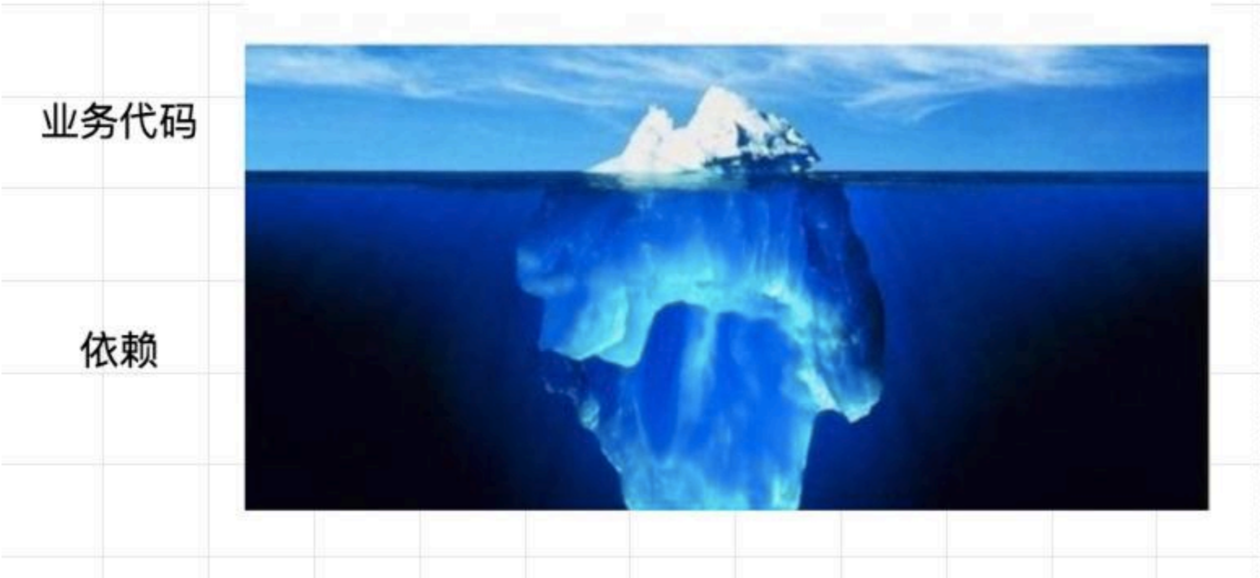
前端工程师为什么要学习算法?

引言

> 随着技术生态的不断发展，遇到一般的业务问题直接把人家的方案拿来用就可以了。
>
> 当前各种技术框架的真实写照：**足够好用，足够简单，简单到你不需要知道如何去实现的。**

上图~



问题来了，作为一个集智慧和才华于一身的程序员，自己的价值在哪里?

未来的前端工程师，应该..... 比如说：

- 开发一个辅助业务的**脚手架工具**
- 面对复杂业务时，能不能尽可能从代码的角度**提高其性能** （最优解）
- Git 推送远程仓库时所用到的最小编辑距离算法
- webpack的Tree-shaking 如何实现，加速打包如何实现
- 如何深层次的去了解和学习一个源码的设计思想
    - vue中keep-alive组件的LRU(缓存淘汰算法)（最近最少使用）
    - vue和react在patch对比children的时候分别用到了什么算法，二者各有什么**优劣势**?
    - 3.0对于2.0的提升，diff算法的改变?
    - js的内存管理机制
    - react Hooks 是如何解决纯函数无法持久化状态的问题?
- 等等 ~ 很多

总结

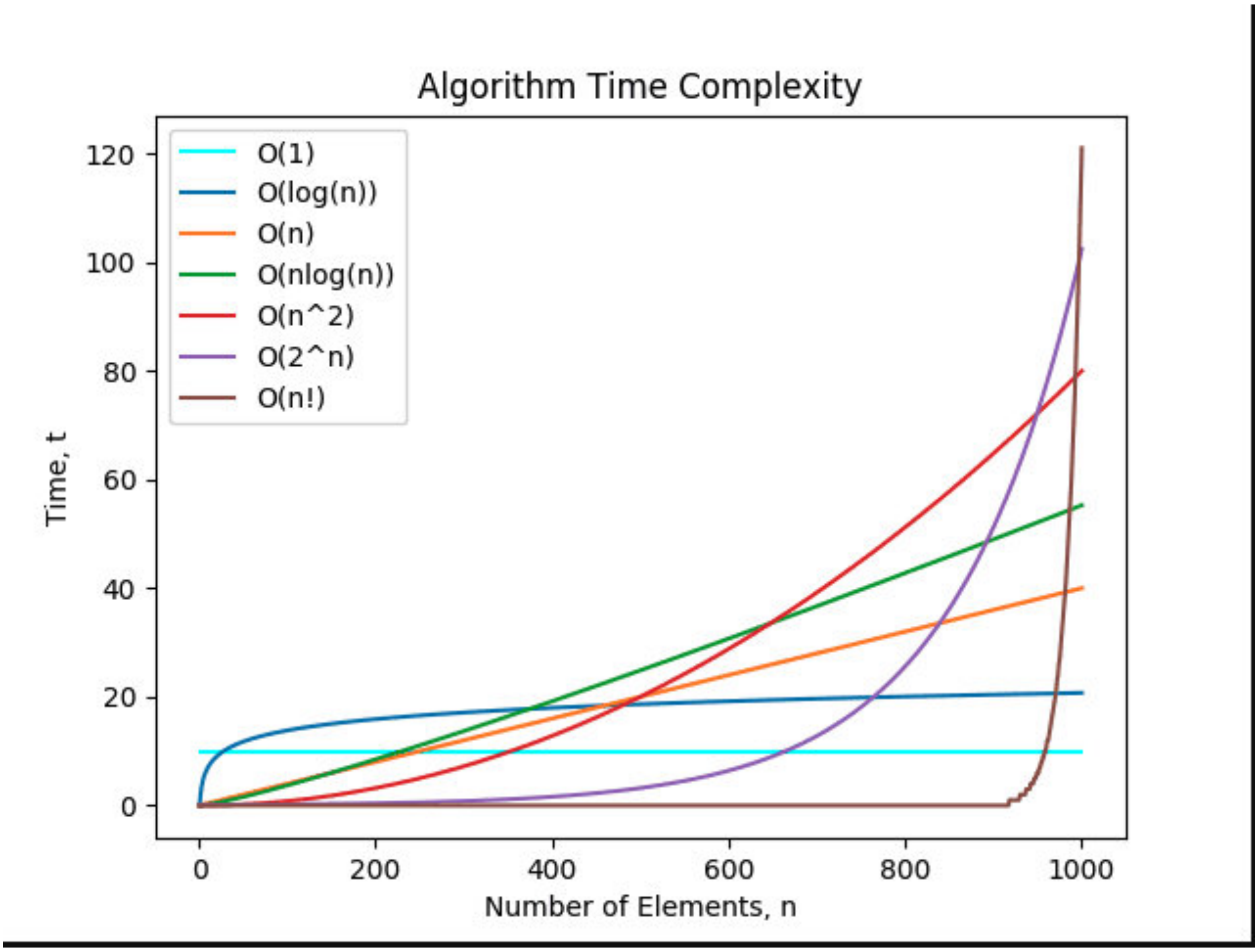
- 如果不懂算法，我们能在前端走多久?
- 凡是需要跨过一定智商门槛才能掌握的技术，都不会轻易的流行。

