谢字希

乐于总结，乐于开源

https://github.com/XieZixiUSTC/code1024

(arr[i]&1 判断奇数，bool flag=true;// 符号，标志位，判断正负，和前导 0 一样，for(auto&[_, i] : mp)

2 人赞同了该回答

★相同点：

●都是地址的概念；

指针指向一块内存，它的内容是所指内存的地址；而引用则是某块内存的别名。

★不同点：

●指针是一个实体，而引用仅是个别名；

●引用只能在定义时被初始化一次，之后不可变；指针可变；引用“从一而终”，指针可以“见异思迁”；

●引用没有 const，指针有 const，const 的指针不可变；

●引用不能为空，指针可以为空；

● “sizeof 引用”得到的是所指向的变量(对象)的大小，而“sizeof 指针”得到的是指针本身的大小；

●指针和引用的自增(++)运算意义不一样；

●引用是类型安全的，而指针不是 (引用比指针多了类型检查

c 语言中所有传递给函数的参数都是传值方式进行的。

传值，是把实参的值赋值给行参，那么对行参的修改，不会影响实参的值
传地址，是传值的一种特殊方式，只是他传递的是地址，不是普通的如 int，那么传地址以后，实参和行参都指向同一个对象

传引用，真正的以地址的方式传递参数，传递以后，行参和实参都是同一个对象，只是他们名字不同而已， 对行参的修改将影响实参的值。
传递引用与传指针、传值的区别？

(1)传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象(在主调函数中)的操作。

(2)使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作;而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本;如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。

(3)使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用"*指针变量名"的形式进行运算，这很容易产生错误且程序的阅读性较差;另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

从本质上来说，传值和传指针都是传值方式，除了传引用之外。

---

10. 正则表达式匹配（当等于星号的时候，前面 J-1 与 i 不相等，只能匹配 0 次，当前 f[i][j]取决于 f[i][j-2],把当前*字母看做不存在，那么就是 j-2 与 i 判断了。就是字符串的这个字母要去 pattern 前面（j-2）找匹配

相等可以匹配 0 次，或者匹配 1+ci,f[i][j]=f[i-1][j],匹配 1 次，就是当前 a 被匹配了，那么就是 a 前面的字符串与剩下的 pattern 匹配，之后能否匹配多次，就看下一个了。本质是每次字符，都要找到匹配的。）

难度困难 2031

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 解释："a" 无法匹配 "aa" 整个字符串。

示例 2:

输入：s = "aa" p = "a*"输出：true 解释：因为 '*' 代表可以匹配零个或多个前面的那一个元素, 在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。

示例 3：

输入：s = "ab" p = ".*"输出：true 解释：".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。

示例 4：

输入：s = "aab" p = "c*a*b"输出：true 解释：因为 '*' 表示零个或多个，这里 'c' 为 0 个,'a' 被重复一次。因此可以匹配字符串 "aab"。

示例 5：

输入：s = "mississippi" p = "mis*is*p*."输出：false

方法一：动态规划

思路与算法

题目中的匹配是一个「逐步匹配」的过程：我们每次从字符串 p 中取出一个字符或者「字符 + 星号」的组合，并在 s 中进行匹配。对于 p 中一个字符而言，它只能在 s 中匹配一个字符，匹配的方法具有唯一性；而对于 p 中字符 + 星号的组合而言，它可以在 s 中匹配任意自然数个字符，并不具有唯一性。因此我们可以考虑使用动态规划，对匹配的方案进行枚举。

我们用 $f[i][j]$ 表示 s 的前 i 个字符与 p 中的前 j 个字符是否能够匹配。在进行状态转移时，我们考虑 p 的第 j 个字符的匹配情况：

- 如果 p 的第 j 个字符是一个小写字母，那么我们必须在 s 中匹配一个相同的小写字母，即

$$f[i][j] = \begin{cases} f[i-1][j-1], & s[i] = p[j] \\ \text{false}, & s[i] \neq p[j] \end{cases}$$

  也就是说，如果 s 的第 i 个字符与 p 的第 j 个字符不相同，那么无法进行匹配；否则我们可以匹配两个字符串的最后一个字符，完整的匹配结果取决于两个字符串前面的部分。

- 如果 p 的第 j 个字符是 `*`，那么就表示我们可以对 p 的第 $j-1$ 个字符匹配任意自然数次。在匹配 0 次的情况下，我们有

$$f[i][j] = f[i][j-2]$$

  也就是我们「浪费」了一个字符 + 星号的组合，没有匹配任何 s 中的字符。
  在匹配 $1, 2, 3, \cdots$ 次的情况下，类似地我们有

$$\begin{aligned} &f[i][j] = f[i-1][j-2], && \text{if } s[i] = p[j-1] \\ &f[i][j] = f[i-2][j-2], && \text{if } s[i-1] = s[i] = p[j-1] \\ &f[i][j] = f[i-3][j-2], && \text{if } s[i-2] = s[i-1] = s[i] = p[j-1] \\ &\cdots\cdots \end{aligned}$$

  如果我们通过这种方法进行转移，那么我们就需要枚举这个组合到底匹配了 s 中的几个字符，会增导致时间复杂度增加，并且代码编写起来十分麻烦。我们不妨换个角度考虑这个问题：字母 + 星号的组合在匹配的过程中，本质上只会有两种情况：

  - 匹配 s 末尾的一个字符，将该字符扔掉，而该组合还可以继续进行匹配；
  - 不匹配字符，将该组合扔掉，不再进行匹配。

  如果按照这个角度进行思考，我们可以写出很精巧的状态转移方程：

$$f[i][j] = \begin{cases} f[i-1][j] \text{ or } f[i][j-2], & s[i] = p[j-1] \\ f[i][j-2], & s[i] \neq p[j-1] \end{cases}$$

- 在任意情况下，只要 $p[j]$ 是 `.`，那么 $p[j]$ 一定成功匹配 s 中的任意一个小写字母。

最终的状态转移方程如下：

$$f[i][j] = \begin{cases} \text{if } (p[j] \neq \text{`*'}) = \begin{cases} f[i-1][j-1], & matches(s[i], p[j]) \\ \text{false}, & \text{otherwise} \end{cases} \\ \text{otherwise} = \begin{cases} f[i-1][j] \text{ or } f[i][j-2], & matches(s[i], p[j-1]) \\ f[i][j-2], & \text{otherwise} \end{cases} \end{cases}$$

其中 $matches(x, y)$ 判断两个字符是否匹配的辅助函数。只有当 y 是 `.` 或者 x 和 y 本身相同时，这两个字符才会匹配。

细节

动态规划的边界条件为 $f[0][0] = \text{true}$，即两个空字符串是可以匹配的。最终的答案即为 $f[m][n]$，其中 m 和 n 分别是字符串 s 和 p 的长度。由于大部分语言中，字符串的字符下标是从 0 开始的，因此在实现上面的状态转移方程时，需要注意状态中每一维下标与实际字符下标的对应关系。

在上面的状态转移方程中，如果字符串 p 中包含一个「字符 + 星号」的组合（例如 `a*`），那么在进行状态转移时，会先将 `a` 进行匹配（当 $p[j]$ 为 `a` 时），再将 `a*` 作为整体进行匹配（当 $p[j]$ 为 `*` 时）。然而，在题目描述中，我们**必须**将 `a*` 看成一个整体，因此将 `a` 进行匹配是不符合题目要求的。看来我们进行了额外的状态转移，这样会对最终的答案产生影响吗？这个问题留给读者进行思考。

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.size();
        int n = p.size();
```

```
        auto matches = [&](int i, int j) {//lambda 函数，表示下标 s[0..i-1] p[0..j-1] s[i-1]与p[j-1]是否相等
            if (i == 0) {//0 就是空串了，不能匹配
                return false;
            }
            if (p[j - 1] == '.') {//任意匹配
                return true;
            }
            return s[i - 1] == p[j - 1];//这俩是否匹配上了
        };
//星号换成
 if (!matches(i, j - 1)) {//匹配 0 次
                        f[i][j] = f[i ][j-2]
                    }
Else{
f[i][j]=(f[i][j-2]||f[i-1][j])}
//////////////////////////////
        vector<vector<int>> f(m + 1, vector<int>(n + 1));//默认 false
        f[0][0] = true;
        for (int i = 0; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (p[j - 1] == '*') {//当前数，不能单独一颗星
                    f[i][j] |= f[i][j - 2];//等于匹配一次和之前
                    if (matches(i, j - 1)) {//匹配 0 次+
                        f[i][j] |= f[i - 1][j];//匹配
                    }
                }
                else {
                    if (matches(i, j)) {//不是星号，如果能匹配，取决于前面那个
                        f[i][j] |= f[i - 1][j - 1];
                    }
                }
            }
        }
        return f[m][n];
    }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/regular-expression-matching/solution/zheng-ze-biao-da-shi-pi-pei-by-leetcode-solution/
来源：力扣（LeetCode）

69. x 的平方根（r,最后在左边，更小，返回呗）

难度简单 652
实现 int sqrt(int x) 函数。
计算并返回 x 的平方根，其中 x 是非负整数。
由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。
示例 1:
输入: 4 输出: 2
示例 2:
输入: 8 输出: 2 说明: 8 的平方根是 2.82842...,
由于返回类型是整数，小数部分将被舍去。
方法二：二分查找
由于 xx 平方根的整数部分 \textit{ans}ans 是满足 k^2 \leq xk
2
≤x 的最大 kk 值，因此我们可以对 kk 进行二分查找，从而得到答案。

二分查找的下界为 00，上界可以粗略地设定为 xx。在二分查找的每一步中，我们只需要比较中间元素 \textit{mid}mid 的平方与 xx 的大小关系，并通过比较的结果调整上下界的范围。由于我们所有的运算都是整数运算，不会存在误差，因此在得到最终的答案 \textit{ans}ans 后，也就不需要再去尝试 \textit{ans} + 1ans+1 了。

C++JavaPython3Golang

```
class Solution {
public:
    int mySqrt(int x) {
        int l = 0, r = x, ans = -1;
        while (l <= r) {
            int mid = l + (r - l) / 2;
            if ((long long)mid * mid <= x) {
                ans = mid;
                l = mid + 1;
            } else {
                r = mid - 1;
            }
        }
        return ans;//r 也行
    }
};
```

469. 凸多边形（相邻两条边，叉乘同号，用%n 防止越界一个 trick，考虑共线等于 0，就继续 pre=0，并且用一个=0 排除）

难度中等 21

给定一个按顺序连接的多边形的顶点，判断该多边形是否为凸多边形。（凸多边形的定义）

注：

顶点个数至少为 3 个且不超过 10,000。

坐标范围为 -10,000 到 10,000。

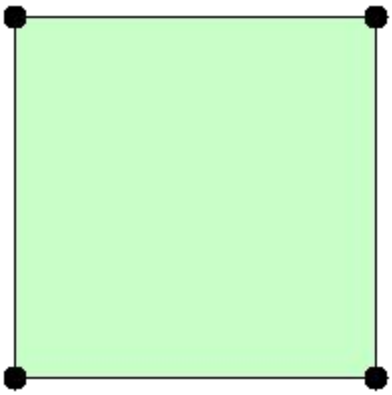
你可以假定给定的点形成的多边形均为简单多边形（简单多边形的定义）。换句话说，保证每个顶点处恰好是两条边的汇合点，并且这些边 互不相交 。

示例 1：

[[0,0],[0,1],[1,1],[1,0]]

输出： True

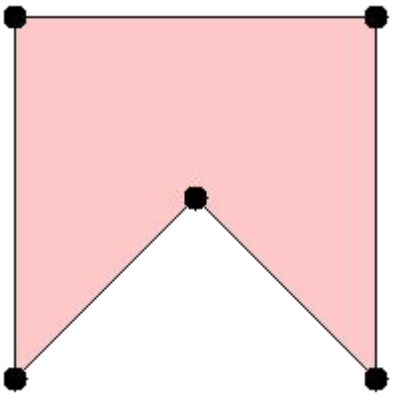
解释：



示例 2：

[[0,0],[0,10],[10,10],[10,0],[5,5]]

