- 基础的数据结构必须掌握
- 暴力法很重要，只不过后面更多的时候想做出来的是性能更高的算法
- 了解暴力法的算法瓶颈以及各种数据结构的特点，通过暴力法先找到问题的可行解、大概解，然后通过组合各种数据结构去优化，直至逼近最优解
- 再往后就是必须要掌握的一些算法，比如 搜索、剪枝、空间换时间、二分法等，其中搜索和空间换时间下面又有小类别，比如dfs、bfs、字符串搜索的RK/KMP、哈希表、前缀和等等
- 算法复杂度讲解：https://github.com/suukii/Articles/blob/master/articles/dsa/big_O_complexity.md
- 基础数据结构
  - 线性结构
    - 数组 array
    - 队列 queue

      > 先进先出 FIFO (first in, first out)

      HTTP协议，层层装包，层层拆包

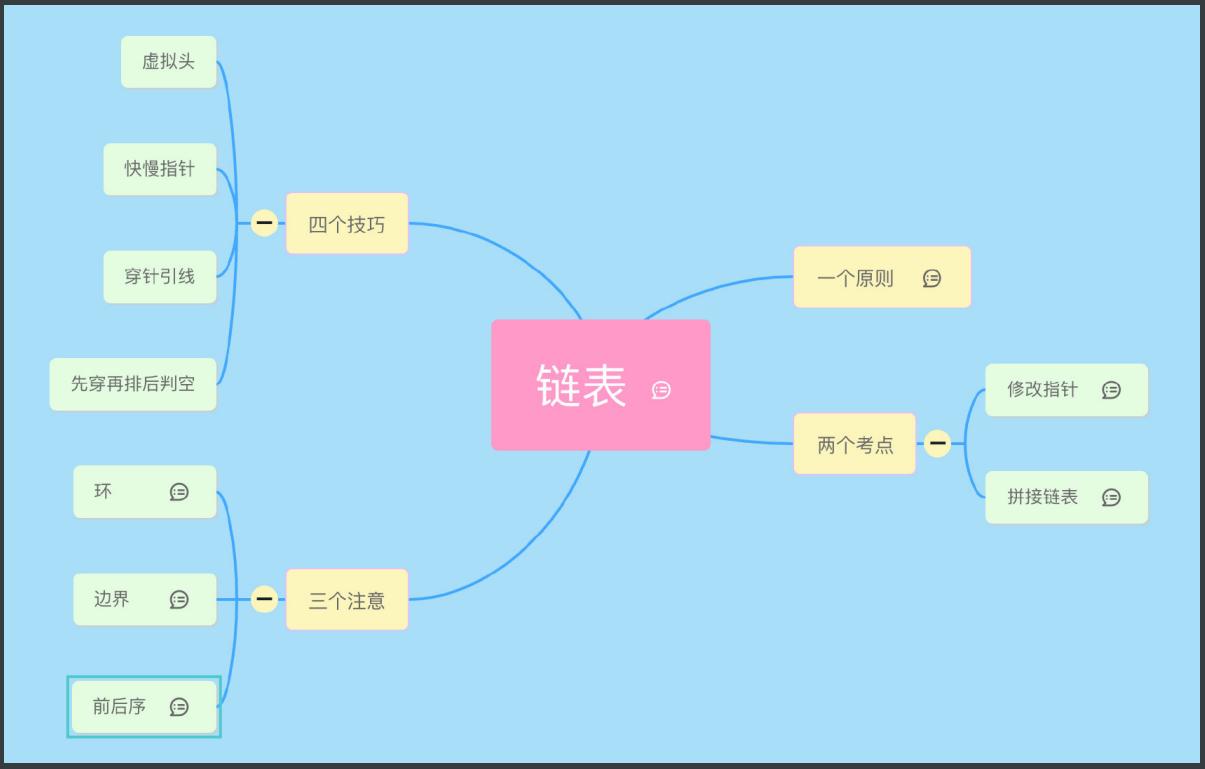
      常见的应用有优先队列
    - 栈 stack

      > 后进先出 LIFO (last in, first out)