两大概念：时间复杂度和空间复杂度

常见的时间复杂度

- `O(1)`: Constant Complexity: Constant 常数复杂度
- `O(log n)`: Logarithmic Complexity: 对数复杂度
- `O(n)`: Linear Complexity: 线性时间复杂度
- `O(n^2)`: N square Complexity 平方方
- `O(n^3)`: N square Complexity 立立方方
- `O(2^n)`: Exponential Growth 指数
- `O(n!)`: Factorial 阶乘

上图



空间复杂度：是对一个算法在运行过程中临时占用存储空间大小的一个量度。

- 一个程序执行时需要的存储空间和存储本身所使用的指令、常数、变量和输入数据
- 对数据进行操作的工作单元
- 一些为实现计算所需信息的辅助空间。

排序算法

> 将一串数据依照特定排序方式进行排列，产生需要的输出结果。经处理后的数据便于筛选和计算，大大提高了计算效率

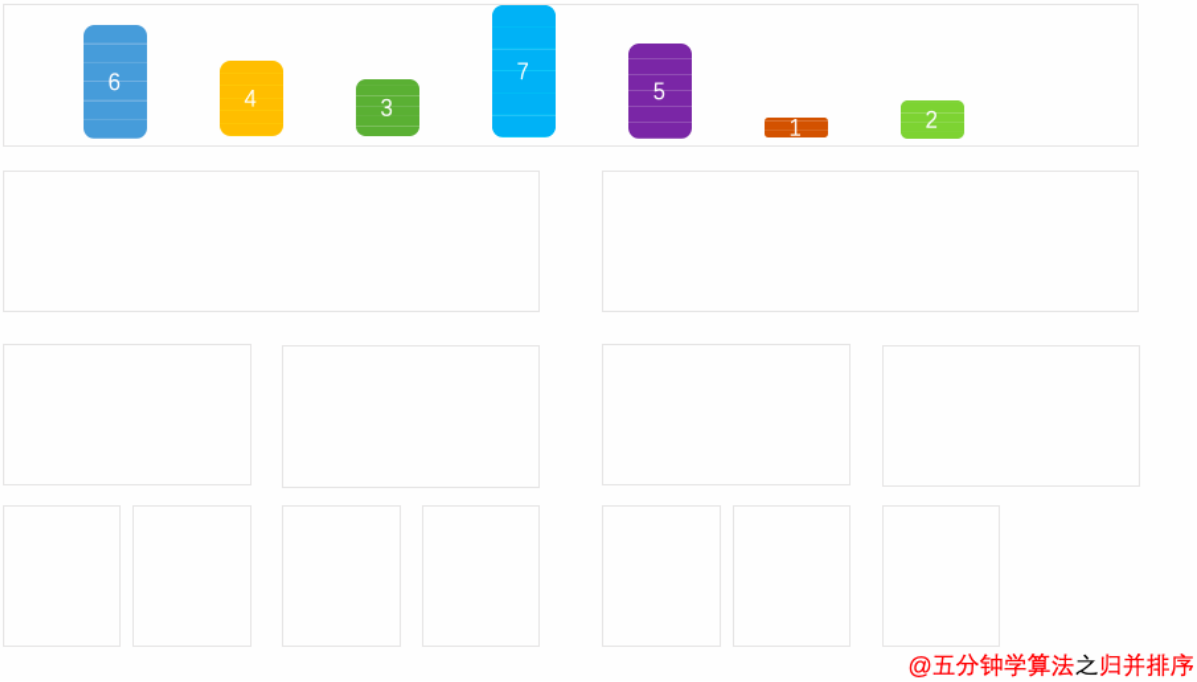
算法基础 常见的排序算法很多种 简单了列举了几种比较常见的

- 冒泡排序

- 快速排序
- 选择排序
- 插入排序
- 归并排序

归并排序

排序算法动画



示例代码:

```
    function mergeSort(arr) {
        if (arr.length < 2) {
            return arr;
        }
        let middle = Math.floor(arr.length / 2);
        let left = arr.slice(0, middle);
        let right = arr.slice(middle);
        let result = merge(mergeSort(left), mergeSort(right));
        return result;
    }

    function merge(left, right) {
        let result = [];
        let i = 0;
        let k = 0;
        while (i < left.length && k < right.length) {
            if (left[i] < right[k]) {
                result.push(left[i++]);
            } else {
                result.push(right[k++]);
            }
```

```
        }
        return result.concat(left.slice(i)).concat(right.slice(k));
    }
```

推荐资料：

[前端基本排序算法]

[前端排序算法总结]

排序算法时间复杂度统计：

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

扩展问题：

Array.prototype.sort() 用的是哪一种排序？

mdn对于sort的解释：

- 用[原地算法]对数组的元素进行排序，并返回数组。默认排序顺序是在将元素转换为字符串，然后比较它们的UTF-16代码单元值序列时构建的
- 取决于具体实现，因此无法保证排序的时间和空间复杂性

[JavaScript专题之解读 v8 排序源码]

