leetcode cpp

目录

数字的补数

只出现一次的数字III

贪心算法

分配问题

分发饼干

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。对每个孩子 i，都有一个胃口值 g[i]，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j，都有一个尺寸 s[j] 。如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

示例 1:

> “
> 输入: g = [1,2,3], s = [1,1]
> 输出: 1

示例 2:

> “
> 输入: g = [1,2], s = [1,2,3]
> 输出: 2

两种角度，一，从小孩角度，先满足胃口小的，把大于且最接近的饼干分配给他；二，从饼干角度，先分配分量大的饼干给胃口小于且最接近的孩子。将两数组分别排序，遍历比较即可实现。

角度一:

```
int findContentChildren(vector<int>& children, vector<int>& cookies) {
    sort(children.begin(), children.end());
    sort(cookies.begin(), cookies.end());
    int child = 0, cookie = 0;
    while (child < children.size() && cookie < cookies.size()) {
        if (children[child] <= cookies[cookie])
            child++;
        cookie++;
    }
    return child;
}
```

角度二:

```
static bool cmp(int &a, int &b) {
    return a > b;
```

```
}
int findContentChildren(vector<int>& children, vector<int>& cookies) {
    sort(children.begin(), children.end(), cmp);
    sort(cookies.begin(), cookies.end(), cmp);
    int child = 0, cookie = 0;
    while (child < children.size() && cookie < cookies.size()) {
        if (cookies[cookie] >= children[child])
            cookie++;
        child++;
    }
    return cookie;
}
```

分发糖果

n 个孩子站成一排。给你一个整数数组 ratings 表示每个孩子的评分。你需要按照以下要求，给这些孩子分发糖果：

- 每个孩子至少分配到 1 个糖果。
- 相邻两个孩子评分更高的孩子会获得更多的糖果。
- 请你给每个孩子分发糖果，计算并返回需要准备的最少糖果数目。

示例 1：

> 输入：ratings = [1,0,2]
> 输出：5

示例 2：

> 输入：ratings = [1,2,2]
> 输出：4

首先每个孩子分一个，再从左往右遍历一遍，保证每个孩子相对右边相邻孩子糖果数是正确的，再从右向左遍历一遍，保证每个孩子相对左边相邻孩子糖果数正确，最后求和即可。

```
int candy(vector<int>& ratings) {
    int size = ratings.size();
    if (size < 2)
        return size;
    vector<int> num(size, 1);
    for (int i = 1; i < size; i++) {
        if (ratings[i] > ratings[i-1])
            num[i] = num[i-1] + 1;
    }
    for (int i = size - 1; i > 0; i--) {
```

```
        if (ratings[i] < ratings[i-1] && num[i-1] <= num[i])
            num[i-1] = num[i] + 1;
    }
    return accumulate(num.begin(), num.end(), 0);
}
```

区间问题

无重叠区间

给定一个区间的集合 intervals，其中 intervals[i] = [starti, endi]。返回需要移除区间的最小数量，使剩余区间互不重叠。

示例 1:

> 输入: intervals = [[1,2],[2,3],[3,4],[1,3]]
> 输出: 1

示例 2:

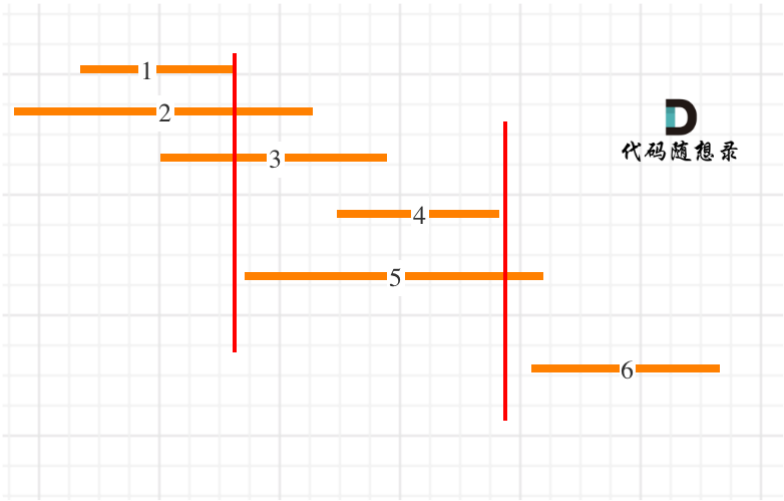
> 输入: intervals = [[1,2], [1,2], [1,2]]
> 输出: 2

示例 3:

> 输入: intervals = [[1,2], [2,3]]
> 输出: 0

又是一个排序题目，两个角度：一，将区间右端升序排列，从左向右删除区间（右边界越小越好）；二，将区间左端降序排列，从左向右删除区间（左边界越大越好）。

如下图所示:



角度一：

```cpp
int eraseOverlapIntervals(vector<vector<int>>& intervals) {
    if (intervals.empty())
        return 0;
    int n = intervals.size();
    sort(intervals.begin(), intervals.end(), [](vector<int>& a, vector<int>& b){return a[1] < b[1];});
    int removed = 0, prev = intervals[0][1];
    for (int i = 1; i < n; i++) {
        if (intervals[i][0] < prev)
            removed++;
        else
            prev = intervals[i][1];
    }
    return removed;
}
```

角度二：

```cpp
int eraseOverlapIntervals(vector<vector<int>>& intervals) {
    if (intervals.empty())
        return 0;
    int n = intervals.size();
    sort(intervals.begin(), intervals.end(), [](vector<int>& a, vector<int>& b){return a[0] > b[0];});
    int removed = 0, prev = intervals[0][0];
    for (int i = 1; i < n; i++) {
        if (intervals[i][1] > prev)
            removed++;
        else
            prev = intervals[i][0];
    }
    return removed;
}
```

练习

种花问题

假设有一个很长的花坛，一部分地块种植了花，另一部分却没有。可是，花不能种植在相邻的地块上，它们会争夺水源，两者都会死去。

给你一个整数数组 flowerbed 表示花坛，由若干 0 和 1 组成，其中 0 表示没种植花，1 表示种植了花。另有一个数 n ，能否在不打破种植规则的情况下种入 n 朵花？能则返回 true ，不能则返回 false。

示例 1：

> 输入：flowerbed = [1,0,0,0,1], n = 1
> 输出：true

示例 2：

> 输入：flowerbed = [1,0,0,0,1], n = 2
> 输出：false

问题可以简化为一个基本模型，两端有花，中间空缺，更复杂的情况可以通过切割得到多个简单情况，且互相独立。又因为基本模型中两端的花地位相同，所以直接从左向右遍历，能种则种，即可得到全局最优。

这里提供另一个想法，虽然与贪心算法无关，但很巧妙，把花坛两端加上0，就可以将两端的特殊情况化为一般，只要有连续的3片空地就能种一朵花。

```
bool canPlaceFlowers(vector<int>& flowerbed, int n) {
    for (int i = 0; i < flowerbed.size(); i++) {
        if (flowerbed[i] == 0
            && (i == 0 || flowerbed[i-1] == 0)
            && (i == flowerbed.size() - 1 || flowerbed[i+1] == 0)) {
            n--;
            flowerbed[i] = 1;
        }
    }
    return n <= 0;
}
```

> 注意：这里用了 || 符号的短路性，判断过程不会越界

用最少数量的箭引爆气球

有一些球形气球贴在一堵用 XY 平面表示的墙面上。墙面上的气球记录在整数数组 points ，其中 points[i] = [xstart, xend] 表示水平直径在 xstart 和 xend 之间的气球。你不知道气球的确切 y 坐标。

一支弓箭可以沿着 x 轴从不同点完全垂直地射出。在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足 xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。弓箭一旦被射出之后，可以无限地前进。

给你一个数组 points，返回引爆所有气球所必须射出的最小弓箭数。

示例 1：

>

> 输入：points = [[10,16],[2,8],[1,6],[7,12]]
> 输出：2

示例 2：

> 输入：points = [[1,2],[3,4],[5,6],[7,8]]
> 输出：4

示例 3：

> 输入：points = [[1,2],[2,3],[3,4],[4,5]]
> 输出：2

这题和上面的无重叠区间很相像，本题要求最少的箭头数量，能一块扎爆的是有重叠区间的，所以在一些重叠的区间中留下一个就行，其他的移除，和例题本质上是一样的，目标都是把一组互不重叠的区间全部找出来，稍微修改一下例题的代码即可。

```
int findMinArrowShots(vector<vector<int>>& points) {
    if (points.empty())
        return 0;
    sort(points.begin(), points.end(),
         [](vector<int>& a, vector<int>& b){return a[1] < b[1];});
    int removed = 0, prev = points[0][1];
    for (int i = 1; i < points.size(); i++) {
        if (points[i][0] <= prev)
            removed++;
        else
            prev = points[i][1];
    }
    return points.size() - removed;
}
```

> 注意：本题中区间边界重合也算重合

划分字母区间

字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。返回一个表示每个字符串片段的长度的列表。

> 输入：S = "ababcbacadefegdehijhklij"
> 输出：[9,7,8]

这题依然可以转化为区间问题，两次遍历得到每个字母的始末位置，然后合并区间即可。

```cpp
vector<int> partitionLabels(string s) {
    vector<int> last(26, 0);
    for (int i = 0; i < s.size(); i++)
        last[s[i] - 'a'] = i;
    vector<int> ans;
    int start = 0, end = 0;
    for (int i = 0; i < s.size(); i++) {
        if (last[s[i] - 'a'] > end)
            end = last[s[i] - 'a'];
        if (end == i) {
            ans.push_back(end - start + 1);
            start = end + 1;
        }
    }
    return ans;
}
```

买卖股票的最佳时机II

给你一个整数数组 prices ， 其中 prices[i] 表示某支股票第 i 天的价格。

在每一天，你可以决定是否购买或出售股票。你在任何时候最多只能持有一股股票。你也可先购买，然后在同一天出售。返回你能获得的最大利润。

示例 1:

> “
> 输入：prices = [7,1,5,3,6,4]
> 输出：7

示例 2:

> “
> 输入：prices = [1,2,3,4,5]
> 输出：4

示例 3:

> “
> 输入：prices = [7,6,4,3,1]
> 输出：0

题目只要求最大利润，所以直接贪心计算差价即可。

```cpp
int maxProfit(vector<int>& prices) {
    int ans = 0;
    for (int i = 0; i < prices.size() - 1; i++) {
        if (prices[i] < prices[i+1])
            ans += (prices[i+1] - prices[i]);
    }
    return ans;
}
```

* 根据身高重建队列

假设有打乱顺序的一群人站成一个队列，数组 people 表示队列中一些人的属性（不一定按顺序）。每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi ，前面正好有 ki 个身高大于或等于 hi 的人。

请你重新构造并返回输入数组 people 所表示的队列。返回的队列应该格式化为数组 queue，其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性（queue[0] 是排在队列前面的人）。

示例 1：

> “
> 输入：people = [[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]
> 输出：[[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]

示例 2：

> “
> 输入：people = [[6,0],[5,0],[4,0],[3,2],[2,2],[1,4]]
> 输出：[[4,0],[5,0],[2,2],[3,2],[1,4],[6,0]]

先排序，身高降序排列，然后 k 升序，先将最高的按 k 值升序放入队列中，然后插入个子矮的，也是按 k 值插入即可，不会影响前面已经插入好的。

```cpp
vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
    sort(people.begin(), people.end(), [](vector<int> &a, vector<int> &b)
{
        if (a[0] > b[0] || (a[0] == b[0] && a[1] < b[1]))
            return true;
        return false;
    });
    vector<vector<int>> ans;
    for (auto &n : people)
        ans.insert(ans.begin() + n[1], n);
    return ans;
}
```

非递减数列

给你一个长度为 n 的整数数组 nums，请你判断在最多改变 1 个元素的情况下，该数组能否变成一个非递减数列。

我们是这样定义一个非递减数列的：对于数组中任意的 i (0<=i<=n-2)，总满足 nums[i] <= nums[i + 1]。

示例 1:

> 输入: nums = [4,2,3]
> 输出: true

示例 2:

> 输入: nums = [4,2,1]
> 输出: false

只要求出最少修改次数，然后与 1 比较即可，对于不符合条件的两个相邻的数，有两种方案，一，修改前面一个，二，修改后面一个，具体采用哪种方案要看修改后是否会影响前面已经符合的部分，比较这两个数的前面一个数和这两个数中后面一个数的大小即可做出判断。

> 注：应修改至恰好满足，即不符合的变成相等的，这样对其他部分影响小，修改次数也最少

```cpp
bool checkPossibility(vector<int>& nums) {
    int ans = 0;
    for (int i = 0; i < nums.size() - 1; i++) {
        if (nums[i] > nums[i+1]) {
            if (ans >= 1)
                return false;
            if (i == 0 || nums[i+1] >= nums[i-1])
                nums[i] = nums[i+1];
            else if (nums[i+1] < nums[i-1])
                nums[i+1] = nums[i];
            ans++;
        }
    }
    return true;
}
```

双指针

算法解释

双指针主要用于遍历数组，两个指针指向不同的元素，从而协同完成任务。也可以延伸到多个数组的多个指针。

若两个指针指向同一数组，遍历方向相同且不会相交，则也称为滑动窗口（两个指针包围的区域即为当前的窗口），经常用于区间搜索。

若两个指针指向同一数组，但是遍历方向相反，则可以用来进行搜索，待搜索的数组往往是排好序的。

对于 C++ 语言，指针还可以玩出很多新的花样。一些常见的关于指针的操作如下：

指针与常量

```
int x;
int *p1 = &x; // 指针可以被修改，值也可以被修改
const int *p2 = &x; // 指针可以被修改，值不可以被修改（const int）
int *const p3 = &x; // 指针不可以被修改（*const），值可以被修改
const int *const p4 = &x; // 指针不可以被修改，值也不可以被修改
```

指针函数与函数指针

```
// addition是指针函数，一个返回类型是指针的函数
int* addition(int a, int b) {
    int *sum = new int(a + b);
    return sum;
}
int subtraction(int a, int b) {
    return a - b;
}
int operation(int x, int y, int (*func)(int, int)) {
    return (*func)(x,y);
}

// minus是函数指针，指向函数的指针
int (*minus)(int, int) = subtraction;
int *m = addition(1, 2);
int n = operation(3, *m, minus);
```

Two Sum

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。

以长度为 2 的整数数组 [index1, index2] 的形式返回这两个整数的下标 index1 和index2。

你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。你所设计的解决方案必须只使用常量级的额外空间。

示例 1：

> 输入：numbers = [2,7,11,15], target = 9
> 输出：[1,2]

示例 2：

> 输入：numbers = [2,3,4], target = 6
> 输出：[1,3]

示例 3：

> 输入：numbers = [-1,0], target = -1
> 输出：[1,2]

采用方向相反的双指针遍历数组，因为已经排好序，所以如果两个数之和小于目标值，则左指针右移，若大于目标值，则右指针左移。

```
vector<int> twoSum(vector<int>& numbers, int target) {
    int l = 0, r = numbers.size() - 1, sum;
    while (l < r) {
        sum = numbers[l] + numbers[r];
        if (sum == target)
            break;
        if (sum < target)
            l++;
        else
            r--;
    }
    return vector<int>{l + 1, r + 1};
}
```

归并有序数组

合并两个有序数组

给你两个按非递减顺序排列的整数数组 nums1 和 nums2，另有两个整数 m 和 n，分别表示 nums1 和 nums2 中的元素数目。

请你合并 nums2 到 nums1 中，使合并后的数组同样按非递减顺序排列。

注意：最终，合并后数组不应由函数返回，而是存储在数组 nums1 中。为了应对这种情况，nums1 的初始长度为 m + n，其中前 m 个元素表示应合并的元素，后 n 个元素为 0，应忽略。nums2 的长度为 n。

示例 1：

> “
> 输入：nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
> 输出：[1,2,2,3,5,6]

示例 2：

> “
> 输入：nums1 = [1], m = 1, nums2 = [], n = 0
> 输出：[1]

示例 3：

> “
> 输入：nums1 = [0], m = 0, nums2 = [1], n = 1
> 输出：[1]
> 注意：因为 m = 0 ，所以 nums1 中没有元素。nums1 中仅存的 0 仅仅是为了确保合并结果可以顺利存放到 nums1 中。

因为这两个数组已经排好序，我们可以把两个指针分别放在两个数组的末尾，即 nums1 的 m−1 位和 nums2 的 n−1 位。每次将较大的那个数字复制到 nums1 的后边，然后向前移动一位。
因为我们也要定位 nums1 的末尾，所以我们还需要第三个指针 pos，以便复制。

> “
> 注意：如果 nums1 的数字已经复制完，不要忘记把 nums2 的数字继续复制；如果 nums2 的数字已经复制完，剩余 nums1 的数字不需要改变，因为它们已经被排好序。

```cpp
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    int pos = m-- + n-- - 1;
    while (m >= 0 && n >= 0)
        nums1[pos--] = nums1[m]>nums2[n] ? nums1[m--] : nums2[n--];
    while (n >= 0)
        nums1[pos--] = nums2[n--];
}
```

快慢指针

环形链表II

给定一个链表的头节点 head，返回链表开始入环的第一个节点。如果链表无环，则返回 null。

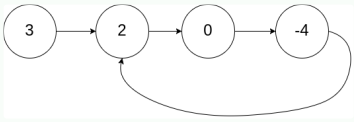
如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 pos 是 -1，则在该链表中没有环。注意：pos 不作为参数进行传递，仅仅是为了标识链表的实际情况。

链表定义如下：

```cpp
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
```

不允许修改链表。

示例：

> 输入： 
>
> head = [3,2,0,-4], pos = 1
> 输出：返回索引为 1 的链表节点

对于链表找环路的问题，有一个通用的解法——**快慢指针**（Floyd 判圈法）。给定两个指针，分别命名为 slow 和 fast，起始位置在链表的开头。每次 fast 前进两步，slow 前进一步。如果 fast 可以走到尽头，那么说明没有环路；如果 fast 可以无限走下去，那么说明一定有环路，且一定存在一个时刻 slow 和 fast 相遇。当 slow 和 fast 第一次相遇时，我们将 fast 重新移动到链表开头，并让 slow 和 fast 每次都前进一步。当 slow 和 fast 第二次相遇时，相遇的节点即为环路的开始点。

```cpp
ListNode *detectCycle(ListNode *head) {
    ListNode *slow = head, *fast = head;
    do {
```

```
        if (!fast || !fast->next)
            return NULL;
        fast = fast->next->next;
        slow = slow->next;
    } while (fast != slow);
    fast = head;
    while (fast != slow){
        slow = slow->next;
        fast = fast->next;
    }
    return fast;
}
```

滑动窗口

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 ""。s,t只由英文字母组成。

> 注意：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 1：

> 输入：s = "ADOBECODEBANC", t = "ABC"
> 输出："BANC"

示例 2：

> 输入：s = "a", t = "a"
> 输出："a"

示例 3:

> 输入: s = "a", t = "aa"
> 输出: ""

本题使用滑动窗口求解，即两个指针 l 和 r 都是从最左端向最右端移动，且 l 的位置一定在 r 的左边或重合。先统计 t 中字符数量，滑动窗口至包含所有字符，l 左移，得到最短子串。

```
string minWindow(string S, string T) {
    vector<int> chars(128, 0);
```

```
    vector<bool> flag(128, false);
    for (char t : T) {
        flag[t] = true;
        chars[t]++;
    }
    int cnt = 0, l = 0, min_l = 0, min_size = S.size() + 1;
    for (int r = 0; r < S.size(); r++) {
        if (flag[S[r]]) {
            if (--chars[S[r]] >= 0)
                cnt++;
            while (cnt == T.size()) {
                if (r - l - 1 < min_size) {
                    min_l = l;
                    min_size = r - l + 1;
                }
                if (flag[S[l]] && ++chars[S[l]] > 0)
                    cnt--;
                l++;
            }
        }
    }
    return min_size>S.size() ? "" : S.substr(min_l, min_size);
}
```

练习

平方数之和

给定一个非负整数 c ，你要判断是否存在两个整数 a 和 b，使得 $a^2 + b^2 = c$ 。

示例 1:

> 输入：c = 5
> 输出：true

示例 2:

> 输入：c = 3
> 输出：false

双指针，一个指向0，另一个指向根号c取整，左指针向右移或者右指针向左移，就可以避免双重循环解决问题。

```
bool judgeSquareSum(int c) {
    long l = 0, r = sqrt(c) + 1;
    long ans = 0;
```

```
    while(l <= r) {
        ans = l * l + r * r;
        if (ans == c)
            return true;
        if (ans < c)
            l++;
        else
            r--;
    }
    return false;
}
```

验证回文字符串II

给定一个非空字符串 s，最多删除一个字符。判断是否能成为回文字符串。

示例 1:

> 输入: s = "aba"
> 输出: true

示例 2:

> 输入: s = "abca"
> 输出: true

示例 3:

> 输入: s = "abc"
> 输出: false

双指针，一个在开头，一个在末尾，相向移动，不符合的字符记录下来即可。

```
bool checkPalindrome(const string &s, int low , int high) {
    for(int i = low, j = high; i < j; i++,j--) {
        if(s[i] != s[j])
            return false;
    }
    return true;
}
bool validPalindrome(string s) {
    int l = 0;
    int r = s.size() - 1;
    while(l < r) {
        if(s[l] == s[r]) {
```

```
            l++;
            r--;
        }
        else
            return checkPalindrome(s, l+1, r) || checkPalindrome(s, l, r-1);
    }
    return true;
}
```

