面试中常见的时间复杂度

- O(1) 极少
- O(logn) 几乎都是二分法
- O(√n) 几乎是分解质因数
- O(n) 高频
- O(nlogn) 一般都可能要排序
- O(n^2) 数组，枚举，动态规划
- O(n^3) 数组，枚举，动态规划
- O(2n) 与组合有关的搜索
- O(n!) 与排列有关的搜索

据时间复杂度倒推算法是面试常用策略，如:比O(n)更优的时间复杂度只能是O(logn)的二分法

面试中是否使用 Recursion 的几个判断条件：

1. 面试官是否要求了不使用 Recursion （如果你不确定，就向面试官询问）
2. 不用 Recursion 是否会造成实现变得很复杂
3. Recursion 的深度是否会很深
4. 题目的考点是 Recursion vs Non-Recursion 还是就是考你是否会Recursion？ **记住：不要自己下判断，要跟面试官讨论！** 面试中通常极少考到 Non-Recursion，考 Recursion 的时候比较多

二分

基本思想

将n个元素分成个数大致相同的两半，取a[n/2]与欲查找的x作比较，如果x=a[n/2]则找到x，算法终止。如果x<a[n/2]，则我们只要在数组a的左半部继续搜索x（这里假设数组元素呈升序排列），如果x>a[n/2]，则我们只要在数组a的右 半部继续搜索x

实现二分

```
//非递归
class BinarySearch {
    public static int binarySearch(int[] nums, int low , int high, int value)
{
        //value是欲查找的值
        int left = low;
        int right = high - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (value < nums[mid]) {
                high = mid - 1;
            } else if (value > nums[mid]) {
                low = mid + 1;
            } else {
                return mid;
            }
        }
        return - 1;
    }
}
```

```
//递归
class BinarySearch {
    public static int binarySearch(int[] nums, int low, int high, int value) {
        if (low > high) {
            return -1;
        }
        int mid = left + (right - left) / 2;
        if (value < nums[mid]) {
            return binarySearch(nums, low, mid - 1, value);
        } else if(value > nums[mid]) {
            return binarySearch(nums, mid + 1, high, value);
        } else {
            return mid;
        }
    }
}
```

面试套路

- 一般会给一个数组，让你找数组中第一个/最后一个满足某个条件的位置
- 如果有重复的数？ 可以证明，无法保证在 Log(N) 的时间复杂度内解决 例子：[1,1,1,1,1....,1] 里藏着一个0 最坏情况下需要把每个位置上的1都看一遍，才能找到最后一个有0 的位置
- 保留下有解的那一半或者去掉无解的一半 Search in Rotated Sorted Array
- 确定答案范围 + 验证答案大小 Median of K Sorted Arrays

递归

什么是递归

函数自己调用自己。Recursion是代码的实现方式，并不能算是一种算法。递归就是当多重循环层数不确定的时候 一个更优雅的实现多重循环的方式。

```
def recursion(self, n, visited, path):
  if len(path) == n:
    # do something
    return

  for i in range(1, n+1):
    if i not in visited:
      path.append(i)
      self.recursion(n, visited, path)
      path.pop()
```

递归三要素

递归三要素是实现递归的重要步骤：

- 递归的定
- 递归的拆解
- 递归的出口

DFS与BFS

DFS

碰到找所有方案的题，基本可以确定是 DFS。除了二叉树以外的90%DFS的题，要么是排列，要么是组合

排列组合问题

组合问题

问题模型：求出所有满足条件的“组合”。 判断条件：组合中的元素是顺序无关的。 时间复杂度：与 2^n 相关

排列问题

问题模型：求出所有满足条件的“排列”。 判断条件：组合中的元素是顺序“相关”的。 时间复杂度：与 n! 相关

通用的DFS时间复杂度计算公式

O(答案个数 * 构造每个答案的时间)

基本方案

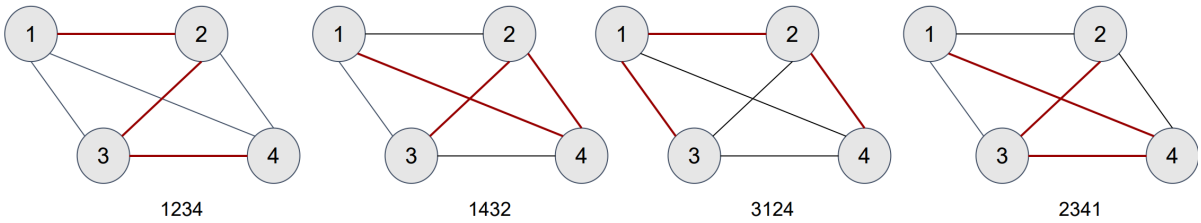
- 找N个数组成的全排列

  案例一：找N个数组成的全排列

  点：每个数为一个点

  边：任意两两点之间都有连边，且为无向边

  路径：= 排列 = 从任意点出发到任意点结束经过每个点一次且仅一次的路径



- 找所有满足某个条件的方案：

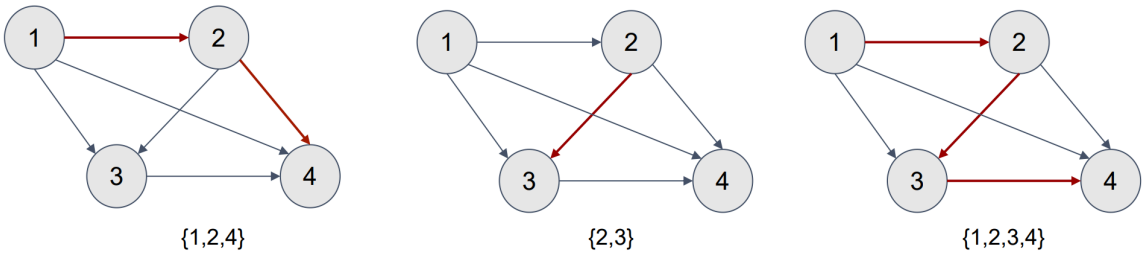
  1. 找到图中的所有满足条件的路径
  2. 路径 = 方案 = 图中节点的排列组合
  3. 点、边、路径需要自己分析

