动窗口伪代码
int left = 0, right = 0;
while(right < nums.size()){
    // 增大窗口
    window.push_back(nums[right]);
    right++;

    // 缩小窗口
    while(window needs shrink){
        window.pop_front(); // nums[left]
        left++;
    }
}
```

基于滑动窗口算法框架写出的代码，时间复杂度是 $O(N)$，比嵌套 `for` 循环的暴力解法效率高。

> 1. 为啥是 $O(N)$？
>
> - 简单说，指针 `left`，`right` 不会回退（它们的值只增不减），所以字符串/数组中的每个元素都只会进入窗口一次，然后被移出窗口一次。
> - 反观嵌套 `for` 循环的暴力解法，那个 `j` 会回退，所以某些元素会进入和离开窗口多次，所以时间复杂度就是 $O(N^2)$ 了。
>
> 2. 为啥滑动窗口能在 $O(N)$ 的时间穷举子数组？
>
> - 这个问题本身就是错误的，**滑动窗口并不能穷举出所有子串**。要想穷举出所有子串，必须用那个嵌套 `for` 循环。
> - 然而对于某些题目，**并不需要穷举所有子串**，就能找到题目想要的答案。滑动窗口就是这种场景下的一套算法模板，帮你对穷举过程进行剪枝优化，避免冗余计算。

因为本文的例题大多是子串相关的题目，字符串实际上就是数组，所以我就把输入设置成字符串了。你做题的时候根据具体题目自行变通即可：

```
// 滑动窗口算法伪码框架
void slidingWindow(string s) {
    // 用合适的数据结构记录窗口中的数据，根据具体场景变通
    // 比如说，我想记录窗口中元素出现的次数，就用 map
    // 如果我想记录窗口中的元素和，就可以只用一个 int
```

```
    auto window = ...

    int left = 0, right = 0;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        window.add(c);
        // 增大窗口
        right++;

        // 进行窗口内数据的一系列更新
        ...

        // *** debug 输出的位置 ***
        printf("window: [%d, %d)\n", left, right);
        // 注意在最终的解法代码中不要 print
        // 因为 IO 操作很耗时，可能导致超时

        // 判断左侧窗口是否要收缩
        while (window needs shrink) {
            // d 是将移出窗口的字符
            char d = s[left];
            window.remove(d);
            // 缩小窗口
            left++;

            // 进行窗口内数据的一系列更新
            ...
        }
    }
}
```

框架中两处 `...` 表示的更新窗口数据的地方，在具体的题目中，你需要做的就是往这里面填代码逻辑。

2、最小覆盖子串

- 参考:[最小覆盖子串]
- 给你一个字符串 `s` 、一个字符串 `t` 。返回 `s` 中涵盖 `t` 所有字符的最小子串。如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `""` 。
- 注：对于 `t` 中重复字符，我们寻找的子字符串中该字符数量必须不少于 `t` 中该字符数量。如果 `s` 中存在这样的子串，我们保证它是唯一的答案。
- 示例：
  输入：`s = "ADOBECODEBANC", t = "ABC"`
  输出：`"BANC"`

滑动窗口算法的思路是这样：

1. 我们在字符串 `S` 中使用双指针中的左右指针技巧，初始化 `left = right = 0`，把索引左闭右开区间 `[left, right)` 称为一个「窗口」。

> 为什么要「**左闭右开**」区间?
>
> - 理论上你可以设计两端都开或者两端都闭的区间，但设计为左闭右开区间是最方便处理的。
> - 因为这样初始化 `left = right = 0` 时区间 `[0, 0)` 中没有元素，但只要让 `right` 向右移动（扩大）一位，区间 `[0, 1)` 就包含一个元素 `0` 了。
> - 另外，注意：**当前窗口长度，就是 `right-left`，不需要`right-left+1`的**。

2. 我们先不断地增加 `right` 指针扩大窗口 `[left, right)`，直到窗口中的字符串符合要求（包含了 `T` 中的所有字符）。
3. 此时，我们停止增加 `right`，转而不断增加 `left` 指针缩小窗口 `[left, right)`，直到窗口中的字符串不再符合要求（不包含 `T` 中的所有字符了）。同时，每次增加 `left`，我们都要更新一轮结果。
4. 重复第 `2` 和第 `3` 步，直到 `right` 到达字符串 `S` 的尽头。

这个思路其实也不难，**第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解**，也就是最短的覆盖子串。左右指针轮流前进，窗口大小增增减减，就好像一条毛毛虫，一伸一缩，不断向右滑动，这就是「滑动窗口」这个名字的来历。

- 现在开始套模板，只需要思考以下几个问题：
  1. 什么时候应该移动 right 扩大窗口？窗口加入字符时，应该更新哪些数据?
  2. 什么时候窗口应该暂停扩大，开始移动 left 缩小窗口？从窗口移出字符时，应该更新哪些数据?
  3. 我们要的结果应该在扩大窗口时还是缩小窗口时进行更新?

```cpp
class Solution {
public:
    // 滑动窗口
    string minWindow(string s, string t) {
        int n = s.size();
        // 创建两个哈希表，用于检测是否达到要求
        unordered_map<char, int> need, window;
        for(char& c: t)  need[c]++;     // debug 半天，发现这里 t 写成 s 了。。。

        // 窗口定义左闭右开 [left, right)
        int left = 0, right = 0;
        int valid = 0;          // 已经完成要求的不同字符数量
        int start = 0, len = n+1;
        while(right < n){
            // 增大窗口
            char c = s[right];
            right++;
            // 更新数据
            if(need.count(c)){
                window[c]++;
                if(window[c] == need[c])  valid++;
            }

            // 是否已达到要求
```

```
            while(valid == need.size()){
                // 先更新答案
                if(right-left < len){    // 长度不用 r-l+1，因为r本身就是更大 1
                    start = left;
                    len = right-left;
                }
                // 缩小窗口
                char d = s[left];
                left++;
                // 更新数据
                if(need.count(d)){
                    if(window[d] == need[d])  valid--;
                    window[d]--;
                }
            }
        }

        // 返回答案
        return len>n ? "" : s.substr(start, len);
    }
};
```

> 这里再强调一下，里面的`if(window[d] == need[d])`用的很妙的，保证了不会多加少加、多减少减。

3、字符串排列

- 参考：字符串的排列
- 给你两个字符串 `s1` 和 `s2` ，写一个函数来判断 `s2` 是否包含 `s1` 的 **排列**(排列是字符串中所有字符的重新排序。)。如果是，返回 `true` ；否则，返回 `false` 。换句话说， `s1` 的排列之一是 `s2` 的 子串 。
- 示例1：
  输入：`s1 = "ab" s2 = "eidbaooo"`
  输出：`true`
  解释：`s2 包含 s1 的排列之一 ("ba").`
- 示例2：
  输入：`s1= "ab" s2 = "eidboaoo"`
  输出：`false`

> 相当给你一个 S 和一个 T，请问你 S 中是否存在一个子串，包含 T 中所有字符且不包含其他字符?

```
class Solution {
public:
    // 滑动窗口
    bool checkInclusion(string s1, string s2) {
        int n1 = s1.size(), n2 = s2.size();
        // 需要的数据
        unordered_map<char, int> need, window;
        for(char& c: s1)  need[c]++;
        int valid = 0;
        int left = 0, right = 0;
```

```
        while(right < n2){
            // 扩大
            char c = s2[right];
            right++;
            if(need.count(c)){
                window[c]++;
                if(window[c] == need[c])  valid++;
            }
            // 注意收缩条件不一样
            if(right-left == n1){
                // 完成
                if(valid == need.size())  return true;
                // 缩小
                char d = s2[left];
                left++;
                if(need.count(d)){
                    if(window[d] == need[d])  valid--;
                    window[d]--;
                }
            }
        }
        // 最终出来肯定是失败
        return false;
    }
};
```

> 其实，你会发现，若是匹配的小窗是固定长度的，那么里面收缩的条件就是 `if` 而不是 `while`，当然你使用 `while (right - left >= t.size())` 也是能做的，只不过实际上就只执行一次。

4、找所有字母异位词

- 参考：[找到字符串中所有字母异位词]
- 给定两个字符串 `s` 和 `p`，找到 `s` 中所有 `p` 的 **异位词**(字母异位词是通过重新排列不同单词或短语的字母而形成的单词或短语，并使用所有原字母一次) 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。
- 示例:
  输入: `s = "cbaebabacd", p = "abc"`
  输出: `[0,6]`
  解释:起始索引等于 `0` 的子串是 `"cba"`, 它是 `"abc"` 的异位词。起始索引等于 `6` 的子串是 `"bac"`, 它是 `"abc"` 的异位词。

```
class Solution {
public:
    // 滑动窗口
    vector<int> findAnagrams(string s, string p) {
        int n = s.size(), m = p.size();
        // 所需数据
        unordered_map<char, int> need, window;
        for(char &t: p)  need[t]++;
        int valid = 0;
        vector<int> ans;
```

```
11
12          int left = 0, right = 0;
13          while(right < n){
14              // 增大
15              char c = s[right];
16              right++;
17              if(need.count(c)){
18                  window[c]++;
19                  if(window[c] == need[c])  valid++;
20              }
21
22              // 缩小
23              if(right-left == m){
24                  if(valid == need.size())  ans.push_back(left);
25                  char d = s[left];
26                  left++;
27                  if(need.count(d)){
28                      if(window[d] == need[d])  valid--;
29                      window[d]--;
30                  }
31              }
32          }
33
34          // 返回答案
35          return ans;
36      }
37  };
```

5、最长无重复子串

- 参考：[无重复字符的最长子串]
- 给定一个字符串 `s` ，请你找出其中不含有重复字符的 **最长** 子串(连续的非空字符序列) 的长度。
- 示例：
  输入: `s = "abcabcbb"`
  输出: `3`
  解释: 因为无重复字符的最长子串是 `"abc"`，所以其长度为 `3`。

下面是我的解法：

```
1   class Solution {
2   public:
3       // 滑动窗口
4       int lengthOfLongestSubstring(string s) {
5           int n = s.size();
6           // 所需数据
7           unordered_set<char> window;     // 只需要哈希集合即可
8           int ans = 0;
9
10          int left = 0, right = 0;
11          while(right < n){
```

```
            // 增大
            char c = s[right];
            right++;

            // 存在，缩小
            while(window.find(c) != window.end()){
                char d = s[left];
                left++;
                window.erase(d);
            }

            // 现在不存在了，可插入
            if(window.find(c) == window.end()){
                window.insert(c);
                ans = max(ans, right-left);
            }
        }

        // 返回答案
        return ans;
    }
};
```

如若套用前面模版，也可以：

```
class Solution {
public:
    // 滑动窗口
    int lengthOfLongestSubstring(string s) {
        int n = s.size();
        // 所需数据
        unordered_map<char, int> window;
        int ans = 0;

        int left = 0, right = 0;
        while(right < n){
            // 增大
            char c = s[right];
            right++;
            window[c]++;

            // 缩小
            while(window[c] > 1){
                char d = s[left];
                left++;
                window[d]--;
            }

            // 更新答案
            ans = max(ans, right-left);
        }
```

```
27
28          // 返回答案
29          return ans;
30      }
31  };
```

你只要能回答出来以下几个问题，就能运用滑动窗口算法：

1. 什么时候应该扩大窗口？
2. 什么时候应该缩小窗口？
3. 什么时候应该更新答案？

> 更多强化经典题：滑动窗口-强化
>
> 1. 将 x 减到 0 的最小操作数
> 2. 乘积小于 K 的子数组
> 3. 最大连续1的个数 III
> 4. 替换后的最长重复字符
>
> 建议在第二天来完成。

(四) 二分搜索

1、代码框架

```cpp
int binarySearch(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;

    while (...) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            ...
        } else if (nums[mid] < target) {
            left = ...
        } else if (nums[mid] > target) {
            right = ...
        }
    }
}
```

> 分析二分查找的一个技巧是：不要出现 `else`，而是把所有情况用 `else if` 写清楚，这样可以清楚地展现所有细节。
>
> 其中 `...` 标记的部分，就是可能出现细节问题的地方，当你见到一个二分查找的代码时，首先注意这几个地方。
>
> 另外提前说明一下，计算 `mid` 时需要防止溢出，代码中 `left + (right - left) / 2` 就和 `(left + right) / 2` 的结果相同，但是有效防止了 `left` 和 `right` 太大，直接相加导致溢出的情况。

2、寻找一个数（基本）

- 参考：二分查找
- 即搜索一个数，如果存在，返回其索引，否则返回 `-1`。

```cpp
class Solution {
public:
    // 标准的二分搜索框架，返回目标元素的索引，不存在则返回 -1
    int search(vector<int>& nums, int target) {
        int left = 0, right = nums.size()-1;     // 注意
        while(left <= right){                    // 注意
            int mid = left + (right-left)/2;
            if(nums[mid] == target)
                return mid;
            else if(nums[mid] < target)
                left = mid+1;                    // 注意
            else if(nums[mid] > target)
                right = mid-1;                   // 注意
        }
        return -1;
    }
};
```

> 分析：
> 我们这个算法中使用的是 `[left, right]` 两端都闭的区间，**这个区间其实就是每次进行搜索的区间**。那么：
>
> 1. `while` 循环的条件是 `<=` 而不是 `<`。因为 `[4,4]` 是有元素的，而 `[5,4]` 才没有元素了，就停止。
> 2. 初始化 `right` 是 `n-1`，而不是 `n`。因为初始是 `[0,n-1]` 才是可以搜索的空间，而 `[0,n]` 的最后 `n` 是不可搜索的。
>
> 若是采用 `[left,right)` 那么你可以分析得到循环结束，应该是 类似 `[4,4)` 也就是 `while` 用 `left < right` 就结束了。同时初始化应该是 `[0,n)` 刚好把所有元素包含且不含最后的 `n`。
>
> 为什么是 `left = mid + 1, right = mid - 1`?
>
> - 明确了「**搜索区间**」这个概念，而且本算法的搜索区间是两端都闭的，即 `[left, right]`。那么当我们发现索引 `mid` 不是要找的 `target` 时，下一步应该去搜索哪里呢？当然是去搜索区间 `[left, mid-1]` 或者区间 `[mid+1, right]` 对不对？因为 `mid` 已经搜索过，应该从搜索区间中去除。
>
> 此算法有什么缺陷?
>
> - 比如说给你有序数组 `nums = [1,2,2,2,3]`，`target` 为 `2`，此算法返回的索引是 `2`，没错。但是如果我想得到 `target` 的左侧边界，即索引 `1`，或者我想得到 `target` 的右侧边界，即索引 `3`，这样的话此算法是无法处理的。
> - 这样的需求很常见，你也许会说，找到一个 `target`，然后向左或向右线性搜索不行吗？可以，但是不好，因为这样难以保证二分查找对数级的复杂度了。

3、寻找左边界

```cpp
int left_bound(vector<int> &nums, int target){
    int left = 0, right = nums.size();  // 注意 [left, right)
    while(left < right){                // 注意
        int mid  = left + (right-left)/2;
        if(nums[mid] == target)
            right = mid;
        else if(nums[mid] < target)
            left = mid+1;               // 注意
        else if(nums[mid] > target)
            right = mid;                // 注意
    }
    return left;
}
```

> **如果 `target` 不存在，搜索左侧边界的二分搜索返回的索引是大于 `target` 的最小索引。**
>
> - 举个例子，`nums = [2,3,5,7]`，`target = 4`，`left_bound` 函数返回值是 `2`，因为元素 `5` 是大于 `4` 的最小元素。
>
> 为什么是 `left = mid + 1` 和 `right = mid`?
>
> - 「**搜索区间**」是 `[left, right)` 左闭右开，所以当 `nums[mid]` 被检测之后，下一步应该去 `mid` 的左侧或者右侧区间搜索，即 `[left, mid)` 或 `[mid + 1, right)`。
>
> 为什么该算法能够搜索左侧边界?
>
> - 关键在于对于 `nums[mid] == target` 这种情况的处理是 `right = mid;`可见，找到 `target` 时不要立即返回，而是缩小「搜索区间」的上界 `right`，在区间 `[left, mid)` 中继续搜索，即不断向左收缩，达到锁定左侧边界的目的。
>
> 为什么返回 `left` 而不是 `right`?
>
> - 一样，都可以。终止条件是 `left == right`。

可以直接拿来写 `floor` 函数。

```cpp
// 在一个有序数组中，找到「小于 target 的最大元素的索引」
// 比如说输入 nums = [1,2,2,2,3]，target = 2，函数返回 0，因为 1 是小于 2 的最大元素。
// 再比如输入 nums = [1,2,3,5,6]，target = 4，函数返回 2，因为 3 是小于 4 的最大元素。
int floor(vector<int> &nums, int target) {
    // 当 target 不存在，比如输入 [4,6,8,10]，target = 7
    // left_bound 返回 2，减一就是 1，元素 6 就是小于 7 的最大元素
    // 当 target 存在，比如输入 [4,6,8,8,8,10]，target = 8
    // left_bound 返回 2，减一就是 1，元素 6 就是小于 8 的最大元素
    return left_bound(nums, target) - 1;
}
```

> 但是，如果非必要，不要自己手写，**尽可能用编程语言提供的标准库函数**，可以节约时间，而且标准库函数的行为在文档里都有明确的说明，不容易出错。

如果想让 `target` 不存在时返回 `-1` 其实很简单，在返回的时候额外判断一下 `nums[left]` 是否等于 `target` 就行了，如果不等于，就说明 `target` 不存在。需要注意的是，访问数组索引之前要保证索引不越界

```
// 如果索引越界，说明数组中无目标元素，返回 -1
if (left < 0 || left >= nums.length)
    return -1;
// 判断一下 nums[left] 是不是 target
return nums[left] == target ? left : -1;
```

> 提示：其实上面的 `if` 中 `left < 0` 这个判断可以省略，因为对于这个算法，`left` 不可能小于 `0`，你看这个算法执行的逻辑，`left` 初始化就是 `0`，且只可能一直往右走，那么只可能在右侧越界。不过我这里就同时判断了，因为在访问数组索引之前保证索引在左右两端都不越界是一个好习惯，没有坏处。另一个好处是让二分的模板更统一，降低你的记忆成本，因为等会儿寻找右边界的时候也有类似的出界判断。

只要明白了**搜索区间**的概念，实际上，可以统一一下，仍然使用左右都闭的。

```
int left_bound(vector<int> &nums, int target){
    int left = 0, right = nums.size()-1;    // 注意，采用区间[left, right]
    while(left <= right){                   // 注意
        int mid = left + (right-left)/2;
        if(nums[mid] == target)
            right = mid-1;  // 收紧右边界，不用怕找到小的了，因为后面返回的是left
        else if(nums[mid] < target)
            left = mid+1;   // 区间改为 [mid+1, right]
        else if(nums[mid] > target)
            right = mid-1;  // 区间改为 [left, mid-1]
    }
    // 第 1 种返回
    // return left;

    // 第 2 种返回
    if(left<0 || right>=nums.size())  return -1;
    return nums[left] == target? left : -1;
}
```

4、寻找右边界

先还是看左闭右开的代码：

```
int right_bound(vector<int> &nums, int target){
    int left = 0, right = nums.size();  // [left, right)
    while(left < right){                // 注意
        int mid = left + (right-left)/2;
        if(nums[mid] == target)
            left = mid+1;   // 收紧，去查 [mid+1, right)
        else(nums[mid] < target)
            left = mid+1;   // 进入 [mid+1, right)
        else(nums[mid] > target)
            right = mid;    // 进入 [left, mid)
    }
    // 注意
    return left-1;
}
```

> 为什么返回 `left - 1`？为什么不是返回 `right`？
>
> - 终止条件是 `left == right`，所以 `left` 和 `right` 是一样的，你非要体现右侧的特点，返回 `right - 1` 好了。
> - 为什么要减一，这是搜索右侧边界的一个特殊点，关键在锁定右边界时的这个条件判断：`left = mid + 1;`。
> - 因为我们对 `left` 的更新必须是 `left = mid + 1`，就是说 `while` 循环结束时，`nums[left]` 一定不等于 `target` 了，而 `nums[left-1]` 可能是 `target`。
> - 至于为什么 `left` 的更新必须是 `left = mid + 1`，当然是为了把 `nums[mid]` 排除出搜索区间，这里就不再赘述。
>
> **如果 `target` 不存在，搜索右侧边界的二分搜索返回的索引是小于 `target` 的最大索引。**
>
> - 比如 `nums = [2,3,5,7]`，`target = 4`，`right_bound` 函数返回值是 `1`，因为元素 `3` 是小于 `4` 的最大元素。

统一一下，现在可以改成左右都闭的写法了：
而且由于此时终止条件变成了 `left+1 == right` 了，那么刚好 `right = left-1`，可以直接换成 `right` 返回。

```cpp
int right_bound(vector<int> &nums, int target){
    int left = 0, right = nums.size()-1;    // 采用 [left, right]
    while(left <= right){
        int mid = left + (right-left)/2;
        if(nums[mid] == target)
            left = mid+1;
        else if(nums[mid] < target)
            left = mid+1;
        else if(nums[mid] > target)
            right = mid-1;
    }
    // 第 1 种返回
    // return right;

    // 第 2 种返回
    if(right<0 || right>=nums.size())  return -1;
    return nums[right]==target? right : -1;
}
```

现在，你可以去做 在排序数组中查找元素的第一个和最后一个位置

在左右边界的代码中，所有的情况，都是需要变动 `mid` 的，如 `mid+1` 或 `mid-1`，记住！

```cpp
class Solution {
public:
    // 二分查找框架
    int left_bound(vector<int> &nums, int target){
        int left = 0, right = nums.size()-1;    // 区间[left, right]
        while(left <= right){
            int mid = left + (right-left)/2;
            if(nums[mid] == target)
```

```
                right = mid-1;
            else if(nums[mid] < target)
                left = mid+1;
            else if(nums[mid] > target)
                right = mid-1;
        }
        // return left;
        if(left<0 || left>=nums.size())  return -1;
        return nums[left]==target? left : -1;
    }

    int right_bound(vector<int> &nums, int target){
        int left = 0, right = nums.size()-1;    // 区间[left, right]
        while(left <= right){
            int mid = left + (right-left)/2;
            if(nums[mid] == target)
                left = mid+1;
            else if(nums[mid] < target)
                left = mid+1;
            else if(nums[mid] > target)
                right = mid-1;
        }
        // return right;
        if(right<0 || right>=nums.size())  return -1;
        return nums[right]==target? right : -1;
    }

    vector<int> searchRange(vector<int>& nums, int target) {
        // 左边界
        int left = left_bound(nums, target);
        int right = right_bound(nums, target);
        return vector<int>{left, right};
    }
};
```

二分思维的精髓就是：**通过已知信息尽可能多地收缩（折半）搜索空间**，从而增加穷举效率，快速找到目标。

但实际题目中不会直接让你写二分代码，我会在 【二分查找的运用】 和 【二分查找的更多习题】 中进一步讲解如何把二分思维运用到更多算法题中

(五) 递归（一个视角+两种思维）

> **一个视角是指「树」的视角，两种思维模式是指「遍历」和「分解问题」两种思维模式。**
>
> 1. 算法的本质是穷举，递归是一种重要的穷举手段，递归的正确理解方法是从「**树**」的角度理解。
> 2. 编写递归算法，有两种思维模式：一种是通过「**遍历**」一遍树得到答案，另一种是通过「**分解问题**」得到答案。

1、从树的角度理解递归

- **斐波那契数列**
- 参考：509.斐波那契数

```
// 斐波那契数列
int fib(int n) {
    if (n < 2) {
        return n;
    }
    return fib(n - 1)
         + fib(n - 2);
}

// 二叉树遍历函数
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    traverse(root.left);
    traverse(root.right);
}
```

- **全排列问题**
- 参考:46.全排列

```
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;   // 记录一条完整答案
    vector<bool> visit; // 记录当前访问情况

public:
    // 从树的角度理解递归
    vector<vector<int>> permute(vector<int>& nums) {
        int n = nums.size();
        visit.assign(n, false);
        // 只需要访问根节点即可
        backtrack(nums);
        return ans;
    }

    // 多叉树遍历子节点
    void backtrack(vector<int>& nums){
        // 已经访问到叶节点了
        if(path.size() == nums.size()){
            ans.push_back(path);
            return;
        }

        // 遍历子节点
        for(int i=0; i<nums.size(); i++){
```

```
            // 未访问过的，都是子节点
            if(visit[i])  continue;
            path.push_back(nums[i]);
            visit[i] = true;

            // 访问这个子节点的所在的树
            backtrack(nums);

            // 退回
            path.pop_back();
            visit[i] = false;
        }
    }
};
```

可以抽象看成：

```
// 全排列算法主要结构
void backtrack(int[] nums, List<Integer> track) {
    if (track.size() == nums.length) {
        return;
    }
    for (int i = 0; i < nums.length; i++) {
        backtrack(nums, track);
    }
}

// 多叉树遍历函数
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    for (TreeNode child : root.children) {
        traverse(child);
    }
}
```

你应该已经感觉到了，「树」结构是一个非常有效的数据结构。**把问题抽象成树结构，然后用代码去遍历这棵树**，就是**递归的本质**。

2、编写递归的两种思维模式

> 上面讲的两道例题中，它们虽然都抽象成了一棵**递归树**，但**斐波那契数列**使用的是「**分解问题**」的思维模式求解，**全排列**使用的是「**遍历**」的思维模式求解。

- **分解问题的思维模式**
  如果你想用「分解问题」的思维模式来写递归算法，那么这个**递归函数**一定要有一个**清晰的定义**，说明这个函数参数的含义是什么，返回什么结果。
- 二叉树的最大深度
  参考：104.二叉树的最大深度

```cpp
class Solution {
public:
    // 分解问题的递归
    // 递归函数有清晰定义：f(父) = max{ f(子) } + 1
    int maxDepth(TreeNode* root) {
        // 已经抵达叶节点的子节点（空）
        if(root == NULL)  return 0;

        // 递归：分解问题
        return max(maxDepth(root->left), maxDepth(root->right)) + 1;
    }
};
```

- **遍历的思维模式**
  递归树上的节点并没有一个明确的含义，只是记录了之前所做的一些选择。所有全排列，就是所有叶子节点上的结果。这种思维模式称为「遍历」。
  如果你想用「遍历」的思维模式来写递归算法，那么你需要一个**无返回值**的遍历函数，在遍历的**过程中收集结果**。

```cpp
// 全排列算法主要结构
vector<vector<int>> ans;
vector<int> path;

// 递归树遍历
void backtrack(vector<int>& nums){
    // 到达叶节点，收集结果
    if(path.size() == nums.size()){
        ans.push_back(path);
        return;
    }

    for(int i=0; i<nums.size(); i++){
        // 做出选择
        path.push_back(nums[i]);

        backtrack(nums);

        // 撤销选择
        path.pop_back();
    }
}
```

再看前面的二叉树的最大深度，实际上遍历整棵树，在遍历的过程更新最大深度，这样当遍历完所有节点时，必然可以求出整棵树的最大深度：

```cpp
class Solution {
private:
    int cur = 0;
    int ans = 0;

```

```
public:
    // 递归：遍历思维也可以
    // 遍历，无返回值，过程中收集结果
    void traverse(TreeNode* root){
        // 到达叶节点，收集答案
        if(root == NULL){
            ans = max(ans, cur);
            return;
        }

        cur++;
        traverse(root->left);
        traverse(root->right);
        cur--;
    }
    // 主函数
    int maxDepth(TreeNode* root) {
        traverse(root);
        return ans;
    }
};
```

- 参考以下步骤，运用自如地写出递归算法：
    1. 首先，这个问题是否可以**抽象成一棵树**结构？如果可以，那么就要用递归算法了。
    2. 如果要用递归算法，那么就思考「遍历」和「分解问题」这两种思维模式，看看哪种更适合这个问题。
    3. 如果用「**分解**问题」的思维模式，那么一定要写清楚这个**递归函数的定义**是什么，然后利用这个定义来分解问题，利用子问题的答案推导原问题的答案。
    4. 如果用「**遍历**」的思维模式，那么要用一个**无返回值**的递归函数，单纯起到遍历递归树，到达**叶节点收集结果**的作用。

> 「**分解问题**」的思维模式就对应着后面要讲解的 动态规划算法 和 分治算法。
>
> 「**遍历**」的思维模式就对应着后面要讲解的 DFS/回溯算法。
>
> 在 二叉树习题章节，把所有二叉树相关的题目都用这两种思维模式来解一遍。你只要把二叉树玩明白了，这些递归算法就都玩明白了，真的很简单。

(六) 动态规划

> - 动态规划问题的一般形式就是**求最值**。动态规划其实是运筹学的一种最优化方法，只不过在计算机问题上应用比较多，比如说让你求最长递增子序列呀，最小编辑距离呀等等。
> - 求解动态规划的核心问题是**穷举**。因为要求最值，肯定要把所有可行的答案穷举出来，然后在其中找最值呗。
> - 只有列出正确的「**状态转移方程**」，才能正确地穷举。
>
> 1. 是否具备「**最优子结构**」：是否能够通过子问题的最值得到原问题的最值
> 2. 存在「**重叠子问题**」：需要你使用「备忘录」或者「DP table」来优化穷举过程，避免不必要的计算
>
> 动态规划三要素：状态转移方程、最优子结构、重叠子问题

```python
# 自顶向下递归的动态规划
def dp(状态1, 状态2, ...):
    for 选择 in 所有可能的选择:
        # 此时的状态已经因为做了选择而改变
        result = 求最值(result, dp(状态1, 状态2, ...))
    return result

# 自底向上迭代的动态规划
# 初始化 base case
dp[0][0][...] = base case
# 进行状态转移
for 状态1 in 状态1的所有取值:
    for 状态2 in 状态2的所有取值:
        for ...
            dp[状态1][状态2][...] = 求最值(选择1, 选择2...)
```

1、斐波那契数列

参考：509.斐波那契数

> 斐波那契数列没有求最值，所以严格来说不是动态规划问题。主要是为了体现重叠子问题。

- 但凡遇到需要递归的问题，最好都画出递归树，这对你分析算法的复杂度，寻找算法低效的原因都有巨大帮助。

- 二叉树节点总数为指数级别，所以子问题个数为 $O(2^n)$，计算解决一个子问题的时间为$O(1)$，总复杂度$O(2^n)$。
  注：满二叉树的节点个数是$N = 2^{h-1} - 1$

带备忘录的递归解法

- 即然耗时的原因是重复计算，那么我们可以造一个「备忘录」，每次算出某个子问题的答案后别急着返回，先记到「备忘录」里再返回；每次遇到一个子问题先去「备忘录」里查一查，如果发现之前已经解决过这个问题了，直接把答案拿出来用，不要再耗时去计算了。
- 一般使用一个数组充当这个「备忘录」，当然你也可以使用哈希表（字典），思想都是一样的。

```cpp
class Solution {
private:
    vector<int> memo;
public:
    // 递归：备忘录
    int fib(int n) {
        memo.assign(n+1, 0);
        return dp(n);
    }

    int dp(int n){
        if(n==0 || n==1)  return n;

        if(memo[n]!=0)  return memo[n];
```

```
15
16          memo[n] = dp(n-1) + dp(n-2);
17          return memo[n];
18      }
19  };
```

- 子问题个数为 $O(n)$，时间为 $O(1)$，总复杂度是 $O(n)$。

- 带**备忘录的递归**解法的效率已经和迭代的动态规划解法一样了。实际上，这种解法和常见的动态规划解法已经差不多了，只不过这种解法是「**自顶向下**」进行「**递归**」求解
- 我们更常见的**动态规划**代码是「**自底向上**」进行「**递推**」求解。

带dp表的动态规划

```
1   class Solution {
2   private:
3       vector<int> dp;
4   public:
5       // 带dp的动态规划
6       int fib(int n) {
7           if(n == 0)  return 0;
8           dp.assign(n+1, 0);
9           // base case
10          dp[0] = 0, dp[1] = 1;
11          // 状态转移
12          for(int i=2; i<=n; i++)
13              dp[i] = dp[i-1] + dp[i-2];
14
15          return dp[n];
16      }
17  };
```

- 千万不要看不起暴力解，动态规划问题最困难的就是写出这个暴力解，即**状态转移方程**。
- 只要写出暴力解，优化方法无非是用`备忘录`或者`DP table`，再无奥妙可言。

2、凑零钱

参考：322.零钱兑换

- 给你`k`种面值的硬币，面值分别为`c1, c2 ... ck`，每种硬币的数量无限，再给一个总金额`amount`，问你最少需要几枚硬币凑出这个金额，如果不可能凑出，算法返回`-1`。算法的函数签名如下：
  `int coinChange(vector<int>& coins, int amount);`
- 这个问题是动态规划问题，因为它具有「**最优子结构**」的。要符合「最优子结构」，**子问题间必须互相独立**。

> 什么是子问题相互独立?
> 比如问题是考出最高的总成绩，那么将每门科目考到最高，这些子问题是互相独立，互不干扰的，那就是符号最优子结构。
> 如果加一条限制，数学越好语文就会越差，那么就不是相互独立的了，也就不符合最优子结构。

1. 确定「**状态**」，也就是原问题和子问题中会变化的变量。
2. 确定「**选择**」，也就是导致「状态」产生变化的行为。
3. 明确 **dp** 函数/数组的定义。

先看直接单纯写出递归：