删除字母匹配字符串

给你一个字符串 s 和一个字符串数组 dictionary，找出并返回 dictionary 中最长的字符串，该字符串可以通过删除 s 中的某些字符得到。均只由小写英文字母组成。

如果答案不止一个，返回长度最长且字母序最小的字符串。如果答案不存在，则返回空字符串。

示例 1：

> “
> 输入：s = "abpcplea", dictionary = ["ale","apple","monkey","plea"]
> 输出："apple"

示例 2：

> “
> 输入：s = "abpcplea", dictionary = ["a","b","c"]
> 输出："a"

双指针，一个用于遍历 s，另一个遍历 d 中的字符串，记录满足题目要求的字符串即可。

```
string findLongestWord(string s, vector<string>& dictionary) {
    string ans = "";
    for (string d : dictionary) {
        int m = 0, n = 0;
        if (ans.size() < d.size()
            || (ans.size() == d.size() && ans.compare(d) > 0)) {
            while (m < s.size() && n < d.size()) {
                if (s[m] == d[n])
                    n++;
                m++;
            }
            if (n == d.size())
                ans = d;
        }
    }
    return ans;
```

```
}
```

至多包含 k 个不同字符的最长子串

给定一个字符串 s，找出至多包含 k 个不同字符的最长子串 T 的长度。

示例 1:

> 输入: s = "eceba", k = 2
> 输出: 3

示例 2:

> 输入: s = "aa", k = 1
> 输出: 2

滑动窗口。

```
int lengthOfLongestSubstringKDistinct(string s, int k) {
    unordered_map<char,int> m;
    int maxlen = 0;
    for(int i = 0, j = 0; i < s.size(); ++i) {
        if(m.size() <= k)
            m[s[i]]++;
        while(m.size() > k) {
            if(--m[s[j]] == 0)
                m.erase(s[j]);
            j++;
        }
        maxlen = max(maxlen, i-j+1);
    }
    return maxlen;
}
```

> 因为这题是付费的，所以我并不确定是否完全AC，仅供参考

二分查找

算法解释

二分查找也常被称为二分法或者折半查找，每次查找时通过将待查找区间分成两部分并只取一部分继续查找，将查找的复杂度大大减少。对于一个长度为 O(n) 的数组，二分查找的时间复杂度为 O(log n)。二分查找适用对象必须是排好序的数组。

具体到代码上，二分查找时区间的左右端取开区间还是闭区间在绝大多数时候都可以，因此有些初学者会容易搞不清楚如何定义区间开闭性。这里我提供两个小诀窍，第一是尝试熟练使用一种写法，比如左闭右开（满足 C++、Python 等语言的习惯）或左闭右闭（便于处理边界条件），尽量只保持这一种写法；第二是在做题时思考如果最后区间只剩下一个数或者两个数，自己的写法是否会陷入死循环，如果某种写法无法跳出死循环，则考虑尝试另一种写法。

二分查找也可以看作双指针的一种特殊情况，但我们一般会将二者区分。双指针类型的题，指针通常是一步一步移动的，而在二分查找里，指针每次移动半个区间长度。

求开方

x 的平方根

给你一个非负整数 x，计算并返回 x 的算术平方根。由于返回类型是整数，结果只保留整数部分，小数部分将被舍去。

注意：不允许使用任何内置指数函数和算符，例如 pow(x, 0.5) 或者 x ** 0.5。

示例 1：

> 输入：x = 4
> 输出：2

示例 2：

> 输入：x = 8
> 输出：2

在 [0, x] 区间二分查找。另外还有一个更快的牛顿迭代法 $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

二分查找：

```
int mySqrt(int a) {
    if (a == 0)
        return a;
```

```
    int l = 1, r = a, mid, sqrt;
    while (l <= r) {
        mid = (l + r) / 2;
        sqrt = a / mid;
        if (sqrt == mid)
            return mid;
        else if (mid > sqrt)
            r = mid - 1;
        else
            l = mid + 1;
    }
    return r;
}
```

牛顿迭代法：

```
int mySqrt(int a) {
    long x = a;
    while (x * x > a)
        x = (x + a / x) / 2;
    return x;
}
```

查找区间

查找元素始末位置

给你一个按照非递减顺序排列的整数数组 nums，和一个目标值 target。请你找出给定目标值在数组中的开始位置和结束位置。如果数组中不存在目标值 target，返回 [-1, -1]。

你必须设计并实现时间复杂度为 O(log n) 的算法解决此问题。

示例 1：

> 输入：nums = [5,7,7,8,8,10], target = 8
> 输出：[3,4]

示例 2：

> 输入：nums = [5,7,7,8,8,10], target = 6
> 输出：[-1,-1]

示例 3：

> 输入：nums = [], target = 0
> 输出：[-1,-1]

二分查找定位到目标值后向前向后扩大区间即可。

```cpp
vector<int> range(vector<int>& nums, int mid, int target) {
    int l = mid, r = mid;
    while (l >= 0 && nums[l] == target)
        l--;
    while (r < nums.size() && nums[r] == target)
        r++;
    return vector<int>{l+1, r-1};
}
vector<int> searchRange(vector<int>& nums, int target) {
    if (nums.empty())
        return vector<int> {-1, -1};
    int l = 0, r = nums.size() - 1, mid;
    while (l <= r) {
        mid = (l + r) / 2;
        if (nums[mid] == target)
            return range(nums, mid, target);
        else if (nums[mid] < target)
            l = mid + 1;
        else
            r = mid - 1;
    }
    return vector<int> {-1, -1};
}
```

优化后（寻找上下边界也使用二分查找）：

```cpp
vector<int> searchRange(vector<int>& nums, int target) {
    if (nums.empty())
        return vector<int>{-1, -1};
    int lower = lower_bound(nums, target);
    int upper = upper_bound(nums, target) - 1;
    if (lower == nums.size() || nums[lower] != target)
        return vector<int>{-1, -1};
    return vector<int>{lower, upper};
}
int lower_bound(vector<int> &nums, int target) {
    int l = 0, r = nums.size(), mid;
    while (l < r) {
        mid = (l + r) / 2;
        if (nums[mid] >= target)
            r = mid;
        else
            l = mid + 1;
    }
    return l;
}
int upper_bound(vector<int> &nums, int target) {
```

```
    int l = 0, r = nums.size(), mid;
    while (l < r) {
        mid = (l + r) / 2;
        if (nums[mid] > target)
            r = mid;
        else
            l = mid + 1;
    }
    return l;
}
```

旋转数组查找数字

搜索旋转排序数组II

已知存在一个按非降序排列的整数数组 nums，数组中的值不必互不相同。在传递给函数之前，nums 在预先未知的某个下标 k（0 <= k < nums.length）上进行了旋转，使数组变为了 [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]（下标 从 0 开始 计数）。例如，[0,1,2,4,4,4,5,6,6,7] 在下标 5 处经旋转后可能变为 [4,5,6,6,7,0,1,2,4,4]。

给你旋转后的数组 nums 和一个整数 target，请你编写一个函数来判断给定的目标值是否存在于数组中。如果 nums 中存在这个目标值 target，则返回 true，否则返回 false。

你必须尽可能减少整个操作步骤。

示例 1：

> “
> 输入：nums = [2,5,6,0,0,1,2], target = 0
> 输出：true

示例 2：

> “
> 输入：nums = [2,5,6,0,0,1,2], target = 3
> 输出：false

其实数组即使被旋转过一次，也并不影响二分查找的使用，如果中点值小于右端点，则右半区间为排好序的；若中点值等于右端点，不能确定，尝试右端点左移重新取中点；若中点值大于右端点，则左半区间为排好序的。继续二分查找即可。

```
bool search(vector<int>& nums, int target) {
    int l = 0, r = nums.size() - 1, mid;
    while (l <= r) {
        mid = (l + r) / 2;
        if (nums[mid] == target)
            return true;
        if (nums[mid] < nums[r]) {
```

```
            if (nums[mid] <= target && target <= nums[r])
                l = mid + 1;
            else
                r = mid - 1;
        }
        else if (nums[mid] == nums[r])
            r--;
        else {
            if (nums[l] <= target && target <= nums[mid])
                r = mid - 1;
            else
                l = mid + 1;
        }
    }
    return false;
}
```

练习

寻找旋转排序数组的最小值II

已知一个长度为 n 的数组，预先按照升序排列，经由 1 到 n 次旋转后，得到输入数组。例如，原数组 nums = [0,1,4,4,5,6,7] 在变化后可能得到：

- 若旋转 4 次，则可以得到 [4,5,6,7,0,1,4]
- 若旋转 7 次，则可以得到 [0,1,4,4,5,6,7]

注意，数组 [a[0], a[1], a[2], ..., a[n-1]] 旋转一次的结果为数组 [a[n-1], a[0], a[1], a[2], ..., a[n-2]]。

给你一个可能存在重复元素值的数组 nums，它原来是一个升序排列的数组，并按上述情形进行了多次旋转。请你找出并返回数组中的最小元素。

你必须尽可能减少整个过程的操作步骤。

示例 1：

> “
> 输入：nums = [1,3,5]
> 输出：1

示例 2：

> “
> 输入：nums = [2,2,2,0,1]
> 输出：0

别看题目旋转多少次，实际上是上一题的不同表述，同样是一直二分查找至最小值。

```cpp
int findMin(vector<int>& nums) {
    int l = 0, r = nums.size() - 1, mid;
    while (l < r) {
        mid = (l + r) / 2;
        if (nums[mid] < nums[r])
            r = mid;
        else if (nums[mid] == nums[r])
            r--;
        else
            l = mid + 1;
    }
    return nums[l];
}
```

有序数组中的单一元素

给你一个仅由整数组成的有序数组，其中每个元素都会出现两次，唯有一个数只会出现一次。请你找出并返回只出现一次的那个数。

你设计的解决方案必须满足 O(log n) 时间复杂度和 O(1) 空间复杂度。

示例 1:

> “
> 输入: nums = [1,1,2,3,3,4,4,8,8]
> 输出: 2

示例 2:

> “
> 输入: nums = [3,3,7,7,10,11,11]
> 输出: 10

二分查找，根据中点值及其相邻的左右值，以及左右区间的奇偶性判断在哪一半区间。

```cpp
int singleNonDuplicate(vector<int>& nums) {
    int l = 0, r = nums.size() - 1, mid;
    while (l < r) {
        mid = (l + r) / 2;
        if (nums[mid] == nums[mid - 1]) {
            if (mid % 2 == 0)
                r = mid - 2;
            else
                l = mid + 1;
        }
        else if (nums[mid] == nums[mid + 1]) {
            if (mid % 2 == 0)
                l = mid + 2;
```

```
            else
                r = mid - 1;
        }
        else
            return nums[mid];
    }
    return nums[l];
}
```

寻找两个正序数组的中位数

给定两个大小分别为 m 和 n 的正序（从小到大）数组 nums1 和 nums2。请你找出并返回这两个正序数组的中位数。

算法的时间复杂度应该为 O(log (m+n))。

示例 1：

> 输入：nums1 = [1,3], nums2 = [2]
> 输出：2.00000

示例 2：

> 输入：nums1 = [1,2], nums2 = [3,4]
> 输出：2.50000

根据中位数的定义，当 m+n 是奇数时，中位数是两个有序数组中的第 (m+n)/2 个元素，当 m+n 是偶数时，中位数是两个有序数组中的第 (m+n)/2 个元素和第 (m+n)/2+1 个元素的平均值。因此，这道题可以转化成寻找两个有序数组中的第 k 小的数，其中 k 为 (m+n)/2 或 (m+n)/2+1。

假设两个有序数组分别是 A 和 B。要找到第 k 个元素，我们可以比较 A[k/2-1] 和 B[k/2-1]，有三种情况：

- A[k/2-1] < B[k/2-1]，则可以排除 A[0]~A[k/2-1]
- A[k/2-1] > B[k/2-1]，同理
- A[k/2-1] = B[k/2-1]，可以合并进 1 中

有以下三种情况需要特殊处理：

- 如果 A[k/2-1] 或者 B[k/2-1] 越界，那么我们可以选取对应数组中的最后一个元素。在这种情况下，我们必须根据排除数的个数减少 k 的值，而不能直接将 k 减去 k/2。
- 如果一个数组为空，说明该数组中的所有元素都被排除，我们可以直接返回另一个数组中第 k小的元素。
- 如果 k=1，我们只要返回两个数组首元素的最小值即可。

```cpp
int getKthElement(const vector<int>& nums1, const vector<int>& nums2, int k) {
    int m = nums1.size();
    int n = nums2.size();
    int index1 = 0, index2 = 0;
    while (true) {
        if (index1 == m)
            return nums2[index2 + k - 1];
        if (index2 == n)
            return nums1[index1 + k - 1];
        if (k == 1)
            return min(nums1[index1], nums2[index2]);
        int newIndex1 = min(index1 + k / 2 - 1, m - 1);
        int newIndex2 = min(index2 + k / 2 - 1, n - 1);
        int pivot1 = nums1[newIndex1];
        int pivot2 = nums2[newIndex2];
        if (pivot1 <= pivot2) {
            k -= newIndex1 - index1 + 1;
            index1 = newIndex1 + 1;
        }
        else {
            k -= newIndex2 - index2 + 1;
            index2 = newIndex2 + 1;
        }
    }
}
double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
    int totalLength = nums1.size() + nums2.size();
    if (totalLength % 2 == 1)
        return getKthElement(nums1, nums2, (totalLength + 1) / 2);
    else
        return (getKthElement(nums1, nums2, totalLength / 2) + getKthElement(nums1, nums2, totalLength / 2 + 1)) / 2.0;
}
```

排序算法

常用排序算法

以下是一些最基本的排序算法。虽然在 C++ 里可以通过 std::sort() 快速排序，而且刷题时很少需要自己手写排序算法，但是熟习各种排序算法可以加深自己对算法的基本理解，以及解出由这些排序算法引申出来的题目。

快速排序

采用**左闭右开**的二分写法，代码模板如下：

```cpp
void quick_sort(vector<int> &nums, int l, int r) {
    //特殊情况，没有数或只有一个数，无需排序，直接返回
    if (l + 1 >= r)
        return;
    //取区间左端点值为排序对象，目标是比它大的都在右边，比它小的都在左边
    int first = l, last = r - 1, key = nums[first];
    while (first < last) {
        //寻找比key小的，往前调
        while(first < last && nums[last] >= key)
            last--;
        nums[first] = nums[last];//因为nums[first]的值在key中，不会丢失
        //比key大的，往后调
        while (first < last && nums[first] <= key)
            first++;
        nums[last] = nums[first];
    }
    nums[first] = key;  //key来到了属于它的位置
    //继续排左半区间和右半区间
    quick_sort(nums, l, first);
    quick_sort(nums, first + 1, r);
}
```

归并排序

归并排序主要是采用分而治之的思想，将数组不断分解成多个小数组，然后从排序好的小数组中按大小挑出数填充结果数组。

```cpp
void merge_sort(vector<int> &nums, int l, int r, vector<int> &temp) {
    if (l + 1 >= r)
        return;
    int m = (l + r) / 2;
    //分解
    merge_sort(nums, l, m, temp);
```

```
    merge_sort(nums, m, r, temp);
    //归并原理
    int p = l, q = m, i = l;
    //分别从两个小数组开头开始，先挑小的放进结果数组
    while (p < m || q < r) {
        if (q >= r || (p < m && nums[p] <= nums[q]))
            temp[i++] = nums[p++];
        else
            temp[i++] = nums[q++];
    }
    //临时的数组赋回原数组
    for (i = l; i < r; i++)
        nums[i] = temp[i];
}
```

插入排序

先排好小的元素。

```
void insertion_sort(vector<int> &nums, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i; j > 0 && nums[j] < nums[j-1]; j--)
            swap(nums[j], nums[j-1]);
    }
}
```

冒泡排序

先排好大的元素。

```
void bubble_sort(vector<int> &nums, int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-1-i; j++) {
            if (a[j] < a[j+1])
                swap(nums[j], nums[j+1]);
        }
    }
}
```

选择排序

选定第一个元素为最小元，然后向后比较找真正的最小元，与之交换。

```cpp
void selection_sort(vector<int> &nums, int n) {
    int min;
    for (int i = 0; i < n-1; i++) {
        min = i;
        for (int j = i+1; j < n; j++) {
            if (nums[j] < nums[min])
                min = j;
        }
        swap(nums[min], nums[i]);
    }
}
```

快速选择

数组中第 k 大的元素

给定整数数组 nums 和整数 k，请返回数组中第 k 大的元素。

请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

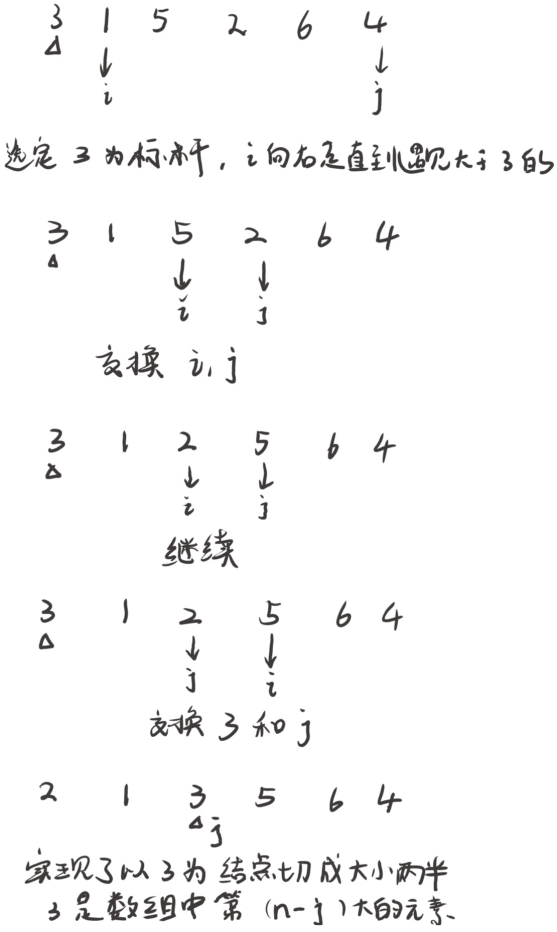
示例 1:

> “
> 输入: [3,2,1,5,6,4] 和 k = 2
> 输出: 5

示例 2:

> “
> 输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
> 输出: 4

快速选择一般用于求解 k-th Element 问题，可以在 O(n) 时间复杂度，O(1) 空间复杂度完成求解工作。快速选择的实现和快速排序相似，首先选定数组第一个值为基准值，将小于它的放在左边，大于它的放在右边，然后根据情况判断继续搜索左区间还是右区间。




```cpp
int findKthLargest(vector<int>& nums, int k) {
    int l = 0, r = nums.size() - 1, target = nums.size() - k;
    while (l < r) {
        int mid = quickSelection(nums, l, r);
        if (mid == target)
            return nums[mid];
        if (mid < target)
            l = mid + 1;
        else
            r = mid - 1;
    }
    return nums[l];
}
// 辅函数 - 快速选择，目标是把数组按大小切成两半，返回的是分界点
int quickSelection(vector<int>& nums, int l, int r) {
    int i = l + 1, j = r;
    while (true) {
        while (i < r && nums[i] <= nums[l])
            i++;
        while (l < j && nums[j] >= nums[l])
            j--;
        if (i >= j)
            break;
        swap(nums[i], nums[j]);
    }
    swap(nums[l], nums[j]);
```

```
    return j;
}
```

桶排序

前 k 个高频元素

给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。你可以按任意顺序返回答案。

示例 1:

> 输入: nums = [1,1,1,2,2,3], k = 2
> 输出: [1,2]

示例 2:

> 输入: nums = [1], k = 1
> 输出: [1]

顾名思义，桶排序的意思是为每个值设立一个桶，桶内记录这个值出现的次数（或其它属性），然后对桶进行排序。针对样例1来说，我们先通过桶排序得到四个桶 [1,2,3,4]，它们的值分别为 [4,2,1,1]，表示每个数字出现的次数。

紧接着，我们对桶的频次进行排序，前 k 大个桶即是前 k 个频繁的数。这里我们可以使用各种
排序算法，甚至可以再进行一次桶排序，把每个旧桶根据频次放在不同的新桶内。针对样例来说，因为目前最大的频次是 4，我们建立 [1,2,3,4] 四个新桶，它们分别放入的旧桶为 [[3,4],[2],[],[1]]，表示不同数字出现的频率。最后，我们从后往前遍历，直到找到 k 个旧桶。

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    unordered_map<int, int> counts;
    int max_count = 0;
    for (const int & num : nums) {
        counts[num]++;
        max_count = max(max_count, counts[num]);
    }
    vector<vector<int>> buckets(max_count + 1);
    for (const auto & p : counts)
        buckets[p.second].push_back(p.first);
    vector<int> ans;
    for (int i = max_count; i >= 0; i--) {
        for (const int & num : buckets[i])
            ans.push_back(num);
        if (ans.size() == k)
```

```
            break;
        }
        return ans;
    }
```

练习

根据字符出现频率排序

给定一个字符串 s，根据字符出现的频率对其进行降序排序。一个字符出现的频率是它出现在字符串中的次数。

返回已排序的字符串。如果有多个答案，返回其中任何一个。

示例 1:

> 输入: s = "tree"
> 输出: "eert"

示例 2:

> 输入: s = "cccaaa"
> 输出: "cccaaa"

示例 3:

> 输入: s = "Aabb"
> 输出: "bbAa"

即桶排序。