      常见的应用有进制转换，括号匹配，单调栈，栈混洗，中缀表达式（用的很少），后缀表达式（逆波兰表达式）等。
    - 链表

      > 理解比较慢的一定要「多画图」！

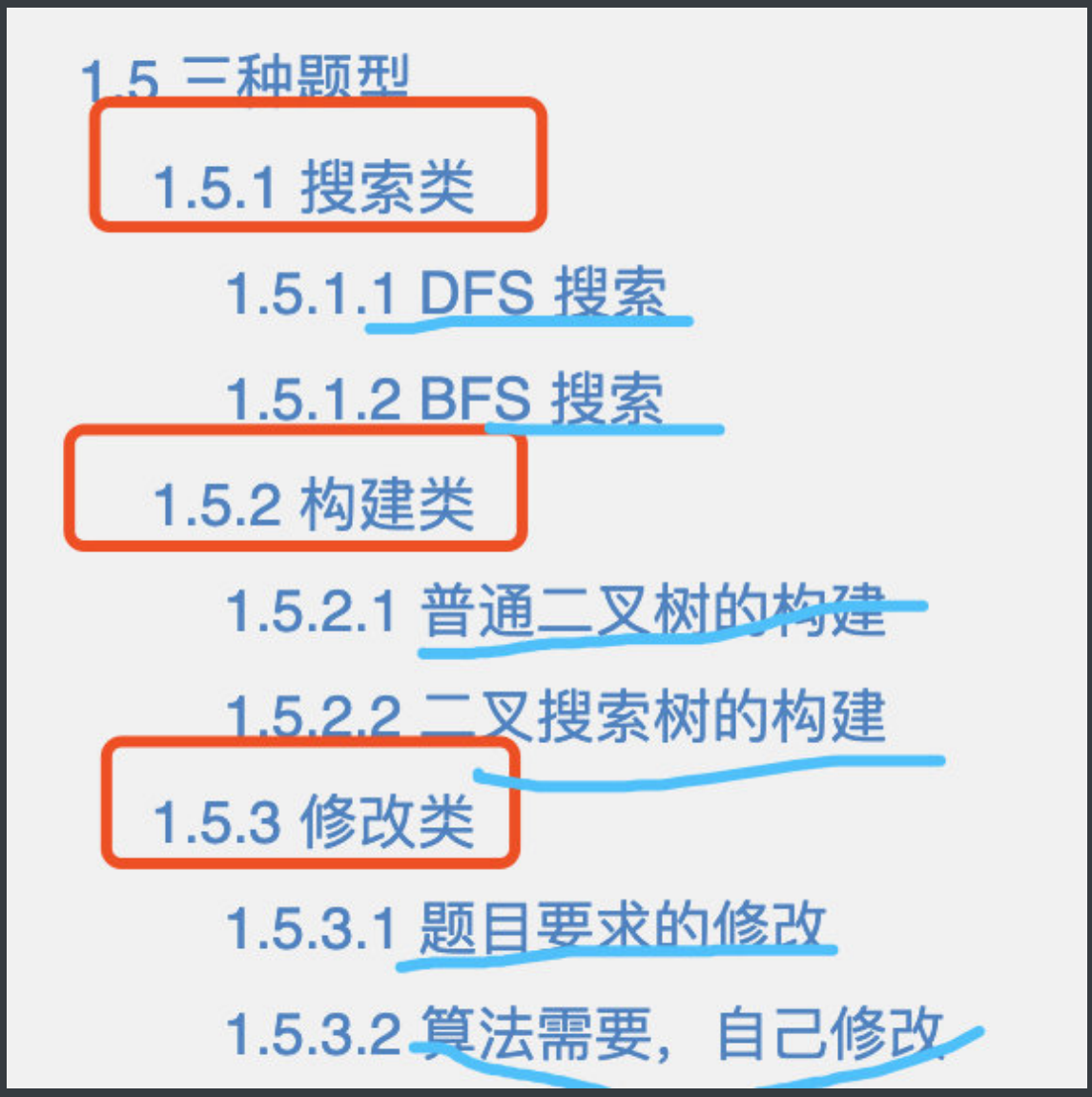
      合并K个有序链表、寻找环、相交链表、回文链表、链表反转、翻转链表的一部分等



- 非线性结构
  - 树
    - 二叉树

      前中后层序遍历、生成二叉树、寻找深度、根节点到叶子节点的和、翻转二叉树、对称二叉树、二叉树的最大宽度

      > BFS 的核心在于求最短问题时候可以提前终止，这才是它的核心价值，层次遍历是不需要提前终止 BFS 的副产物



- AVL平衡二叉树（很少考）

  最早被发明的自平衡二叉查找树。在 AVL 树中，任一节点对应的两棵子树的最大高度差为 1，因此它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下的时间复杂度都是 O(logn)。增加和删除元素的操作则可能需要借由一次或多次树旋转，以实现树的重新平衡。