```cpp
class Solution {
public:
    // 暴力递归
    int coinChange(vector<int>& coins, int amount) {
        return dp(coins, amount);
    }
    // dp(n) 是凑成n所需最少数量
    int dp(vector<int>& coins, int amount){
        // base case
        if(amount == 0)  return 0;
        if(amount < 0 )  return -1;

        int ans = -1;
        for(int coin: coins){
            // 计算子问题的解
            int subProblem = dp(coins, amount-coin);
            // 无解跳过
            if(subProblem == -1)  continue;
            // 有解更新
            if(ans == -1)  ans = subProblem+1;
            else ans = min(ans, subProblem+1);
        }

        return ans;
    }
};
```

这实际上就是暴力计算：

$$dp(n) = \begin{cases} -1, & n < 0 \\ 0, & n = 0 \\ min\{dp(n - coin) + 1 \quad | \quad coin \in coins\} & n > 0 \end{cases} \tag{1}$$

画出递归树：

- 递归算法的时间复杂度分析：**子问题总数 x 解决每个子问题所需的时间**。
- 假设目标金额为 `n`，给定的硬币个数为 `k`，那么递归树最坏情况下高度为 `n`（全用面额为 1 的硬币），然后再假设这是一棵满 `k` 叉树，则节点的总数在 $O(k^n)$ 这个数量级。每个子问题的复杂度，由于每次递归包含一个 `for` 循环，复杂度为 $O(k)$，相乘得到总时间复杂度为 $O(k^n)$，指数级别。
- 这个问题其实就解决了，只不过需要消除一下重叠子问题。

带备忘录的递归
就只需要在每次进入子问题前，看是不是计算过即可。

1. 计算前，看备忘录是不是计算过。
2. 离开前，在备忘录记下计算结果。

```cpp
class Solution {
private:
    vector<int> memo;

public:
    // 递归：改进备忘录
    int coinChange(vector<int>& coins, int amount) {
        memo.assign(amount+1, -666);    // 先赋值为特殊记号，表示未计算过
        return dp(coins, amount);
    }
    // dp(n) 是凑成n所需最少数量
    int dp(vector<int>& coins, int amount){
        // base case
        if(amount == 0)  return 0;
        if(amount < 0 )  return -1;

        // 计算前，先看当前问题是不是计算过了
        if(memo[amount] != -666)
            return memo[amount];

        int ans = -1;
        for(int coin: coins){
            // 计算子问题的解
            int subProblem = dp(coins, amount-coin);
            // 无解跳过
            if(subProblem == -1)  continue;
            // 有解更新
            if(ans == -1)  ans = subProblem+1;
            else ans = min(ans, subProblem+1);
        }

        // 离开之前，记下备忘录
        memo[amount] = ans;
        return ans;
    }
};
```

- 子问题总数不会超过金额数 `n`，即子问题数目为 $O(n)$。处理一个子问题的时间不变，仍是 $O(k)$，所以总的时间复杂度是 $O(kn)$。

带dp的动态规划

```cpp
class Solution {
private:
    vector<int> dp;

public:
    // 动态规划：dp表
```

```
    int coinChange(vector<int>& coins, int amount) {
        dp.assign(amount+1, amount+1);  // 直接初始化为比金额还大 1，表示组不成

        // base case
        dp[0] = 0;

        // 递推
        for(int i=1; i<=amount; i++){
            for(int coin: coins){
                if(i-coin < 0)  continue;
                dp[i] = min(dp[i], dp[i-coin]+1);
            }
        }

        return dp[amount]>amount? -1 : dp[amount];
    }
};
```

- 显然时间复杂度$O(kn)$

> 为什么凑零钱问题**不能用贪心**算法来解？因为想要硬币数最少，那就总是先使用面额大的硬币来凑？
>
> - 这题不能用贪心，贪心的意思是说，一路按照最优选择选下去，就一定能得到正确答案（专业术语叫做贪心选择性质）。而这道题是不满足贪心选择性质的。你如果无脑选用大额硬币，不一定就能凑出目标金额，即便凑出了，也不一定是最小的数量。
> - 比如要凑8块，有1，4，5面值的钱，贪心就是【5，1，1，1】，正确答案是【4，4】
> - 那么这时候你就要尝试其他较小的金额了，所以说到底还是得暴力穷举所有情况，需要用递归进行暴力穷举。通过观察暴力穷举解法代码，我们发现这道题存在最优子结构和重叠子问题，所以逐步优化写出了带备忘录的递归穷举解法（动态规划）。
> - 注意上面这个思考过程，从**暴力穷举**算法开始，逐步尝试各种**优化**方法。至于「贪心」「动态规划」这种名词，只不过是对不同优化过程的代称罢了。

计算机解决问题其实没有任何特殊的技巧，它唯一的解决办法就是穷举，穷举所有可能性。算法设计无非就是先思考「如何穷举」，然后再追求「如何聪明地穷举」。

1. 列出**状态转移方程**，就是在解决「**如何穷举**」的问题。
   之所以说它难，一是因为很多穷举需要递归实现，二是因为有的问题本身的解空间复杂，不那么容易穷举完整。
2. **备忘录**、`DP table` 就是在追求「**如何聪明地穷举**」。
   用空间换时间的思路，是降低时间复杂度的不二法门。

(七) 回溯(DFS)

> 其实回溯算法和我们常说的 DFS 算法基本可以认为是同一种算法，它们的细微差异在之后章节说明。
>
> - 抽象地说，解决一个**回溯**问题，实际上就是**遍历一棵决策树**的过程，树的**每个叶子节点存放着一个合法答案**。你把整棵树遍历一遍，把叶子节点上的答案都收集起来，就能得到所有的合法答案。
> - 站在回溯树的一个节点上，你只需要思考 3 个问题：

> 1. **路径**：也就是已经做出的选择。
> 2. **选择列表**：也就是你当前可以做的选择。
> 3. **结束条件**：也就是到达决策树底层，无法再做选择的条件。

```
result = []
def backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return

    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

其核心就是 for 循环里面的递归，在递归调用之前「做选择」，在递归调用之后「撤销选择」。

1、全排列

参考：46.全排列

> 我们这次讨论的全排列问题不包含重复的数字，包含重复数字的扩展场景在之后章节。
>
> 我们定义的 backtrack 函数其实就像一个指针，在这棵树上游走，同时要正确维护每个节点的属性，每当走到树的底层叶子节点，其「路径」就是一个全排列。

框架：

```
void traverse(TreeNode* root) {
    for (TreeNode* child : root->childern) {
        // 前序位置需要的操作
        traverse(child);
        // 后序位置需要的操作
    }
}
```

> 疑问：多叉树 DFS 遍历框架的前序位置和后序位置应该在 for 循环外面，并不应该是在 for 循环里面呀？为什么在回溯算法中跑到 for 循环里面了？
> 是的，DFS 算法的前序和后序位置应该在 for 循环外面，不过回溯算法和 DFS 算法略有不同，解答回溯/DFS 算法的若干疑问 会具体讲解，这里可以暂且忽略这个问题。

前序和后序只是两个很有用的时间点：

前序遍历的代码在进入某一个节点之前的那个时间点执行，后序遍历代码在离开某个节点之后的那个时间点执行。
再看前面的：

```
for 选择 in 选择列表:
    # 做选择
    将该选择从选择列表移除
    路径.add(选择)
    backtrack(路径, 选择列表)
    # 撤销选择
    路径.remove(选择)
    将该选择再加入选择列表
```

好的，直接看一下完整代码：

```cpp
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;
    vector<bool> visit;

public:
    // 回溯法
    vector<vector<int>> permute(vector<int>& nums) {
        visit.assign(nums.size(), false);
        backtrace(nums);
        return ans;
    }

    void backtrace(vector<int>& nums){
        // 抵达叶节点，收集结果
        if(path.size() == nums.size()){
            ans.push_back(path);
            return;
        }

        // 进行一步决策选择
        for(int i=0; i<nums.size(); i++){
            if(visit[i])  continue;
            // 做选择
            path.push_back(nums[i]);
            visit[i] = true;
            // 进入下一层决策树
            backtrace(nums);
            // 撤销选择
            path.pop_back();
            visit[i] = false;
        }
    }
};
```

稍微做了些变通，没有显式记录「选择列表」，而是通过 used 数组排除已经存在 track 中的元素，从而推导出当前的选择列表：

- 但是必须说明的是，不管怎么优化，都符合回溯框架，而且时间复杂度都不可能低于 $O(N!)$，因为穷举整棵决策树是无法避免的，你最后肯定要穷举出 `N!` 种全排列结果。
  这也是回溯算法的一个特点，不像动态规划存在重叠子问题可以优化，**回溯算法就是纯暴力穷举，复杂度一般都很高**。
  动态规划和回溯算法底层都把问题抽象成了树的结构，但这两种算法在思路上是完全不同的。在 二叉树心法（纲领篇） 你将看到动态规划和回溯算法更深层次的区别和联系。

(八) BFS

- **DFS/回溯**算法的本质就是递归遍历一棵穷举树（多叉树），而多叉树的递归遍历又是从二叉树的递归遍历衍生出来的。所以我说 DFS/回溯算法的本质是**二叉树的递归遍历**。
- **BFS** 算法的本质就是遍历一幅**图**。而图的遍历算法其实就是多叉树的遍历算法加了个 visited 数组防止死循环；多叉树的遍历算法又是从二叉树遍历算法衍生出来的。所以我说 BFS 算法的本质就是二叉树的**层序遍历**。

1、算法框架

- 在前文图的数据结构里面，已经讲解了有三种写法，这里使用第二种。

```cpp
// 从 s 开始 BFS 遍历图的所有节点，且记录遍历的步数
// 当走到目标节点 target 时，返回步数
int bfs(const Graph& graph, int s, int target) {
    vector<bool> visited(graph.size(), false);
    queue<int> q;
    q.push(s);
    visited[s] = true;
    // 记录从 s 开始走到当前节点的步数
    int step = 0;
    while (!q.empty()) {
        int sz = q.size();
        for (int i = 0; i < sz; i++) {
            int cur = q.front();
            q.pop();
            cout << "visit " << cur << " at step " << step << endl;
            // 判断是否到达终点
            if (cur == target) {
                return step;
            }
            // 将邻居节点加入队列，向四周扩散搜索
            for (int to : neighborsOf(cur)) {
                if (!visited[to]) {
                    q.push(to);
                    visited[to] = true;
                }
            }
        }
        step++;
    }
```

```
    // 如果走到这里，说明在图中没有找到目标节点
    return -1;
}
```

2、滑动谜题

参考：773.滑动谜题

- 给你一个 `2x3` 的滑动拼图，用一个 `2x3` 的数组 `board` 表示。拼图中有数字 `0~5` 六个数，其中数字 `0` 就表示那个空着的格子，你可以移动其中的数字，当 `board` 变为 `[[1,2,3],[4,5,0]]` 时，赢得游戏。
  请你写一个算法，计算赢得游戏需要的最少移动次数，如果不能赢得游戏，返回 `-1`。
  比如说输入的二维数组 `board = [[4,1,2],[5,0,3]]`，算法应该返回 `5`：

  如果输入的是 `board = [[1,2,3],[5,4,0]]`，则算法返回 `-1`，因为这种局面下无论如何都不能赢得游戏。

> - 抽象出来的图结构也是会包含环的，所以需要一个 `visited` 数组记录已经走过的节点，避免成环导致死循环。
> - 注意，此时是整张图的状态完全和之前一样了才是成环了，所以`visited`的索引，应该是一整张图的状态。但二维数组这种可变数据结构是无法直接加入哈希集合的。
> - 再用一点技巧，想办法把二维数组转化成一个不可变类型才能存到哈希集合中。常见的解决方案是把二维数组序列**化成一个字符串**，这样就可以直接存入哈希集合了。

其中比较有技巧性的点在于，二维数组有「**上下左右**」的概念，压缩成一维的字符串后后，还怎么把数字 0 和上下左右的数字进行交换？对于这道题，题目说输入的数组大小都是 2 x 3，所以我们可以直接手动写出来这个映射：

```
// 记录一维字符串的相邻索引
int map[6][3] = {
    {1, 3},
    {0, 2, 4},
    {1, 5},
    {0, 4},
    {1, 3, 5},
    {2, 4}
}
```

这个映射的含义就是，**在一维字符串中，索引 `i` 在二维数组中的的相邻索引为 `neighbor[i]`**：

这样，无论数字 0 在哪里，都可以通过这个索引映射得到它的相邻索引进行交换了。

> 如果是 `m x n` 的二维数组，怎么办？
> 用代码生成它的一维索引映射。
> 观察上图就能发现，如果二维数组中的某个元素 `e` 在一维数组中的索引为 `i`，那么 `e` 的左右相邻元素在一维数组中的索引就是 `i - 1` 和 `i + 1`，而 `e` 的上下相邻元素在一维数组中的索引就是 `i - n` 和 `i + n`，其中 `n` 为二维数组的列数。

```
// 生成邻居表
vector<vector<int>> generateNeighborTable(int m, int n){
    vector<vector<int>> map(m*n);
```

```
    for(int i=0; i<m*n; i++){
        vector<int> one;
        // 不是第一列，有左侧邻居
        if(i%n != 0)  one.push_back(i-1);
        // 不是最后列，有右侧邻居
        if(i%n != n-1)  one.puhs_back(i+1);
        // 不是第一行，有上侧邻居
        if(i-n >= 0)  one.push_back(i-n);
        // 不是最后行，有下侧邻居
        if(i+n < m*n)  one.push_back(i+n);
        map[i] = one;
    }
    return map;
}
```

好的，至此，解法已经出来：

```
class Solution {
private:
    string target = "123450";        // 目标状态图
    unordered_set<string> visit;     // 记录已出现过的状态，防止成环
    // 索引表，map[2] 表示 s[2] 的上下左右邻居在 s 中的索引
    vector<vector<int>> map = {
        {1, 3},
        {0, 2, 4},
        {1, 5},
        {0, 4},
        {1, 3, 5},
        {2, 4}
    };

public:
    // BFS 框架
    int slidingPuzzle(vector<vector<int>>& board) {
        //Tips: 转为 string 操作（方便 1 判断相同、2 哈希存储）
        string start = "000000";
        int m = board.size(), n = board[0].size();
        for(int i=0; i<m; i++)
            for(int j=0; j<n; j++)
                start[i*n+j] = board[i][j] + '0';

        /* --- BFS 框架 --- */
        queue<string> Q;
        Q.push(start);
        visit.insert(start);
        int step = 0;
        while(!Q.empty()){
            // 本层所有的状态图
            int sz = Q.size();
            for(int i=0; i<sz; i++){
                string cur = Q.front();  Q.pop();
```

```
                // 已经抵达
                if(cur == target)  return step;
                // 下一层状态图入队
                for(string next: getNeighbor(cur)){
                    // 防止成环
                    if(visit.count(next))  continue;
                    // 入队
                    Q.push(next);
                    visit.insert(next);
                }
            }
            // 进入下层，步数加一
            step++;
        }
        /* --- BFS 框架 --- */

        // 失败
        return -1;
    }

    // 返回cur的下一步状态图
    vector<string> getNeighbor(string cur){
        vector<string> all;
        int zero = cur.find('0');
        for(int idx: map[zero]){
            string next = cur;
            next[zero] = cur[idx];
            next[idx] = cur[zero];
            all.push_back(next);
        }
        return all;
    }
};
```

你会发现，这里对于 `visit` 只有 `insert` 并没有 `erase`，因为并不会回滚，我们是 BFS，广度优先遍历不存在回滚状态。

3、解开密码锁的最少次数

参考：752.打开转盘锁

- 你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有10个数字： `'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'` 。每个拨轮可以自由旋转：例如把 `'9'` 变为 `'0'`， `'0'` 变为 `'9'` 。每次旋转都只能旋转一个拨轮的一位数字。
  锁的初始数字为 `'0000'` ，一个代表四个拨轮的数字的字符串。
  列表 `deadends` 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。
  字符串 `target` 代表可以解锁的数字，你需要给出解锁需要的最小旋转次数，如果无论如何不能解锁，返回 `-1` 。

- 示例1：
  输入：`deadends = ["8888"], target = "0009"`
  输出：`1`
  解释：把最后一位反向旋转一次即可 `"0000" -> "0009"`。
- 示例2：
  输入：`deadends = ["0201","0101","0102","1212","2002"], target = "0202"`
  输出：`6`
  解释：可能的移动序列为 `"0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202"`。注意 `"0000" -> "0001" -> "0002" -> "0102" -> "0202"` 这样的序列是不能解锁的，因为当拨动到 "0102" 时这个锁就会被锁定。

千万不要陷入细节，尝试去想各种具体的情况。要知道算法的本质就是**穷举**，我们直接从 "0000" 开始暴力穷举，把所有可能的拨动情况都穷举出来，难道还怕找不到最少的拨动次数么？

```
class Solution {
private:
    unordered_set<string> visit;
    unordered_set<string> dead;

public:
    // BFS 框架
    int openLock(vector<string>& deadends, string target) {
        // 死亡状态也用哈希集合
        for(string state: deadends)  dead.insert(state);
        queue<string> Q;
        Q.push("0000");
        int step = 0;
        while(!Q.empty()){
            int sz = Q.size();
            for(int i=0; i<sz; i++){
                // 读取队头
                string cur = Q.front();  Q.pop();
                // 成功状态
                if(cur == target)  return step;
                // 死亡状态
                if(dead.count(cur))  continue;

                // 下一层入队
                for(string next : getNeighbors(cur)){
                    // 防成环(不会矛盾的，是使用全部四个字符，一起作为状态来哈希的)
                    if(visit.count(next))  continue;
                    Q.push(next);
                    visit.insert(next);
                }
            }

            // 进入下一层，步数+1
            step++;
        }

```

```
        // 最终失败
        return -1;
    }

    // 获取 cur 的旋转一次的可能(8个)
    vector<string> getNeighbors(string cur){
        vector<string> all;
        char map[10][2] = {
            {'9','1'}, {'0','2'}, {'1','3'},{'2','4'}, {'3','5'},
            {'4','6'}, {'5','7'}, {'6','8'},{'7','9'}, {'8','0'}
        };
        for(int i=0; i<4; i++){
            string next = cur;
            next[i] = map[cur[i]-'0'][0];
            all.push_back(next);
            next[i] = map[cur[i]-'0'][1];
            all.push_back(next);
        }
        return all;
    }
};
```

> 注：这里对于死亡状态的处理，也可以在准备入队的时候就直接不入队。各有利弊。

4、双向 BFS 优化

> 下面再介绍一种 BFS 算法的优化思路：双向 BFS，可以提高 BFS 搜索的效率。
> 在一般的面试笔试题中，普通的 BFS 算法已经够用了，如果遇到超时无法通过，或者面试官的追问，可以考虑解法是否需要双向 BFS 优化。
>
> 双向 BFS 就是从标准的 BFS 算法衍生出来的：
> **传统的 BFS 框架是从起点开始向四周扩散，遇到终点时停止；**
> **而双向 BFS 则是从起点和终点同时开始扩散，当两边有交集的时候停止。**

- 图示中的树形结构，如果终点在最底部，按照传统 BFS 算法的策略，会把整棵树的节点都搜索一遍，最后找到 `target`；而双向 BFS 其实只遍历了半棵树就出现了交集，也就是找到了最短距离。
- 当然从 Big O 表示法分析算法复杂度的话，这两种 BFS 在最坏情况下都可能遍历完所有节点，所以**理论时间复杂度都是 $O(N)$**，但**实际运行中双向 BFS 确实会更快**一些。

> 双向 BFS 的局限性：你必须**知道终点**在哪里，才能使用双向 BFS 进行优化。
> 上述两题可以直接知道终点。
> 但是比如求树的最小高度，就不知道终点。

以密码锁为例，这里直接给代码，不再讲解，实际上考试和竞赛，就用上述的代码就行。这里的优化，只需要知道有这么一个思路，在面试时可以讲出来就行，不追求代码。

```
class Solution {
public:
    int openLock(vector<string>& deadends, string target) {
```

```
        unordered_set<string> deads(deadends.begin(), deadends.end());

        // base case
        if (deads.count("0000")) return -1;
        if (target == "0000") return 0;

        // 用集合不用队列，可以快速判断元素是否存在
        unordered_set<string> q1;
        unordered_set<string> q2;
        unordered_set<string> visited;

        int step = 0;
        q1.insert("0000");
        visited.insert("0000");
        q2.insert(target);
        visited.insert(target);

        while (!q1.empty() && !q2.empty()) {

            // 在这里增加步数
            step++;

            // 哈希集合在遍历的过程中不能修改，所以用 newQ1 存储邻居节点
            unordered_set<string> newQ1;

            // 获取 q1 中的所有节点的邻居
            for (const string& cur : q1) {
                // 将一个节点的未遍历相邻节点加入集合
                for (const string& neighbor : getNeighbors(cur)) {
                    // 判断是否到达终点
                    if (q2.count(neighbor)) {
                        return step;
                    }
                    if (!visited.count(neighbor) && !deads.count(neighbor)) {
                        newQ1.insert(neighbor);
                        visited.insert(neighbor);
                    }
                }
            }
            // newQ1 存储着 q1 的邻居节点
            q1 = newQ1;
            // 因为每次 BFS 都是扩散 q1，所以把元素数量少的集合作为 q1
            if (q1.size() > q2.size()) {
                swap(q1, q2);
            }
        }
        return -1;
    }

    // 将 s[j] 向上拨动一次
    string plusOne(string s, int j) {
        if (s[j] == '9')
```

```
            s[j] = '0';
        else
            s[j] += 1;
        return s;
    }

    // 将 s[i] 向下拨动一次
    string minusOne(string s, int j) {
        if (s[j] == '0')
            s[j] = '9';
        else
            s[j] -= 1;
        return s;
    }

    vector<string> getNeighbors(string s) {
        vector<string> neighbors;
        for (int i = 0; i < 4; i++) {
            neighbors.push_back(plusOne(s, i));
            neighbors.push_back(minusOne(s, i));
        }
        return neighbors;
    }
};
```

双向 BFS 还是遵循 BFS 算法框架的，但是有几个细节区别：

1. 不再使用队列存储元素，而是改用 哈希集合，方便**快速判两个集合是否有交集**。
2. 调整了 `return step` 的位置。因为双向 BFS 中不再是简单地判断是否到达终点，而是判断两个集合是否有交集，所以要在计算出邻居节点时就进行判断。
3. 还有一个优化点，每次都保持 `q1` 是元素数量较小的集合，这样可以一定程度减少搜索次数。
   因为按照 BFS 的逻辑，队列（集合）中的元素越多，扩散邻居节点之后新的队列（集合）中的元素就越多；在双向 BFS 算法中，如果我们每次都选择一个较小的集合进行扩散，那么占用的空间增长速度就会慢一些，效率就会高一些。

不过话说回来，**无论传统 BFS 还是双向 BFS，无论做不做优化，从 Big O 衡量标准来看，时间复杂度都是一样的**，只能说双向 BFS 是一种进阶技巧，算法运行的速度会相对快一点，掌握不掌握其实都无所谓。

(九) 二叉树系列

> 二叉树解题的思维模式分两类：
>
> 1. 是否可以通过遍历一遍二叉树得到答案？如果可以，用一个 traverse 函数配合外部变量来实现，这叫「**遍历**」的思维模式。
> 2. 是否可以定义一个递归函数，通过子问题（子树）的答案推导出原问题的答案？如果可以，写出这个递归函数的定义，并充分利用这个函数的返回值，这叫「**分解问题**」的思维模式。
>   无论使用哪种思维模式，你都需要思考：如果单独抽出**一个二叉树节点**，它需要**做什么事情**？需要**在什么时候（前/中/后序位置）做**？其他的节点不用你操心，递归函数会帮你在所有节点上执行相同的操作。

算法	思想	关注点	操作位置	区别
动态规划	分解	整个子树	后序位置	\
回溯	遍历(递归)	树枝(边)	\	做选择、撤销选择在for循环里面
DFS	遍历(递归)	节点	\	做选择、撤销选择在for循环外面
BFS	遍历(迭代)	节点	\	\

1、二叉树的重要性、深入理解前中后序

- 举个例子，比如两个经典排序算法 快速排序 和 归并排序，对于它俩，你有什么理解?
- 如果你告诉我，**快速排序就是个二叉树的前序遍历，归并排序就是个二叉树的后序遍历**，那么我就知道你是个算法高手了。

```
void sort(vector<int>& nums, int lo, int hi) {
    // ****** 前序遍历位置 ******
    // 通过交换元素构建分界点 p
    int p = partition(nums, lo, hi);
    // ************************

    sort(nums, lo, p - 1);
    sort(nums, p + 1, hi);
}

// 定义：排序 nums[lo..hi]
void sort(vector<int>& nums, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    // 排序 nums[lo..mid]
    sort(nums, lo, mid);
    // 排序 nums[mid+1..hi]
    sort(nums, mid + 1, hi);

    // ****** 后序位置 ******
    // 合并 nums[lo..mid] 和 nums[mid+1..hi]
    merge(nums, lo, mid, hi);
    // **********************
}
```

旨在说明，二叉树的算法思想的运用广泛，甚至可以说，只要涉及递归，都可以抽象成二叉树的问题。

1、你理解的二叉树的前中后序遍历是什么，仅仅是三个顺序不同的 List 吗?

2、请分析，后序遍历有什么特殊之处?

3、请分析，为什么多叉树没有中序遍历?

```cpp
void traverse(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    // 前序位置
    traverse(root->left);
    // 中序位置
    traverse(root->right);
    // 后序位置
}
```

先不管所谓前中后序，单看 traverse 函数，你说它在做什么事情？其实它就是一个能够遍历二叉树所有节点的一个函数，和你遍历数组或者链表本质上没有区别：

```cpp
// 迭代遍历数组
void traverse(vector<int>& arr) {
    for (int i = 0; i < arr.size(); i++) {

    }
}

// 递归遍历数组
void traverse(vector<int>& arr, int i) {
    if (i == arr.size()) {
        return;
    }
    // 前序位置
    traverse(arr, i + 1);
    // 后序位置
}

struct ListNode {
    int val;
    ListNode *next;
};

// 迭代遍历单链表
void traverse(ListNode* head) {
    for (ListNode* p = head; p != nullptr; p = p->next) {

    }
}

// 递归遍历单链表
void traverse(ListNode* head) {
    if (head == nullptr) {
        return;
    }
    // 前序位置
    traverse(head->next);
    // 后序位置
```

```
}
```

单链表和数组的遍历可以是迭代的，也可以是递归的，二叉树这种结构无非就是二叉链表，它没办法简单改写成 for 循环的迭代形式，所以我们遍历二叉树一般都使用递归形式。你也注意到了，只要是递归形式的遍历，都可以有前序位置和后序位置，分别在递归之前和递归之后。

所谓**前序位置**，就是**刚进入一个节点（元素）的时候**，**后序位置**就是**即将离开一个节点（元素）的时候**，那么进一步，你把代码写在不同位置，代码执行的时机也不同：

比如说，如果让你**倒序打印**一条单链表上所有节点的值，你怎么搞？实现方式当然有很多，但如果你对递归的理解足够透彻，可以利用后序位置来操作：

```cpp
// 递归遍历单链表，倒序打印链表元素
void traverse(ListNode* head) {
    if (head == nullptr) {
        return;
    }
    traverse(head->next);
    // 后序位置
    cout << head->val << endl;
}
```

本质上是利用**递归的堆栈**帮你实现了倒序遍历的效果。

> 前序位置的代码在刚刚进入一个二叉树节点的时候执行；
> 后序位置的代码在将要离开一个二叉树节点的时候执行；
> 中序位置的代码在一个二叉树节点左子树都遍历完，即将开始遍历右子树的时候执行。
>
> 你可以发现每个节点都有「唯一」属于自己的前中后序位置，所以我说前中后序遍历是遍历二叉树过程中处理每一个**节点**的**三个特殊时间点**。
> 因为二叉树的每个节点只会进行**唯一一次左子树切换右子树**，而多叉树节点可能有很多子节点，会多次切换子树去遍历，所以**多叉树节点没有「唯一」的中序遍历位置**。

> **二叉树的所有问题，就是让你在前中后序位置注入巧妙的代码逻辑，去达到自己的目的，你只需要单独思考每一个节点应该做什么，其他的不用你管，抛给二叉树遍历框架，递归会在所有节点上做相同的操作。**

2、两种解题思路

> 二叉树题目的递归解法可以分两类思路，
> 第一类是**遍历**一遍二叉树得出答案，
> 第二类是通过**分解**问题计算出答案，
> 这两类思路分别对应着 **回溯**算法核心框架 和 **动态规划**核心框架。

二叉树的最大深度这个问题来举例：

- **遍历思路**：所谓最大深度就是根节点到「最远」叶子节点的最长路径上的节点数。遍历一遍二叉树，用一个外部变量记录每个节点所在的深度，取最大值就可以得到最大深度。
- **分解思路**：一棵二叉树的最大深度可以通过子树的最大深度推导出来。

那么我们再回头看看最基本的**二叉树前中后序遍历**，就比如力扣第 144 题「二叉树的前序遍历」，让你计算前序遍历结果。

普通解法，就是**遍历思想**：

```
class Solution {
public:
    // 存放前序遍历结果
    vector<int> res;

    // 返回前序遍历结果
    vector<int> preorderTraversal(TreeNode* root) {
        traverse(root);
        return res;
    }

    // 二叉树遍历函数
    void traverse(TreeNode* root) {
        if (root == nullptr) {
            return;
        }
        // 前序位置
        res.push_back(root->val);
        traverse(root->left);
        traverse(root->right);
    }
};
```

能否改成**分解思想**?
一棵二叉树的前序遍历结果 = 根节点 + 左子树的前序遍历结果 + 右子树的前序遍历结果。

```
class Solution {
public:
    // 定义：输入一棵二叉树的根节点，返回这棵树的前序遍历结果
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> res;
        if (root == nullptr) {
            return res;
        }
        // 前序遍历的结果，root->val 在第一个
        res.push_back(root->val);
        // 利用函数定义，后面接着左子树的前序遍历结果
        vector<int> left = preorderTraversal(root->left);
        res.insert(res.end(), left.begin(), left.end());
        // 利用函数定义，最后接着右子树的前序遍历结果
        vector<int> right = preorderTraversal(root->right);
        res.insert(res.end(), right.begin(), right.end());
        return res;
    }
};
```

综上，遇到一道二叉树的题目时的通用思考过程是：
1、**是否可以通过遍历一遍二叉树得到答案**？如果可以，用一个 traverse 函数配合外部变量来实现。
2、**是否可以定义一个递归函数，通过子问题（子树）的答案推导出原问题的答案**？如果可以，写出这个递归函数的定义，并充分利用这个函数的返回值。
3、**无论使用哪一种思维模式，你都要明白二叉树的每一个节点需要做什么，需要在什么时候（前中后序）做。**

> 小题狂练-二叉树系列

3、后序位置的特殊之处

先简单说下前序和中序：

- 前序位置本身其实没有什么特别的性质，之所以你发现好像很多题都是在前序位置写代码，实际上是因为我们习惯把那些对前中后序位置不敏感的代码写在前序位置罢了。
- 中序位置主要用在 BST（二叉搜索树） 场景中，你完全可以把 BST 的中序遍历认为是遍历有序数组。

> 仔细观察，**前中后序位置的代码，能力依次增强**。
>
> - 前序位置的代码只能从函数参数中获取**父节点传递来的数据**。
> - 中序位置的代码不仅可以获取参数数据，还可以获取到**左子树通过函数返回值**传递回来的数据。
> - 后序位置的代码最强，不仅可以获取参数数据，还可以同时获取到**左右子树通过函数返回值**传递回来的数据。
>
> 所以，某些情况下把代码移到后序位置效率最高；**有些事情，只有后序位置的代码能做**。

举些具体的例子来感受下它们的能力区别。现在给你一棵二叉树，我问你两个简单的问题：
1、如果把根节点看做第 1 层，如何打印出每一个节点所在的层数？
2、如何打印出每个节点的左右子树各有多少节点？
第一题：

```
// 二叉树遍历函数
void traverse(TreeNode* root, int level) {
    if (root == nullptr) {
        return;
    }
    // 前序位置
    printf("Node %d at level %d", root->val, level);
    traverse(root->left, level + 1);
    traverse(root->right, level + 1);
}

// 这样调用
traverse(root, 1);
```

第二题：

```
// 定义：输入一棵二叉树，返回这棵二叉树的节点总数
int count(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }
    int leftCount = count(root->left);
    int rightCount = count(root->right);
    // 后序位置
    printf("节点 %p 的左子树有 %d 个节点，右子树有 %d 个节点",
            root, leftCount, rightCount);

    return leftCount + rightCount + 1;
}
```

