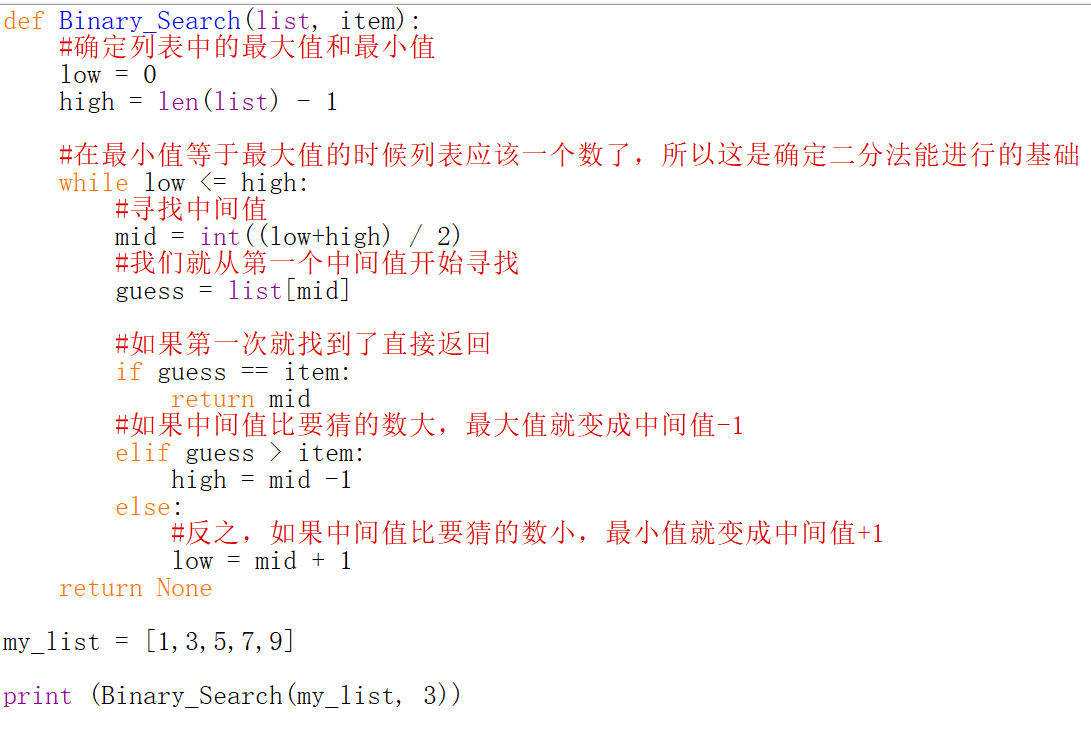
第一章：算法介绍

1. 二分法的复杂度为O(log n),是最快的

2. 代码实现：



第二章：选择排序

1. 计算机内存就像很多抽屉的柜子，每个柜子有个地址

2. 当需要往柜子里存一堆数据的时候，到底是用array还是链表

3. 用array的话，所有数据都是相邻并且挨着的，这样就很难空出一大块地址给要存的数据，并且如果要添加数据，则整个array都得移动，但是好处在于可以一次性访问整个array

4. 用list的话，前一个数据指向后一个数据，不用占据那么多的内存，节省空间，互相不挨着，知道前一个就能得到下一个的地址，但是查找时很不方便，如果要找后面一点的元素则必须把前面所以元素都先过一遍。

5. Array支持快速读取，链表则支持快速插入和删除

6. 一个Array中所有元素都必须是同一个类型

7. 选择排序的原理是先建立一个新的空列表，然后在老列表中找到最大或最小值（取决于我们是要升序还是降序），然后把找到的值放入那个新列表，然后再回到老列表继续找最值，再放入新列表，如此循环，复杂度为O(n^2),因为需要把两个列表都过一遍

8. 代码实现



第三章：递归

1. 当写一个递归function的时候，一定要让他知道什么时候停止，所以每个递归function都有两个部分，base case和recursive case，base case会设定什么时候就不调用自己了