- 二叉搜索树

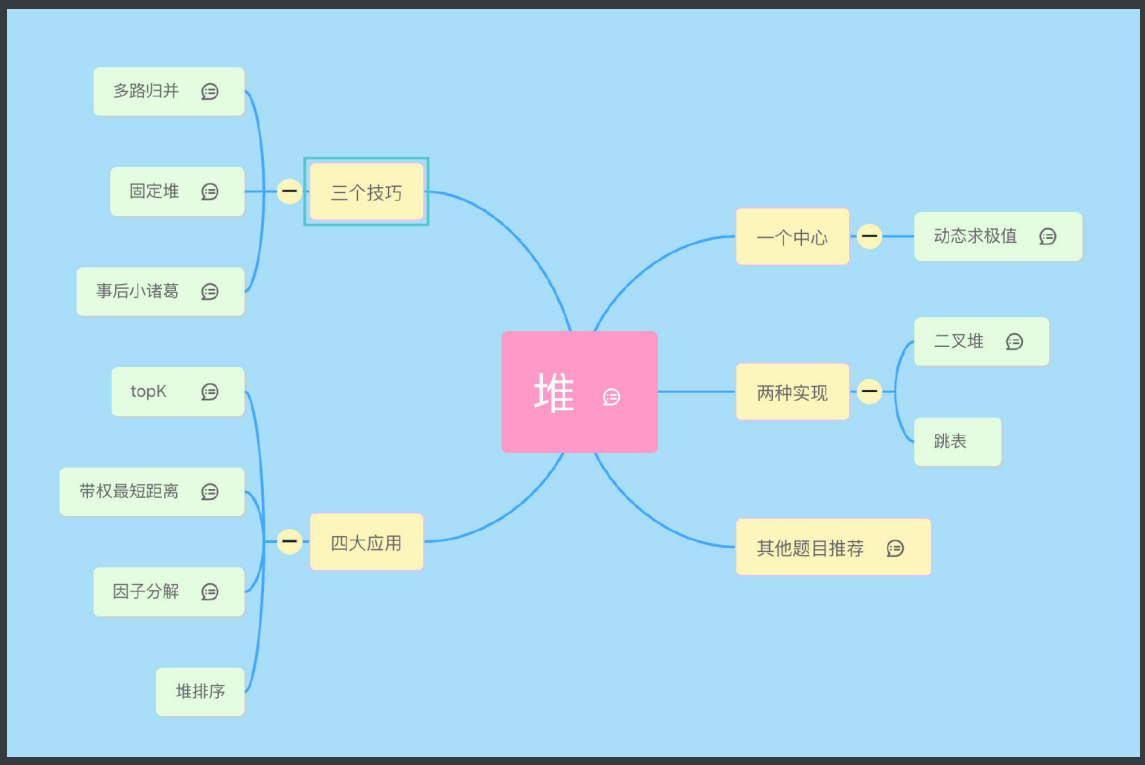
  1. 若左子树不空，则左子树上所有节点的值均小于它的根节点的值；
  2. 若右子树不空，则右子树上所有节点的值均大于它的根节点的值；
  3. 左、右子树也分别称为二叉排序树；
  4. 没有键值相等的节点。