输出： False



解释：

解题思路

二货装 B 面试官出的题，你这是考 SDE 吗？利用凸多边形的性质，所有边的叉乘

* 使用向量叉积，

 * a = (x1,y1)

 * b = (x2,y2)

 * aXb = x1y2-x2y1

 * 叉积小于 0，那么 a 在 b 的逆时针方向，

 * 叉积大于 0，a 在 b 的顺时针方向

 * 叉积等于 0，a,b 方向相同

作者：lovXin

链接：https://leetcode-cn.com/problems/convex-polygon/solution/java-xiang-liang-nei-ji-fa-ru-men-ji-yue-du-nan-du/

来源：力扣（LeetCode）

代码

// The normal vector, often simply called the "normal," to a surface is a vector which is perpendicular to the surface at a given point. When normals are considered on closed surfaces, the inward-pointing normal (pointing towards the interior of the surface) and outward-pointing normal are usually distinguished.

```
class Solution {
public:
    bool isConvex(vector<vector<int>>& points) {
        long n = points.size();
        long pre = 0;
        long cur = 0;
        for (int i = 0; i < n; i++) {（或者是 i-n+1，用百分号来弄）
            int dx1 = points[(i + 1) % n][0] - points[i][0];//防止越界吗
            int dx2 = points[(i + 2) % n][0] - points[i][0];
            int dy1 = points[(i + 1) % n][1] - points[i][1];
            int dy2 = points[(i + 2) % n][1] - points[i][1];
```

```
                // {dx1, dy1} is vector 1, {dx2, dy2} is vector 2
                // we want to calculate. For convex polygon the dot
                // product of vector 1 and vector 2 should always be
                // positive (or always negative)
                cur = dx1 * dy2 - dx2 * dy1;//叉乘不共线=0 共线
                if (cur != 0) {
                    if (cur * pre < 0) {//同号相乘正数
                        return false;
                    } else {
                        pre = cur;
                    }
                }
            }

            return true;
        }
};
```

作者：jyj407
链接：https://leetcode-cn.com/problems/convex-polygon/solution/zhong-gui-zhong-ju-gao-zhong-shu-xue-by-hyesj/
来源：力扣（LeetCode）

最短路径问题---Dijkstra 算法详解

Ouyang_Lianjun 2017-03-08 16:42:46 536900 收藏 1920
分类专栏： 数据结构 文章标签： 算法 数据结构 最短路径问题 Dijkstra 算法

前言
Nobody can go back and start a new beginning,but anyone can start today and make a new ending.
Name:Willam
Time:2017/3/8

1、最短路径问题介绍

问题解释：
从图中的某个顶点出发到达另外一个顶点的所经过的边的权重和最小的一条路径，称为最短路径

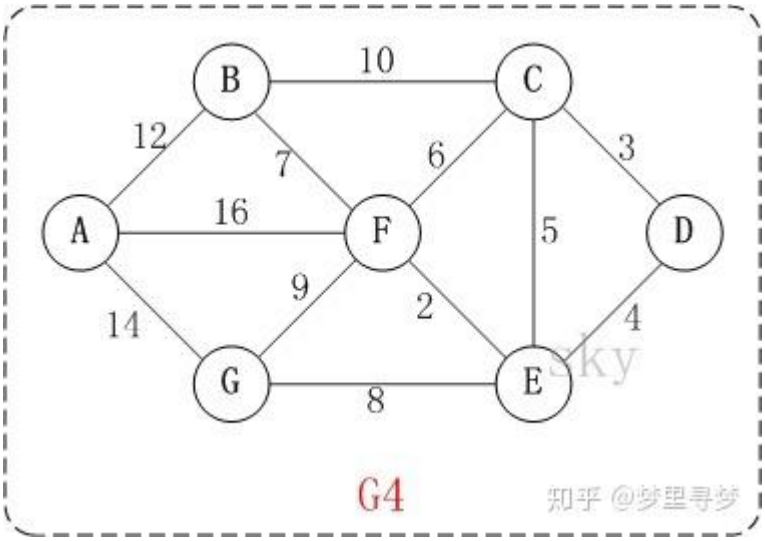
解决问题的算法：

迪杰斯特拉算法（Dijkstra 算法）
弗洛伊德算法（Floyd 算法）
SPFA 算法
这篇博客，我们就对 Dijkstra 算法来做一个详细的介绍

通俗易懂理解——dijkstra 算法求最短路径

集合 S 集合 u，一个距离数组，s 就是顶点，u 是备选的，每次选择最近的加入 s，那么顶点最短距离固定了，每次更新不是顶点的距离。

梦里寻梦
Future has arrived. It commences now.
关注他
365 人赞同了该文章
看这个算法的时候，虽然也是看到各种例子，但是对例子的说明，很多博客写的让我一脸懵，真为自己的智商感到着急。接下去我也将用一个例子来说明这个算法，希望初学者看到我的这篇可以更加浅显易懂。
先引用别人的关于该算法的定义，有耐心的可以看看，也可以直接跳到例子。
迪杰斯特拉(Dijkstra)算法是典型最短路径算法，用于计算一个节点到其他节点的最短路径。它的主要特点是以起始点为中心向外层层扩展(广度优先搜索思想)，直到扩展到终点为止。
基本思想
通过 Dijkstra 计算图 G 中的最短路径时，需要指定起点 s(即从顶点 s 开始计算)。
此外，引进两个集合 S 和 U。S 的作用是记录已求出最短路径的顶点(以及相应的最短路径长度，就是固定下来了)，而 U 则是记录还未求出最短路径的顶点(以及该顶点到起点 s 的距离，这里距离为固定下来)。
初始时，S 中只有起点 s；U 中是除 s 之外的顶点，并且 U 中顶点的路径是”起点 s 到该顶点的路径”。然后，从 U 中找出路径最短的顶点（贪心），并将其加入到 S 中；接着，更新 U 中的顶点和顶点对应的路径。 然后，再从 U 中找出路径最短的顶点，并将其加入到 S 中；接着，更新 U 中的顶点和顶点对应的路径。 … 重复该操作，直到遍历完所有顶点。
操作步骤
初始时，S 只包含起点 s；U 包含除 s 外的其他顶点，且 U 中顶点的距离为”起点 s 到该顶点的距离”[例如，U 中顶点 v 的距离为(s,v)的长度，然后 s 和 v 不相邻，则 v 的距离为∞]。
从 U 中选出”距离最短的顶点 k”，并将顶点 k 加入到 S 中；同时，从 U 中移除顶点 k。
更新 U 中各个顶点到起点 s 的距离。之所以更新 U 中顶点的距离，是由于上一步中确定了 k 是求出最短路径的顶点，从而可以利用 k 来更新其它顶点的距离；例如，(s,v)的距离可能大于(s,k)+(k,v)的距离。
重复步骤(2)和(3)，直到遍历完所有顶点。
单纯的看上面的理论可能比较难以理解，下面通过实例来对该算法进行说明。以 D 为开头，求 D 到各个点的最短距离。



第 1 步：初始化距离，其实指与 D 直接连接的点的距离。dis[c]代表 D 到 C 点的最短距离，因而初始 dis[C]=3，dis[E]=4，dis[D]=0，其余为无穷大。设置集合 S 用来表示已经找到的最短路径。此时，S={D}。现在得到 D 到各点距离{D(0)，C(3)，E（4），F（*），G（*），B（*），A(*)}，其中*代表未知数也可以说是无穷大，括号里面的数值代表 D 点到该点的最短距离。

第 2 步：不考虑集合 S 中的值，因为 dis[C]=3，是当中距离最短的，所以此时更新 S，S={D,C}。接着我们看与 C 连接的点，分别有 B，E，F，已经在集合 S 中的不看，dis[C-B]=10，因而 dis[B]=dis[C]+10=13， dis[F]=dis[C]+dis[C-F]=9， dis[E]=dis[C]+dis[C-E]=3+5=8>4(初始化时的 dis[E]=4)不更新。此时{D(0)，C(3)，E（4），F（9），G（*），B（13），A(*)}。

第 3 步：在第 2 步中，E 点的值 4 最小，更新 S={D，C，E}，此时看与 E 点直接连接的点，分别有 F， G。 dis[F]=dis[E]+dis[E-F]=4+2=6（比原来的值小，得到更新），dis[G]=dis[E]+dis[E-G]=4+8=12（更新）。此时{D(0)，C(3)，E（4），F（6），G（12），B（13），A(*)}。

第 4 步：在第 3 步中，F 点的值 6 最小，更新 S={D，C，E，F}，此时看与 F 点直接连接的点，分别有 B， A， G。 dis[B]=dis[F]+dis[F-B]=6+7=13， dis[A]=dis[F]+dis[F-A]=6+16=22，dis[G]=dis[F]+dis[F-G]=6+9=15>12（不更新）。此时{D(0)，C(3)，E（4），F（6），G（12），B（13），A(22)}.

第 5 步：在第 4 步中，G 点的值 12 最小，更新 S={D，C，E，F，G}，此时看与 G 点直接连接的点，只有 A。dis[A]=dis[G]+dis[G-A]=12+14=26>22(不更新)。{D(0)，C(3)，E（4），F（6），G（12），B（13），A(22)}.

第 6 步：在第 5 步中，B 点的值 13 最小，更新 S={D，C，E，F，G，B}，此时看与 B 点直接连接的点，只有 A。dis[A]=dis[B]+dis[B-A]=13+12=25>22(不更新)。{D(0)，C(3)，E（4），F（6），G（12），B（13），A(22)}.

第 6 步：最后只剩下 A 值，直接进入集合 S={D，C，E，F，G，B，A}，此时所有的点都已经遍历结束，得到最终结果{D(0)，C(3)，E（4），F（6），G（12），B（13），A(22)}.

相信看完以上内容都可以理解 dijkstra 算法，不过以上内容用文字表述就感觉看起来有些累赘，不过又没有比较好的作图工具，就勉强看着吧。

另外，我发现我的数据更新步骤跟其他人的不太一样，这也是我没有理解别人说的原因之一，我不明白他们的数值是如何更新的，虽然结果都一样。参考博客如下，作者的更新步骤就跟我不同，不

```
#include <stdio.h>
#include <vector>
#include <iostream>
using namespace::std;

#define INF 0x7fffffff
#define maxN 50
#pragma warning(disable:4996)

#define USE_C 1
#define NOT_USE_C 0

#define USE_CPP 1

int matrix[maxN][maxN];
// C++实现
void Dijkstra_cpp(vector<vector<int>>&vec, vector<int>& result, int v0){
    vector<int> visited(vec.size(), 0); // 表示顶点是否被选中，0：顶点未被选中；1：顶点已被选中
    int last_visited = 0;
    visited[v0] = 1; // 选中起始顶点
    result[0] = 0;

    for (int i = 0; i < vec.size() - 1; ++i) { // N 个顶点需要做 N - 1 次循环，最后一次不用更新
        // 查看顶点周围的所有点 松弛操作
        for (int j = 0; j < vec.size(); ++j) { // 循环遍历所有顶点
            if (visited[j] == 0){ // 保证被查看的新顶点没有被访问到
                if (vec[v0][j] != 0){ // 保证当前顶点（V0）与新顶点（j）之间有路径
                    int dist = vec[v0][j] + last_visited; // 计算 V0 到 J 的路径距离
                    if (dist < result[j])result[j] = dist; // 用新路径代替原来的路径
                }
            }
        }
        // 找出最小值
        int minIndex = 0;
        while (visited[minIndex] == 1) minIndex++; // 找第一个没有被选中的节点，从这开始
        for (int j = minIndex; j < vec.size(); ++j) {
            if (visited[j] == 0 && result[j] < result[minIndex]){//没有被选择，并且距离最小
                minIndex = j;
            }
        }
 //最小值
        last_visited = result[minIndex]; // 更新最小值的距离，就是要走的点
        visited[minIndex] = 1; // 将最小值顶点选中
```

```
            v0 = minIndex; // 下次查找从最限制顶点开始
        }
    }
```