2. stack就两个操作，进栈和出栈（push和pop）

第四章：快速排序

1. 首先快排是分治法中的一种，首先建立一个simple case as the base case，然后简化当前的问题到base case的程度

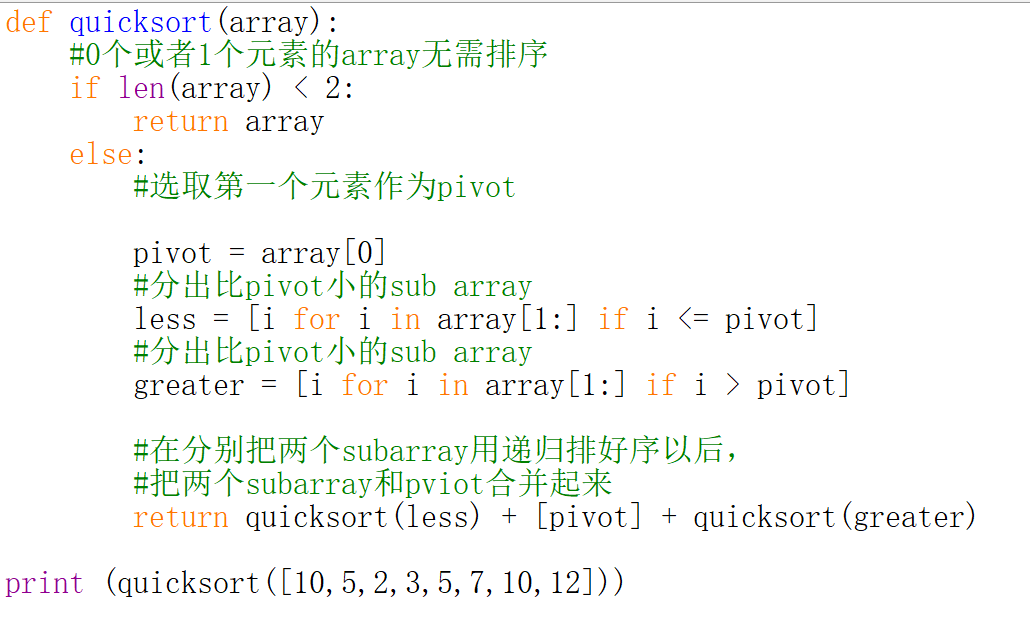
2. 快排的步骤，首先选一个pivot，然后把列表分成两部分，比pivot小的和比pivot大的，最后对两个部分分别递归调用快排法

3. 分治法说白了就是每一次operation都要把问题量缩小，直至base case，所以一般的base case都是empty array或者只有一个变量的array

4. 快排的平均时间复杂度是nlogn，这是随机选pivot的结果

5. Quicksort和mergesort比较的时候constant就有区别了，因为两个的时间复杂度都是nlogn

6. 代码实现：



第五章：Hash Table

1. Hash函数就是你输入一个string然后返回一个number，正规的来讲是hash函数映射string到number

2. Hash表有限定要求，首先必须是consistent，即每次输入的string对应返回的number都应该一样，并且不同的词要对应的数字要都不一样

3. Python里面实现hash table的方法就是字典，一个key对应一个value

4. Hashtable的三大主要应用：匹配查询（商店里苹果的价钱），防止重复输入（投票系统拒绝已经投过的人继续投票），网络缓存

5. collision：two keys have been assigned the same slot

6. 解决collision的一个经典办法就是当多个key 映射到同一个slot的时候，在那个slot建立一个链表

7. Hash table的average case是O（1），这是constant time，意思是耗费时间不会随着hash table的大小而改变，worst case，要花O（n）

8. 为了避免collision，有两点需要注意，low load factor和good hash function

9. load factor的计算方法：number of items in hash table/total number of slots

10. 比较好的一个时间resize是当load factor超过0.7的时候

第六章：BFS

1. BFS算法是用于解决最短路径问题

2. 解决最短路径的过程，先把路径模拟成graph，然后用bfs解决

3. Graph是由节点和edge（边）组成的

4. Queue的操作有enqueue（加入元素）和dequeue（删除元素）

5. 由于其FIFO的特性，可以很好的用在bfs的search list上，因为search list需要先处理先进来的人

6. Hash table可以用于实现Graph，这里相当于映射node到所有的neighbors

7. one way的被称为单向图，directed graph,有箭头，无箭头的就是无向图，两边的元素互为neighbor

8. 实现一个graph很简单：

首先graph = {}，创建一个dict来代表hash table

然后graph[“you”] = [“alice”, “bob”]，一级node，直接和你联系的

然后在定义二级，三级node直至graph[“joy”] = [],这代表joy后面没有neighbor了

9. 用双端队列实现，当有共同朋友的时候，会出现，search一个人两次的情况，所以必须mark一下已经search过得人，具体做法是keep a list

10.用bfs实现在自己的network中卖芒果人的最短路径



11. bfs是时间复杂度是O(V+E)，v是node，e是edge，在芒果的例子中相当于是O（number of people + number of edges）

