

数据结构与算法基础

2020 年 1 月 4 日

1 数据结构基础

- 打印加入 ([UTF8]{ctex})
- 选用 XeLaTeX / LuaLatex
- 文件名改为英文

1.1 List

1.1.1 定义

- Stores data elements based on an sequential, most commonly 0 based, index.
- Based on tuples from set theory. (元组 + 可变的集合 -> 可变的元组)。
- They are one of the oldest, most commonly used data structures.

1.1.2 重点

- Advantages: indexing
- Disadvantages: inserting(except append), and deleting(except pop).
- Linear data structures, are the most basic. 最基本结构: 线性数组.
 - Stack 栈
 - Queue 队列
- Non-linear data structures. 非线性数据结构
 - Graphs 图 $[[1,2], [3, 4]]$
 - Trees 树 中序遍历 $[1,[2,4,5],3]$

1.1.3 增删改查排及时间复杂度

- Append : $O(1)$
- Insertion: $O(N)$ ($l[a:b] = \dots$)
- Pop : $O(1)$

- Delete: $O(N)$ depends on i ; $O(N)$ in worst case
- Indexing: $O(1)$
- Search: traversal list $O(N)$
- Sort: $O(N\log N)$

1.1.4 基本函数

```
[2]: # stack (list)
A= [1,2,3]
A.append([4]) # [1, 2, 3, [4]]
A.extend([4]) # [1, 2, 3, [4], 4]
print('extend: ', A)
A.insert(1,4) # 在 i 处插入 value [1, 4, 2, 3, [4], 4]
A.remove(1) # 除去值为 i 的数 [4, 2, 3, [4], 4]
x = A.pop(3) # 除去第 i 个数 [4, 2, 3, 4]
x = A.index(4) # 0 返回符合的第一个序号 x = 0
A.reverse() # 转置
A.sort(reverse = True) # 排序 + 转置 [5, 3, 1]
from copy import copy, deepcopy
a = copy(A) # 浅复制
a = deepcopy(A) # 深复制
```

extend: [1, 2, 3, [4], 4]

1.2 Linked list

1.2.1 定义

- Stores data with nodes that point to other nodes. 用节点储存数据并指向其他节点
 - 节点的描述: Nodes, at its most basic it has one datum and one reference (another node) 在最基础的情况下, 节点拥有一个数据和一个参考 (其他节点)
 - 链表的描述: A linked list chains nodes together by pointing one node's reference to another. 节点通过指向另一个节点从而使链表连接在一起

1.2.2 重点

- Advantage: Add and deletion
- Disadvantage: indexing and searching.
- Doubly linked list has nodes that also references the previous node. 双向链表有两个参考, 一个是之前的 node, 一个是之后的 node

- Circularly linked list is simple linked list whose tail, the last node, references the head, the first node. 成环链表是由简单的链表首末相连组成
- 可以实现栈和队列

1.2.3 增删改查排及时间复杂度

- Add node: $O(1)$
- Deletion: $O(1)$
- indexing: $O(N)$
- Search: traversal linklist $O(N)$
- Sort: $O(N\log N)$

1.3 Stack, Queue and Heap queue(priority queue)

1.3.1 Stack 栈

- Last-In-First-Out (LIFO) concept
- `append()` / `pop()`
- can be made from **list**
- can be made from **linklist**, by having the head be the only place for insertion and removal.
可用链表构成, 插入: 建立新节点, 连接到头节点; 移除: 移除头节点, 返回下一个
– 参见

1.3.2 Queue 队列

- First-In-First-Out (FIFO) principle
- Lists are not efficient to implement a queue 列表不能有效的执行队列
– 故调用容器-双端队列 `deque(double-end-queue)`
– `from collections import deque d = deque(iterable)` (“deck”), If iterable is not specified, the new deque is empty.
– `append(x)` / `appendleft(x)` / `pop()` / `popleft()` / `extend(iterable)` / `extendleft(iterable)`
 $O(1)$ 双向 `append` or `pop`, $O(1)$
- can be made from **linklist** that only removes from head and adds to tail 也可用链表构成, 从头部移除, 从尾部添加
– 参见

1.3.3 Heap queue 堆

- 最小值在堆顶


1.3.4 基本函数

```
[2]: # Queue (deque) 队列
from collections import deque
B = 'desk' # iterable
d = deque(B) # 转化 B 为队列 d
d = deque() # 建立新队列 d
d.append('c')
d.appendleft('e') # 'edeskc'
d.extend(a)
d.extendleft(a) # 扩展最左边
x = d.pop() # 注意! 没有 i 参数 (list 有 i 参数)
d.popleft() # 删除最左边的, 即先“进”的, 无 (i)
d.reverse() # 转置
print(d)
d.rotate(1) # 向右循环移动 n 步。如果 n 是负数, 就向左循环。e.g 把倒数第 1 个调换到前面
print(d)
```

```
deque([3, 4, 4, 'c', 'e', 4, 4, 3])
```

```
deque([3, 3, 4, 4, 'c', 'e', 4, 4])
```

```
[3]: # Heap queue 堆
from heapq import heapify, heappop, heappush, heappushpop, heapreplace, \
    nlargest, nsmallest
# 创建一个堆, 可以使用 list 来初始化为 []
C = [] # 建立新 heapq
# 创建一个堆或者可以通过一个函数 heapify(), 来把一个 list 转换成堆
C = [2, 3, 4]
heapify(C) # 转化 C 为堆, 原地, 线性时间内
heappush(C, 2) # 添加 2, 注意! 只能添加值! 不能添加 list 等数据结构, 下面的删除也是这样!
x = heappop(C)
heappushpop(C, 2) # 先添加, 再删除顶端的
heapreplace(C, 5) # 先删除顶端的, 再添加
nlargest(2, C) # nlargest(n, iterable, key=None) 返回前 n 个符合 k 条件的最大值
(e.g. k = lambda x: x[0]) = sorted(iterable, key=key, reverse=True)[:n]
```

```
nsmallest(2, C) # nsmallest(n, iterable, key=None) 返回前 n 个最小值 =   
→ sorted(iterable, key=key, reverse=False)[:n]
```

[3]: [3, 4]

1.3.5 经典应用

1.4 Hash Table 及 string

1.4.1 定义

- 结构: Stores data with **key value pairs**.
- 功能: Hash function accept a key and return an output unique only to that specific key
 - This is known as hashing, which is the concept that an input and an output have a **one-to-one correspondence** to map information.
 - Hash function has a **unique address** in memory for that data.

1.4.2 重点

- 优势: 插入, 删除, 搜索 insertion, deletion, and searching
- Hash collisions(哈希冲突) are when a hash function returns the **same output** for **two distinct inputs**.
 - All hash function have its problem
 - This is often accommodated(解决) for **having the hash table being very large**
- Hashes are important for **dictionary** and **database indexing**.

1.4.3 增删改查排及空间复杂度

- Store : $O(1)$ $d[k] = v$
- Pop: $O(1)$ **d.pop(k)** 删除 key 对应的 pair, 并返回 value
- Pop item: $O(1)$ **d.popitem()** 删除并返回最后一个 pair
- Delete: $O(1)$ $\text{del } d[k]$
- Search: $O(1)$ $d[k]$

1.4.4 应用

应用总结

- string 相关概念
 - subsequence 可以不连续且有序的子序列 / substring 连续但有序的子序列/ the length of the longest palindromes that can be built with those letters 可以不连续, 可以无序的

1.5 二叉树 Binary Tree

1.5.1 定义

- 一个像树一样的数据结构，其中每个节点最多有两个孩子 a tree like data structure where every node has at most two children
- 有左右子节点 There is one left and right child node

1.5.2 重点

- 旨在优化搜索和排序 Designed to optimize searching and sorting
- 简并树是母节点只有一个子节点的树，如果完全单侧，则就是链表。A **degenerate tree** is an unbalanced tree, which if entirely one-sided is essentially a linked list.
- 它们比其他数据结构更容易实现。They are comparably simple to implement than other data structure.
- 二叉搜索树 **binary search tree**
 - 使用可比较键来指定孩子的方向。Uses comparable keys to assign which direction a child is.
 - 左子节点的键小于其父节点 Left child has a key smaller than its parent node.
 - 右子节点的键大于其父节点 Right child has a key greater than its parent node.
 - 不能有重复的节点 There can be no duplicate node.
 - 由于上述原因，它比二叉树更有可能用作数据结构 Because of the above it is more likely to be used as a data structure than a binary tree.

1.5.3 增删改查排及空间复杂度

- Insertion: $O(\log N)$
- Indexing: $O(\log N)$
- Search: $O(\log N)$

2 搜索基础 Search Basics

2.1 宽度优先搜索 Breadth First Search

2.1.1 定义

- 通过从根开始首先搜索树的级别来搜索树（或图）的算法 An algorithm that search a tree (or graph) by searching levels of the tree first, starting at the root.
- 它找到每个级别相同的节点，最经常从左到右移动。

- 在执行此操作时，它将跟踪当前级别上节点的子节点
- 检查完一个级别后，它将移至下一级别的最左侧节点
- 最右下角的节点最后被评估（最深且距离其级别最远的节点）。

2.1.2 重点

- 最适合搜索宽于深的树 Optimal for searching a tree that is **wider than it is deep**.
- 当遍历树时，使用队列来存储有关树的信息 Uses a **queue** to store information about the tree while it traverses a tree.
 - 因为它使用队列，所以比深度优先搜索要占用更多的内存。 Because it uses a queue it is more memory intensive than depth first search.
 - 队列使用更多的内存，因为它需要存储指针 (?) The queue uses more memory because it needs to store pointers.