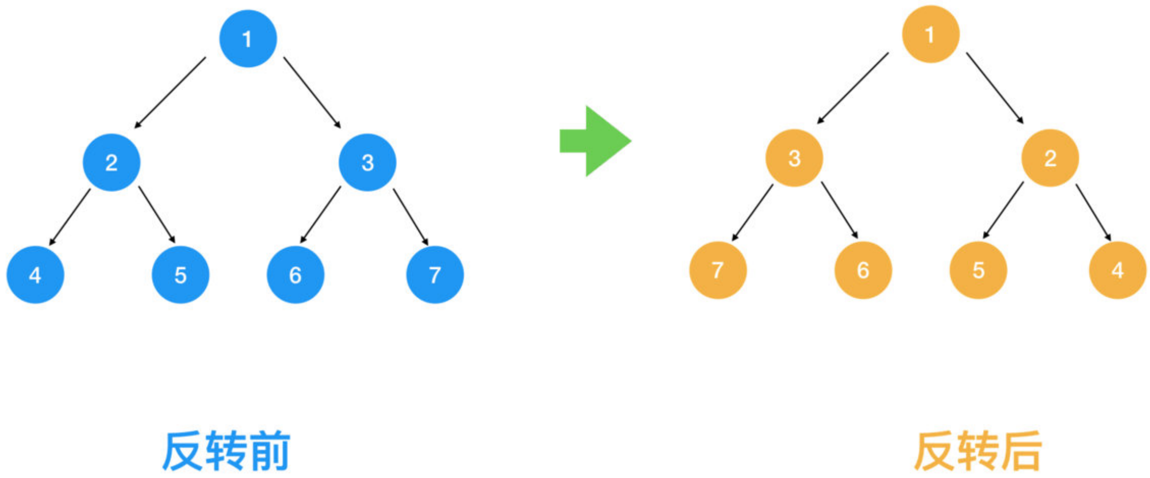
二叉树

> 是一种非线性的数据结构，主要应用于高效率的搜索和排序

讲个故事

反转二叉树

> 所以只要答对这道题，你就可以超越世界级大牛，问鼎码林之巅~



示例代码：

```
function mirrot (root) {
    if(!root){
        return null
    }
    let temp =  root.left
    root.left = root.right
    root.right = temp
    if(root.left){
        root.left = mirrot(root.left)
    }
```

```
        if(root.right){
            root.right = mirrot(root.right)
        }
        return root
    }
```

扩展

- 二叉树的先中后序遍历
- 二叉树的层序遍历（广度优先）

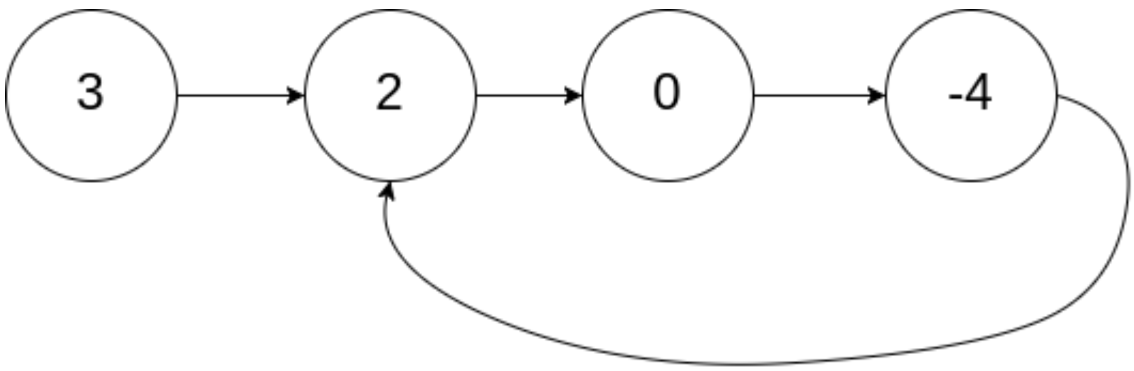
链表

> 链表是一种非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的，链表的插入操作可以达到O(1)d的时间复杂度，应用场景： 各种文本，图形的编辑器中的撤销重做，实现图、hashMap等高级数据结构

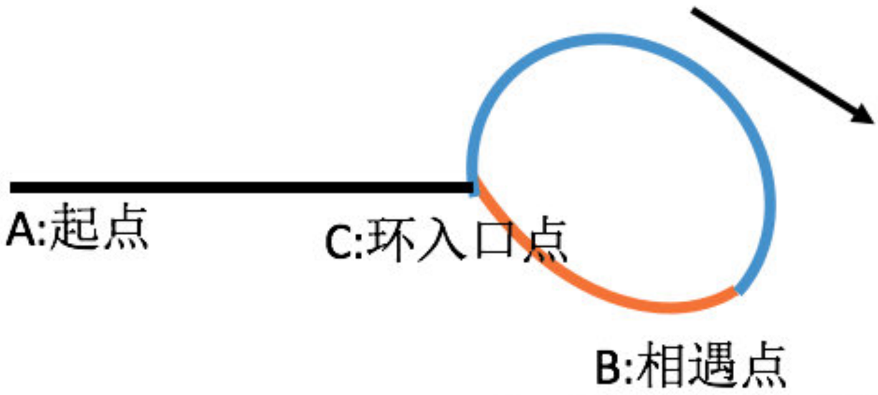
判断一个链表是否有环

链接：https://leetcode-cn.com/problems/linked-list-cycle

```
输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。
示例  2：
输入：head = [1,2], pos = 0
输出：true
解释：链表中有一个环，其尾部连接到第一个节点。
示例 3：
输入：head = [1], pos = -1
输出：false
解释：链表中没有环。
```



算法思想：



假设 x 为环前面的路程（黑色路程），A-C

a 为环入口到相遇点的路程（蓝色路程，假设顺时针走），

c 为环的长度（蓝色+橙色路程）

公式推导：

```
slow = x+m*c+a  走了m圈
fast = x+n*c+a  走了n圈

2slow = fast  假设两个人的速度不一致  快的人 速度是慢的人的二倍

2x+2(m*c)+2a = x+n*c+a

x = (n-2m)*c-a = (n-2m-1)*c+c-a   碰撞点b到连接点的距离=头节点到连接点的距离
```

代码：

```
var hasCycle = function(head) {

  let fastP = head
  let slowP = head
  while (fastP) { // 当前快指针指向了节点
    if (!fastP.next) return false // 下一个为null了，没有环
    slowP = slowP.next // 快的前面都有节点，慢的前面当然有
    fastP = fastP.next.next // 推进2个节点
    if (slowP === fastP) return true // 快慢指针相遇，有环
  }
  // todo
  return false // fastP为null了也始终不相遇
};
```