743. 网络延迟时间(用 int_max 表示没有连接，或者没有初始化，小 trick，建立图，找最近点，松弛更新距离，visit, g, dis，其中 graph 的下标表示连接关系用 max 表示没有连接。)

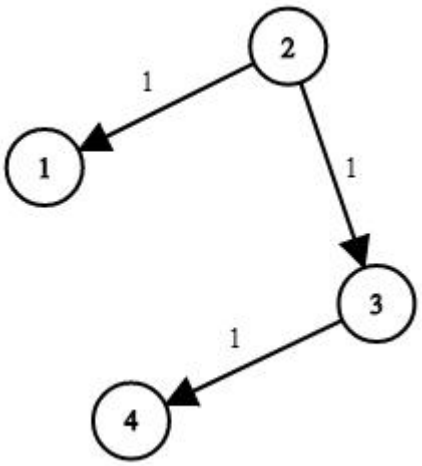
难度中等 245

有 n 个网络节点，标记为 1 到 n。

给你一个列表 times，表示信号经过 **有向** 边的传递时间。 times[i] = (ui, vi, wi)，其中 ui 是源节点，vi 是目标节点， wi 是一个信号从源节点传递到目标节点的时间。

现在，从某个节点 K 发出一个信号。需要多久才能使所有节点都收到信号？如果不能使所有节点收到信号，返回 -1 。

示例 1：



输入：`times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2` **输出：**`2`

示例 2：

输入：`times = [[1,2,1]], n = 2, k = 1` **输出：**`1`

示例 3：

输入：`times = [[1,2,1]], n = 2, k = 2` **输出：**`-1`

```
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<vector<int>> g(n + 1, vector<int>(n + 1, INT_MAX/2));//创建一个图
        vector<int> dis(n + 1, INT_MAX / 2), visit(n + 1, 0);//距离，和访问
        /*初始化邻接矩阵与 dis 数组*/
        for (auto& it : times) {
            g[it[0]][it[1]] = it[2];//这表示权重，时间
            if (it[0] == k)     dis[it[1]] = it[2];//起始点，开始的权重（顶点的权重）
        }
        /*源点 k 无需访问,dis[0]是个不需要的值,防止后面查找最大值错误*/
        dis[k] = 0, visit[k] = 1, dis[0] = 0;
        for (int cnt = 1; cnt < n; cnt++) {//最后一个 n 不取，不用更新
            /*找到离源点最短的点*/    1 找顶点
            int mi = INT_MAX/2, book = 0;
            for (int i = 1; i <= n; i++) {
                if (dis[i] < mi && !visit[i]) {
```

```
                    mi = dis[i];//最短距离，只能在 max 之外搜索
                    book = i;//索引
                }
            }

            /*如果源点无法到达任何一个点,直接返回*/
            if (mi == INT_MAX / 2)      return -1;//还是最大值找不到
            visit[book] = 1;//标记

            /*松弛操作,以 book 为中心点进行扩展*/ 2  松弛操作
            for (int i = 1; i <= n; i++) {
                /*如果 book 到 i 不为无穷，即有一条边的话，进行松弛操作*/这里，i 访问了，就不用
If(visit[i]) continue;'
                if (g[book][i] != INT_MAX/2 && dis[i] > dis[book] + g[book][i])//表明可达
                    dis[i] = dis[book] + g[book][i];
            }
        }
        /*答案处理*/
        int ret = *max_element(dis.begin(), dis.end());
        return ret == INT_MAX / 2 ? -1 : ret;
    }
};
```

作者：happysnaker
链接：https://leetcode-cn.com/problems/network-delay-time/solution/dirkdtra-by-happysnaker-vjii/
来源：力扣（LeetCode）

322. 零钱兑换（看看当前 amount，取当前硬币，之前的最小值。dp 表示组成金额最小的数字，每次遍历，选取一个当前的 coins[i]，与剩余最小的+1）

难度中等 1193

给定不同面额的硬币 *coins* 和一个总金额 *amount*。编写一个函数来计算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` **解释：**`11 = 5 + 5 + 1`

示例 2：

输入：`coins = [2], amount = 3` **输出：**`-1`

示例 3：

输入：`coins = [1], amount = 0` **输出：**`0`

示例 4：

输入：`coins = [1], amount = 1` **输出：**`1`

示例 5：

输入：`coins = [1], amount = 2` **输出：**`2`

方法二：动态规划
算法

我们采用自下而上的方式进行思考。仍定义 F(i)F(i) 为组成金额 ii 所需最少的硬币数量，假设在计算 F(i)F(i) 之前，我们已经计算出 F(0)到 F(i-1) 的答案。 则 F(i)F(i) 对应的转移方程应为

方法二：动态规划

算法

我们采用自下而上的方式进行思考。仍定义 $F(i)$ 为组成金额 i 所需最少的硬币数量，假设在计算 $F(i)$ 之前，我们已经计算出 $F(0)-F(i-1)$ 的答案。则 $F(i)$ 对应的转移方程应为

$$F(i)=\min_{j=0\ldots n-1} F(i-cj)+1$$

其中 cj 代表的是第 j 枚硬币的面值，即我们枚举最后一枚硬币面额是 cj，那么需要从 $i-cj$ 这个金额的状态 $F(i-cj)$ 转移过来，再算上枚举的这枚硬币数量 1 的贡献，由于要硬币数量最少，所以 $F(i)$ 为前面能转移过来的状态的最小值加上枚举的硬币数量 1 。

例子1：假设

```
coins = [1, 2, 5], amount = 11
```

则，当 $i==0$ 时无法用硬币组成，为 0 。当 $i<0$ 时，忽略 $F(i)$

F(i)	最小硬币数量
F(0)	0 //金额为0不能由硬币组成
F(1)	1 //$F(1)=min(F(1-1),F(1-2),F(1-5))+1=1$
F(2)	1 //$F(2)=min(F(2-1),F(2-2),F(2-5))+1=1$

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        int Max = amount + 1;
        vector<int> dp(amount + 1, Max);//表示这些钱中，组成 amount 最小的数值
        dp[0] = 0;
        for (int i = 1; i <= amount; ++i) {
            for (int j = 0; j < (int)coins.size(); ++j) {
                if (coins[j] <= i) {
                    dp[i] = min(dp[i], dp[i - coins[j]] + 1);//取 coins[j]需要几个
                }
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
};
```

869. 重新排序得到 2 的幂((a & (a - 1)) == 0,借一位，表示是 2 的幂，就是二进制上只有一位了，do while，

next_permutation，下一个全排列，不能有全导 0)

难度中等 40

给定正整数 *N* ，我们按任何顺序（包括原始顺序）将数字重新排序，注意其前导数字不能为零。

如果我们可以通过**上述方式得到 2 的幂**，返回 *true*；否则，返回 *false*。

示例 1：

输入：1 **输出：**true

示例 2：

输入：10 **输出：**false

示例 3：

输入：16 **输出：**true

示例 4：

输入：24 **输出：**false

示例 5：

输入：46 输出：true
next_permutation 产生全排列

```
class Solution {
public:
    bool reorderedPowerOf2(int n) {
        string s = to_string(n);

        sort(s.begin(), s.end());
```

```
        do{
            if(check(s))
                return true;
        }while(next_permutation(s.begin(), s.end()));//可认为字符串的字典序
        return false;
    }

    bool check(string &s){
        if(s.length() > 1 && s[0] == '0')
            return false;
        int a = stoi(s);
        return (a & (a - 1)) == 0;//每次借的位数，肯定到自身，就是表面，只有一个位置有数，就是 2 的幂了。
    }
};
```

110101

和十进制计算一样不够减向前面借，十进制向前借一位是 10，二进制是 2 所以 110110 减一后为 110101.

按照 STL 文档的描述，next_permutation 函数将按字母表顺序生成给定序列的下一个较大的排列，直到整个序列为降序为止。prev_permutation 函数与之相反，是生成给定序列的上一个较小的排列。

这是一个求一个排序的下一个排列的函数，可以遍历全排列,要包含头文件 <algorithm>

使用方法：next_permutation(数组头地址，数组尾地址);若下一个排列存在，则返回真，如果不存在则返回假

若求上一个排列，则用 prev_permutation

作者：Raymond_YP
链接：
https://leetcode-cn.com/problems/reordered-power-of-2/solution/c-wei-yun-suan-by-raymond_yp-bzn8/
来源：力扣（LeetCode）

按照 STL 文档的描述，next_permutation 函数将按字母表顺序生成给定序列的下一个较大的排列，直到整个序列为降序为止。prev_permutation 函数与之相反，是生成给定序列的上一个较小的排列。

这是一个求一个排序的下一个排列的函数，可以遍历全排列,要包含头文件<algorithm>

使用方法：next_permutation(数组头地址，数组尾地址);若下一个排列存在，则返回真，如果不存在则返回假

若求上一个排列，则用 prev_permutation

————————————

do-while 与 while-do 区别为：跳出循环不同、执行次数不同、优先操作不同。

一、跳出循环不同

1、do-while：do-while 不可以通过 break 在循环过程中跳出。

2、while-do：while-do 可以通过 break 在循环过程中跳出。

二、执行次数不同

1、do-while：do-while 至少会执行一次循环体。（先执行）

2、while-do：while-do 可能会出现一次都不执行循环体的情况。

三、优先操作不同

1、do-while：do-while 优先执行循环体，再判断执行条件是否符合要求。

2、while-do：while-do 优先判断执行条件是否符合要求，再执行循环体。

1550. 存在连续三个奇数的数组（直接暴力&1=1 就是奇数）

难度简单 8

给你一个整数数组 *arr*，请你判断数组中**是否存在**连续三个元素都是奇数的情况：如果存在，请返回 *true* ；否则，返回 *false* 。

示例 1：

输入：arr = [2,6,4,1]**输出：**false **解释：**不存在连续三个元素都是奇数的情况。

示例 2：

输入：arr = [1,2,34,3,4,5,7,23,12]**输出：**true **解释：**存在连续三个元素都是奇数的情况，即 [5,7,23] 。

方法一：枚举
思路与算法

枚举所有的连续的三个元素，判断这三个元素是否都是奇数，如果是，则返回 true。如果所有的连续的三个元素中，没有一个满足条件，返回 false。

代码

C++JavaJavaScriptPython3

```
class Solution {
public:
    bool threeConsecutiveOdds(vector<int>& arr) {
        int n = arr.size();
        for (int i = 0; i <= n - 3; ++i) {
            if ((arr[i] & 1) & (arr[i + 1] & 1) & (arr[i + 2] & 1)) {
                return true;
            }
        }
        return false;
    }
};
```

作者：LeetCode-Solution
链接：
https://leetcode-cn.com/problems/three-consecutive-odds/solution/cun-zai-lian-xu-san-ge-qi-shu-de-shu-zu-by-leetcod/
来源：力扣（LeetCode）

写入 AB（优先先数字更大的，除非前面有俩了，就写另外一个）

三个不能相连，优先写数字更大的。
算法思路
这是一道简单的贪心算法
由样例可知，把握以下两个思路编写算法：

在允许情况下，先写入当前字母数量多的字母 x。（否则，会出现大量字母连在一起的情况，不符合题目要求）
如果前 2 个字母已经是 x 的情况下，则写入另一个字母。
复杂度分析
时间复杂度：O(A+B)
循环进行 A+B 次
空间复杂度：O(A+B)
代码

