数据结构与算法

- 数组 和 二维数组（矩阵）
  - cyc2018
- 链表
  - cyc2018
- 栈和队列
  - cyc2018
- Hash表
  - cyc2018
  - set 去重、记录 元素 存在与否
  - map 用来 计数 等
- 字符串
  - cyc2018
- 树：
  重点 框架！
  - 二叉树
  - 红黑树
- 图
  - cyc2018
    字节跳动 不考。。。
    浪费时间
    - 二分图：
      边的两端（顶点 --》相邻顶点） 不同着色 法！
    - 拓扑排序：
      入度 = 0 + 广度优先搜索 queue 来做
    - 并查集
      --》还没看
- 位运算

- 题库
  - github
    - 模板
    - labuladong
      - 备份网页
      - 本地文档
  - 本地代码
  - Lintcode
- 排序 算法
- 四大算法思想
  - 贪心
    - 1、思想：
      在 限制值 有限 的情况下，每次 做选择时，
      选择 每消耗一个限制值，获取的 期望值 更大的那个方案

      也可能是 获取 同样 期望值，限制值 消耗 最小，一样的意思！
    - 2、贪心算法 做出的最终选择，不一定是 最优的
      其正确性，需要验证，一般多举几个例子，验证下 就好了。

      不正确的原因是：
      场景 不适用
      前一步的选择，这里 会 限制死 后一步的选择，让其 在此时 不能 充分选择 全局最优的了！
  - 分治
    - 分而治之：
      适用场景：

      1、原问题 与 分解成的小问题 具有 相同的模式；

      2、原问题 分解成的子问题 之间 可以独立求解，子问题之间 没有相关性，

      这一点是 分治算法 跟 动态规划 的明显区别，等我们讲到动态规划的时候，会详细对比这两种算法；

      3、具有 分解的 终止条件，也就是说，当问题足够小时，可以直接求解；

      4、可以将子问题 合并成 原问题，而且 这个 合并操作的 复杂度 不能太高！
      否则就起不到 减小 算法 总体复杂度的效果了。
    - 常用场景：
      1、比如 归并排序

      2、也可以用于 分布式 等，
      比如 海量 爬取网页、数据处理
    - 实现方式：
      常用 递归！
  - 回溯
    - 实现方式：
      - 常用 递归！
      - 枚举 每一步可能的选择，递归 执行下去
    - 主要解决 搜索问题，
      即 从 一组 可能的解中，找出 一个 满足要求的解
    - 时间复杂度，一般是 指数级的，比较差
      相对来说，一般 动态规划 可以解决 类似问题，

      且 效率 提升很多
      不过 空间 复杂度 也会提升；

      所以说 dp 是一种 空间 换 时间的 方案
  - 动态规划

- 链表
  - leetcode 字节的：
  - leetcode
    链表：
    1、哨兵思想
    2、边界条件 一开始 就写上！

    1、两链表合并：
    (1)最简单的了，从头开始 两两比较，小的拼接上，别忘记 最后将 非空的，追加到 next上
    (2)递归做法，从头开始比较，小的拼接上，然后 让 小.next 代表之前链表 和 另一个链表 调用相同方法！

    2、单链表反转：
    （1）递归：函数内 递归 reverse(current , prev)
    （2）迭代 都行

    3、链表 对应元素相加
    (1) 长度不同时，短的补0；
    (2) 两结点数值相加，可能 >10，所以要用一个 carry 携带 sum / 10；
    所以 其实 sum = node1.value + node2.value + carry
    (3) 最后，两链表加完 跳出循环后，carry > 0，还要 拼接 一个 node

    4、链表 排序：比较难！
    （1）归并！和 数组排序一样的思路，找到 分区点（快、慢指针）；
    （2）为优化空间，归并思想 由 递归方式 改写成 迭代方式
    （从 小区间 合并排序，然后区间 依次x2 到 大区间），进行logn轮，当然，每轮拼接 俩小区间的 起始index 要自己计算

    5、判断链表有环
    (1) 从前往后，用 hashSet 记录 未访问过的结点，若 出现 某结点 已经在set里面了，说明 二次指向这个结点，这个就是 入环点
    (2) 用一个证明过的结论：
    用 快慢指针，从头开始遍历，两指针 第一次相遇的点（必然在环内），记为 intersect；
    然后 用另外两个同速度的指针，分别从 头 和 intersect 向后遍历，他们下一次 相遇的点 必然就是 入环点
    -->这个 可以 推导、证明的

    6、判断链表相交
    (1) 遍历一个list，用set记录所有的node；然后 再去遍历另一个list，发现contains 说明相交，否则不相交
    (2) 控制 相等路径法，让两个指针 分别从 两个链表的 起点 开始向后遍历，任一指针 到了当前链表的尾部，就让其指向 另一个链表起点再来一轮；这样的话，A+B 和 B+A 两条路径，走的路程是一样的；
    若 二者相交，必然在 相交点 就开始相遇，即 p1=p2；否则 直到最后也不会相遇（同时为null，也是相等，会跳出循环）；
    所以，跳出循环后，最后返回的要么是 相交点，要么是 null

    7、多链表合并（k个）
    (1) 暴力 合并：
    从头开始，两两合并，然后 和 下一个链表合并，直到最后
    (2) 利用优先级队列，即小顶堆：
    每次都 存k个链表的 最前面的结点，
    队列 一旦不为空，就poll出来
    （优先级队列、小顶堆的特性 会保证 入队时 就堆化，最终 最上面的是 最小的）
    加入到最终list中；
    然后，判断 被poll出的那个链表 后续还有没有结点，有的话，顶替上去，add入队列（还是会 自动排序）
    直到，最后 队列空了。返回 最终list 就好
    (3) 归并、合并：
    以 函数 merge(lists, left , right) 递归的 拆分 （middle）、合并 各小区间（区间 元素 此时是一个个的链表），直到 最后整个的合并完毕