> 如果这道题 改一下：改为求环的入口结点？应该怎么做？

```
        // 此处可以判断出链表是否存在环
        let p = head;
        // 返回的节点
        // 定理：碰撞点p到连接点的距离=头节点到连接点的距离
        while (p !== slow) {
            p = p.next;
            slow = slow.next;
        }
        return p;
```

- 反转链表
- 链表是什么

哈希表

> 是根据键（Key）而直接访问在内存储数据的数据结构，它通过计算一个关于键值的函数，将所需查询的数据映射到表中，加快了查找速度。这个映射函数称做散列函数，存放记录的数组称做**散列表**。可以快速的定位到要查找的数据，查询的时间复杂度接近O(1) ,应用场景举例： 手机联系人

两数之和

```
给定一个整数数组 nums  和一个目标值 target，请你在该数组中找出和为目标值的那两个整数，并返回他们的数组下标。
你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

示例：
给定 nums = [2, 7, 11, 15], target = 9
因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
```

```
var twoSum = function (nums, target) {
        let map = new Map();
        for (let i = 0; i < nums.length; i++) {
            map.set(num[i], i);
        }
        for (let j = 0; j < nums.length; j++) {
            if (map.has(target - nums[j]) && map.get(target - nums[i]) !== j) {
                return [j, map.get(target - nums[i])];
            }
        }
  };
```

扩展：

- 三数之和
- 四数之和

  - 暴力法 四层遍历 求值 注意去重 问题
  - 哈希表替代暴力解法 n4==》n3
- 无重复字符的最长子串

滑动窗口

> 用以解决数组/字符串的子元素问题，它可以将嵌套的循环问题，转换为单循环问题，利用决策单调性来降低时间复杂度。tcp的滑动窗口协议等

滑动窗口的最大值

> leetcode 地址： https://leetcode-cn.com/problems/sliding-window-maximum

```
给定一个数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。
返回滑动窗口中的最大值。

进阶：
你能在线性时间复杂度内解决此题吗？

示例：
输入：nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
输出：[3,3,5,5,6,7]
解释：

  滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7

提示：

1 <= nums.length <= 10^5
-10^4 <= nums[i] <= 10^4
1 <= k <= nums.length
```

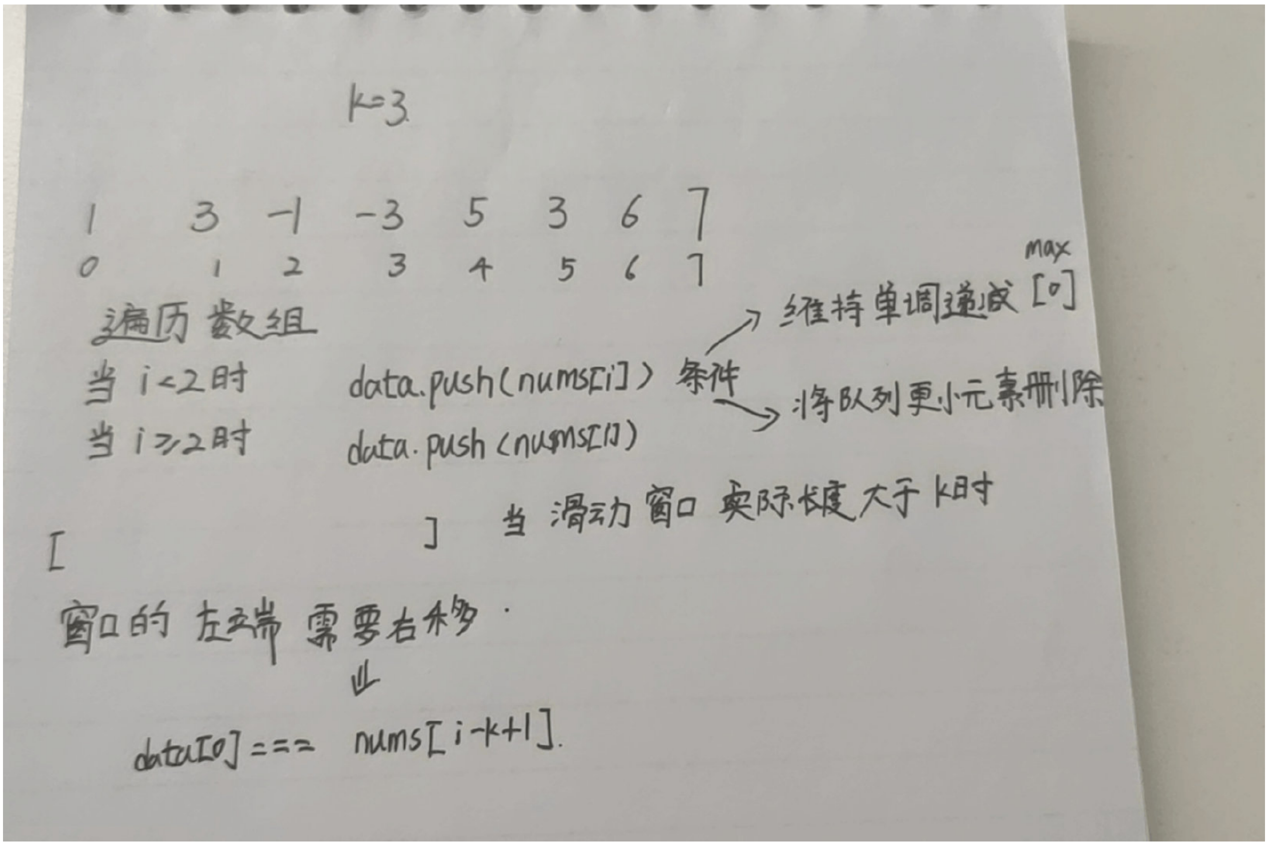
解题方法：

解法1

```
var maxSlidingWindow = function (nums, k) {
      let result = [];
      let arr = nums.slice();
      for (let i = 0; i < nums.length - k + 1; i++) {
          let temp = arr.slice(i, i + k);
          result.push(Math.max.apply(this, temp));
      }
      return result;
  };
```