2.1.3 复杂度

- Search: $O(V + E)$ under the graph is represented by the adjacency list structure
- Each edge is labeled twice. E is number of edges 边的数量
- Each vertex is labeled twice. V is number of vertices 顶点的数量

2.2 深度优先搜索 Depth First Search

2.2.1 定义

- 一种算法，该算法通过从根开始首先搜索树的深度来搜索树(或图) An algorithm that searches a tree (or graph) by searching depth of the tree first, starting at the root.
 - 它沿着一棵树向左走，直到不能走远为止。 It traverses left down a tree until it cannot go further.
 - 一旦到达分支的末尾，它将遍历该分支的右子节点，如果可能的话，再返回右子节点。 Once it reaches the end of a branch, it traverses back up trying the right child of nodes on that branch, and if possible left from the right children.
 - 检查完分支后，它移到根的右侧节点，然后尝试在所有子节点上向左移动，直到到达底部 When finished examining a branch, it moves to the node right of the root then tries to go left on all its children until it reaches the bottom.
 - 最右边的节点最后被评估（所有祖先右边的节点）。 The right most node is evaluated last(the node that is right of all its ancestors)

2.2.2 重点

- 最适合搜索比宽更深的树 Optimal for searching a tree that is deeper than it is wide

- 使用 list 将节点压入 Uses a stack to push nodes onto:
 - 由于堆栈是 LIFO，因此它不需要跟踪节点指针，因此与广度优先搜索相比，其内存密集度较低。Because a stack is LIFO, it does not need to keep track of the nodes pointers and is therefore less memory intensive than breath than BFS.
 - 一旦无法继续前进，它将开始评估 list 里存储的值 Once it can't go further left it begins evaluating the stack. 一旦无法继续前进，它将开始评估堆栈。

2.2.3 复杂度

- Search: $O(V + E)$ under the graph is represented by the adjacency list structure
- Each edege is labeled twice. E is number of edges 边的数量
- Each vertex is labeled twice. V is number of vertices 顶点的数量

2.2.4 BFS vs DFS

- The simple answer to this question is that it depends on the size and shape of the tree.
 - For wide, shallow tree use BFS 宽而浅
 - For deep, narrow tree use DFS. 深而窄

2.2.5 细微差别 Nuances

- 因为 BFS 使用队列来存储有关节点及其子节点的信息，所以它使用的内存可能比计算机上可用的内存更多（这种情况很少存在）。Because BFS uses queue to store information about the node and its children, it could use more memory than is available on your computer.
- 如果在非常深的树上使用 DFS，则可能会不必要地深入搜索，就需要减枝操作。If using a DFS on a tree that is very deep, you might go unnecessarily deep in the search.
- 宽度优先搜索往往是一种循环算法，即用 while 和 for。BFS tends to be a **looping algorithm**.
- 深度优先搜索往往是一种递归算法，即用递归算法。DFS tends to be a **recursive algorithm**.

3 高效排序基础（比较排序） Efficient Sorting Basic

- 基本应用：排序类题目
- 扩展应用：合并类题目（比较过程中，每个元素都遍历到了，故可以实现合并）
- 至少需要 time complexity: $O(N \log N)$

3.1 Merge sort 合并排序法

3.1.1 定义

A comparison based sorting algorithm, 流程如下

* Divide entire dataset into groups of at most two * Compares each number one at a time, moving

the smallest number to left of the pair. * Once all pairs sorted it, then compares left most elements of the two leftmost pairs to create sorted group of four with the smallest numbers on the left and the largest ones on the right. * This process is repeated until there in only one set. * 如图 Merge-Sort.png

3.1.2 重点

- This is one of most basic sorting algorithm.
- Know that it divides all the data into small possible sets then compares them.

3.1.3 复杂度

- Time
 - Relation: $T(n) = 2T(n/2) + O(n)$
 - Best Case Sort: $O(N\log N)$
 - Average Case Sort: $O(N\log N)$
 - Worst Case Sort: $O(N\log N)$
 - Prove the $N\log N$
 - * Recurrence Tree method: $T(n) = 2T(n/2) + O(n) = 2^{2T(n/2)} + 2O(n/2) + O(n)$
 $= 2^{2T(n/2)} + 2O(n) = 2^{mT(n/2)} + mO(n)$; 又因为 $n/2^m = 1$ 时, $m = \log(2)n$.
故 $T(n) = nT(1) + \log(2)n * O(n) = n + n\log(2)n = n\log_2(n) = n\log n$
 - * Master method $T(n) = 2T(n/2) + O(n)$, 故为 $n\log n$
- Space
 - $O(N)$
- 发挥性能的应用范围
 - data is huge
 - data is stored in external storage
 - 链表
- 稳定性
 - 稳定
- 原地性 In-place
 - 原地算法定义: (in-place algorithm) 基本上不需要额外辅助的数据结构, 然而, 允许少量额外的辅助变量来转换数据的算法。当算法运行时, 输入的数据通常会被要输出的部分覆盖掉
 - 非原地算法: 每次 merged list, 即新产生的 arr, 都不是原有的 arr 了, 而是小规模排序数组, 故用到了额外辅助的数据结构, 不属于原地算法

3.1.4 方法核心

- 把容器元素分拆成小单位，再依次比较、排列、合并

3.1.5 代码模板

```
[4]: '''
Merge Sort temple/basic code
edge case: list is None / only one element
过程 :
    MergeSort(arr[], l,  r)
    If r > l
    1. Find the middle point to divide the array into two halves:
        middle m = (l+r)/2
    2. Call mergeSort for first half:
        Call mergeSort(arr, l, m)
    3. Call mergeSort for second half:
        Call mergeSort(arr, m+1, r)
    4. Merge the two halves sorted in step 2 and 3:
        Call merge(arr, l, m, r)
'''

class SortList:
    # Merge sort
    def mergeSort(self, arr):
        if len(arr) == 0 or len(arr) == 1:
            return

        if len(arr) >= 2:
            # 普通划分
            mid = len(arr) // 2 # find the mid of the array
            L = arr[:mid] # divide the array elements
            R = arr[mid:] # into 2 halves

            # 递归划分
            self.mergeSort(L) # Sorting the first half
            self.mergeSort(R) # Sorting the second half
```

```

        # 合并, input: 两个已排序的 list, L, R. 过程: 对 L, R 进行合并排序。
output: 合并完成后的一个 list
        i = j = k = 0
        while i < len(L) and j < len(R): # copy data to temp arrays L and R
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # while i < len(L): # process left data in L part
        #     arr[k] = L[i]
        #     k += 1
        #     i += 1
        if i < len(L):
            arr[k:] = L[i:]

        # while j < len(R): # process right data in R part
        #     arr[k] = R[j]
        #     k += 1
        #     j += 1
        if j < len(R):
            arr[k:] = R[j:]

        return arr

x = SortList()
x.mergeSort([9,8,7,4, 5, 6, 3, 2, 1])

```

[4]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

3.2 自下而上排序法 Bottom-up Merge sort

3.2.1 定义

A comparison based sorting algorithm, * first merges pairs of adjacent lists of 1 element * Then merge pairs of adjacent lists of 2 elements * And next merge pairs of adjacent lists of 4 elements * And so on. Until the whole list is merged.

3.2.2 复杂度及属性

- Time
 - Relation: $T(n) = 2T(n/2) + O(n)$
 - Best Case Sort: $O(N\log N)$
 - Average Case Sort: $O(N\log N)$
 - Worst Case Sort: $O(N\log N)$
 - Prove the $N\log N$
 - * Recurrence Tree method: $T(n) = 2T(n/2) + O(n) = 2^{2T(n/2)} + 2O(n/2) + O(n)$
 $= 2^{2T(n/2)} + 2O(n) = 2^{mT(n/2)} + mO(n)$; 又因为 $n/2^m = 1$ 时, $m = \log(2)n$.
故 $T(n) = nT(1) + \log(2)n * O(n) = n + n\log(2)n = n\log(2)n = n\log n$
 - * Master theorem $T(n) = 2T(n/2) + O(n)$, 故为 $n\log n$
- Space
 - $O(1)$ # 随是否新建存储结构而变化
- 稳定性
 - 稳定
- 原地性
 - 原地

3.2.3 代码模板

```
[5]: # Bottom-up Merge Sort template/basic code
# edge case: list is None / only one element
# 易错点: 别忘记 while interval < length, subsort 求的是段, 23.Merge k Sorted list 求的是点

class SortList:
    def mergeSort(self, arr): # Merge sort by bottom-up merge sort
        if len(arr) == 0 or len(arr) == 1: # edge case
            return arr

        length = len(arr)
```

```

        interval = 1
        while interval < length:
            for i in range(0, length, 2 * interval): # bottom-up merge sort
                arr[i: i + 2 * interval] = self.subsort(arr[i: i + interval],
↪arr[i + interval: i + interval * 2])
                interval = interval * 2
        return arr

    def subsort(self, list1, list2): # sort two sorted list
        if not list1:
            return list2
        if not list2:
            return list1

        new_list = []
        left_len = len(list1)
        right_len = len(list2)
        i, j = 0, 0
        while i < left_len and j < right_len:
            if list1[i] < list2[j]:
                new_list.append(list1[i])
                i += 1
            else:
                new_list.append(list2[j])
                j += 1

        if i != left_len:
            new_list.extend(list1[i:left_len])
        if j != right_len:
            new_list.extend(list2[j:right_len])
        return new_list

x = SortList()
x.mergeSort([2,1,0])

```

[5]: [0, 1, 2]

3.2.4 应用（合并排序及自上而下排序）

总结

- 链表的排序，包括常数空间，非常数空间
- 需要两两比较的题目。e.g. 求转置数数量
- 合并类型题目。e.g. 合并 k 链表/列表
- 外置排序

解决方案模板

- 明确 基本结构与要求
 - 输入 含有数个元素容器
 - 输出 格式!!
 - * 原容器输出, 即原容器保持维度不变的输出 e.g. list sort -> 递归时直接 self.sort(L)
 - * 新参数输出, 即新参数, 或者原容器维度改变的输出 e.g. sort linklist, sort list[list]
-> 递归时 sub_l = self.sort(L)
 - 要求 合并方式排序元素等
- 明确 题目类型
- 确定 idea （解题算法）
- 思考 不同之处
- 修改 Method