- Tire树（字典树/前缀树）

  immutableJS，immer

- N叉树

- 堆（像国内面试的话，前端对于堆的要求几乎为0，但是外企丢这块会很看重，也包括下面的图）



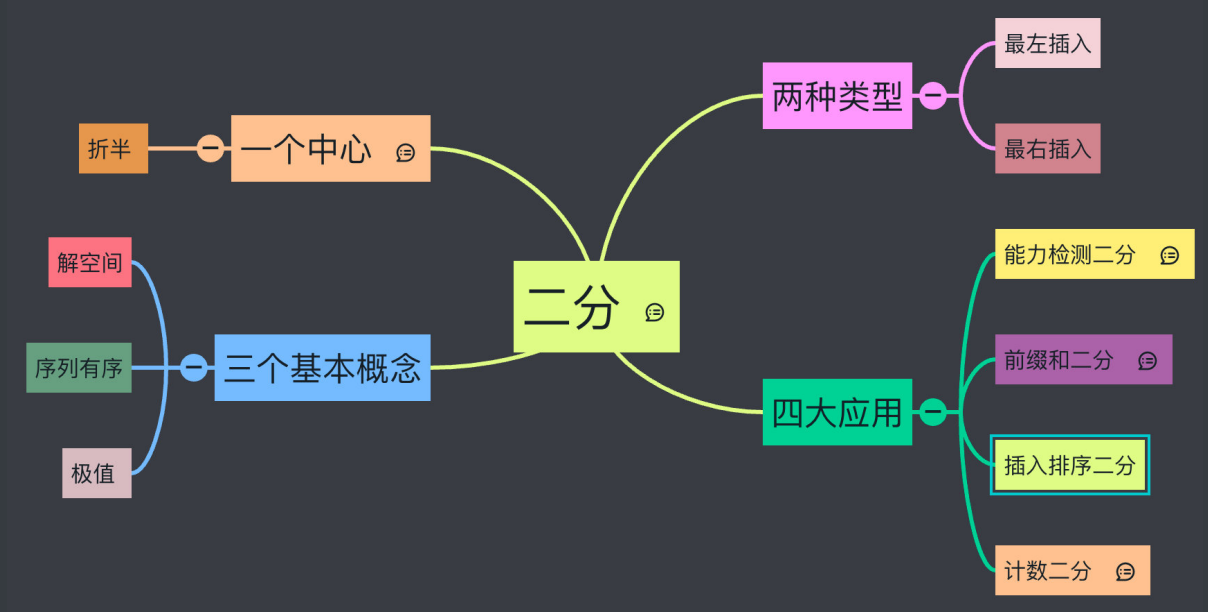
- 图（国内面试很少了，有兴趣可以了解下，然后连通性这块其实很多算不上是图吧，还是有一部分是需要向量和权重知识）

  一般的图题目有两种，一种是搜索题目，一种是动态规划题目。基本上都是先建图，再遍历寻找可行解。

  拓扑排序（bfs和dfs都可）、最小生成树（出现频率很低）、寻找最短路径（Dijkstra 算法、Floyd-Warshall 算法、贝尔曼-福特算法-bellman）、二分图（基本不用看）

- 算法类
  - 栈
    - 有效的括号、逆波兰表达式求值、字符串解码
  - 二分
    - 搜索旋转排序数组、搜索二维矩阵、最长上升子序列、水位上升的泳池中游泳
    - **二分法的本质在于它的「二段性」而不是「单调性」**
    - 一个问题能否用二分解决的关键在于检测一个值的时候是否可以排除解空间中的一半元素。比如我前面反复提到的**如果 x 不行，那么解空间中所有小于等于 x 的值都不行**。
    - 大多有序、定义搜索区间、定义循环结束条件、取中间元素和目标元素做对比**很重要！！**、收缩区间，舍弃非法解

- > 查找一个数
- > 寻找最左边的满足条件的值
- > 寻找最右边的满足条件的值
- > 寻找最左插入位置
- > 寻找最右插入位置



- 排序（各排序算法原理、复杂度、稳定性）
  - https://blog.csdn.net/yushiyi6453/article/details/76407640

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

- **双指针**（滑动窗口、快慢指针、首尾指针）
  - 环形链表、盛最多水的容器、爱生气的书店老板(滑动窗口经典题)
- 回溯（核心是枚举）
  - 路经总和系列、全排列系列、单词搜索系列、格雷编码、括号生成
- 贪心
  - 分发饼干、无重叠区间
- 第K大（堆、优先队列、单调栈）
  - 前K个高频元素、有序矩阵中第K小的元素、数组中第K个最大元素
- 动态规划（记忆化递归）
  - 背包问题、最长子序列、爬楼梯、零钱兑换系列、股票系列、打家劫舍系列
- 前缀和
  - 区间子数组个数、子数组异或查询、删除一次得到子数组最大和
- 位运算
  - 汉明距离、子集、解码异或后的数组、位1的个数、二进制手表

```
x>>1 和 x>>>1 的区别
算术右移 >>  ：舍弃最低位，高位用「符号位」填补。也叫有符号右移
逻辑右移 >>> ：舍弃最低位，高位用 0 填补。也叫无符号右移
```

对于负数而言，其二进制最高位是 1，如果使用算术右移，那么高位填补的仍然是 1。也就是 n 永远不会为 0。
比如，在二分查找的时候，mid越界了，或者说 left+right >> 1 中的 left+right 为负数，就是说最终值永远是负数这样。

n & (n - 1)
每次都会消除掉二进制最后一个为1的位，比如：
101 & 100 -> 100
10110 & 10101 -> 10100
// 位与 ‘&’ 当两个二进制位都为1的时候才为1