```
string strWithout3a3b(int A, int B) {
        string s;
        int atemp = 0, btemp = 0;
        int temp = A + B;
        while (s.size() < temp)，表示满了
        {
                if (A > B && atemp < 2 || A <= B && btemp == 2)
                {
                        s.push_back('a');
                        A--;
                        atemp++;
                        btemp = 0;
                }
                else
                {
                        s.push_back('b');
                        B--;
                        atemp = 0;
                        btemp++;
                }
        }
        return s;
}
```

作者：Smart_Shelly
链接：
https://leetcode-cn.com/problems/string-without-aaa-or-bbb/solution/984-c-jian-dan-tan-xin-de-zi-fu-chuan-chu-li-by-sm/
来源：力扣（LeetCode）

402. 移掉 K 位数字（单调递增栈，如果进来的数字，大于栈顶的数字，删除栈

顶元素，如果删除之前的话，后面更大的数，会替换他，会比当前还大，字典序。所以删除栈顶的，更小的替换他，用 vector 模拟一个栈）

难度中等 548

给定一个以字符串表示的非负整数 num，移除这个数中的 k 位数字，使得剩下的数字最小。

注意:

- num 的长度小于 10002 且 ≥ k。
- num 不会包含任何前导零。

示例 1：

输入：num = "1432219", k = 3

输出："1219"

解释：移除掉三个数字 4, 3, 和 2 形成一个新的最小的数字 1219。

示例 2：

输入：num = "10200", k = 1

输出："200"

解释：移掉首位的 1 剩下的数字为 200. 注意输出不能有任何前导零。

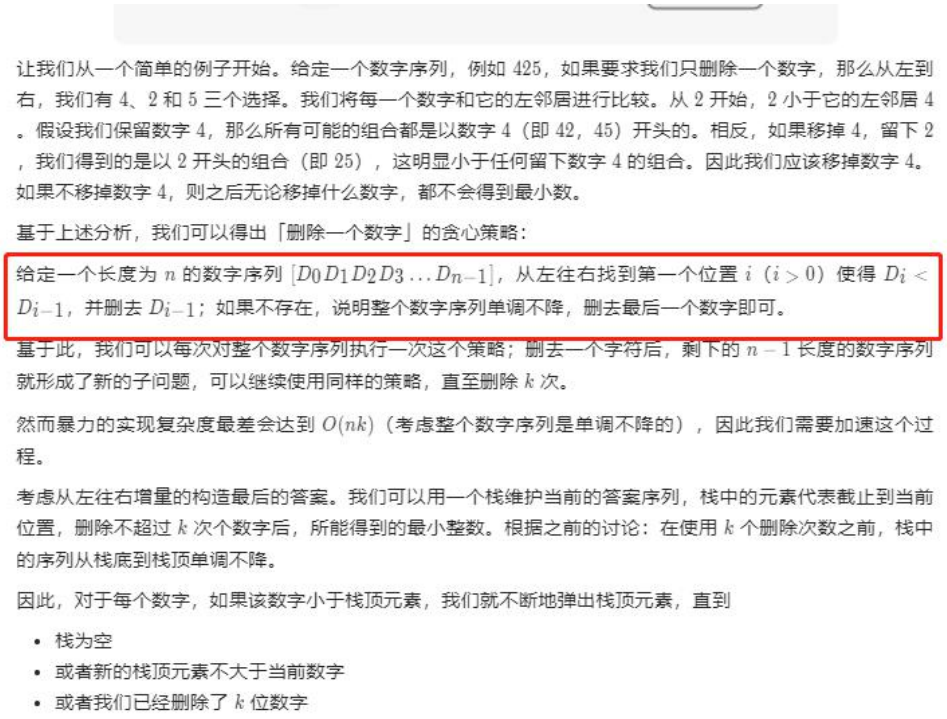
示例 **3：**

输入：num = "10", k = 2

输出："0"

解释：从原数字移除所有的数字，剩余为空就是 0。

基于上述分析，我们可以得出「删除一个数字」的贪心策略：



让我们从一个简单的例子开始。给定一个数字序列，例如 425，如果要求我们只删除一个数字，那么从左到右，我们有 4、2 和 5 三个选择。我们将每一个数字和它的左邻居进行比较。从 2 开始，2 小于它的左邻居 4。假设我们保留数字 4，那么所有可能的组合都是以数字 4（即 42，45）开头的。相反，如果移掉 4，留下 2，我们得到的是以 2 开头的组合（即 25），这明显小于任何留下数字 4 的组合。因此我们应该移掉数字 4。如果不移掉数字 4，则之后无论移掉什么数字，都不会得到最小数。

基于上述分析，我们可以得出「删除一个数字」的贪心策略：

给定一个长度为 n 的数字序列 $[D_0D_1D_2D_3\dots D_{n-1}]$，从左往右找到第一个位置 i（$i>0$）使得 $D_i < D_{i-1}$，并删去 D_{i-1}；如果不存在，说明整个数字序列单调不降，删去最后一个数字即可。

基于此，我们可以每次对整个数字序列执行一次这个策略；删去一个字符后，剩下的 $n-1$ 长度的数字序列就形成了新的子问题，可以继续使用同样的策略，直至删除 k 次。

然而暴力的实现复杂度最差会达到 $O(nk)$（考虑整个数字序列是单调不降的），因此我们需要加速这个过程。

考虑从左往右增量的构造最后的答案。我们可以用一个栈维护当前的答案序列，栈中的元素代表截止到当前位置，删除不超过 k 次个数字后，所能得到的最小整数。根据之前的讨论：在使用 k 个删除次数之前，栈中的序列从栈底到栈顶单调不降。

因此，对于每个数字，如果该数字小于栈顶元素，我们就不断地弹出栈顶元素，直到

- 栈为空
- 或者新的栈顶元素不大于当前数字
- 或者我们已经删除了 k 位数字

；如果不存在，说明整个数字序列单调不降，删去最后一个数字即可。

基于此，我们可以每次对整个数字序列执行一次这个策略；删去一个字符后，剩下的 n-1n-1 长度的数字序列就形成了新的子问题，可以继续使用同样的策略，直至删除 kk 次。

然而暴力的实现复杂度最差会达到 O(nk)O(nk)（考虑整个数字序列是单调不降的），因此我们需要加速这个过程。

考虑从左往右增量的构造最后的答案。我们可以用一个栈维护当前的答案序列，栈中的元素代表截止到当前位置，删除不超过 kk 次个数字后，所能得到的最小整数。根据之前的讨论：在使用 kk 个删除次数之前，栈中的序列从栈底到栈顶单调不降。

因此，对于每个数字，如果该数字小于栈顶元素，我们就不断地弹出栈顶元素，直到

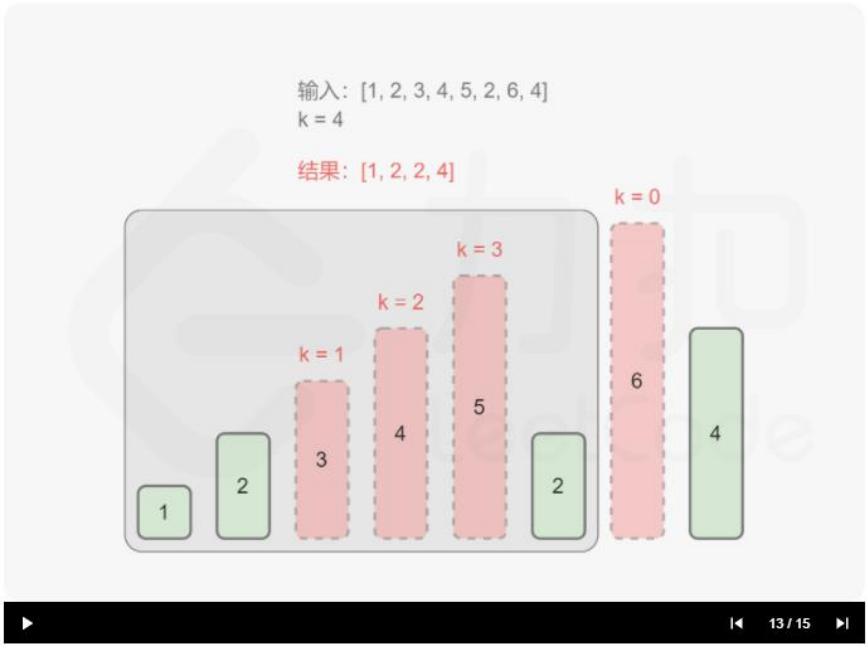
栈为空
或者新的栈顶元素不大于当前数字
或者我们已经删除了 kk 位数字

上述步骤结束后我们还需要针对一些情况做额外的处理：

如果我们删除了 m 个数字且 m<k 这种情况下我们需要从序列尾部删除额外的 k-m 个数字。

如果最终的数字序列存在前导零，我们要删去前导零。
如果最终数字序列为空，我们应该返回 0。
最终，从栈底到栈顶的答案序列即为最小数。

考虑到栈的特点是后进先出，如果通过栈实现，则需要将栈内元素依次弹出然后进行翻转才能得到最小数。为了避免翻转操作，**可以使用双端队列代替栈的实现。**



上述步骤结束后我们还需要针对一些情况做额外的处理：

- 如果我们删除了 m 个数字且 $m < k$，这种情况下我们需要从序列尾部删除额外的 $k - m$ 个数字。

```
class Solution {
public:
    string removeKdigits(string num, int k) {
        vector<char> stk;用一个 vector,模拟栈
        for (auto& digit: num) {
            while (stk.size() > 0 && stk.back() > digit && k) {
                stk.pop_back();
                k -= 1;
            }
            stk.push_back(digit);
        }

        for (; k > 0; --k) {//判断不为空吧
            stk.pop_back();
        }

        string ans = "";
        bool isLeadingZero = true;
        for (auto& digit: stk) {
            if (isLeadingZero && digit == '0') {//看第一个
```