排序链表，且不要求 space complexity - merge sort

- 为什么用方法？
 - 要求排序
 - 相对于快排，合并排序不需要太多的对链表节点的访问 (access)。（链表储存不是连续分块储存的 (continuous block of memory)，不利于节点访问，与 list 相反）
- 时间/空间复杂度
 - $O(N\log N)$ / $O(N)$
- 例题： 148. Sort List (linked list).py
 - 递归：递归部分返回头节点；每段的完整切割（要 next=None）；先确定 R 头，再确定 L 尾

排序链表，且 using constant space complexity. - bottom-up merge sort

- 为什么用这个方法？
 - using constant space complexity.
- 时间/空间复杂度

- $O(N \log N)$ / $O(1)$
- 例题: 148. Sort List (linked list).py
 - while interval < length + while head1 迭代;
 - dummy,fake-tail 的使用;
 - 独立 split 函数, input->head,interval ; output -> next head
 - 独立 merge 函数, input->head1,head2; output -> new head, new tail

计算转置数的数量 Inversion Count Problem

- 为什么用方法?
 - 利用 $\text{inv_count} = \text{inv_count} + (\text{len}(L) - i)$ 的性质
- 时间/空间复杂度
 - $O(N \log N)$ / $O(N)$
- 例题:
 - 775. Global and Local Inversions.py
 - 例题: Count Inversions in an array.py

合并 K 个排序数组 Merge k sorted arrays

3-way Merge Sort

外置排序 External Sorting

合并链表题

- 为什么用 merge sort?
 - 要求合并, 即排序的扩展
- 23. Merge k Sorted Lists.py - MergeSort method
 - 基本结构与要求:
 - * 输入: 含有数个元素为 link list 的 head 的容器
 - * 要求: 合并并排序所有 link list
 - * 输出: 合并后的 link list 的 head
 - 明确题目类型: 大合并+小排序类
 - idea: Merge Sort
 - 不同之处:
 - * 合并: 链表形式合并, 生成新部分链表
 - * 输出: 新生成的链表头, 而不是原容器
 - Method:

- * divide lists into groups of at most two
- * merge and sort linklist in every groups into one linklist
- * merge and sort every two new linklist to create new linklist
- * process is repeated until there is only one linklist

3.3 快速排序 Quick sort

3.3.1 定义

- A comparison based sorting algorithm (与 merge sort 相同)
 - Divides entire dataset in half by selecting the middle element and putting all smaller elements to the left of the element and larger ones to the right.
 - It repeats this process on the left side until it is comparing only two elements at which point the left side is sorted.
 - When the left side is finished sorting, it performs the same operation on the right side.
- Computer architecture favors the quick-sort process.

3.3.2 重点

- While it has the same Big O as (or worse in the same cases) many other sorting algorithm, it is often faster in practice than many other sorting algorithms, such as merge sort. 虽然有相同或者更坏的时间复杂度，但是实际中，它要更快，比如比归并快
- Know that it halves the data set by the average continuously until all the information is sorted.

3.3.3 复杂度及属性

- Time
 - Best case: $O(N \log N)$
 - * when the partition process always picks the middle element as pivot.
 - * equal to $T(n) = 2T(n/2) + O(n)$, can be proved by the Master theorem.
 - Average case: $O(N \log N)$
 - * when partition puts $O(n/9)$ elements in one set and $O(9n/10)$ elements in other set.
 - * $T(n) = T(n/9) + T(9n/10) + O(n)$
 - Worst case: $O(N^2)$
 - * when the partition process always picks greatest or smallest element as pivot
 - * equal to $T(n) = T(n-1) + O(n)$
 - Relation: $T(n) = T(k) + T(n - k - 1) + O(n)$
 - * $T(k)$, $T(n - k - 1)$ are for recursion calls

* $O(n)$ is for partition process

* k is the number of elements which are smaller than pivot

- Space
 - $O(N)$
- 发挥性能的应用范围
 - most architectures in most real-world data
 - 数组
 - implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data.
- 稳定性
 - 不稳定
 - 可以加入索引作为比较参数之一，即既比较值大小，也比较索引，来实现稳定。
- 原地性 In-place
 - 属于原地算法（直接修改输入数据而不是将输入数据复制一份处理之后再覆盖回去）
 - 即不考虑递归的情况下，使用的空间为 $O(1)$
- 尾递归
 - 为尾递归

3.3.4 方法核心

- 把容器元素比较、排列，再进一步在小部分里比较、排列

3.3.5 代码模板

```
[6]: # 讨论 if i < j 和 i += 1 的作用:
# 仅不满足 x < arr[j] 时:
#     if 一定成立 - 有无都正常
#     要寻找比 x 大的数, 而 i += 1 存在时, 直接跳过 arr[i], 这个等于 arr[j], 且明显小于
#     x 的数
#     而 i += 1 不存在时, 重复 i += 1, 再进行之后流程
#     故有无 i += 1 都正常
# 仅不满足 i < j 时, 即 i = j 了, ij 重合:
#     if 存在, i += 1 存在
#         arr[i] = arr[j] 和 i += 1 都不运行, 整个程序运行结束, x 赋给 arr[i]
#     if 存在, i += 1 不存在
#         arr[i] = arr[j] 不运行, 整个程序运行结束, x 赋给 arr[i]
#     if 不存在, i += 1 存在
#         arr[j] 赋给 arr[i], i = j, 即 arr[i] 不变, i + 1,
```

```

# 而下面的又进行了 arr[j] 赋给 arr[i], 整个程序结束, x 赋给 arr[i+1]
# 出错
# if 不存在, i += 1 不存在
# arr[j] 赋给 arr[i], i = j, 即 arr[i] 不变, 整个程序结束, x 赋给 arr[i]
#
#
# 故正确的是:
# if 存在, i += 1 存在
# if 存在, i += 1 不存在
# if 不存在, i += 1 不存在

```

'''

QuickSort 代码模板

思路: 挖坑填数, 从大规模到小规模排序, 参见 <https://blog.csdn.net/morewindows/article/details/6684558>

方法:

取一个数作为基准数

划分区间过程, 把数列中大于基准数的数放到左区间, 把数列中小于基准数的数放在右区间再对左右区间重复第二步, 直到左右区间只有一个数

choose one value of nums as the pivot

put all smaller elem to the left of the pivot and larger ones to the right

repeat this process on the left and right side until left/right side one

→ has one value

易错点:

由于 *nums* 有重复数值, 故使用 *pivot < nums[j]*, *nums[i] <= pivot*, 其中使用 “=” 来覆盖所有大小关系, 如没有重复数值, = 可以去掉

R = nums[i + 1: r + 1], 因为跳过 *i*, 故从 *i+1* 开始, 而不是 *i* 开始

nums[l:i] = self.quickSort(L), 因为输入的 *L* 和输出都是相当于一个与 *nums* 无关的值, 故要把返回值与 *nums* 联系起来

'''

class Sort:

def quicksort(**self**, nums):

if len(nums) == 0 **or** len(nums) == 1:

return nums

```

l = 0
r = len(nums) - 1

i = l
j = r
pivot = nums[i]
while i < j:
    while i < j and pivot <= nums[j]:
        j -= 1
    nums[i] = nums[j]
    while i < j and nums[i] < pivot:
        i += 1
    nums[j] = nums[i]
nums[i] = pivot
L = nums[l:i]
R = nums[i + 1: r + 1]
nums[l:i] = self.quicksort(L) # 产生了一部分，就要把这一部分赋值回去，不然不起作用的
nums[i + 1: r + 1] = self.quicksort(R)
return nums

x = Sort()
print(x.quicksort([5,1,1,2,0,0]))

```

[0, 0, 1, 1, 2, 5]

3.3.6 应用

总结

- 分两类，并使用常数空间的题目。e.g. 奇偶排序，稳定空间正负排序
- 数组的排序

奇偶排序

- 905. Sort Array By Parity / <https://www.geeksforgeeks.org/segregate-even-odd-numbers-set->

稳定空间，正负排序

- <https://www.geeksforgeeks.org/move-negative-numbers-beginning-positive-end-constant-extra-space/>

迭代法实现快排 <https://www.geeksforgeeks.org/iterative-quick-sort/>

3-Way 快排

链表排序

3.4 气泡排序 Bubble sort

3.4.1 定义

- 基于比较的排序算法 A comparison based sorting algorithm
- 从头至尾, 依次比较 $arr[i]$ 和 $arr[i+1]$, 将较小的元素向左移动 It iterates left to right comparing every couplet, moving the smaller element to the left.
- 重复此过程, 直到不再将元素移到左侧 It repeats this process until it no longer moves an element to the left.

3.4.2 重点

- 尽管实现起来非常简单, 但在这三种排序方法中效率最低。 While it is very simple to implement, it is the least efficient of these three sorting methods.
- 知道它向右移动一个空间, 一次比较两个元素, 然后向左缩小将要比较的空间。 Know that it moves one space to the right comparing two elements at a time and moving the smaller on to left.