12. 拓扑排序就是把不同的任务按照必要的完成先后顺序排序，比如task a 要建立在task b至上，那么task a就要排在task b的后面

13. Tree也是graph的一种，就是不能往回指向的图

第七章：Dijkstra算法

1. bfs是帮助寻找最短路径，dijlstra是帮助找最快路径

2. dijkstra的四个步骤：找到离你最近的node，检查这个node到它的neighbor有没有更短的时间，有的话就更新，重复以上过程直到graph所有node都被执行，算出最终路径

3. dij算法适用于赋权图的，赋权图就是edge被赋值的图，所以如果想计算非赋权图的最短路径，就用bfs，如果是赋权图的最短路径，就用dij算法

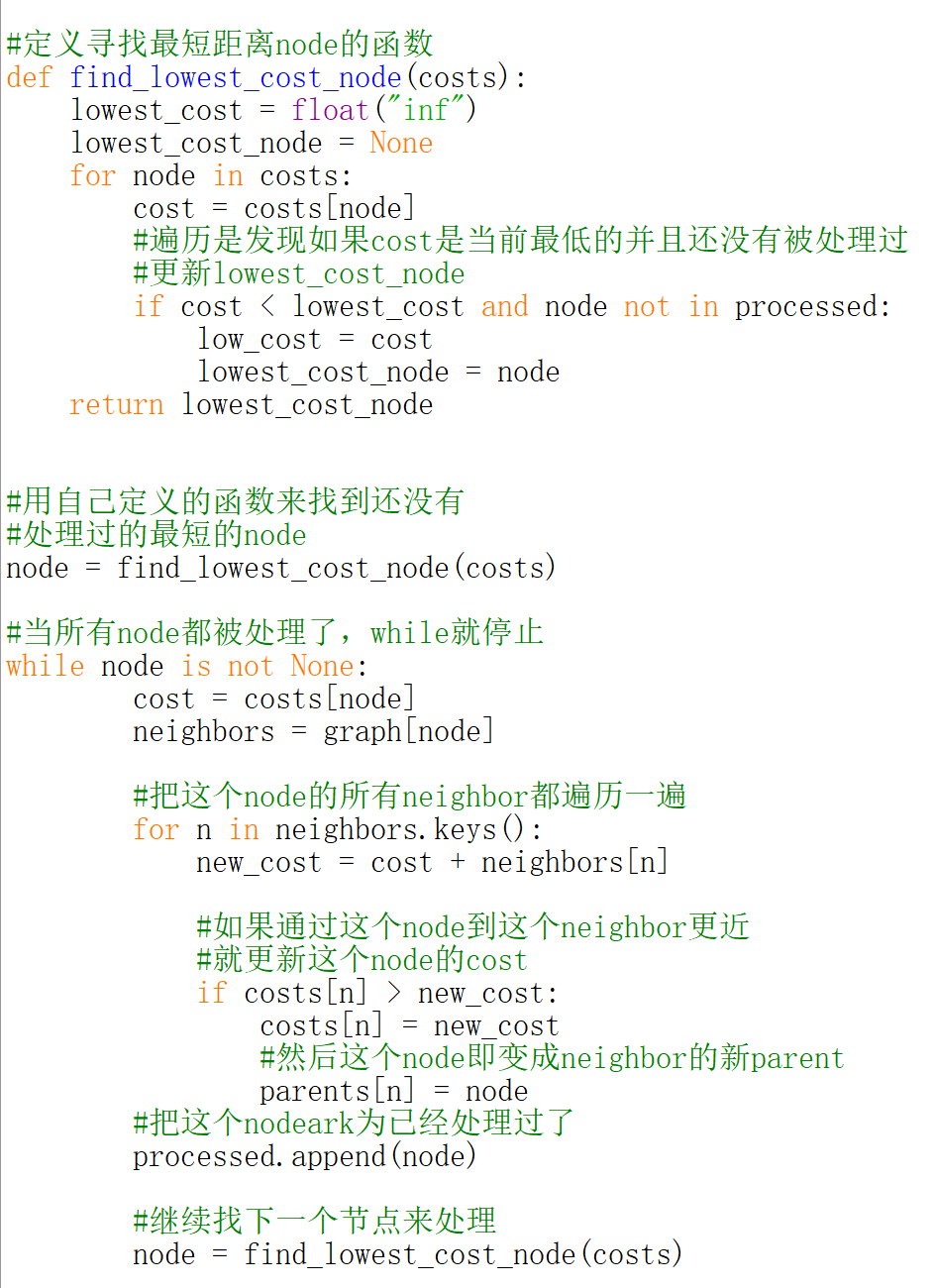
4. dij算法只能用于有向图，因为无向图会造成循环，那样用于都不会找到最短路径