```
                continue;
            }
            isLeadingZero = false;
            ans += digit;
        }
        return ans == "" ? "0" : ans;
    }
};
```

作者：LeetCode-Solution
链接：
https://leetcode-cn.com/problems/remove-k-digits/solution/yi-diao-kwei-shu-zi-by-leetcode-solution/
来源：力扣（LeetCode）

1401. 圆和矩形是否有重叠（找矩形上距离圆心最近的点的距离是否小于等于r，分圆心在哪里，矩形左边，中间，右边，与圆心 xy 最小的距离 dx dy）

难度中等 23

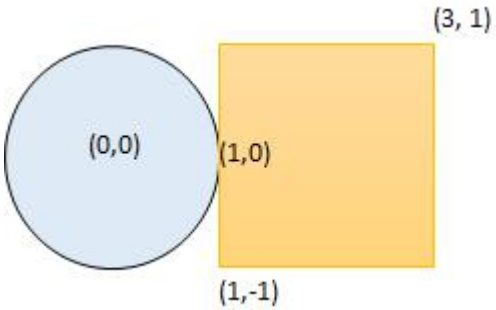
给你一个以 (*radius*, *x_center*, *y_center*) 表示的圆和一个与坐标轴平行的矩形 (*x1*, *y1*, *x2*, *y2*)，

其中 (*x1*, *y1*) 是矩形左下角的坐标，(*x2*, *y2*) 是右上角的坐标。

如果圆和矩形有重叠的部分，请你返回 True ，否则返回 False 。

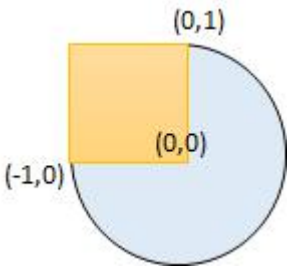
换句话说，请你检测是否 **存在** 点 (xi, yi) ，它既在圆上也在矩形上（两者都包括点落在边界上的情况）。

示例 1：



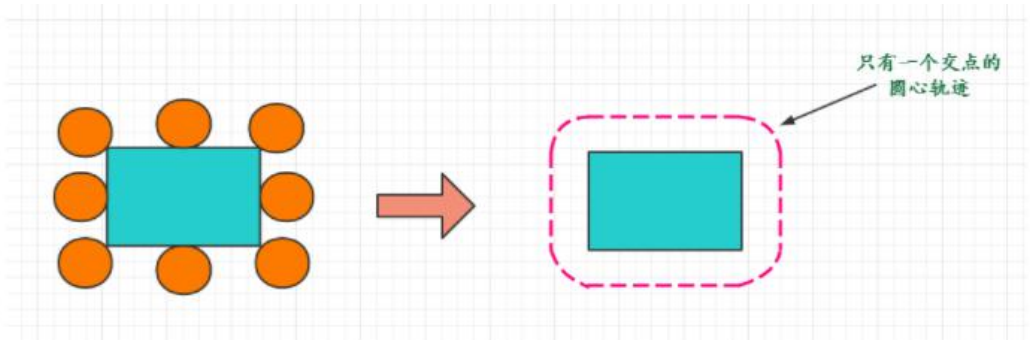
输入：`radius = 1, x_center = 0, y_center = 0, x1 = 1, y1 = -1, x2 = 3, y2 = 1` **输出：**`true` **解释：**圆和矩形有公共点 `(1,0)`

示例 2：



输入：`radius = 1, x_center = 0, y_center = 0, x1 = -1, y1 = 0, x2 = 0, y2 = 1` **输出：**`true`

解题思路
这个题目方法还不少，我这里说一种比较简单的，考虑一个交点的圆心轨迹：



从这里的图中，可以看出该圆心轨迹和矩形的最近距离为 `raduis`

在矩形四条边的范围内，轨迹和边的直线距离为 `radius`
在矩形的四个顶点附近，轨迹是一个以顶点为圆心，半径为 `radius` 的 `1/4` 圆

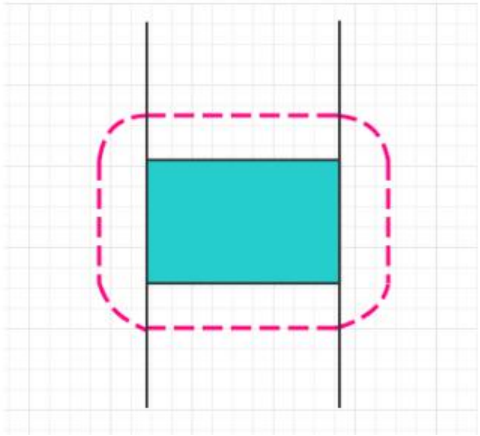
所以只要给定的圆心在该轨迹内部（包括轨迹上），那么圆和矩形就会出现交集；而圆心在轨迹外部，则不会出现交集。

判断的方法：

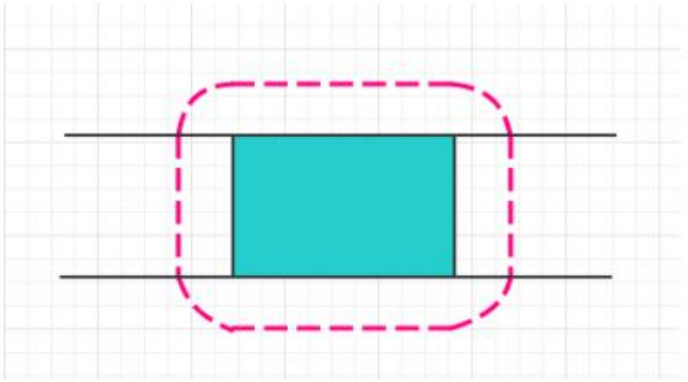
当圆心 x_centre 位于下图的左右范围内，圆心 y_centre 要在圆心轨迹的上下边界内，即 [y1 - radius, y2 + raduis]

判断的方法：

- 当圆心 x_centre 位于下图的左右范围内，圆心 y_centre 要在圆心轨迹的上下边界内，即 [y1 - radius, y2 + raduis]



当圆心 y_centre 位于下图的上下范围内，圆心 x_centre 要在圆心轨迹的左右边界内，即 [x1 - radius, x2 + raduis]



其他的情况时，则需要判断圆心和四个顶点中的最短距离和圆心的孰大孰小

```
int delta_x = min(abs(x_center - x1), abs(x_center - x2));
int delta_y = min(abs(y_center - y1), abs(y_center - y2));
return delta_x * delta_x + delta_y * delta_y <= radius * radius;
```

完整代码

```
class Solution {
```

```
public:
    bool checkOverlap(int radius, int x_center, int y_center, int x1, int y1, int
x2, int y2) {
        if(x_center >= x1 && x_center <= x2){
            return y_center >= (y1 - radius) && y_center <= (y2 + radius);
        }else if(y_center >= y1 && y_center <= y2){
            return x_center >= (x1 - radius) && x_center <= (x2 + radius);
        }else{
            int delta_x = min(abs(x_center - x1), abs(x_center - x2));
            int delta_y = min(abs(y_center - y1), abs(y_center - y2));
            return delta_x * delta_x + delta_y * delta_y <= radius * radius;
        }
    }
};
```

欢迎指出不足~

作者：zuo-10
链接：
https://leetcode-cn.com/problems/circle-and-rectangle-overlapping/solution/jian-dan-tu-jie-zhao-yi-ge-jiao-dian-de-yuan-xin-g/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

基本思路：矩形区域内点离圆心最近的点距离小于等于圆的半径

矩形区域特点是 x,y 取值相互独立
又距离公式对 x,y 分别单调
分别取最小距离即可

又由 x1 < x2 , y1 < y2 可以简化最小距离获取过程
对于 x
x_center < x1 < x2 =>min(dx) = x1 - x_center
x1 <= x_center <= x2 =>min(dx) = 0
x1 < x2 < x_center =>min(dx) = x_center - x2
对于 y 同理

javascriptcppcjavacsharppypython3golangrubyswiftscalakotlinrustphp

```
class Solution {
public:
    bool checkOverlap(int radius, int x_center, int y_center, int x1, int y1, int
x2, int y2) {
        int dx = x1 > x_center ? x1 - x_center : x2 < x_center ? x_center - x2 : 0;
        int dy = y1 > y_center ? y1 - y_center : y2 < y_center ? y_center - y2 : 0;
        return dx * dx + dy * dy <= radius * radius;
```

```
    }
};
```

作者：ayaphis
链接：
https://leetcode-cn.com/problems/circle-and-rectangle-overlapping/solution/yan-zheng-ju-xing-qu-yu-nei-de-dian-dao-yuan-xin-z/
来源：力扣（LeetCode）

233. 数字 1 的个数（要死，这个题目，找规律，数学归纳法，每个位数 1 的个数是多少，n/d*10+min(max(n%d-i+1, 0L), i)）

难度困难 210

给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数。

示例 1：

输入： n = 13 **输出：** 6

示例 2：

输入：n = 0 输出：0
解题思路
优化的解法是求出 1~n 数字中每位出现 1 的次数相加。

我们举个例子先：12345

首先个位出现 1 的情况有 1234 + 1 种，因为只要个位为 1 即可，高位可以随便取，为什么 +1 呢，是因为可以取 0。

十位出现 1 的情况有 1239 + 1 种

同理百位出现 1 的情况有 1299 + 1 种，千位 1999 + 1 种，万位 2345 + 1 种。

那么 1 的总的出现次数为 1235 + 1240 + 1300 + 2000 + 2346 = 8121

我们举得例子中每位都大于 0 ，如果等于 0 的话

12045

第三位为 0，那么第三位出现 1，其他位的取值范围位 [0， 1199]，那么总次数为 1200

综上，我们可以得出一个结论

先定义几个变量，cur 表示当前位，idx 表示当前的位数(从 0 开始算)，left 表示高位的值，right 表示低位的值，

1 2 3 4 5

  cur

如果当前位为 1，当前位出现 1 的次数为 left*10^idx + right + 1

如果当前位为 0，当前位出现 1 的次数为 left*(10^idx)

其余情况，当前位出现 1 的次数为 (left+1)*(10^idx)

其实就是相当于除去当前位其他位一共可以组成多大的数字。应该不难理解。

代码

```
class Solution {
public:
    int countDigitOne(int n) {
        if(n < 0) return 0;
        int ans = 0;
        int left, right = 0, idx = 0, cur = 0;
        while(n){
            cur = n % 10;
            left = n/10;
            if(cur == 0){
                ans += left*pow(10, idx);
            }else if(cur == 1){
                ans += left*pow(10, idx) + right + 1;
            }else{
                ans += (left + 1)*pow(10, idx);
```

```
            }
            right += cur*pow(10, idx);
            n /= 10;
            idx ++;
        }
        return ans;
    }
};
```

作者：yizhe-shi

链接：
https://leetcode-cn.com/problems/number-of-digit-one/solution/c-dui-mei-wei-jin-xing-fen-xi-by-yizhe-shi-2/

来源：力扣（LeetCode）

解题思路

数学归纳法

由上图所示，我们可以观察到每 10 个数，个位上的 '1' 就会出现一次。同样的，每 100 个数，十位上的 '1' 就会出现一次。这个规律可以用 $(n/(i*10))*i$ 公式来表示。

同时，如果十位上的数是 '1'，那么最后 '1' 的数量要加上 $x+1$，其中 x 是个位上的数值。如果十位上的数大于 '1'，那么十位上为 '1' 的所有的数都是符合要求的，这时候最后 '1' 的数量要加 10。

这个规律可以用公式 $\min(\max((\text{n mod } (\text{i}*10)) - i + 1, 0), i)$ 来表示。

我们来看一个例子吧，有一个数 $n = 1234$。

个位上 '1' 的数量 = 1234/10 (对应 1,11,21,...1221) + $\min(4, 1)$ (对应 1231) = 124

十位上 '1' 的数量 = $(1234/100)*10$ (对应 10,11,12,...,110,111,...1919) + $\min(21, 10)$ (对应 1210,1211,...1219) = 130

百位上 '1' 的数量 = $(1234/1000)*100$ (对应 100,101,102,...,199) + $\min(135, 100)$ (对应1100,1101...1199) = 200

千位上 '1' 的数量 = $(1234/10000)*10000$ + $\min(235, 1000)$ (对应1000,1001,...1234) = 235

因此，总数 = $124 + 130 + 200 + 235 = 689$。

算法

- 将 i 从 1 遍历到 n，每次遍历 i 扩大 10 倍：
  - $(n/(i*10))*i$ 表示 $(i*10)$ 位上 '1' 的个数。
  - $\min(\max((\text{n mod } (\text{i}*10)) - i + 1, 0), i)$ 表示需要额外数的 $(i*10)$ 位上 '1' 的个数。

每 10 位数，个位数会多 1，同理，每 100，十位数+1,每个位数，都是从 0 到 n 筛选

计算 1 在 个数， 十位，百位的对应的个数
（1） 个位 1 11 21 31 ... 91 -> n/10 + (n%10 != 0) 各位能否取到 1 11
（2） 十位
100 10 个 1
200 20 个 1
1600 1610=160 个 1
1610 1610 + 1 = 161 个 1
161x 161 + x 个 1
(考虑最大只能是 10 个)
--> (n/100)*10 + min(10, max(n%100 - 10 + 1, 0))
（3） 百位类似总结
(n/1000)*100 + min(100, max(n%1000 - 100 + 1, 0))

定义 i 表示对应的位数，如百位 ，i=100， 十位则 i=10 那么对应的位数下 1 的个数为： (n / (i*10))i + **min(max((n%(i*10))-i+1, 0), i)**

知道规律后，就是分别 i 从 1， 10， 100 递增直到大于 n 则结束，不断计算公式得到结果即可

代码