一个节点在第几层，你从根节点遍历过来的过程就能顺带记录，用递归函数的参数就能传递下去；而以一个节点为根的整棵子树有多少个节点，你必须遍历完子树之后才能数清楚，然后通过递归函数的返回值拿到答案。
结合这两个简单的问题，你品味一下后序位置的特点，只有后序位置才能通过返回值获取子树的信息。
那么换句话说，**一旦你发现题目和子树有关，那大概率要给函数设置合理的定义和返回值，在后序位置写代码了**。

543.二叉树的直径

- 所谓二叉树的「直径」长度，就是任意两个结点之间的路径长度（经过的连线数量）。最长「直径」并不一定要穿过根结点。
- 解决这题的关键在于，每一条二叉树的「直径」长度，就是**一个节点的左右子树的最大深度之和**。
  求整棵树中的最长「直径」，那直截了当的思路就是遍历整棵树中的每个节点，然后通过每个节点的左右子树的最大深度算出每个节点的「直径」，最后把所有「直径」求个最大值即可。

先看下面的解法，或许你会觉得奇怪：

```
class Solution {
public:
    // 记录最大直径的长度
    int maxDiameter = 0;

    int diameterOfBinaryTree(TreeNode* root) {
        // 对每个节点计算直径，求最大直径
        traverse(root);
        return maxDiameter;
    }

private:
    // 遍历二叉树
    void traverse(TreeNode* root) {
        if (root == nullptr) {
            return;
        }
        // 对每个节点计算直径
        int leftMax = maxDepth(root->left);
        int rightMax = maxDepth(root->right);
        int myDiameter = leftMax + rightMax;
        // 更新全局最大直径
```

```
        maxDiameter = max(maxDiameter, myDiameter);

        traverse(root->left);
        traverse(root->right);
    }

    // 计算二叉树的最大深度
    int maxDepth(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        int leftMax = maxDepth(root->left);
        int rightMax = maxDepth(root->right);
        return 1 + max(leftMax, rightMax);
    }
};
```

这里实际上是使用前序遍历来做题，你会觉得奇怪，是前序位置还没得到子树的深度情况，不得不去调用递归函数获得数据。而后再对子树进行遍历。时间复杂度也高达 $O(n^2)$
然而，后序遍历就不需要这么复杂，因为它这个位置天然的就已经获取了左右子树的数据情况。如下：

```
class Solution {
private:
    int ans = 0;
public:
    int diameterOfBinaryTree(TreeNode* root) {
        maxDeep(root);
        return ans;
    }

    int maxDeep(TreeNode* root){
        if(root == NULL)  return 0;
        int ld = maxDeep(root->left);
        int rd = maxDeep(root->right);

        ans = max(ans, ld+rd);

        return max(ld+1, rd+1);
    }
};
```

> 思考题：请你思考一下，**运用后序遍历的题目使用的是「遍历」的思路还是「分解问题」的思路**?
> 利用后序位置的题目，一般都使用「分解问题」的思路。因为当前节点接收并利用了子树返回的信息，这就意味着你把原问题分解成了当前节点 + 左右子树的子问题。
> 反过来，如果你写出了类似一开始的那种递归套递归的解法，大概率也需要反思是不是可以通过后序遍历优化了。
> 参考练习题：
>
> - 二叉树心法-后序篇
> - 二叉搜索树-后序篇

- 后序位置解题

4、以树的视角看动归/回溯/DFS算法的区别和联系

> 动归/DFS/回溯算法都可以看做二叉树问题的扩展，只是它们的关注点不同：
>
> - **动态规划**算法属于**分解**问题（分治）的思路，它的关注点在整棵「**子树**」。
> - **回溯**算法属于**遍历**的思路，它的关注点在节点间的「**树枝**」。
> - **DFS** 算法属于**遍历**的思路，它的关注点在单个「**节点**」。
>
> 此外，两个遍历的区别在于：回溯算法和 DFS 算法代码中「**做选择**」和「**撤销选择**」的**位置不同**。
> DFS 算法把「做选择」「撤销选择」的逻辑放在 **for 循环外面**；
> 回溯算法把「做选择」「撤销选择」的逻辑放在 **for 循环里面**。

例子1：分解思路（动态规划）
给你一棵二叉树，请你用分解问题的思路写一个 count 函数，计算这棵二叉树共有多少个节点。

```
// 定义：输入一棵二叉树，返回这棵二叉树的节点总数
int count(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }
    // 当前节点关心的是两个子树的节点总数分别是多少
    // 因为用子问题的结果可以推导出原问题的结果
    int leftCount = count(root->left);
    int rightCount = count(root->right);
    // 后序位置，左右子树节点数加上自己就是整棵树的节点数
    return leftCount + rightCount + 1;
}
```

这就是动态规划分解问题的思路，它的**着眼点永远是结构相同的整个子问题**，类比到二叉树上就是「**子树**」。

- 参考例题就是斐波那契数列：

```
int fib(int N) {
    if (N == 1 || N == 2) return 1;
    return fib(N - 1) + fib(N - 2);
}
```

例子2：遍历思路（回溯）
给你一棵二叉树，请你用遍历的思路写一个 traverse 函数，打印出遍历这棵二叉树的过程。

```
void traverse(TreeNode* root) {
    // 如果节点为空（说明已经越过叶节点），就返回
    if (root == nullptr) return;
    // 从节点 root 出发，沿着 left 边走
    printf("从节点 %s 进入节点 %s", root, root->left);
    traverse(root->left);
```

```
    // 从节点 root 的左边回来
    printf("从节点 %s 回到节点 %s", root->left, root);

    // 从节点 root 出发，沿着 right 边走
    printf("从节点 %s 进入节点 %s", root, root->right);
    traverse(root->right);
    // 从节点 root 的右边回来
    printf("从节点 %s 回到节点 %s", root->right, root);
}

// 多叉树也一样
// 多叉树节点
class Node {
public:
    int val;
    std::vector<Node*> children;
};

void traverse(Node* root) {
    if (root == NULL) return;
    for (Node* child : root->children) {
        std::cout << "从节点 " << root->val << " 进入节点 " << child->val << std::endl;
        traverse(child);
        std::cout << "从节点 " << child->val << " 回到节点 " << root->val << std::endl;
    }
}
```

你看，这就是回溯算法遍历的思路，它的**着眼点永远是在节点之间移动的过程**，类比到二叉树上就是「**树枝**」。

- 参考例题，就是全排列：

```
// 回溯算法核心部分代码
void backtrack(int[] nums) {
    // 回溯算法框架
    for (int i = 0; i < nums.length; i++) {
        // 做选择
        used[i] = true;
        track.addLast(nums[i]);

        // 进入下一层回溯树
        backtrack(nums);

        // 取消选择
        track.removeLast();
        used[i] = false;
    }
}
```

例子3：遍历思路（DFS）

给你一棵二叉树，请你写一个 traverse 函数，把这棵二叉树上的每个节点的值都加一。

```
void traverse(TreeNode* root) {
    if (root == nullptr) return;
    // 遍历过的每个节点的值加一
    root->val++;
    traverse(root->left);
    traverse(root->right);
}
```

你看，这就是 DFS 算法遍历的思路，它的**着眼点永远是在单一的节点上**，类比到二叉树上就是处理每个「**节点**」。

- 参考题目，岛屿问题

```
// DFS 算法核心逻辑
void dfs(int[][] grid, int i, int j) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n) {
        return;
    }
    if (grid[i][j] == 0) {
        return;
    }
    // 遍历过的每个格子标记为 0
    grid[i][j] = 0;
    dfs(grid, i + 1, j);
    dfs(grid, i, j + 1);
    dfs(grid, i - 1, j);
    dfs(grid, i, j - 1);
}
```

综上，**动态规划关注整棵「子树」，回溯算法关注节点间的「树枝」，DFS 算法关注单个「节点」**。
有了这些铺垫，你就很容易理解为什么回溯算法和 DFS 算法代码中「做选择」和「撤销选择」的位置不同了。

```
// DFS 算法把「做选择」「撤销选择」的逻辑放在 for 循环外面
void dfs(Node* root) {
    if (!root) return;
    // 做选择
    printf("enter node %s\n", root->val.c_str());
    for (Node* child : root->children) {
        dfs(child);
    }
    // 撤销选择
    printf("leave node %s\n", root->val.c_str());
}

// 回溯算法把「做选择」「撤销选择」的逻辑放在 for 循环里面
void backtrack(Node* root) {
    if (!root) return;
    for (Node* child : root->children) {
        // 做选择
```

```
        printf("I'm on the branch from %s to %s\n", root->val.c_str(), child->val.c_str());
        backtrack(child);
        // 撤销选择
        printf("I'll leave the branch from %s to %s\n", child->val.c_str(), root->val.c_str());
    }
}
```

看到了吧，你**回溯算法必须把「做选择」和「撤销选择」的逻辑放在 for 循环里面，否则怎么拿到「树枝」的两个端点**?

5、层序遍历

- 二叉树题型主要是用来培养递归思维的，而层序遍历属于迭代遍历。

```
// 输入一棵二叉树的根节点，层序遍历这棵二叉树
void levelTraverse(TreeNode* root) {
    if (root == nullptr) return;
    queue<TreeNode*> q;
    q.push(root);

    // 从上到下遍历二叉树的每一层
    while (!q.empty()) {
        int sz = q.size();
        // 从左到右遍历每一层的每个节点
        for (int i = 0; i < sz; i++) {
            TreeNode* cur = q.front();
            q.pop();
            // 将下一层节点放入队列
            if (cur->left != nullptr) {
                q.push(cur->left);
            }
            if (cur->right != nullptr) {
                q.push(cur->right);
            }
        }
    }
}
```

这里面 while 循环和 for 循环分管从上到下和从左到右的遍历：

BFS 算法框架 就是从二叉树的**层序遍历**扩展出来的，常用于求无权图的最短路径问题。

拓展：玩的足够花的话，会有各种方法实现层序遍历：
第一个花活：

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

```
    vector<vector<int>> levelTraverse(TreeNode* root) {
        if (root == nullptr) {
            return res;
        }
        // root 视为第 0 层
        traverse(root, 0);
        return res;
    }

    void traverse(TreeNode* root, int depth) {
        if (root == nullptr) {
            return;
        }
        // 前序位置，看看是否已经存储 depth 层的节点了
        if (res.size() <= depth) {
            // 第一次进入 depth 层
            res.push_back(vector<int> {});
        }
        // 前序位置，在 depth 层添加 root 节点的值
        res[depth].push_back(root->val);
        traverse(root->left, depth + 1);
        traverse(root->right, depth + 1);
    }
};
```

这种思路从结果上说确实可以得到层序遍历结果，但其本质还是二叉树的前序遍历，或者说 DFS 的思路，而不是层序遍历，或者说 BFS 的思路。因为这个解法是依赖前序遍历自顶向下、自左向右的顺序特点得到了正确的结果。抽象点说，**这个解法更像是从左到右的「列序遍历」，而不是自顶向下的「层序遍历」**。所以对于计算最小距离的场景，这个解法完全等同于 DFS 算法，没有 BFS 算法的性能的优势。

第二个花活：

```
class Solution {
private:
    vector<vector<int>> res;

public:
    vector<vector<int>> levelTraverse(TreeNode* root) {
        if (root == NULL) {
            return res;
        }
        vector<TreeNode*> nodes;
        nodes.push_back(root);
        traverse(nodes);
        return res;
    }

    void traverse(vector<TreeNode*>& curLevelNodes) {
        // base case
        if (curLevelNodes.empty()) {
```

```
            return;
        }

        // 前序位置，计算当前层的值和下一层的节点列表
        vector<int> nodeValues;
        vector<TreeNode*> nextLevelNodes;
        for (TreeNode* node : curLevelNodes) {
            nodeValues.push_back(node->val);
            if (node->left != NULL) {
                nextLevelNodes.push_back(node->left);
            }
            if (node->right != NULL) {
                nextLevelNodes.push_back(node->right);
            }
        }

        // 前序位置添加结果，可以得到自顶向下的层序遍历
        res.push_back(nodeValues);

        traverse(nextLevelNodes);

        // 后序位置添加结果，可以得到自底向上的层序遍历结果
        // res.push_back(nodeValues);
    }
};
```

这个 traverse 函数很像递归遍历单链表的函数，其实就是**把二叉树的每一层抽象理解成单链表的一个节点进行遍历**。相较上一个递归解法，这个递归解法是自顶向下的「层序遍历」，更接近 BFS 的奥义，可以作为 BFS 算法的递归实现扩展一下思维。

> 问：最后，不懂什么时候要新增traverse函数，什么时候直接用题目给的原函数递归。比如二叉树的直径，解法是需要新增一个递归函数maxDepth。我尝试直接递归原函数diameterOfBinaryTree，但行不通？
> 答：因为要在 maxDepth 的后序位置**操作外部变量，题目最终要的答案是那个外部变量，而不是 maxDepth 的返回值**。所以这种情况下**只能新建一个函数**。如果题目要的答案恰好是你函数的返回值，那么你才可以直接用原函数递归。

(十) 排列、组合、子集（回溯）

无论是排列、组合还是子集问题，简单说无非就是让你从序列 `nums` 中以给定规则取若干元素，主要有以下几种变体：

- 形式一、**元素无重不可复选**，即 `nums` 中的元素都是唯一的，每个元素最多只能被使用一次，这也是最基本的形式。
  以组合为例，如果输入 `nums = [2,3,6,7]`，和为 `7` 的组合应该只有 `[7]`。
- 形式二、**元素有重不可复选**，即 `nums` 中的元素有可能存在重复，每个元素最多只能被使用一次。
  以组合为例，如果输入 nums = [2,5,2,1,2]，和为 7 的组合应该有两种 [2,2,2,1] 和 [5,2]。

- 形式三、**元素无重可复选**，即 `nums` 中的元素都是唯一的，每个元素可以被使用若干次。
  以组合为例，如果输入 nums = [2,3,6,7]，和为 7 的组合应该有两种 [2,2,3] 和 [7]。
- 当然，也可以说有第四种形式，即元素可重可复选。但既然元素可复选，那又何必存在重复元素呢？**元素去重之后就等同于形式三**，所以这种情况不用考虑。
  上面用组合问题举的例子，但排列、组合、子集问题都可以有这三种基本形式，所以共有 9 种变化。
  除此之外，题目也可以再添加各种限制条件，比如让你求和为 `target` 且元素个数为 `k` 的组合，那这么一来又可以衍生出一堆变体。
  但**无论形式怎么变化，其本质就是穷举所有解**，而这些解呈现**树形结构**，所以合理使用**回溯算法框架**，稍改代码框架即可把这些问题一网打尽。

记住如下子集问题和排列问题的回溯树：

首先，**组合问题和子集问题其实是等价的**，这个后面会讲；至于之前说的三种**变化**形式，无非是在这两棵树上**剪掉或者增加**一些树枝罢了。

1、子集（元素无重不可复选）

参考：78.子集

- 给你一个无重复元素的数组 `nums`，其中每个元素最多使用一次，请你返回 `nums` 的所有子集。

```cpp
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;

public:
    // 回溯：元素无重复，不可重选
    vector<vector<int>> subsets(vector<int>& nums) {
        backtrace(nums, 0);
        return ans;
    }
    // 框架
    void backtrace(vector<int>& nums, int start){
        // 进入一个节点，就开始收集结果（前序位置收集，看回溯树上的收集点）
        ans.push_back(path);
        // 继续添加一个元素
        for(int i=start; i<nums.size(); i++){
            path.push_back(nums[i]);
            backtrace(nums, i+1);
            path.pop_back();
        }
    }
};
```

我们通过保证元素之间的相对顺序不变来防止出现重复的子集。

2、组合（元素无重不可复选）

如果你能够成功的生成所有无重子集，那么你稍微改改代码就能生成所有无重组合了。
比如说，让你在 `nums = [1,2,3]` 中拿 `2` 个元素形成所有的组合，你怎么做？稍微想想就会发现，大小为 `2` 的所有组合，不就是所有大小为 `2` 的子集嘛。所以我说组合和子集是一样的：**大小为 k 的组合就是大小为 k 的子集**。
参考：77.组合

- 给定两个整数 `n` 和 `k`，返回范围 `[1, n]` 中所有可能的 `k` 个数的组合。

反映到代码上，只需要稍改 base case，控制算法**仅仅收集第 k 层节点的值**即可：

```cpp
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;

public:
    // 回溯：组合，就是 k大小子集（这里nums = [1,2,...,n]）
    vector<vector<int>> combine(int n, int k) {
        backtrace(n, k, 1);
        return ans;
    }
    // 框架
    void backtrace(int n, int k, int start){
        // base case：进入节点，收集时，只收集k大小的。且提前终止，不必再往下了。
        if(path.size() == k){
            ans.push_back(path);
            return;
        }
        // 添加一个新元素进入(直接用索引作为数字了)
        for(int i=start; i<=n; i++){
            path.push_back(i);
            backtrace(n, k, i+1);
            path.pop_back();
        }
    }

};
```

3、排列（元素无重不可复选）

就是全排列问题，前面讲过。
参考：46.全排列

- 给定一个不含重复数字的数组 `nums`，返回其所有可能的全排列。

  用 `used` 数组标记已经在路径上的元素避免重复选择，然后收集所有叶子节点上的值，就是所有全排列的结果

```cpp
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;
    // 防复选
    vector<bool> used;

public:
    // 回溯：排列，无重复，不复选
    vector<vector<int>> permute(vector<int>& nums) {
        used.assign(nums.size(), false);
        backtrace(nums);
        return ans;
    }
    // 框架
    void backtrace(vector<int>& nums){
        // 到达叶节点才收集结果
        if(path.size() == nums.size()){
            ans.push_back(path);
            return;
        }
        // 添加一个新元素
        for(int i=0; i<nums.size(); i++){
            if(used[i])  continue;
            path.push_back(nums[i]);
            used[i] = true;
            backtrace(nums);
            path.pop_back();
            used[i] = false;
        }
    }
};
```

但如果题目不让你算全排列，而是让你算**元素个数为 k 的排列**，怎么算？
也很简单，改下 `backtrack` 函数的 `base case`，仅收集第 `k` 层的节点值即可。

4、子集/组合（元素有重不可复选）

参考：90.子集II

- 给你一个整数数组 `nums`，其中可能包含重复元素，请你返回该数组所有可能的子集。
- 比如输入 `nums = [1,2,2]`，你应该输出：
  `[ [],[1],[2],[1,2],[2,2],[1,2,2] ]`

可以看到，[2] 和 [1,2] 这两个结果出现了重复，所以我们需要进行剪枝，如果一个节点有多条值相同的树枝相邻，则只遍历第一条，剩下的都剪掉，不要去遍历：

体现在代码上，需要先进行排序，让**相同的元素靠在一起**，如果发现 `nums[i] == nums[i-1]`，则**跳过**：

```cpp
class Solution {
private:
```

```
    vector<vector<int>> ans;
    vector<int> path;

public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        sort(nums.begin(), nums.end()); // 先排序，保证相同元素在一起
        backtrace(nums, 0);
        return ans;
    }
    // 回溯框架：子集，有重复，不复选。顺序保证非重复性
    void backtrace(vector<int>& nums, int start){
        // 一进来就收集结果
        ans.push_back(path);
        // 添加一个元素
        for(int i=start; i<nums.size(); i++){
            // 剪枝：相同元素不重复选（用当前和前一个来判断更好，可以思考为什么）
            if(i>start && nums[i] == nums[i-1])  continue;
            path.push_back(nums[i]);
            backtrace(nums, i+1);
            path.pop_back();
        }
    }
};
```

组合问题和子集问题是等价的，下面看：
参考：40.组合总和II

- 给你输入 `candidates` 和一个目标和 `target`，从 `candidates` 中找出中所有和为 `target` 的组合。`candidates` 可能存在重复元素，且其中的每个数字最多只能使用一次。
- 说这是一个组合问题，其实换个问法就变成子集问题了：请你计算 `candidates` 中所有和为 `target` 的子集。
- 只要额外用一个 `trackSum` 变量记录回溯路径上的元素和，然后将 `base case` 改一改即可

```
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;
    int curSum = 0;

public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        // 先排序：让相同的数字在一起
        sort(candidates.begin(), candidates.end());
        // 在回溯
        backtrace(candidates, target, 0);
        return ans;
    }
    // 回溯框架：子集，有重复，不复选。顺序保证非重复性。
    void backtrace(vector<int>& candidates, int target, int start){
        // 达到目标才收集结果，同时由于全正数，可以提前结束
```

```
        if(curSum >= target){
            if(curSum == target)  ans.push_back(path);
            return;
        }
        // 添加一个
        for(int i=start; i<candidates.size(); i++){
            // 剪枝
            if(i>start && candidates[i] == candidates[i-1])  continue;
            // 添加
            path.push_back(candidates[i]);
            curSum += candidates[i];
            backtrace(candidates, target, i+1);
            path.pop_back();
            curSum -= candidates[i];
        }
    }
};
```

5、排列（元素有重不可复选）

参考：47.全排列II

- 给你输入一个可包含重复数字的序列 `nums`，请你写一个算法，返回所有可能的全排列。
- 比如输入 `nums = [1,2,2]`，函数返回：
  `[ [1,2,2],[2,1,2],[2,2,1] ]`

完整代码：

```
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;
    // 防复选
    vector<bool> used;

public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        used.assign(nums.size(), false);
        // 先排序
        sort(nums.begin(), nums.end());
        backtrace(nums);
        return ans;
    }
    // 回溯框架：排列，有重复，不复选。used保证不复选。
    void backtrace(vector<int>& nums){
        // 叶子节点收集结果
        if(path.size() == nums.size()){
            ans.push_back(path);
            return;
        }
        // 添加一个
```

```
        for(int i=0; i<nums.size(); i++){
            // 防复选
            if(used[i])  continue;
            // 剪枝：相同元素，一定是前面已经入path，这里才能接第二个
            if(i>0 && nums[i] == nums[i-1] && !used[i-1])  continue;
            // 添加
            path.push_back(nums[i]);
            used[i] = true;
            backtrace(nums);
            path.pop_back();
            used[i] = false;
        }
    }
};
```

- 比如 `[1,2,2']` 和 `[1,2',2]` 应该只被算作同一个排列，但被算作了两个不同的排列。
  所以现在的关键在于，如何设计剪枝逻辑，把这种重复去除掉?
  答案是，**保证相同元素在排列中的相对位置保持不变**。
- 如果 `nums = [1,2,2',2'']`，我只要保证重复元素 `2` 的相对位置固定，比如说 `2 -> 2' -> 2''`，也可以得到无重复的全排列结果。仔细思考，应该很容易明白其中的原理：
  标准全排列算法之所以出现重复，是因为把相同元素形成的排列序列视为不同的序列，但实际上它们应该是相同的；
  而如果固定相同元素形成的序列顺序，当然就避免了重复。
- 那么反映到代码上，你注意看这个剪枝逻辑：

```
// 新添加的剪枝逻辑，固定相同的元素在排列中的相对位置
if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
    // 如果前面的相邻相等元素没有用过，则跳过
    continue;
}
// 选择 nums[i]
```

当出现重复元素时，比如输入 `nums = [1,2,2',2'']`，`2'` 只有在 `2` 已经被使用的情况下才会被选择，同理，`2''` 只有在 `2'` 已经被使用的情况下才会被选择，这就保证了相同元素在排列中的相对位置保证固定。

> 这里拓展一下，如果你把上述剪枝逻辑中的 `!used[i - 1]` 改成 `used[i - 1]`，其实也可以通过所有测试用例，但效率会有所下降，这是为什么呢？之所以这样修改不会产生错误，是因为这种写法相当于维护了 `2'' -> 2' -> 2` 的相对顺序，最终也可以实现去重的效果。
> 但为什么这样写效率会下降呢？因为这个写法剪掉的树枝不够多。
> 比如输入 `nums = [2,2',2'']`，产生的回溯树如下：
>
> 如果用绿色树枝代表 `backtrack` 函数遍历过的路径，红色树枝代表剪枝逻辑的触发，那么 `!used[i - 1]` 这种剪枝逻辑得到的回溯树长这样：
>
> 而 `used[i - 1]` 这种剪枝逻辑得到的回溯树如下：

排列去重，也有读者提出别的剪枝思路：

```cpp
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;
    // 防复选
    vector<bool> used;

public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        used.assign(nums.size(), false);
        // 先排序
        sort(nums.begin(), nums.end());
        backtrace(nums);
        return ans;
    }
    // 回溯框架：排列，有重复，不复选。used保证不复选。
    void backtrace(vector<int>& nums){
        // 叶子节点收集结果
        if(path.size() == nums.size()){
            ans.push_back(path);
            return;
        }
        // 添加一个
        int preNum = 666;   // 本层刚刚的前一个选择。题目有效是-10 ~ 10
        for(int i=0; i<nums.size(); i++){
            // 防复选
            if(used[i])  continue;
            // 剪枝：相同情况不重复做
            if(nums[i] == preNum)  continue;

            // 添加
            path.push_back(nums[i]);
            used[i] = true;
            preNum = nums[i];   // 记录本层前一次选的

            backtrace(nums);

            path.pop_back();
            used[i] = false;
        }
    }
};
```

6、子集/组合（元素无重可复选）

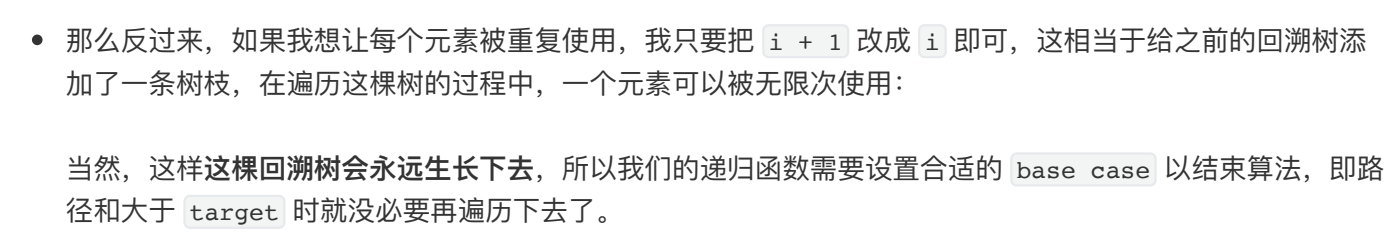
参考：39.组合总和

- 给你一个无重复元素的整数数组 `candidates` 和一个目标和 `target`，找出 `candidates` 中可以使数字和为目标数 `target` 的所有组合。`candidates` 中的每个数字可以无限制重复被选取。

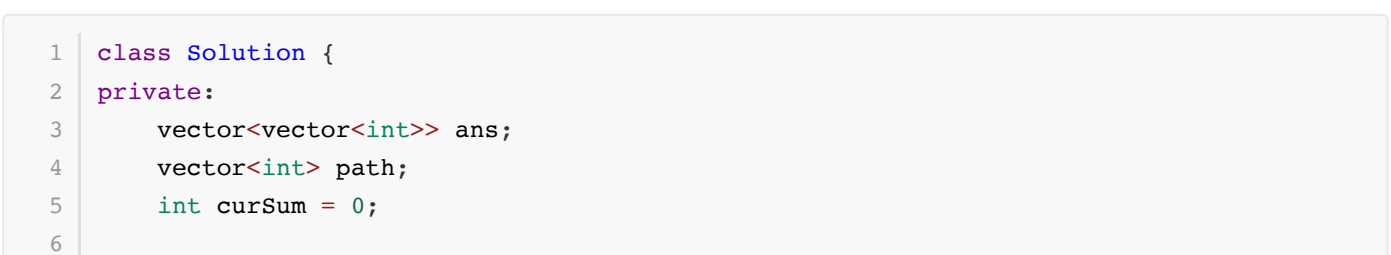
- 输入 `candidates = [1,2,3]`，`target = 3`，算法应该返回：
  `[ [1,1,1],[1,2],[3] ]`

> 想解决这种类型的问题，也得回到回溯树上，我们不妨先思考思考，标准的子集/组合问题是如何保证不重复使用元素的？
> 答案在于 `backtrack` 递归时输入的参数 `start`。

- 这个 `i` 从 `start` 开始，那么下一层回溯树就是从 `start + 1` 开始，从而保证 `nums[start]` 这个元素不会被重复使用：



- 那么反过来，如果我想让每个元素被重复使用，我只要把 `i + 1` 改成 `i` 即可，这相当于给之前的回溯树添加了一条树枝，在遍历这棵树的过程中，一个元素可以被无限次使用：



当然，这样**这棵回溯树会永远生长下去**，所以我们的递归函数需要设置合适的 `base case` 以结束算法，即路径和大于 `target` 时就没必要再遍历下去了。

```cpp
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;
    int curSum = 0;

public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        // 排序，是为了可以提前结束（而且这里因为回溯树会一直生长，必须提前结束）
        // 更正：不必，组合问题，是会遍历所有可能的，并不需要排序。前面题排序是为了去重
        // sort(candidates.begin(), candidates.end());
        backtrace(candidates, target, 0);
        return ans;
    }
    // 回溯框架：组合，无重复，可复选。顺序保证复用正确。
    void backtrace(vector<int>& candidates, int target, int start){
        // base case
        if(curSum >= target){
            if(curSum == target)  ans.push_back(path);
            return;
        }

        // 添加一个
        for(int i=start; i<candidates.size(); i++){
            path.push_back(candidates[i]);
            curSum += candidates[i];
            backtrace(candidates, target, i);   // 把 i+1 改为 i
            path.pop_back();
            curSum -= candidates[i];
        }
    }
};
```

7、排列（元素无重可复选）

参考：力扣上没有题目**直接**考察这个场景，可以用46.全排列来自己测试输入输出。

- 我们不妨先想一下，`nums` 数组中的元素无重复且可复选的情况下，会有哪些排列？
- 比如输入 `nums = [1,2,3]`，那么这种条件下的全排列共有 $3^3 = 27$ 种：

```
[
  [1,1,1],[1,1,2],[1,1,3],[1,2,1],[1,2,2],[1,2,3],[1,3,1],[1,3,2],[1,3,3],
  [2,1,1],[2,1,2],[2,1,3],[2,2,1],[2,2,2],[2,2,3],[2,3,1],[2,3,2],[2,3,3],
  [3,1,1],[3,1,2],[3,1,3],[3,2,1],[3,2,2],[3,2,3],[3,3,1],[3,3,2],[3,3,3]
]
```

标准的全排列算法利用 `used` 数组进行剪枝，避免重复使用同一个元素。
如果允许重复使用元素的话，**直接放飞自我**，去除所有 `used` 数组的剪枝逻辑就行了。

```cpp
class Solution {
private:
    vector<vector<int>> ans;
    vector<int> path;
    // 防复选
    // vector<bool> used;

public:
    // 回溯
    vector<vector<int>> permute(vector<int>& nums) {
        // used.assign(nums.size(), false);
        backtrace(nums);
        return ans;
    }
    // 框架
    void backtrace(vector<int>& nums){
        // 到达叶节点才收集结果
        if(path.size() == nums.size()){
            ans.push_back(path);
            return;
        }
        // 添加一个新元素
        for(int i=0; i<nums.size(); i++){
            // if(used[i])  continue;
            path.push_back(nums[i]);
            // used[i] = true;
            backtrace(nums);
            path.pop_back();
            // used[i] = false;
        }
    }
};
```

8、最后总结

题型	重复说明	是否要 sort	保证非重复性的关键点	收集结果的时机	收集位置
子集	无重复，不复选	No	按顺序控制下一层 `start=i+1`	每一个节点都收集	前序位置
组合	无重复，不复选	No	按顺序控制下一层 `start=i+1`	达到k大小的节点收集	前序位置
排列	无重复，不复选	No	使用 `used` 记录防复用	到达叶子节点（n大小）才收集	前序位置
子集/组合	有重复，不复选	Yes	按顺序控制下一层`start=i+1` + 相同元素排一起	达到题目要求的节点才收集	前序位置
排列	有重复，不复选	Yes	使用 `used` 记录防复用 + 相同元素排一起	到达叶子节点(n 大小)才收集	前序位置
子集/组合	无重复，可复选	No	改 `start = i` 即可复用，注意控制结束	达到题目要求的节点才收集	前序位置
排列	无重复，可复选	No	不用`used` 即可，放飞自我	到达叶子节点(n大小)才收集	前序位置

形式一、元素无重不可复选

```cpp
// 组合/子集问题回溯算法框架
void backtrack(vector<int>& nums, int start) {
    // 收集节点结果
    if(meet the conditions){
        ans.push_back(path);
        // return ; // 也可能不需要
    }

    // 回溯算法标准框架
    for (int i = start; i < nums.size(); i++) {
        // 做选择
        track.push_back(nums[i]);

        // 注意参数
        backtrack(nums, i + 1);

        // 撤销选择
        track.pop_back();
    }
}

// 排列问题回溯算法框架
void backtrack(vector<int>& nums) {
    // 收集节点结果
    if(path.size() == nums.size()){
```

```
        ans.push_back(path);
        return ;
    }

    for (int i = 0; i < nums.size(); i++) {
        // 剪枝逻辑
        if (used[i])  continue;
        // 做选择
        used[i] = true;
        path.push_back(nums[i]);

        backtrack(nums);

        // 撤销选择
        path.pop_back();
        used[i] = false;
    }
}
```