```
string frequencySort(string s) {
    unordered_map<char, int> counts;
    int max_count = 0;
    for (const char & a : s) {
        counts[a]++;
        max_count = max(max_count, counts[a]);
    }
    vector<vector<char>> buckets(max_count + 1);
    for (const auto & p : counts)
        buckets[p.second].push_back(p.first);
    string ans;
    for (int i = max_count; i >= 0; i--) {
        for (const int & a : buckets[i]) {
            for (int j = 0; j < i; j++)
                ans.push_back(a);
```

```
        }
    }
    return ans;
}
```

颜色分类

给定一个包含红色、白色和蓝色共 n 个元素的数组 nums，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

必须在不使用库的sort函数的情况下解决这个问题。

示例 1：

> “
> 输入：nums = [2,0,2,1,1,0]
> 输出：[0,0,1,1,2,2]

示例 2：

> “
> 输入：nums = [2,0,1]
> 输出：[0,1,2]

桶排序。

```
void sortColors(vector<int>& nums) {
    unordered_map<int, int> counts;
    int max_count = 0;
    for (const int & num : nums)
        counts[num]++;
    int i = 0;
    for (i = 0; i < counts[0]; i++)
        nums[i] = 0;
    for ( ; i < counts[1] + counts[0]; i++)
        nums[i] = 1;
    for ( ; i < counts[2] + counts[1] + counts[0]; i++)
        nums[i] = 2;
    return;
}
```

搜索算法

算法解释

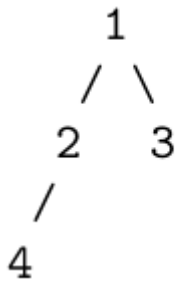
搜索算法是利用计算机的高性能来有目的的穷举一个问题解空间的部分或所有的可能情况，从而求出问题的解的一种方法。

在大规模实验环境中，通常通过在搜索前，根据条件降低搜索规模；根据问题的约束条件进行剪枝；利用搜索过程中的中间解，避免重复计算这几种方法进行优化。

深度优先搜索

深度优先搜索（depth-first seach，DFS）在搜索到一个新的节点时，立即对该新节点进行遍历；因此遍历需要用先入后出的栈来实现，也可以通过与栈等价的递归来实现。对于树结构而言，由于总是对新节点调用遍历，因此看起来是向着“深”的方向前进。

考虑如下一颗简单的树。我们从 1 号节点开始遍历，假如遍历顺序是从左子节点到右子节点，那么按照优先向着“深”的方向前进的策略，假如我们使用递归实现，我们的遍历过程为 1（起始节点）->2（遍历更深一层的左子节点）->4（遍历更深一层的左子节点）->2（无子节点，返回父结点）->1（子节点均已完成遍历，返回父结点）->3（遍历更深一层的右子节点）->1（无子节点，返回父结点）->结束程序（子节点均已完成遍历）。如果我们使用栈实现，我们的栈顶元素的变化过程为 1->2->4->3。



深度优先搜索也可以用来检测环路：记录每个遍历过的节点的父节点，若一个节点被再次遍历且父节点不同，则说明有环。我们也可以用之后会讲到的拓扑排序判断是否有环路，若最后存在入度不为零的点，则说明有环。

有时我们可能会需要对已经搜索过的节点进行标记，以防止在遍历时重复搜索某个节点，这种做法叫做状态记录或记忆化（memoization)。

岛屿的最大面积

给你一个大小为 m x n 的二进制矩阵 grid。

岛屿是由一些相邻的 1 (代表土地) 构成的组合，这里的「相邻」要求两个 1 必须在水平或者竖直的四个方向上相邻。你可以假设 grid 的四个边缘都被 0（代表水）包围着。

岛屿的面积是岛上值为 1 的单元格的数目。计算并返回 grid 中最大的岛屿面积。如果没有岛屿，则返回面积为 0。

示例 1：

> 输入:
> [[1,0,1,1,0,1,0,1],
> [1,0,1,1,0,1,1,1],
> [0,0,0,0,0,0,0,1]]
> 输出: 6

示例 2：

> 输入：grid = [[0,0,0,0,0,0,0,0]]
> 输出：0

此题是十分标准的搜索题，我们可以拿来练手深度优先搜索。一般来说，深度优先搜索类型的题可以分为主函数和辅函数，主函数用于遍历所有的搜索位置，判断是否可以开始搜索，如果可以即在辅函数进行搜索。辅函数则负责深度优先搜索的递归调用。

当然，我们也可以使用栈（stack）实现深度优先搜索，但因为栈与递归的调用原理相同，而递归相对便于实现，因此刷题时笔者推荐使用递归式写法，同时也方便进行回溯（见下节）。不过在实际工程上，直接使用栈可能才是最好的选择，一是因为便于理解，二是更不易出现递归栈满的情况。

1.使用栈：

```
vector<int> direction{-1, 0, 1, 0, -1};
int maxAreaOfIsland(vector<vector<int>>& grid) {
    int m = grid.size(), n = m ? grid[0].size() : 0;
    int local_area, area = 0, x, y;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j]) {
                local_area = 1;
                grid[i][j] = 0;
                stack<pair<int, int>> island;
                island.push({i, j});
                while (!island.empty()) {
                    auto [r, c] = island.top();
                    island.pop();
                    for (int k = 0; k < 4; k++) {
                        x = r + direction[k];
                        y = c + direction[k+1];
                        if (x >= 0 && x < m
                            && y >= 0 && y < n && grid[x][y] == 1) {
```

```
                            grid[x][y] = 0;
                            local_area++;
                            island.push({x, y});
                        }
                    }
                }
                area = max(area, local_area);
            }
        }
    }
    return area;
}
```

> 注意：这里我们使用了一个小技巧，对于四个方向的遍历，可以创造一个数组 [-1, 0, 1, 0, -1]，每相邻两位即为上下左右四个方向之一。

h5 2.使用递归：

在辅函数里，一个一定要注意的点是辅函数内递归搜索时，边界条件的判定。边界判定一般有两种写法，一种是先判定是否越界，只有在合法的情况下才进行下一步搜索（即判断放在调用递归函数前）；另一种是不管三七二十一先进行下一步搜索，待下一步搜索开始时再判断是否合法（即判断放在辅函数第一行）。

第一种：

```
vector<int> direction{-1, 0, 1, 0, -1};
// 主函数
int maxAreaOfIsland(vector<vector<int>>& grid) {
    if (grid.empty() || grid[0].empty())
        return 0;
    int max_area = 0;
    for (int i = 0; i < grid.size(); i++) {
        for (int j = 0; j < grid[0].size(); j++) {
            if (grid[i][j] == 1)
                max_area = max(max_area, dfs(grid, i, j));
        }
    }
    return max_area;
}
// 辅函数
int dfs(vector<vector<int>>& grid, int r, int c) {
    if (grid[r][c] == 0)
        return 0;
    grid[r][c] = 0;
    int x, y, area = 1;
    for (int i = 0; i < 4; i++) {
        x = r + direction[i];
```

```
        y = c + direction[i+1];
        if (x >= 0 && x < grid.size() && y >= 0 && y < grid[0].size())
            area += dfs(grid, x, y);
    }
    return area;
}
```

第二种:

```
vector<int> direction{-1, 0, 1, 0, -1};
// 主函数
int maxAreaOfIsland(vector<vector<int>>& grid) {
    if (grid.empty() || grid[0].empty())
        return 0;
    int max_area = 0;
    for (int i = 0; i < grid.size(); i++) {
        for (int j = 0; j < grid[0].size(); j++) {
            max_area = max(max_area, dfs(grid, i, j));
        }
    }
    return max_area;
}
// 辅函数
int dfs(vector<vector<int>>& grid, int r, int c) {
    if (r < 0 || r >= grid.size()
        || c < 0 || c >= grid[0].size() || grid[r][c] == 0)
        return 0;
    grid[r][c] = 0;
    return 1 + dfs(grid, r + 1, c) + dfs(grid, r - 1, c)
        + dfs(grid, r, c + 1) + dfs(grid, r, c - 1);
}
```

省份数量

有 n 个城市，其中一些彼此相连，另一些没有相连。如果城市 a 与城市 b 直接相连，且城市 b 与城市 c 直接相连，那么城市 a 与城市 c 间接相连。省份是一组直接或间接相连的城市，组内不含其他没有相连的城市。

给你一个 n x n 的矩阵 isConnected ，其中 isConnected[i][j] = 1 表示第 i 个城市和第 j 个城市直接相连，而 isConnected[i][j] = 0 表示二者不直接相连。

返回矩阵中省份的数量。

示例 1:

> “

> 输入：isConnected =
> [[1,1,0],
> [1,1,0],
> [0,0,1]]
> 输出：2

示例 2：

> “
> 输入：isConnected =
> [[1,0,0],
> [0,1,0],
> [0,0,1]]
> 输出：3

实际上与上一题是一样的，上题中矩阵每个元素都可看做一个结点，上下左右相邻即有关系；本题中每一行或列为一个结点，元素为 1 即有关系。这里我们采用第一种递归写法。

```cpp
// 主函数
int findCircleNum(vector<vector<int>>& friends) {
    int n = friends.size(), count = 0;
    vector<bool> visited(n, false);
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(friends, i, visited);
            count++;
        }
    }
    return count;
}
// 辅函数
void dfs(vector<vector<int>>& friends, int i, vector<bool>& visited) {
    visited[i] = true;
    for (int k = 0; k < friends.size(); k++) {
        if (friends[i][k] == 1 && !visited[k])
            dfs(friends, k, visited);
    }
}
```

太平洋大西洋水流问题

有一个 m × n 的矩形岛屿，与太平洋和大西洋相邻。“太平洋”处于大陆的左边界和上边界，而“大西洋”处于大陆的右边界和下边界。

这个岛被分割成一个由若干方形单元格组成的网格。给定一个 m x n 的整数矩阵 heights，heights[r][c] 表示坐标 (r, c) 上单元格高于海平面的高度。

岛上雨水较多，如果相邻单元格的高度小于或等于当前单元格的高度，雨水可以直接向北、南、东、西流向相邻单元格。水可以从海洋附近的任何单元格流入海洋。

返回网格坐标 result 的 2D 列表，其中 result[i] = [ri, ci] 表示雨水从单元格 (ri, ci) 流动既可流向太平洋也可流向大西洋。

示例 1：

> 输入: heights =
> [[1,2,2,3,5],
> [3,2,3,4,4],
> [2,4,5,3,1],
> [6,7,1,4,5],
> [5,1,1,2,4]]
> 输出: [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]

示例 2：

> 输入: heights =
> [[2,1],
> [1,2]]
> 输出: [[0,0],[0,1],[1,0],[1,1]]

虽然题目要求的是满足向下流能到达两个大洋的位置，如果我们对所有的位置进行搜索，那么在不剪枝的情况下复杂度会很高。因此我们可以反过来想，从两个大洋开始向上流，这样我们只需要对矩形四条边进行搜索。搜索完成后，只需遍历一遍矩阵，满足条件的位置即为两个大洋向上流都能到达的位置。

```cpp
vector<int> direction{-1, 0, 1, 0, -1};
// 主函数
vector<vector<int>> pacificAtlantic(vector<vector<int>>& matrix) {
    if (matrix.empty() || matrix[0].empty())
        return {};
    vector<vector<int>> ans;
    int m = matrix.size(), n = matrix[0].size();
    vector<vector<bool>> can_reach_p(m, vector<bool>(n, false));
    vector<vector<bool>> can_reach_a(m, vector<bool>(n, false));
    for (int i = 0; i < m; i++) {
        dfs(matrix, can_reach_p, i, 0);
        dfs(matrix, can_reach_a, i, n - 1);
    }
    for (int i = 0; i < n; i++) {
        dfs(matrix, can_reach_p, 0, i);
        dfs(matrix, can_reach_a, m - 1, i);
    }
    for (int i = 0; i < m; i++) {
```

```
        for (int j = 0; j < n; j++) {
            if (can_reach_p[i][j] && can_reach_a[i][j])
                ans.push_back(vector<int>{i, j});
        }
    }
    return ans;
}
// 辅函数
void dfs(const vector<vector<int>>& matrix, vector<vector<bool>>&
can_reach,
         int r, int c) {
    if (can_reach[r][c])
        return;
    can_reach[r][c] = true;
    int x, y;
    for (int i = 0; i < 4; i++) {
        x = r + direction[i];
        y = c + direction[i+1];
        if (x >= 0 && x < matrix.size()
            && y >= 0 && y < matrix[0].size() && matrix[r][c] <=
matrix[x][y])
            dfs(matrix, can_reach, x, y);
    }
}
```

回溯法

回溯法（backtracking）是优先搜索的一种特殊情况，又称为试探法，常用于需要记录节点状态的深度优先搜索。通常来说，排列、组合、选择类问题使用回溯法比较方便。

顾名思义，回溯法的核心是回溯。在搜索到某一节点的时候，如果我们发现目前的节点（及其子节点）并不是需求目标时，我们回退到原来的节点继续搜索，并且把在目前节点修改的状态还原。这样的好处是我们可以始终只对图的总状态进行修改，而非每次遍历时新建一个图来储存状态。在具体的写法上，它与普通的深度优先搜索一样，都有 [修改当前节点状态]→[递归子节点]的步骤，只是多了回溯的步骤，变成[修改当前节点状态]→[递归子节点]→[回改当前节点状态]。

两个小诀窍，一是按引用传状态，二是所有的状态修改在递归完成后回改。

回溯法修改一般有两种情况，一种是修改最后一位输出，比如排列组合；一种是修改访问标
记，比如矩阵里搜字符串。

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。你可以按任意顺序返回答案。

示例 1：

> 输入：nums = [1,2,3]
> 输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

示例 2：

> 输入：nums = [0,1]
> 输出：[[0,1],[1,0]]

示例 3：

> 输入：nums = [1]
> 输出：[[1]]

按数组序穷举所有可能的排列，已经取过的标记一下，找到一组后向前回溯，取消标记，继续按此方法搜索。

```
vector<vector<int>> permute(vector<int>& nums) {
    vector<vector<int>> result;
    vector<int> temp;
    vector<int> isIn(nums.size(), 0);
    retSeq(result, temp, nums, 0, isIn);
    return result;
}
void retSeq(vector<vector<int>> &result, vector<int> temp, vector<int>
nums, int len, vector<int> isIn) {
    if(len == nums.size())
        result.push_back(temp);
    else {
        for(int i = 0; i < nums.size(); i++) {
            if(isIn[i] == 0) {
                isIn[i] = 1;
                temp.push_back(nums[i]);
                len++;
                retSeq(result, temp, nums, len, isIn);
                len--;
                temp.pop_back();
                isIn[i] = 0;
            }
        }
    }
}
```

>

> 这里提供一份更简洁优雅的代码，因为数字互不重复，所以穷举所有交换即可。代码框架和上面还是一样的。

```cpp
// 主函数
vector<vector<int>> permute(vector<int>& nums) {
    vector<vector<int>> ans;
    backtracking(nums, 0, ans);
    return ans;
}
// 辅函数
void backtracking(vector<int> &nums, int level, vector<vector<int>> &ans)
{
    if (level == nums.size() - 1) {
        ans.push_back(nums);
        return;
    }
    for (int i = level; i < nums.size(); i++) {
        swap(nums[i], nums[level]); // 修改当前节点状态
        backtracking(nums, level+1, ans); // 递归子节点
        swap(nums[i], nums[level]); // 回改当前节点状态
    }
}
```

不难看出回溯算法代码框架如下：

```
主函数 {
        定义结果数组;
        调用辅函数(目标数组，结果数组，递归层数);
        返回结果数组;
}
辅函数 {
        递归出口;
        循环 {
            完成当前层的操作;
            递归调用下一层的辅函数;
            回溯当前层的改变量;
        }
}
```

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。

你可以按任何顺序返回答案。

示例 1：

> “

> 输入：n = 4, k = 2
> 输出：
> [[2,4],
> [3,4],
> [2,3],
> [1,2],
> [1,3],
> [1,4]]

示例 2：

> “
> 输入：n = 1, k = 1
> 输出：[[1]]

和上一题区别在于取的数字个数和不考虑取出顺序，第一点只是影响了结果数组大小，第二点可以这样解决：向后取数字，控制取出的组合升序，这样就避免了重复。

```
// 主函数
vector<vector<int>> combine(int n, int k) {
    vector<vector<int>> ans;
    vector<int> comb(k, 0);
    int count = 0;
    backtracking(ans, comb, count, 1, n, k);
    return ans;
}
// 辅函数
void backtracking(vector<vector<int>>& ans, vector<int>& comb, int&
count, int pos, int n, int k) {
    if (count == k) {
        ans.push_back(comb);
        return;
    }
    for (int i = pos; i <= n; i++) {
        comb[count++] = i;
        backtracking(ans, comb, count, i + 1, n, k);
        count--;
    }
}
```

单词搜索

给定一个 m x n 二维字符网格 board 和一个字符串单词 word。如果 word 存在于网格中，返回 true；否则，返回 false。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例 1：

> 输入：board =
> [["A","B","C","E"],
> ["S","F","C","S"],
> ["A","D","E","E"]],
> word = "ABCCED"
> 输出：true

示例 3：

> 输入：board =
> [["A","B","C","E"],
> ["S","F","C","S"],
> ["A","D","E","E"]],
> word = "ABCB"
> 输出：false

在我们对任意位置进行深度优先搜索时，我们先标记当前位置为已访问，以避免重复遍历（如防止向右搜索后又向左返回）；在所有的可能都搜索完成后，再回改当前位置为未访问，防止干扰其它位置搜索到当前位置。

```
bool exist(vector<vector<char>>& board, string word) {
    if (board.empty())
        return false;
    int m = board.size(), n = board[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    bool find = false;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            backtracking(i, j, board, word, find, visited, 0);
    }
    return find;
}
void backtracking(int i, int j, vector<vector<char>>& board, string&
word, bool& find, vector<vector<bool>>& visited, int pos) {
    if (i < 0 || i >= board.size() || j < 0 || j >= board[0].size())
        return;
    if (visited[i][j] || find || board[i][j] != word[pos])
        return;
    if (pos == word.size() - 1) {
        find = true;
        return;
    }
    visited[i][j] = true;
```

```
    backtracking(i + 1, j, board, word, find, visited, pos + 1);
    backtracking(i - 1, j, board, word, find, visited, pos + 1);
    backtracking(i, j + 1, board, word, find, visited, pos + 1);
    backtracking(i, j - 1, board, word, find, visited, pos + 1);
    visited[i][j] = false;
}
```

N皇后

按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。

给你一个整数 n，返回所有不同的 n 皇后问题的解决方案。每一种解法包含一个不同的 n 皇后问题的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。

示例 1：

> “
> 输入：n = 4
> 输出：[[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]

示例 2：

> “
> 输入：n = 1
> 输出：[["Q"]]

和上一题是类似的，本题也是通过修改状态矩阵来进行回溯。不同的是，我们需要对每一行、列、左斜、右斜建立访问数组，来记录它们是否存在皇后。另外，本题中限制了每一行只有一个皇后，所以不用遍历矩阵中每一个位置，只需遍历每一行即可。

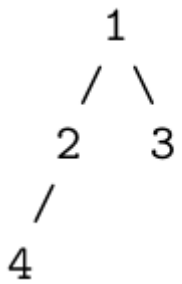
```
vector<vector<string>> solveNQueens(int n) {
    vector<vector<string>> ans;
    if (n == 0)
        return ans;
    vector<string> board(n, string(n, '.'));
    vector<bool> column(n, false), ldiag(2*n-1, false), rdiag(2*n-1, false);
    backtracking(ans, board, column, ldiag, rdiag, 0, n);
    return ans;
}
void backtracking(vector<vector<string>> &ans, vector<string> &board, vector<bool> &column, vector<bool> &ldiag, vector<bool> &rdiag, int row, int n) {
    if (row == n) {
        ans.push_back(board);
        return;
```

```
    }
    for (int i = 0; i < n; ++i) {
        if (column[i] || ldiag[n-row+i-1] || rdiag[row+i])
            continue;
        board[row][i] = 'Q';
        column[i] = ldiag[n-row+i-1] = rdiag[row+i] = true;
        backtracking(ans, board, column, ldiag, rdiag, row+1, n);
        board[row][i] = '.';
        column[i] = ldiag[n-row+i-1] = rdiag[row+i] = false;
    }
}
```

广度优先搜索

广度优先搜索（breadth-first search，BFS）不同与深度优先搜索，它是一层层进行遍历的，因此需要用先入先出的队列而非先入后出的栈进行遍历。由于是按层次进行遍历，广度优先搜索时按照“广”的方向进行遍历的，也常常用来处理最短路径等问题。

考虑如下一颗简单的树。我们从 1 号节点开始遍历，假如遍历顺序是从左子节点到右子节点，那么按照优先向着“广”的方向前进的策略，队列顶端的元素变化过程为 [1]->[2->3]->[4]，其中方括号代表每一层的元素。



这里要注意，深度优先搜索和广度优先搜索都可以处理可达性问题，即从一个节点开始是否能达到另一个节点。

最短的桥

在给定的二维二进制数组 A 中，存在两座岛（岛是由四面相连的 1 形成的一个最大组）现在，我们可以将 0 变为 1，以使两座岛连接起来，变成一座岛。

返回必须翻转的 0 的最小数目（可以保证答案至少是 1）

示例 1：

> “
> 输入：A = [[0,1],[1,0]]
> 输出：1

示例 2：

> “

> 输入：A = [[0,1,0],[0,0,0],[0,0,1]]
> 输出：2

示例 3：

> “
> 输入：A = [[1,1,1,1,1],[1,0,0,0,1],[1,0,1,0,1],[1,0,0,0,1],[1,1,1,1,1]]
> 输出：1

本题实际上是求两个岛屿间的最短距离，因此我们可以先通过任意搜索方法找到其中一个岛屿，然后利用广度优先搜索，查找其与另一个岛屿的最短距离。