```
class Solution {
public:
    int countDigitOne(int n) {
        int res = 0;
        long d;
        for (long i = 1;  i <= n; i *= 10)
        {
            d = i * 10;
            res += (n / d) * i + min(max((n % d - i + 1), 0L), i);
        }
```

先看看，各位上按照每 10 个的变化，之后看看隐藏了多少，顶多就是 10 个假设（十位）11 19 111，这三个，十位相减+1，就可以 （大于 19 又是一个 111 循环了）

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

C 语言中 0L 是什么数据？

botng004 2016.04.10 浏览 354 次 其他分享 举报

1 个回答

满意答案

liujunjie528

2016.04.10

f(0L);

等价于

long a=0;

f(n);

同理有 1L 1234L 等等写法。

是为了让编译器不要算错。

作者：ffreturn
链接：
https://leetcode-cn.com/problems/number-of-digit-one/solution/cshuang-bai-de-ji-yu-shu-xue-tui-dao-de-6j6qe/
来源：力扣（LeetCode）

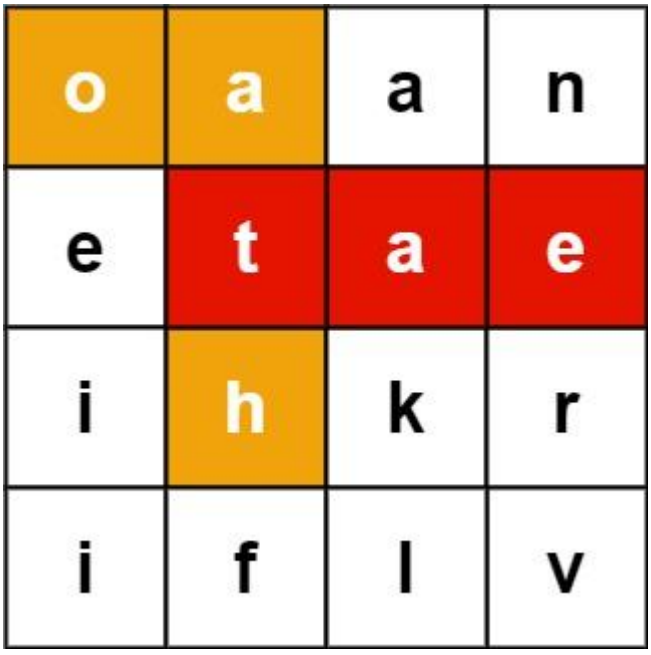
212. 单词搜索 II（字典树，先将单词加入进去，之后搜索，看里面是否有下一个，并且有标志位 string，还有个回溯，.表示访问过了，root->[c]!=nullptr 不存在，或者等于.访问过了）

难度困难 365

给定一个 *m* x *n* 二维字符网格 *board* 和一个单词（字符串）列表 *words*，找出所有同时在二维网格和字典中出现的单词。

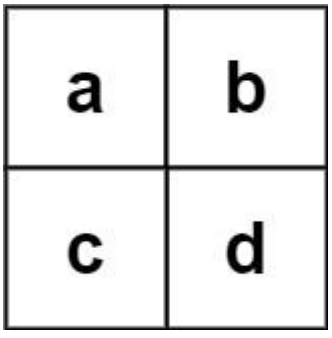
单词必须按照字母顺序，通过 **相邻的单元格** 内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。

示例 1：



输入：`board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]`**输出：**`["eat","oath"]`

示例 2：



输入：board = [["a","b"],["c","d"]], words = ["abcb"]**输出：**[]

提示：

- *m == board.length*
- *n == board[i].length*
- *1 <= m, n <= 12*
- *board[i][j]* 是一个小写英文字母
- *1 <= words.length <= 3 * 104*
- *1 <= words[i].length <= 10*
- *words[i]* 由小写英文字母组成
- *words* 中的所有字符串互不相同

通过次数 33,046 提交次数 72,528

```
class TrieNode{
public:
    string word = "";
    vector<TrieNode*> nodes;
    TrieNode():nodes(26, 0){}
};

class Solution {
    int rows, cols;
    vector<string> res;
public:
    vector<string> findWords(vector<vector<char>>& board, vector<string>& words)
{
```

```
        rows = board.size();
        cols = rows ? board[0].size():0;
        if(rows==0 || cols==0) return res;

        //建立字典树的模板
        TrieNode* root = new TrieNode();
        for(string word:words){
            TrieNode *cur = root;
            for(int i=0; i<word.size(); ++i){
                int idx = word[i]-'a';
                if(cur->nodes[idx]==0) cur->nodes[idx] = new TrieNode();
                cur = cur->nodes[idx];
            }
            cur->word = word;
        }

        //DFS 模板
        for(int i=0; i<rows; ++i){
            for(int j=0; j<cols; ++j){
                dfs(board, root, i, j);
            }
        }
        return res;
    }

    void dfs(vector<vector<char>>& board, TrieNode* root, int x, int y){
        char c = board[x][y];
        //递归边界
        if(c=='.' || root->nodes[c-'a']==0) return;
        root = root->nodes[c-'a'];
        if(root->word!=""){
            res.push_back(root->word);
            root->word = "";
        }

        board[x][y] = '.';
        if(x>0) dfs(board, root, x-1, y);
        if(y>0) dfs(board, root, x, y-1);
        if(x+1<rows) dfs(board, root, x+1, y);
        if(y+1<cols) dfs(board, root, x, y+1);
        board[x][y] = c;
    }
};
```

作者：talanto_linyi
链接：
https://leetcode-cn.com/problems/word-search-ii/solution/c-jian-dan-qing-xi-de-trieshu-ti-jie-by-talanto_li/
来源：力扣（LeetCode）

【C++字典树 Trie 全文注释】剪枝 0ms100%

笑融君 L2

发布于 2021-01-071.1k 字典树 C++回溯

- 这里提供一个 C++适合该题的字典树结构，记录一下我遇到的坑
- 字典树向上回溯剪枝，技巧上最好是逻辑地删除，物理直接 delete 可能造成野指针异常
- 逻辑上删除就是节点控制字段置-1
- 代码已经全文注释了，具体就看代码吧！

```
// 字典树
class TrieNode {
private:

   TrieNode* next[26];     // 孩子列表[0-25]表示['a'-'z']，nullptr 表示无
   TrieNode* parent;       // 该结点的父结点/双亲结点
   char flag;              // 控制标记。-1 逻辑已删除； 0：作为中间链路；1：有效单词结束
   int numsOfNext;         // 当前结点的孩子个数，叶结点为 0

public:
   // 构造函数
   // param: [bool] isRoot - 初始化的根结点需要置 true
   TrieNode(bool isRoot = false) {
       for (int i = 0; i < 26; ++i) next[i] = nullptr;
       flag = isRoot;
       parent = nullptr;
       numsOfNext = 0;
   }
```

```
    // 在字典树的根结点插入单词记录 (若存在则跳过，默认在单词结尾的字符结点置其控制标记 flag 为 1)
    // param: [TrieNode*] root - 一般传字典树根结点
    // param: [string] word - 单词
    // return: [TrieNode*] - 返回插入的单词末尾字符结点
    static TrieNode* insert(TrieNode* root, string word) {
        auto ptr = root;
        for (char c : word) {
            if (ptr->next[c-'a'] == nullptr) {
                ptr->next[c-'a'] = new TrieNode();
                ptr->next[c-'a']->parent = ptr;
                ++ptr->numsOfNext;
            }
            ptr = ptr->next[c-'a'];
        }
        ptr->flag = true;
        return ptr;
    }

    // 在字典树的某个结点插入下一个字符结点 (若存在则跳过，需要指定其是否是单词结尾)
    // param: [TrieNode*] root - 字典树的某个结点
    // param: [char] c - 字符
    // param: [bool] isEnd - 是否以此为结尾够成新单词
    // return: [TrieNode*] - 返回插入的新字符结点
    static TrieNode* insert(TrieNode* root, char c, bool isEnd) {
        if (root->next[c-'a'] == nullptr) {
            root->next[c-'a'] = new TrieNode();
            root->next[c-'a']->parent = root;
            ++root->numsOfNext;
        }
        root->next[c-'a']->flag = isEnd;
        return root->next[c-'a'];
    }

    // 在字典树的根个结点查找单词记录 (需满足单词结尾的字符结点其控制标记 flag 为 1)
    // param: [TrieNode*] root - 一般传字典树根结点
    // param: [string] word - 单词
    // return: [bool] - 是否存在
    static bool has(TrieNode* root, string word) {
        if (!root) return false;
        auto ptr = root;
        for (char c : word) {
            if (ptr->next[c-'a'] == nullptr || ptr->next[c-'a']->flag == -1) {
                return false;
```

```
            }
            ptr = ptr->next[c-'a'];
        }
        return ptr->flag == 1;
    }

    // 在字典树的某结点查找是否存在下一个指定的字符
    // param: [TrieNode*] root - 字典树的某结点
    // param: [char] c - 查找的字符
    // return: [bool] - 是否存在
    static bool has(TrieNode* root, char c) {
        if (!root) return false;
        if (root->next[c-'a'] == nullptr || root->next[c-'a']->flag == -1) {
            return false;
        }
        return true;
    }

    // 逻辑上删除(flag 置-1)，在字典树的某子结点开始向上追溯父结点/双亲结点并删除，若父结点/双亲结点有其他分叉或其 flag 为 1(作为单词结尾)则停止删除
    // param: [TrieNode*] root - 字典树的某结点
    // return: [TrieNode*] - 返回停止删除的父结点
    static TrieNode* del_up_logicly(TrieNode* root) {
        auto cur = root;
        do {
            cur->flag = -1;
            cur = cur->parent;
            --cur->numsOfNext;

        } while (cur->flag <= 0 && cur->numsOfNext == 0);
        return cur;
    }

    //=========================================================
    // setter & getter 不做详细注释

    // 返回当前结点的下一个指定字符结点
    static TrieNode* getNextNode(TrieNode* root, char c) {
        return root->next[c-'a'];
    }

    // 返回当前结点的父结点/双亲结点
    static TrieNode* getParentNode(TrieNode* root) {
        return root->parent;
```

```
    }

    // 返回当前结点是否是单词结尾
    static bool isEnd(TrieNode* root) {
        return root->flag == 1;
    }

    // 返回当前结点的分支 next 数量
    static int getNumsOfNext(TrieNode* root) {
        return root->numsOfNext;
    }

    // 设置当前结点是否是单词结尾
    static void setEnd(TrieNode* root, bool isEnd) {
        root->flag = isEnd;
    }
};

// 字典树方法
class Solution {
public:

    int R, C;           // 最大行列值
    TrieNode* trie;     // 字典树根结点
    vector<string> ans; // 答案数组
    int need;           // 还需要多少答案数量

    vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
        R = board.size();
        C = board[0].size();

        need = words.size();        // 设置答案需求数量

        trie = new TrieNode(true);  // 创建字典树根节点

        // 创建需要单词的字典树
        for (string& word : words) {
            trie->insert(trie, word);
        }

        // DFS 回溯搜索
        string temp;
        for (int row = 0; row < R; ++row) {
            for (int col = 0; col < C; ++col) {
```

```
                DFS(board, row, col, 0, trie, temp);
            }
        }

        return ans;
    }

    // 深度回溯字符矩阵
    // 试探回溯字符是否够成答案需要的单词
    // param: [int] depth - 递归/回溯/DFS 深度，因为单词长度最多 10，因此我们只需要递归深度到 10 即可返回
    // param: [TrieNode*] curNode - 当前字典树字符结点，初始化传入根结点
    // param: [string&] word - 回溯单词，临时存储当前组成的单词
    void DFS(vector<vector<char>>& board, int row, int col, int depth, TrieNode* curNode, string& word) {

        // DFS 深度 10
        if (depth >= 10) return;

        // 不需要再寻找了，因为答案已经找完了
        if (need == 0) return;

        // 坐标超出范围
        if (row < 0 || row >= R || col < 0 || col >= C) return;

        // 已被访问过(标记过)
        if (board[row][col] < 'a') return;

        // 在字典树中，要么该字符结点已被删除，或者没有/不存在
        if (!trie->has(curNode, board[row][col])) return;

        // 当前能够构成的单词
        word += board[row][col];

        // 取出下一个字符结点
        curNode = trie->getNextNode(curNode, board[row][col]);