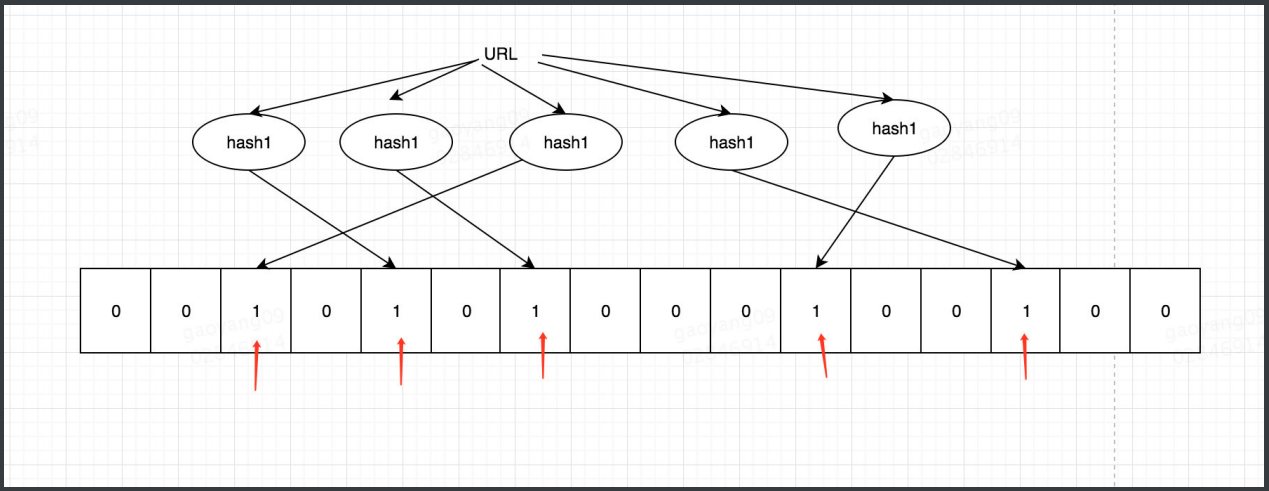
为什么二进制左移nums的长度，就是所有子集的数量？？ 2^N个
这里有个概念，是「笛卡尔积」。每个数字出现与不出现都是最多两次，所以最终结果就是2^N个

path.length >= 1 << n
1<<n 其实就是 2**n

两个数字异或的结果a^b是将 a 和 b 的二进制每一位进行运算，得出的数字。
运算的逻辑是果同一位的数字相同则为 0，不同则为 1

- 小岛题合集（小鹏汽车就问这个了）
  - https://leetcode-cn.com/problems/number-of-islands/
  - https://leetcode-cn.com/problems/max-area-of-island/solution/695-dao-yu-de-zui-da-mian-ji-dfspython3-by-fe-luci/
  - https://leetcode-cn.com/problems/as-far-from-land-as-possible/solution/python-tu-jie-chao-jian-dan-de-bfs1162-di-tu-fen-x/
  - 扩展，如果数据量很大怎么办，比如10^9 * 10^9. 分割+组合（连通性-并查集）
  - 所谓的小岛题，核心就是是求连通区域。不严谨的讲，小岛问题是 DFS 的子专题。
- 并查集
  - 朋友圈、连通网络的操作次数、尽量减少恶意软件的传播
- 单调栈 适合的题目是求解**下一个大于 xxx**或者**下一个小于 xxx**这种题目。或者部分第K个

    - 柱状图中最大的矩形、每日温度、移掉K位数字
- 扩展一些必须要知道的知识
    - 布隆过滤器 - 解决了**可能存在** 和 **一定不存在** 的问题。
        - 布隆过滤器其实就是 `bit + 多个散列函数` 。k 次 hash(ip) 会生成多个索引，并将其 k 个索引位置的二进制置为 1。
        - 如果经过 k 个索引位置的值都为 1，那么认为其**可能存在**(因为有冲突的可能)。
        - 如果有一个不为 1，那么**一定不存在**（一个值经过散列函数得到的值一定是唯一的），这也是布隆过滤器的一个重要特点。