```
vector<int> direction{-1, 0, 1, 0, -1};
int shortestBridge(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    queue<pair<int, int>> points;
    bool flipped = false;
    //DFS找到第一个岛屿，标记为2
    for (int i = 0; i < m; i++) {
        if (flipped)
            break;
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 1) {
                dfs(points, grid, m, n, i, j);
                flipped = true;
                break;
            }
        }
    }
    //BFS，一层层向外扩展至找到第二个岛屿
    int x, y;
    int level = 0;
    while (!points.empty()){
        level++;
        //n_points每减一，队列里就排除一个，减到0说明往外扩一层还没有找到，层数加一，继续扩
        int n_points = points.size();
        while (n_points--) {
            auto [r, c] = points.front();
            points.pop();
            for (int k = 0; k < 4; k++) {
                x = r + direction[k];
                y = c + direction[k+1];
                if (x >= 0 && y >= 0 && x < m && y < n) {
                    if (grid[x][y] == 2)
                        continue;
                    if (grid[x][y] == 1)
                        return level;
```

```
                    points.push({x, y});
                    grid[x][y] = 2;
                }
            }
        }
    }
    return 0;
}
void dfs(queue<pair<int, int>>& points, vector<vector<int>>& grid, int m,
int n, int i, int j) {
    if (i < 0 || j < 0 || i == m || j == n || grid[i][j] == 2)
        return;
    //将边界一圈一个个压入队列
    if (grid[i][j] == 0) {
        points.push({i, j});
        return;
    }
    grid[i][j] = 2;
    dfs(points, grid, m, n, i - 1, j);
    dfs(points, grid, m, n, i + 1, j);
    dfs(points, grid, m, n, i, j - 1);
    dfs(points, grid, m, n, i, j + 1);
}
```

单词接龙II

按字典 wordList 完成从单词 beginWord 到单词 endWord 转化，一个表示此过程的转换序列是形式上像 beginWord -> s1 -> s2 -> ... -> sk 这样的单词序列，并满足：

- 每对相邻的单词之间仅有单个字母不同。
- 转换过程中的每个单词 si（1 <= i <= k）必须是字典 wordList 中的单词。注意，beginWord 不必是字典 wordList 中的单词。
- sk == endWord

给你两个单词 beginWord 和 endWord ， 以及一个字典 wordList 。请你找出并返回所有从 beginWord 到 endWord 的最短转换序列，如果不存在这样的转换序列，返回一个空列表。每个序列都应该以单词列表 [beginWord, s1, s2, ..., sk] 的形式返回。

示例 1：

> 输入：beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
> 输出：[["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]

示例 2：

>

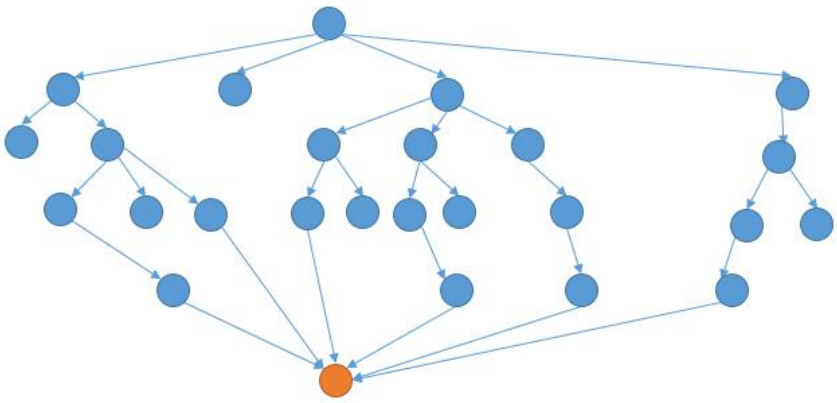
> 输入：beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
> 输出：[]

双向BFS构建节点树，再用DFS找出所有通路。

我们可以把起始字符串、终止字符串、以及单词表里所有的字符串想象成节点。若两个字符串只有一个字符不同，那么它们相连。因为题目需要输出修改次数最少的所有修改方式，因此我们可以使用广度优先搜索，求得起始节点到终止节点的最短距离。

我们同时还使用了一个小技巧：我们并不是直接从起始节点进行广度优先搜索，直到找到终止节点为止；而是从起始节点和终止节点分别进行广度优先搜索，每次只延展当前层节点数最少的那一端，这样我们可以减少搜索的总结点数。

在搜索结束后，我们还需要通过回溯法来重建所有可能的路径。



```
unordered_map<string, vector<string>> tree;//节点图，把所有符合只差一个字符的单词联系起来
vector<vector<string>> ans;//存储结果
vector<vector<string> > findLadders(string beginWord, string endWord, vector<string> & wordList) {
    //特判
    if(wordList.size() == 0 || find(wordList.begin(), wordList.end(), endWord) == wordList.end())
        return {};
    //双向BFS
    swap(beginWord, endWord);//这里交换是为了建立从尾到头的树
    unordered_set<string> bfsFromBegin{beginWord};//从上到下BFS
    unordered_set<string> bfsFromEnd{endWord};//从下到上
    unordered_set<string> dirc(wordList.begin(), wordList.end());//从单词目录中取出所有单词，用于标记是否搜索过
    bool findFlag = false, reverseFlag = false;//上下是否相交，是否转换搜索方向
    while(!bfsFromBegin.empty()){
        unordered_set<string> next;//存放下一层的搜索对象
        for(string s : bfsFromBegin)
            dirc.erase(s);//擦除已经搜索过的
        for(string s1 : bfsFromBegin){
            //寻找只相差一个字符的单词
```

```
            for(int i = 0; i < s1.size(); i++){
                string s2 = s1;
                for(char c = 'a'; c <= 'z'; c++){
                    s2[i] = c;
                    if(dirc.count(s2) == 0)
                        continue;
                    if(bfsFromEnd.count(s2))
                        findFlag = true;
                    else
                        next.insert(s2);
                    reverseFlag ? tree[s2].push_back(s1) :
tree[s1].push_back(s2);
                }
            }
        }
        bfsFromBegin = next;
        //每次只搜索节点次数少的那端
        if(bfsFromBegin.size() > bfsFromEnd.size()){
            reverseFlag = !reverseFlag;
            swap(bfsFromBegin, bfsFromEnd);
        }
        if(findFlag)
            break;
    }
    //DFS寻找所有通路
    vector<string> cur = {beginWord};
    dfs(cur, beginWord, endWord);
    return ans;
}
void dfs(vector<string>& cur, string curWord, string endWord){
    if(curWord == endWord){
        //搜索方向是反着的，要翻转
        reverse(cur.begin(), cur.end());
        ans.push_back(cur);
        reverse(cur.begin(), cur.end());
        return;
    }
    for(string s : tree[curWord]){
        cur.push_back(s);
        dfs(cur, s, endWord);
        cur.pop_back();
    }
}
```

> “
>
> 注意：从前向后DFS会超时，所以从后向前DFS，面向数据编程了属于是（

练习

被围绕的区域

给你一个 m x n 的矩阵 board，由若干字符 'X' 和 'O'，找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

示例 1：

> 输入：board =
> [["X","X","X","X"],
> ["X","O","O","X"],
> ["X","X","O","X"],
> ["X","O","X","X"]]
> 输出：[["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]

示例 2：

> 输入：board = [["X"]]
> 输出：[["X"]]

主要问题在于如何判断是否是被包围的，可以dfs每一个边界元素，把与之相连的标记为'A'，然后遍历矩阵赋值即可。

```
int n, m;
void dfs(vector<vector<char>>& board, int x, int y) {
    if (x < 0 || x >= n || y < 0 || y >= m || board[x][y] != 'O')
        return;
    board[x][y] = 'A';
    dfs(board, x + 1, y);
    dfs(board, x - 1, y);
    dfs(board, x, y + 1);
    dfs(board, x, y - 1);
}
void solve(vector<vector<char>>& board) {
    n = board.size();
    if (n == 0)
        return;
    m = board[0].size();
    for (int i = 0; i < n; i++) {
        dfs(board, i, 0);
        dfs(board, i, m - 1);
    }
    for (int i = 1; i < m - 1; i++) {
        dfs(board, 0, i);
        dfs(board, n - 1, i);
```

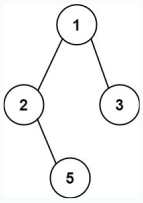
```
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (board[i][j] == 'A')
                board[i][j] = 'O';
            else if (board[i][j] == 'O')
                board[i][j] = 'X';
        }
    }
}
```

二叉树的所有路径

给你一个二叉树的根节点 root，按任意顺序返回所有从根节点到叶子节点的路径。

叶子节点是指没有子节点的节点。

示例 1：

> 输入： 
>
> root = [1,2,3,null,5]
> 输出：["1->2->5","1->3"]

示例 2：

> 输入：root = [1]
> 输出：["1"]

DFS

```
void construct_paths(TreeNode* root, string path, vector<string>& paths)
{
    if (root != nullptr) {
        path += to_string(root->val);
        if (root->left == nullptr && root->right == nullptr)
            paths.push_back(path);
        else {
            path += "->";
            construct_paths(root->left, path, paths);
            construct_paths(root->right, path, paths);
        }
    }
}
```

```cpp
vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> paths;
    construct_paths(root, "", paths);
    return paths;
}
```

全排列II

给定一个可包含重复数字的序列 nums，按任意顺序返回所有不重复的全排列。

示例 1：

> 输入：nums = [1,1,2]
> 输出：
> [[1,1,2],[1,2,1],[2,1,1]]

示例 2：

> 输入：nums = [1,2,3]
> 输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

重复的来源是相同数字的位置可交换，所以只需限制相同数字的相对位置为固定的即可，可以对数组排序，按照数组下标的相对位置。

```cpp
vector<int> vis;
void backtrack(vector<int>& nums, vector<vector<int>>& ans, int idx,
vector<int>& perm) {
    if (idx == nums.size()) {
        ans.emplace_back(perm);
        return;
    }
    for (int i = 0; i < nums.size(); i++) {
        //不能取的两种情况，一是取过了，二是相同数字但位置在前的还未被取
        if (vis[i] || (i > 0 && nums[i] == nums[i - 1] && !vis[i - 1]))
            continue;
        perm.emplace_back(nums[i]);//比push_back效率高
        vis[i] = 1;
        backtrack(nums, ans, idx + 1, perm);
        vis[i] = 0;
        perm.pop_back();
    }
}
vector<vector<int>> permuteUnique(vector<int>& nums) {
    vector<vector<int>> ans;
    vector<int> perm;
    vis.resize(nums.size());
```

```
    sort(nums.begin(), nums.end());
    backtrack(nums, ans, 0, perm);
    return ans;
}
```

组合总和II

给定一个候选人编号的集合 candidates 和一个目标数 target，找出 candidates 中所有可以使数字和为 target 的组合。candidates 中的每个数字在每个组合中只能使用一次。

注意：解集不能包含重复的组合。

示例 1:

> 输入: candidates = [10,1,2,7,6,1,5], target = 8,
> 输出:
> [[1,1,6],[1,2,5],[1,7],[2,6]]

示例 2:

> 输入: candidates = [2,5,2,1,2], target = 5,
> 输出:
> [[1,2,2],[5]]

带重复的回溯 = 排序 + 判断。

```
vector<vector<int>> res;
vector<int> temp;
void backtrack(vector<int>& candidates, int target, int index) {
    if(target == 0) {
        res.push_back(temp);
        return;
    }
    for(int i = index; i < candidates.size() && target-candidates[i] >= 0; i++) {
        //避免重复，只需要避免取的数和上一层回溯拿掉的数相等即可
        if(i > index && candidates[i] == candidates[i-1])
            continue;
        temp.push_back(candidates[i]);
        backtrack(candidates, target-candidates[i], i+1);
        temp.pop_back();
    }
}
vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
    sort(candidates.begin(), candidates.end());
```

```
    backtrack(candidates, target, 0);
    return res;
}
```

解数独

编写一个程序，通过填充空格来解决数独问题。数独的解法需遵循如下规则：

- 数字 1-9 在每一行只能出现一次。
- 数字 1-9 在每一列只能出现一次。
- 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。（请参考示例图）

数独部分空格内已填入了数字，空白格用 '.' 表示。

示例 1：

> 输入：
>
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

> 输出：
>
5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

类似 N 皇后，建立数组标记行、列和九宫格内的数字出现情况，然后回溯法填数字即可。

```
bool line[10][10], column[10][10], block[3][3][10];
bool dfs(vector<vector<char>>& board) {
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            char temp = board[i][j];
            if (temp == '.') {
                for (int k = 1; k <= 9; k++) {
                    if (!line[i][k] && !column[j][k] && !block[i/3][j/3]
[k]) {
                        board[i][j] = k + '0';
                        line[i][k] = true;
                        column[j][k] = true;
                        block[i / 3][j / 3][k] = true;
                        if (dfs(board))
                            return true;
                        board[i][j] = '.';
                        line[i][k] = false;
```

```
                        column[j][k] = false;
                        block[i / 3][j / 3][k] = false;
                    }
                }
                return false;
            }
        }
    }
    return true;
}
void solveSudoku(vector<vector<char>>& board) {
    memset(line, 0, sizeof(line));
    memset(column, 0, sizeof(column));
    memset(block, 0, sizeof(block));
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            char temp = board[i][j] - '0';
            if (temp != '.' - '0') {
                line[i][temp] = true;
                column[j][temp] = true;
                block[i / 3][j / 3][temp] = true;
            }
        }
    }
    dfs(board);
}
```

最小高度树

树是一个无向图，其中任何两个顶点只通过一条路径连接。换句话说，一个任何没有简单环路的连通图都是一棵树。

给你一棵包含 n 个节点的树，标记为 0 到 n - 1。给定数字 n 和一个有 n - 1 条无向边的 edges 列表（每一个边都是一对标签），其中 edges[i] = [ai, bi] 表示树中节点 ai 和 bi 之间存在一条无向边。任意两节点只有一条路径。

可选择树中任何一个节点作为根。当选择节点 x 作为根节点时，设结果树的高度为 h。在所有可能的树中，具有最小高度的树（即，min(h)）被称为最小高度树。

请你找到所有的最小高度树并按任意顺序返回它们的根节点标签列表。

树的高度是指根节点和叶子节点之间最长向下路径上边的数量。

示例 1：

> “
> 输入：n = 4, edges = [[1,0],[1,2],[1,3]]
> 输出：[1]

示例 2：

> 输入：n = 6, edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]
> 输出：[3,4]

暴力解法会超时，这里采用类似拓扑排序（好高级的名词doge)的方法。目标是找到与距离最远的叶节点的距离最小的根节点，如果把树按照最长路径给捋成一条链子，那么很显然应该选取根节点为链子中点的一个或两个节点，如何取到呢，可以BFS逐层删掉叶节点，最后剩下两个或一个时即为所求的根节点。

```
vector<int> findMinHeightTrees(int n, vector<vector<int>>& edges) {
    if (n == 1)
        return {0};
    if (n == 2)
        return {0, 1};
    vector<int> degree(n);   //每个节点对应的度数
    map<int,vector<int>> m;  //邻接表
    vector<int> ans; //结果
    for (int i = 0; i < edges.size(); i++){
        int u=edges[i][0];
        int v=edges[i][1];
        degree[u]++;
        degree[v]++;
        m[u].push_back(v);
        m[v].push_back(u);
    }
    queue<int> q;
    //把叶子节点入队列
    for (int i = 0; i < n; i++)
        if (degree[i] == 1)
            q.push(i);
    //从外向内一层一层剥
    int k = 0;
    while (n - k > 2){
        int sz=q.size();
        for (int i = 0; i < sz; i++){
            int t = q.front();
            degree[t]--;
            q.pop();
            k++;
            //加入t的邻接叶子节点
            for (auto j : m[t]){
                degree[j]--;
                if (degree[j] == 1)
                    q.push(j);
            }
        }
```

```
    }
    for (int i = 0; i < n - k; i++) {
        ans.push_back(q.front());
        q.pop();
    }
    return ans;
}
```

动态规划

算法解释

动态规划（Dynamic Programming, DP）在查找有很多重叠子问题的情况的最优解时有效。它将问题重新组合成子问题。为了避免多次解决这些子问题，它们的结果都逐渐被计算并被保存，从简单的问题直到整个问题都被解决。因此，动态规划保存递归时的结果，因而不会在解决同样的问题时花费时间

通俗一点来讲，动态规划和其它遍历算法（如深/广度优先搜索）都是将原问题拆成多个子问题然后求解，他们之间最本质的区别是，动态规划保存子问题的解，避免重复计算。解决动态规划问题的关键是找到状态转移方程，这样我们可以通过计算和储存子问题的解来求解最终问题。同时，我们也可以对动态规划进行空间压缩，起到节省空间消耗的效果。这一技巧笔者将在之后的题目中介绍。

在一些情况下，动态规划可以看成是带有状态记录（memoization）的优先搜索。状态记录的意思为，如果一个子问题在优先搜索时已经计算过一次，我们可以把它的结果储存下来，之后遍历到该子问题的时候可以直接返回储存的结果。动态规划是自下而上的，即先解决子问题，再解决父问题；而用带有状态记录的优先搜索是自上而下的，即从父问题搜索到子问题，若重复搜索到同一个子问题则进行状态记录，防止重复计算。如果题目需求的是最终状态，那么使用动态规划比较方便；如果题目需要输出所有的路径，那么使用带有状态记录的优先搜索会比较方便。

一维dp

爬楼梯

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

示例 1:

> “
> 输入：n = 2
> 输出：2

示例 2:

> “
> 输入：n = 3
> 输出：3

问题可以分解为两大类，第一类，最后一步爬了两个台阶，第二类，最后一步爬了一个台阶。设 $f(k)$ 为上到 k 阶台阶的总方案数，则状态转移方程为 $f(k) = f(k-1) + f(k-2)$

```cpp
int climbStairs(int n) {
    if (n <= 2)
        return n;
    vector<int> dp(n+1, 0);
    dp[1] = 1;
    dp[2] = 2;
    for (int i = 3; i <= n; i++)
        dp[i] = dp[i - 1] + dp[i - 2];
    return dp[n];
}
```

可以看出，就是斐波那契数列问题，可以进行空间压缩，重复利用存储空间，代码如下：

```cpp
int climbStairs(int n) {
    if (n <= 2)
        return n;
    int pre2 = 1, pre1 = 2, cur;
    for (int i = 2; i < n; i++) {
        cur = pre1 + pre2;
        pre2 = pre1;
        pre1 = cur;
    }
    return cur;
}
```

打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

示例 1：

> “
> 输入：[1,2,3,1]
> 输出：4

示例 2：

> “
> 输入：[2,7,9,3,1]
> 输出：12

设共有 k 个房间，有两种获得最高总金额的可能。

一是偷了第 k 个房间，总金额为前 k-2 个房间的最高总金额加上第 k 个房间的金额；二是没偷第 k 个房间，总金额为前 k-1 个房间的最高总金额，取两者中的较大者。设最高总金额为 $f(k)$，则状态转移方程为 $f(k) = max(f(k-2) + value[k], \quad f(k-1))$.

边界条件为 $f(1) = value[1], \quad f(2) = max(value[1], \quad value[2])$.

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

同样可以进行空间压缩，代码如下：

```cpp
int rob(vector<int>& nums) {
    if (nums.empty())
        return 0;
    int n = nums.size();
    if (n == 1)
        return nums[0];
    int pre2 = 0, pre1 = 0, cur;
    for (int i = 0; i < n; i++) {
        cur = max(pre2 + nums[i], pre1);
        pre2 = pre1;
        pre1 = cur;
    }
    return cur;
}
```

等差数列划分

如果一个数列至少有三个元素，并且任意两个相邻元素之差相同，则称该数列为等差数列。给你一个整数数组 nums，返回数组 nums 中所有为等差数组的子数组个数。

子数组是数组中的一个连续序列。

示例 1：

> 输入：nums = [1,2,3,4]
> 输出：3

示例 2：

> 输入：nums = [1]
> 输出：0

这道题略微特殊，因为要求是等差数列，可以很自然的想到子数组必定满足 num[i]-num[i-1] = num[i-1]-num[i-2]。定义以数字 nums[i] 结尾的等差子数组个数为 dp[i]，最后对 dp 数组求和即可。

```
int numberOfArithmeticSlices(vector<int>& nums) {
    int n = nums.size();
    if (n < 3)
        return 0;
    vector<int> dp(n, 0);
    for (int i = 2; i < n; i++) {
        if (nums[i] - nums[i-1] == nums[i-1] - nums[i-2])
            dp[i] = dp[i-1] + 1;
    }
    return accumulate(dp.begin(), dp.end(), 0);
}
```

二维dp

最小路径和

给定一个包含非负整数的 m x n 网格 grid，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

示例 1：

> 输入：grid = [[1,3,1],[1,5,1],[4,2,1]]
> 输出：7

示例 2：

> 输入：grid = [[1,2,3],[4,5,6]]
> 输出：12

二维dp就建立二维数组，dp[i][j] 表示从左上角到位置 [i,j] 的最小路径和，

状态转移方程为 $dp[i][j] = min(dp[i-1][j],\ \ dp[i][j-1]) + grid[i][j]$，注意边界。

边界条件为 $dp[0][0] = grid[0][0]$.