        // 如果这个字符结点可以组成单词的话
        if (trie->isEnd(curNode)) {
            // 修改当前字符结点不能组成单词
            trie->setEnd(curNode, false);
            // 加入答案数组
            ans.push_back(word);
            // 减少需求数量
```

```
            --need;
            if (!need) return;

            // 如果该字符节点为叶结点
            if (trie->getNumsOfNext(curNode) == 0) {
                // 逻辑地向上回溯删除该分叉
                trie->del_up_logicly(curNode);
                // 回溯// 回溯单词
                word.pop_back();
                return;
            }
        }

        // 标记访问
        board[row][col] -= 26;

        DFS(board, row+1, col, depth+1, curNode, word);
        DFS(board, row-1, col, depth+1, curNode, word);
        DFS(board, row, col+1, depth+1, curNode, word);
        DFS(board, row, col-1, depth+1, curNode, word);

        // 回溯单词
        word.pop_back();
        // 回溯标记/访问
        board[row][col] += 26;
        return;
    }

};
```

作者：xiao-rong-jun
链接：
https://leetcode-cn.com/problems/word-search-ii/solution/czi-dian-shu-triequan-wen-zhu-shi-jian-z-w4fk/
来源：力扣（LeetCode）

我写的字典树

nullptr=0,NULL 就是 C 语言的

```
class TrieNode{
public:
    string word = "";
    vector<TrieNode*> nodes;//26 个 trie *next[26]={nullptr}
    TrieNode():nodes(26, 0){}//构造函数，初始化 26 个，空指针，初始化 node
```

```
};

class Solution {
    int rows, cols;
    vector<string> res;
public:
    vector<string> findWords(vector<vector<char>>& board, vector<string>& words)
{
        rows = board.size();
        cols = rows ? board[0].size():0;
        if(rows==0 || cols==0) return res;

        //建立字典树的模板
        TrieNode* root = new TrieNode();
        for(string word:words){
            TrieNode *cur = root;
            for(int i=0; i<word.size(); ++i){
                int idx = word[i]-'a';
                if(cur->nodes[idx]==0) cur->nodes[idx] = new TrieNode();
                cur = cur->nodes[idx];
            }
            cur->word = word;//结束标志，并且用当前的 word 表示
        }

        //DFS 模板
        for(int i=0; i<rows; ++i){
            for(int j=0; j<cols; ++j){
                dfs(board, root, i, j);
            }
        }
        return res;
    }

    void dfs(vector<vector<char>>& board, TrieNode* root, int x, int y){
        char c = board[x][y];
        //递归边界
        if(c=='.' || root->nodes[c-'a']==0) return;//没有这个起点，或者访问过
        root = root->nodes[c-'a'];//下一个
        if(root->word!=""){//结束标志
            res.push_back(root->word);
            root->word = "";//已经有了，不会再出现，相当于 visit 了这个单词
        }

        board[x][y] = '.';
```

```
        if(x>0) dfs(board, root, x-1, y);//四个方向
        if(y>0) dfs(board, root, x, y-1);
        if(x+1<rows) dfs(board, root, x+1, y);
        if(y+1<cols) dfs(board, root, x, y+1);
        board[x][y] = c;//回溯
    }
};
```

作者：talanto_linyi
链接：
https://leetcode-cn.com/problems/word-search-ii/solution/c-jian-dan-qing-xi-de-trieshu-ti-jie-by-talanto_li/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

124. 二叉树中的最大路径和（只不过加了权重呗，左右有个和 0 比较的，不一定要啊，用递归。每次得到当前数，最大的。选边的时候，选最大的左右边+root）

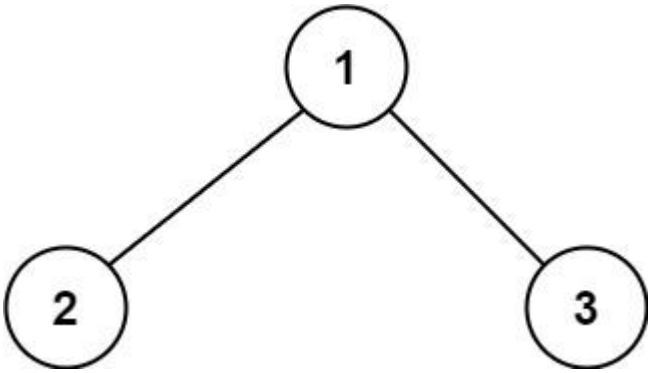
难度困难 1000

路径 被定义为一条从树中任意节点出发，沿父节点-子节点连接，达到任意节点的序列。同一个节点在一条路径序列中 **至多出现一次** 。该路径 **至少包含一个** 节点，且不一定经过根节点。

路径和 是路径中各节点值的总和。

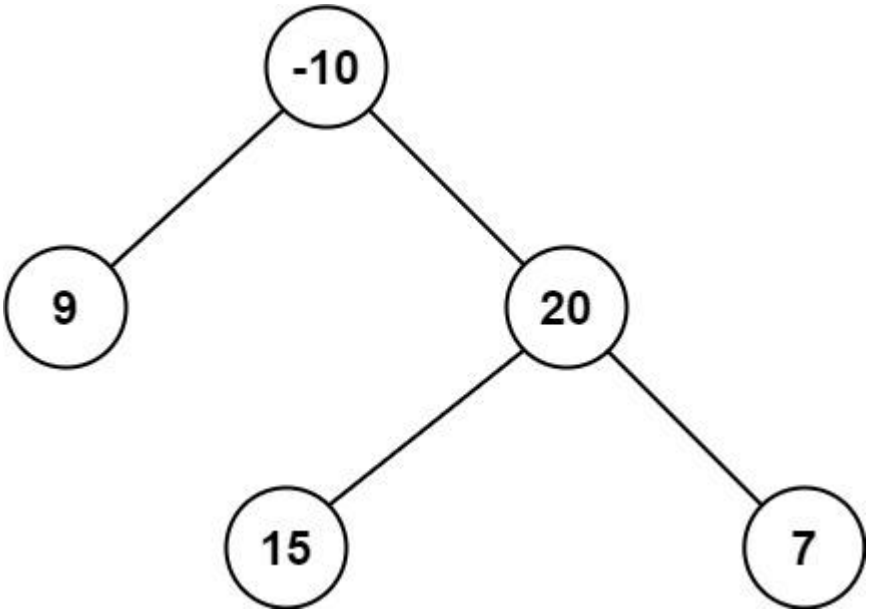
给你一个二叉树的根节点 *root* ，返回其 **最大路径和** 。

示例 1：



输入：root = [1,2,3]**输出：**6 **解释：**最优路径是 2 -> 1 -> 3 ，路径和为 2 + 1 + 3 = 6

示例 2：



输入：root = [-10,9,20,null,null,15,7]**输出：**42 **解释：**最优路径是 15 -> 20 -> 7 ，路径和为 15 + 20 + 7 = 42

提示：

- 树中节点数目范围是 *[1, 3 * 104]*
- *-1000 <= Node.val <= 1000*

通过次数 112,831 提交次数 259,056

在真实的面试中遇到过这道题？
是
否

```
class Solution {
private:
    int maxSum = INT_MIN;

public:
    int maxGain(TreeNode* node) {
        if (node == nullptr) {
            return 0;
        }

        // 递归计算左右子节点的最大贡献值
        // 只有在最大贡献值大于 0 时，才会选取对应子节点
        int leftGain = max(maxGain(node->left), 0);
        int rightGain = max(maxGain(node->right), 0);

        // 节点的最大路径和取决于该节点的值与该节点的左右子节点的最大贡献值
        int priceNewpath = node->val + leftGain + rightGain;

        // 更新答案
        maxSum = max(maxSum, priceNewpath);

        // 返回节点的最大贡献值
        return node->val + max(leftGain, rightGain);
    }

    int maxPathSum(TreeNode* root) {
        maxGain(root);
        return maxSum;
    }
};
```

作者：LeetCode-Solution
链接：
https://leetcode-cn.com/problems/binary-tree-maximum-path-sum/solution/er-cha-shu-zhong-de-zui-da-lu-jing-he-by-leetcode-/
来源：力扣（LeetCode）

238. 除自身以外数组的乘积（先记录除去自身外，左边乘积，在计算右边的乘积，后面的是倒序）

难度中等 789

给你一个长度为 n 的整数数组 *nums*，其中 n > 1，返回输出数组 *output* ，其中 *output[i]* 等于 *nums* 中除 *nums[i]* 之外其余各元素的乘积。

示例:

输入: [1,2,3,4]输出: [24,12,8,6]

提示：题目数据保证数组之中任意元素的全部前缀元素和后缀（甚至是整个数组）的乘积都在 32 位整数范围内。

说明: 请**不要使用除法，**且在 O(n) 时间复杂度内完成此题。

进阶：
你可以在常数空间复杂度内完成这个题目吗？（ 出于对空间复杂度分析的目的，输出数组**不被视为**额外空间。）

```
class Solution {public:
    vector<int> productExceptSelf(vector<int>& nums) {
        int n = nums.size();
        vector<int> res(n, 0);
        int k = 1;
        for (int i = 0; i < n; i ++) {
            res[i] = k;
            k *= nums[i];//始终是当前 i 的最左边
        }
        k = 1;
        for (int i = n - 1; i >= 0; i --) {
```

```
            res[i] *= k;
            k *= nums[i];//倒序，始终是 i 的最右边
        }
        return res;
    }
};
```

乘积 = 当前数左边的乘积 * 当前数右边的乘积

328. 奇偶链表（将奇数和偶数分别排列，最后将奇数与后面接一个偶数）

难度中等 409

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是**节点编号的奇偶性**，而不是节点的值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nodes 为节点总数。

示例 1:

输入: 1->2->3->4->5->NULL **输出:** 1->3->5->2->4->NULL

示例 2:

输入: 2->1->3->5->6->4->7->NULL **输出:** 2->3->6->7->1->5->4->NULL

说明:

- 应当保持奇数节点和偶数节点的相对顺序。
- 链表的第一个节点视为奇数节点，第二个节点视为偶数节点，以此

class Solution {

public:

ListNode* oddEvenList(ListNode* head) {

```
            if (head == nullptr) {
                return head;
            }
            ListNode* evenHead = head->next;//记录偶数头
            ListNode* odd = head;//记录奇数头，返回这个 head
            ListNode* even = evenHead;//偶数
            while (even != nullptr && even->next != nullptr) {//偶数末尾，先得到奇数
                odd->next = even->next;奇数下一个是偶数下一个
                odd = odd->next;更新奇数
                even->next = odd->next;偶数下一个是奇数下一个
                even = even->next;更新偶数
            }
            odd->next = evenHead;//奇数连接偶数头
            return head;返回头
        }
    };
    //交叉合并
 void mergeList(ListNode* l1, ListNode* l2) {//交叉合并
        ListNode* l1_tmp;
        ListNode* l2_tmp;
        while (l1 != nullptr && l2 != nullptr) {//结束一轮返回原来的状态
            l1_tmp = l1->next;//每次前链表拿出下一个
            l2_tmp = l2->next;//每次拿出后链表下一个

            l1->next = l2;//前链表的下一个是后链表
            l1 = l1_tmp;//更新前链表

            l2->next = l1;//后的下一个是前链表
            l2 = l2_tmp;//更新后链表
        }
    }
};
```

作者：LeetCode-Solution

链接：https://leetcode-cn.com/problems/odd-even-linked-list/solution/qi-ou-lian-biao-by-leetcode-solution/

来源：力扣（LeetCode）

673. 最长递增子序列的个数(分别记录dp[i]count[i]，某个元素结尾的最长递增序列长度，还有出现的个数，记录最长的 max,如果在 0-i-1 存在 j nums[i]<num[j]，存在 dp[j]+1>dp[i],那么后面的最长的个数，j+1 长度更长，覆盖 cout[i],count[i]=count[j],如果 dp[j]+1=dp[i]等于的话，就是 count[i]+=count[j]，j 只要+1 等于 i 的长度，长度相等)

难度中等 292

给定一个未排序的整数数组，找到最长递增子序列的个数。

示例 1:

输入: [1,3,5,4,7]**输出:** 2 **解释:** 有两个最长递增子序列，分别是 [1, 3, 4, 7] 和[1, 3, 5, 7]。

示例 2:

输入：[2,2,2,2,2]**输出：**5 **解释：**最长递增子序列的长度是 1，并且存在 5 个子序列的长度为 1，因此输出 5。