    - 笛卡尔积
    - 排列组合相关知识
    - 通过数据规模快速区分复杂度的方法

数据规模	算法可接受时间复杂度
<= 10	O(n!)
<= 20	O(2^n)
<= 100	O(n^4)
<= 500	O(n^3)
<= 2500	O(n^2)
<= 10^6	O(nlogn)
<= 10^7	n
<= 10^14	$O(\sqrt{n})$
-	O(logn)

- 部分减少代码量的快捷操作（熟能生巧）
  - mid = Math.floor(left - (left + right)/2) - > Math.floor((left + right) /2) -> left+right >> 1 「部分情况下需要用 >>> 」
  - **二分法的本质在于它的「二段性」而不是「单调性」**，所以寻找的一定是题目能否允许在满足某个条件时舍去一部分
  - 简单的判定1次或者0次的组合问题，不考虑重复的情况下，情况一共有2**n个，比如 123 相互排列组合，共有8种情况

    ```
    000
    111
    ```

  - 进制转换，十进制转k进制：`a1 * k0 + a2 * k1 + ... + an * kn-1`

    ```
    func sum_with(base, N):
        return sum(1 * base ** i for i in range(N))
    ```

- 等比数列求和公式可以来简化时间复杂度 (最小好进制)

```
const sum_with = (base, N) => Math.floor((1 - base ** N) / (1 - base))
```

- 一定要注意大数字的取模或者使用BigInt。 x % 1000000007
- 搜索二叉树定义：**二叉树上任何一个节点的左子树上的点都比该节点小，右子树上的点都比该节点大，这样的二叉树称为搜索二叉树。**
- 位运算-异或运算，**两次异或相当于无操作**

```
arr[i] ^ arr[i+1] = encoded[i]
arr[i] ^ encoded[i] = arr[i+1]

a^a=0; 任何数字和自己异或都是0
a^0=a; 任何数字和0异或还是他自己
a^b^a=a^a^b 异或运算具有交换律

二进制。
000 ^ 101 -> 101
101 ^ 111 -> 010
```

- 已知两个点，A(x1,y1)、B(x2,y2)，两点之间的距离为：|AB| = v``````(x2-x1)^2 + (y2-y1)^2

- 如何学会通过多角度去看待一个问题(动态规划（含背包）、二分)
- 我刷题常用的一些工具和模版函数
  - https://chrome.google.com/webstore/detail/leetcode-cheatsheet/fniccleejlofifaakbgppmbbcdfjonle
  - https://wiki.jikexueyuan.com/list/sort/
  - https://binarysearch.com/
  - B站、YouTube
- 如何在面试前快速提升算法面试通过率
  - https://codetop.cc/home
  - 面试小抄