3.4.3 复杂度及属性

- Time
 - Best case: $O(N)$
 - Average case: $O(N^2)$
 - Worst case: $O(N^2)$
- Space
 - $O(1)$
- 稳定性
 - 稳定: 在排序过程中, 元素两两交换时, 相同元素的前后顺序并没有改变

- 原地性 In-place
 - 属于原地算法（直接修改输入数据而不是将输入数据复制一份处理之后再覆盖回去）
 - 使用的空间为 $O(1)$

3.4.4 代码模板

```
[2]: class Solution:
    def bubbleSort(self, arr):
        if len(arr) <= 1:
            return arr

        for i in range(len(arr)): # offer index to scale the size for sort
            for j in range(len(arr) - i - 1):
                if arr[j] > arr[j+1]:
                    arr[j], arr[j+1] = arr[j+1], arr[j] # smaller is ahead of
                    ↪ bigger

        return arr

x = Solution()
print(x.bubbleSort([9, 8, 7, 6, 5, 4, 3, 2, 1]))
```

[1, 2, 3, 4, 5, 6, 7, 8, 9]

3.5 Heap sort 堆排序

3.6 插入排序

<https://ms2008.github.io/2017/03/23/insertion-sort/>

3.7 选择排序

<https://ms2008.github.io/2017/03/19/selection-sort/>

4 非比较基础排序 not comparison-based sort

4.1 计数排序 Counting Sort

4.1.1 基本步骤

- 找出待排序的数组中最大和最小的元素

- 统计数组中每个值为 i 的元素出现的次数，存入数组 C 的第 i 项
- 对所有的计数累加（从 C 中的第一个元素开始，每一项和前一项相加）
- 反向填充目标数组：将每个元素 i 放在新数组的第 $C(i)$ 项，每放一个元素就将 $C(i)$ 减去 1

4.1.2 特点

- 只适非负整数排序, 数比较小
- Time complexity: $O(n)$, n is numbers of value
- Space complexity: $O(k)$ k is kinds of value - 线性时间排序

4.2 桶排序 bucket sort

4.2.1 基本步骤

- 根据元素的值，把元素分散到各个桶里（每个桶能够容纳一定范围内的值）
- 挨个把每个桶里的元素进行内部排序
- 再把元素输出

4.2.2 特点

- Time complexity: $O(n+k)$, n 是元素个数， k 是桶个数
- 桶可以是树 tree

4.3 基数排序 radix sort

4.3.1 基本步骤

- 从低位到高位，轮流比较每个值的相同位数，没有位的补零，即同时比较几个数的个位，排序，再比较十位
- 每个位数的排序是稳定的

—

4.4 希尔排序 shell sort

[]:

5 排序总结

- In exceptional cases insertion-sort or radix-sort are much better than the generic quicksort / mergesort / heapsort answers.

6 基本算法类型 Basic Types of Algorithm

6.1 递归算法 Recursive Algorithms

6.1.1 基本概念

定义

- 在其定义中, 调用自身的算法。An algorithm that calls itself in its definition.
 - 基本情况时, 条件语句用于中断递归 **Base case** a conditional statement that is used to break the recursion.
 - 递归情况时, 条件语句用于触发递归 **Recursive case** a conditional statement that is used to trigger the recursion.

重点

- 堆栈级别太深/堆栈溢出 **Stack level too deep and stack overflow**
 - 如果从递归算法中看到这两种情况, 那么就搞砸了。
 - 这意味着基本情况永远不会被触发, 因为递归有缺陷, 或者实例/问题如此之大, 以至于耗尽了所有内存
 - 正确是使用递归的必要条件: 是否会遇到基本条件
 - 经常在 DFS 中出现

6.1.2 回溯法 Backtracking (DFS 思想)

概念

- 回溯法又称试探法, 当探索到某一步时, 发现原先的选择达不到目标, 就退回一步重新选择, 这种走不通就退回再走的技术为回溯法。
- 回溯法是一种选优搜索法, 按选优条件向前搜索, 以达到目标。但当探索到某一步时, 发现原先选择并不优或达不到目标, 就退回一步重新选择, 这种走不通就退回再走的技术为回溯法, 而满足回溯条件的某个状态的点称为“回溯点”

基本思想

- 在包含问题的所有解的解空间树中, 按照深度优先搜索的策略, 从根节点出发深度探索解空间树。当探索到某一结点时, 要先判断该节点是否包含问题的解, 如果包含, 就从该结点出发继续探索下去, 如果该节点不包含问题的解, 则逐层向其祖先节点回溯。(其实回溯法就是对隐式图的深度优先搜索算法)
- 若用回溯法求问题的所有解时, 要回溯到根, 且根节点的所有可行的子树都要被搜索一遍才结束。

- 若使用回溯法求任一个解时，只要搜索到问题的一个解就可以结束。

基本步骤

1. 针对所给问题，确定问题的解空间：首先应明确定义问题的解空间，问题的解空间至少包含问题的一个最优解
- 确定结点的扩展搜索规则
 - 以 深度优先 方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

6.1.3 分治法 Divide-conquer algorithm

基本概念 将一个规模为 N 的问题分解为 K 个规模较小的子问题，这些子问题相互独立，且与原问题性质相同。求出子问题的解后进行合并，就可以得到原问题的解。

适用情况的特征

1. 该问题的规模缩小到一定的程度就可以容易的解决
- e.g. 排序规模越小，越容易解决
 - 可以分解为若干个规模较小的相同问题，即各子问题具有最优解，就能求出整个问题的最优解
 - 应用分治法的前提，也反映了递归思想的应用
 - e.g. 归并排序把整个分成小部分
 - 利用该问题分解出的子问题的解可以通过某种固定方式合并为该问题的解
 - 确定使用分治的判断条件，若不成立，则考虑贪心或 DP
 - e.g. 归并排序，两个有序子序列再次合并
 - 该问题分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题
 - 涉及到分治法的效率，如果子问题不独立，则分治法需要做许多不必要的工作，重复解公共子问题。
 - 若不满足，则用 DP 比较好
 - e.g. DP 抢劫问题， $dp[i] = \max(dp[i-2] + arr[i], dp[i-1])$ ，故子问题不独立，用 DP 比较好

一般步骤

- 分解：将要解决的问题划分成若干规模较小的同类问题
- 求解：当问题划分得足够小时，用较简单的方法解决，否则递归地解各个子问题
- 合并：按原问题的要求，将子问题逐层合并构成原问题的解

经典问题

1. 二分搜索

- 大数整除法
- Strassen 矩阵乘法
- 棋盘覆盖
- 归并排序
- 快速排序
- 线性时间选择
- 最接近点对问题
- 汉诺塔

6.1.4 回溯法和分治法的区别

- 思想上：
 - 回溯：线性递进思想，一步一步走向更多的实例，比如从 list 的 index=0 到 index=end
 - 分治：从整体到局部思想。
- 实现上：
 - 回溯：
 - * 结束条件-> $i = \text{len}(s)$, 即遍历到最后一个数
 - 分治：
 - * 结束条件-> $\text{len}(\text{arr}) == 0/1$, 即划分的部分为 1 个元素或者无元素
- 应用范围：
 - 回溯：找各种可能性
 - 分治：排序

6.2 迭代算法

6.2.1 定义

- 一种算法，被重复且次数有限的调用，每次都是一次迭代。An algorithm is called repeatedly but for a finite number of times, each time being a single iteration.
 - 通常用于数据集的增量移动 move incrementally through a data set.

6.3 贪心算法 Greedy Algorithm

6.3.1 定义

- An algorithm that, while executing, selects only the information that meets a certain criteria.
- The general five components, from Wiki:
 - A candidate set, from which a solution created.
 - A selection function, which chooses the best candidate to be added to the solution.

- A feasibility function(可行性), that is used to determine if a candidate can be used to contribute to a solution.
- An objective function, which will assign a value to a solution, or a partial solution.
- A solution function, which will indicate when we have discovered a complete solution.
- Key

6.3.2 基本概念

- 在对问题求解时，总是做出在当前看来最好的选择。即不从整体上最优加以考虑，它所做出的仅是在某种意义上的局部最优解。
- 贪心算法只对部分问题才能得到整体最优解，选择的贪心策略必须具备无后效性。
 - 无后效性：状态 受前不受后

6.3.3 适用前提

- 使用局部最优策略能产生全局最优解的问题
- 实际上，贪心算法适用的情况很少。针对问题时，可以先选择该问题下的几个实际数据进行分析，就可以做出判断。

6.3.4 基本步骤

1. 建立数学模型来描述问题（找规律？）
 - 把求解的问题分成若干个子问题（减少问题规模，设想只有 1 个，只有 2 个的情况）
 - 对每一个子问题求解，得到子问题的局部最优解
 - 把子问题的局部最优解合成原来解问题的一个解

6.3.5 实现框架

```
class Solution: # 以纸币问题举例
* def Greedy(self, coin, target): * 寻找最大的可用纸币 # 从问题的某一初始解出发

    * while money < target: # 朝给定总目标前进一步
        * 求次大纸币可以用几张 # 利用可行的决策，求出可行解的一个解元素
    * 统计所有面值的纸币的数量和，返回 # 由所有解元素合成问题的一个可行解
```

6.4 动态规划 dynamic plannig

6.4.1 基本概念

将一个问题分解成若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

6.4.2 与分治法的差别

适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

6.4.3 适用情况的特征

需要满足如下三个特征： 1. 最优化原理：问题的最优解所包含的子问题的解也是最优的（称该问题具有最优子结构）。* 贪心，分治也满足 * 无后效性：状态 受前不受后 * 贪心满足，分治无关 * 有重叠子问题：子问题之间不独立，一个子问题在下一个阶段决策中可能被多次使用到。* 不是适用的必要条件，但若不具有，则 DP 不具有优势。 * 贪心没有子问题，分治法各个子问题独立

6.4.4 基本步骤

设计步骤 初始状态 -> 决策 1 -> 决策 2 -> ... -> 决策 n -> 结束状态

1. 划分阶段：按照原问题的时间或空间特征，把原问题分为若干个阶段（子问题）。在划分阶段时，注意划分后的阶段（子问题）一定是要有序的或者是可排序的，否则问题就无法求解。* e.g. 原问题为求 n 阶台阶的所有走法的数量，子问题是求 1 阶、2 阶台阶、...、n-1 阶台阶的走法
2. 确定状态和状态变量：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。* 无后效性：如果在某个阶段上过程的状态已知，则从此阶段以后过程的发展变化仅与此阶段的状态有关，而与过程在此阶段以前的阶段所经历过的状态无关。* e.g. 第 i 个状态即为 i 阶台阶的所有走法数量。
3. 确定决策并写出状态转移方程：根据相邻两个阶段的状态之间的关系来确定决策方法和状态转移方程。* e.g. $dp[i] = dp[i-1] + dp[i-2]$ ($i \geq 3$)
4. 寻找边界条件：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件* e.g. 1 阶台阶与 2 阶台阶的走法。1 阶台阶有一种走法，2 阶台阶有两种走法，即 $dp[1] = 1, dp[2] = 2$