```cpp
int minPathSum(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<vector<int>> dp(m, vector<int>(n, 0));
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (i == 0 && j == 0)
                dp[i][j] = grid[i][j];
            else if (i == 0)
                dp[i][j] = dp[i][j-1] + grid[i][j];
            else if (j == 0)
                dp[i][j] = dp[i-1][j] + grid[i][j];
            else
                dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j];
        }
    }
    return dp[m-1][n-1];
}
```

同样可以进行空间压缩至一维，因为 dp 矩阵的每一个值只和左边和上面的值相关，当我们遍历到第 i 行第 j 列时，第 j-1 列已经更新过了，现在的 dp[j-1] 代表 dp[i][j-1] 的值；d[j] 待更新，现在的 dp[j] 代表 dp[i-1][j] 的值。代码如下：

```cpp
int minPathSum(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<int> dp(n, 0);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (i == 0 && j == 0)
                dp[j] = grid[i][j];
            else if (i == 0)
                dp[j] = dp[j-1] + grid[i][j];
            else if (j == 0)
                dp[j] = dp[j] + grid[i][j];
            else
                dp[j] = min(dp[j], dp[j-1]) + grid[i][j];
        }
    }
    return dp[n-1];
}
```

“01”矩阵

给定一个由 0 和 1 组成的矩阵 mat，请输出一个大小相同的矩阵，其中每一个格子是 mat 中对应位置元素到最近的 0 的距离。两个相邻元素间的距离为 1。

示例 1：

> “

> 输入：mat = [[0,0,0],[0,1,0],[0,0,0]]
> 输出：[[0,0,0],[0,1,0],[0,0,0]]

示例 2：

> “
> 输入：mat = [[0,0,0],[0,1,0],[1,1,1]]
> 输出：[[0,0,0],[0,1,0],[1,2,1]]

dp[i][j] 表示位置 [i,j] 的元素到最近的 0 的距离，如果限制只能向右下搜索，则状态转移方程为：

$$dp[i][j] = \begin{cases} 1 + min(dp[i-1][j], \quad dp[i][j-1]), & mat[i][j] = 1 \\ 0, & mat[i][j] = 0 \end{cases}$$

左上同理，取最小值即可。

```cpp
vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
    if (matrix.empty())
        return {};
    int n = matrix.size(), m = matrix[0].size();
    // 初始化动态规划的数组，所有的距离值都设置为一个很大的数
    vector<vector<int>> dp(n, vector<int>(m, INT_MAX - 1));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (matrix[i][j] == 0)
                dp[i][j] = 0;
            else {
                //这样写就不用单独写特例了
                if (j > 0)
                    dp[i][j] = min(dp[i][j], dp[i][j-1] + 1);
                if (i > 0)
                    dp[i][j] = min(dp[i][j], dp[i-1][j] + 1);
            }
        }
    }
    for (int i = n - 1; i >= 0; i--) {
        for (int j = m - 1; j >= 0; j--) {
            if (matrix[i][j] != 0) {
                if (j < m - 1)
                    dp[i][j] = min(dp[i][j], dp[i][j+1] + 1);
                if (i < n - 1)
                    dp[i][j] = min(dp[i][j], dp[i+1][j] + 1);
            }
        }
    }
    return dp;
}
```

最大正方形

在一个由 '0' 和 '1' 组成的二维矩阵内，找到只包含 '1' 的最大正方形，并返回其面积。

示例 1：

> 输入：matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
> 输出：4

示例 2：

> 输入：matrix = [["0","1"],["1","0"]]
> 输出：1

示例 3：

> 输入：matrix = [["0"]]
> 输出：0

dp[i][j] 为以 [i,j] 为右下角的正方形的最大边长，若 matrix[i][j] = 0，则 dp[i][j] = 0；否则我们假设 dp[i][j] = k，其充分条件为 dp[i-1][j-1]、dp[i][j-1] 和 dp[i-1][j] 的值必须都不小于 k−1，否则 [i,j] 位置不可以构成一个面积为 k^2 的正方形。同理，如果这三个值中的最小值为 k−1，则 [i,j] 位置一定且最大可以构成一个面积为 k^2 的正方形。

```cpp
int maximalSquare(vector<vector<char>>& matrix) {
    if (matrix.empty() || matrix[0].empty())
        return 0;
    int m = matrix.size(), n = matrix[0].size(), max_side = 0;
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (matrix[i-1][j-1] == '1')
                dp[i][j] = min(dp[i-1][j-1], min(dp[i][j-1], dp[i-1][j]))
+ 1;
            max_side = max(max_side, dp[i][j]);
        }
    }
    return max_side * max_side;
}
```

分割类型题

完全平方数

完全平方数是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

给你一个整数 n，返回和为 n 的完全平方数的最少数量。

示例 1：

> 输入：n = 12
> 输出：3

示例 2：

> 输入：n = 13
> 输出：2

对于分割类型题，动态规划的状态转移方程通常并不依赖相邻的位置，而是依赖于满足分割条件的位置。我们定义一个一维矩阵 dp，其中 dp[i] 表示数字 i 最少可以由几个完全平方数相加构成。在本题中，位置 i 只依赖 $i-k^2$ 的位置，如 i - 1、i - 4、i - 9 等等，才能满足完全平方分割的条件。因此，状态转移方程为：

$$dp[i] = 1 + min(dp[i-1], dp[i-4], dp[i-9] \cdots\cdots)$$

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

解码方法

一条包含字母 A-Z 的消息通过以下映射进行了编码：

> 'A' -> "1"
> 'B' -> "2"
> ...
> 'Z' -> "26"

要解码已编码的消息，所有数字必须基于上述映射的方法，反向映射回字母（可能有多种方法）。例如，"11106" 可以映射为：

>

> "AAJF" ，将消息分组为 (1 1 10 6)
> "KJF" ，将消息分组为 (11 10 6)

注意，消息不能分组为 (1 11 06)，因为 "06" 不能映射为 "F"，这是由于 "6" 和 "06" 在映射中并不等价。

给你一个只含数字的非空字符串 s，请计算并返回解码方法的总数 。

题目数据保证答案肯定是一个 32 位的整数。

示例 1：

> “
> 输入：s = "12"
> 输出：2

示例 2：

> “
> 输入：s = "226"
> 输出：3

示例 3：

> “
> 输入：s = "0"
> 输出：0

就是要把一个大数字分割为一堆 1-26 的数字，设 dp[i] 表示字符串前 i 个字符的解码方法数，则状态转移方程为：

$$dp[i] = \begin{cases} dp[i-1], & s[i] \neq 0 \\ dp[i-2], & s[i-1] \neq 0 \quad \& \quad s[i-1] * 10 + s[i] <= 26 \end{cases}$$

重叠部分叠加即可，注意数组下标从 0 开始。边界条件为 dp[0] = 1。

```
int numDecodings(string s) {
    int n = s.length();
    if (n == 0 || s[0] == '0')
        return 0;
    vector<int> dp(n+1, 0);
    dp[0] = 1;
    for (int i = 1; i <= n; i++) {
        if (s[i-1] != '0')
            dp[i] += dp[i-1];
        if (i > 1 && s[i-2] != '0' && (s[i-2] - '0') * 10 + (s[i-1] -
'0') <= 26)
            dp[i] += dp[i-2];
    }
    return dp[n];
```

```
}
```

单词拆分

给你一个字符串 s 和一个字符串列表 wordDict 作为字典。请你判断是否可以利用字典中出现的单词拼接出 s。s 和 wordDict[i] 仅有小写英文字母组成，wordDict 中的所有字符串互不相同。

注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。

示例 1：

> 输入: s = "leetcode", wordDict = ["leet", "code"]
> 输出: true

示例 2：

> 输入: s = "applepenapple", wordDict = ["apple", "pen"]
> 输出: true

示例 3：

> 输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
> 输出: false

能否拼接，可以转化为能否拆分，类似于完全平方数分割问题，这道题的分割条件由集合内的字符串决定，因此在考虑每个分割位置时，需要遍历字符串集合，以确定当前位置是否可以成功分割。注意对于位置 0，需要初始化值为真。

```cpp
bool wordBreak(string s, vector<string>& wordDict) {
    int n = s.length();
    vector<bool> dp(n + 1, false);
    dp[0] = true;
    for (int i = 1; i <= n; i++) {
        for (const string & word: wordDict) {
            int len = word.length();
            if (i >= len && s.substr(i - len, len) == word)
                dp[i] = dp[i] || dp[i - len];
        }
    }
    return dp[n];
}
```

子序列问题

最长递增子序列

给你一个整数数组 nums，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。

示例 1：

> “
> 输入：nums = [10,9,2,5,3,7,101,18]
> 输出：4

示例 2：

> “
> 输入：nums = [0,1,0,3,2,3]
> 输出：4

示例 3：

> “
> 输入：nums = [7,7,7,7,7,7,7]
> 输出：1

对于子序列问题，第一种动态规划方法是，定义一个 dp 数组，其中 dp[i] 表示以 i 结尾的符合某性质的子序列的个数。在处理好每个位置后，统计一遍各个位置的结果即可得到题目要求的结果。

在本题中，dp[i] 可以表示以 i 结尾的、最长递增子序列长度。对于每一个位置 i，如果其之前的某个位置 j 所对应的数字小于位置 i 所对应的数字，则我们可以获得一个以 i 结尾的、长度为 dp[j]+1 的子序列。为了遍历所有情况，我们需要 i 和 j 进行两层循环，其时间复杂度为 O(n^2)。

```cpp
int lengthOfLIS(vector<int>& nums) {
    int max_length = 0, n = nums.size();
    if (n <= 1)
        return n;
    vector<int> dp(n, 1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j])
                dp[i] = max(dp[i], dp[j] + 1);
        }
        max_length = max(max_length, dp[i]);
    }
    return max_length;
}
```

本题还可以使用二分查找将时间复杂度降低为 O(n log n)。我们定义一个 dp 数组，其中 dp[k] 存储长度为 k+1 的最长递增子序列的最后一个数字。我们遍历每一个位置 i，如果其对应的数字大于 dp 数组中所有数字的值，那么我们把它放在 dp 数组尾部，表示最长递增子序列长度加 1；如果我们发现这个数字在 dp 数组中比数字 a 大、比数字 b 小，则我们将 b 更新为此数字，使得之后构成递增序列的可能性增大。以这种方式维护的 dp 数组永远是递增的，因此可以用二分查找加速搜索。

```cpp
int lengthOfLIS(vector<int>& nums) {
    int n = nums.size();
    if (n <= 1)
        return n;
    vector<int> dp;
    dp.push_back(nums[0]);
    for (int i = 1; i < n; i++) {
        if (dp.back() < nums[i])
            dp.push_back(nums[i]);
        else {
            auto itr = lower_bound(dp.begin(), dp.end(), nums[i]);
            *itr = nums[i];
        }
    }
    return dp.size();
}
```

最长公共子序列

给定两个仅由小写英文字母组成的字符串 text1 和 text2，返回这两个字符串的最长公共子序列的长度。如果不存在公共子序列，返回 0。

一个字符串的子序列是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。

两个字符串的公共子序列是这两个字符串所共同拥有的子序列。

示例 1：

> “
> 输入：text1 = "abcde", text2 = "ace"
> 输出：3

示例 2：

> “
> 输入：text1 = "abc", text2 = "abc"
> 输出：3

示例 3：

> “
> 输入：text1 = "abc", text2 = "def"
> 输出：0

对于子序列问题，第二种动态规划方法是，定义一个 dp 数组，其中 dp[i] 表示到位置 i 为止的子序列的性质，并不必须以 i 结尾。这样 dp 数组的最后一位结果即为题目所求，不需要再对每个位置进行统计。

在本题中，我们可以建立一个二维数组 dp，其中 dp[i][j] 表示到第一个字符串位置 i 为止、到第二个字符串位置 j 为止、最长的公共子序列长度。这样一来我们就可以很方便地分情况讨论这两个位置对应的字母相同与不同的情况了。

```
int longestCommonSubsequence(string text1, string text2) {
    int m = text1.length(), n = text2.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (text1[i-1] == text2[j-1])
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[m][n];
}
```

背包问题

背包问题是一种组合优化的 NP 完全问题：有 N 个物品和容量为 W 的背包，每个物品都有自己的体积 w 和价值 v，求拿哪些物品可以使得背包所装下物品的总价值最大。如果限定每种物品只能选择 0 个或 1 个，则问题称为 0-1 背包问题；如果不限定每种物品的数量，则问题称为无界背包问题或完全背包问题。

h5 1.“0-1”背包问题

我们可以用动态规划来解决背包问题。以 0-1 背包问题为例。我们可以定义一个二维数组 dp 存储最大价值，其中 dp[i][j] 表示前 i 件物品体积不超过 j 的情况下能达到的最大价值。在我们遍历到第 i 件物品时，在当前背包总容量为 j 的情况下，如果我们不将物品 i 放入背包，那么 dp[i][j] = dp[i-1][j]，即前 i 个物品的最大价值等于只取前 i-1 个物品时的最大价值；如果我们将物品 i 放入背包，假设第 i 件物品体积为 w，价值为 v，那么我们得到 dp[i][j] = dp[i-1][j-w] + v。我们只需在遍历过程中对这两种情况取最大值即可，总时间复杂度和空间复杂度都为 O(NW)。

```
dp[i][j] = max(dp[i-1][j], dp[i-1][j-w] + v)
```

转化为代码语言如下：

```cpp
int knapsack(vector<int> weights, vector<int> values, int N, int W) {
    vector<vector<int>> dp(N + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= N; i++) {
        int w = weights[i-1], v = values[i-1];
        for (int j = 1; j <= W; j++) {
            if (j >= w)
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-w] + v);
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    return dp[N][W];
}
```

我们可以进一步对 0-1 背包进行空间优化，将空间复杂度降低为 O(W)。从状态转移方程可以看出，考虑第 i 个物品时，只与上一行有关，因此我们可以去掉 dp 矩阵的第一个维度，在考虑物品 i 时变成 dp[j] = max(dp[j], dp[j-w] + v)。这里要注意我们在遍历每一行的时候必须**逆向遍历**，这样才能够调用上一行物品 i-1 时 dp[j-w] 的值；若按照从左往右的顺序进行正向遍历，则 dp[j-w] 的值在遍历到 j 之前就已经被更新成物品 i 的值了。

```cpp
int knapsack(vector<int> weights, vector<int> values, int N, int W) {
    vector<int> dp(W + 1, 0);
    for (int i = 1; i <= N; i++) {
        int w = weights[i-1], v = values[i-1];
        for (int j = W; j >= w; j--)
            dp[j] = max(dp[j], dp[j-w] + v);
    }
    return dp[W];
}
```

h5 2.完全背包问题

在完全背包问题中，一个物品可以拿多次。假设物品体积均为 2，假设我们遍历到物品 i = 2 且其体积为 w = 2，价值为 v = 3；对于背包容量 j = 5，最多只能装下 2 个该物品。那么我们的状态转移方程就变成了 dp[2][5] = max(dp[1][5], dp[1][3] + 3, dp[1][1] + 6)。如果采用这种方法，假设背包容量无穷大而物体的体积无穷小，我们这里的比较次数也会趋近于无穷大，远超 O(NW) 的时间复杂度。

怎么解决这个问题呢？我们发现在 dp[2][3] 的时候我们其实已经考虑了 dp[1][3] 和 dp[2][1] 的情况，而在 dp[2][1] 也已经考虑了 dp[1][1] 的情况。因此，对于拿多个物品的情况，我们只需考虑 dp[2][3] 即可，即 dp[2][5] = max(dp[1][5], dp[2][3] + 3)。这样，我们就得到了完全背包问题的状态转移方程：

```
dp[i][j] = max(dp[i-1][j], dp[i][j-w] + v)
```

其与 0-1 背包问题的差别仅仅是把状态转移方程中的第二个 i-1 变成了 i。

```cpp
int knapsack(vector<int> weights, vector<int> values, int N, int W) {
    vector<vector<int>> dp(N + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= N; i++) {
        int w = weights[i-1], v = values[i-1];
        for (int j = 1; j <= W; j++) {
            if (j >= w)
                dp[i][j] = max(dp[i-1][j], dp[i][j-w] + v);
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    return dp[N][W];
}
```

同样的，我们也可以利用空间压缩将时间复杂度降低为 O(W)。这里要注意我们在遍历每一行的时候必须**正向遍历**，因为我们需要利用当前物品在第 j-w 列的信息。

```cpp
int knapsack(vector<int> weights, vector<int> values, int N, int W) {
    vector<int> dp(W + 1, 0);
    for (int i = 1; i <= N; i++) {
        int w = weights[i-1], v = values[i-1];
        for (int j = w; j <= W; j++)
            dp[j] = max(dp[j], dp[j-w] + v);
    }
    return dp[W];
}
```

分割等和子集

给你一个只包含正整数的非空数组 nums。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

示例 1：

> “
> 输入：nums = [1,5,11,5]
> 输出：true

示例 2：

> “
> 输入：nums = [1,2,3,5]
> 输出：false

等价于“0-1”背包问题，输出是否可以取出总和正好为 sum/2 的数字，把数字大小看做体积，不用考虑价值，dp[i][j] 表示是否能正好取出体积为 j 的物品

```cpp
bool canPartition(vector<int>& nums) {
```

```
    int sum = accumulate(nums.begin(), nums.end(), 0);
    if (sum % 2)
        return false;
    int target = sum / 2, n = nums.size();
    vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));
    dp[0][0] = true;
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= target; j++) {
            if (j < nums[i - 1])
                dp[i][j] = dp[i-1][j];
            else
                dp[i][j] = dp[i-1][j] || dp[i-1][j - nums[i-1]];
        }
    }
    return dp[n][target];
}
```

空间压缩如下：

```
bool canPartition(vector<int>& nums) {
    int sum = accumulate(nums.begin(), nums.end(), 0);
    if (sum % 2)
        return false;
    int target = sum / 2, n = nums.size();
    vector<bool> dp(target + 1, false);
    dp[0] = true;
    for (int i = 1; i <= n; i++) {
        for (int j = target; j >= nums[i-1]; j--) {
            dp[j] = dp[j] || dp[j-nums[i-1]];
        }
    }
    return dp[target];
}
```

一和零

给你一个二进制字符串数组 strs 和两个整数 m 和 n。请你找出并返回 strs 的最大子集的长度，该子集最多有 m 个 0 和 n 个 1。如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的子集 。

示例 1：

> 输入：strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3
> 输出：4

示例 2：

> “
> 输入：strs = ["10", "0", "1"], m = 1, n = 1
> 输出：2

这里有两个不同维度的体积，dp[i][j][k] 表示在遍历到第 i 个物品时，在 0 的个数小于等于 m，1 的个数小于等于 n 时，所能取到的最大子集长度，则状态转移方程为：

$$dp[i][j][k] = \begin{cases} dp[i-1][j][k] \\ dp[i-1][j-num0][k-num1]+1 \end{cases}$$

```
pair<int, int> count(const string & s){
    int count0 = s.length(), count1 = 0;
    for (const char & c: s) {
        if (c == '1') {
            count1++;
            count0--;
        }
    }
    return make_pair(count0, count1);
}
int findMaxForm(vector<string>& strs, int m, int n) {
    int length = strs.size();
    vector<vector<vector<int>>> dp(length + 1, vector<vector<int>>(m + 1, vector<int>(n + 1)));
    for (int i = 1; i <= length; i++) {
        auto [zeros, ones] = count(strs[i-1]);
        for (int j = 0; j <= m; j++) {
            for (int k = 0; k <= n; k++) {
                if (j < zeros || k < ones)
                    dp[i][j][k] = dp[i-1][j][k];
                else
                    dp[i][j][k] = max(dp[i-1][j][k], dp[i-1][j-zeros][k-ones] + 1);
            }
        }
    }
    return dp[length][m][n];
}
```

零钱兑换

给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。计算并返回可以凑成总金额所需的 最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

你可以认为每种硬币的数量是无限的。

示例 1：

> 输入：coins = [1, 2, 5], amount = 11
> 输出：3

示例 2：

> 输入：coins = [2], amount = 3
> 输出：-1

示例 3：

> 输入：coins = [1], amount = 0
> 输出：0

完全背包问题，dp[i] 表示总金额为 i 时的最少硬币个数，则状态转移方程为：

$$dp[i] = \min_{j=0,1,\cdots} dp[i - c_j] + 1$$

因为遍历 j 时会实时更新 dp[i] 的值，所以方程变为：

$$dp[i] = min(dp[i], dp[i - current\_coin] + 1)$$

为了避免dp[i]刚开始就被取到，初始值取为amount + 1，同时也可以判断最后是否输出-1。

```
int coinChange(vector<int>& coins, int amount) {
    if (coin.empty())
        return -1;
    vector<int> dp(amount + 1, amount + 1);
    dp[0] = 0;
    for (int i = 1; i <= amount; i++) {
        for (int& coin : coins) {
            if (i >= coin)
                dp[i] = min(dp[i], dp[i-coin] + 1);
        }
    }
    return (dp[amount] == amount + 1) ? -1 : dp[amount]
}
```

字符串编辑

编辑距离

给你两个单词 word1 和 word2，请返回将 word1 转换成 word2 所使用的最少操作数。你可以对一个单词进行三种操作：插入一个字符，删除一个字符，替换一个字符。

示例 1：

> 输入：word1 = "horse", word2 = "ros"
> 输出：3

示例 2：

> 输入：word1 = "intention", word2 = "execution"
> 输出：5

类似于题目 1143，我们使用一个二维数组 dp[i][j]，表示将第一个字符串到位置 i 为止，和第二个字符串到位置 j 为止，最多需要几步编辑。当第 i 位和第 j 位对应的字符相同时，dp[i][j] 等于 dp[i-1][j-1]；当二者对应的字符不同时，修改的消耗是 dp[i-1][j-1] + 1，插入 i 位置/删除 j 位置的消耗是 dp[i][j-1] + 1，插入 j 位置/删除 i 位置的消耗是 dp[i-1][j] + 1。边界条件为 dp[0][j] = j，dp[i][0] = i