部分代码模版：

```
// 并查集
class UF {
    constructor(size) {
        this.parents = Array(size).fill(0).map((_, i) => i)
        this.sizes = Array(size).fill(1)
        this.cnt = size
    }

    // 得到集合数量
    getSizeOfSet(x) {
        const px = this.findSet(x)
        return this.sizes[px]
    }

    // 查找元
    findSet(x) {
        if (x !== this.parents[x]) {
            // 路径优化，在findSet进行 会将这个节点到元的所有元素的parents指向元
            this.parents[x] = this.findSet(this.parents[x])
        }
        return this.parents[x]
    }

    // 合并集合
    unionSet(x, y) {
        if (this.connected(x, y)) return
        const px = this.findSet(x);
        const py = this.findSet(y);
        // 将较短的集合的元的parents指向较长的集合的元
        if (this.sizes[px] > this.sizes[py]) {
            this.parents[py] = px;
            this.sizes[px] += this.sizes[py];
        } else {
```

```
            this.parents[px] = py;
            this.sizes[py] += this.sizes[px];
        }
        this.cnt -= 1
    }

    // 连通性
    connected(x, y) {
        return this.findSet(x) === this.findSet(y)
    }
}
```

```
// 并查集TS版本
class UnionFind {
    private parents: Array<number>;
    private sizes: Array<number>;

    constructor(size: number) {
        this.parents = Array(size)
            .fill(0)
            .map((_, i) => i);
        this.sizes = Array(size).fill(1);
    }

    getSizeOfSet(x: number): number {
        const px = this.findSet(x);
        return this.sizes[px];
    }

    findSet(x: number): number {
        if (x !== this.parents[x]) {
            this.parents[x] = this.findSet(this.parents[x]);
        }
        return this.parents[x];
    }
```

```
    unionSet(x: number, y: number): void {
        const px: number = this.findSet(x);
        const py: number = this.findSet(y);
        if (px === py) return;
        if (this.sizes[px] > this.sizes[py]) {
            this.parents[py] = px;
            this.sizes[px] += this.sizes[py];
        } else {
            this.parents[px] = py;
            this.sizes[py] += this.sizes[px];
        }
    }
}
```

```
// 翻转链表

当前指针 =  头指针
前一个节点 = null;
while 当前指针不为空 {
  下一个节点 = 当前指针.next;
    当前指针.next = 前一个节点
    前一个节点 = 当前指针
    当前指针 = 下一个节点
}
return 前一个节点;
```

```
// 判断质数
function isPrimes(num) {
    //两个较小数另外处理
    if (num == 2 || num == 3)
        return 1;
    //不在6的倍数两侧的一定不是质数
    if (num % 6 != 1 && num % 6 != 5)
        return 0;
```

```
    let tmp = parseInt(Math.sqrt(num));
    //在6的倍数两侧的也可能不是质数
    for (let i = 5; i <= tmp; i += 6)
        if (num % i == 0 || num % (i + 2) == 0)
            return 0;
    //排除所有，剩余的是质数
    return 1;
}
```

```
// 单调栈
const stack = []; // 定义一个栈
const resLen = num.length - k;
for (let i = 0; i < num.length; i++) {
  // 判断栈存在，并且栈的最后一个元素与num[i]进行对比或其他操作
  while (k && stack.length && stack[stack.length - 1] > num[i]) {
    stack.pop() // 出栈
    k--
  }
  stack.push(num[i]) // 入栈
}
// 每次入栈的元素，根据题意而定
// 每次判断的条件也是大同小异，配合题目即可

// 单调栈-模版
function monostoneStack (T) {
  let stack = [];
  let result = [];
  for (let i = 0; i < T.length; i++) {
    result[i] = 0;
    while (stack.length > 0 && T[stack[stack.length - 1]] < T[i]) {
      let peek = stack.pop();
      result[peek] = i - peek;
    }
    stack.push(i);
  }
```

```
  return result;
};
```

```
// 滑动窗口 - 模版

初始化慢指针 = 0
初始化 ans

for 快指针 in 可迭代集合
    更新窗口内信息
    while 窗口内不符合题意
        扩展或者收缩窗口
        慢指针移动
    更新答案
返回 ans
```

```
// 小岛题 - 从一个或多个入口 DFS 即可。 DFS 的时候，我们往四个方向延伸即可。
seen = set()
def dfs(i, j):
  if i 越界 or j 越界: return
  if (i, j) in seen: return
  temp = board[i][j]
  # 标记为访问过
  seen.add((i, j))
  # 上
  dfs(i + 1, j)
  # 下
  dfs(i - 1, j)
  # 右
  dfs(i, j + 1)
  # 左
  dfs(i, j - 1)
  # 撤销标记
  seen.remove((i, j))
# 单点搜索
```

```
    dfs(0, 0)
    # 多点搜索
    for i in range(M):
        for j in range(N):
            dfs(i, j)
```

```
    // 回溯模版

    const visited = {}
    function dfs(i) {
        if (满足特定条件) {
            // 返回结果 or 退出搜索空间
        }

        visited[i] = true // 将当前状态标为已搜索
        dosomething(i) // 对i做一些操作
        for (根据i能到达的下个状态j) {
            if (!visited[j]) { // 如果状态j没有被搜索过
                dfs(j)
            }
        }
        undo(i) // 恢复i
    }
```

...

大厂主要考察的数据结构和算法

算法，主要是以下几种：

- 基础技巧：分治、二分、贪心
- 排序算法：快速排序、归并排序、计数排序
- 搜索算法：回溯、递归、深度优先遍历，广度优先遍历，二叉搜索树等
- 图论：最短路径、最小生成树
- 动态规划：背包问题、最长子序列

数据结构，主要有如下几种：

- 数组与链表：单 / 双向链表
- 栈与队列
- 哈希表
- 堆：最大堆 / 最小堆
- 树与图：最近公共祖先、并查集
- 字符串：前缀树（字典树） / 后缀树

坚持（如果你可以一直坚持下去，以上我所说的所有东西，你现在都可以没有！）

做到了以上几点，我们还需要坚持。这个其实是最难的，不管做什么事情，坚持都是最重要也是最难的。

坚持下去，会有突然间成长的一天。