形式二、元素有重不可复选

```
sort(nums.begin(), nums.end());
// 组合/子集问题回溯算法框架
void backtrack(vector<int>& nums, int start) {
    // 收集节点结果
    if(meet the conditions){
        ans.push_back(path);
        // return ; // 也可能不需要
    }

    // 回溯算法标准框架
    for (int i = start; i < nums.size(); i++) {
        // 剪枝逻辑，跳过值相同的相邻树枝
        if (i > start && nums[i] == nums[i - 1])  continue;
        // 做选择
        path.push_back(nums[i]);

        // 注意参数
        backtrack(nums, i + 1);

        // 撤销选择
        path.pop_back();
    }
}

sort(nums.begin(), nums.end());
// 排列问题回溯算法框架
void backtrack(vector<int>& nums, vector<bool>& used) {
    // 收集节点结果
    if(path.size() == nums.size()){
        ans.push_back(path);
```

```
        return ;
    }

    for (int i = 0; i < nums.size(); i++) {
        // 剪枝逻辑
        if (used[i])  continue;
        // 剪枝逻辑，固定相同的元素在排列中的相对位置
        if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1])  continue;
        // 做选择
        path.push_back(nums[i]);
        used[i] = true;

        backtrack(nums, used);

        // 撤销选择
        path.pop_back();
        used[i] = false;
    }
}
```

形式三、元素无重可复选

```
// 组合/子集问题回溯算法框架
void backtrack(vector<int>& nums, int start) {
    // 收集节点结果
    if(meet the conditions){
        ans.push_back(path);
        // return ; // 也可能不需要
    }

    // 回溯算法标准框架
    for (int i = start; i < nums.size(); i++) {
        // 做选择
        path.push_back(nums[i]);

        // 注意参数
        backtrack(nums, i);

        // 撤销选择
        path.pop_back();
    }
}

// 排列问题回溯算法框架
void backtrack(vector<int>& nums) {
    // 收集节点结果
    if(path.size() == nums.size()){
        ans.push_back(path);
        return ;
    }

```

```
    for (int i = 0; i < nums.size(); i++) {
        // 做选择
        path.push_back(nums[i]);

        backtrack(nums);

        // 撤销选择
        path.pop_back();
    }
}
```

只要从树的角度思考，这些问题**看似复杂多变，实则改改 `base case` 就能解决**。

> 后面，还会在球盒模型：回溯算法穷举的两种视角 中进一步解说回溯。

(十一) 贪心算法

举个简单的例子，就能直观的展现贪心算法了。

- 比方说现在有两种钞票，面额分为为 1 元和 100 元，每种钞票的数量无限，但现在你只能选择 10 张，请问你应该如何选择，才能使得总金额最大?
- 那你肯定会说，这还用问么？肯定是 10 张全拿 100 元的钞票，共计 1000 元，这就是最优策略，但凡犹豫一秒就是傻瓜。
- 你这么说，也对，也不对。说你对，因为这确实是最优解法，没毛病。说你不对，是因为这个解法暴露的是你只想捞钱的本质 (¬‿¬) ，跳过了算法的产生、优化过程，不符合计算机思维。
- 那计算机就要提问了，**一切算法的本质是穷举**，现在你还没有穷举出所有可能的解法，凭什么说这就是最优解呢？按照算法思维，这个问题的本质是做 10 次选择，每次选择有两种可能，分别是 1 元和 100 元，一共有 2^{10} 种可能的选择。所以你心里首先应该出现一棵高度为 10 的二叉树来穷举所有可行解，遍历这些可行解，然后可以得到最优解。

```
// 定义：做 n 次选择，返回可以获得的最大金额
int findMax(int n) {
    if (n == 0) return 0;

    // 这次选择 1 元，然后递归求解剩下的 n - 1 次选择的最大值
    int result1 = 1 + findMax(n - 1);
    // 这次选择 100 元，然后递归求解剩下的 n - 1 次选择的最大值
    int result2 = 100 + findMax(n - 1);

    // 返回两种选择中的最大值
    return Math.max(result1, result2);
}
```

`findMax(n - 1)` 的值肯定都一样，那么 `100 + findMax(n - 1)` 必然大于 `1 + findMax(n - 1)`，因此可以进行优化:

```
// 优化一、没必要对两种选择进行比较了
```

```
int findMax(int n) {
    if (n == 0) return 0;
    int result = 100 + findMax(n - 1);
    return result;
}

// 优化二、递归改为迭代
int findMax(int n) {
    int result = 0;
    for (int i = 0; i < n; i++) {
        result += 100;
    }
    return result;
}

// 优化三、直接计算结果就行了
int findMax(int n) {
    return 100 * n;
}
```

这就是贪心算法，复杂度从 $O(2^n)$ 优化到了 $O(1)$，堪称离谱。

1、贪心选择性质

- 作为对比，我们稍微改一改题目：
  现在有两种钞票，面额分别为 1 元和 100 元，每种钞票的数量无限。现在给你一个目标金额 `amount`，请问你最少需要多少张钞票才能凑出这个金额?
  这道题其实就是 动态规划解题套路框架 中讲解的凑零钱问题。
- 为了方便讲解，本文开头的最大金额问题我们称为「问题一」，这里的最少钞票数量问题我们称为「问题二」。
- 我们是如何解决问题二的？首先也是抽象出递归树，写出指数级别的暴力穷举算法，然后发现了重叠子问题，于是用备忘录消除重叠子问题，这就是标准的动态规划算法的求解过程，不能再优化了。
- 所以，这两个问题到底有什么区别？区别在于，**问题二没有贪心选择性质，而问题一有**。

> **贪心选择性质**就是说能够通过局部最优解**直接**推导出全局最优解。

对于问题一，局部最优解就是每次都选择 100 元，因为 100 > 1；对于问题二，局部最优解也是每次都选择 100 元，因为每张面额尽可能大，所需的钞票数量就能尽可能少。但区别在于，**问题一中每一次选择的局部最优解组合起来就是全局最优解，而问题二中不是**。
比方说目标金额 amount = 3，虽然每次选择 100 元是局部最优解，但想凑出 3 元，只能选择 3 张 1 元，局部最优解不一定能构成全局最优解。对于问题二的场景，不符合贪心选择性质，所以不能用贪心算法，只能穷举所有可行解，才能计算出最优解。

> **贪心选择性质 vs 最优子结构**
>
> - 动态规划：问题必须要有「最优子结构」性质。

> - 贪心算法：问题必须要有「贪心选择」性质。
>   最优子结构的意思是说，现在我已经把**所有子问题的最优解**都求出来了，然后我可以**基于这些子问题的最优解推导出原问题的最优解**。
>   贪心选择性质的意思是说，我只需要进行**一些局部最优**的选择策略，就能直接知道哪个子问题的解是最优的了，且这个局部最优解可以推导出原问题的最优解。此时此刻我就能知道，**不需要等到所有子问题的解算出来才知道**。
> - 所以贪心算法的效率一般都比较高，因为它不需要遍历完整的解空间。

2、例题实战

参考：55.跳跃游戏

- 给你一个非负整数数组 `nums` ，你最初位于数组的 第一个下标 。数组中的每个元素代表你在该位置可以跳跃的最大长度。
  判断你是否能够到达最后一个下标，如果可以，返回 `true` ；否则，返回 `false` 。
- 输入：`nums = [2,3,1,1,4]`
  输出：`true`
  解释：可以先跳 1 步，从下标 0 到达下标 1, 然后再从下标 1 跳 3 步到达最后一个下标。
- 注意题目说 `nums[i]` 表示可以跳跃的最大长度，不是固定长度。假设 `nums[i] = 3`，意味着你可以在索引 `i` 往前跳 `1` 步、`2` 步或 `3` 步。
- 我们先思考暴力穷举解法吧，如何穷举所有可能的跳跃路径？你心里的那棵多叉树画出来了没有？假设 `N` 为 `nums` 的长度，这道题相当于做 `N` 次选择，每次选择有 `nums[i]` 种选项，想要穷举所有的跳跃路径，就是一棵高度为 `N` 的多叉树，每个节点有 `nums[i]` 个子节点。
- 这个算法本质上还是穷举了所有可能的选择，可以走动态规划那一套流程进行优化，但是这里我们先不急，可以再仔细想想，这个问题有没有贪心选择性质？
- 这里面有个细节，比方说你现在站在 `nums[i] = 3` 的位置，你可以跳到 `i+1`，`i+2`，`i+3` 三个位置，此时你真的需要分别跳过去，然后递归求解子问题 `dp(i+1)`，`dp(i+2)`，`dp(i+3)`，最后通过子问题的答案来决定 `dp(i)` 的结果吗？
  其实不用的，`i+1`，`i+2`，`i+3` 三个候选项，它们谁能走得最远，你就选谁，准没错。
  具体来说：
  1. `i+1` 能走到的最远距离是 `i+1+nums[i+1]`
  2. `i+2` 能走到的最远距离是 `i+2+nums[i+2]`
  3. `i+3` 能走到的最远距离是 `i+3+nums[i+3]`
     你看看谁最大，就选谁。
- 这就是**贪心选择**性质，通过**局部最优解就能推导全局最优解，不需要等到递归计算出所有子问题的答案才能做选择**。

```cpp
class Solution {
public:
    bool canJump(vector<int>& nums) {
        // 直接遍历所有元素，更新最远即可
        int farthest = 0;
        for(int i=0; i<=farest && i<nums.size(); i++)
            farthest = max(farthest, i+nums[i]);
        // 返回
        return farthest >= nums.size()-1;
    }
};
```

参考:45.跳跃游戏II

- 给定一个长度为 `n` 的 `0` 索引整数数组 `nums`。初始位置为 `nums[0]`。
  每个元素 `nums[i]` 表示从索引 `i` 向前跳转的最大长度。换句话说，如果你在 `nums[i]` 处，你可以跳转到任意 `nums[i + j]` 处:
  `0 <= j <= nums[i]`
  `i + j < n`
  返回到达 `nums[n - 1]` 的最小跳跃次数。生成的测试用例可以到达 `nums[n - 1]`。
- 输入: `nums = [2,3,1,1,4]`
  输出: `2`
  解释: 跳到最后一个位置的最小跳跃数是 2。从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

暴力穷举肯定也是可以做的，就类似上面的那道题，只不过修改一下 dp 数组的定义，返回值从 boolean 改成 int，表示最少需要跳跃的次数就行了。

```cpp
// 定义：从索引 p 跳到最后一格，至少需要 dp(nums, p) 步
int dp(int nums[], int p);

// 题目问的就是 dp(nums, 0) 的结果，base case 就是当 p 超过最后一格时，不需要跳跃
if (p >= nums.length - 1) {
    return 0;
}
```

动态规划做法:

```cpp
class Solution {
public:
    vector<int> memo;
    // 主函数
    int jump(vector<int>& nums) {
        int n = nums.size();
        // 备忘录都初始化为 n，相当于 INT_MAX
        // 因为从 0 跳到 n - 1 最多 n - 1 步
        memo = vector<int>(n, n);

        return dp(nums, 0);
```

```
    }

    // 定义：从索引 p 跳到最后一格，至少需要 dp(nums, p) 步
    int dp(vector<int>& nums, int p) {
        int n = nums.size();
        // base case
        if (p >= n - 1) {
            return 0;
        }
        // 子问题已经计算过
        if (memo[p] != n) {
            return memo[p];
        }
        int steps = nums[p];
        // 你可以选择跳 1 步，2 步...
        for (int i = 1; i <= steps; i++) {
            // 穷举每一个选择
            // 计算每一个子问题的结果
            int subProblem = dp(nums, p + i);
            // 取其中最小的作为最终结果
            memo[p] = min(memo[p], subProblem + 1);
        }
        return memo[p];
    }
};
```

这个解法已经通过备忘录消除了冗余计算，时间复杂度是 `递归深度 x 每次递归需要的时间复杂度`，即 $O(N^2)$，在 LeetCode 上是无法通过所有用例的，会超时。

所以进一步的优化就只能是贪心算法了，我们要仔细思考是否存在贪心选择性质，是否能够通过局部最优解推导全局最优解，避免全量穷举所有的可能解。

和上面的题目是一样的优化思路：我们真的需要递归地计算出每一个子问题的结果，然后求最值吗？其实不需要。

上图这种情况，我们站在索引 0 的位置，可以向前跳 1，2 或 3 步，你说应该选择跳多少呢？

肯定是跳到索引 2 位置的，为什么？因为 2 的选择很多啊，你 3 能去的地方，我 2 都可以去。

这就是贪心选择性质，我们不需要真的递归穷举出所有选择的具体结果来比较求最值，而只需要每次选择那个最有潜力的局部最优解，最终就能得到全局最优解。

```
class Solution {
public:
    // 贪心
    int jump(vector<int>& nums) {
        int n = nums.size();
        int end = 0, farthest = 0;
        int steps = 0;
        for(int i=0; i<n-1; i++){    // 思考是i<n-1
            farthest = max(farthest, i+nums[i]);
            if(i == end){
                steps++;             // 这一步是准备跳了，实际上还没跳（但知道一定会跳的）
```

```
                end = farthest;      // 这里更新的最远，实际上是这一步的可跳范围，还没跳出去
            }
        }
        return steps;
    }
};


// 下面是我自己的写法
class Solution {
public:
    // 贪心
    int jump(vector<int>& nums) {
        int n = nums.size();
        if(n == 1)  return 0;

        int steps = 0;
        int cur = 0, best = 0;
        for(int i=0; i<=cur+nums[cur] && i<n; i++){
            // 当前已经可以到了，那就直接到，不必探索最优待选，反正都是走一步了
            if(cur+nums[cur] >= n-1){
                steps++;
                break;
            }

            // 碰到更优的，则更新待选
            best = i+nums[i] > best+nums[best] ? i : best;

            // 抵达边界，开始真的选了一个走去
            if(i == cur+nums[cur]){
                cur = best;
                steps++;
            }
        }

        return steps;
    }
};
```

时间复杂度是 $O(N)$，空间复杂度是 $O(1)$，可以通过所有测试用例。

3、贪心算法的解题步骤

- 贪心算法的关键在于问题是否具备**贪心选择性质**，所以只能具体问题具体分析，**没办法抽象出一套固定的算法模板或者思维模式**，判断一道题是否是贪心算法。
- 经验是，**没必要刻意地识别一道题是否具备贪心选择性质**。你只需时刻记住，**算法的本质是穷举**，遇到任何题目都要**先想暴力穷举**思路，穷举的过程中如果**存在冗余计算**，就用备忘录**优化**掉。
- 如果提交结果**还是超时，那就说明不需要穷举所有的解空间就能求出最优解，这种情况下肯定需要用到贪心算法**。你可以仔细观察题目，是否可以提前排除掉一些不合理的选择，是否可以**直接**通过局部最优解推导全局最优解。

(十二) 分治算法

分治思想和分治算法不一样。

1、分治思想

- 广义的分治思想是一个宽泛的概念，本站教程中也经常称之为「**分解问题的思路**」。
- 分治思想就是把一个问题分解成若干个子问题，然后分别解决这些子问题，最后合并子问题的解得到原问题的解，这种思想广泛存在于递归算法中。
  例如：

```
int fib(int n) {
    // base case
    if (n == 0 || n == 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}

// 定义：输入一棵二叉树的根节点，返回这棵树的节点总数
int count(TreeNode root) {
    // base case
    if (root == null) {
        return 0;
    }
    // 先算出左右子树的节点个数
    int leftCount = count(root.left);
    int rightCount = count(root.right);

    // 左右子树的节点个数加上自己，就是整棵树的节点个数
    return leftCount + rightCount + 1;
}
```

- 动态规划算法 属不属于分治思想？
  也属于，因为所有动态规划算法都是把大问题分解成了结构相同规模更小的子问题，通过子问题的最优解合并得到原问题的最优解，只不过这个过程中有一些特殊的优化操作罢了。
- 前面讲过，递归算法只有两种思路，一种是遍历的思路，另一种是分解问题的思路（分治思想）。
  遍历思路的典型代表就是 回溯算法，那么除了回溯算法之外，其他递归算法都可以归为分解问题的思路（分治思想）。
- 「**分治思想」占据了递归算法的半壁江山**，那么当我们说「分治算法」的时候，具体是指什么呢？是不是可以说上面列举的这些递归算法都是「分治算法」呢？其实不是的。

2、分治算法

狭义的分治算法也是运用分治思想的递归算法，但它有一个特征，是上面列举的算法所不具备的：
把问题分解后进行求解，相比于不分解直接求解，时间复杂度更低。
符合这个特征的算法，我们才称之为「**分治算法**」。

上面列举的算法，它们本身就只能分解求解，不存在「直接求解」的解法，所以只说它们运用了分治思想，不说它们是分治算法。

- 比如 **桶排序算法**，桶排序的思路是把待排序数组分成若干个桶，然后对每个桶分别进行插入排序，最后把所有有序桶合并，这样时间复杂度能降到 $O(n)$。
  直接用 **插入排序** 的复杂度是 $O(n^2)$，而分解后再用插入排序，总的时间复杂度就能降到 $O(n)$，这种才算分治算法。

那么这里面是什么道理，为什么分而治之的复杂度更低呢？如果把所有问题都分而治之，是不是都能得到更低的复杂度呢？
下面就来详细地对比探究一下，什么情况下分治思想能降低复杂度，什么时候不可以，以及其中的原理所在。

3、无效的分治

理论上讲，很多算法都可以用分解问题的思路改写成递归算法，但大部分情况下这种改写是无意义的。

- 求一个数组的和，这个算法的时间复杂度是 $O(n)$，空间复杂度是 $O(1)$。

```java
int getSum(int[] nums) {
    int sum = 0;
    for (int i = 0; i < nums.length; i++) {
        sum += nums[i];
    }
    return sum;
}
```

完全可以用分解问题的思路把这个问题改写成递归算法：
你问我所有元素的和，我就把这个问题分解成第一个元素和剩余元素的和，这就是分治思想呀。

```java
// 定义：返回 nums[start..] 的元素和
int getSum2(int[] nums, int start) {
    // base case
    if (start == nums.length) {
        return 0;
    }
    // nums[start..] 的元素和可以分解成第一个元素和剩余元素的和
    return nums[start] + getSum2(nums, start + 1);
}
```

递归树的形态类似一条链表，高度为 $O(n)$，这是因为每次递归调用都是 `start + 1`，所以递归树退化成了链表。
这个算法的时间复杂度是 $O(n)$，空间复杂度是 $O(n)$。
可以看到，这个算法的时间复杂度并没有比迭代算法更低，而且还多了 $O(n)$ 的空间复杂度。
那改一下，改成中间劈一半？

```java
int getSum3(int[] nums, int start, int end) {
```

```
    // base case
    if (start == end) {
        return nums[start];
    }

    int mid = start + (end - start) / 2;
    // 计算 nums[start..mid] 的和
    int leftSum = getSum3(nums, start, mid);
    // 计算 nums[mid+1..end] 的和
    int rightSum = getSum3(nums, mid + 1, end);

    // 合并得到 nums[start..end] 的和
    return leftSum + rightSum;
}
```

`getSum2` 算法的递归树退化成了链表，堆栈（树高）的空间复杂度是 $O(n)$；而这个 `getSum3` 算法从中间二分，递归树就是一个较为平衡的二叉树，所以堆栈（树高）的空间复杂度是 $O(\log n)$。
时间复杂度还是 $O(n)$，因为递归调用的次数（二叉树节点数）是 $O(n)$，每次递归调用只做几次加减法，时间复杂度是 $O(1)$，所以总的时间复杂度是 $O(n)$。

综上，这两种分治算法改写都属于无效的分治，没有降低时间复杂度，反而由于递归而增加了空间复杂度。
这也是预期之内的事情，数组元素求和，时间复杂度在怎么优化都不可能低于 $O(n)$，因为你至少得遍历一遍所有元素对吧，这么遍历一次就要 $O(n)$ 的时间了，怎么可能优化呢?

1. **分治的思想是广泛存在的**，几乎所有算法都可以改写成递归分治的形式。
2. **分治思想不等于高效**。不要听到 XX 算法就觉得高大上，很多时候，改写成分治解法并不能带来什么实际的好处，甚至可能增加空间复杂度，因为递归调用需要堆栈空间。

3、**用二分的方式进行分治可以将递归树的深度从 $O(n)$ 降低到 $O(logn)$，确实有优化效果**。对于上面这个元素求和的例子，无论怎么分治都不如原解法高效，但可以看出二分的分治方式是确实有助于减少递归树的高度。

4、有效的分治

参考：23.合并 K 个升序链表

- 给你一个链表数组，每个链表都已经按升序排列。
  请你将所有链表合并到一个升序链表中，返回合并后的链表。
- 示例 1：
  输入：`lists = [[1,4,5],[1,3,4],[2,6]]`
  输出：`[1,1,2,3,4,4,5,6]`

在 单链表双指针技巧汇总 中，我介绍的解法是利用 **优先级队列** 这种数据结构对链表节点进行动态排序，这种解法的时间复杂度是 $O(N \log k)$，空间复杂度是 $O(k)$，其中 `k` 代表链表的条数，`N` 代表 `k` 条链表节点的总数，在本文中，我们不再依赖额外的数据结构，而是直接用分治算法解决这个问题，时间复杂度依然是 $O(N \log k)$。

先看简单的。
参考：21.合并 2 个升序列表

- 将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

- 输入：`l1 = [1,2,4], l2 = [1,3,4]`
  输出：`[1,1,2,3,4,4]`
  这个问题解答过，如下：

```cpp
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        // 虚拟头结点
        ListNode dummy(-1), *p = &dummy;
        ListNode *p1 = l1, *p2 = l2;

        while (p1 != nullptr && p2 != nullptr) {
            // 比较 p1 和 p2 两个指针
            // 将值较小的的节点接到 p 指针
            if (p1->val > p2->val) {
                p->next = p2;
                p2 = p2->next;
            } else {
                p->next = p1;
                p1 = p1->next;
            }
            // p 指针不断前进
            p = p->next;
        }

        if (p1 != nullptr) {
            p->next = p1;
        }

        if (p2 != nullptr) {
            p->next = p2;
        }

        return dummy.next;
    }
};
```

时间复杂度 $O(l1 + l2)$。

下面我们来思考如何合并 k 个有序链表。
先想一个暴力解吧，运用上面的这个 `mergeTwoLists` 函数把 `k` 个链表两两合并，都合并到第一个链表上。

```cpp
ListNode mergeKLists(ListNode[] lists) {
    if (lists.length == 0) {
        return null;
    }
    // 把 k 个有序链表都合并到 lists[0] 上
    ListNode l0 = lists[0];
    for (int i = 1; i < lists.length; i++) {
        l0 = mergeTwoLists(l0, lists[i]);
    }
```

```
    return l0;
}

ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    // 见上文
}
```

但是，链表 l_0 和 l_1 会被遍历 `k-1` 次，l_2 会被遍历 `k-2` 次，以此类推，最后一条链表 l_{k-1} 会被遍历 `1` 次。
看到冗余计算了吗？**越靠前的链表被重复遍历的次数越多，这就是这个算法低效的原因**。我们只要减少这种重复，就能提高算法的效率。
刚刚这个，就等效于下面的代码：

```
// 定义：合并 lists[start..] 为一个有序链表
ListNode mergeKLists2(ListNode[] lists, int start) {
    if (start == lists.length - 1) {
        return lists[start];
    }
    // 合并 lists[start + 1..] 为一个有序链表
    ListNode subProblem = mergeKLists2(lists, start + 1);

    // 合并 lists[start] 和 subProblem，就得到了 lists[start..] 的有序链表
    return mergeTwoLists(lists[start], subProblem);
}
```

这就和前面的 `getSum2` 很像。递归树的形态类似一个单链表，高度为 $O(k)$。
不难发现**重复的次数取决于树高**，上面这个算法的递归树很不平衡，导致递归树退化成链表，树高变为 $O(k)$。
如果能让递归树尽可能地平衡，就能减小树高，进而减少链表的重复遍历次数，提高算法的效率。
如何让递归树平衡呢？就类似上面 `getSum3` 函数的思路，把链表从中间分成两部分，分别递归合并为有两个序链表，最后再将这两部分合并成一个有序链表。

```
// 定义：合并 lists[start..end] 为一个有序链表
ListNode mergeKLists3(ListNode[] lists, int start, int end) {
    if (start == end) {
        return lists[start];
    }

    int mid = start + (end - start) / 2;
    // 合并左半边 lists[start..mid] 为一个有序链表
    ListNode left = mergeKLists3(lists, start, mid);

    // 合并右半边 lists[mid+1..end] 为一个有序链表
    ListNode right = mergeKLists3(lists, mid + 1, end);

    // 合并左右两个有序链表
    return mergeTwoLists(left, right);
}
```

该算法的时间复杂度相当于是把 `k` 条链表分别遍历 $O(\log k)$ 次。那么假设 `k` 条链表的元素总数是 N，该算法的时间复杂度就是 $O(N \log k)$，和 单链表双指针技巧汇总 中介绍的优先级队列解法相同。
再来看空间复杂度，该算法的空间复杂度只有递归树堆栈的开销，也就是 $O(\log k)$，要优于优先级队列解法的 $O(k)$。

5、总结

1. 分治思想在递归算法中是广泛存在的，甚至一些非递归算法，都可以强行改写成分治递归的形式，但并不是所有算法都能用分治思想提升效率。
2. **把递归算法抽象成递归树，如果递归树节点的时间复杂度和树的深度相关，那么使用分治思想对问题进行二分，就可以使递归树尽可能平衡，进而优化总的时间复杂度**。
3. 反之，如果递归树节点的时间复杂度和树的深度无关，那么使用分治思想就没有意义，反而可能引入额外的空间复杂度。

> - `getSum` 函数即便改为递归形式，每个递归节点做的事情无非就是一些加减运算，所以递归节点的时间复杂度总是 $O(1)$，和树的深度无关，所以分治思想不起作用。
> - 而 `mergeKLists` 函数中，每个递归节点都需要合并两个链表，这两个链表是子节点返回的，其长度和递归树的高度相关，所以使用分治思想可以优化时间复杂度。

(十三) 时空复杂度分析

本文会篇幅较长，会涵盖如下几点：

1、利用时间复杂度反推解题思路，减少试错时间。
2、时间都去哪儿了？哪些常见的编码失误会导致算法超时。
3、Big O 表示法的几个基本特点。
4、非递归算法中的时间复杂度分析。
5、数据结构 API 的效率衡量方法（摊还分析）。
6、递归算法的时间/空间复杂度的分析方法，这部分是重点，我会用动态规划和回溯算法举例。

1、复杂度反推解题

- **你应该在开始写代码之前就留意题目给的数据规模**，因为复杂度分析可以避免你在错误的思路上浪费时间，有时候它甚至可以直接告诉你这道题用什么算法。
- 举例来说吧，比如一个题目给你输入一个数组，其长度能够达到 10^6 这个量级，那么我们肯定可以知道这道题的时间复杂度大概要小于 $O(N^2)$，得优化成 $O(N \log N)$ 或者 $O(N)$ 才行。因为如果你写的算法是 $O(N^2)$ 的，最大的复杂度会达到 10^12 这个量级，在大部分判题系统上都是跑不过去的。
  为了把复杂度控制在 $O(N \log N)$ 或者 $O(N)$，我们的选择范围就缩小了，可能符合条件的做法是：**对数组进行排序处理、前缀和、双指针、一维 dp 等等**，从这些思路切入就比较靠谱。
  像**嵌套 for 循环、二维 dp、回溯算法**这些思路，基本可以直接排除掉了。
- 举个更直接的例子，如果你发现题目给的数据规模很小，比如数组长度 N 不超过 20 这样的，那么我们可以断定这道题大概率要用**暴力穷举**算法。
  因为判题平台肯定是尽可能扩大数据规模难为你，它一反常态给这么小的数据规模，肯定是因为**最优解就是指数/阶乘级别**的复杂度。你放心用
  回溯算法 招呼它就行了，不用想别的算法了。

2、编码失误导致异常

- 这些错误会产生预期之外的时间消耗，拖慢你的算法运行，甚至导致超时。

编码失误	原因	解决办法
使用了标准输出	标准输出属于 I/O 操作，会很大程度上拖慢你的算法代码运行	输出语句注释掉
错误地传参数	尤其是函数是递归函数时，错误地传值几乎必然导致超时或超内存	传引用
接口对象的底层实现不明	Java 会有，C++不用管	自己创建

3、Big O 表示法

Big O 记号的数学定义：

$$O(g(n)) = \{f(n)：存在正常量c和n_0，使得对所有n \geq n_0，有0 \leq f(n) \leq c * g(n)\} \tag{2}$$

常用的这个符号 O 其实代表一个**函数的集合**，比如 $O(n^2)$ 代表着一个由 $g(n) = n^2$ **派生出来的一个函数集合**。

1. 只保留增长速率最快的项，其他的项可以省略。
2. Big O 记号表示复杂度的「上界」。

4、算法分析

- 数据结构：一般看平均时间复杂度，而不是最坏时间复杂度。
- 非递归算法：正常计算即可。
- 递归算法：
    1. `递归算法的时间复杂度 = 递归树的节点个数 x 每个节点的时间复杂度`
    2. `递归算法的空间复杂度 = 递归树的高度 + 算法申请的存储空间`

5、最后总结

1. 复杂度分析是一种技术工具，我们应该灵活运用这个工具，辅助我们又快又好地写出解法代码。
2. Big O 标记代表一个函数的集合，用它表示时空复杂度时代表一个上界，所以如果你和别人算的复杂度不一样，可能你们都是对的，只是精确度不同罢了。
3. 时间复杂度的分析不难，关键是你要透彻理解算法到底干了什么事。非递归算法中嵌套循环的复杂度依然可能是线性的；数据结构 API 需要用平均时间复杂度衡量性能；递归算法本质是遍历递归树，时间复杂度取决于递归树中节点的个数（递归次数）和每个节点的复杂度（递归函数本身的复杂度）。

第一章、经典数据结构算法

(一) 链表

参考题：

1. 82. 删除排序链表中的重复元素 II

```cpp
class Solution {
public:
    // 链表分解思想：分成一条重复的链表，一条独特的链表
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode *dummy1 = new ListNode(101);   // 题给数据 -100 -- +100
        ListNode *dummy2 = new ListNode();
        ListNode *p1 = dummy1, *p2 = dummy2;
        ListNode *p = head;
        while(p != NULL){
            // 重复节点
            if((p->next != NULL && p->val == p->next->val) || (p->val == p1->val)){
                p1->next = p;
                p1 = p1->next;
                p = p->next;
                p1->next = NULL;
            }
            // 不重复节点
            else{
                p2->next = p;
                p2 = p2->next;
                p = p->next;
                p2->next = NULL;
            }
            // 注意上面的最后一步！！！断开原链表
        }

        // 返回不重复链表即可
        return dummy2->next;
    }
};
```

2. 264. 丑数 II

可以先做附：204.计算质数

```cpp
class Solution {
public:
    // 素数筛（注意 0/1 既不是素数、也不是合数）
    // 返回小于n的质数数量
    // 请完全背下来！！！！！一点不用改
    int countPrimes(int n) {
        vector<bool> F(n, true);
        // F[0] = F[1] = false; // 用不上，而且可能会n<1

        // 去除合数
        for(int i=2; i*i<n; i++)            // 提前 i*i 计算
            if(F[i]){                       // 只有素数才筛选
```

```
                for(int j=i*i; j<n; j+=i)  // 从 i*i 开始，每次+=i
                    F[j] = false;
            }

        // 统计结果
        int cnt = 0;
        for(int i=2; i<n; i++)  if(F[i])  cnt++;
        return cnt;
    }
};
```

本题

```
class Solution {
public:
    // 理解为三个链表合并
    int nthUglyNumber(int n) {
        vector<int> ugly(n);
        int p2 = 0, p3 = 0, p5 = 0;   // 索引
        int r2 = 1, r3 = 1, r5 = 1;   // 下一个计算结果
        int p = 0;
        while(p < n){
            // 选最小的接上
            int minR = min({r2, r3, r5});
            ugly[p] = minR;
            p++;
            // 更新被取走的链表的下一个计算结果
            if(minR == r2){
                r2 = ugly[p2] * 2;
                p2++;
            }
            if(minR == r3){
                r3 = ugly[p3] * 3;
                p3++;
            }
            if(minR == r5){
                r5 = ugly[p5] * 5;
                p5++;
            }
        }
        // 返回
        return ugly[n-1];
    }
};
```

3. 有序矩阵中第 K 小的元素

```
class Solution {
public:
    // 优先队列，合并多行有序链表，找到第k小
```

```
    int kthSmallest(vector<vector<int>>& matrix, int k) {
        int m = matrix.size(), n = matrix[0].size();
        // 优先队列
        auto cmp = [](const vector<int>& a, const vector<int>& b){
            return a[0] > b[0];
        };
        priority_queue<vector<int>, vector<vector<int>>, decltype(cmp)> minHeap;

        // 初始化：元素值， 元素坐标 x,y
        for(int i=0; i<m; i++)  minHeap.push({matrix[i][0], i, 0});

        // 依次插入 k 个即可
        int ans;
        while(k>0){
            // 当前第几小的
            vector<int> cur = minHeap.top();
            minHeap.pop();
            ans = cur[0];
            // 谁被去掉，换那条的上来(取完了就不用了)
            int i = cur[1], j = cur[2];
            if(j+1 < matrix[i].size())  minHeap.push({matrix[i][j+1], i, j+1});
            // 继续
            k--;
        }

        return ans;
    }
};
```