```
int minDistance(string word1, string word2) {
    int m = word1.length(), n = word2.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0)
                dp[i][j] = j;
            else if (j == 0)
                dp[i][j] = i;
            else
                dp[i][j] = min(dp[i-1][j-1] + ((word1[i-1] == word2[j-1]) ? 0 : 1), min(dp[i][j-1] + 1, dp[i-1][j] + 1));
        }
    }
    return dp[m][n];
}
```

只有两个键的键盘

最初记事本上只有一个字符 'A'。你每次可以对这个记事本进行两种操作：

> Copy All（复制全部）：复制这个记事本中的所有字符（不允许仅复制部分字符）。
> Paste（粘贴）：粘贴上一次复制的字符。

给你一个数字 n，你需要使用最少的操作次数，在记事本上输出恰好 n 个 'A'。返回能够打印出 n 个 'A' 的最少操作次数。

示例 1：

> 输入：3
> 输出：3

示例 2：

> 输入：1
> 输出：0

不同于以往通过加减实现的动态规划，这里需要乘除法来计算位置，因为粘贴操作是倍数增加的。设 dp[i] 表示得到 i 个 'A' 的最少操作数，要得到 i 个，对于 i 的因子 j，从 j 个到 i 个最少操作次数等价于 1 到 i/j，即 dp[i/j]，从 1 到 j 最少操作次数为 dp[j]，所以状态转移方程为：

$$dp[i] = dp[j] + dp[i/j]$$

若 i 为素数，则只能通过一次复制，若干次粘贴得到，故边界条件为 dp[i] = i。

```
int minSteps(int n) {
    vector<int> dp(n + 1);
    for (int i = 2; i <= n; i++) {
        dp[i] = i;
        for (int j = 2; j * j <= i; j++) {
            if (i % j == 0) {
                dp[i] = dp[j] + dp[i/j];
                break;
            }
        }
    }
    return dp[n];
}
```

正则表达式匹配

给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。

> '.' 匹配任意单个字符
> '*' 匹配零个或多个前面的那一个元素

所谓匹配，是要涵盖整个字符串 s 的，而不是部分字符串。

示例 1：

> 输入：s = "aa", p = "a"
> 输出：false

示例 2:

> Input: s = "aab", p = "c*a*b"
> Output: true
> 我们可以重复 c 零次，重复 a 两次。

示例 3:

> 输入：s = "ab", p = ".*"
> 输出：true

提示:

- s 只包含从 a-z 的小写字母。
- p 只包含从 a-z 的小写字母，以及字符 . 和 *。
- 保证每次出现字符 * 时，前面都匹配到有效的字符

我们可以使用一个二维数组 dp，其中 dp[i][j] 表示以 i 截止的字符串是否可以被以 j 截止的正则表达式匹配。根据正则表达式的不同情况，即字符、星号，点号，我们可以分情况讨论来更新 dp 数组。状态转移方程如下:

$$dp[i][j] = \begin{cases} p[j]\text{为小写字母} \begin{cases} dp[i-1][j-1], & s[i]=p[j] \\ false, & s[i]\neq p[j] \end{cases} \\ p[j]='.', \quad dp[i-1][j-1] \\ p[j]='*' \begin{cases} dp[i][j-2], & s[i]\neq p[j-1] \\ dp[i-1][j] \quad or \quad dp[i][j-2], & s[i]=p[j-1], (\text{包含} p[j-1]='.') \end{cases} \end{cases}$$

其中 p[j] = '*' 时的情况比较复杂，本质上只有两种操作方式:

- 匹配一次后继续向前匹配，即 dp[i-1][j]，s[i]=s[i-1]=…=p[j-1]的情况。
- 匹配0次，扔掉p[j-1]和'*'，继续比较，即 dp[i][j-2]。

代码如下，注意数组下标。

```
bool isMatch(string s, string p) {
    int m = s.size(), n = p.size(), i, j;
    vector<vector<int>> dp(m+1, vector<int>(n+1, false));
    dp[0][0] = true;//空串能够匹配
    for(i = 0; i <= m; i++) {
        for(j = 1; j <= n; j++) {
            if(p[j-1] == '*') {
                dp[i][j] |= dp[i][j-2];//*匹配0次前面的字符
                if(match(s,p,i,j-1)) //s第i个和p的第j-1个可以匹配，*匹配再多匹配一次i字符
                    dp[i][j] |= dp[i-1][j];
            }
            else {
                if(match(s,p,i,j))//必须是i、j能够匹配
```

```
                    dp[i][j] |= dp[i-1][j-1];
                }
            }
        }
        return dp[m][n];
}
bool match(string &s, string &p, int i, int j)  {   //第i,j个字符能匹配
    return i>0 && (p[j-1] == '.' || p[j-1] == s[i-1]);
}
```

股票交易

买卖股票的最佳时期

给定一个数组 prices，它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。

你只能选择某一天买入这只股票，并选择在未来的某一个不同的日子卖出该股票。设计一个算法来计算你所能获取的最大利润。返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0。

示例 1:

> “
> 输入：[7,1,5,3,6,4]
> 输出：5

示例 2:

> “
> 输入：prices = [7,6,4,3,1]
> 输出：0

我们可以遍历一遍数组，在每一个位置 i 时，记录 i 位置之前所有价格中的最低价格，然后将当前的价格作为售出价格，查看当前收益是不是最大收益即可。

```
int maxProfit(vector<int>& prices) {
    int n = prices.size();
    vector<int> dp(n + 1);
    dp[1] = prices[0];
    int max = 0;
    for (int i = 1; i < n; i++) {
        if (prices[i] < dp[i])
            dp[i+1] = prices[i];
        else {
            dp[i+1] = dp[i];
            if (prices[i] - dp[i+1] > max)
                max = prices[i] - dp[i+1];
        }
```

```
    }
    return max;
}
```

> 实际上记录最低价用一个变量即可，这里只是为了DP而DP

买卖股票的最佳时机IV

给定一个整数数组 prices，它的第 i 个元素 prices[i] 是一支给定的股票在第 i 天的价格。设计一个算法来计算你所能获取的最大利润。你最多可以完成 k 笔交易。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1：

> 输入：k = 2, prices = [2,4,1]
> 输出：2

示例 2：

> 输入：k = 2, prices = [3,2,6,5,0,3]
> 输出：7

如果 k 大于总天数的一半，那么我们一旦发现可以赚钱就可以进行买卖；这里一半的原因是因为当天股价是不变的，因此一次买卖需要两天。如果 k 小于总天数，我们用 buy[i][j] 表示对于数组 prices[0..i] 中的价格而言，进行恰好 j 笔交易，并且当前手上持有一支股票，这种情况下的最大利润；用 sell[i][j] 表示恰好进行 j 笔交易，并且当前手上不持有股票，这种情况下的最大利润。

> 为了方便分析，买入不算交易，卖出才算一次交易

那么我们可以对状态转移方程进行推导。对于 buy[i][j]，我们考虑当前手上持有的股票是否是在第 i 天买入的。可以得到状态转移方程：

$$buy[i][j] = max(buy[i-1][j], sell[i-1][j] - price[i])$$

同理对于 sell[i][j]，我们可以得到状态转移方程：

$$sell[i][j] = max(sell[i-1][j], buy[i-1][j-1] + price[i])$$

在上述的状态转移方程中，确定边界条件是非常重要的步骤。我们可以考虑将所有的 buy[0][0..k] 以及 sell[0][0..k] 设置为边界。

对于 buy[0][0..k]，由于只有 prices[0] 唯一的股价，因此我们不可能进行过任何交易，那么我们可以将所有的 buy[0][1..k] 设置为一个非常小的值，表示不合法的状态。而对于 buy[0][0]，它的值为 −prices[0]，即「我们在第 0 天以 prices[0] 的价格买入股票」是唯一满足手上持有股票的方法。

同理我们可以将所有的 sell[0][1..k] 设置为一个非常小的值，表示不合法的状态。而对于 sell[0][0]，它的值为 0，即「我们在第 0 天不做任何事」是唯一满足手上不持有股票的方法。

在设置完边界之后，我们就可以使用二重循环，在 i∈[1,n),j∈[0,k] 的范围内进行状态转移。需要注意的是，sell[i][j] 的状态转移方程中包含 buy[i−1][j−1]，在 j=0 时其表示不合法的状态，因此在 j=0 时，我们无需对 sell[i][j] 进行转移，让其保持值为 0 即可。

> “
> 注意，不一定交易次数多了就利润高

```
int maxProfit(int k, vector<int>& prices) {
    int days = prices.size();
    if (days < 2)
        return 0;
    if (k * 2 >= days)
        return maxProfitUnlimited(prices);
    vector<int> buy(k + 1, INT_MIN), sell(k + 1, INT_MIN/2);
    buy[0] = -prices[0];
    sell[0] = 0;
    for (int i = 1; i < days; i++) {
        for (int j = 0; j <= k; j++) {
            buy[j] = max(buy[j], sell[j] - prices[i]);
            if (j > 0)
                sell[j] = max(sell[j], buy[j-1] + prices[i]);
        }
    }
    return *max_element(sell.begin(), sell.end());
}
int maxProfitUnlimited(vector<int> prices) {
    int maxProfit = 0;
    for (int i = 1; i < prices.size(); i++) {
        if (prices[i] > prices[i-1])
            maxProfit += prices[i] - prices[i-1];
    }
    return maxProfit;
}
```

买卖股票之含冷冻期

给定一个整数数组 prices，其中第 prices[i] 表示第 i 天的股票价格。设计一个算法计算出最大利润。

在满足以下约束条件下，你可以尽可能地完成更多的交易：卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

> “
> 输入: prices = [1,2,3,0,2]
> 输出: 3

示例 2:

> “
> 输入: prices = [1]
> 输出: 0

考虑第 i 天结束后的情况，有三种状态:

- 手里有一支股票，最大收益用 dp[i][0] 表示
- 手里没有股票，今天刚卖掉，明天处于冷冻期，最大收益用 dp[i][1] 表示
- 手里没有股票，早就卖掉了，明天不是冷冻期，最大收益用 dp[i][2] 表示

则状态转移方程为:

$$
\begin{aligned}
dp[i][0] &= max(dp[i-1][0], dp[i-1][2] - prices[i]) \\
dp[i][1] &= dp[i-1][0] + prices[i] \\
dp[i][2] &= max(dp[i-1][1], dp[i-1][2])
\end{aligned}
$$

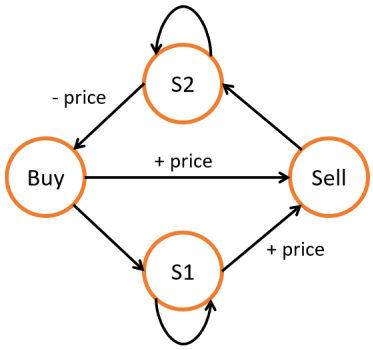
因为最后一天持有股票显然不是最大收益，所以最终答案为 max(dp[n][1], dp[n][2]);

考虑边界条件:

$$
\begin{aligned}
dp[1][0] &= -prices[0] \\
dp[1][1] &= 0 \\
dp[1][2] &= 0
\end{aligned}
$$

```cpp
int maxProfit(vector<int>& prices) {
    if (prices.empty())
        return 0;
    int n = prices.size();
    vector<vector<int>> dp(n+1, vector<int>(3));
    dp[1][0] = -prices[0];
    dp[1][1] = 0;
    dp[1][2] = 0;
    for (int i = 2; i <= n; i++) {
        dp[i][0] = max(dp[i-1][0], dp[i-1][2] - prices[i-1]);
        dp[i][1] = dp[i-1][0] + prices[i-1];
        dp[i][2] = max(dp[i-1][1], dp[i-1][2]);
    }
    return max(dp[n][1], dp[n][2]);
}
```

我们也可以使用状态机来解决这类复杂的状态转移问题，通过建立多个状态以及它们的转移方式，我们可以很容易地推导出各个状态的转移方程。如图所示，我们可以建立四个状态来表示带有冷却的股票交易，以及它们的之间的转移方式。




```cpp
int maxProfit(vector<int>& prices) {
    int n = prices.size();
    if (n == 0)
        return 0;
    vector<int> buy(n), sell(n), s1(n), s2(n);
    s1[0] = buy[0] = -prices[0];
    sell[0] = s2[0] = 0;
    for (int i = 1; i < n; i++) {
        buy[i] = s2[i-1] - prices[i];
        s1[i] = max(buy[i-1], s1[i-1]);
        sell[i] = max(buy[i-1], s1[i-1]) + prices[i];
        s2[i] = max(s2[i-1], sell[i-1]);
    }
    return max(sell[n-1], s2[n-1]);
}
```

练习

打家劫舍II

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，今晚能够偷窃到的最高金额。

示例 1:

> “
> 输入：nums = [2,3,2]
> 输出：3

示例 2:

> 输入：nums = [1,2,3,1]
> 输出：4

示例 3：

> 输入：nums = [1,2,3]
> 输出：3

与之前的区别在于若偷了第一间，则偷窃范围为1 ~ n-1，若没偷第一间，则偷窃范围为2 ~ n

```
int rob(vector<int>& nums) {
    if (nums.empty())
        return 0;
    int n = nums.size();
    if (n == 1)
        return nums[0];
    if (n == 2)
        return max(nums[0], nums[1]);
    vector<int> dp1(n, 0), dp2(n, 0);
    dp1[1] = nums[0];
    dp2[1] = nums[1];
    for (int i = 2; i < n; i++) {
        dp1[i] = max(dp1[i-2] + nums[i-1], dp1[i-1]);
        dp2[i] = max(dp2[i-2] + nums[i], dp2[i-1]);
    }
    return max(dp1[n-1], dp2[n-1]);
}
```

最大子数组和

给你一个整数数组 nums，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

子数组是数组中的一个连续部分。

示例 1：

> 输入：nums = [-2,1,-3,4,-1,2,1,-5,4]
> 输出：6

示例 2：

>

> 输入：nums = [1]
> 输出：1

示例 3：

> “
> 输入：nums = [5,4,-1,7,8]
> 输出：23

dp[i] 表示以 nums[i] 结尾的连续子数组的最大和，则
$dp[i] = max(dp[i-1] + nums[i], nums[i])$

```
int maxSubArray(vector<int>& nums) {
    int n = nums.size();
    vector<int> dp(n);
    dp[0] = nums[0];
    for (int i = 1; i < n; i++)
        dp[i] = max(dp[i-1] + nums[i], nums[i]);
    return *max_element(dp.begin(), dp.end());
}
```

可以将一维空间压缩为常量，考虑到 dp[i] 只和 dp[i-1] 相关，于是我们可以只用一个变量 pre 来维护对于当前 dp[i] 的 dp[i-1] 的值是多少，从而让空间复杂度降低到 O(1)

```
int maxSubArray(vector<int>& nums) {
    int n = nums.size();
    int pre = nums[0], cur, Max = nums[0];
    for (int i = 1; i < n; i++) {
        cur = max(pre + nums[i], nums[i]);
        pre = cur;
        if (cur > Max)
            Max = cur;
    }
    return Max;
}
```

整数拆分

给定一个正整数 n，将其拆分为 k 个正整数的和（k >= 2），并使这些整数的乘积最大化。返回你可以获得的最大乘积。

示例 1:

> “
> 输入: n = 2
> 输出: 1

示例 2:

> 输入: n = 10
> 输出: 36

设 dp[i] 表示将 i 拆分后的最大乘积，假设将 i 拆分为 j 和 i-j，或者继续拆分，取较大值即可，即 $dp[i] = \underset{0<j<i}{max}(j \times (i-j), j \times dp[i-j])$

```
int integerBreak(int n) {
    vector <int> dp(n + 1);
    dp[0] = dp[1] = 0;
    for (int i = 2; i <= n; i++) {
        int Max = 0;
        for (int j = 1; j < i; j++)
            Max = max(Max, max(j * (i - j), j * dp[i - j]));
        dp[i] = Max;
    }
    return dp[n];
}
```

两个字符串的删除操作

给定两个单词 word1 和 word2，返回使得 word1 和 word2 相同所需的最小步数。每步可以删除任意一个字符串中的一个字符。

示例 1:

> 输入: word1 = "sea", word2 = "eat"
> 输出: 2

示例 2:

> 输入: word1 = "leetcode", word2 = "etco"
> 输出: 4

dp[i][j] 表示使 word1[0:i] 和 word2[0:j] 相同的最少删除操作次数。

$$dp[i][j] = \begin{cases} dp[i-1][j-1], & word1[i] = word2[j] \\ min(dp[i-1][j], dp[i][j-1]) + 1, & word1[i] \neq word2[j] \end{cases}$$
$$dp[0][j] = j;$$
$$dp[i][0] = i;$$

> 上述表示中，word[0:i]表示前 i 个元素

```
int minDistance(string word1, string word2) {
    int m = word1.size(), n = word2.size();
```

```
    vector<vector<int>> dp(m + 1, vector<int>(n + 1));
    for (int i = 1; i <= m; ++i)
        dp[i][0] = i;
    for (int j = 1; j <= n; ++j)
        dp[0][j] = j;
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (word1[i - 1] == word2[j - 1])
                dp[i][j] = dp[i - 1][j - 1];
            else
                dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + 1;
        }
    }
    return dp[m][n];
}
```

压缩到一维，如下：

```
int minDistance(string word1, string word2) {
    int m = word1.size(), n = word2.size();
    vector<int> dp(n + 1);
    for (int j = 0; j <= n; j++)
        dp[j] = j;
    for (int i = 1; i <= m; i++) {
        int last = dp[0];
        dp[0] = i;
        for (int j = 1; j <= n; j++) {
            int old = dp[j];
            if (word1[i - 1] == word2[j - 1])
                dp[j] = last;
            else
                dp[j] = min(dp[j], dp[j - 1]) + 1;
            last = old;
        }
    }
    return dp[n];
}
```

> “
> 注意：这里用last记录dp[i-1][j-1]，因为在正向遍历中它会先被更新为dp[i][j-1]

最长数对链

给出 n 个数对。在每一个数对中，第一个数字总是比第二个数字小。现在，我们定义一种跟随关系，当且仅当 b < c 时，数对(c, d) 才可以跟在 (a, b) 后面。我们用这种形式来构造一个数对链。

给定一个数对集合，找出能够形成的最长数对链的长度。你不需要用到所有的数对，你可以以任何顺序选择其中的一些数对来构造。

示例：

> 输入：[[1,2], [2,3], [3,4]]
> 输出：2
> 解释：最长的数对链是 [1,2] -> [3,4]

dp[i] 表示以数对 i 结尾的最长数对链长度，则当 j > i 且 pairs[i][1] < pairs[j][0] 时有 $dp[j] = max(dp[j], dp[i] + 1)$

```cpp
int findLongestChain(vector<vector<int>>& pairs) {
    sort(pairs.begin(), pairs.end());
    int n = pairs.size();
    vector<int> dp(n+1, 1);
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            if (pairs[i][1] < pairs[j][0])
                dp[j] = max(dp[j], dp[i] + 1);
        }
    }
    return *max_element(dp.begin(), dp.end());
}
```

摆动序列

如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为摆动序列。第一个差（如果存在的话）可能是正数或负数。仅有一个元素或者含两个不等元素的序列也视作摆动序列。

例如，[1, 7, 4, 9, 2, 5] 是一个摆动序列，因为差值 (6, -3, 5, -7, 3) 是正负交替出现的。相反，[1, 4, 7, 2, 5] 和 [1, 7, 4, 5, 5] 不是摆动序列，第一个序列是因为它的前两个差值都是正数，第二个序列是因为它的最后一个差值为零。

子序列可以通过从原始序列中删除一些（也可以不删除）元素来获得，剩下的元素保持其原始顺序。

给你一个整数数组 nums，返回 nums 中作为摆动序列的最长子序列的长度。

示例 1：

> 输入：nums = [1,7,4,9,2,5]
> 输出：6

示例 2：

> 输入：nums = [1,17,5,10,13,15,10,5,16,8]
> 输出：7

示例 3：

> 输入：nums = [1,2,3,4,5,6,7,8,9]
> 输出：2

up[i] 表示范围 0~i 最长摆动子序列长度，末尾向上摆，down[i] 表示末尾向下摆，则

$$up[i] = \begin{cases} up[i-1] & nums[i] \le nums[i-1] \\ max(up[i-1], down[i-1]+1) & nums[i] > nums[i-1] \end{cases}$$
$$down[i] = \begin{cases} down[i-1] & nums[i] \ge nums[i-1] \\ max(down[i-1], up[i-1]+1) & nums[i] < nums[i-1] \end{cases}$$
$$up[0] = down[0] = 1$$

这里的状态转移方程可能很难理解，下面证明一下 up[i] 的式子，down同理：

当 nums[i]<=nums[i-1] 时，找不到比 up[i-1] 更长的了，因为任意以 nums[i] 结尾的末尾向上摆的都可以把 nums[i] 换成 nums[i-1]，且若不以 nums[i] 结尾，就等价于 up[i-1]，所以 up[i] = up[i-1];

当 nums[i]>nums[i-1]，如果不取 nums[i]，则 up[i]=up[i-1]，如果取 nums[i]，则分别考虑如何从 up[i-1] 和 down[i-1] 转移过来，如果从 up[i-1] 转移过来，那么必须经过 down[i-1]，所以只考虑 down[i-1] 即可，设末尾元素为 nums[j]，若nums[j]>=nums[i-1]，则可以替换为 nums[i-1]，后面接上 nums[i]，若 nums[j]<nums[i-1]，则可以直接在后面加上 nums[i]，总之就是 up[i] = down[i-1] + 1;