简化步骤

1. 分析最优解的性质，并刻画其结构特征

- e.g. 抢劫问题， $dp[i]$ 的最优解是有两个结构构成的， $dp[i-1]$ 和 $dp[i-2] + arr[i]$
- 递归的定义最优解（迭代也行？）
- e.g. `for i in range(len(arr))`

- 以自底而上，或自顶而下的记忆化方式，计算出最优值
- 自底而上, 即 DP 法: 建立一个表格, 从最小的子问题开始向上求解较大的子问题, 把每一个子问题的解全部填入表格中, 直到表格中出现原问题的解为止
- 自顶而下的记忆化方式, 即备忘录法: 将递归遇到的子问题的解保存在一个表中, 以便下一个递归遇到同样的子问题时快速求解。
- 当一个问题所有子问题都至少要解一次时, 用动态规划比用备忘录方法要好, 此时, 动态规划算法没有任何多余的计算。当子问题空间中的部分问题可不必要求解时, 用备忘录方法较有利, 因为从其控制结构可以看出, 该方法只解那些确实需要求解的子问题。
- e.g. 迭代 for 完全, 求出最优值
- 根据计算最优值时得到的信息, 构造问题的最优解
- e.g. $dp[\text{len}(\text{arr})-1]$ 的最优值就是问题的最优解?

动态规划三要素及决策表

1. 问题的阶段 (原问题和子问题)

- 每个阶段的状态 (第 i 个状态代表什么)
- 从前一个阶段到后一个阶段之间的递推关系 (转移方程)
- 递推关系必须是从次小的问题开始到较大的问题之间的转化
- 整个求解过程就可以用一个最优决策表来描述, 最优决策表是一个二维表, 其中行表示决策的阶段, 列表示问题状态, 表格需要填写的数据一般对应此问题的在某个阶段某个状态下的最优值 (如最短路径, 最长公共子序列, 最大价值等), 填表的过程就是根据递推关系, 从 1 行 1 列开始, 以行或者列优先的顺序, 依次填写表格, 最后根据整个表格的数据通过简单的取舍或者运算求得问题的最优解。 $f(n,m)=\max\{f(n-1,m), f(n-1,m-w[n])+P(n,m)\}$

6.4.5 基本框架

```
[ ]: class Solution: # 以楼梯问题举例
    def findbest(self, n):
        dp = [0]*(n + 1) # 建立空的 dp 阵

        for j in range(1, m): # 第一个阶段, e.g. 第 1, 第 2 阶的上法
            dp[j] = 初始值

        for i in range(3,n): # 转移方程的转化
            dp[i] = dp[i-1] + dp[i-2]

        return dp[n]
```

6.5 分支限界法（BFS 思想）

6.5.1 基本概念

- 分支限界法的求解目标：找出满足约束条件的一个解，或者在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解
 - 回溯法的求解目标：找出 T 中满足约束条件的所有解

6.5.2 分支搜索算法

- 所谓‘分支’就是采用 广度优先 BFS 的策略，依次搜索 E-结点的所有分支，也就是所有相邻节点，抛弃不满足约束条件的结点，其余结点加入活节点表。
- 然后从表中选择一个结点作为下一个 E-结点，继续搜索。选择下一个 E-结点的方式不同，则会有几种不同的分支搜索方式

1. FIFO 搜索：即先进先出搜索，用队列 deque 实现

- LIFO 搜索：即后进先出搜索，用栈（列表）list 实现
- 优先队列式搜索：即最小值一直在顶端，用优先队列（堆）priority queue/heap 实现

6.5.3 分支限界搜索算法

6.5.4 分支限界法（BFS）和回溯法（DFS）的一些区别

- 方法对解空间树的搜索方式
- 分支限界法：BFS Breath First Search
- 回溯法：DFS Depth First Search
- 存储结点的常用数据结构
- 分支限界法：deque
- 回溯法：list
- 结点存储特性
- 分支限界法：每个节点只有一次被遍历的机会，然后找出满足约束条件的一个解或者特定意义下的最优解
- 回溯法：所有可行子节点被遍历，放入 list 中，进行筛选
- 常用应用
- 分支限界法：
- 回溯法：

6.5.5 分支限界法（BFS）和回溯法（DFS）的一些区别

- 方法对解空间树的搜索方式
- 分支限界法：BFS Breath First Search

- 回溯法：DFS Depth First Search
- 存储结点的常用数据结构
- 分支限界法：deque
- 回溯法：list
- 结点存储特性
- 分支限界法：每个节点只有一次被遍历的机会，然后找出满足约束条件的一个解或者特定意义下的最优解
- 回溯法：所有可行子节点被遍历，放入 list 中，进行筛选
- 常用应用
- 分支限界法：
- 回溯法：

6.5.6 基本框架

```
[1]: class Solution: # 以楼梯问题举例
    def findbest(self, n):
        dp = [0]*(n + 1) # 建立空的 dp 阵

        for j in range(1, m): # 第一个阶段, e.g. 第 1, 第 2 阶的上法
            dp[j] = 初始值

        for i in range(3,n): # 转移方程的转化
            dp[i] = dp[i-1] + dp[i-2]

        return dp[n]
```

6.6 分支限界法（BFS 思想）

6.6.1 基本概念

- 分支限界法的求解目标：找出满足约束条件的一个解，或者在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解
 - 回溯法的求解目标：找出 T 中满足约束条件的所有解

6.6.2 分支搜索算法

- 所谓‘分支’就是采用 广度优先 BFS 的策略，依次搜索 E-结点的所有分支，也就是所有相邻节点，抛弃不满足约束条件的结点，其余结点加入活节点表。

- 然后从表中选择一个结点作为下一个 E-结点，继续搜索。选择下一个 E-结点的方式不同，则会有几种不同的分支搜索方式

1. FIFO 搜索：即先进先出搜索，用队列 deque 实现

- LIFO 搜索：即后进先出搜索，用栈（列表）list 实现
- 优先队列式搜索：即最小值一直在顶端，用优先队列（堆）priority queue/heap 实现

6.6.3 分支限界搜索算法

6.6.4 分支限界法（BFS）和回溯法（DFS）的一些区别

- 方法对解空间树的搜索方式
- 分支限界法：BFS Breath First Search
- 回溯法：DFS Depth First Search
- 存储结点的常用数据结构
- 分支限界法：deque
- 回溯法：list
- 结点存储特性
- 分支限界法：每个节点只有一次被遍历的机会，然后找出满足约束条件的一个解或者特定意义下的最优解
- 回溯法：所有可行子节点被遍历，放入 list 中，进行筛选
- 常用应用
- 分支限界法：
- 回溯法：

7 Google 面试官方指南

7.1 电面

- 与潜在的同僚或上司面，即专业面试
- 持续 30-60mins
- 使用 Google Doc, 并建议使用扩音器或者耳机
- 面试内容是数据结构和算法
- 开放式问题，我来明确问题，设计要求
- 用一种算法来解释这个问题
- 写代码.
- 能写多少写多少，先写大框架，然后再完善？(write what comes and but then refine it later)
- 需要考虑 corner case(和 edge case, 二者不同)

- 考虑生产就绪 (production ready) -> 很稳定, 可维护的 (用类?), 可扩展的 (用类?), 有记录 (注释?)
- 优化代码, 然后测试并寻找 bugs -> 没有 follow up 题目?

7.2 现场面试

7.2.1 BQ

- 见 BQ 总结

7.2.2 写代码

内容

- 代码技巧
- 科技领域的专业知识
- 关于工具的知识, 关于编程语言的知识 -> ML 框架如何使用? Python 语言的特性?!!!
- 会进行来回讨论, 而不是固定某一种想法 -> 交流思路, 不仅听面试官的思路, 也阐述自己的!!!
- 会深度的讨论我的方法 -> 思考问题的时候, 要深入, 找到最优的办法 !!!

过程

- 整个过程中, 都要无拘束的问面试官问题, 来保证完全理解了他们的问题!!!
- 我们问问题时, 也可以自由发问。内容可以包括: 团队, 文化等。这样利于判断这个工作是否适合我。!!!

7.3 BQ 准备方法

- 可以预测百分之九十的问题, Google 'most common interview question', 并列出前 20 个问题!!!
 - 参考 27 Most Common Job Interview Questions and Answers _ Inc.com.pdf
- 写答案并熟记: 针对列出的每个问题, 写出答案, 并熟记, 能够快速的答案出来。!!!
- 有备份方案: 针对每个问题, 写出三个答案!!!
- 针对一个问题, 需要不同的, 同样优质的答案, 这个可以保证第一个面试官不喜欢的时候, 第二个可以听到不同的答案。
- 针对 coding – 充分的解释: 为了让他们理解我的思维过程, 故解释我的思路和决定。 – 针对 coding
- Google 不仅评价技术能力, 而且评价我们思考问题和试图解决问题的能力。!!
- 要清晰的陈述和确认问题的假设, 去明确问题是合理的 -> 即确认问题的 corner case 等
- 事实支持: 每一个问题的回答都要有故事/具体做了什么来进行论证。

- 针对 coding – 细化并确认问题。
- 大部分的问题是开放式的，目的是了解我们对问题的类别和信息了解多少-> 熟悉 DS 和算法类别，熟悉题型。
- 考察思考问题和解决问题的主要方法 -> 把思考问题的过程说出来 (idea 和 method 中)，并大胆的去问考官来确认问题!!!
- 针对 coding – 改进答案过程
- 先 brute force, 后优化方法，并且解释在做什么和为什么这么做-> 做什么: method, 为什么: 降低 time/space complexity !!!
- 大声思考对问题的最初想法 -> 考察对 DS 和算法的掌握, e.g. 某某是一类问题，通常用...来解决
- 练习: 大声练习面试答案，直到您可以清楚简洁地讲出每个故事

7.4 coding 的准备方法

7.4.1 Coding 练习

- 问 HR 用白板还是 Chromebook
- 不要太看重小的语法问题，告诉面试官即可

7.4.2 Coding

- 起码掌握一种编程语言
- 知道 APIs, OOD(Object Orientated Design), 编程。知道如何测试 code, 想出合适的 corner case (包含 edge case)!!