解法2 滑动窗口

解题思想：




```
    var maxSlidingWindow = function (nums, k) {
      let data = [];
      let result = [];
      for (let i = 0; i < nums.length; i++) {
          if (i < k - 1) {
              while (data.length > 0 && data[data.length - 1] < nums[i]) {
                  data.pop();
              }
              data.push(nums[i]);
          } else {
              while (data.length > 0 && data[data.length - 1] < nums[i]) {
                  data.pop();
              }
```

```
                data.push(nums[i]);
                result.push(data[0]);
                if (data.length > 0 && data[0] === nums[i - k + 1]) {
                    data.shift();
                }
            }
        }
        return result;
    };
```

- 长度最小的子数组

赛马问题

已知有25匹马，5个跑道，每个跑道只能容一匹马，没有计时器，至少需要比赛多少次，可以找出最快的前三匹马

题解:

赛马问题

A1	B1	C1	D1	E1
A2	B2	C2	D2	E2
A3	B3	C3	D3	E3
A4	B4	C4	D4	E4
A5	B5	C5	D5	E5

已知有30匹马，5个跑道，每个跑道只能容一匹马，没有计时器，至少需要比赛多少次，可以找出最快的前三匹马?

推导公式

> 之前一篇文章读到的 但是具体的推导过程 忘记了~~ 大家可以验证下这个公式

```
一共有 n\*n 匹马，赛场有 n 个赛道，试问最少得比多少场才能找出跑得最快的 k 匹马。

w(n,k)相当于将质量为 2,3,4, ..., k 的球放入最大承重为 n 的桶中，最少需要的桶的个数

如果 k = 1, f(n,k) = n + 1

如果k>1 && k <= n, f(n,k) = n + 1 + w(n,k)

如果k>n, f(n,k) = n + 1 + w(n,k) + k - n
```

剪枝

剪枝顾名思义，就是删去一些不重要的节点，来减小计算或搜索的复杂度。

- 在决策树和神经网络中，剪枝可以有效缓解过拟合问题并减小计算复杂度；

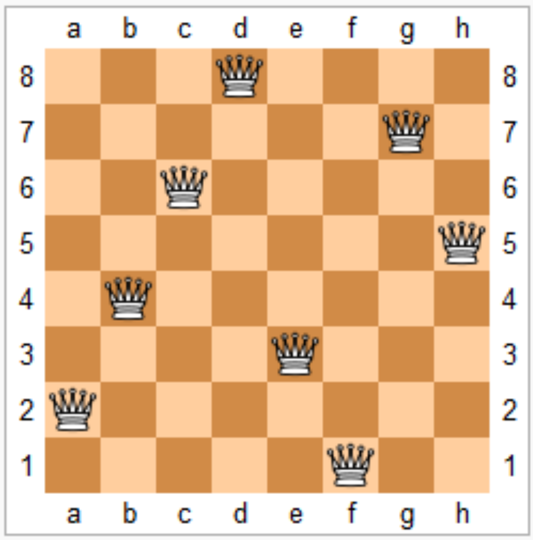
- 在搜索算法中，可以减小搜索范围，提高搜索效率。

经典案例 Deep Blue Vs Garry kasparov(卡斯帕罗夫)

经典算法：n皇后问题（剪枝+回溯）

> 由4皇后和8皇后 演变而来，是回溯算法的经典案例，也是一个比较悠久的数学问题

n皇后I(n>=4)



One solution to the eight queens puzzle

```
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。
上图为 8 皇后问题的一种解法。
给定一个整数 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明确的 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：
输入：4
```

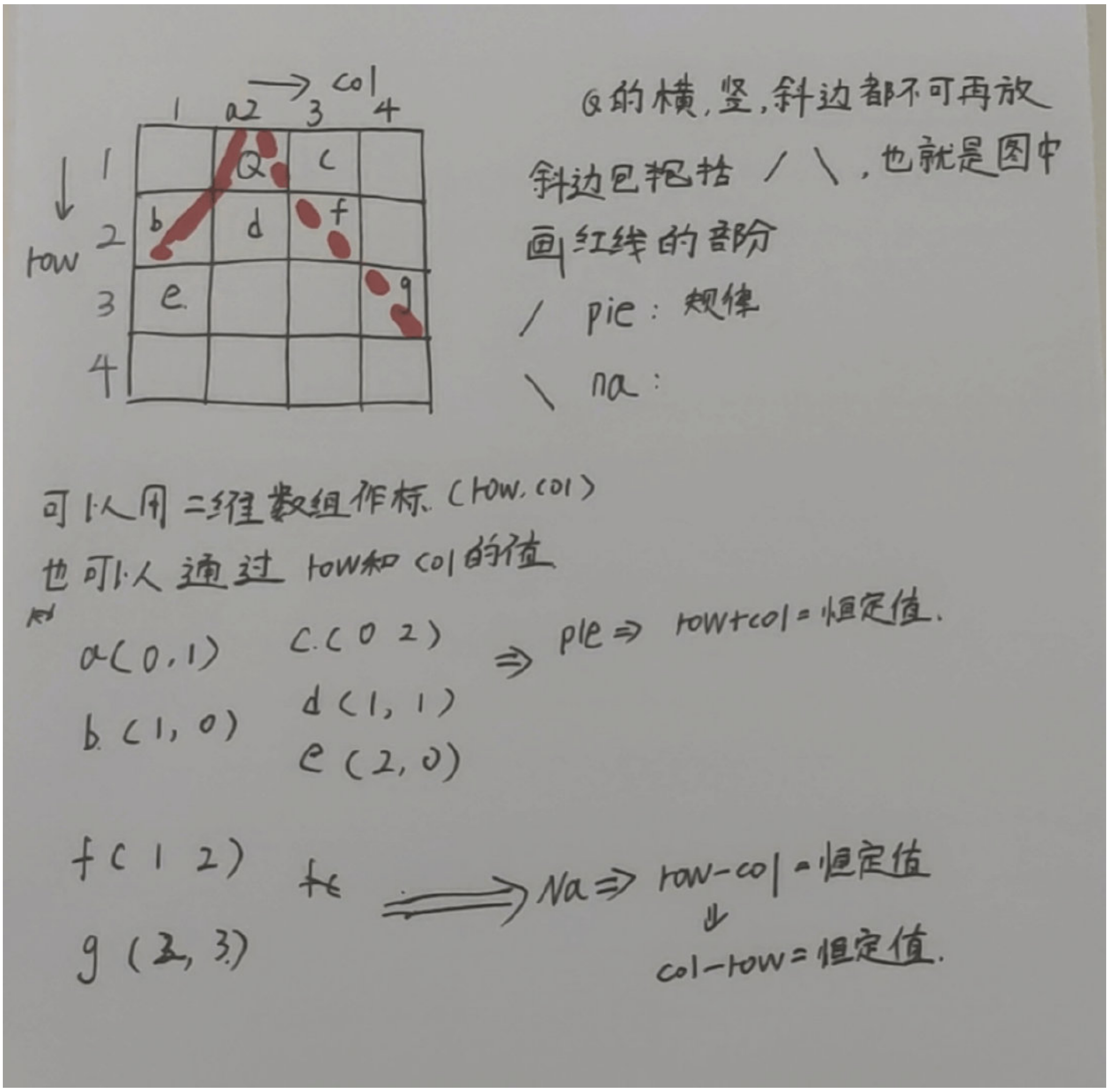
```
输出： [
 [".Q..",  // 解法 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // 解法 2
  "Q...",
  "...Q",
  ".Q.."]
]
解释： 4 皇后问题存在两个不同的解法。
```