```
int wiggleMaxLength(vector<int>& nums) {
    int n = nums.size();
    if (n <= 2) {
        if (n == 2 && nums[0] != nums[1])
            return 2;
        else
            return 1;
    }
    vector<int> up(n, 1), down(n, 1);
    for (int i = 1; i < n; i++) {
        if (nums[i] > nums[i-1]) {
            up[i] = max(down[i-1] + 1, up[i-1]);
            down[i] = down[i-1];
        }
        else if (nums[i] < nums[i-1]) {
            down[i] = max(up[i-1] + 1, down[i-1]);
            up[i] = up[i-1];
        }
        else {
```

```
            up[i] = up[i-1];
            down[i] = down[i-1];
        }
    }
    return max(up[n-1], down[n-1]);
}
```

显然可以压缩为常量空间，代码如下：

```
int wiggleMaxLength(vector<int>& nums) {
    int n = nums.size();
    if (n <= 2) {
        if (n == 2 && nums[0] != nums[1])
            return 2;
        else
            return 1;
    }
    int up = 1, down = 1;
    for (int i = 1; i < n; i++) {
        if (nums[i] > nums[i-1])
            up = max(down + 1, up);
        else if (nums[i] < nums[i-1])
            down = max(up + 1, down);
    }
    return max(up, down);
}
```

目标和

给你一个整数数组 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。

示例 1：

> 输入：nums = [1,1,1,1,1], target = 3
> 输出：5
> 解释：一共有 5 种方法让最终目标和为 3。
> -1 + 1 + 1 + 1 + 1 = 3
> +1 - 1 + 1 + 1 + 1 = 3
> +1 + 1 - 1 + 1 + 1 = 3
> +1 + 1 + 1 - 1 + 1 = 3
> +1 + 1 + 1 + 1 - 1 = 3

示例 2：

> 输入：nums = [1], target = 1
> 输出：1

一个比较巧妙的方法，只考虑加法，剩下的全部做减法，设总和为 sum，则加法目标和为 $(sum + target)/2$，然后就转化为了“0-1”背包问题。

```
int findTargetSumWays(vector<int>& nums, int target) {
    int sum = accumulate(nums.begin(), nums.end(), 0);
    if ((sum + target) % 2 != 0)
        return 0;
    int n = (sum + target) / 2;
    if (n < 0)
        n = -n;
    vector<int> dp(n+1, 0);
    dp[0] = 1;
    for (int i = 1; i <= nums.size(); i++) {
        for (int j = n; j >= nums[i-1]; j--)
            dp[j] += dp[j-nums[i-1]];
    }
    return dp[n];
}
```

买卖股票之含手续费

给定一个整数数组 prices，其中 prices[i] 表示第 i 天的股票价格；整数 fee 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。返回获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

示例 1：

> 输入：prices = [1, 3, 2, 8, 4, 9], fee = 2
> 输出：8

示例 2：

> 输入：prices = [1,3,7,5,10,3], fee = 3
> 输出：6

第 i 天结束时，dp[i][0] 表示手里没有股票的最大利润，dp[i][1] 表示手里有股票的最大利润，则

$$
\begin{aligned}
dp[i][0] &= max(dp[i-1][0], dp[i-1][1] + prices[i-1] - fee) \\
dp[i][1] &= max(dp[i-1][1], dp[i-1][0] - prices[i-1]) \\
dp[1][0] &= 0 \\
dp[1][1] &= -prices[0]
\end{aligned}
$$

```cpp
int maxProfit(vector<int>& prices, int fee) {
    int n = prices.size();
    if (n == 1)
        return 0;
    vector<vector<int>> dp(n + 1, vector<int>(2));
    dp[1][0] = 0;
    dp[1][1] = -prices[0];
    for (int i = 2; i <= n; i++) {
        dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i-1] - fee);
        dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i-1]);
    }
    return dp[n][0];
}
```

空间压缩大法：

```cpp
int maxProfit(vector<int>& prices, int fee) {
    int n = prices.size();
    if (n == 1)
        return 0;
    int sell = 0, buy = -prices[0];
    for (int i = 2; i <= n; i++) {
        sell = max(sell, buy + prices[i-1] - fee);
        buy = max(buy, sell - prices[i-1]);
    }
    return sell;
}
```

分治法

算法解释

顾名思义，分治问题由“分”（divide）和“治”（conquer）两部分组成，通过把原问题分为子问题，再将子问题进行处理合并，从而实现对原问题的求解。我们在排序章节展示的归并排序就是典型的分治问题，其中“分”即为把大数组平均分成两个小数组，通过递归实现，最终我们会得到多个长度为 1 的子数组;“治”即为把已经排好序的两个小数组合成为一个排好序的大数组，从长度为 1 的子数组开始，最终合成一个大数组。

另外，自上而下的分治可以和 memoization 结合，避免重复遍历相同的子问题。如果方便推导，也可以换用自下而上的动态规划方法求解。

表达式问题

为运算表达式设计优先级

给你一个由数字和运算符（只包含+，-，*）组成的字符串 expression ，按不同优先级组合数字和运算符，计算并返回所有可能组合的结果。你可以按任意顺序返回答案。

生成的测试用例满足其对应输出值符合 32 位整数范围，不同结果的数量不超过 104。

示例 1：

> “
> 输入：expression = "2-1-1"
> 输出：[0,2]

示例 2：

> “
> 输入：expression = "2*3-4*5"
> 输出：[-34,-14,-10,-10,10]
> 解释：
> (2*(3-(4*5))) = -34
> ((2*3)-(4*5)) = -14
> ((2*(3-4))*5) = -10
> (2*((3-4)*5)) = -10
> (((2*3)-4)*5) = 10

利用分治思想，我们可以把加括号转化为，对于每个运算符号，先执行处理两侧的数学表达式，再处理此运算符号。注意特殊情况，即字符串内无运算符号，只有数字。

```
1 vector<int> diffWaysToCompute(string input) {
2     vector<int> ways;
```

```
    for (int i = 0; i < input.length(); i++) {
        char c = input[i];
        if (c == '+' || c == '-' || c == '*') {
            //分
            vector<int> left = diffWaysToCompute(input.substr(0, i));
            vector<int> right = diffWaysToCompute(input.substr(i + 1));
            //治
            for (const int & l: left) {
                for (const int & r: right) {
                    switch (c) {
                        case '+': ways.push_back(l + r); break;
                        case '-': ways.push_back(l - r); break;
                        case '*': ways.push_back(l * r); break;
                    }
                }
            }
        }
    }
    if (ways.empty())
        ways.push_back(stoi(input));    //stoi()函数将string类型转为int
    return ways;
}
```

我们发现，某些被 divide 的子字符串可能重复出现多次，因此我们可以用 memoization 来去重。或者与其我们从上到下用分治处理 + memoization，不如直接从下到上用动态规划处理。

```
vector<int> diffWaysToCompute(string input) {
    vector<int> data;
    vector<char> ops;
    int num = 0;
    char op = ' ';
    // 将 input 划分成数据（data）和运算符（ops），后面加一个 "+" 方便处理
    istringstream ss(input + "+");
    while (ss >> num && ss >> op) {
        data.push_back(num);
        ops.push_back(op);
    }
    int n = data.size();
    // 三维 vector 数组
    vector<vector<vector<int>>> dp(n, vector<vector<int>>(n, vector<int>
()));
    /*
        dp[i][j] 是一个一维 vector 数组，里面存储着：从第 i+1 个数字到第 j+1
个数字组成的算式的所有可能的运算结果，
        所以最后要输出的就是 dp[0][n-1]。
    */
    for (int i = 0; i < n; i++) {
```

```
        for (int j = i; j >= 0; j--) {
            // 如果 i == j，显然只有一种可能
            if (i == j)
                dp[j][i].push_back(data[i]);
            else {
                /* k 在 i 和 j 之间形成了一个 divide
                 举一个例子，有算式：1-2+3*4
                 当 i = 0, j = 3, k = 1 时，实际上就相当于计算这样一个算式的值：(1-2)+(3*4)，然后 push_back 到 dp[0][3] 里面
                 这样计数可以保证不重不漏
                */
                for (int k = j; k < i; k += 1) {
                    for (auto left : dp[j][k]) {
                        for (auto right : dp[k+1][i]) {
                            int val = 0;
                            switch (ops[k]) {
                                case '+': val = left + right; break;
                                case '-': val = left - right; break;
                                case '*': val = left * right; break;
                            }
                            dp[j][i].push_back(val);
                        }
                    }
                }
            }
        }
    }
    return dp[0][n-1];
}
```

练习

漂亮数组

对于某些固定的 N，如果数组 A 是整数 1, 2, ..., N 组成的排列，使得：对于每个 i < j，都不存在 k 满足 i < k < j 使得 A[k] * 2 = A[i] + A[j]。那么数组 A 是漂亮数组。

给定 N，返回任意漂亮数组 A（保证存在一个）。

示例 1：

> “
> 输入：4
> 输出：[2,1,4,3]

示例 2：

> “

> 输入：5
> 输出：[3,1,2,5,4]

观察表达式，发现左边恒为偶数，则要不满足，只需右边一奇一偶即可，所以可以把数组一分为二，奇数放左边，偶数放右边，现在只要左右两边分别是漂亮数组即可，则 1~N 的奇数或偶数分别可以双射到 1~(N+1)/2 和 1~N/2 的整数，容易证明，映射不影响漂亮数组的性质，所以问题被分解为两个更小规模的子问题，递归处理即可

带 memoization 的分治法，自上而下：

```
map<int, vector<int>> memo;
vector<int> beautifulArray(int n) {
    if(memo.count(n))
        return memo[n];
    vector<int> res(n);
    if(n == 1)
        res[0] = 1;
    else {
        int t = 0;
        for(int& x : beautifulArray((n + 1) / 2))
            res[t++] = 2 * x - 1;
        for(int& x : beautifulArray(n / 2))
            res[t++] = 2 * x;
    }
    memo[n] = res;
    return res;
}
```

当然可以动态规划，自下而上：

```
vector<int> beautifulArray(int n) {
    if (n == 1)
        return {1};
    vector<vector<int>> dp(n + 1, vector<int>());
    dp[1] = {1};
    dp[2] = {1, 2};
    for (int i = 3; i <= n; i++) {
        for (int j = 0; j < (i + 1) / 2; j++)
            dp[i].push_back(dp[(i+1)/2][j] * 2 - 1);
        for (int j = 0; j < i / 2; j++)
            dp[i].push_back(dp[i/2][j] * 2);
    }
    return dp[n];
}
```

戳气球

有 n 个气球，编号为 0 到 n - 1，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。戳破第 i 个气球，你可以获得 nums[i - 1] * nums[i] * nums[i + 1] 枚硬币。 这里的 i - 1 和 i + 1 代表和 i 相邻的两个气球的序号。如果 i - 1或 i + 1 超出了数组的边界，那么就当它是一个数字为 1 的气球。

求所能获得硬币的最大数量。

示例 1：

> 输入：nums = [3,1,5,8]
> 输出：167

示例 2：

> 输入：nums = [1,5]
> 输出：10

为方便处理，把数组两端加上 nums[-1] 和 nums[n]，都等于 1，并保存在 rec 中，即 rec[i] = nums[i-1]，戳气球会导致原本不相邻的气球变成相邻的，所以我们反向考虑，添加气球，定义 solve(i, j) 表示在区间 (i, j) 内可获得的最大硬币数，则可以取一个中间的点 mid（这个mid必须放在最后戳破，因为只有这样可以消除两个子问题的相关性）分别求 solve(i, mid) 和 solve(mid, j)，然后相加即可，于是把原问题分解为了一个个子问题。

```
vector<vector<int>> dp; //记忆化
vector<int> rec; //记录气球值
int maxCoins(vector<int>& nums) {
    int n = nums.size();
    rec.push_back(1);
    for (int i = 0; i < n; i++)
        rec.push_back(nums[i]);
    rec.push_back(1);
    dp.resize(n + 2, vector<int>(n + 2, -1));
    return solve(0, n + 1);
}
int solve(int l, int r) {
    if (l > r)
        return 0;
    if (dp[l][r] != -1)
        return dp[l][r];
    for (int mid = l + 1; mid < r; mid++) {
        int sum = rec[mid] * rec[l] * rec[r];
        sum += solve(l, mid) + solve(mid, r);
        dp[l][r] = max(sum, dp[l][r]);
    }
    return dp[l][r];
```

```
}
```

当然可以使用自下而上的动态规划（dp[i][j] 表示 (i, j) 范围内最大硬币数）：

```cpp
int maxCoins(vector<int>& nums) {
    int n = nums.size();
    vector<vector<int>> dp(n + 2, vector<int>(n + 2));
    vector<int> val(n + 2);
    val[0] = val[n + 1] = 1;
    for (int i = 1; i <= n; i++)
        val[i] = nums[i - 1];
    for (int i = n - 1; i >= 0; i--) {
        for (int j = i + 2; j <= n + 1; j++) {
            for (int k = i + 1; k < j; k++) {
                int sum = val[i] * val[k] * val[j];
                sum += dp[i][k] + dp[k][j];
                dp[i][j] = max(dp[i][j], sum);
            }
        }
    }
    return dp[0][n+1];
}
```

数学问题

公倍数与公因数

辗转相除法

利用辗转相除法，我们可以很方便地求得两个数的最大公因数（greatest common divisor, gcd）；将两个数相乘再除以最大公因数即可得到最小公倍数（least common multiple, lcm）。

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a% b);
}
int lcm(int a, int b) {
    return a * b / gcd(a, b);
}
```

扩展欧几里得算法

进一步地，我们也可以通过扩展欧几里得算法（extended gcd）在求得 a 和 b 最大公因数的同时，也得到它们的系数 x 和 y，从而使 ax + by = gcd(a, b)。

```
int xGCD(int a, int b, int &x, int &y) {
    if (!b) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1, gcd = xGCD(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return gcd;
}
```

质数

质数又称素数，指的是指在大于 1 的自然数中，除了 1 和它本身以外不再有其他因数的自然数。值得注意的是，由质因子分解定理，每一个数都可以分解成质数的乘积，且分解后形式唯一。

计数质数

给定整数 n，返回所有小于非负整数 n 的质数的数量。（n >= 0）

埃拉托斯特尼筛法（Sieve of Eratosthenes，简称埃氏筛法）是非常常用的，判断一个整数是否是质数的方法。并且它可以在判断一个整数 n 时，同时判断所有小于 n 的整数，因此非常适合这道题。其原理也十分易懂：从 1 到 n 遍历，假设当前遍历到 m，则把所有小于 n 的、且是 m 的倍数的整数标为和数；遍历完成后，没有被标为和数的数字即为质数。

```
int countPrimes(int n) {
    if (n <= 2)
        return 0;
    vector<bool> prime(n, true);
    int count = n - 2; // 去掉不是质数的1
    for (int i = 2; i < n; i++) {
        if (prime[i]) { //若一个数已经标记过，则他的倍数肯定也被标记过了，可以跳过
            for (int j = 2 * i; j < n; j += i) {
                if (prime[j]) {
                    prime[j] = false;
                    count--;
                }
            }
        }
    }
    return count;
}
```

利用质数的一些性质，我们可以进一步优化该算法。对于一个质数 x，如果按上文说的我们从 2x 开始标记其实是冗余的，应该直接从 x·x 开始标记，因为 2x,3x,… 这些数一定在 x 之前就被其他数的倍数标记过了，例如 2 的所有倍数，3 的所有倍数等。另外，偶数倍的 x 一定被 2 标记过了，所以 j 增量为 2*i；另外，若一个数有偶数因子，则这个数一定已经被 2 标记过了，所以 i 增量为 2

```
int countPrimes(int n) {
    if (n <= 2)
        return 0;
    vector<bool> prime(n, true);
    int i = 3, sqrtn = sqrt(n), count = n / 2; // 偶数一定不是质数
    while (i <= sqrtn) { // 最小质因子一定小于等于开方数
        for (int j = i * i; j < n; j += 2 * i) { // 避免偶数和重复遍历
            if (prime[j]) {
                count--;
                prime[j] = false;
            }
        }
        do {
            i += 2;
        } while (i <= sqrtn && !prime[i]); // 避免偶数和重复遍历
    }
    return count;
}
```

数字处理

七进制数

给定一个整数 num，将其转化为 7 进制，并以字符串形式输出。

进制转换类型的题，通常是利用除法和取模（mod）来进行计算，同时也要注意一些细节，如负数和零。如果输出是数字类型而非字符串，则也需要考虑是否会超出整数上下界。

```
string convertToBase7(int num) {
    if (num == 0)
        return "0";
    bool is_negative = (num < 0);
    if (is_negative)
        num = -num;
    string ans;
    while (num) {
        int a = num / 7, b = num % 7;
        ans = to_string(b) + ans;
        num = a;
    }
    return is_negative ? "-" + ans: ans;
}
```

阶乘后的零

给定一个整数 n ，返回 n! 结果中尾随零的数量。（n >= 0)

示例 1:

> 输入: n = 3
> 输出: 0

示例 2:

> 输入: n = 5
> 输出: 1

示例 3:

> 输入: n = 0
> 输出: 0

每个尾部的 0 由 2 × 5 = 10 而来，因此我们可以把阶乘的每一个元素拆成质数相乘，统计有多少个 2 和 5，取最小值即可，明显的，质因子 2 的数量远多于质因子 5 的数量，因此我们可以只统计阶乘结果里有多少个质因子 5，即 1~n 的质因子中 5 的个数和

```
int trailingZeroes(int n) {
    int ans = 0;
    for (int i = 5; i <= n; i += 5) {
        int temp = i;
        while(temp % 5 == 0) {
            temp /= 5;
            ans++;
        }
    }
    return ans;
}
```

可以用数学方法再优化，换个角度考虑 1~n 的质因子 p 的个数，首先 p 的倍数的个数为 $n_1 = [\frac{n}{p}]$，贡献了 n_1 个 p，然后 p^2 的倍数个数为 $n_2 = [\frac{n}{p^2}]$，因为在 n_1 中已经算过了一半，所以只多贡献了 n_2 个 p，以此类推，

$$ans = \sum_{k=1}[\frac{n}{p^k}] = \sum_{k=1}[\frac{\frac{n}{p^{k-1}}}{p}]$$

因此我们可以通过不断将 n 除以 5，并累加每次除后的 n，来得到答案。

```
int trailingZeroes(int n) {
    return n == 0 ? 0 : n / 5 + trailingZeroes(n / 5);
}
```

多么简洁，优雅，美妙！数学真伟大！

字符串相加

给定两个字符串形式的非负整数 num1 和 num2，num1 和num2 都不包含任何前导零，计算它们的和并同样以字符串形式返回。

你不能使用任何內建的用于处理大整数的库（比如 BigInteger），也不能直接将输入的字符串转换为整数形式。num1 和num2 都不包含任何前导零

示例 1：

> 输入：num1 = "11", num2 = "123"
> 输出："134"

示例 2：

>

> 输入：num1 = "456", num2 = "77"
> 输出："533"

示例 3：

> “
> 输入：num1 = "0", num2 = "0"
> 输出："0"

因为相加运算是从后往前进行的，所以可以先翻转字符串，再逐位计算。模拟人工手算法即可。

```
string addStrings(string num1, string num2) {
    string output("");
    reverse(num1.begin(), num1.end());
    reverse(num2.begin(), num2.end());
    int onelen = num1.length(), twolen = num2.length();
    if (onelen <= twolen){
        swap(num1, num2);
        swap(onelen, twolen);
    }
    int addbit = 0;
    for (int i = 0; i < twolen; i++){
        int cur_sum = (num1[i]-'0') + (num2[i]-'0') + addbit;
        output += to_string((cur_sum) % 10);
        addbit = cur_sum < 10 ? 0 : 1;
    }
    for (int i = twolen; i < onelen; i++){
        int cur_sum = (num1[i]-'0') + addbit;
        output += to_string((cur_sum) % 10);
        addbit = cur_sum < 10 ? 0 : 1;
    }
    if (addbit)
        output += "1";
    reverse(output.begin(), output.end());
    return output;
}
```

‘3’的幂次方

给定一个整数，写一个函数来判断它是否是 3 的幂次方。如果是，返回 true；否则，返回 false。(n为int类型)

示例 1：

> “
> 输入：n = 27
> 输出：true

示例 2：

> 输入：n = 0
> 输出：false

示例 3：

> 输入：n = 45
> 输出：false

有两种方法，一种是利用对数。设 $log_3(n) = m$，如果 n 是 3 的整数次方，那么 m 一定是整数。

```
bool isPowerOfThree(int n) {
    return fmod(log10(n) / log10(3), 1) == 0;
}
```

另一种方法是，因为在 int 范围内 3 的最大次方是 $3^{19} = 1162261467$，如果 n 是 3 的整数次方，那么 1162261467 除以 n 的余数一定是零；反之亦然。

```
bool isPowerOfThree(int n) {
    return n > 0 && 1162261467 % n == 0;
}
```

随机与取样

打乱数组

给定一个数组，要求在类中实现两个指令函数。第一个函数“shuffle”可以随机打乱这个数组，第二个函数“reset”可以恢复原来的顺序。

输入是一个存有整数数字的数组，和一个包含指令函数名称的数组。输出是一个二维数组，表示每个指令生成的数组。

> Input: nums = [1,2,3], actions: ["shuffle","shuffle","reset"]
> Output: [[2,1,3],[3,2,1],[1,2,3]]
> 在这个样例中，前两次打乱的结果只要是随机生成即可。