7.4.3 算法

- 整体思路: 从上到下，或者从下到上 -> 进一步明确这个概念
 - 自上而下: 递归，分治等
 - 自下而上: DP 等
- 知道复杂度，并知道如何优化，改变 (变的更大? 同级改变)
- 类别
- 排序，搜索，二分搜索 sorting (plus searching, and binary search)
- 分治法 divide-and-conquer
- 动态规划/记忆，简而言之就是动态规划
 - Memoization is a term describing an optimization technique where you cache previously computed results, and return the cached result when the same computation is needed again.

- Dynamic programming is a technique for solving problems of recursive nature, iteratively and is applicable when the computations of the subproblems overlap.
- Dynamic programming is typically implemented using tabulation, but can also be implemented using memoization. So as you can see, neither one is a “subset” of the other.
- 贪心
- 递归
- 有关特殊数据结构的算法
 - 比如链表的特殊算法, slow-fast pointer 法等
- 了解 Big-O 表示法, 并准备讨论复杂的算法, 比如 Dijkstra 和 A* search algorithm。-> 知道这两个算法的都是查找最短路径的, 工作原理, 复杂度。不用应用。!!!
- 在编写代码之前讨论或概述要考虑的算法 -> 与 mock 的过程一致

7.4.4 排序

- 熟悉常见的排序函数和其擅长的数据输入类型!!!
- 了解每种排序方法的时间/空间复杂度, e.g. 例如, 在特殊情况下, 插入排序或基数排序要比通用的 QuickSort / MergeSort / HeapSort 答案好得多。

7.4.5 数据结构

- 掌握尽可能多的数据结构, 最常用的如下
- list
- 链表
- stacks 堆栈
- queue 队列
- 哈希集, 哈希图, 哈希表, 字典
- tree/ binary tree
- heap
- graph 图
- 您应该了解内在的数据结构, 以及每种数据结构倾向于使用哪些算法 !!! -> 了解每种数据结构增删查并改的复杂度, 和每种数据结构倾向于使用哪种算法。

7.4.6 数学 !!!

- 一些面试官会问离散数学问题 basic discrete math
- 这个在谷歌很流行, 因为 counting problem, probability, and other Discrete Math 101 situation surround us.
- 在面试之前, 花一些时间学习 the essentials of elementary probability theory and combinatorics.

- 熟悉 n-choose-k problems and their ilk. 熟悉 n-choose-k 问题及同类的问题。

7.4.7 图!!!

- 考虑一个问题是否可以用图算法解决
- 例如 距离问题, 搜索问题, 联通性问题, 环探测问题。 distance, search, connectivity, cycle-detection, etc
- 表示一个图有三种方法, objects and pointers, matrix, adjacency list 对象和指针, 矩阵, 邻接表
- 熟悉每种的表示方法 each representation
- 熟悉每种的优缺点 its pros and cons
- 知道基本的图遍历方法:DFS 和 BFS
- 知道计算复杂度, 及时空权衡 computational complexity, their tradeoffs
- 可以降低算法的时间复杂度以换取更大的空间, 或减少消耗的空间以换取较慢的执行
- 知道如何在代码中实现

7.4.8 递归

- 很多问题涉及到递归思想和潜在递归解法
- 使用递归去找到比迭代更优雅的解法。

8 面试准备技巧 Interview Preparation Tips

8.1 成功的关键

- 在谷歌, 我们坚信合作和分享思想。最重要的是: 你需要从面试官那得到更多的信息, 从而全面的分析和回答问题。

8.2 技巧

- 可以问面试官问题
- 当被要求给出解决方案时, 首先定义问题, 及划定范围
- 如果不理解, 请寻求帮助, 或者要求进一步要求细化问题。
- 如果需要假设什么, 需要检查这个假设是否成立!
- 描述解决方案的细节, 问题的每个部分都是如何解决的
- 总是让面试官知道你在想什么, 因为他/她对你的思考过程和解决方案同样会感兴趣。另外, 如果你被困住了, 他们可能会在他们知道你在做什么时提供提示。
- 最后, 注意聆听, 如果你的面试官试图帮助你, 不要错过任何提示!

8.3 技术要求

8.3.1 算法复杂度

- 了解大 O

8.3.2 排序

- 知道如何排序
- 不做泡沫排序
- 知道至少一种 $n \log n$ 排序算法的细节，最好是两种（例如，快速排序和合并排序）。在无法进行快速排序的情况下，合并排序可能非常有用，因此请查看一下。

8.3.3 哈希表系列

- 可以说是人类已知的最重要的单一数据结构。绝对应该知道它们的工作方式。
- 能够在面试中使用哈希表→ 使用 dic，以及 hash() 函数

8.3.4 Tree

- 熟悉二叉树，n 元树和 trie 树
- 熟悉至少一种类型的平衡二叉树，无论是红色/黑色树，八角树还是 AVL 树，并了解其实现方式。
- 了解树遍历算法：BFS 和 DFS，并了解有序，后序和预序之间的区别。

8.3.5 Graph

- 图对 Google 很重要
- 如果有机会，请尝试研究更高级的算法，例如 Dijkstra 和 A*。

8.3.6 其他数据结构

- 您应该特别了解最著名的 NP 完全问题类别，例如旅行推销员和背包问题，并且当面试官变相询问您时能够识别它们。找出 NP-complete 的含义。

8.3.7 数学

- 一些面试官提出基本的离散数学问题。与其他公司相比，这在 Google 中更为普遍

8.3.8 操作系统

- 了解流程，线程和并发问题。
- 了解锁，互斥，信号量和监视器以及它们如何工作。

- 了解死锁和活锁以及如何避免死锁。
- 知道进程需要什么资源，线程需要什么，上下文切换如何工作，以及操作系统和底层硬件如何启动它。
- 了解有关调度的知识。https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm
- 世界正迅速向多核迈进，因此请了解“现代”并发构造的基础。

9 Google 面试准备完全指南 Google Interview Preparation For Software Engineer – A Complete Guide

9.1 重点

- 问题解决和编程技巧（DS 和算法），Googlyness（包含了对技术的热情，好奇心，道德，友善，良好的素质等等）是必备的
- 软件工程师或 SWE-II（L3）是入门级的全职软件工程师。在这个级别上，有 4 或 5 个现场回合，他们可能会提出设计问题，但通常不会。
- 检查 Google 教育团队发布的技能列表，他们期望潜在的工程师
 - <https://techdevguide.withgoogle.com/>
- 面试官受过训练，不会对您的答案做出反应，因此不要期望回答是或否。因此请准备好自己，看看冷酷的面孔。

9.2 让 Google 注意自己

- 单页简历，只包括相关工作。尽可能简短，精确。不要做假。提到项目时，要提到项目的复杂度。
 - <https://www.geeksforgeeks.org/resume-building-resources-and-tips/>
- 可以参与 Google Kickstart or Google Code Jam，如果在 Google Code Jam 中，到达第二轮，可能会被 Google Hr 练习
- 保持 LinkedIn, Github 的更新。
 - <https://www.youtube.com/watch?v=BYUy1yvJHxE&t=69s>

9.3 电面

- 平均是两轮
- 可能会有两题，写完起码一题
- 先 brute force，后优化，并考虑时空复杂度
- 在 Google Doc 上练习，无 indentation, syntax highlight and auto-completion.
- 大声说出自己的想法，吐字清晰（困难点），告诉 idea 和 method

9.4 现场面试

- 4-6 轮，包括 lunch interviews, lunch 不是真正的面试，仅仅是与 Google 的接触
- 每轮持续 45min 到 1hour
- 面试内容两种类型：1. 技术编程/一般分析问题。2.BQ

9.4.1 技术编程/一般分析问题 Technical Coding and General Analysis Question

- 技术编程是指编程问题，考察使用 DS 和算法解决真实问题的能力。期待到达最优解
- 一般分析问题是指数学、设计、基于观点的问题，目的是考察思维过程和作为员工的工作方式
- 建议使用 Cracking the Coding Interview，并在如下网站练习
 - <https://practice.geeksforgeeks.org/explore/?company%5B%5D=Google&page=1>
 - <https://careercup.com/>
- 其余参考《面试准备技巧》

9.4.2 系统设计问题

- 测试您设计和扩展基于技术的系统（例如设计 Gmail, youtube, uber 等）的整体能力。
- 参考 <http://blog.gainlo.co/index.php/category/system-design-interview-questions/>
- 参考 《Gainlo 面试指南》
- L4/L3 OO design question 和 系统设计相关的问题。
- 在 Google 中，可扩展性始终很重要。因此，期望设计问题对于大型系统将是一个模棱两可的现实世界问题。他们正在寻找思考的过程，以及如何分解事物以最终获得过于富有创意和可扩展性的解决方案。

9.5 程序员要求

- “我们不单单寻找解决已知问题的程序员，我们更感兴趣能够解决未知问题的程序员”