4. 查找和最小的 K 对数字

```
class Solution {
public:
    // 相当于有 n1 条 n2 长度的链表
    vector<vector<int>> kSmallestPairs(vector<int>& nums1, vector<int>& nums2, int k)
{
        int n1 = nums1.size(), n2 = nums2.size();
        // 存储内容，是 加和数字 aNum 以及 索引 i1, i2
        auto cmp = [](const vector<int>& a, const vector<int>& b){
            return a[0] > b[0];
        };
        priority_queue<vector<int>, vector<vector<int>>, decltype(cmp)> minHeap;

        // 初始化
        for(int i=0; i<n1; i++)  minHeap.push({nums1[i]+nums2[0], i, 0});

        // 更新出 K 个即可
        vector<vector<int>> ans;
        while(k > 0){
            vector<int> cur = minHeap.top();
            minHeap.pop();
```

```
            int i = cur[1], j = cur[2];
            ans.push_back({nums1[i], nums2[j]});
            // 推一个新的（没有就不用推了）
            if(j+1 < nums2.size())
                minHeap.push({nums1[i]+nums2[j+1], i, j+1});
            // 继续
            k--;
        }

        return ans;
    }
};
```

5. 两数相加

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode();
        ListNode *p1 = l1, *p2 = l2;
        ListNode* p = dummy;
        int carry = 0;
        while(p1 || p2){
            int res = carry;
            if(p1)  res += p1->val, p1 = p1->next;
            if(p2)  res += p2->val, p2 = p2->next;

            ListNode *cur = new ListNode(res % 10);
            p->next = cur;
            p = p->next;

            carry = res / 10;
        }
        if(carry){
            ListNode *cur = new ListNode(carry);
            cur->next = NULL;
            p->next = cur;
        }

        return dummy->next;
    }
};
```

6. 两数相加 II

```
class Solution {
public:
    // 当然可以反转链表，不反转怎么办？—— 栈！
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        // 栈存储，甚至直接存储 int 即可
```

```
        stack<int> stk1;
        ListNode *p1 = l1;
        while(p1){
            stk1.push(p1->val);
            p1 = p1->next;
        }
        stack<int> stk2;
        ListNode *p2 = l2;
        while(p2){
            stk2.push(p2->val);
            p2 = p2->next;
        }

        // 回到正常思路了
        stack<int> ans;
        int carry = 0;
        while(!stk1.empty() || !stk2.empty()){
            int res = carry;
            if(!stk1.empty()){
                int a = stk1.top();
                stk1.pop();
                res += a;
            }
            if(!stk2.empty()){
                int b = stk2.top();
                stk2.pop();
                res += b;
            }
            ans.push(res % 10);
            carry = res / 10;
        }
        if(carry)  ans.push(carry);

        // 最后存回链表
        ListNode *dummy = new ListNode();
        ListNode *p = dummy;
        while(!ans.empty()){
            ListNode *r = new ListNode(ans.top());
            r->next = NULL;
            ans.pop();
            p->next = r;
            p = p->next;
        }

        return dummy->next;
    }
};
```

7、方法：花式反转链表

1. 反转整个单链表
   参考206.反转链表

```cpp
class Solution {
public:
    // 反转以 head 为起点的单链表
    ListNode* reverseList(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return head;
        }
        // 由于单链表的结构，至少要用三个指针才能完成迭代反转
        // cur 是当前遍历的节点，pre 是 cur 的前驱结点，nxt 是 cur 的后继结点
        ListNode *pre, *cur, *nxt;
        pre = nullptr; cur = head; nxt = head->next;
        while (cur != nullptr) {
            // 逐个结点反转
            cur->next = pre;
            // 更新指针位置
            pre = cur;
            cur = nxt;
            if (nxt != nullptr) {
                nxt = nxt->next;
            }
        }
        // 返回反转后的头结点
        return pre;
    }
};
```

`reverseList(1->2->3->4) = reverseList(2->3->4) -> 1`
这就是「分解问题」的思路，通过递归函数的定义，把原问题分解成若干规模更小、结构相同的子问题，最后通过子问题的答案组装原问题的解。

```cpp
class Solution {
public:
    // 递归写法
    ListNode* reverseList(ListNode* head) {
        // base case
        if(head == NULL || head->next == NULL)
            return head;
        // 把后续反转，再接上
        ListNode *second = head->next;
        ListNode *last = reverseList(second);
        second->next = head;
        head->next = NULL;

        return last;
    }
};
```

值得一提的是，递归操作链表并不高效。
递归解法和迭代解法相比，时间复杂度都是 $O(N)$，但是迭代解法的空间复杂度是 $O(1)$，而递归解法需要堆栈，空间复杂度是 $O(N)$。

2. 反转链表前 N 个节点

```cpp
ListNode* reverseN(ListNode* head, int n) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }
    ListNode *pre, *cur, *nxt;
    pre = nullptr; cur = head; nxt = head->next;
    while (n > 0) {
        cur->next = pre;
        pre = cur;
        cur = nxt;
        if (nxt != nullptr) {
            nxt = nxt->next;
        }
        n--;
    }
    // 此时的 cur 是第 n + 1 个节点，head 是反转后的尾结点
    head->next = cur;
    // 此时的 pre 是反转后的头结点
    return pre;
}
```

递归写法：

```cpp
// 后驱节点
ListNode* successor = nullptr;

// 反转以 head 为起点的 n 个节点，返回新的头结点
ListNode* reverseN(ListNode* head, int n) {
    if (n == 1) {
        // 记录第 n + 1 个节点
        successor = head->next;
        return head;
    }
    // 以 head.next 为起点，需要反转前 n - 1 个节点
    ListNode* last = reverseN(head->next, n - 1);

    head->next->next = head;
    // 让反转之后的 head 节点和后面的节点连起来
    head->next = successor;
    return last;
}
```

3. 反转链表的一部分
   参考：92. 反转链表 II

```cpp
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int m, int n) {
        if (m == 1) {
            return reverseN(head, n);
        }
        // 找到第 m 个节点的前驱
        ListNode *pre = head;
        for (int i = 1; i < m - 1; i++) {
            pre = pre->next;
        }
        // 从第 m 个节点开始反转
        pre->next = reverseN(pre->next, n - m + 1);
        return head;
    }

    ListNode* reverseN(ListNode* head, int n) {
        if (head == nullptr || head->next == nullptr) {
            return head;
        }
        ListNode *pre, *cur, *nxt;
        pre = nullptr; cur = head; nxt = head->next;
        while (n > 0) {
            cur->next = pre;
            pre = cur;
            cur = nxt;
            if (nxt != nullptr) {
                nxt = nxt->next;
            }
            n--;
        }
        // 此时的 cur 是第 n + 1 个节点，head 是反转后的尾结点
        head->next = cur;
        // 此时的 pre 是反转后的头结点
        return pre;
    }
};
```

递归写法：

```cpp
class Solution {
public:
    // 后驱节点
    ListNode* successor = nullptr;

    // 反转以 head 为起点的 n 个节点，返回新的头结点
    ListNode* reverseN(ListNode* head, int n) {
        if (n == 1) {
            // 记录第 n + 1 个节点
            successor = head->next;
            return head;
```

```
        }
        ListNode* last = reverseN(head->next, n - 1);

        head->next->next = head;
        head->next = successor;
        return last;
    }

    ListNode* reverseBetween(ListNode* head, int m, int n) {
        // base case
        if (m == 1) {
            return reverseN(head, n);
        }
        // 前进到反转的起点触发 base case
        head->next = reverseBetween(head->next, m - 1, n - 1);
        return head;
    }
};
```

4. K 个一组反转链表

参考：25. K 个一组翻转链表

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        if (head == nullptr) return nullptr;
        // 区间 [a, b) 包含 k 个待反转元素
        ListNode *a, *b;
        a = b = head;
        for (int i = 0; i < k; i++) {
            // 不足 k 个，不需要反转了
            if (b == nullptr) return head;
            b = b->next;
        }
        // 反转前 k 个元素
        ListNode *newHead = reverseN(a, k);
        // 此时 b 指向下一组待反转的头结点
        // 递归反转后续链表并连接起来
        a->next = reverseKGroup(b, k);
        return newHead;
    }

    // 上文实现的反转前 N 个节点的函数
    ListNode* reverseN(ListNode* head, int n) {
        if (head == nullptr || head->next == nullptr) {
            return head;
        }
        ListNode *pre, *cur, *nxt;
        pre = nullptr; cur = head; nxt = head->next;
        while (n > 0) {
            cur->next = pre;
```

```
            pre = cur;
            cur = nxt;
            if (nxt != nullptr) {
                nxt = nxt->next;
            }
            n--;
        }
        head->next = cur;
        return pre;
    }
};
```

8、方法：回文链表判断

参考：234.回文链表

- **寻找**回文串的核心思想是从中心向两端扩展
- **判断**回文串，双指针技巧从两端向中间逼近即可
- 那么对于回文链表，怎么办?

1. **思考做法**

链表兼具递归结构，树结构不过是链表的衍生。那么，**链表其实也可以有前序遍历和后序遍历**，借助二叉树后序遍历的思路，不需要显式反转原始链表也可以倒序遍历链表：

```
// 二叉树遍历框架
void traverse(TreeNode* root) {
    // 前序遍历代码
    traverse(root->left);
    // 中序遍历代码
    traverse(root->right);
    // 后序遍历代码
}

// 递归遍历单链表
void traverse(ListNode* head) {
    // 前序遍历代码
    traverse(head->next);
    // 后序遍历代码
}
```

比如说：

```
// 倒序打印单链表中的元素值
void traverse(ListNode* head) {
    if (head == NULL) return;
    traverse(head->next);
    // 后序遍历代码
    cout << head->val << endl;
}
```

本题做法：

```cpp
class Solution {
public:
    // 从左向右移动的指针
    ListNode* left;
    // 从右向左移动的指针
    ListNode* right;

    // 记录链表是否为回文
    bool res = true;

    bool isPalindrome(ListNode* head) {
        left = head;
        traverse(head);
        return res;
    }

    void traverse(ListNode* right) {
        if (right == nullptr) {
            return;
        }

        // 利用递归，走到链表尾部
        traverse(right->next);

        // 后序遍历位置，此时的 right 指针指向链表右侧尾部
        // 所以可以和 left 指针比较，判断是否是回文链表
        if (left->val != right->val) {
            res = false;
        }
        left = left->next;
    }
};
```

这么做的核心逻辑是什么呢？实际上就是**把链表节点放入一个栈，然后再拿出来，这时候元素顺序就是反的**，只不过我们利用的是递归函数的堆栈而已。

2. **优化空间复杂度**
   不必要求

```cpp
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        ListNode* slow, *fast;
        slow = fast = head;
        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next;
        }

```

```
        if (fast != nullptr)
            slow = slow->next;

        ListNode* left = head;
        ListNode* right = reverse(slow);
        while (right != nullptr) {
            if (left->val != right->val)
                return false;
            left = left->next;
            right = right->next;
        }

        return true;
    }

    ListNode* reverse(ListNode* head) {
        ListNode* pre = nullptr, *cur = head;
        while (cur != nullptr) {
            ListNode* next = cur->next;
            cur->next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }
};
```

这种解法虽然高效，但破坏了输入链表的原始结构，能不能避免这个瑕疵呢?

其实这个问题很好解决，关键在于得到p, q这两个指针位置。只要在函数 return 之前加一段代码即可恢复原先链表顺序：`p->next = reverse(q);`

(二) 数组

1、双指针的七道题

1. 有序数组去重，原地修改：快慢指针
2. 移除val元素，原地修改：快慢指针
3. 滑动窗口，背下框架
4. 二分查找：左右指针
5. 两数之和：左右指针
6. 反转数组：左右指针
7. 最长回文串判断：核心子函数是中心向两端扩散的左右指针

> 只要数组有序，就应该想到双指针技巧。

全部代码如下：

```
T1:
class Solution {
public:
    // 快慢指针
    ListNode* deleteDuplicates(ListNode* head) {
        if(head == NULL)  return NULL;
        // fast 探路, slow 填写
        ListNode *slow = head, *fast = head;
        while(fast != NULL){
            if(fast->val != slow->val){
                slow->next = fast;
                slow = slow->next;
            }
            fast = fast->next;
        }
        // 注意切断
        slow->next = NULL;
        return head;
    }
};

T2:
class Solution {
public:
    // 快慢指针
    int removeElement(vector<int>& nums, int val) {
        int fast = 0, slow = 0;
        while(fast < nums.size()){
            if(nums[fast] != val)
                nums[slow++] = nums[fast];
            fast++;
        }
        return slow;
    }
};

T5:
class Solution {
public:
    // 左右指针
    vector<int> twoSum(vector<int>& numbers, int target) {
        int left = 0, right = numbers.size()-1;
        while(left < right){
            int sum = numbers[left] + numbers[right];
            // 完成
            if(sum == target)  return vector<int>{left+1, right+1}; // 题目索引从1开始
            // 让 sum 小一点
            else if(sum > target)  right--;
            // 让 sum 大一点
            else  left++;
        }

```

```
        return vector<int>{-1, -1};
    }
};

T6:
class Solution {
public:
    // 左右指针
    void reverseString(vector<char>& s) {
        int left = 0, right = s.size()-1;
        while(left < right){
            char t = s[left];
            s[left] = s[right];
            s[right] = t;
            left++, right--;
        }
    }
};

T7:
class Solution {
public:
    // 找到以 l, r 为中心的最长子回文串(若l=r，则说明以一个字符为中心)
    string palindrome(string& s, int l, int r){
        // 注意越界
        while(l>=0 && r<s.size() && s[l]==s[r])
            l--, r++;
        // 返回
        return s.substr(l+1, r-l-1);    // 注意，退出循环时，l,r 都扩大了
    }

    string longestPalindrome(string s) {
        string ans = "";
        for(int i=0; i<s.size(); i++){
            // 中心扩散
            string s1 = palindrome(s, i, i);
            string s2 = palindrome(s, i, i+1);
            // 更新答案
            ans = s1.size() > ans.size() ? s1 : ans;
            ans = s2.size() > ans.size() ? s2 : ans;
        }
        return ans;
    }
};

```

2、二维数组的花式遍历

1. 反转字符串的单词
   先反转整个字符串，再反转每个单词。

2. 旋转数组
   做法是：顺时针旋转 90 度 = 先沿左上右下对角线镜像，再每行反转即可。
   若是逆时针旋转呢？顺时针旋转 90 度 = 先沿右上左下对角线镜像，再每行反转即可。
3. 矩阵螺旋遍历
   本质上就是按照“右、下、左、上”的固定循环顺序遍历，每次碰到边界后，改变方向并更新边界。
4. 生成螺旋矩阵
   一样的思路。

全部代码：

```
T1:
class Solution {
public:
    string reverseWords(string s) {
        reverse(s.begin(), s.end());
        string ans;
        stack<char> STK;
        for(char c : s){
            // 吐单词
            if(c == ' '){
                if(!STK.empty()){
                    while(!STK.empty()){
                        ans += STK.top();
                        STK.pop();
                    }
                    ans += ' ';
                }
            }
            // 存单词
            else{
                STK.push(c);
            }
        }
        // 末尾还没吐的
        while(!STK.empty()){
            ans += STK.top();
            STK.pop();
        }
        // 去除尾部空格
        if(ans[ans.size()-1] == ' ')
            ans = ans.substr(0, ans.size()-1);

        return ans;
    }
};

T2:
class Solution {
public:
    // 做法：顺时针旋转 90 度 = 先左上右下对角线镜像，再每行反转
```

```cpp
    void rotate(vector<vector<int>>& matrix) {
        int n = matrix.size();
        // 1. 镜像
        for(int i=0; i<n; i++)
            for(int j=i; j<n; j++)
                swap(matrix[i][j], matrix[j][i]);

        // 2. 每行反转
        for(auto &row : matrix)  reverse(row.begin(), row.end());
    }
};
// 若是逆时针，那么
// 逆时针旋转 90 度 = 先右上左下对角线镜像，再每行反转

T3:
class Solution {
private:
    vector<int> ans;

public:
    // 本质上，是按照 右、下、左、上 的顺序碰边界后改方向
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        int m = matrix.size(), n = matrix[0].size();
        // 方向 0/1/2/3
        int direct = 0;
        // 边界
        int minR = 0, minC = 0, maxR = m-1, maxC = n-1;
        int i = 0, j = 0;
        while(ans.size() < m*n){
            ans.push_back(matrix[i][j]);
            // 右
            if(direct == 0){
                if(j < maxC)
                    j++;
                else{
                    direct = (direct+1) % 4;
                    minR++;
                    i++;
                }
            }
            // 下
            else if(direct == 1){
                if(i < maxR)
                    i++;
                else{
                    direct = (direct+1) % 4;
                    maxC--;
                    j--;
                }
            }
            // 左
            else if(direct == 2){
```

```
                if(j > minC)
                    j--;
                else{
                    direct = (direct+1) % 4;
                    maxR--;
                    i--;
                }
            }
            // 上
            else if(direct == 3){
                if(i > minR)
                    i--;
                else{
                    direct = (direct+1) % 4;
                    minC++;
                    j++;
                }
            }
        }

        return ans;
    }
};

T4:
class Solution {
public:
    // 本质就是按照 右、下、左、上 遍历，碰到边界后，改方向，更新边界
    // 可以换一种写法
    vector<vector<int>> generateMatrix(int n) {
        vector<vector<int>> M(n, vector<int>(n));
        int top = 0, bottom = n-1;
        int left = 0, right = n-1;
        int num = 1;
        while(num <= n*n){
            // 右(注意 if 里面用的是 M 需要扩展的那行)
            if(top <= bottom){
                for(int i=left; i<=right; i++){
                    M[top][i] = num;
                    num++;
                }
                top++;
            }
            // 下
            if(left <= right){
                for(int i=top; i<=bottom; i++){
                    M[i][right] = num;
                    num++;
                }
                right--;
            }
            // 左
```

```
            if(top <= bottom){
                for(int i=right; i>=left; i--){
                    M[bottom][i] = num;
                    num++;
                }
                bottom--;
            }
            // 上
            if(left <= right){
                for(int i=bottom; i>=top; i--){
                    M[i][left] = num;
                    num++;
                }
                left++;
            }
        }

        // 结束返回
        return M;
    }
};

```

3、团灭 nSum 问题

> 这类 nSum 问题就是给你输入一个数组 `nums` 和一个目标和 `target`，让你从 `nums` 选择 `n` 个数，使得这些数字之和为 `target`。

1. 2Sum
   做法：
   - 先排序（若无序）
   - 再左右指针法
     泛化一下：`nums` 中可能有多对儿元素之和都等于 `target`，请你的算法返回所有和为 `target` 的元素对儿，其中不能出现重复。

2. 3Sum
   做法：
   - 排序
   - 固定第一个数（注意别重复）
   - 第二三个数字，调用 2Sum 完成 target-num1
3. 4Sum
   做法：
   - 排序
   - 固定第一个数字（注意别重复）
   - 第二三四个数字，调用 3Sum 完成 target-num1

4. 100Sum? 仍然用4Sum举例
   做法：写个通用的函数，实际上就是递归 nSum
   - 最外层排序，
   - 然后调用 nSum，执行如下：
     - base case 是 2Sum
     - 其余，固定一个数字，求 n-1 Sum

> 1. 问：如果就是希望返回原下标，而阻碍我们第一步排序，怎么办？
>   答：存储为 pair 把下标也记着。
> 2. 问：这不就是子集问题吗？直接回溯法不行吗？
>   答：可以是可以，一切问题都是穷举。但是，随着 nSum 的 n 提高，回溯法的复杂度飙升。

全部代码：

T1：

```
class Solution {
public:
    // 无序 2Sum
    // 排序 + 左右指针
    vector<int> twoSum(vector<int>& numbers, int target) {
        sort(numbers.begin(), numbers.end());
        int l = 0, r = numbers.size() - 1;
        while(l < r){
            int sum = numbers[l] + numbers[r];
            // 正确
            if(sum == target)
                return vector<int>{l+1, r+1};
            // 大了
            else if(sum > target)
                r--;
            // 小了
            else if(sum < target)
                l++;
        }

        // 没找到
        return vector<int>{-1, -1};
    }
};
```

T1-泛化题型：

```
vector<vector<int>> twoSumTarget(vector<int>& nums, int target) {
    // nums 数组必须有序
    sort(nums.begin(), nums.end());
    int lo = 0, hi = nums.size() - 1;
    vector<vector<int>> res;
    while (lo < hi) {
        int sum = nums[lo] + nums[hi];
        int left = nums[lo], right = nums[hi];
```

```
        if (sum < target) {
            while (lo < hi && nums[lo] == left) lo++;
        } else if (sum > target) {
            while (lo < hi && nums[hi] == right) hi--;
        } else {
            res.push_back({left, right});
            while (lo < hi && nums[lo] == left) lo++;
            while (lo < hi && nums[hi] == right) hi--;
        }
    }
    return res;
}

T2:
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        return threeSum(nums, 0);
    }

    // 泛化题目 O(NlogN + n^2)
    vector<vector<int>> threeSum(vector<int>& nums, int target){
        // 先排序
        sort(nums.begin(), nums.end());
        // 第一个数字固定，再寻找 2Sum 即可
        int n = nums.size();
        vector<vector<int>> ans;
        for(int i=0; i<n; i++){
            vector<vector<int>> tuples = twoSum(nums, i+1, target-nums[i]);
            for(auto &tuple : tuples){
                tuple.insert(tuple.begin(), nums[i]);
                ans.push_back(tuple);
            }
            // 避免重复(这里只是保证下一个不同，后续for有+1的)
            while(i+1<n && nums[i]==nums[i+1]) i++;
        }

        return ans;
    }

private:
    // 2Sum O(N)
    vector<vector<int>> twoSum(vector<int> &nums, int start, int target){
        int n = nums.size();
        vector<vector<int>> ans;
        int l = start, r = n-1;
        while(l < r){
            int sum = nums[l] + nums[r];
            int lnum = nums[l], rnum = nums[r];
            if(sum == target){
                ans.push_back({lnum, rnum});
                // 防重复
```

```
                while(l<r && nums[l]==lnum)  l++;
                while(l<r && nums[r]==rnum)  r--;
            }
            else if(sum > target)
                r--;
            else if(sum < target)
                l++;
        }

        return ans;
    }
};

T3:
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        // 先排序
        sort(nums.begin(), nums.end());
        // 固定第一个
        int n = nums.size();
        vector<vector<int>> ans;
        for(int i=0; i<n; i++){
            auto tuples = threeSum(nums, i+1, target-nums[i]);
            for(auto &tuple : tuples){
                tuple.insert(tuple.begin(), nums[i]);
                ans.push_back(tuple);
            }
            while(i+1<n && nums[i]==nums[i+1])  i++;
        }

        return ans;
    }

private:
    vector<vector<int>> threeSum(vector<int>& nums, int start, long target){    // 用 long
        int n = nums.size();
        vector<vector<int>> ans;
        for(int i=start; i<n; i++){
            auto tuples = twoSum(nums, i+1, target-nums[i]);
            for(auto &tuple : tuples){
                tuple.insert(tuple.begin(), nums[i]);
                ans.push_back(tuple);
            }
            while(i+1<n && nums[i]==nums[i+1])  i++;
        }
        return ans;
    }
    vector<vector<int>> twoSum(vector<int>& nums, int start, long target){  // 用 long
        int n = nums.size();
```

```
        vector<vector<int>> ans;
        int l = start, r = n-1;
        while(l < r){
            int sum = nums[l] + nums[r];
            int lnum = nums[l], rnum = nums[r];
            if(sum > target)  r--;
            else if(sum < target)  l++;
            else{
                ans.push_back({lnum, rnum});
                while(l<r && nums[l]==lnum)  l++;
                while(l<r && nums[r]==rnum)  r--;
            }
        }
        return ans;
    }
};


T4：(这里仍然用 4Sum 举例)
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        sort(nums.begin(), nums.end());
        return nSumTarget(nums, 4, 0, target);
    }

private:
    // 注意：调用这个函数之前一定要先给 nums 排序
    // n 填写想求的是几数之和，start 从哪个索引开始计算（一般填 0），target 填想凑出的目标和
    vector<vector<int>> nSumTarget(vector<int>& nums, int n, int start, long target)
{
        int sz = nums.size();
        vector<vector<int>> res;
        // 至少是 2Sum，且数组大小不应该小于 n
        if (n < 2 || sz < n) return res;
        // 2Sum 是 base case
        if (n == 2) {
            // 双指针那一套操作
            int lo = start, hi = sz - 1;
            while (lo < hi) {
                int sum = nums[lo] + nums[hi];
                int left = nums[lo], right = nums[hi];
                if (sum < target) {
                    while (lo < hi && nums[lo] == left) lo++;
                } else if (sum > target) {
                    while (lo < hi && nums[hi] == right) hi--;
                } else {
                    res.push_back({left, right});
                    while (lo < hi && nums[lo] == left) lo++;
                    while (lo < hi && nums[hi] == right) hi--;
                }
            }
```

```
        } else {
            // n > 2 时，递归计算 (n-1)Sum 的结果
            for (int i = start; i < sz; i++) {
                vector<vector<int>> sub = nSumTarget(nums, n - 1, i + 1, target - nums[i]);
                for (vector<int>& arr : sub) {
                    // (n-1)Sum 加上 nums[i] 就是 nSum
                    arr.push_back(nums[i]);
                    res.push_back(arr);
                }
                while (i < sz - 1 && nums[i] == nums[i + 1]) i++;
            }
        }
        return res;
    }
};
```

4、前缀和数组

1. 一维数组：区域和检索
   由于每次都检索区域和，因此只需要存储前缀和待用即可。

   做法：存储前缀和（每次检索只需要 O(1)了）
2. 二维数组：区域和检索
   也有前缀和办法。

   做法：如上

> 注意：
> 前缀和技巧是利用预计算的 preSum 数组快速计算索引区间内的元素和，但实际上它不仅仅局限于求和，也可以用来快速计算区间乘积等其他场景。
>
> 1. 第一个局限性：使用前缀和技巧的前提是原数组 nums 不会发生变化。
> 2. 第二个局限性：前缀和技巧只适用于存在逆运算的场景。（如求最大值，后续会用线段树来解决）

全部代码：

```
T1:
class NumArray {
private:
    vector<int> preSum;

public:
    NumArray(vector<int>& nums) {
        int n = nums.size();
        preSum.resize(n+1);
        preSum[0] = 0;
        for(int i=0; i<n; i++)
            preSum[i+1] = preSum[i] + nums[i];
    }
```

```
    int sumRange(int left, int right) {
        return preSum[right+1] - preSum[left];
    }
};

T2:
class NumMatrix {
private:
    // 记录 (0,0) - (i,j) 的矩阵和
    vector<vector<int>> preSum;
public:
    NumMatrix(vector<vector<int>>& matrix) {
        int m = matrix.size(), n = matrix[0].size();
        preSum.resize(m+1, vector<int>(n+1, 0));
        for(int i=0; i<m; i++)
            for(int j=0; j<n; j++)
                preSum[i+1][j+1] = preSum[i][j+1] + preSum[i+1][j] - preSum[i][j] +
matrix[i][j];
    }

    int sumRegion(int row1, int col1, int row2, int col2) {
        int Ai = row1, Aj = col1;
        int Bi = row2+1, Bj = col2+1;
        return preSum[Bi][Bj] + preSum[Ai][Aj] - preSum[Ai][Bj] - preSum[Bi][Aj];
    }
};
```

5、差分数组

> 差分数组的主要适用场景是频繁对原始数组的某个区间的元素进行增减。

1. 区间加法
   做法：构建 Difference 即可
   - 初始化
   - 区间变化
   - 返回结果
2. 航班预定统计
   做法：用所有航班号作为 Difference 的 nums 即可，最后返回结果
3. 拼车
   做法：用所有站点号作为 Difference 的 nums 即可，最后检测在各个站点时的车人数都是不超载的。

> 问题：
>
> 1. 第一个问题，想要使用查分数组技巧，必须创建一个长度和区间长度一样的差分数组 diff。

2. 第二个问题，前缀和技巧可以快速进行区间查询，查分数组可以快速进行区间增减。能不能把他俩结合起来，既可以快速进行区间增减，又可以随时进行区间查询?
这两个问题是处理区间问题的常见问题。
终极答案是 **线段树** 这种数据结构，它可以在 $O(\log N)$ 的时间复杂度内完成任意长度的区间增减和区间查询操作

```
T1:
class Solution {
public:
    vector<int> getModifiedArray(int length, vector<vector<int>>& updates) {
        // nums 初始化为全 0
        vector<int> nums(length, 0);
        // 构造差分解法
        Difference df(nums);
        for (const auto& update : updates) {
            int i = update[0];
            int j = update[1];
            int val = update[2];
            df.increment(i, j, val);
        }
        return df.result();
    }

    class Difference {
        // 差分数组
        vector<int> diff;

    public:
        Difference(const vector<int>& nums) {
            assert(!nums.empty());
            diff.resize(nums.size());
            // 构造差分数组
            diff[0] = nums[0];
            for (size_t i = 1; i < nums.size(); ++i) {
                diff[i] = nums[i] - nums[i - 1];
            }
        }

        // 给闭区间 [i, j] 增加 val（可以是负数）
        void increment(int i, int j, int val) {
            diff[i] += val;
            if (j + 1 < diff.size()) {
                diff[j + 1] -= val;
            }
        }

        vector<int> result() {
            vector<int> res(diff.size());
            // 根据差分数组构造结果数组
            res[0] = diff[0];
            for (size_t i = 1; i < diff.size(); ++i) {
```

```
                res[i] = res[i - 1] + diff[i];
            }
            return res;
        }
    };
};

T2:
class Solution {
public:
    vector<int> corpFlightBookings(vector<vector<int>>& bookings, int n) {
        // 用一个差分数组来做后续操作
        vector<int> nums(n, 0);
        Difference df(nums);
        for(auto booking : bookings)
            df.add(booking[0]-1, booking[1]-1, booking[2]);
        return df.result();
    }

private:
    // Class: 差分数组
    class Difference {
            vector<int> diff;
        public:
            // 初始化构造差分数组
            Difference(const vector<int> &nums){
                diff.resize(nums.size());
                diff[0] = nums[0];
                for(int i=1; i<nums.size(); i++)
                    diff[i] = nums[i] - nums[i-1];
            }

            // 对区间 [i,j] 都加 val
            void add(int i, int j, int val){
                diff[i] += val;
                if(j+1 < diff.size())  diff[j+1] -= val;
            }

            // 返回完整数组
            vector<int> result(){
                vector<int> ans(diff.size(), 0);
                ans[0] = diff[0];
                for(int i=1; i<diff.size(); i++)
                    ans[i] = ans[i-1] + diff[i];
                return ans;
            }
    };

};


T3:
```

```cpp
class Solution {
public:
    bool carPooling(vector<vector<int>>& trips, int capacity) {
        // 差分数组 nums[i] 表示在站点 i 时，车上应该多少人
        vector<int> nums(1001, 0);  // 站点有 0 - 1000
        Difference df(nums);

        // 所有的中途上下站
        for(auto &trip : trips)
            df.add(trip[1], trip[2]-1, trip[0]); // 站点编号 0 开始

        // 检查所有站点，都是可以完成的
        auto res = df.result();
        for(int i=0; i<res.size(); i++)
            if(res[i] > capacity)
                return false;
        return true;
    }

private:
    class Difference {
        vector<int> diff;
        public:
            Difference(const vector<int> &nums){
                diff.resize(nums.size());
                diff[0] = nums[0];
                for(int i=1; i<nums.size(); i++)
                    diff[i] = nums[i] - nums[i-1];
            }
            void add(int i, int j, int val){
                diff[i] += val;
                if(j+1 < diff.size())  diff[j+1] -= val;
            }
            vector<int> result(){
                vector<int> ans(diff.size());
                ans[0] = diff[0];
                for(int i=1; i<diff.size(); i++)
                    ans[i] = ans[i-1] + diff[i];
                return ans;
            }
    };
};

```

6、滑动窗口延伸：Rabin Karp 字符匹配算法

下面是滑动窗口的框架：

```cpp
// 滑动窗口算法伪码框架
void slidingWindow(string s) {
    // 用合适的数据结构记录窗口中的数据，根据具体场景变通
```

```
    // 比如说，我想记录窗口中元素出现的次数，就用 map
    // 如果我想记录窗口中的元素和，就可以只用一个 int
    auto window = ...

    int left = 0, right = 0;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        window.add(c);
        // 增大窗口
        right++;

        // 进行窗口内数据的一系列更新
        ...

        // *** debug 输出的位置 ***
        printf("window: [%d, %d)\n", left, right);
        // 注意在最终的解法代码中不要 print
        // 因为 IO 操作很耗时，可能导致超时

        // 判断左侧窗口是否要收缩
        while (window needs shrink) {
            // d 是将移出窗口的字符
            char d = s[left];
            window.remove(d);
            // 缩小窗口
            left++;

            // 进行窗口内数据的一系列更新
            ...
        }
    }
}
```