提示：

皇后，是国际象棋中的棋子，意味着国王的妻子。皇后只做一件事，那就是“吃子”。当她遇见可以吃的棋子时，就迅速冲上去吃掉棋子。当然，她横、竖、斜都可走一到七步，可进可退。（引用自 百度百科 - 皇后 ）

解题思路：



示例代码：

```
/**
    * @param {number} n
    * @return {string[][]}
    */
   var solveNQueens = function (n) {
       const cols = new Set();
       const pies = new Set(); // 左对角线的攻击位置
       const nas = new Set(); // 右对角线的攻击位置
       let row = 0;
       let queens = [];
       let res = [];
       recurison(n, row, queens);
       return generateCheckerboard(res, n);
       function recurison(n, row, queens) {
           for (let col = 0; col < n; col++) {
               if (cols.has(col) || pies.has(row + col) || nas.has(row - col))
{
                   continue;
               }
               cols.add(col);
               pies.add(row + col);
               nas.add(row - col);
               queens.push(col);
               // 继续放置下一行
               recurison(n, row + 1, queens);
               queens.pop();
               pies.delete(row + col);
               nas.delete(row - col);
               cols.delete(col);
           }

           if (row >= n) {
               res.push(queens.slice());
               return;
           }
       }
   };

   function generateCheckerboard(res, n) {
       return res.map((queens) => {
           return queens.map((q) => {
               return Array(n)
                   .fill()
                   .map((k, i) => {
                       return i === q ? 'Q' : '.';
                   })
                   .join('');
```

```
            });
        });
    }
```

n皇后解法2

> 使用位运算进行求解，效率最高。但是代码比较绕~而且不容易想 大家理解下这种解法即可~

位运算的运算规则

符号	描述	运算规则
&	与	两个位都为1时，结果才为1
|	或	两个位都为0时，结果才为0
^	异或	两个位相同为0，相异为1
~	取反	0变1，1变0
<<	左移	各二进位全部左移若干位，高位丢弃，低位补0
>>	右移	各二进位全部右移若干位，对无符号数，高位补0，有符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补0（逻辑右移）

常用的位运算操作

- X & 1 == 1 OR == 0
- X = X & (X-1) => 清零最低位的1
- X & -X => 得到最低位的1
- x & ~X => 0
- 将X最右边的n位清零 x & (~0 << n)
- 获取x的第n位值(0或者1) (x >> n) & 1
- 获取x的第n位的幂值 x & (1<< (n - 1))
- 仅将第n位置为1 x | (1 << n)
- 仅将第n位置为0 x & (~(1 << n))
- 将x最高位至第n位(含)清零 x & ((1 << n) - 1)
- 将第n位至第0位(含)清零 X& (~((1 << (n + 1)) - 1))

示例代码

```
/*
```

```
    *
    位运算
    x & -x ： 得到最低位的1
    x & (x-1)： 清零最低位的1
    x & ((1 << n) - 1)： 将x最高位至第n位(含)清零
    (cols | pie | na) ===> 0
    ~(cols | pie | na) ===> -1    将0和1的位置置换
    举例说明     0101010   原先0 代表可放   1为不可放
    变成        1010101   转换后   1代表可放     0 为不可放
    ~(cols | pie | na) & ((1 << n) - 1); 11111111

    let p = bits & -bits;  得到最低位的1
    bits = bits & (bits - 1); 清零最低位的1
    dfs(n, row + 1, cols | p, (pie | p) << 1, (na | p) >> 1);

    说下各个参数的意义
    row+1   为下一层
    cols | p   看图   整个列  需要排除  p所在的这一列
    (pie | p) << 1   下一层   需要排除p所在的这一列的左移一位
    (na | p) >> 1    下一层   需要排除p所在的这一列的右移一位
    */
    let res = 0;
    const dfs = (n, row, cols, pie, na) => {
        if (row >= n) {
            res++;
            return;
        }
        // 得到当前所有的空位
        let bits = ~(cols | pie | na) & ((1 << n) - 1);
        while (bits) {
            // 取最低位的1
            let p = bits & -bits;
            // 把P位置上放入皇后
            bits = bits & (bits - 1);
            dfs(n, row + 1, cols | p, (pie | p) << 1, (na | p) >> 1);
        }
    };
    dfs(n, 0, 0, 0, 0);
    return res;
```

资料：

[知乎：别再问我N皇后了]()

[漫画：什么是八皇后问题]()

贪心算法

> 贪心算法 又称 贪婪算法，在对问题求解的时候，总是做出当前看来最好的选择，问题能够分解成子问题来解决，子问题的最优解能够递推到最终问题的最优解

贪心算法与动态规划的区别：它对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能

股票买卖的最佳时机II

```
给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。
设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。
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: [7,1,5,3,6,4]
输出: 7
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出,
     这笔交易所能获得利润 = 5-1 = 4 。
     随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出,
     这笔交易所能获得利润 = 6-3 = 3 。

示例 2:
输入: [1,2,3,4,5]
输出: 4
解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出,
     这笔交易所能获得利润 = 5-1 = 4 。
     注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
     因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

示例 3:
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。
```