9.5.1 重点

- 面试官在接受面试时通常会有一种心态，即如果他/她可以与应聘者一起每天工作，那么请不要在面试时表现出傲慢或自我的迹象。
- 面试官将在实施代码时检查您是否使用了合适的数据结构和算法。
- 面试官将注意到您如何优化解决方案，有关编程语言选择的知识，编码速度，错过的任何极端情况以及如何分析时间和空间复杂性的方法。
- 他们将检查您如何传达您的思维过程以逻辑方式解决特定问题。
- 他们还将检查您是否能够抓住提示并能够继续解决问题。
- 候选人是否愿意接受新想法？候选人在解决方案上是否灵活？

9.5.2 面试评估标准

- 面试后，所有反馈都是从不同的面试官那里收集的，然后您在一系列不同类别中的评分为 1-4，其中包括您的编码经验和分析能力。然后，将此反馈发送给招聘委员会以做出最终决定。

9.5.3 提示

- 在整个面试中学习“大声思考”，否则面试官将不知道您在想什么。向面试官展示您对问题的思考过程以及解决问题所要遵循的方法。
- 通过在纸或白板上编写代码进行练习。它会在面试过程中真正为您提供帮助。
- 永远不要说你做不到。即使存在您之前从未解决过的问题，或者似乎无法解决的问题，请从不同的角度来不断解决问题，面试官会给您提示。但是，如果您说无法解决问题，那将是一个很大的危险信号，并且您最终可能会被拒绝。
- 在跳转到解决方案之前，请务必提出相关问题，以使其更加清晰。在前往解决方案之前，请务必与面试官核对您的假设并清除所有疑问。
- 我们强烈建议您不要编写伪代码来设计代码。在 45 分钟的采访中，您没有时间这样做。

9.5.4 有用的链接

- Google 面试准备题目 <https://www.geeksforgeeks.org/google-interview-preparation/>
- 面试经验 <https://www.geeksforgeeks.org/tag/google/>
- code 练习 <https://www.geeksforgeeks.org/practice-for-cracking-any-coding-interview/>

10 Gainlo 面试指南

10.1 获得面试机会

- 时间点很重要
- 推荐是关键
- 简历：简洁和易懂 - 描述清楚，同时给人们一些复杂的印象

10.2 构建扎实的基础

10.2.1 最短路径

- 把重点放在计算机科学的基础上
- 做长远规划
- 清晰理解基础知识之后，再尝试面试问题

10.2.2 DS 和算法的定义

- 清楚每个基本数据结构和算法的定义

- 理清概念。e.g.DP 与递归之间的区别，stack 栈和 heap 堆之间的区别。
- 实现算法。e.g. 快速排序，用堆实现找 k 个最大的数字，BFS/DFS

10.2.3 优缺点

- 清楚每种 DS 和算法的优缺点，熟悉他们的特征
- 针对每个 DS 和算法，应该问自己：有什么优缺点？我什么时候可以使用它？

10.2.4 复杂度

- 清楚每种 DS 和算法的复杂度
- 清楚优化的路线
 - 最常见的情况是你的解决方案很慢，面试官要求你对其进行优化。
 - 假设你有一个 $O(n^2)$ 的方法，为了让它更快，我们可以这样想：
一般来说，为了提高速度，我们可以选择更好的数据结构/算法，或者使用更多的内存。
如果我们想使它成为 $O(n \log n)$ ，有几种可用的工具：二分搜索，排序，BST 等等。也许你应该尝试对数组排序，hash/dic 散列是一个选项。另外，DP 是优化递归的好方法。

10.2.5 总结

- 我会建议人们逐个查阅所有基本的数据结构/算法。了解这个概念，弄清楚利弊，并自己实现。在这一步完成之前，不要急于练习编程问题。

10.3 练习编程问题

10.3.1 什么时候

- 我必须在这里强调多次：在熟悉基本的数据结构/算法之前开始练习编程问题是没有意义的。
- 这个想法是，你不需要完成你在网上看到的每一个问题，也不需要浏览所有这些在线资源。
80/20 规则说 80% 的结果来自 20% 的原因。我想帮助你确定你需要关注的 20%。

10.3.2 提示 1：编写足够的代码

10.3.3 提示 2：把想法说出来

- 强烈建议把想法说出来，好处是：
 - 展示沟通技巧
 - 如果你不在正确的路线上，面试官更有可能纠正你
 - 这可以帮助你更清楚地理解你的想法，并防止你在想法足够具体之前编写代码

10.3.4 提示 3：控制时间

- 在练习编码问题时，有多少人注意速度？很少

- 当每个人第一次跟踪时间的时候，它们都会对速度有多慢感到惊讶。人类不善于估计时间
- 对于谷歌编程面试，每一个正好 45 分钟。你会打个招呼，并在第一个 5 分钟内介绍自己，最后，你可以再花 5 分钟提问。在剩余的 35 分钟内，你需要完成 2 个编程问题，其中至少有一个需要编写代码
 - 在头脑清晰之后编写代码
 - 在完成 brute force 之后，才进行优化
 - 不要从头造轮子。比如能用 `sort()`，就不用全部写出 `sort` 的代码
- 找出速度慢的瓶颈
 - 提出正确的方法：如果您提供解决方案的速度很慢，则很可能是您没有足够的练习。
 - 编写可靠的代码：如果您发现自己完成编码的速度很慢，那么您应该为每个练习的问题编写可靠的代码。
- 与计时器练习
- 天真的解决方案优先

10.3.5 返回到基础知识

- 不管你多努力，总是会有更多的无法准备的问题。处理正确的问题是成功的关键。
- 一开始，最好涵盖许多不同类型的问题（例如链表，递归，动态规划等）。不过，以后你应该更专注
- 如果你发现自己的弱点是一个特定的领域，例如树问题。通常有两种情况：
 - 你的基础不够牢固。换句话说，你对基本的数据结构/算法没有清晰的认识。在这种情况下，不管你练习了多少个问题，你的问题总是在那里。你应该做的，是回顾你的教科书，并解决根本问题。
 - 有时候，你只需要多练习。去寻找更多相同类型的问题，并把重点放在这个领域一段时间。这是一个比漫无目的的更有效的方法。

10.3.6 总结

- 你应该挑选在线资源，并在实践中聪明一些
- 清楚自己的实力/弱点并做相应的准备。了解你自己而不是漫无目的地工作，这很重要。
- 如果我希望从本章中得到一件事情，那就是确定你的 20% 的努力，并尽可能多地关注它

10.4 资深工程师、应届生和实习生

10.4.1 相似和不同

- 实习生的系统设计通常是可选的。但是每个人都需要数据结构，算法和测试。
 - 数据结构。候选人应该非常熟悉树，哈希表，栈/队列等所有基本数据结构。一个很好的例子是，如果你正在实现一个 BFS，你应该清楚使用哪个数据结构。

- 算法。如果面试官要求你逐层遍历树，你知道使用哪种算法吗？而且，许多人往往忽视的是时间和空间的复杂性。我想说，如果候选人在分析复杂性方面有困难，我很难给他/她一个好的分数
- 测试。确保代码能够正常工作是非常重要的。有些人甚至是 TDD 的铁杆粉丝。在练习编码问题时，总是问自己如何测试你的解决方案？你会提供哪些测试用例？作为一名面试官，我通常会寻找能够覆盖这些角落案例的候选人，并且能够将代码模块化以便于测试。
- 系统设计。这是高级和低级工程师之间的一大区别。设计一个可扩展和健壮的系统并不是一件容易的事情。我们很快就会在整个章节中讨论这个话题

10.4.2 应届生和实习生

- 专注于数据结构/算法：如果你精通基本的数据结构/算法，你至少完成了 80%。在这个领域花尽可能多的时间和精力
- 项目和实习
 - 做一些很酷的项目，那真的会很不一样。如果是课程项目，个人项目或实习，也没有关系
 - 花时间准备你的介绍。找出你想突出显示的项目，并使你的语言清晰易懂。不要以为面试官具有所有的背景。

10.4.3 总结

- 有时候，资深工程师和学生之间的差异完全在于心态。应届生/实习生对面试太害怕了，因为他们认为他们对软件构建一无所知。但是，由于期望低，只需要有一个扎实的计算机科学基础

10.5 系统设计面试

10.6 电面

10.6.1 把它当作现场面试

- 有两种类型的电话面试 - 非技术面试和技术面试。非技术性面试通常由人力资源部门进行。谈话主要是关于你的背景和激情
- 技术面试是本章我们所关注的内容。除了不是面对面的交流之外，你可以期望它和现场面试完全一样。
- 以 Google 电话面试为例。这大概是 45 分钟，通常会问两个问题。你一定会为其中的至少一个编写代码。
- 在电话面试中不太可能有系统设计问题，所以我的建议只是把它当作现场编程面试。
- 如果你认为电话面试技术性较差且较容易，你可能需要调整你的期望。准备好接受非常技术性的电话面试。即使事情变得比预期更容易，这实际上是一件好事。

10.6.2 熟悉 Google Doc/其他代码共享工具

- 搜索 Google 或 Glassdoor，或只是询问招聘人员。你也应该检查是否需要编译。经验法则是去除尽可能多的意想不到的事情
- 像往常一样练习编程问题，但在特定工具上编写代码。有时可能需要一些时间才能适应，例如谷歌面试官喜欢分享 Google Doc，没有缩进，自动不全或高亮显示。
- 细节真的很重要。干净而易读的代码可以真正令面试官感到震惊。

10.6.3 把想法说出来

- 在电话面试中，把想法说出来更为重要。每个人都知道手机屏幕上的沟通不太顺畅，但是很少有人真正做出改进。
- 最大的好处就是减少不必要的误会。如果我知道他在想什么，当事情不正确时，我一定会提供帮助。另外，最终的代码只是我们评估的一部分，讨论过程和候选人如何分析问题同样重要。
- 谈论的几个方面：
 - 谈谈你的整体策略。你想从一些例子开始吗？你是否正在解决问题的一个简单版本？你打算使用动态规划吗？-》Idea + Method
 - 你关心什么？你是否担心使用太多的内存或算法太慢？-> 优化 Time or Space
 - 你将在以后解决什么问题？例如，你知道当前的算法不处理一个特定的情况，提前提到这一点。-> 目前算法的缺点!!!
 - 你卡在什么上了？当你发现自己没有取得进展时，说清楚你的问题。-> 说出自己的困难