  案例二：找出一个集合的所有子集

  点：集合中的元素

  边：元素与元素之间用**有向边**连接，小的点指向大的点（为了避免选出12和21折后在哪个重复集合）

  路径： = 子集 = 图中任意点出发到任意点结束的一条路径



BFS

什么时候用

- 图的遍历 Traversal in Graph
  1. 层级遍历 Level Order Traversal

2. 由点及面 Connected Component
3. 拓扑排序 Topological Sorting

- 最短路径 Shortest Path in Simple Graph
- 非递归的方式找所有方案 Iteration solution for all possible results

二叉树上的宽度优先搜索

- 使用队列作为主要的数据结构 Queue
- 是否需要实现分层？ 需要分层的算法比不需要分层的算法多一个循环 Java / C++: size=queue.size() 如果直接 for (int i = 0; i < queue.size(); i++) 会怎么样？ 问：为什么 Python 可以直接写 for i in range(len(queue)) ?

图上的宽度优先搜索

- 哈希表

  图中存在环 存在环意味着，同一个节点可能重复进入队列

BFS的时间复杂度

O(N + M)

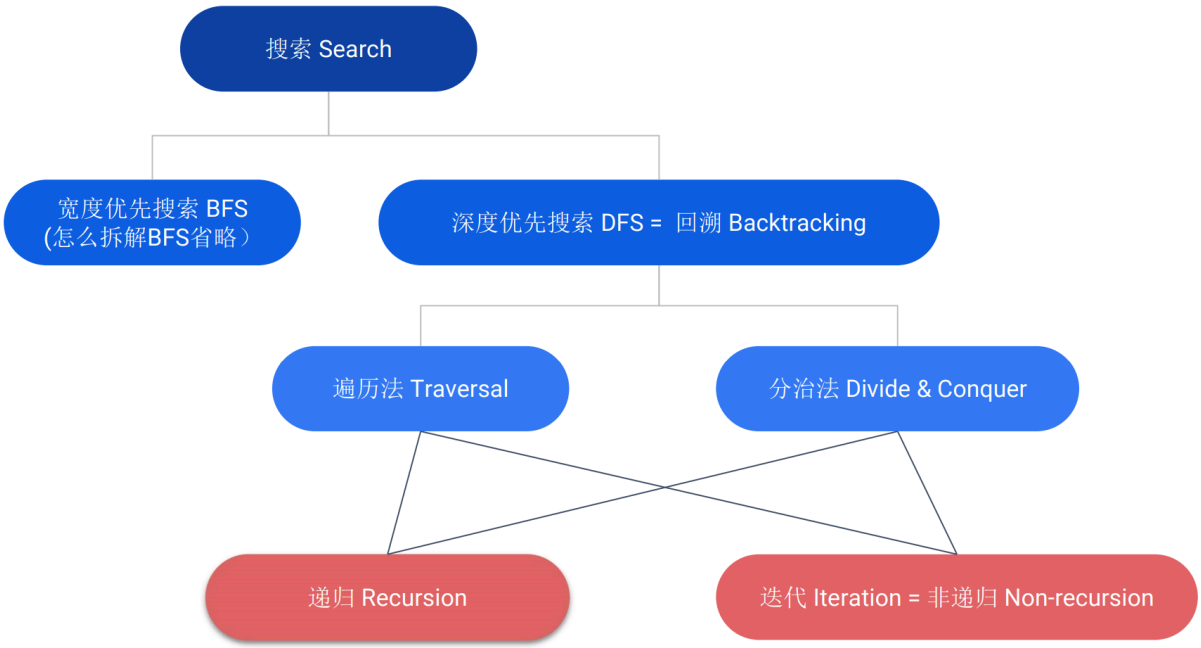
其中 N 为点数，M 为边数

拓扑排序

能够用 BFS 解决的问题，一定不要用 DFS 去做，因为用 Recursion 实现的 DFS 可能造成 StackOverflow

- 拓扑排序并不是传统的排序算法 一个图可能存在多个拓扑序（Topological Order），也可能不存在任何拓扑序
- 入度（In-degree）： 有向图（Directed Graph）中指向当前节点的点的个数（或指向当前节点的边的条数）
- 算法描述：
  1. 统计每个点的入度
  2. 将每个入度为 0 的点放入队列（Queue）中作为起始节点
  3. 不断从队列中拿出一个点，去掉这个点的所有连边（指向其他点的边），其他点的相应的入度 - 1
  4. 一旦发现新的入度为 0 的点，丢回队列中

一张图搞明白：递归，DFS，回溯，遍历，分治，迭代


搜索 Search
宽度优先搜索 BFS
(怎么拆解BFS省略）
深度优先搜索 DFS = 回溯 Backtracking
遍历法 Traversal
分治法 Divide & Conquer
递归 Recursion
迭代 Iteration = 非递归 Non-recursion