解题策略：

```
算法流程：
设 tmp 为第 i-1 日买入与第 i 日卖出赚取的利润，即 tmp = prices[i] - prices[i - 1] ；
复杂度分析：
时间复杂度 O(N) ： 只需遍历一次 price；
空间复杂度 O(1) ： 变量使用常数额外空间。
```

示例代码

```
/**
 * @param {number[]} prices
 * @return {number}
 */
var maxProfit = function (prices) {
    let profit = 0;
    for (let i = 1; i < prices.length; i++) {
        let temp = prices[i] - prices[i - 1];
        profit = temp > 0 ? profit + temp : profit;
    }
    return profit;
};
```

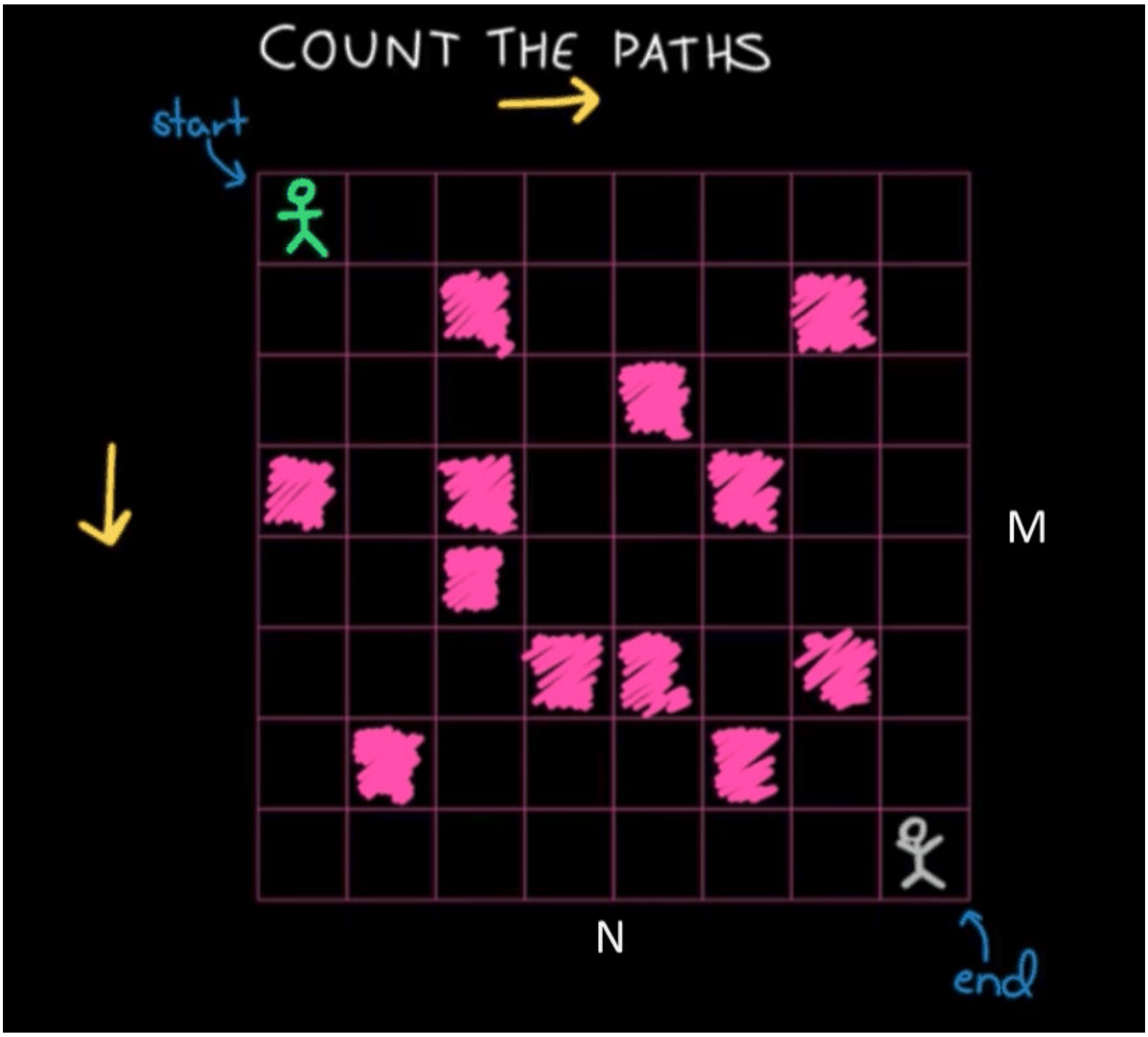
动态规划

> 通过把原问题分解为相对简单的子问题的方式求解复杂问题,常常适用于有重叠子问题和最优子结构性质的问题

从a到b点 有多少种走法

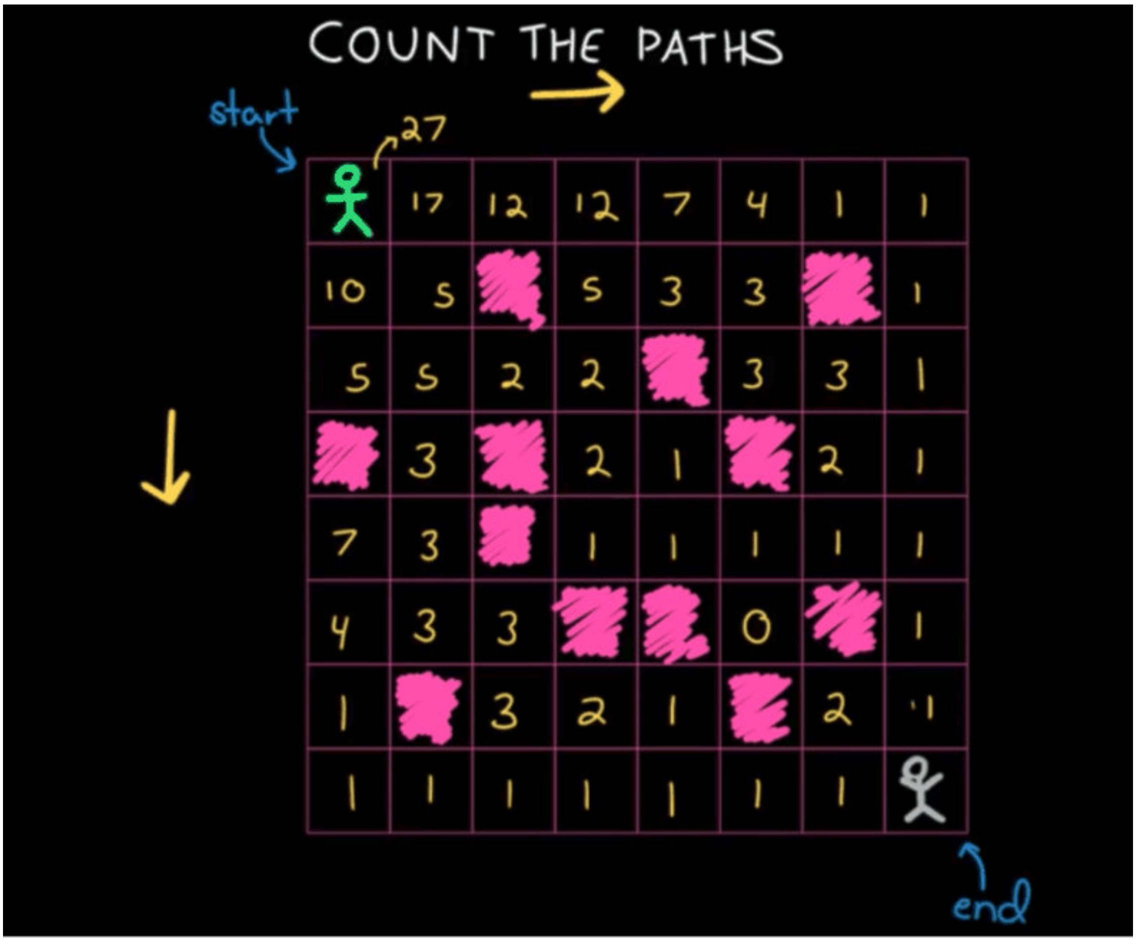
题目:

在一个8*8的平面板上， 有start移动到end点，每次只可以横向或者竖向移动，不能回退，中间的红色部分为障碍物，移动过程中需要避开障碍物， 有多少种走法?



解法

- 自下而上的递推
- 确定初始条件
- 确定状态转移方程 `opt[i][j] = opt[i - 1][j] + opt[i][j - 1]`




```
let arr = [
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 1, 0, 0, 0],
        [1, 0, 1, 0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 1, 0],
        [0, 1, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
    ];

    function countIngPaths(arr) {
        let m = arr.length;
        let n = arr[0].length;
        let opt = [];
        arr.forEach((item) => {
            opt.push(new Array(8).fill(0));
        });
        for (let i = 0; i < m; i++) {
            for (let j = 0; j < n; j++) {
                if (arr[m - i - 1][n - j - 1] === 1) {
                    opt[i][j] = 0;
                } else {
```

```
                    if (i === 0 || j === 0) {
                        opt[i][j] = 1;
                    } else {
                        opt[i][j] = opt[i - 1][j] + opt[i][j - 1];
                    }
                }
            }
        }
        return opt[m-1][n-1]
    }
    countIngPaths(arr);
```

三角形的最小路径和

```
给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。
相邻的结点 在这里指的是 下标 与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。

 例如，给定三角形：

[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
自顶向下的最小路径和为 11（即，2 + 3 + 5 + 1 = 11）。

说明：
如果你可以只使用 O(n) 的额外空间（n 为三角形的总行数）来解决这个问题，那么你的算法会很加分。
```

递归+回溯

- 确定递归的终止条件
- 确定递归体
- 算法自上而下 算出下一层相邻节点的最小值，继而递归下一层

```