10.6.4 沟通技巧

- 慢慢说，说清楚。!!!
 - 当人们紧张时，人们倾向于说得更快，因为这通常是第一次技术性面试。通常，当我难以理解候选人的时候，给我的印象是他脑子里不清楚。只要放慢速度，不要急于求成。这也可以让你听起来更有信心。
- 使用速记来帮助。
 - 这是一个面对面讨论的“技巧”。突出重点。例如，如果你的解决方案包含三个步骤，只需列出概要。实际上，用一些文字来减少不必要的来回讨论，最终可以节省你的时间
- 找一个安静的环境。
 - 你会惊讶于有多少人在恶劣的环境中进行了电话面试。背景噪音，信号差和意外中断是我遇到的三件事情。有时候会让我发疯。
- 每天练习这些技巧

10.6.5 不要作弊

- 面试官能够比预期更容易发现作弊行为

10.6.6 总结

- 尽早练习沟通

10.7 现场面试

10.7.1 现场面试流程

- 总共进行四次技术性面试（通常是上午两个，下午两个）。在午餐时间有一个“午餐面试”，但这不是一个真正的面试，因为你没有被评估，这只是一个与 Google 员工聚会的机会
- 四个编程面试中，至少有一个是系统设计面试
- 面试结束后，每位面试官都会写下一份报告，其中包含了 45 分钟的所有细节和讨论，包括你的代码。该报告可以非常详细地说明，在每个部分花费了多少时间，并且在提示之后你的反应将被包括在内。招聘委员会将根据所有这些报告进行招聘决策

10.7.2 电话面试 VS 现场面试

- 面对面交流意味着整个过程可以更加顺畅，但同时很多人更容易紧张。
- 问题稍微困难一些。

10.7.3 “问题比我想象的要容易得多”

- 总的来说，onsite 只是稍微困难一点。许多面试官仍然会问如 2-sum 的问题，来获得候选人的最初想法
- 谷歌面试中最困难的部分（与其他公司一样）不是提出解决方案。很多人可以在一定程度上解决这个问题。但要获得好成绩，你需要：
 1. 拿出最佳的解决方案
 - 编程干净和无错的代码
 - 在时间限制下完成这两件事情
 - 第二点和第三点是最具挑战性的部分，大多数候选人由于这两个原因而失败。许多人声称问题比他们想象的要容易得多，但他们仍然没有快速地提供无错代码。
 - 从面试官的角度来看，提出没有人能解决的问题是没有意义的。为了评估候选人，我们需要看到分析过程，代码以及他/她能够多快解决问题。
 - 我的任务：
 - 更多关注“基本”问题而不是超级难题。
 - * 只要在 Glassdoor 上对 Uber 面试问题进行一点搜索，就可以了解哪些类型的问题很受欢迎。你很快就会意识到，他们并不像大多数人所想的那样艰难
 - 练习时要特别注意编程。为每个问题写下可靠的代码并跟踪时间。!!!

10.7.4 白板

- 写之前要仔细考虑。事先要有清醒的想法。
- 工整的写作。

10.7.5 沟通

- 整个面试是一个讨论过程，在现场面试中更是如此。
 - 这与考试完全不同，因为你需要与面试官保持沟通。把这个过程当作一个正常的讨论，与你们的同事在一起讨论同样的问题。
- 把想法说出来。谈论你脑海中的任何事情，在讨论之前你不需要有任何具体的想法。→ 有什么说什么，idea + method
- 随意谈谈你卡在了什么地方。
 - 面试官真的很乐意通过给你提示或告诉你你不是在正确的方向来帮助你。不要担心暴露你的弱点，即使你不承认也不会有任何进展。相反，放宽心态，面试过程将会更加愉快。

10.7.6 总结

- 面试前你一定需要很多练习

10.8 非技术问题

- 一般来说，非技术性问题只占面试的一小部分。然而，如果面试官抓住任何危险信号，它可以“杀掉”你。一个例子是文化适应。

10.8.1 面试官在寻找什么

- 总体印象
 - 我想和这个人一起工作吗？这确实是一个非常主观的问题，但通过与候选人交谈和讨论，面试官可以得到一些想法。
- 文化适应
 - 每个公司都有自己的文化，评估候选人是否合适是非常重要的。每种文化都有其优点和缺点，因此都是关于匹配（就像约会一样）。
- 加分和危险信号
 - 编程问题永远不能涵盖一个人的每一个方面。通过讨论过去的经验，候选人感兴趣的领域，面试官实际上是寻找任何加分或危险信号。

10.8.2 自我介绍

- “介绍你自己”可能是唯一确定的问题，没有理由不准备。

- 在好的和不好的介绍之间没有明确的界限，但事先做好一些准备工作肯定会对你有好处。事实上，这不会占用你太多的时间。
- 黄金法则是清晰简明。
 - 首先，列出你真正想要在你的介绍中突出显示的几点（少于三点）。这可能是你过去的项目之一，或者你赢得的一些编程比赛。其次，在 1-2 分钟内压缩所有这些
 - 不需要覆盖尽可能多的细节。如果面试官想知道更多，他们会问。一个常见的错误是过度推销自己。长时间的介绍不仅会让面试官厌烦，而且会缩短你的编程问题的时间
- 常见的错误是我所说的“太模糊”。
 - 往往不是因为候选人沟通不好，而是因为介绍有很多细节，没有相关的背景的人不能理解。找到一个对你的工作不太了解的工程师朋友，测试他/她是否能理解你的介绍。
- 相关性。
 - 强烈建议对每个公司有不同介绍。你想表明你能胜任这个职位，因为你有相关的经验。这绝对不是必需的，但是如果你有的话会是一个加分。如果你以前的所有项目都是完全不相关的（例如，你在硬件上工作，但是在寻找一个软件的职位），那么就把它保持简洁。

10.8.3 文化适应

- 每个公司，无论多大，都有一定的规范和信念，每个人都遵循。这不是一个强迫人们以同样的方式行事的具体规则，也不是人们从不行动的口号。这就是为什么文化是定义它的最好的术语
 - Facebook 的“行动迅速和打破常规”。核心的信念是尽早测试，来避免过度工程。你可以通过 Google 深入了解这一点。
- 关键在于面试官有责任评估候选人是否真正符合这种文化。这并不容易，但从所有的讨论中，人们仍然可以得到一些想法。
 - 例如，在系统设计面试中，面试官肯定知道你是否倾向于过度工程
 - Airbnb。通常情况下，你将有一个完全集中于文化适应的简短面试。你会被问到你的产品经验，为什么你想加入 Airbnb，你也可以谈论你的职业目标。如果你不是真的相信 Airbnb 的使命，只是瞄准加薪，我怀疑你是否会得到录用。

10.8.4 双向选择

- 你不需要适应文化，你应该评估文化
 - 最大的误解是认为面试是考试。但是，我认为这是一个双向的选择。面试过程中，考生应同时对面试官和公司进行评估。在与公司工作的人交谈之后，你可能会有完全不同的印象。
- 在文化适应方面，我认为更好的解释是评估公司是否真的适合你。
 - 如果你不相信它的价值，你不需要显着地调整自己来适应公司。但是，你至少应该调查和了解它的文化。

- 很多人不了解他们正在面试的公司。
 - 对我来说，就像你在和一个女孩约会而不知道她的背景。这就是“为什么加入我们”应该是一个非常简单而直接的问题。如果你知道公司是否适合你，你应该可以在几秒钟内回答。如果你不相信公司的使命，也许你本来不应该申请。

10.8.5 给面试官的问题

- 建议问一下。好处是你会更加了解面试官和公司。而且，向公司展示你的兴趣总是一个好兆头
- 不要认为这个过程是增加机会的一种方式，尽管可能是。一个更好的方法就是询问你脑海中关于公司的任何问题，而且把其他事情考虑好
- 不要问你的面试表现/结果。

10.8.6 总结

- 尽管非技术性问题可能成为面试的重要部分，但事实是，大多数人往往会过度担心。
- 大多数时候，你只需要做你自己。如果他们觉得你在文化方面不太合适，那可能不是什么坏事，因为它比加入公司好得多，一切都不像你想象的那样。
- 在面试之前，你绝对可以准备好非技术性的问题，但是要专注于技术性问题，这是面试的核心

10.9 非谷歌的面试

10.9.1 惊人的相似

- “Google 风格面试”是迄今为止的最佳解决方案

10.9.2 扎实的基础

- 具有类似的面试流程意味着扎实的基础比以往更重要。
- 完全错误的策略是在第一天跳进编程问题。
- 另一方面，如果你从一开始就把注意力集中在基础上，无论你正在准备哪家公司，你都会处于有利地位。这种复合效应不能再重要

10.9.3 持之以恒

- 尽早开始准备，并把整个过程当做自我提升而不是通过测试的一种方法。

10.9.4 公司特定的准备

- 公司可能仍然有一些特定类型的面试。一个例子就是亚马逊在第一轮面试中通常会有一个编程测试。我的建议很简单- 不要太担心。

- 我建议人们仍然按照他们的初步准备计划。总是需要着重于基础和练习编程问题。如果没有掌握基本的数据结构和算法，你将无法通过在线编程测试。
- 在面试之前（也许提前一周），你可以做一些公司特定的准备。强烈建议使用 Glassdoor，你可以找到目标公司最近提出的大量问题。换句话说，你应该在准备的最后一步做这个，或多或少你会发现公司之间的差异。

10.9.5 总结

- 一定是专注于“基础”。大多数人往往目光短浅，大部分时间花在“细枝末节”上。如果没有扎实的基础，你或许能够偶然地解决一个问题，但是你永远无法解决所有问题