先看，你是如何：

1. 在最低位添加一个数字
2. 在最高位删除一个数字

```
// ****** 在最低位添加一个数字 ******
int number = 8264;
// number 的进制
int R = 10;
// 想在 number 的最低位添加的数字
int appendVal = 3;
// 运算，在最低位添加一位
number = R * number + appendVal;
// 此时 number = 82643

// ****** 在最高位删除一个数字 ******
int number = 8264;
```

```
// number 的进制
int R = 10;
// number 最高位的数字
int removeVal = 8;
// 此时 number 的位数
int L = 4;
// 运算，删除最高位数字
number = number - removeVal * Math.pow(R, L-1);
// 此时 number = 264
```

1. 寻找重复子序列
   方法一：直接暴力看所有长度 10 的串是否出现过。（哈希集合）
   方法二：改为通过一个数字就能直接判断是否出现过。（仍然哈希集合，但是避免了每一次都 substr，而只需要在添加答案时使用substr）
2. Rabin-Karp 字符串匹配
   那么借鉴上面的思路，我们不要每次都去一个字符一个字符地比较子串和模式串，而是维护一个滑动窗口，运用滑动哈希算法一边滑动一边计算窗口中字符串的哈希值，拿这个哈希值去和模式串的哈希值比较，这样就可以避免截取子串，从而把匹配算法降低为 $O(N)$，这就是 Rabin-Karp 指纹字符串查找算法的核心逻辑。
   1. 进制采用 R = 256（因为编码字符 ASCII 共 256 个）
   2. 长度看模式串的长度 L = pat.size()
   3. 为解决数字过大，采用取余（模一个大素数1658598167）
   4. 为解决哈希冲突，要进一步直接对比字符串

```
T1-1:
class Solution {
public:
    // 暴力法：哈希集合
    vector<string> findRepeatedDnaSequences(string s) {
        unordered_set<string> seen;
        unordered_set<string> ans;
        for(int i=0; i+9<s.size(); i++){
            string ts = s.substr(i, 10);
            if(seen.find(ts) != seen.end())
                ans.insert(ts);
            else
                seen.insert(ts);
        }
        // 转回数组
        return vector<string>(ans.begin(), ans.end());
    }
};

T1-2:
class Solution {
public:
    // 关键在于可以优化掉不需要取 L=10 的子串。
    vector<string> findRepeatedDnaSequences(string s) {
        // 转为数字
```

```cpp
        vector<int> nums = s2Nums(s);

        unordered_set<int> seen;
        unordered_set<string> ans;

        int R = 4, L = 10;
        int RL = pow(R, L-1);
        int l = 0, r = 0;
        int cur = 0;
        while(r < nums.size()){
            cur = cur*R + nums[r];
            r++;

            if(r-l == L){
                if(seen.find(cur) != seen.end())
                    ans.insert(s.substr(l, L));
                else
                    seen.insert(cur);
                cur = cur - nums[l]*RL;
                l++;
            }
        }

        return vector<string>(ans.begin(), ans.end());
    }
    // 用 4 进制表示 ACGT 即可唯一标识了
    vector<int> s2Nums(const string &s){
        vector<int> nums(s.size());
        for(int i=0; i<s.size(); i++){
            switch(s[i]){
                case 'A':
                    nums[i] = 0;
                    break;
                case 'C':
                    nums[i] = 1;
                    break;
                case 'G':
                    nums[i] = 2;
                    break;
                case 'T':
                    nums[i] = 3;
                    break;
            }
        }
        return nums;
    }
};


T2:
// Rabin-Karp 指纹字符串查找算法
int rabinKarp(string txt, string pat) {
```

```
    // 位数
    int L = pat.length();
    // 进制（只考虑 ASCII 编码）
    int R = 256;
    // 取一个比较大的素数作为求模的除数
    long Q = 1658598167;
    // R^(L - 1) 的结果
    long RL = 1;
    for (int i = 1; i <= L - 1; i++) {
        // 计算过程中不断求模，避免溢出
        RL = (RL * R) % Q;
    }
    // 计算模式串的哈希值，时间 O(L)
    long patHash = 0;
    for (int i = 0; i < pat.length(); i++) {
        patHash = (R * patHash + pat.at(i)) % Q;
    }

    // 滑动窗口中子字符串的哈希值
    long windowHash = 0;

    // 滑动窗口代码框架，时间 O(N)
    int left = 0, right = 0;
    while (right < txt.length()) {
        // 扩大窗口，移入字符
        windowHash = ((R * windowHash) % Q + txt.at(right)) % Q;
        right++;

        // 当子串的长度达到要求
        if (right - left == L) {
            // 根据哈希值判断是否匹配模式串
            if (windowHash == patHash) {
                // 当前窗口中的子串哈希值等于模式串的哈希值
                // 还需进一步确认窗口子串是否真的和模式串相同，避免哈希冲突
                if (pat.compare(txt.substr(left, L)) == 0) {
                    return left;
                }
            }
            // 缩小窗口，移出字符
            windowHash = (windowHash - (txt.at(left) * RL) % Q + Q) % Q;
            // X % Q == (X + Q) % Q 是一个模运算法则
            // 因为 windowHash - (txt[left] * RL) % Q 可能是负数
            // 所以额外再加一个 Q，保证 windowHash 不会是负数

            left++;
        }
    }
    // 没有找到模式串
    return -1;
}
```

7、二分搜索思维

> 注意：下面都采用 "**左闭右开**" 的区间。
> 搜索边界

```cpp
// 搜索左侧边界
int left_bound(vector<int>& nums, int target) {
    if (nums.size() == 0) return -1;
    int left = 0, right = nums.size();

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            // 当找到 target 时，收缩右侧边界
            right = mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid;
        }
    }
    return left;
}
```

```cpp

// 搜索右侧边界
int right_bound(vector<int>& nums, int target) {
    if (nums.size() == 0) return -1;
    int left = 0, right = nums.size();

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            // 当找到 target 时，收缩左侧边界
            left = mid + 1;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid;
        }
    }
    return left - 1;
}
```

下面看如何把二分搜索进行泛化：
首先，你要从题目中抽象出一个自变量 `x`，一个关于 `x` 的函数 `f(x)`，以及一个目标值 `target`。同时，`x`，`f(x)`，`target` 还要满足以下条件：

1. `f(x)` 必须是在 `x` 上的单调函数（单调增单调减都可以）。

2. 题目是让你计算满足约束条件 `f(x) == target` 时的 `x` 的值。
   那么，前面的二分搜索实际上就是：`int f(int x, vector<int>& nums) { return nums[x]; }`
   如果遇到一个算法问题，能够把它抽象成上面的图，就可以对它运用二分搜索算法。

```
// 函数 f 是关于自变量 x 的单调递增函数
int f(int x, int nums[]) {
    return nums[x];
}

int left_bound(int nums[], int length, int target) {
    if (length == 0) return -1;
    int left = 0, right = length;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (f(mid, nums) == target) {
            // 当找到 target 时，收缩右侧边界
            right = mid;
        } else if (f(mid, nums) < target) {
            left = mid + 1;
        } else if (f(mid, nums) > target) {
            right = mid;
        }
    }
    return left;
}
```

下面就是做题：

1. 875. 爱吃香蕉的珂珂
   做法：
   - 抽象出 y = f(k) 表示以速度 k 吃所需的总时间，注意 f 单调递减。
   - 二分查找出左边界即可。
2. 1011. 在 D 天内送达包裹的能力
   做法：
   - 抽象出 y = f(cap) 表示运载能力 cap 下的所需天数，注意 f 单调递减。
   - 二分查找左边界。
3. 410. 分割数组的最大值
   做法：
   - 抽象出 y = f(sum) 表示当最大和不能超过 sum 时，所分成的子数组个数。注意 f 单调递减。
   - 二分查找左边界。

> 总结来说，如果发现题目中存在**单调关系**，就可以尝试使用**二分搜索**的思路来解决。搞清楚单调性和二分搜索的种类，通过分析和画图，就能够写出最终的代码。

全部代码：

```
T1:
class Solution {
public:
    // 本质上，可以抽象成二分搜索的函数图，那么就能使用二分搜索，找左边界即可
    int minEatingSpeed(vector<int>& piles, int h) {
        int maxK = *max_element(piles.begin(), piles.end());
        int l = 1, r = maxK+1;   // [l,r)
        while(l < r){
            int mid = l + (r-l)/2;
            int y = f(mid, piles);
            if(y == h)
                r = mid;
            else if(y < h)
                r = mid;
            else if(y > h)
                l = mid+1;
        }
        // 左边界
        return l;
    }
    // 返回：以 k 速度吃完所需的时间
    // f 是单调递减
    int f(int k, const vector<int>& piles){
        int sum = 0;
        for(int num : piles)
            sum += num/k + (num%k != 0);
        return sum;
    }
};

T2:
class Solution {
public:
    // 最小值问题：二分查找左边界
    int shipWithinDays(vector<int>& weights, int days) {
        int maxCap = accumulate(weights.begin(), weights.end(), 0);
        int l = 1, r = maxCap+1;    // [l, r)
        while(l < r){
            int mid = l + (r-l)/2;
            int y = f(mid, weights);
            if(y == days)
                r = mid;
            else if(y > days)
                l = mid+1;
            else if(y < days)
                r = mid;
        }
        // 返回左边界
        return l;
    }
    // 函数值：以 cap 运载能力时所需要的天数（f单调递减）
    int f(int cap, const vector<int>& weights){
```

```
        int cnt = 0, sum = 0;
        for(const int &w : weights){
            if(w > cap)  return INT_MAX;     // 注意判断

            if(sum+w <= cap)  sum += w;
            else  cnt++, sum = w;
        }
        if(sum)  cnt++;
        return cnt;
    }

};

T3:
class Solution {
public:
    // 抽象出二分查找左边界
    int splitArray(vector<int>& nums, int k) {
        int maxSum = accumulate(nums.begin(), nums.end(), 0);
        int l = 0, r = maxSum+1;    // [l, r)
        while(l < r){
            int mid = l + (r-l)/2;
            int y = f(mid, nums);
            if(y == k)
                r = mid;
            else if(y < k)
                r = mid;
            else if(y > k)
                l = mid+1;
        }
        // 左边界
        return l;
    }
    // 函数值：一个子数组和最大能为 sum 时，可以分成几个块（单调递减）
    int f(int sum, const vector<int>& nums){
        int cnt = 0, curSum = 0;
        for(const int &num : nums){
            if(num > sum)  return INT_MAX;  // 注意判断

            if(curSum + num <= sum)
                curSum += num;
            else
                cnt++, curSum = num;
        }
        if(curSum)  cnt++;
        return cnt;
    }
};
```

8、带权随机选择算法

- 参考528. 按权重随机选择

1. 根据权重数组 w 生成前缀和数组 preSum。
2. 生成一个取值在 preSum 之内的随机数，用二分搜索算法寻找大于等于这个随机数的最小元素索引。
3. 最后对这个索引减一（因为前缀和数组有一位索引偏移），就可以作为权重数组的索引，即最终答案:

注意几个细节：

- 丢出的随机数 target 应该是范围 `[1, lastPreSum]`
- 二分搜索大于等于的左边界

```cpp
class Solution {
private:
    vector<int> preSum;
    mt19937 gen;
    uniform_int_distribution<> dis;

public:
    Solution(vector<int>& w) {
        int n = w.size();
        preSum.resize(n+1);
        preSum[0] = 0;
        for(int i=0; i<n; i++)
            preSum[i+1] = preSum[i] + w[i];
        // 随机分布的整数
        dis = uniform_int_distribution<>(1, preSum[n]);
    }

    int pickIndex() {
        int target = dis(gen);
        return left_bound(target) - 1;
    }

    int left_bound(int target){
        int l = 0, r = preSum.size();   // [l, r)
        while(l < r){
            int mid = l + (r-l)/2;
            int y = preSum[mid];
            if(y == target)
                r = mid;
            else if(y > target)
                r = mid;
            else if(y < target)
                l = mid+1;
        }
        return l;
    }
};
```

9、田忌赛马决策算法

- 参考：870. 优势洗牌
  这就像田忌赛马的情景，nums1 就是田忌的马，nums2 就是齐王的马，数组中的元素就是马的战斗力。

注意：节约的策略是没问题的，但是没有必要。这也是本题有趣的地方，需要绕个脑筋急转弯：

- 我们暂且把田忌的一号选手称为 `T1`，二号选手称为 `T2`，齐王的一号选手称为 `Q1`。
- 如果 `T2` 能赢 `Q1`，你试图保存己方实力，让 `T2` 去战 `Q1`，把 `T1` 留着是为了对付谁？
- 显然，你担心齐王还有战力大于 `T2` 的马，可以让 `T1` 去对付。
- 但是你仔细想想，现在 `T2` 已经是可以战胜 `Q1` 的，`Q1` 可是齐王的最快的马耶，齐王剩下的那些马里，怎么可能还有比 `T2` 更强的马？
- 所以，没必要节约，最后我们得出的策略就是：
  将齐王和田忌的马按照战斗力排序，然后按照排名一一对比。如果田忌的马能赢，那就比赛，如果赢不了，那就换个垫底的来送人头，保存实力。

```cpp
class Solution {
public:
    vector<int> advantageCount(vector<int>& nums1, vector<int>& nums2) {
        int n = nums1.size();
        // 由于需要和 nums2 的每一个比，nums2 记住下标
        vector<pair<int, int>> pairs2;
        for(int i=0; i<n; i++)  pairs2.push_back({nums2[i], i});
        // 从大到小排序
        sort(pairs2.begin(), pairs2.end(), greater<pair<int, int>>());
        sort(nums1.begin(), nums1.end(), greater<int>());

        vector<int> ans(n);
        int l = 0, r = n-1; // 左右指针控制当前 nums1 的情况
        for(auto &p2 : pairs2){
            int num2 = p2.first;
            int i = p2.second;
            // 打得过就直接打
            if(nums1[l] > num2)
                ans[i] = nums1[l], l++;
            // 打不过，用最菜的上来
            else
                ans[i] = nums1[r], r--;
        }
        return ans;
    }
};
```

(三) 队列、栈

1、队列与栈的互相实现

1. 用栈实现队列
   做法：使用两个栈S1、S2就能实现，S1 作为队尾，S2 作为队头，push向 S1 进，pop取 S2 的，S2 空时取出全部 S1。
   均摊时间复杂度：$O(1)$（因为每个元素最多搬运一次）
2. 用队列实现栈
   做法：比较简单暴力，主要需要取栈顶时，把所有元素重新排队，直到队尾的元素已经到队头了，就可以了。
   时间复杂度：pop操作时是$O(N)$，其他操作$O(1)$。

> 本质上是需要实现两个功能就行，`top/front`和`pop`就行，因为只有这两个操作是对外显示状态的。

栈实现队列比较巧妙，下图只给出这个的图解和代码：

```cpp
#include <stack>

class MyQueue {
private:
    std::stack<int> s1, s2;

public:
    MyQueue() {
    }

    // 添加元素到队尾
    void push(int x) {
        s1.push(x);
    }

    // 返回队头元素
    int peek() {
        if (s2.empty())
            // 把 s1 元素压入 s2
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        return s2.top();
    }

    // 删除队头元素并返回
    int pop() {
        // 先调用 peek 保证 s2 非空
        peek();
        int poppedValue = s2.top();
        s2.pop();
        return poppedValue;
    }

     // 判断队列是否为空
     // 两个栈都为空才说明队列为空
```

```
    bool empty() {
        return s1.empty() && s2.empty();
    }
};
```

2、单调栈

栈（stack）是很简单的一种数据结构，先进后出的逻辑顺序，符合某些问题的特点，比如说函数调用栈。单调栈实际上就是栈，只是利用了一些巧妙的逻辑，使得每次新元素入栈后，栈内的元素都保持有序（单调递增或单调递减）。

听起来有点像堆（heap）？不是的，单调栈用途不太广泛，只处理一类典型的问题，比如「下一个更大元素」，「上一个更小元素」等。

1. 模版

- 输入一个数组 `nums`，请你返回一个等长的结果数组，结果数组中对应索引存储着下一个更大元素，如果没有更大的元素，就存 `-1`。
- 输入一个数组 `nums = [2,1,2,4,3]`
- 你返回数组 `[4,2,4,-1,-1]`

- 抽象思考：把数组的元素想象成并列站立的人，元素大小想象成人的身高。
  下面是代码，这个做法比直接简单粗暴的二次循环要更快，时间复杂度是 $O(n)$。

```
vector<int> calculateGreaterElement(vector<int>& nums) {
    int n = nums.size();
    vector<int> res(n);
    stack<int> S;
    // 倒着进
    for(int i=n-1; i>=0; i--){
        // 保持单调栈是递减的(栈中比当前数字还矮的出去)
        while(!S.empty() && nums[i] >= S.top())
            S.pop();
        // 目前的栈顶就是第一个比 cur 大的
        res[i] = S.empty()? -1 : S.top();
        S.push(nums[i]);
    }
    return res;
}
```

2. 496. 下一个更大元素 I
   做法：
   - 直接把 `nums2` 的全部元素都完成单调栈，存储答案在哈希表
   - 然后根据 `nums1` 去查表即可。
3. 739. 每日温度
   做法：
   - 仍然是单调栈即可

- 注意存储的是下标（因为答案是问几天之后）

4. 环形数组：503. 下一个更大元素 II
   比如输入 `[2,1,2,4,3]` 你应该返回 `[4,2,4,-1,4]`，因为拥有了环形属性，最后一个元素 3 绕了一圈后找到了比自己大的元素 4。
   做法：把数组翻倍即可。（实际上，你可以不用真的翻倍，而是采用取模来实现即可）

```
T2:
class Solution {
public:
    // 单调栈 + 哈希表
    // 直接把 nums2 全部完成，然后 nums1 去查表即可
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
        // 完成 nums2
        int n2 = nums2.size();
        unordered_map<int, int> TB;
        stack<int> S;
        for(int i=n2-1; i>=0; i--){
            while(!S.empty() && nums2[i] >= S.top())
                S.pop();
            TB[nums2[i]] = S.empty() ? -1 : S.top();
            S.push(nums2[i]);
        }
        // nums1 去查表
        int n1 = nums1.size();
        vector<int> ans(n1);
        for(int i=0; i<n1; i++)
            ans[i] = TB[nums1[i]];
        // 完成
        return ans;
    }
};

T3:
class Solution {
public:
    // 单调栈
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        int n = temperatures.size();
        stack<int> S;
        vector<int> ans(n);
        for(int i=n-1; i>=0; i--){
            while(!S.empty() && temperatures[i] >= temperatures[S.top()])
                S.pop();
            ans[i] = S.empty() ? 0 : S.top()-i;
            S.push(i);  // 注意是存储下标
        }
        return ans;
    }
};

```

```
T4:
class Solution {
public:
    // 单调栈 + 数组翻倍（取模技巧）
    vector<int> nextGreaterElements(vector<int>& nums) {
        int n = nums.size();
        stack<int> S;
        vector<int> ans(n);
        for(int i=2*n-1; i>=0; i--){
            while(!S.empty() && nums[i%n] >= S.top())
                S.pop();
            ans[i%n] = S.empty() ? -1 : S.top();
            S.push(nums[i%n]);
        }
        return ans;
    }
};
```

3、单调队列

- 给你一个数组 `window`，已知其最值为 `A`，如果给 `window` 中添加一个数 `B`，那么比较一下 `A` 和 `B` 就可以立即算出新的最值；但如果要从 `window` 数组中减少一个数，就不能直接得到最值了，因为如果减少的这个数恰好是 `A`，就需要遍历 `window` 中的所有元素重新寻找新的最值。
- 这个场景很常见，但不用单调队列似乎也可以，比如 `优先级队列（二叉堆）` 就是专门用来动态寻找最值的，我创建一个大（小）顶堆，不就可以很快拿到最大（小）值了吗？如果单纯地维护最值的话，优先级队列很专业，队头元素就是最值。但优先级队列无法满足标准队列结构「**先进先出**」的时间顺序，因为优先级队列底层利用二叉堆对元素进行动态排序，元素的出队顺序是元素的大小顺序，和入队的先后顺序完全没有关系。
- 现在需要一种新的队列结构，既能够维护队列元素「**先进先出**」的时间顺序，又能够正确维护队列中所有元素的**最值**，这就是「单调队列」结构。
- 「单调队列」这个数据结构主要用来辅助解决**滑动窗口**相关的问题。
- 前文 滑动窗口核心框架 把滑动窗口算法作为双指针技巧的一部分进行了讲解，但有些稍微复杂的滑动窗口问题不能只靠两个指针来解决，需要上更先进的数据结构。
- 参考：239. 滑动窗口最大值

```
输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

核心：实现单调队列

- 「单调队列」的核心思路和「单调栈」类似，push 方法依然在队尾添加元素，但是要把前面比自己小的元素都删掉。
- 可以想象，加入数字的大小代表人的体重，体重大的会把前面体重不足的压扁，直到遇到更大的量级才停住。

- 注意，虽然说下面的 push 有 while 循环，但是时间复杂度是常数级，因为每一个元素只会操作一次。整个代码全部执行完，也才只有每个元素操作一次，即整个解法的时间复杂度是 $O(n)$。
- 那么push 时间复杂度是 $O(k)$，k 是窗口大小。

```cpp
class MonoQueue {
private:
    deque<int> DQ;
public:
    // 保证单调队列
    void push(int num){
        // 这里没有等于，思考：因为pop是按值pop的，相同的要保留
        while(!DQ.empty() && num > DQ.back())
            DQ.pop_back();
        DQ.push_back(num);
    }
    // 注意，不是都 pop 的，最大值才pop
    void pop(int num){
        if(num == DQ.front())
            DQ.pop_front();
    }
    // 队头就是当前队列的最大值
    int max(){
        return DQ.front();
    }
};

class Solution {
public:
    // 滑动窗口 + 单调队列
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        int n = nums.size();
        MonoQueue MQ;
        vector<int> ans;
        // 先压 k-1 个
        for(int i=0; i<=k-2; i++)  MQ.push(nums[i]);
        // 一个一个压
        for(int i=k-1; i<n; i++){
            MQ.push(nums[i]);
            ans.push_back(MQ.max());
            MQ.pop(nums[i-k+1]);
        }
        return ans;
    }
};

```

> 拓展延伸
> 最后，提出几个问题请大家思考：

1. 本文给出的 `MonotonicQueue` 类只实现了 `max` 方法，你是否能够再额外添加一个 `min` 方法，在 $O(1)$ 的时间返回队列中所有元素的最小值？
2. 本文给出的 `MonotonicQueue` 类的 `pop` 方法还需要接收一个参数，这不那么优雅，而且有悖于标准队列的 API，请你修复这个缺陷。
3. 请你实现 `MonotonicQueue` 类的 `size` 方法，返回单调队列中元素的个数（注意，由于每次 `push` 方法都可能从底层的 `q` 列表中删除元素，所以 `q` 中的元素个数并不是单调队列的元素个数）。

```cpp
// 单调队列的通用实现，可以高效维护最大值和最小值
template <typename E>
class MonotonicQueue {
public:
    // 标准队列 API，向队尾加入元素
    void push(E elem);

    // 标准队列 API，从队头弹出元素，符合先进先出的顺序
    E pop();

    // 标准队列 API，返回队列中的元素个数
    int size();

    // 单调队列特有 API，O(1) 时间计算队列中元素的最大值
    E max();

    // 单调队列特有 API，O(1) 时间计算队列中元素的最小值
    E min();
};
```

(四) 二叉树

待完成...

(五) 二叉树强化

待完成...

(六) 二叉树拓展

待完成...

(七) 经典数据结构设计

1、LRU 算法

- LRU 缓存淘汰算法就是一种常用策略。LRU 的全称是 Least Recently Used，也就是说我们认为最近使用过的数据应该是是「有用的」，很久都没用过的数据应该是无用的，内存满了就优先删那些很久没用过的数据。
- 参考：146. LRU 缓存

- 哈希表查找快，但是数据无固定顺序；链表有顺序之分，插入删除快，但是查找慢。所以结合一下，形成一种新的数据结构：哈希链表 LinkedHashMap。
- LRU 缓存算法的核心数据结构就是哈希链表，双向链表和哈希表的结合体。

```cpp
// 双向链表节点
class Node {
public:
    int key, val;
    Node* next, *prev;
    Node(int k, int v) : key(k), val(v), next(nullptr), prev(nullptr) {}
};

// 双向链表
class DoubleList {
private:
    // 头尾虚节点
    Node* head;
    Node* tail;
    // 链表元素数
    int size;

public:
    DoubleList() {
        // 初始化双向链表的数据
        head = new Node(0, 0);
        tail = new Node(0, 0);
        head->next = tail;
        tail->prev = head;
        size = 0;
    }

    // 在链表尾部添加节点 x，时间 O(1)
    void addLast(Node* x) {
        x->prev = tail->prev;
        x->next = tail;
        tail->prev->next = x;
        tail->prev = x;
        size++;
    }

    // 删除链表中的 x 节点（x 一定存在）
    // 由于是双链表且给的是目标 Node 节点，时间 O(1)
    void remove(Node* x) {
        x->prev->next = x->next;
        x->next->prev = x->prev;
        size--;
    }

    // 删除链表中第一个节点，并返回该节点，时间 O(1)
    Node* removeFirst() {
```

```
        if (head->next == tail)
            return nullptr;
        Node* first = head->next;
        remove(first);
        return first;
    }

    // 返回链表长度，时间 O(1)
    int getSize() { return size; }
};

class LRUCache {
private:
    // key -> Node(key, val)
    unordered_map<int, Node*> map;
    // Node(k1, v1) <-> Node(k2, v2)...
    DoubleList cache;
    // 最大容量
    int cap;

public:
    LRUCache(int capacity) {
        this->cap = capacity;
    }

    int get(int key) {
        if (!map.count(key)) {
            return -1;
        }
        // 将该数据提升为最近使用的
        makeRecently(key);
        return map[key]->val;
    }

    void put(int key, int val) {
        if (map.count(key)) {
            // 删除旧的数据
            deleteKey(key);
            // 新插入的数据为最近使用的数据
            addRecently(key, val);
            return;
        }

        if (cap == cache.getSize()) {
            // 删除最久未使用的元素
            removeLeastRecently();
        }
        // 添加为最近使用的元素
        addRecently(key, val);
    }

    void makeRecently(int key) {
```

```
        Node* x = map[key];
        // 先从链表中删除这个节点
        cache.remove(x);
        // 重新插到队尾
        cache.addLast(x);
    }

    void addRecently(int key, int val) {
        Node* x = new Node(key, val);
        // 链表尾部就是最近使用的元素
        cache.addLast(x);
        // 别忘了在 map 中添加 key 的映射
        map[key] = x;
    }

    void deleteKey(int key) {
        Node* x = map[key];
        // 从链表中删除
        cache.remove(x);
        // 从 map 中删除
        map.erase(key);
    }

    void removeLeastRecently() {
        // 链表头部的第一个元素就是最久未使用的
        Node* deletedNode = cache.removeFirst();
        // 同时别忘了从 map 中删除它的 key
        int deletedKey = deletedNode->key;
        map.erase(deletedKey);
    }
};
```

2、LFU 算法

- LFU 算法的淘汰策略是 Least Frequently Used，也就是每次淘汰那些使用次数最少的数据。
- 从实现难度上来说，LFU 算法的难度大于 LRU 算法。
- LFU 算法相当于是把数据按照访问频次进行排序，这个需求恐怕没有那么简单，而且还有一种情况，如果多个数据拥有相同的访问频次，我们就得删除最早插入的那个数据。也就是说 LFU 算法是淘汰访问频次最低的数据，如果访问频次最低的数据有多条，需要淘汰最旧的数据。
- 参考类似题目：895. 最大频率栈

解决方案：

1. 使用一个 `HashMap` 存储 `key` 到 `val` 的映射，就可以快速计算 `get(key)`。

```
unordered_map<int, int> keyToVal;
```

2. 使用一个 `HashMap` 存储 `key` 到 `freq` 的映射，就可以快速操作 `key` 对应的 `freq`。

```
unordered_map<int, int> keyToFreq;
```

3. 如果在容量满了的时候进行插入，则需要将 `freq` 最小的 `key` 删除，如果最小的 `freq` 对应多个 `key`，则删除其中最旧的那一个。
    - 需要 `freq` 到 `key` 的映射，用来找到 `freq` 最小的 `key`。
    - `freq` 对 `key` 是一对多的关系，即一个 `freq` 对应一个 `key` 的列表。
    - 用一个变量 `minFreq` 来记录当前最小的 `freq`。
    - 希望 `freq` 对应的 `key` 的列表是存在时序的。
    - 希望能够快速删除 `key` 列表中的任何一个 `key`，因为如果频次为 `freq` 的某个 `key` 被访问，那么它的频次就会变成 `freq+1`，就应该从 `freq` 对应的 `key` 列表中删除，加到 `freq+1` 对应的 `key` 的列表中。

```
unordered_map<int, unordered_set<int>> freqToKeys;
int minFreq = 0;
```

完整代码：
代码比较难写，请记住需要三个关键的表 KV、KF、FK 表，面试需要能说得出口。

```
class LFUCache {
    // key 到 val 的映射，我们后文称为 KV 表
    std::unordered_map<int, int> keyToVal;
    // key 到 freq 的映射，我们后文称为 KF 表
    std::unordered_map<int, int> keyToFreq;
    // freq 到 key 列表的映射，我们后文称为 FK 表
    std::unordered_map<int, std::list<int>> freqToKeys;
    // 记录最小的频次
    int minFreq;
    // 记录 LFU 缓存的最大容量
    int cap;

public:
    LFUCache(int capacity) {
        this->cap = capacity;
        this->minFreq = 0;
    }

    int get(int key) {
        if (keyToVal.find(key) == keyToVal.end()) {
            return -1;
        }
        // 增加 key 对应的 freq
        increaseFreq(key);
        return keyToVal[key];
    }

    void put(int key, int val) {
        if (this->cap <= 0) return;
```

```
30
31          // 若 key 已存在，修改对应的 val 即可
32          if (keyToVal.find(key) != keyToVal.end()) {
33              keyToVal[key] = val;
34              // key 对应的 freq 加一
35              increaseFreq(key);
36              return;
37          }
38
39          // key 不存在，需要插入
40          // 容量已满的话需要淘汰一个 freq 最小的 key
41          if (keyToVal.size() >= this->cap) {
42              removeMinFreqKey();
43          }
44
45          // 插入 key 和 val，对应的 freq 为 1
46          // 插入 KV 表
47          keyToVal[key] = val;
48          // 插入 KF 表
49          keyToFreq[key] = 1;
50          // 插入 FK 表
51          freqToKeys[1].push_back(key);
52          // 插入新 key 后最小的 freq 肯定是 1
53          this->minFreq = 1;
54      }
55
56  private:
57      void removeMinFreqKey() {
58          // freq 最小的 key 列表
59          auto& keyList = freqToKeys[this->minFreq];
60          // 其中最先被插入的那个 key 就是该被淘汰的 key
61          int deletedKey = keyList.front();
62          keyList.pop_front();
63          if (keyList.empty()) {
64              freqToKeys.erase(this->minFreq);
65              // 如果 freqToKeys[minFreq] 为空，需要更新 minFreq
66          }
67          // 删除 KV 表中的该 key
68          keyToVal.erase(deletedKey);
69          // 删除 KF 表中的该 key
70          keyToFreq.erase(deletedKey);
71      }
72
73      void increaseFreq(int key) {
74          int freq = keyToFreq[key];
75          // 更新 KF 表
76          keyToFreq[key] = freq + 1;
77          // 更新 FK 表
78          // 将 key 从 freq 对应的列表中删除
79          auto& oldList = freqToKeys[freq];
80          oldList.remove(key);
81          // 将 key 加入 freq + 1 对应的列表中
```

```
        freqToKeys[freq + 1].push_back(key);
        // 如果 freq 对应的列表空了，移除这个 freq
        if (oldList.empty()) {
            freqToKeys.erase(freq);
            // 如果这个 freq 恰好是 minFreq，更新 minFreq
            if (freq == this->minFreq) {
                this->minFreq++;
            }
        }
    }
};
```

3、随机集合、黑名单随机数

- 参考：380. O(1) 时间插入、删除和获取随机元素