    let triangle = [[2], [3, 4], [6, 5, 7], [4, 1, 8, 3]];
    var minimumTotal = function (triangle) {
        let rows = 0;
        let cols = 0;
        return getMinPaths(triangle, rows, cols);
    };

    function getMinPaths(triangle, rows, cols) {
```

```
        if (rows === triangle.length - 1) {
            return triangle[rows][cols];
        }
        const left = getMinPaths(triangle, rows + 1, cols);
        const rigth = getMinPaths(triangle, rows + 1, cols + 1);
        return Math.min(left, rigth) + triangle[rows][cols];
    }
```

动态规划解决

```
[
     [2],
    [3,4],    i-1
   [6,5,7],   i,j
  [4,1,8,3]  i+1,
]
```

- 自上而下
- 初始状态为首行首列

```
 let triangle = [[2], [3, 4], [6, 5, 7], [4, 1, 8, 3]];
    var minimumTotal = function (triangle) {
        let m = triangle.length;
        for (let i = 1; i < m; i++) {
            for (let j = 0; j <= i; j++) {
                triangle[i][j] =
                    Math.min(turnValue(triangle[i - 1][j]),
turnValue(triangle[i - 1][j - 1])) +
                    triangle[i][j];
            }
        }
        return Math.min(...triangle[m - 1]);  // 取最后一行中的最小值
    };
    function turnValue(value) {
        return value === undefined ? Infinity : value;
    }
    console.log(minimumTotal(triangle));
```

- 自下而上
- 初始状态为最后一行

疑问解读:

问题：为什么`let i = len - 2; i >= 0; i--`

如果我们写成 `let i=0; i< dp.length;i++` 会有什么样的问题?

```
首先我们是自下而上的推理
由下一步  推导上一步的结果
  如果是这种let i=0; i< dp.length;i++
  举例:  dp[1,0] = Math.min(dp[2,1],dp[2,0])+dp[1,0]
那么  我们在获取dp1的时候   需要知道Math.min(dp[2,1],dp[2,0])的值 ，但是此时是无法感知的
```

示例代码

```
let triangle = [[2], [3, 4], [6, 5, 7], [4, 1, 8, 3]];
    var minimumTotal = function (triangle) {
        let dp = JSON.parse(JSON.stringify(triangle));
        const len = dp.length;
        for (let i = len - 2; i >= 0; i--) {
            for (let j = 0; j <= i; j++) {
                dp[i][j] = Math.min(dp[i + 1][j], dp[i + 1][j + 1]) +
triangle[i][j];
            }
        }
        return dp[0][0];
    };
    console.log(minimumTotal(triangle));
```

最后

数据结构和算法的知识体系很庞大，里面衍生的问题很多。很多场景可以通过多种方法来解决，

希望大家可以通过这次分享对算法有一个比较系统的了解。多多动手才是王道~

毕竟，行胜于言~

谢谢各位前来捧场~