```
class Solution {
public:
    int findNumberOfLIS(vector<int>& nums) {
        if (nums.size() <= 1) return nums.size();
        vector<int> dp(nums.size(), 1);//数组，最长
        vector<int> count(nums.size(), 1);//结尾的个数
        int maxCount = 0;
        for (int i = 1; i < nums.size(); i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    if (dp[j] + 1 > dp[i]) {
                        dp[i] = dp[j] + 1;
                        count[i] = count[j];
                    } else if (dp[j] + 1 == dp[i]) {
                        count[i] += count[j];
                    }
                }
                if (dp[i] > maxCount) maxCount = dp[i];//更新长度
            }
        }
        int result = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (maxCount == dp[i]) result += count[i];
        }
        return result;
    }
};
```

作者：carlsun-2
链接：
https://leetcode-cn.com/problems/number-of-longest-increasing-subsequence/solution/673-zui-chang-di-zeng-zi-xu-lie-de-ge-sh-606o/
来源：力扣（LeetCode）

这道题可以说是 300.最长上升子序列 的进阶版本

确定 dp 数组（dp table）以及下标的含义
这道题目我们要一起维护两个数组。

dp[i]：i 结尾最长递增子序列的长度为 dp[i]

count[i]：以 nums[i]为结尾的字符串，最长递增子序列的个数为 count[i]

确定递推公式
在 300.最长上升子序列 中，我们给出的状态转移是：

```
if (nums[i] > nums[j]) dp[i] = max(dp[i], dp[j] + 1);
```

即：位置 i 的最长递增子序列长度 等于 j 从 0 到 i-1 各个位置的最长升序子序列 + 1 的最大值。

本题就没那么简单了，我们要考虑两个维度，一个是 dp[i]的更新，一个是 count[i]的更新。

那么如何更新 count[i]呢？

以 nums[i]为结尾的字符串，最长递增子序列的个数为 count[i]。

那么在 nums[i] > nums[j]前提下，如果在[0, i-1]的范围内，找到了 j，使得 dp[j] + 1 > dp[i]，说明找到了一个更长的递增子序列。（寻找的个数，长度，至少大于 dp[j]+1 大于或等于 dp[i]的更新）**子集，比 i，小于尾部，就是小于 1,相差 1,加起来，大于 1，更新**

那么以 j 为结尾的子串的最长递增子序列的个数，就是**最新的以 i 为结尾的子串的最长递增子序列的**个数，即：count[i] = count[j]。//对的，前面 i 有几个最长，加上自身，就有几个 j

在 nums[i] > nums[j]前提下，如果在[0, i-1]的范围内，找到了 j，使得 dp[j] + 1 == dp[i]，说明找到了两个**相同长度的递增子序列**。

那么以 i 为结尾的子串的最长递增子序列的个数 就应该加上以 j 为结尾的子串的最长递增子序列的个数，即：**count[i] += count[j];//j+1（自身），与自己的 i 长度相等，相加**

作者：carlsun-2
链接：
https://leetcode-cn.com/problems/number-of-longest-increasing-subsequence/solution/673-zui-chang-di-zeng-zi-xu-lie-de-ge-sh-606o/
来源：力扣（LeetCode）

34. 在排序数组中查找元素的第一个和最后一个位置（左右边界，有个不存在，返回-1 操作）

难度中等 950

给定一个按照升序排列的整数数组 *nums*，和一个目标值 *target*。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]*。

进阶：

- 你可以设计并实现时间复杂度为 *O(log n)* 的算法解决此问题吗？

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：

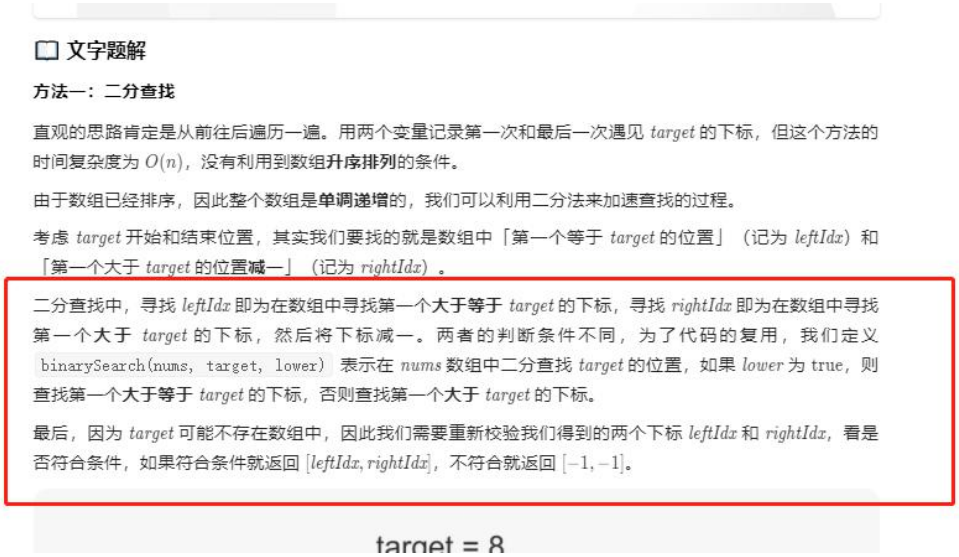
输入：`nums = [], target = 0` **输出：**`[-1,-1]`

提示：

- *0 <= nums.length <= 105*
- *-109 <= nums[i] <= 109*
- *nums* 是一个非递减数组
- *-109 <= target <= 109*

```
class Solution {
public:
    int binarySearch(vector<int>& nums, int target, bool lower) {
        int left = 0, right = (int)nums.size() - 1, ans = (int)nums.size();
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] > target || (lower && nums[mid] >= target)) {
                right = mid - 1;//左边界，右边收缩
                ans = mid;
            } else {
                left = mid + 1;
            }
        }
        return ans;
    }

    vector<int> searchRange(vector<int>& nums, int target) {
        int leftIdx = binarySearch(nums, target, true);
        int rightIdx = binarySearch(nums, target, false) - 1;
        if (leftIdx <= rightIdx && rightIdx < nums.size() && nums[leftIdx] == target
&& nums[rightIdx] == target) {
            return vector<int>{leftIdx, rightIdx};
        }
        return vector<int>{-1, -1};
    }
};
```



文字题解

方法一：二分查找

直观的思路肯定是从前往后遍历一遍。用两个变量记录第一次和最后一次遇见 $target$ 的下标，但这个方法的时间复杂度为 $O(n)$，没有利用到数组**升序排列**的条件。

由于数组已经排序，因此整个数组是**单调递增**的，我们可以利用二分法来加速查找的过程。

考虑 $target$ 开始和结束位置，其实我们要找的就是数组中「第一个等于 $target$ 的位置」（记为 $leftIdx$）和「第一个大于 $target$ 的位置减一」（记为 $rightIdx$）。

二分查找中，寻找 $leftIdx$ 即为在数组中寻找第一个**大于等于** $target$ 的下标，寻找 $rightIdx$ 即为在数组中寻找第一个**大于** $target$ 的下标，然后将下标减一。两者的判断条件不同，为了代码的复用，我们定义 `binarySearch(nums, target, lower)` 表示在 $nums$ 数组中二分查找 $target$ 的位置，如果 $lower$ 为 true，则查找第一个**大于等于** $target$ 的下标，否则查找第一个**大于** $target$ 的下标。

最后，因为 $target$ 可能不存在数组中，因此我们需要重新校验我们得到的两个下标 $leftIdx$ 和 $rightIdx$，看是否符合条件，如果符合条件就返回 $[leftIdx, rightIdx]$，不符合就返回 $[-1, -1]$。

target = 8

作者：LeetCode-Solution

链接：

https://leetcode-cn.com/problems/find-first-and-last-position-of-element-in-sorted-array/solution/zai-pai-xu-shu-zu-zhong-cha-zhao-yuan-su-de-di-3-4/

149. 直线上最多的点数（用最小公约数，来保证分子分母都是整数，看看每个点上有多少共线。bool flag = true; // 符号，标志位，判断正负，和前导 0 一样，for (auto& [_, i] : mp) {// 这样使用，两重循环，从 0-I, i+1，因为要是 i 与之前共线，那么 i 的共线点肯定之前出现，并且更小）

难度困难 234

给定一个二维平面，平面上有 n 个点，求最多有多少个点在同一条直线上。

方法二：用行列式求三角形面积，再判断是否为 0.

$$S_{ABC}=\frac{1}{2}\begin{Vmatrix} x1 & y1 & 1 \\ x2 & y2 & 1 \\ x3 & y3 & 1 \end{Vmatrix}$$

S=(1/2)*(x1*y2+x2*y3+x3*y1-x1*y3-x2*y1-x3*y2)

示例 1:

输入： [[1,1],[2,2],[3,3]]**输出：** 3 **解释：**

^

|

```
|        o
|     o
|  o
+------------->
0  1  2  3  4
```

示例 2:

输入: `[[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]`**输出:** `4` **解释:**

```
^
|
|  o
|     o        o
|        o
|  o       o
+------------------->
0  1  2  3  4  5  6
```

这题首先直接用 unordered_map<doule, int>是可以过的，但是存在精度问题

可以用 pair<int, int>保存分子分母约分后的结果，由于 pair 没有 hash 函数，只能退而求其次用 map 了

作者：moreality
链接：
https://leetcode-cn.com/problems/max-points-on-a-line/solution/149-zhi-xian-shang-zui-duo-de-dian-shu-d-oyjj/

```
class Solution {
public:
    // 求最大公约数
    int gcd(int a, int b) {
        int t;
        if (a < b) swap(a, b);
        while(b > 0) {
            t = a % b;//只要 a 是 b 的倍数，就返回来 b,取余等于 0。
            a = b;
            b = t;
        }
        return a;
    }
    // 斜率  k = (x1 - x2) / (y1 - y2)
    pair<int, int> k (int x1, int x2, int y1, int y2) {
        bool flag = true; // 符号
        int up, down; // 分子，分母
        if (y1 - y2 == 0) return {INT_MAX, INT_MAX};
        else {
            int dx = x1 - x2;
            int dy = y1 - y2;
            // 符号相反
            if ((dx ^ dy) < 0) flag = false;
            dx = abs(dx), dy = abs(dy);
            up = dx / gcd(dx, dy);//保证了这里是最小的整数，不会是浮点数
            down = dy / gcd(dx, dy);
        }
        return flag ? make_pair(up, down) : make_pair(-up, down);
    }

    int maxPoints(vector<vector<int>>& points) {
        map<pair<int, int>, int> mp; // (斜率，数目)
        int ans = 1;
        for (int i = 0; i < points.size(); ++i) {
            for (int j = i + 1; j < points.size(); ++j) {//双重循环//从下一个点开始
//为什么不找前面呢，因为前面如果共线，肯定是前面更大。
                ++mp[k(points[i][0], points[j][0], points[i][1], points[j][1])];
            }
            for (auto& [_, i] : mp) {//这样使用
                ans = max(ans, i + 1);  //加上自身的点
            }
            mp.clear();//每次清空，返回最大值
```

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

作者：moreality
链接：
https://leetcode-cn.com/problems/max-points-on-a-line/solution/149-zhi-xian-shang-zui-duo-de-dian-shu-d-oyjj/
来源：力扣（LeetCode）