```
class RandomizedSet {
private:
    // 存储元素的值
    vector<int> nums;
    // 记录每个元素对应在 nums 中的索引
    unordered_map<int, int> valToIndex;

public:
    RandomizedSet() {
        srand((unsigned)time(NULL));
    }

    bool insert(int val) {
        // 若 val 已存在，不用再插入
        if (valToIndex.count(val)) {
            return false;
        }
        // 若 val 不存在，插入到 nums 尾部，
        // 并记录 val 对应的索引值
        valToIndex[val] = nums.size();
        nums.push_back(val);
        return true;
    }

    bool remove(int val) {
        // 若 val 不存在，不用再删除
        if (!valToIndex.count(val)) {
            return false;
        }
        // 先拿到 val 的索引
        int index = valToIndex[val];
        // 将最后一个元素对应的索引修改为 index
        int lastElement = nums.back();
        valToIndex[lastElement] = index;
        // 交换 val 和最后一个元素
```

```
        swap(nums[index], nums.back());
        // 在数组中删除元素 val
        nums.pop_back();
        // 删除元素 val 对应的索引
        valToIndex.erase(val);
        return true;
    }

    int getRandom() {
        // 随机获取 nums 中的一个元素
        int randomIndex = rand() % nums.size();
        return nums[randomIndex];
    }
};
```

- 参考：710. 黑名单中的随机数

```
class Solution {
private:
    int sz;
    unordered_map<int, int> mapping;

public:
    Solution(int N, vector<int>& blacklist) {
        sz = N - blacklist.size();

        // 转化成集合，方便快速查询
        unordered_set<int> blackSet;
        for(int& b : blacklist){
            blackSet.insert(b);
        }

        int last = N - 1;
        for(int& b : blacklist){
            // 如果 b 已经在区间 [sz, N)，可以直接忽略
            if(b >= sz) continue;
            // 跳过所有黑名单中的数字
            while(blackSet.count(last)>0) {
                last--;
            }
            // 将黑名单中的索引映射到合法数字
            mapping[b] = last;
            last--;
        }
    }

    int pick() {
        // 随机选取一个索引
        int index = rand() % sz;

        // 这个索引命中了黑名单，需要被映射到其他位置
```

```
        if(mapping.count(index) > 0){
            return mapping[index];
        }
        // 若没命中黑名单，则直接返回
        return index;
    }
};
```

4、TreeMap 与 TreeSet

5、基本线段树

1. 链式实现与数组实现
   最直接的想法，就是使用类似二叉树节点的 SegmentNode 类来实现线段树，我们不妨称之为线段树的链式实现：

```
// 线段树节点
class SegmentNode {
    // 当前节点对应的索引区间
    int l, r;

    // 当前区间内元素的聚合值
    int mergeValue;

    // 左右子节点
    SegmentNode left = null, right = null;
}

// 线段树
class SegmentTree {
    SegmentNode root;

    // ...
}
```

因为线段树是一种近似于完全二叉树的结构，所以也可以用数组来存储线段树，不需要真的使用 SegmentNode 构建树结构，我们不妨称这种实现方式为线段树的数组实现：

```
// 线段树
class SegmentTree {
    // 用数组存储完全二叉树结构
    int[] tree;

    // ...
}
```

链式更容易理解，数组的更抽象但效率更高，实际应用中，直接采用数组的。

- 参考：307. 区域和检索 - 数组可修改

```cpp
#include <functional>
#include <iostream>
#include <stdexcept>
#include <vector>

// 线段树节点
class SegmentNode {
public:
    // 该节点表示的区间范围 [l, r]
    int l, r;
    // [l, r] 区间元素的聚合值（如区间和、区间最大值等）
    int mergeVal;
    SegmentNode* left;
    SegmentNode* right;

    SegmentNode(int mergeVal, int l, int r)
        : mergeVal(mergeVal), l(l), r(r), left(nullptr), right(nullptr) {}
};

class SegmentTree {
private:
    SegmentNode* root;
    std::function<int(int, int)> merger;

    // 定义：将 nums[l..r] 中的元素构建成线段树，返回根节点
    SegmentNode* build(const std::vector<int>& nums, int l, int r) {
        // 区间内只有一个元素，直接返回
        if (l == r) {
            return new SegmentNode(nums[l], l, r);
        }
        // 从中间切分，递归构建左右子树
        int mid = l + (r - l) / 2;
        SegmentNode* left = build(nums, l, mid);
        SegmentNode* right = build(nums, mid + 1, r);
        // 根据左右子树的聚合值，计算当前根节点的聚合值
        SegmentNode* node = new SegmentNode(merger(left->mergeVal, right->mergeVal),
l, r);
        // 组装左右子树
        node->left = left;
        node->right = right;
        return node;
    }

    void update(SegmentNode* node, int index, int value) {
        if (node->l == node->r) {
            // 找到了目标叶子节点，更新值
            node->mergeVal = value;
            return;
        }

```

```
        int mid = node->l + (node->r - node->l) / 2;
        if (index <= mid) {
            // 若 index 较小，则去左子树更新
            update(node->left, index, value);
        } else {
            // 若 index 较大，则去右子树更新
            update(node->right, index, value);
        }
        // 后序位置，左右子树已经更新完毕，更新当前节点的聚合值
        node->mergeVal = merger(node->left->mergeVal, node->right->mergeVal);
    }

    int query(SegmentNode* node, int qL, int qR) {
        if (qL > qR) {
            throw std::invalid_argument("Invalid query range");
        }

        if (node->l == qL && node->r == qR) {
            // 命中了目标区间，直接返回
            return node->mergeVal;
        }

        // 未直接命中区间，需要继续向下查找
        int mid = node->l + (node->r - node->l) / 2;
        if (qR <= mid) {
            // node.l <= qL <= qR <= mid
            // 目标区间完全在左子树中
            return query(node->left, qL, qR);
        } else if (qL > mid) {
            // mid < qL <= qR <= node.r
            // 目标区间完全在右子树中
            return query(node->right, qL, qR);
        } else {
            // node.l <= qL <= mid < qR <= node.r
            // 目标区间横跨左右子树
            // 将查询区间拆分成 [qL, mid] 和 [mid + 1, qR] 两部分，分别向左右子树查询
            // 最后将左右子树的查询结果合并
            return merger(
                query(node->left, qL, mid),
                query(node->right, mid + 1, qR)
            );
        }
    }

public:
    // 创建线段树
    // 输入数组 nums 和一个聚合函数 merger，merger 用于计算区间的聚合值
    SegmentTree(const std::vector<int>& nums, std::function<int(int, int)> merger)
        : merger(merger) {
        root = build(nums, 0, nums.size() - 1);
    }

```

```
    void update(int index, int value) {
        update(root, index, value);
    }

    int query(int qL, int qR) {
        return query(root, qL, qR);
    }
};

int main() {
    std::vector<int> arr = {1, 3, 5, 7, 9};
    // 示例，创建一棵求和线段树
    SegmentTree st(arr, [](int a, int b) { return a + b; });

    std::cout << st.query(1, 3) << std::endl; // 3 + 5 + 7 = 15
    st.update(2, 10);
    std::cout << st.query(1, 3) << std::endl; // 3 + 10 + 7 = 20

    return 0;
}
```

下面看数组实现线段树：

- 这里我们也把数组的索引 `0` 作为根节点，那么对于索引为 `i` 的节点，它的左子节点索引为 `2*i+1`，右子节点索引为 `2*i+2`。
- 数组实现线段树和数组实现二叉堆有一个重要不同：
  假设数组元素个数为 n，那么存储线段树的数组容量应该初始化为 $4n$，而二叉堆数组只需要 n 的容量。
  （为了确保数组能够存储线段树的所有节点）

```
#include <iostream>
#include <vector>
#include <functional>
#include <stdexcept>

class ArraySegmentTree {
    // 用数组存储线段树结构
    std::vector<int> tree;
    // 元素个数
    int n;
    std::function<int(int, int)> merger;

    // 定义：对 nums[l..r] 区间的元素构建线段树，rootIndex 是根节点
    void build(const std::vector<int>& nums, int l, int r, int rootIndex) {
        if (l == r) {
            // 区间内只有一个元素，设置为叶子节点
            tree[rootIndex] = nums[l];
            return;
        }

        // 从中间切分，递归构建左右子树
```

```
        int mid = l + (r - l) / 2;
        int leftRootIndex = leftChild(rootIndex);
        int rightRootIndex = rightChild(rootIndex);
        // 递归构建 nums[l..mid]，根节点为 leftRootIndex
        build(nums, l, mid, leftRootIndex);
        // 递归构建 nums[mid+1..r]，根节点为 rightRootIndex
        build(nums, mid + 1, r, rightRootIndex);

        // 后序位置，左右子树已经构建完毕，更新当前节点的聚合值
        tree[rootIndex] = merger(tree[leftRootIndex], tree[rightRootIndex]);
    }

    void update(int l, int r, int rootIndex, int index, int value) {
        // 当前节点为 rootIndex，对应的区间为 [l, r]
        // 去子树更新 nums[index] 为 value
        if (l == r) {
            // 找到了目标叶子节点，更新值
            tree[rootIndex] = value;
            return;
        }

        int mid = l + (r - l) / 2;
        if (index <= mid) {
            // 若 index 较小，则去左子树更新
            update(l, mid, leftChild(rootIndex), index, value);
        } else {
            // 若 index 较大，则去右子树更新
            update(mid + 1, r, rightChild(rootIndex), index, value);
        }

        // 后序位置，左右子树已经更新完毕，更新当前节点的聚合值
        tree[rootIndex] = merger(tree[leftChild(rootIndex)],
tree[rightChild(rootIndex)]);
    }

    int query(int l, int r, int rootIndex, int qL, int qR) {
        if (qL == l && r == qR) {
            // 命中了目标区间，直接返回
            return tree[rootIndex];
        }

        int mid = l + (r - l) / 2;
        int leftRootIndex = leftChild(rootIndex);
        int rightRootIndex = rightChild(rootIndex);
        if (qR <= mid) {
            // node.l <= qL <= qR <= mid
            // 目标区间完全在左子树中
            return query(l, mid, leftRootIndex, qL, qR);
        } else if (qL > mid) {
            // mid < qL <= qR <= node.r
            // 目标区间完全在右子树中
            return query(mid + 1, r, rightRootIndex, qL, qR);
```

```
        } else {
            // node.l <= qL <= mid < qR <= node.r
            // 目标区间横跨左右子树
            // 将查询区间拆分成 [qL, mid] 和 [mid + 1, qR] 两部分，分别向左右子树查询
            return merger(query(l, mid, leftRootIndex, qL, mid),
                          query(mid + 1, r, rightRootIndex, mid + 1, qR));
        }
    }

    int leftChild(int pos) {
        return 2 * pos + 1;
    }

    int rightChild(int pos) {
        return 2 * pos + 2;
    }

public:
    ArraySegmentTree(const std::vector<int>& nums, std::function<int(int, int)>
mergeFunc)
        : n(nums.size()), merger(mergeFunc) {
        // 分配 4 倍数组长度的空间，存储线段树
        tree.resize(4 * n);
        build(nums, 0, n - 1, 0);
    }

    void update(int index, int value) {
        update(0, n - 1, 0, index, value);
    }

    int query(int qL, int qR) {
        if (qL < 0 || qR >= n || qL > qR) {
            throw std::invalid_argument("Invalid range: [" + std::to_string(qL) + ",
" + std::to_string(qR) + "]");
        }
        return query(0, n - 1, 0, qL, qR);
    }
};

int main() {
    std::vector<int> arr = {1, 3, 5, 7, 9};
    // 示例，创建一棵求和线段树
    ArraySegmentTree st(arr, [](int a, int b) { return a + b; });

    std::cout << st.query(1, 3) << std::endl; // 3 + 5 + 7 = 15
    st.update(2, 10);
    std::cout << st.query(1, 3) << std::endl; // 3 + 10 + 7 = 20
    return 0;
}
```

6、动态线段树

7、懒更新线段树

8、Tire 树

9、朋友圈时间线算法

10、考场座位分配算法

11、计算器的设计

12、两个二叉堆实现中位数算法

13、数组去重问题

(八) 图

1、有向图：环检测与拓扑排序

> - 进行拓扑排序之前，先要确保图中没有环。
> - 图的「逆后序遍历」顺序，就是拓扑排序的结果。

- 参考：207. 课程表
  做法：把题目的输入转化成一幅有向图，然后再判断图中是否存在环。

```
class Solution {
private:
    vector<vector<int>> G;  // 图
    vector<bool> visit;     // 为了访问所有节点的路
    bool flag = true;       // 是否能完成
    vector<bool> onPath;    // traverse 用

public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        int n = prerequisites.size();
        // 把数组转为图
        G.resize(numCourses);
        for(int i=0; i<n; i++){
            int from = prerequisites[i][1];
            int to = prerequisites[i][0];
            G[from].push_back(to);
        }

        // 检测图是否有环即可
        visit.resize(numCourses, false);
        onPath.resize(numCourses, false);
        for(int i=0; i<numCourses && flag; i++)
            traverse(i);
```

```
        return flag;
    }

    // 从节点 start 为起点开始遍历
    void traverse(int start){
        if(!flag)  return;  // 已经失败就别进行了

        // 若当前节点成环了
        if(onPath[start]){
            flag = false;
            return;
        }

        // 不成环需要继续走
        // 优化：若走过，那肯定是不会成环
        if(visit[start])  return;
        visit[start] = true;
        // 这条优化很重要！是在traverse里面就要优化，而不是在主函数里！

        onPath[start] = true;
        for(int nxt : G[start])
            traverse(nxt);
        onPath[start] = false;
    }
};
```

2. 拓扑排序

- 参考：210. 课程表 II

```
class Solution {
private:
    vector<vector<int>> G;  // 图
    vector<bool> visit;
    vector<bool> onPath;
    bool flag = true;
    vector<int> ans;    // 答案

public:
    // 拓扑排序：后序遍历
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        int n = prerequisites.size();
        // 构建图(按照指向先修课来做，因为每次会先完成先修课，再完成本次课)
        G.resize(numCourses);
        for(int i=0; i<n; i++){
            int from = prerequisites[i][0];
            int to = prerequisites[i][1];
            G[from].push_back(to);
        }
        // 下面遍历，同时把答案得到
        visit.resize(numCourses, false);
```

```
        onPath.resize(numCourses, false);
        for(int i=0; i<numCourses  && flag; i++)
            traverse(i);

        // 不能完成
        if(!flag)  return vector<int>{};
        return ans;
    }
    // 从节点 start 开始遍历图
    void traverse(int start){
        if(!flag)  return;

        if(onPath[start]){
            flag = false;
            return;
        }

        if(visit[start])  return;
        visit[start] = true;

        onPath[start] = true;
        for(int i : G[start])
            traverse(i);
        onPath[start] = false;
        ans.push_back(start);
    }
};
```

实际上，上述两个都是 DFS 算法，其实也可以改换思想，写 BFS 的方法：

- 主要是记录所有的节点的入度，同时记录下有多少元素曾经入栈cnt。
- 先把所有入度为 0 的压入栈 Q
- 再依次取出栈顶元素 e 操作，所有 e 的指向元素的入度减一，若减到 0 了则入栈 Q
- 最终栈空，查看cnt == nodeNum 就是答案。

> 对于拓扑排序，实际上就只需要记录下入栈顺序就是答案：`ans[cnt] = e` 即可。

```
class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        // 建图，有向边代表「被依赖」关系
        vector<int>* graph = buildGraph(numCourses, prerequisites);
        // 构建入度数组
        vector<int> indegree(numCourses);
        for (auto& edge : prerequisites) {
            int from = edge[1], to = edge[0];
            // 节点 to 的入度加一
            indegree[to]++;
        }

```

```
        // 根据入度初始化队列中的节点
        queue<int> q;
        for (int i = 0; i < numCourses; i++) {
            if (indegree[i] == 0) {
                // 节点 i 没有入度，即没有依赖的节点
                // 可以作为拓扑排序的起点，加入队列
                q.push(i);
            }
        }

        // 记录遍历的节点个数
        int count = 0;
        // 开始执行 BFS 循环
        while (!q.empty()) {
            // 弹出节点 cur，并将它指向的节点的入度减一
            int cur = q.front();
            q.pop();
            count++;
            for (int next : graph[cur]) {
                indegree[next]--;
                if (indegree[next] == 0) {
                    // 如果入度变为 0，说明 next 依赖的节点都已被遍历
                    q.push(next);
                }
            }
        }

        // 如果所有节点都被遍历过，说明不成环
        return count == numCourses;
    }

    // 建图函数
    vector<int>* buildGraph(int n, vector<vector<int>>& edges) {
        // 图中共有 numCourses 个节点
        vector<vector<int>> graph(numCourses);
        for (int i = 0; i < numCourses; i++) {
            graph[i] = vector<int>();
        }
        for (auto& edge : prerequisites) {
            int from = edge[1], to = edge[0];
            // 添加一条从 from 指向 to 的有向边
            // 边的方向是「被依赖」关系，即修完课程 from 才能修课程 to
            graph[from].push_back(to);
        }
        return graph;
    }
};
```

```
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
```

```cpp
        // 建图，和环检测算法相同
        vector<vector<int>> graph = buildGraph(numCourses, prerequisites);
        // 计算入度，和环检测算法相同
        vector<int> indegree(numCourses);

        for (vector<int> edge : prerequisites) {
            int from = edge[1], to = edge[0];
            indegree[to]++;
        }

        // 根据入度初始化队列中的节点，和环检测算法相同
        queue<int> q;

        for (int i = 0; i < numCourses; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }

        // 记录拓扑排序结果
        vector<int> res(numCourses);

        // 记录遍历节点的顺序（索引）
        int count = 0;

        // 开始执行 BFS 算法
        while (!q.empty()) {
            int cur = q.front();
            q.pop();

            // 弹出节点的顺序即为拓扑排序结果
            res[count] = cur;
            count++;

            for (int next : graph[cur]) {
                indegree[next]--;
                if (indegree[next] == 0) {
                    q.push(next);
                }
            }
        }

        if (count != numCourses) {
            // 存在环，拓扑排序不存在
            return {};
        }

        return res;
    }

private:
    vector<vector<int>> buildGraph(int n, vector<vector<int>>& edges) {
```

```
        // 图中共有 numCourses 个节点
        vector<vector<int>> graph(numCourses);
        for (int i = 0; i < numCourses; i++) {
            graph[i] = vector<int>();
        }
        for (auto& edge : prerequisites) {
            int from = edge[1], to = edge[0];
            // 添加一条从 from 指向 to 的有向边
            // 边的方向是「被依赖」关系，即修完课程 from 才能修课程 to
            graph[from].push_back(to);
        }
        return graph;
    }
};
```

2、名流问题

- 参考：997. 找到小镇的法官
- 参考：277. 搜寻名人
- 给你 n 个人的社交关系（你知道任意两个人之间是否认识），然后请你找出这些人中的「名人」。
  所谓「名人」有两个条件：
  1、所有其他人都认识「名人」。
  2、「名人」不认识任何其他人。

> 这个节点没有一条指向其他节点的有向边；且其他所有节点都有一条指向这个节点的有向边。
> 或者说的专业一点，名人节点的出度为 0，入度为 n - 1。

- 这里的第一个题目是可以直接这样做的，但是第二个就不行了！
- 力扣第 277 题「搜寻名人」就是这个经典问题，不过**并不是直接把邻接矩阵传给你**，而是只告诉你总人数 `n`，同时提供一个 API `knows` 来查询人和人之间的社交关系。很明显，knows API 本质上还是在访问邻接矩阵。

> `bool knows(int i, int j);`能够返回 i 是否认识 j
> 三种方法：

1. 暴力
2. 优化 1
3. 优化 2

> 注意：
> 我再重复一遍所谓「名人」的定义：1、所有其他人都认识名人。2、名人不认识任何其他人。
> 这个定义就很有意思，**它保证了人群中最多有一个名人**。
> 实际上你仔细分析会发现：两个人之间一共就 4 种关系。
>
> 1. 对于情况一，cand 认识 other，所以 cand 肯定不是名人，排除。因为名人不可能认识别人。
> 2. 对于情况二，other 认识 cand，所以 other 肯定不是名人，排除。
> 3. 对于情况三，他俩互相认识，肯定都不是名人，可以随便排除一个。

> 4. 对于情况四，他俩互不认识，肯定都不是名人，可以随便排除一个。因为名人应该被所有其他人认识。
>
> 综上，**只要观察任意两个之间的关系，就至少能确定一个人不是名人。**（优化1）

进一步优化：理解上面的优化解法，其实可以不需要额外的空间解决这个问题。（优化 2）

T1暴力：

```cpp
class Solution {
public:
    int findCelebrity(int n) {
        for (int cand = 0; cand < n; cand++) {
            int other;
            for (other = 0; other < n; other++) {
                if (cand == other) continue;
                // 保证其他人都认识 cand，且 cand 不认识任何其他人
                // 否则 cand 就不可能是名人
                if (knows(cand, other) || !knows(other, cand)) {
                    break;
                }
            }
            if (other == n) {
                // 找到名人
                return cand;
            }
        }
        // 没有一个人符合名人特性
        return -1;
    }
};
```

T2优化 1：

```cpp
class Solution {
public:
    int findCelebrity(int n) {
        if (n == 1) return 0;
        // 将所有候选人装进队列
        queue<int> q;
        for (int i = 0; i < n; i++) {
            q.push(i);
        }
        // 一直排除，直到只剩下一个候选人停止循环
        while (q.size() >= 2) {
            // 每次取出两个候选人，排除一个
            int cand = q.front();
            q.pop();
            int other = q.front();
            q.pop();
            if (knows(cand, other) || !knows(other, cand)) {
                // cand 不可能是名人，排除，让 other 归队
                q.push(other);
            } else {
                // other 不可能是名人，排除，让 cand 归队
                q.push(cand);
```

```
            }
        }

        // 现在排除得只剩一个候选人，判断他是否真的是名人
        int cand = q.front();
        for (int other = 0; other < n; other++) {
            if (other == cand) {
                continue;
            }
            // 保证其他人都认识 cand，且 cand 不认识任何其他人
            if (!knows(other, cand) || knows(cand, other)) {
                return -1;
            }
        }
        // cand 是名人
        return cand;
    }
};

T3优化 2:
class Solution {
public:
    int findCelebrity(int n) {
        int cand = 0;
        for (int other = 1; other < n; other++) {
            if (!knows(other, cand) || knows(cand, other)) {
                // cand 不可能是名人，排除
                // 假设 other 是名人
                cand = other;
            } else {
                // other 不可能是名人，排除
                // 什么都不用做，继续假设 cand 是名人
            }
        }

        // 现在的 cand 是排除的最后结果，但不能保证一定是名人
        for (int other = 0; other < n; other++) {
            if (cand == other) continue;
            // 需要保证其他人都认识 cand，且 cand 不认识任何其他人
            if (!knows(other, cand) || knows(cand, other)) {
                return -1;
            }
        }

        return cand;
    }
};
```

3、二分图判定

- 二分图的定义：二分图的顶点集可分割为两个互不相交的子集，图中每条边依附的两个顶点都分属于这两个子集，且两个子集内的顶点不相邻。

  给你一幅「图」，请你用两种颜色将图中的所有顶点着色，且使得任意一条边的两个端点的颜色都不相同，你能做到吗?
  这就是图的「双色问题」，其实这个问题就等同于二分图的判定问题，如果你能够成功地将图染色，那么这幅图就是一幅二分图，反之则不是：

说白了就是遍历一遍图，一边遍历一边染色，看看能不能用两种颜色给所有节点染色，且相邻节点的颜色都不相同。

- 参考：785. 判断二分图
- 有 DFS、BFS 两种写法。

```cpp
class Solution {
private:
    bool flag = true;
    vector<int> color;  // 0 未上色 1 红色 2 蓝色

public:
    // 二分图算法
    bool isBipartite(vector<vector<int>>& graph) {
        int n = graph.size();
        color.resize(n);
        // 所有子图需要都能满足
        for(int i=0; i<n && flag; i++)
            if(color[i]==0){
                color[i] = 1;   // 第一个染色为 1
                dfs(graph, i);
            }

        return flag;
    }
    // 遍历子图 DFS
    void dfs(vector<vector<int>>& G, int start){
        if(!flag)  return;

        for(int i : G[start]){
            // 若未上色，上不同色
            if(color[i]==0){
                color[i] = color[start]==1 ? 2 : 1;
                dfs(G, i);
            }
            // 若已上色，判断是否满足条件
            else{
                if(color[i] == color[start])
                    flag = false;
            }
        }
    }
```

```
};

class Solution {
private:
    bool flag = true;
    vector<int> color;  // 0 未上色 1 红色 2 蓝色

public:
    // 二分图算法
    bool isBipartite(vector<vector<int>>& graph) {
        int n = graph.size();
        color.resize(n);
        // 所有子图需要都能满足
        for(int i=0; i<n && flag; i++)
            if(color[i]==0){
                color[i] = 1;   // 第一个染色为 1
                bfs(graph, i);
            }

        return flag;
    }
    // 遍历子图 BFS
    void bfs(vector<vector<int>>& G, int start){
        if(!flag)  return;
        queue<int> Q;
        Q.push(start);
        while(!Q.empty() && flag){
            int cur = Q.front();  Q.pop();
            for(int i : G[cur]){
                if(color[i]==0){
                    color[i] = color[cur]==1 ? 2 : 1;
                    Q.push(i);
                }
                else{
                    if(color[i]==color[cur])
                        flag = false;
                }
            }
        }
    }
};
```

- 参考题：886. 可能的二分法

```
class Solution {
private:
    bool flag = true;
    vector<int> color;  // 0未染色 1 蓝 2 红
    vector<vector<int>> G;

public:
```

```cpp
    bool possibleBipartition(int n, vector<vector<int>>& dislikes) {
        color.resize(n, 0);
        G.resize(n);
        // 构建图(不喜欢的人连接在一起，后面染色成不同即可)
        for(auto &p : dislikes){
            int a = p[0]-1, b = p[1]-1;
            G[a].push_back(b);
            G[b].push_back(a);
        }

        // 开始染色
        for(int i=0; i<n && flag; i++){
            if(color[i] == 0){
                color[i] = 1;
                dfs(i);
            }
        }

        return flag;
    }

    // DFS
    void dfs(int start){
        if(!flag)  return;

        for(int i : G[start]){
            if(color[i]==0){
                color[i] = color[start]==1 ? 2 : 1;
                dfs(i);
            }
            else{
                if(color[i]==color[start])
                    flag = false;
            }
        }
    }
};
```

4、Union-Find 并查集

- 并查集（Union-Find）算法是一个专门针对「动态连通性」的算法。
- 动态连通性
  「连通」是一种等价关系（自反、对称、传递）
  连通性、连通分量。

- 我们使用森林（若干棵树）来表示图的动态连通性，用数组来具体实现这个森林。
- 怎么用森林来表示连通性呢？我们设定树的每个节点有一个指针指向其父节点，如果是根节点的话，这个指针指向自己。
- 如果某两个节点被连通，则让其中的（任意）一个节点的根节点接到另一个节点的根节点上。

- 如果节点 p 和 q 连通的话，它们一定拥有相同的根节点。

下面是基本的版本：

```
class UF {
    // 记录连通分量
    int count;
    // 节点 x 的父节点是 parent[x]
    vector<int> parent;

public:
    // 构造函数，n 为图的节点总数
    UF(int n) {
        // 一开始互不连通
        this->count = n;
        // 父节点指针初始指向自己
        parent.resize(n);
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }

    // 返回某个节点 x 的根节点
    int find(int x) {
        // 根节点的 parent[x] == x
        while (parent[x] != x)
            x = parent[x];
        return x;
    }

    // 返回当前的连通分量个数
    void union_(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ)
            return;
        // 将两棵树合并为一棵
        parent[rootP] = rootQ;
        // parent[rootQ] = rootP 也一样

        // 两个分量合二为一
        count--;
    }

    bool connected(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        return rootP == rootQ;
    }

    int count() {
        return count;
```

```
    }
};
```

由于可能不太平衡，每个树可能会退化成为一个链表，那么时间复杂度都是 $O(n)$。

下面是平衡性优化：
我们其实是希望，小一些的树接到大一些的树下面，这样就能避免头重脚轻，更平衡一些。解决方法是额外使用一个 size 数组，记录每棵树包含的节点数，我们不妨称为「重量」。

```cpp
class UF {
private:
    int count;
    vector<int> parent;
    // 新增一个数组记录树的“重量”
    vector<int> size;

public:
    UF(int n) {
        count = n;
        parent.resize(n);
        // 最初每棵树只有一个节点
        // 重量应该初始化 1
        size.resize(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    void union_(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ)
            return;

        // 小树接到大树下面，较平衡
        if (size[rootP] > size[rootQ]) {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        } else {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        }
        count--;
    }
};
```

树的高度大致在 $\log N$ 这个数量级，极大提升执行效率。
此时，`find`，`union`，`connected` 的时间复杂度都下降为 $O(\log N)$。

下面，还可以进行路径压缩：
其实我们并不在乎每棵树的结构长什么样，只在乎根节点。因为无论树长啥样，树上的每个节点的根节点都是相同的，所以能不能进一步压缩每棵树的高度，使树高始终保持为常数？

这样每个节点的父节点就是整棵树的根节点，`find` 就能以 $O(1)$ 的时间找到某一节点的根节点，相应的，`connected` 和 `union` 复杂度都下降为 $O(1)$。
如果使用路径压缩技巧，那么 size 数组的平衡优化就没有必要了。

```
class UF {
private:
    // 连通分量个数
    int _count;
    // 存储每个节点的父节点
    vector<int> parent;

public:
    // n 为图中节点的个数
    UF(int n) {
        _count = n;
        parent.resize(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    // 将节点 p 和节点 q 连通
    void union_(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);

        if (rootP == rootQ)
            return;

        parent[rootQ] = rootP;
        // 两个连通分量合并成一个连通分量
        _count--;
    }

    // 判断节点 p 和节点 q 是否连通
    bool connected(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        return rootP == rootQ;
    }

    // 最终优化
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
```

```
        return parent[x];
    }

    // 返回图中的连通分量个数
    int count() {
        return _count;
    }
};
```

Union-Find 算法的复杂度可以这样分析：

- 构造函数初始化数据结构需要 $O(N)$ 的时间和空间复杂度；
- 连通两个节点 `union`、判断两个节点的连通性 `connected`、计算连通分量 `count` 所需的时间复杂度均为 $O(1)$。
- 参考：547. 省份数量

```
// 实现并查集
class UnionFind {
private:
    int count;
    vector<int> parent;

public:
    UnionFind(int n){
        // 初始全不通，每个人根节点为自己
        count = n;
        parent.resize(n);
        for(int i=0; i<n; i++)
            parent[i] = i;
    }
    // 背下来
    int find(int a){
        if(parent[a] != a)
            parent[a] = find(parent[a]);
        return parent[a];
    }
    void union_(int a, int b){
        int ap = find(a);
        int bp = find(b);
        if(ap != bp){
            parent[ap] = bp;
            count--;
        }
    }
    bool connected(int a, int b){
        int ap = find(a);
        int bp = find(b);
        return ap == bp;
    }
    int size(){
        return count;
```

```
    }
};

class Solution {
public:
    // 连通分量的求解
    int findCircleNum(vector<vector<int>>& isConnected) {
        int n = isConnected.size();
        UnionFind UF(n);
        for(int i=0; i<n; i++)
            for(int j=i; j<n; j++)
                if(isConnected[i][j])
                    UF.union_(i, j);
        return UF.size();
    }
};
```

更多参考题：

- 1361. 验证二叉树
- 200. 岛屿数量
- *261. 以图判树
- 310. 最小高度树
- 368. 最大整除子集
- 547. 省份数量
- *582. 杀掉进程
- *737. 句子相似性 II
- 947. 移除最多的同行或同列石头
- 765. 情侣牵手
- 924. 尽量减少恶意软件的传播

5、Kruskal MST最小生成树（并查集+排序贪心）

- 最小生成树算法主要有 Prim 算法（普里姆算法）和 Kruskal 算法（克鲁斯卡尔算法）两种，这两种算法虽然都运用了贪心思想，但从实现上来说差异还是蛮大的。
- 最小生成树
    - 先说「树」和「图」的根本区别：**树不会包含环，图可以包含环**。
    - 如果一幅图没有环，完全可以拉伸成一棵树的模样。说的专业一点，**树就是「无环连通图」**。
    - **生成树**是含有图中所有顶点的「无环连通子图」。
    - 那么最小生成树很好理解了，所有可能的生成树中，权重和最小的那棵生成树就叫「**最小生成树**」。

- Kruskal 算法的一个难点是保证生成树的合法性，因为在构造生成树的过程中，你首先得保证生成的那玩意是棵树（不包含环）对吧，那么 Union-Find 算法就是帮你干这个事儿的。

- 参考：261. 以图判树
- 给你输入编号从 `0` 到 `n-1` 的 `n` 个结点，和一个无向边列表 `edges`（每条边用节点二元组表示），请你判断输入的这些边组成的结构是否是一棵树。
- 我们可以思考一下，**什么情况下加入一条边会使得树变成图（出现环）**？
- 对于添加的这条边，如果该边的两个节点本来就在同一连通分量里，那么添加这条边会产生环；
- 反之，如果该边的两个节点不在同一连通分量里，则添加这条边不会产生环。
- 判断两个节点是否连通（是否在同一个连通分量中）就是 Union-Find 算法的拿手绝活。
- Kruskal 算法
  所谓最小生成树，就是图中若干边的集合`mst`，你要保证这些边：
  1. 包含图中的所有节点。
  2. 形成的结构是树结构（即不存在环）。
  3. 权重和最小。
     关键在于第 3 点，如何保证得到的这棵生成树是权重和最小的：
     贪心思路：**将所有边按照权重从小到大排序，从权重最小的边开始遍历，如果这条边和 mst 中的其它边不会形成环，则这条边是最小生成树的一部分，将它加入 mst 集合；否则，这条边不是最小生成树的一部分，不要把它加入 mst 集合。**
- 参考：1135. 最低成本联通所有城市
- 参考：1584. 连接所有点的最小费用