我们采用经典的 Fisher-Yates 洗牌算法，原理是通过随机交换位置来实现随机打乱，有正向和反向两种写法，且实现非常方便。注意这里“reset”函数以及类的构造函数的实现细节。

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

```
    Solution(vector<int>& nums): origin(nums) {}
    vector<int> reset() {
        return origin;
    }
    vector<int> shuffle() {
        if (origin.empty()) return {};
        vector<int> shuffled(origin);
        int n = origin.size();
        // 反向洗牌：（正向效果相同
        for (int i = n - 1; i >= 0; i--)
            swap(shuffled[i], shuffled[rand() % (i + 1)]);
        return shuffled;
    }
};
```

或者直接使用random_shuffle()

```
vector<int> shuffle() {
    if (origin.empty()) return {};
    vector<int> shuffled = origin;
    random_shuffle(shuffled.begin(), shuffled.end());
    return shuffled;
}
```

按权重随机选择

给定一个数组，数组每个位置的值表示该位置的权重，要求按照权重的概率去随机采样。

> “
> Input: weights = [1,3], actions: ["pickIndex","pickIndex","pickIndex"]
> Output: [0,1,1]

设数组 w 的权重之和为 total。根据题目的要求，我们可以看成将 [1,total] 范围内的所有整数分成 n 个部分（其中 n 是数组 w 的长度），第 i 个部分恰好包含 w[i] 个整数，并且这 n 个部分两两的交集为空。随后我们在 [1,total] 范围内随机选择一个整数 x，如果整数 x 被包含在第 i 个部分内，我们就返回 i。

一种较为简单的划分方法是按照从小到大的顺序依次划分每个部分。例如 w=[3,1,2,4] 时，权重之和 total=10，那么我们按照 [1,3],[4,4],[5,6],[7,10] 对 [1,10] 进行划分，使得它们的长度恰好依次为 3,1,2,4。可以发现，每个区间的左边界是在它之前出现的所有元素的和加上 1，右边界是到它为止的所有元素的和。因此，如果我们用 pre[i] 表示数组 w 的前缀和：

$$pre[i] = \sum_{k=0}^{i} w[k]$$

第i个区间的左边界就是$pre[i] - w[i] + 1$，右边界为$pre[i]$。

当划分完成后，假设我们随机到了整数 x，我们希望找到满足：
$pre[i]-w[i]+1 \leq x \leq pre[i]$ 的 i 并将其作为答案返回。由于 pre[i] 是单调递增的，因此我们可以使用二分查找快速找到 i，即找出最小的满足 x≤pre[i] 的下标 i。

```cpp
class Solution {
    vector<int> sums;
    public:
    Solution(vector<int> weights): sums(std::move(weights)) {
        partial_sum(sums.begin(), sums.end(), sums.begin());
    }
    int pickIndex() {
        int pos = (rand() % sums.back()) + 1;
        return lower_bound(sums.begin(), sums.end(), pos) - sums.begin();
    }
};
```

> “
>
> partial_sum用于计算局部和（start, end, storage)
>
> lower_bound用于找出范围内不小于num的第一个元素（start, end, storage)

链表随机节点

给定一个单向链表，要求设计一个算法，可以随机取得其中的一个数字。

> “
>
> Input: 1->2->3->4->5
>
> Output: 3

不同于数组，在未遍历完链表前，我们无法知道链表的总长度。这里我们就可以使用水库采样：遍历一次链表，在遍历到第 m 个节点时，有 1/m 的概率选择这个节点覆盖掉之前的节点选择。

我们提供一个简单的，对于水库算法随机性的证明。对于长度为 n 的链表的第 m 个节点，最后被采样的充要条件是它被选择，且之后的节点都没有被选择。这种情况发生的概率为 $\frac{1}{m} \times \frac{m}{m+1} \times \frac{m+1}{m+2} \ldots = \frac{1}{n}$ 因此每个点都有均等的概率被选择。

```cpp
class Solution {
    ListNode* head;
    public:
    Solution(ListNode* n): head(n) {}
    int getRandom() {
        int ans = head->val;
        ListNode* node = head->next;
        int i = 2;
        while (node) {
            if ((rand() % i) == 0)
                ans = node->val;
```

```
            i++;
            node = node->next;
        }
        return ans;
    }
};
```

练习

Excel表列名称

给你一个整数 columnNumber，返回它在 Excel 表中相对应的列名称。

例如：

> A -> 1
> B -> 2
> ...
> Z -> 26
> AA -> 27
> AB -> 28
> ...

示例 1：

> 输入：columnNumber = 1
> 输出："A"

示例 2：

> 输入：columnNumber = 28
> 输出："AB"

和正常 0~25 的 26 进制相比，本质上就是每一位多加了 1。假设 A = 0，B = 1，那么 AB = 26 * 0 + 1 * 1，而现在 AB = 26 * (0 + 1) + 1 * (1 + 1)，所以只要在处理每一位的时候减 1，就可以按照正常的 26 进制来处理

```cpp
string convertToTitle(int columnNumber) {
    string ans;
    while(columnNumber) {
        columnNumber--;
        int re = columnNumber % 26;
        ans.push_back('A' + re);
        columnNumber /= 26;
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

二进制求和

给你两个二进制字符串，返回它们的和（用二进制表示）。输入为非空字符串且只包含数字 1 和 0。

示例 1:

> 输入: a = "11", b = "1"
> 输出: "100"

示例 2:

> 输入: a = "1010", b = "1011"
> 输出: "10101"

翻转，进位计算

```cpp
string addBinary(string a, string b) {
    string ans;
    reverse(a.begin(), a.end());
    reverse(b.begin(), b.end());
    int n = max(a.size(), b.size()), carry = 0;
    for (size_t i = 0; i < n; ++i) {
        carry += i < a.size() ? (a.at(i) == '1') : 0;
        carry += i < b.size() ? (b.at(i) == '1') : 0;
        ans.push_back(carry % 2 ? '1' : '0');
        carry /= 2;
    }
    if (carry)
        ans.push_back('1');
    reverse(ans.begin(), ans.end());
    return ans;
}
```

除自身以外数组的乘积

给你一个整数数组 nums，返回数组 answer，其中 answer[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积。

题目数据保证数组 nums之中任意元素的全部前缀元素和后缀的乘积都在 32 位整数范围内。请不要使用除法，且在 O(n) 时间复杂度内完成此题。

示例 1:

> 输入: nums = [1,2,3,4]
> 输出: [24,12,8,6]

示例 2:

> 输入: nums = [-1,1,0,-3,3]
> 输出: [0,0,9,0,0]

L[i] 表示 nums[i] 左侧所有数字的乘积，R[i] 表示右侧，初始化后遍历数组即可

```cpp
vector<int> productExceptSelf(vector<int>& nums) {
    int length = nums.size();
    vector<int> L(length, 0), R(length, 0);
    vector<int> answer(length);
    L[0] = 1;
    for (int i = 1; i < length; i++)
        L[i] = nums[i - 1] * L[i - 1];
    R[length - 1] = 1;
    for (int i = length - 2; i >= 0; i--)
        R[i] = nums[i + 1] * R[i + 1];
    for (int i = 0; i < length; i++)
        answer[i] = L[i] * R[i];
    return answer;
}
```

最少移动次数使数组相等II

给你一个长度为 n 的整数数组 nums，返回使所有数组元素相等需要的最少移动数。在一步操作中，你可以使数组中的一个元素加 1 或者减 1。

示例 1:

> 输入： nums = [1,2,3]
> 输出： 2

示例 2：

> 输入：nums = [1,10,2,9]
> 输出：16

很容易想到，都变成中位数所需移动次数较少，证明为什么用中位数，其实很简单，如果target往两侧偏移，如果一侧数字少一侧数字多，往多的一侧移动必然使得结果减少。直到target两侧的数字数量完全相等时，取得最小值。这里使用快速选择找到 target

```cpp
int minMoves2(vector<int>& nums) {
    nth_element(nums.begin(), nums.begin() + nums.size() / 2, nums.end());
    int target = nums[nums.size()/2], ans = 0;
    for(int& x: nums)
        ans += abs(x - target);
    return ans;
}
```

多数元素

给定一个大小为 n 的数组 nums，返回其中的多数元素。多数元素是指在数组中出现次数大于 n/2 的元素。你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1：

> 输入：nums = [3,2,3]
> 输出：3

示例 2：

> 输入：nums = [2,2,1,1,1,2,2]
> 输出：2

Boyer-Moore 投票算法：

1. 我们维护一个候选众数 candidate 和它出现的次数 count。初始时 candidate 可以为任意值，count 为 0；
2. 我们遍历数组 nums 中的所有元素，对于每个元素 x，在判断 x 之前，如果 count 的值为 0，我们先将 x 的值赋予 candidate，随后我们判断 x：
    - 如果 x 与 candidate 相等，那么计数器 count 的值增加 1；
    - 如果 x 与 candidate 不等，那么计数器 count 的值减少 1。
3. 在遍历完成后，candidate 即为整个数组的众数（这里众数指个数 > n/2）。

这里提供一个比较好理解的证明：

首先遍历数组时，遍历到的人将有一次投票权，相同的数字代表同一个人，在投票中，所有人都是利己的，即只会给自己投支持票，给不是自己的候选人投反对票。对于第一个候选人来说，如果他不是 maj，count迟早会变为 0，因为他自己最多给自己投的票数不超过一半，而其他人都会投反对票；即使反对票全部是 maj 投的（这里是极端情况），那么以 count=0 为分界，前面一段数组中 maj 也只占了一半，所以后面一段数组中众数还是 maj。如果第一个候选人是 maj，若 count 到最后都没有变为 0，当然选出来的就是 maj，若 count 在中间变为 0 了，那么前面一段数组中 maj 没占到一半票数，所以后面一段众数还是 maj。把前面一段丢掉不考虑，继续在后面一段中投票，可以看出是重复上述过程，所以最终候选人一定是是 maj

```cpp
int majorityElement(vector<int>& nums) {
    int candidate = 0, count = 0;
    for (int num : nums) {
        if (num == candidate)
            count++;
        else if (--count < 0) {
                candidate = num;
                count = 1;
        }
    }
    return candidate;
}
```

用rand7()实现rand10()

给定方法 rand7 可生成 [1,7] 范围内的均匀随机整数，试写一个方法 rand10 生成 [1,10] 范围内的均匀随机整数。你只能调用 rand7() 且不能调用其他方法。请不要使用系统的 Math.random() 方法。

每个测试用例将有一个内部参数 n，即你实现的函数 rand10() 在测试时将被调用的次数。请注意，这不是传递给 rand10() 的参数。

> “
> 输入: 3
> 输出: [3,8,10]

两个rand7()相乘，选择概率相等的一些数，映射到 [1,10] 即可；

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	8	9	10	1	2	3	4
3	5	6	7	8	9	10	1
4	2	3	4	5	6	7	8
5	9	10	1	2	3	4	5
6	6	7	8	9	10	*	*
7	*	*	*	*	*	*	*

```c
int rand10() {
    int row, col, idx;
    do {
        row = rand7();
        col = rand7();
        idx = col + (row - 1) * 7;
    } while (idx > 40);
    return 1 + (idx - 1) % 10;
}
```

快乐数

编写一个算法来判断一个数 n 是不是快乐数。

「快乐数」 定义为：

> 对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和。然后重复这个过程直到这个数变为 1，也可能是 无限循环但始终变不到 1。
> 如果这个过程结果为 1，那么这个数就是快乐数。

如果 n 是快乐数就返回 true；不是，则返回 false。

示例 1：

> 输入：n = 19
> 输出：true
> 解释：
> $1^1 + 9^2 = 82$
> $8^2 + 2^2 = 68$
> $6^2 + 8^2 = 100$
> $1^2 + 0^2 + 0^2 = 1$

对于 3 位数的数字，它不可能大于 $243(3\times9^2)$。这意味着它要么被困在 243 以下的循环内，要么跌到 1。4 位或以上的数字在每一步都会丢失一位，直到降到 3 位为止。所以只有两种情况，一是进入循环，二是得到 1。把每个中间数字作为一个节点，可以使用前面章节讲过的快慢指针进行环路检测。如果 n 是一个快乐数，即没有循环，那么快跑者最终会比慢跑者先到达数字 1。如果 n 不是一个快乐的数字，那么最终快跑者和慢跑者将在同一个数字上相遇

```c
int bitSquareSum(int n) {
    int sum = 0;
    while(n > 0) {
        int bit = n % 10;
        sum += bit * bit;
```

```
        n = n / 10;
    }
    return sum;
}
bool isHappy(int n) {
    int slow = n, fast = n;
    do {
        slow = bitSquareSum(slow);
        fast = bitSquareSum(fast);
        fast = bitSquareSum(fast);
    } while(slow != fast);
    return slow == 1;
}
```

位运算

常用技巧

位运算是算法题里比较特殊的一种类型，它们利用二进制位运算的特性进行一些奇妙的优化和计算。常用的位运算符号包括：“∧”按位异或（取不同）、“&”按位与（取交）、“|”按位或（取并）、“~”取反、“<<” 算术左移和“>>”算术右移。

除此之外，`n & (n - 1)` 可以去除 n 的位级表示中最低的那一位，例如对于二进制表示 11110100 ，减去 1 得到 11110011，这两个数按位与得到 11110000。`n & (-n)` 可以得到 n 的位级表示中最低的那一位，例如对于二进制表示 11110100，取负得到 00001100，这两个数按位与得到 00000100。

位运算基础问题

汉明距离

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给你两个整数 x 和 y，计算并返回它们之间的汉明距离。

示例 1：

> “
> 输入：x = 1, y = 4
> 输出：2
> 解释：
> 1　(0 0 0 1)
> 4　(0 1 0 0)

示例 2：

> “
> 输入：x = 3, y = 1
> 输出：1

按位异或，统计 1 的个数即可

```
int hammingDistance(int x, int y) {
    int diff = x ^ y, ans = 0;
    while (diff) {
        ans += diff & 1;
        diff >>= 1;
    }
    return ans;
}
```

颠倒二进制位

颠倒给定的 32 位无符号整数的二进制位。

使用算术左移和右移，可以很轻易地实现二进制的翻转。

```
uint32_t reverseBits(uint32_t n) {
    uint32_t ans = 0;
    for (int i = 0; i < 32; i++) {
        ans <<= 1;
        ans += n & 1;
        n >>= 1;
    }
    return ans;
}
```

只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗?

示例 1:

> 输入: [2,2,1]
> 输出: 1

示例 2:

> 输入: [4,1,2,1,2]
> 输出: 4

需要用到异或运算的性质：

- a ^ a = 0
- a ^ 0 = a
- a ^ b ^ a = b ^ (a ^ a) = b（交换律和结合律）

所以只要全部异或最后结果就是答案

```
int singleNumber(vector<int>& nums) {
    int ans = 0;
    for (const int & num: nums)
        ans ^= num;
    return ans;
}
```

二进制特性

利用二进制的一些特性，我们可以把位运算使用到更多问题上。例如，我们可以利用二进制和位运算输出一个数组的所有子集。假设我们有一个长度为 n 的数组，我们可以生成长度为 n 的所有二进制，1 表示选取该数字，0 表示不选取。这样我们就获得了 2^n 个子集。

4的幂

给定一个整数，写一个函数来判断它是否是 4 的幂次方。如果是，返回 `true`；否则，返回 `false`。

首先我们考虑一个数字是不是 2 的（整数）次方：如果一个数字 n 是 2 的整数次方，那么它 的二进制一定是0...010...0 这样的形式；考虑到 n − 1 的二进制是 0...001...1，这两个数求按位与 的结果一定是 0。因此如果 n & (n - 1) 为 0，那么这个数是 2 的次方。如果这个数也是 4 的次方，那二进制表示中 1 的位置必须为奇数位。我们可以把 n 和二进制的 10101...101（即十进制下的 1431655765）做按位与，如果结果不为 0，那么说明这个数是 4 的 次方。

```
bool isPowerOfFour(int n) {
    return n > 0 && !(n & (n - 1)) && (n & 1431655765);
}
```

最大单词长度乘积

给你一个字符串数组 words，找出并返回 length(words[i]) * length(words[j]) 的最大值，并且这两个单词不含有公共字母。如果不存在这样的两个单词，返回 0。

怎样快速判断两个字母串是否含有重复数字呢？可以为每个字母串建立一个长度为 26 的二进制数字，每个位置表示是否存在该字母。如果两个字母串含有重复数字，那它们的二进制表示的按位与不为 0。同时，我们可以建立一个哈希表来存储字母串（在数组的位置）到二进制数字的映射关系，方便查找调用。

```
int maxProduct(vector<string>& words) {
    unordered_map<int, int> hash;
    int ans = 0;
    for (const string & word : words) {
        int mask = 0, size = word.size();
        for (const char & c : word)
            mask |= 1 << (c - 'a');
        hash[mask] = max(hash[mask], size);
        for (const auto& [h_mask, h_len]: hash) {
            if (!(mask & h_mask))
                ans = max(ans, size * h_len);
        }
    }
    return ans;
}
```

比特位计数

给定一个非负整数 n，求从 0 到 n 的所有数字的二进制表达中，分别有多少个 1。

本题可以利用动态规划和位运算进行快速的求解。定义一个数组 dp，其中 dp[i] 表示数字 i 的二进制含有 1 的个数。对于第 i 个数字，如果它二进制的最后一位为 1，那么它含有 1 的个数 则为 dp[i-1] + 1；如果它二进制的最后一位为 0，那么它含有 1 的个数和其算术右移结果相同，即 dp[i>>1]。

```
vector<int> countBits(int num) {
    vector<int> dp(num+1, 0);
    for (int i = 1; i <= num; ++i)
        dp[i] = i & 1? dp[i-1] + 1: dp[i>>1];
    // 等价于dp[i] = dp[i&(i-1)] + 1;
    return dp;
}
```

练习

丢失的数字

给定一个包含 [0, n] 中 n 个数的数组 nums ，找出 [0, n] 这个范围内没有出现在数组中的那个数。

示例 1：

> 输入：nums = [3,0,1]
> 输出：2
> 解释：n = 3，因为有 3 个数字，所以所有的数字都在范围 [0,3] 内。2 是丢失的数字，因为它没有出现在 nums 中。

示例 2：

> 输入：nums = [0,1]
> 输出：2

实际是上面例题的变种，只需在后面加上 0-n 的数字，即可转为只出现一次的数字那一题

```cpp
int missingNumber(vector<int>& nums) {
    int res = 0;
    int n = nums.size();
    for (int i = 0; i < n; i++)
        res ^= nums[i];
    for (int i = 0; i <= n; i++)
        res ^= i;
    return res;
}
```

注意这种巧妙优雅的写法，看似没加，实际等于加了

交替位二进制数

给定一个正整数，检查它的二进制表示是否总是 0、1 交替出现：换句话说，就是二进制表示中相邻两位的数字永不相同。

对输入 n 的二进制表示右移一位后，得到的数字再与 n 按位异或得到 a。当且仅当输入 n 为交替位二进制数时，a 的二进制表示全为 1（不包括前导 0）。这里进行简单证明：当 a 的每一位为 1 时，当且仅当 n 的所有相邻位相异，即 n 为交替位二进制数。

将 a 与 a + 1 按位与，当且仅当 a 的二进制表示全为 1 时，结果为 0。这里进行简单证明：当且仅当 a 的二进制表示全为 1 时，a + 1 可以进位，并将原最高位置为 0，按位与的结果为 0。否则，不会产生进位，两个最高位都为 1，相与结果不为 0。

```cpp
bool hasAlternatingBits(int n) {
    long a = n ^ (n >> 1);
    return (a & (a + 1)) == 0;
}
```

很简单的两条性质，但不容易想到

数字的补数

对整数的二进制表示取反（0 变 1 ，1 变 0）后，再转换为十进制表示，可以得到这个整数的补数。

与同位数的最大的取异或

```cpp
int findComplement(int num) {
    long tmp = num, c = 1;
    while (tmp) {
        c <<= 1;
        tmp >>= 1;
    }
    return num ^ (c - 1);
}
```

只出现一次的数字III

给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那两个元素。你可以按任意顺序返回答案。

进阶：你的算法应该具有线性时间复杂度。你能否仅使用常数空间复杂度来实现?

例题的进阶版，全部异或后剩下 x_1 和 x_2 的异或结果，x 显然不会等于 0，因为如果 x=0，那么说明 $x_1 = x_2$，这样就不是只出现一次的数字了。因此，我们可以使用位运算 x & -x 取出 x 的二进制表示中最低位那个 1，设其为第 l 位，那么 x_1 和 x_2 中的某一个数的二进制表示的第 l 位为 0，另一个数的二进制表示的第 l 位为 1。

这样一来，我们就可以把 nums 中的所有元素分成两类，其中一类包含所有二进制表示的第 l 位为 0 的数，另一类包含所有二进制表示的第 l 位为 1 的数。可以发现：

- 对于任意一个在数组 nums 中出现两次的元素，该元素的两次出现会被包含在同一类中；
- 对于任意一个在数组 nums 中只出现了一次的元素，即 x_1 和 x_2 它们会被包含在不同类中。

因此，如果我们将每一类的元素全部异或起来，那么其中一类会得到 x_1，另一类会得到 x_2。这样我们就找出了这两个只出现一次的元素。

```cpp
vector<int> singleNumber(vector<int>& nums) {
    int xorsum = 0;
    for (int num: nums)
        xorsum ^= num;
    // 防止溢出
    int lsb = (xorsum == INT_MIN ? xorsum : xorsum & (-xorsum));
    int type1 = 0, type2 = 0;
    for (int num: nums) {
        if (num & lsb)
            type1 ^= num;
        else
            type2 ^= num;
    }
    return {type1, type2};
}
```