```cpp
// 先实现并查集
class UnionFind {
private:
    int count;
    vector<int> parent;

public:
    UnionFind(int n){
        // 最初全独立，自己指自己
        count = n;
        parent.resize(n);
        for(int i=0; i<n; i++)
            parent[i] = i;
    }
    int find(int a){
        if(parent[a] != a)
            parent[a] = find(parent[a]);
        return parent[a];
    }
    void union_(int a, int b){
        int ap = find(a);
        int bp = find(b);
        if(ap != bp){
            parent[ap] = bp;
            count--;
        }
```

```
    }
    bool connected(int a, int b){
        int ap = find(a);
        int bp = find(b);
        return ap == bp;
    }
    int size(){
        return count;
    }
};


class Solution {
public:
    int minCostConnectPoints(vector<vector<int>>& points) {
        int n = points.size();
        // 把所有可能的边生成，且按权重排序
        vector<vector<int>> edges;
        for(int i=0; i<n; i++){
            for(int j=i+1; j<n; j++){
                int d = abs(points[i][0]-points[j][0]) + abs(points[i][1]-points[j][1]);
                edges.push_back({i, j, d});
            }
        }
        sort(edges.begin(), edges.end(), [](const vector<int>& A, const vector<int>& B){
            return A[2] < B[2];
        });

        // 按照从小到大把还没连通的边加入
        UnionFind UF(n);
        int mst = 0;
        for(auto e : edges){
            int i = e[0], j = e[1], d = e[2];
            if(!UF.connected(i, j)){
                UF.union_(i, j);
                mst += d;
            }
            // 若已经全部连完，可以提前结束
            if(UF.size() == 1)  break;
        }

        // 返回最小总权重
        return mst;
    }
};
```

Kruskal 算法，主要的难点是利用 Union-Find 并查集算法向最小生成树中添加边，配合排序的贪心思路，从而得到一棵权重之和最小的生成树。
空间复杂度：$O(E+V)$
时间复杂度：$O(E\log E+E)$ 主要是排序的时间。

6、Prim MST最小生成树（BFS+切分贪心）

> Kruskal 和 Prim 都是最小生成树的算法，两种算法的时间复杂度是一样的。会使用一个即可。
> 推荐使用前面的 Kruskal 即可，因为就是结合并查集 UF + 权重排序贪心即可。
> 这里的话需要重新理解，因此作为理解即可，实战使用前面的算法。

- 切分定理：
  - 「切分」就是将一幅图分为两个不重叠且非空的节点集合。
  - 对于任意一种「切分」，其中权重最小的那条「横切边」一定是构成最小生成树的一条边。

    既然每一次「切分」一定可以找到最小生成树中的一条边，那我就随便切呗，每次都把权重最小的「横切边」拿出来加入最小生成树，直到把构成最小生成树的所有边都切出来为止。

    Prim 算法的逻辑就是这样，每次切分都能找到最小生成树的一条边，然后又可以进行新一轮切分，直到找到最小生成树的所有边为止。
    同时，注意计算有方法：

- 参考：1135. 最低成本联通所有城市
- 参考：1584. 连接所有点的最小费用

```cpp
class Solution {
public:
    int minCostConnectPoints(vector<vector<int>>& points) {
        int n = points.size();
        vector<list<vector<int>>> graph = buildGraph(n, points);
        return Prim(graph).weightSum();
    }

private:
    vector<list<vector<int>>> buildGraph(int n, vector<vector<int>>& points) {
        vector<list<vector<int>>> graph(n);
        // 生成所有边及权重
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                int xi = points[i][0], yi = points[i][1];
                int xj = points[j][0], yj = points[j][1];
                int weight = abs(xi - xj) + abs(yi - yj);
                // 用 points 中的索引表示坐标点
                graph[i].push_back({i, j, weight});
                graph[j].push_back({j, i, weight});
            }
```

```
        }
        return graph;
    }
};

class Prim {
private:
    // 核心数据结构，存储「横切边」的优先级队列
    priority_queue<vector<int>, vector<vector<int>>, function<bool(vector<int>,
vector<int>)>> pq;
    // 类似 visited 数组的作用，记录哪些节点已经成为最小生成树的一部分
    vector<bool> inMST;
    // 记录最小生成树的权重和
    int weightSum_ = 0;
    // graph 是用邻接表表示的一幅图，
    // graph[s] 记录节点 s 所有相邻的边，
    // 三元组 int[]{from, to, weight} 表示一条边
    vector<list<vector<int>>> graph;

public:
    Prim(vector<list<vector<int>>>& graph) : graph(graph),
        pq([](vector<int> a, vector<int> b) {
            // 按照边的权重从小到大排序
            return a[2] > b[2];
        }) {
        // 图中有 n 个节点
        int n = graph.size();
        this->inMST.resize(n, false);

        // 随便从一个点开始切分都可以，我们不妨从节点 0 开始
        inMST[0] = true;
        cut(0);
        // 不断进行切分，向最小生成树中添加边
        while (!pq.empty()) {
            vector<int> edge = pq.top(); pq.pop();
            int to = edge[1];
            int weight = edge[2];
            if (inMST[to]) {
                // 节点 to 已经在最小生成树中，跳过
                // 否则这条边会产生环
                continue;
            }
            // 将边 edge 加入最小生成树
            weightSum_ += weight;
            inMST[to] = true;
            // 节点 to 加入后，进行新一轮切分，会产生更多横切边
            cut(to);
        }
    }

    // 将 s 的横切边加入优先队列
    void cut(int s) {
```

```
        // 遍历 s 的邻边
        for (auto& edge : graph[s]) {
            int to = edge[1];
            if (inMST[to]) {
                // 相邻接点 to 已经在最小生成树中，跳过
                // 否则这条边会产生环
                continue;
            }
            // 加入横切边队列
            pq.push(edge);
        }
    }

    // 最小生成树的权重和
    int weightSum() {
        return weightSum_;
    }

    // 判断最小生成树是否包含图中的所有节点
    bool allConnected() {
        for (bool connected : inMST) {
            if (!connected) {
                return false;
            }
        }
        return true;
    }
};
```

总时间复杂度：$O(E \log E)$

7、Dijkstra 单源最短路径

> Dijkstra 算法是一种用于计算图中单源最短路径的算法，本质上是一个经过特殊改造的 BFS 算法，改造点有两个：
>
> 1. 使用 优先级队列，而不是普通队列进行 BFS 算法。
> 2. 添加了一个备忘录，记录起点到每个可达节点的最短路径权重和。

- 参考：743. 网络延迟时间

> 注意：
>
> - `distTo[id]` 永远维护到达 `id` 节点的最小距离。
> - 而 `State` 里面那个 `distFromStart`，维护的是 `State.id` 这个节点**入队时**距离起点的距离。

```
class State {
public:
    int id;
    int distFromStart;
```

```cpp
    State(int i, int d): id(i), distFromStart(d) {}
};

auto cmp = [](const State& a, const State& b){
    return a.distFromStart > b.distFromStart;
};

class Solution {
private:
    // distTo[i] 表示从 start 到 i 的最小路径，动态更新
    vector<int> distTo;
    // 优先队列，把目前节点按照距离start更近排序
    priority_queue<State, vector<State>, decltype(cmp)> PQ;
    // 邻接矩阵，含距离
    vector<vector<int>> G;

public:
    // 单源最短路径，再看最大的
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        // 转为邻接矩阵
        G.resize(n, vector<int>(n, INT_MAX));
        for(int i=0; i<times.size(); i++){
            int from = times[i][0]-1, to = times[i][1]-1;
            G[from][to] = times[i][2];
        }

        // 计算start到所有点的最短
        distTo.resize(n, INT_MAX);
        distTo[k-1] = 0;    // 从起始节点出发
        PQ.push(State(k-1, 0));

        while(!PQ.empty()){
            // 取出一个路程最短的，用来更新他可达的下一跳的最短
            State cur = PQ.top();  PQ.pop();
            int cid = cur.id, cdist = cur.distFromStart;

            // 如果当前距离大于已知最短距离，跳过
            if (cdist > distTo[cid]) continue;

            // 遍历邻居
            for(int v=0; v<n; v++){
                if(G[cid][v] != INT_MAX){
                    int path = cdist + G[cid][v];
                    if(path < distTo[v]){
                        distTo[v] = path;
                        PQ.push(State(v, path));
                    }
                }
            }
        }

        // 最大值就是答案
```

```
57          int maxT = *max_element(distTo.begin(), distTo.end());
58          return maxT==INT_MAX ? -1 : maxT;
59      }
60  };
```

- 其他题目：
  1514. 概率最大的路径
  1631. 最小体力消耗路径
  310. 最小高度树
  542. 01 矩阵
  329. 矩阵中的最长递增路径

第二章、经典暴力搜索算法

(一) DFS、回溯

待完成...

(二) BFS

待完成...

第三章、经典动态规划算法

(一) 基本技巧

待完成...

(二) 子序列类型

待完成...

(三) 背包类型

> 总结：
>
> | 问题 | dp 定义 |
> | --- | --- |
> | 0-1背包 | `dp[i][j]` 表示在前 i 个物品、背包容量 j 时，能够达到的最大价值 |
> | 子集背包 | `dp[i][j]` 表示在前 i 个物品、凑成 j 大小，能否完成 |
> | 完全背包 | `dp[i][j]` 表示在前 i 个物品、凑成 j 大小，能有多少种 |

> 三个的递推公式分别如下：
> $dp[i][j] = max(dp[i-1][j], dp[i-1][j-Wt[i-1]] + Val[i-1])$
> $dp[i][j] = dp[i-1][j] \ || \ dp[i-1][j-nums[i-1]]$
> $dp[i][j] = dp[i-1][j] + dp[i][j-nums[i-1]]$
> 尤其注意，最后一个是充分利用前面的计算信息，甚至利用了本层`i`刚刚计算的信息！

1、0-1 背包

> 背包问题的 dp 数组定义：
> `dp[i][w]` 的定义如下：对于前 `i` 个物品，当前背包的容量为 `w`，这种情况下可以装的最大价值是 `dp[i][w]`。
>
> - 比如说，如果 `dp[3][5] = 6`，其含义为：对于给定的一系列物品中，若只从前 `3` 个物品中选择，当背包容量为 `5` 时，最多可以装下的价值为 `6`。
> - 最终答案就是 `dp[N][W]`
> - `base case` 就是 `dp[0][..] = dp[..][0] = 0`

```
int[][] dp[N+1][W+1]
dp[0][..] = 0
dp[..][0] = 0

for i in [1..N]:
    for w in [1..W]:
        dp[i][w] = max(
            dp[i-1][w], // 把物品 i 装进背包
            dp[i-1][w - wt[i-1]] + val[i-1] //不把物品 i 装进背包
        )
return dp[N][W]
```

```
// 0-1背包问题，重量，价值，背包容量
int zeroOneBag(vector<int>& Wt, vector<int>& Val, int cap){
    int n = Wt.size();
    // dp[i][j] 表示从前i个物品中选，容量为j时，能达到的最大价值
    vector<vector<int>> dp(n+1, vector<int>(cap+1, 0));
    // 初始化
    for(int i=0; i<=n; i++)  dp[i][0] = 0;
    for(int i=0; i<=cap; i++)  dp[0][i] = 0;
    // 自底向上
    for(int i=1; i<=n; i++){
        for(int w=1; w<=cap; w++){
            // 不能装
            if(Wt[i-1] > w)  dp[i][w] = dp[i-1][w];
            // 能装
            else  dp[i][w] = max(
                dp[i-1][w],
                dp[i-1][w-Wt[i-1]] + Val[i-1]
            )
        }
    }
```

```
21
22      // 返回答案
23      return dp[n][cap];
24  }
```

- 参考题：3180. 执行操作可获得的最大总奖励 I

```
1   class Solution {
2   public:
3       int maxTotalReward(vector<int>& rewardValues) {
4           int n = rewardValues.size();
5           // 排序(因为一定是从小到大的取值)
6           sort(rewardValues.begin(), rewardValues.end());
7           int m = 2 * rewardValues[n-1] - 1;    // 无论如何，最大总奖励只能是最大元素的 2 倍 -
    1
8
9           // dp[i][j] 表示从前 i 个物品选择，能否达到总奖励 j
10          vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
11          // 初始化
12          for(int i=0; i<=n; i++)  dp[i][0] = true;
13          for(int j=1; j<=m; j++)  dp[0][j] = false;
14          // 动态规划
15          for(int i=1; i<=n; i++){
16              for(int j=1; j<=m; j++){
17                  int jd = j - rewardValues[i-1];
18                  // 选当前物品：满足 1.有更小的总奖励 2.当前物品奖励 大于 总奖励
19                  if(jd >= 0 && rewardValues[i-1] > jd)
20                      dp[i][j] = dp[i-1][j] || dp[i-1][jd];
21                  // 不选当前物品
22                  else
23                      dp[i][j] = dp[i-1][j];
24              }
25          }
26
27          // 在 dp[n][j] 里面的最大的 j 就是答案
28          int ans = 0;
29          for(int j=0; j<=m; j++)  if(dp[n][j])  ans = j;
30          return ans;
31      }
32  };
```

2、子集背包

- 参考：416. 分割等和子集
- 输入一个只包含正整数的非空数组 `nums`，请你写一个算法，判断这个数组是否可以被分割成两个子集，使得两个子集的元素和相等。
- 为什么说是背包问题?
  给一个可装载重量为 `sum / 2` 的背包和 `N` 个物品，每个物品的重量为 `nums[i]`。现在让你装物品，是否存在一种装法，能够恰好将背包装满?

```cpp
class Solution {
public:
    // 本质是背包问题，即能否装满背包 sum/2
    bool canPartition(vector<int>& nums) {
        // 必须偶数
        int sum = accumulate(nums.begin(), nums.end(), 0);
        if(sum%2)  return false;
        int n = nums.size();
        int m = sum/2;

        // dp[i][j] 表示前 i 个物品，能否凑成 j 大小
        vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
        // 初始化
        for(int i=0; i<=n; i++)  dp[i][0] = true;
        for(int j=1; j<=m; j++)  dp[0][j] = false;
        // 动态规划
        for(int i=1; i<=n; i++){
            for(int j=1; j<=m; j++){
                int ok = j-nums[i-1];
                if(ok>=0)
                    dp[i][j] = dp[i-1][j] || dp[i-1][ok];
                else
                    dp[i][j] = dp[i-1][j];
            }
        }
        // 答案
        return dp[n][m];
    }
};
```

注意到 `dp[i][j]` 都是通过上一行 `dp[i-1][..]` 转移过来的，之前的数据都不会再使用了。
优化：（唯一需要注意的是 j 应该从后往前反向遍历，因为每个物品（或者说数字）只能用一次，以免之前的结果影响其他的结果。）

```cpp
class Solution {
public:
    // 本质是背包问题，即能否装满背包 sum/2
    bool canPartition(vector<int>& nums) {
        int sum = accumulate(nums.begin(), nums.end(), 0);
        if(sum%2)  return false;
        int n = nums.size();
        int m = sum/2;

        vector<bool> dp(m+1, false);
        // 初始化
        dp[0] = true;
        // 动态规划
        for(int i=1; i<=n; i++){
            for(int j=m; j>=0; j--){    // 反过来
                int ok = j-nums[i-1];
                if(ok>=0)  dp[j] = dp[j] || dp[ok];
```

```
            }
        }
        // 答案
        return dp[m];
    }
};
```

3、完全背包

- 参考：518. 零钱兑换 II
- 给你一个整数数组 `coins` 表示不同面额的硬币，另给一个整数 `amount` 表示总金额。请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 `0` 。
  假设每一种面额的硬币有无限个。
- 有一个背包，最大容量为 `amount`，有一系列物品 `coins`，每个物品的重量为 `coins[i]`，每个物品的数量无限。请问有多少种方法，能够把背包恰好装满?
- 暴力：实际上无非需要多次选用硬币。
- 优化：但是只需要修改一处，就可以不用浪费遍历，你要做的只是利用前面已经用过这个新硬币的结果！

```
暴力:
class Solution {
public:
    int change(int amount, vector<int>& coins) {
        int n = coins.size();
        // dp[i][j] 表示前 i 种物品，凑出 j 的方法数
        vector<vector<long long>> dp(n+1, vector<long long>(amount+1, 0));
        // 初始化
        for(int i=0; i<=n; i++)  dp[i][0] = 1;
        for(int j=1; j<=amount; j++)  dp[0][j] = 0;
        // 动态规划
        for(int i=1; i<=n; i++){
            for(int j=1; j<=amount; j++){
                int coin = coins[i-1];
                for(int ok=j; ok >= 0; ok-=coin)
                    dp[i][j] += dp[i-1][ok];
            }
        }
        // 答案
        return dp[n][amount];
    }
};

优化:
class Solution {
public:
    int change(int amount, vector<int>& coins) {
        int n = coins.size();
        // dp[i][j] 表示前 i 种物品，凑出 j 的方法数
        vector<vector<int>> dp(n+1, vector<int>(amount+1, 0));
        // 初始化
```

```
        for(int i=0; i<=n; i++)  dp[i][0] = 1;
        for(int j=1; j<=amount; j++)  dp[0][j] = 0;
        // 动态规划
        for(int i=1; i<=n; i++){
            for(int j=1; j<=amount; j++){
                int ok = j-coins[i-1];
                if(ok >= 0)
                    dp[i][j] = dp[i-1][j] + dp[i][ok]; // 注意这里不是 dp[i-1][ok]
                else
                    dp[i][j] = dp[i-1][j];
            }
        }
        // 答案
        return dp[n][amount];
    }
};
```

4、目标和

> 我们前文经常说回溯算法和递归算法有点类似，有的问题如果实在想不出状态转移方程，尝试用回溯算法暴力解决也是一个聪明的策略，总比写不出来解法强。
> 那么，回溯算法和动态规划到底是啥关系？它俩都涉及递归，算法模板看起来还挺像的，都涉及做「选择」，真的酷似父与子。

- 参考：494. 目标和
- 给你一个非负整数数组 `nums` 和一个整数 `target` 。
- 向数组中的每个整数前添加 `'+'` 或 `'-'` ，然后串联起所有整数，可以构造一个 `表达式` ：
    - 例如，nums = [2, 1] ，可以在 2 之前添加 '+' ，在 1 之前添加 '-' ，然后串联起来得到表达式 "+2-1" 。返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。

显然可以使用回溯法，就是每层选择就行了。

- 时间复杂度：$O(2^n)$

```
回溯法：
class Solution {
private:
    int ans = 0;

public:
    // 回溯法
    int findTargetSumWays(vector<int>& nums, int target) {
        backtrace(nums, target, 0);
        return ans;
    }
    // 当前选择 k ，剩余需要凑成 remain
    void backtrace(vector<int>& nums, int remain, int k){
        // base case
        if(k == nums.size()){
            if(remain == 0)  ans++;
```

```
            return;
        }

        // 选择本层，并进入下层
        // 选择1. 加
        remain -= nums[k];
        backtrace(nums, remain, k+1);
        remain += nums[k];

        // 选择2. 减
        remain += nums[k];
        backtrace(nums, remain, k+1);
        remain -= nums[k];
    }
};
```

动态规划之所以比暴力算法快，是因为动态规划技巧消除了重叠子问题。回溯算法遍历递归树的过程中，也有可能出现相同的子树，我们也可以提前识别并避免重复遍历这些子树，从而提高效率。

```
class Solution {
private:
    unordered_map<string, int> memo;    // 备忘录
public:
    // 动态规划备忘录
    int findTargetSumWays(vector<int>& nums, int target) {
        return dp(nums, 0, target);
    }
    // dp(i, remain) 表示从 nums[i..] 选出达到 remain 的方法数
    int dp(vector<int>& nums, int i, int remain){
        // base case
        if(i == nums.size()){
            if(remain == 0)  return 1;
            else return 0;
        }

        // 本层计算
        // 算前先看是否算过了
        string cur = to_string(i) + ',' + to_string(remain);
        if(memo.find(cur) != memo.end())
            return memo[cur];
        int res = dp(nums, i+1, remain-nums[i]) + dp(nums, i+1, remain+nums[i]);
        memo[cur] = res;
        return res;
    }
};
```

其实，这个问题可以转化为一个子集划分问题，而子集划分问题又是一个典型的背包问题。动态规划总是这么玄学，让人摸不着头脑.....

首先，如果我们把 `nums` 划分成两个子集 `A` 和 `B`，分别代表分配 `+` 的数和分配 `-` 的数，那么他们和 `target` 存在如下关系：

```
sum(A) - sum(B) = target
sum(A) = target + sum(B)
sum(A) + sum(A) = target + sum(B) + sum(A)
2 * sum(A) = target + sum(nums)
```

综上，可以推出 `sum(A) = (target + sum(nums)) / 2`，也就是把原问题转化成：`nums` 中存在几个子集 `A`，使得 `A` 中元素的和为 `(target + sum(nums)) / 2`？

- 变成背包问题的标准形式：
  有一个背包，容量为 `sum`，现在给你 `N` 个物品，第 `i` 个物品的重量为 `nums[i - 1]`（注意 `1 <= i <= N`），每个物品只有一个，请问你有几种不同的方法能够恰好装满这个背包？

```cpp
class Solution {
public:
    // 数学发现：就是凑成 (target+sum) / 2 的子集数量
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum = accumulate(nums.begin(), nums.end(), 0);
        int TS = target + sum;
        // 必须非负，必须偶数
        if(TS<0 || TS%2)  return 0;
        else  return subNum(nums, TS/2);
    }
    // 在 nums 中选出子集达到 target 有多少种(全是正整数或 0)
    // 这个肯定是可以用回溯的，但是千万别！要不然这个数学规律你也是发现了
    // 应该使用动态规划，才能优化时间！！！
    int subNum(vector<int>& nums, int target){
        int n = nums.size();
        // dp[i][j] 从前 i 个元素选出达到和为 j 的方法数
        vector<vector<int>> dp(n+1, vector<int>(target+1, 0));
        // 初始化
        // for(int i=0; i<=n; i++)  dp[i][0] = 1;
        // for(int j=1; j<=target; j++)  dp[0][j] = 0;
        dp[0][0] = 1;   // 由于有 0 的存在，必须这样写（因为可能会更新后面的和为 0 的情况）
        // 动态规划
        for(int i=1; i<=n; i++){
            for(int j=0; j<=target; j++){
                int ok = j-nums[i-1];
                if(ok >= 0)
                    dp[i][j] = dp[i-1][j] + dp[i-1][ok];
                else
                    dp[i][j] = dp[i-1][j];
            }
        }
        // 答案
        return dp[n][target];
    }
};
```

(四) 游戏类型

待完成...

(五) 贪心类型

1、解题框架

请见前文：第零章-贪心算法

- 什么是贪心算法呢？贪心算法可以认为是动态规划算法的一个特例，相比动态规划，使用贪心算法需要满足更多的条件（贪心选择性质），但是效率比动态规划要高。
- 比如说一个算法问题使用暴力解法需要指数级时间，如果能使用动态规划消除重叠子问题，就可以降到多项式级别的时间，如果满足贪心选择性质，那么可以进一步降低时间复杂度，达到线性级别的。
- 什么是贪心选择性质呢，简单说就是：每一步都做出一个局部最优的选择，最终的结果就是全局最优。注意哦，这是一种特殊性质，其实只有一部分问题拥有这个性质。

2、老司机加油算法

- 参考：134. 加油站
- 在一条环路上有 `n` 个加油站，其中第 `i` 个加油站有汽油 `gas[i]` 升。
- 你有一辆油箱容量无限的的汽车，从第 `i` 个加油站开往第 `i+1` 个加油站需要消耗汽油 `cost[i]` 升。你从其中的一个加油站出发，开始时油箱为空。
- 给定两个整数数组 `gas` 和 `cost` ，如果你可以按顺序绕环路行驶一周，则返回出发时加油站的编号，否则返回 `-1` 。如果存在解，则 保证 它是 唯一 的。
- 解法 1：暴力把每个站点试一遍
  注意，肯定是无法通过动态规划优化的，因为动态规划这里的状态是两个变量（站点、剩余汽油数），那和暴力的N^2的复杂度是一样的。
  所以说这道题肯定不是通过简单的剪枝来优化暴力解法的效率（动态规划），而是需要我们发现一些隐藏较深的规律，从而减少一些冗余的计算。
- 解法 2：画图，把每过一站的实际变化计算出来 `change[i] = gas[i]-cost[i]`。那我们就是希望找到一个位置，是一直是正数的即可。那就是在最低点开始！

- 解法 3：贪心。如果选择站点 `i` 作为起点，走到站点 `j` 时发现变负数了，那么 `i` 和 `j` 中间的任意站点 `k` 都不可能作为起点，因为他们走到 `j` 都会变成负数。

```
T1:
class Solution {
public:
    // 暴力解法：从每个出发点都试一遍
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int n = gas.size();
        int ans = -1;
        for(int i=0; i<n; i++){
```

```
            int cur = 0;    // 当前汽油量
            for(int j=0; j<n; j++){
                int p = (i+j)%n;    // 当前位置
                cur += gas[p] - cost[p];
                if(cur < 0)  break;
            }
            if(cur >= 0)
                return i;
        }
        return -1;
    }
};

T2:
class Solution {
public:
    // 暴力解法：从每个出发点都试一遍
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int n = gas.size();
        // 计算每一站的收益、亏损
        int sum = 0;
        int low = -1, lowSum = INT_MAX;
        for(int i=0; i<n; i++){
            sum += gas[i] - cost[i];
            if(sum <= lowSum){
                // 这站一过会进入最低，那么从下一站为起点出发
                low = (i+1)%n;
                lowSum = sum;
            }
        }

        // 若 sum 负数，显然是不能完成
        if(sum < 0)  return -1;
        else  return low;
    }
};

T3:
class Solution {
public:
    // 贪心， i 走到 j 时刚刚变成负数，那么i-j的全部不会是起点，他们走到 j 都会变成负数
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int n = gas.size();
        // 总和负数，肯定失败
        int sum = 0;
        for(int i=0; i<n; i++)  sum += gas[i] - cost[i];
        if(sum < 0)  return -1;

        // 按照贪心方法走
        int start = 0;
        int startSum = 0;   // 从 start 开始走的 sum
        for(int i=0; i<n; i++){
```

```
            startSum += gas[i] - cost[i];
            // 若发现负数了，那么认为从下一站开始走
            if(startSum < 0){
                start = i+1;
                startSum = 0;
            }
        }

        return start % n;
    }
};
```

3、区间调度问题

- 参考：435. 无重叠区间
- 给你很多形如 `[start, end]` 的闭区间，请你设计一个算法，算出这些区间中最多有几个互不相交的区间。
- 做法：按照结束时间排序，然后依次筛选。

> 区间问题肯定按照区间的起点或者终点进行排序。

```
auto cmp = [](const vector<int>&a, const vector<int>& b){
    return a[1] < b[1];
};

class Solution {
public:
    // 贪心：区间调度
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        int n = intervals.size();
        if(n == 0)  return 0;
        // 按照结束时间排序，从小到大
        sort(intervals.begin(), intervals.end(), cmp);
        int ans = 1;    // 至少第一个肯定可以
        int curEnd = intervals[0][1];
        for(int i=1; i<n; i++){
            if(intervals[i][0] >= curEnd){
                ans++;
                curEnd = intervals[i][1];
            }
        }

        // 答案(要移除的个数)
        return n - ans;
    }
};
```

- 参考：452. 用最少数量的箭引爆气球

下面两种写法，实际上是一样的：

```cpp
auto cmp = [](const vector<int>& a, const vector<int>& b){
    return a[0] < b[0];
};

class Solution {
public:
    // 贪心：区间调度
    int findMinArrowShots(vector<vector<int>>& points) {
        int n = points.size();
        // 这次按照开始位置排序，从小到大
        sort(points.begin(), points.end(), cmp);
        // 先选第一个气球
        int cnt = 1;
        int curEnd = points[0][1];  // 保证第一个球要能被射中
        for(int i=1; i<n; i++){
            int s = points[i][0], e = points[i][1];
            // 只要气球在末尾之内，那就一起射
            if(s <= curEnd){
                curEnd = min(curEnd, e);    // 注意更新末尾
            }
            // 末尾不够了，开始新的一箭
            else{
                cnt++;
                curEnd = e;
            }
        }

        return cnt;
    }
};
```

```cpp
auto cmp = [](const vector<int>& a, const vector<int>& b){
    return a[1] < b[1];
};

class Solution {
public:
    // 贪心：可以直接按照结束早晚排序（认为每次就卡在最后的位置进行射击）
    int findMinArrowShots(vector<vector<int>>& points) {
        int n = points.size();
        sort(points.begin(), points.end(), cmp);
        int cnt = 1;
        int end = points[0][1];
        for(int i=1; i<n; i++){
            if(points[i][0] > end){
                cnt++;
                end = points[i][1];
            }
        }

        return cnt;
```

```
    }
};
```

4、扫描线技巧-安排会议室

- 参考：253. 会议室 II
- 炼码：920. 会议室1
- 炼码：919. 会议室2
- 给你输入若干形如 `[begin, end]` 的区间，代表若干会议的开始时间和结束时间，请你计算至少需要申请多少间会议室。
- 换句话说，如果把每个会议的起始时间看做一个线段区间，那么题目就是让你求最多有几个重叠区间，仅此而已。

> 首先，差分数组是可以解决这个问题的，就是把每个时间都看做一个点，给每个区间进行加+1，最后看哪个点的数字最大，就是所需要的会议室数量了。
>
> 这个方法所需要的数组太大了，下面看别的方法。

基于差分数组的思路，我们可以推导出一种更高效，更优雅的解法。

1. 我们首先把这些会议的时间区间进行投影：红色的点代表每个会议的开始时间点，绿色的点代表每个会议的结束时间点。
2. 现在假想有一条带着计数器的线，在时间线上从左至右进行扫描，每遇到**红色**的点，计数器 `count` 加一，每遇到**绿色**的点，计数器 `count` 减一。
   这样一来，每个时刻有多少个会议在同时进行，就是计数器 `count` 的值，`count` 的最大值，就是需要申请的会议室数量。

```
class Solution {
public:
    int minMeetingRooms(vector<Interval> &intervals) {
        int n = intervals.size();
        vector<int> start(n);
        vector<int> end(n);
        for(int i=0; i<n; i++){
            start[i] = intervals[i].start;
            end[i] = intervals[i].end;
        }
        // 各自从小到大排序
        sort(start.begin(), start.end());
        sort(end.begin(), end.end());
        // 扫描计算即可(双指针)
        int ans = 0;
        int cnt = 0;
        int i = 0, j = 0;
        while(i<n && j<n){
            // 下一个点是开始点/结束点?
            // 开始点
            if(start[i] < end[j]){
```

```
                cnt++;
                i++;
            }
            // 结束点
            else{
                cnt--;
                j++;
            }
            ans = max(ans, cnt);
        }

        return ans;
    }
};
```

5、视频剪辑

- 参考：1024. 视频拼接
- 区间问题肯定按照区间的起点或者终点进行排序。

> 这题的细节问题，卡得我叫苦不迭。
> 这里的两层循环的关于 i 是否增加的设计，要细细琢磨。

```
// 按照起点升序，相同起点，结束点降序
auto cmp = [](const vector<int>& a, const vector<int>& b){
    if(a[0] == b[0])
        return a[1] > b[1];
    else
        return a[0] < b[0];
};

class Solution {
public:
    // 按照起点排序
    int videoStitching(vector<vector<int>>& clips, int time) {
        int n = clips.size();
        sort(clips.begin(), clips.end(), cmp);
        // 起始一定要有 0
        if(clips[0][0] > 0)  return -1;

        int cnt = 0;
        int curEnd = 0;    // 表示当前拼接好的结束位置，在这个范围内，可以一直挑选下一个片段
        int nxtEnd = 0;    // 表示挑选的下一个判断的末尾位置，还没拼接上来，正在待选
        int i = 0;
        while(i<n && clips[i][0]<=curEnd){
            // 在范围内挑选
            while(i<n && clips[i][0]<=curEnd){
                nxtEnd = max(nxtEnd, clips[i][1]);
                i++;
            }

```

```
            // 挑完了，开始拼接
            cnt++;
            curEnd = nxtEnd;

            // 完成
            if(curEnd >= time)
                return cnt;
        }

        return -1;
    }
};
```

第四章、其他常见算法技巧

(一) 数学运算技巧

待完成...

(二) 经典面试题

待完成...