5. dij算法不能用于negative-weight edge，加权的数必须是正数

6. 如果想在negative-weight edge中找到最短路径，要用到bellman-ford算法，此算法不在本书涵盖范围，可日后查看

7. 代码实现：





第八章：贪心算法

1. 原理就是每一步都选择局部最优解，最后就会得到全局最优解

2. 贪心算法其实是一种近似算法，非常接近最优解，在最优解要耗费太多时间的时候采用的一种算法

3. 贪心算法复杂度为O(n^2)

4. 旅行者有多个城市要去，怎么选最佳路径的时候，这种问题，的operation组合是n!,如果数量庞大根本不可能算出最优解，因为组合太多，所以这个时候就会考虑用贪心算法，得到近似最优解

5. 判断NP complete问题的几个要点：

元素少时算法运行速度非常快，但是元素一旦大规模增加，速度回变得非常忙

涉及所有组合（all combination of xxx）问题一般都是NP complete

不能用分治法，必须考虑所有可能的时候

如果涉及到序列且难以解决，就有可能是np

涉及集合且难以解决也可能是np

如果问题能转换为set-covering问题就有可能是np

第九章：动态规划

1.动态规划就是把问题分成小问题，解决了小问题以后再合并解决大问题，背包问题中，我们把背包分成小背包，计算出小背包能装的最大价值，这样当有剩余空间时就可以用小背包的值补上从而算出max value

2. 每个动态规划问题开始之前都要做一个grid网格，分成row和column来对背包进行划分

3. 每个小网格的值有计算公式:

I是row，j是column

Cell[i][j] = max of下面两个元素:

The previous max(value at cell[i-1][j])

Value of current item + value of the remaining space(cell[i-1][j-item’s weight])

4. DP不能解决fraction问题，要么整个拿走，要么都别拿，不能那一半或者零点几，如果要解决零点几的问题还是首推greedy，因为贪心就是一个接着一个的拿最有价值的东西

5. DP要实现必须是每个子问题都是离散的，即每个子问题直接不存在互相依赖的关系，完全独立

6.动态规划能在约束条件下找到最优解

7. 动态规划中每个grid都是一个子问题，所以怎么建立网格是个值得思考的问题，首先要确定网格的value，然后就是怎么把这个问题划分成子问题，横纵坐标分别是什么

8. 记住重点，cell的value是我们想要优化的东西

9. longest common substring问题：

If word1[i] = word2[j]:

Cell[i][j] = cell[i-1][j-1] + 1

Else:

Cell[i][j] = 0s

10. longest common sequence问题:

If word1[i] == word2[j]:

Cell[i][j] = cell[i-1][j-1] + 1

Else:

Cell[i][j] = max(cell[i-1][j], cell[i][j-1])

11.DP问题的难点就在于怎么样划分子问题，怎样设置value和怎样找出最后的公式

12. DP实际应用，DNA相似度识别，diff方法（文件区别），word中的自动换行

第十章：K最邻近算法（KNN）

1. KNN算法可以用于对数据进行分类，进而用于推荐系统，预测系统等等

2. 对于KNN算法，要进行feature extraction，一般都会有多个feature，为他们打分，然后算和其他用户之间的距离，越近的表明越相似，推荐也就越准确

3．KNN是由classification和regression组合而成的，前者分类，后者预测

4. 实际应用中距离公式很少用，一般用的比较多的是余弦相似度（cosine similarity）,不过这个不在本书范畴，日后可以补充

5.选取feature的时候不能带有偏见

第十一章：下一步

1. 二分法有个问题就是当给array插入新数据时必须对整个array重新排序，否则下一次二分法就不能用了，为了解决这个问题，可以用到一个数据结构binary search tree
2. 在bst中，左边的node永远是小于右边的node，bst的读取，删减元素的复杂度都是O（logn）
3. 搜索引擎中比较常用的一个数据结果是反向索引（inverted index），也会用到hash table来映射
4. Fourier transform（傅里叶变换），能进行分离分析，可用于信号处理，音乐压缩，因为他可以判断不同的频率
5. 单个主机的情况下可以用parallel algorithm来提高performance，但是如果有上千个core，就会用到分布式算法，mapreduce，比如hadoop
6. Map就是take an array and apply the same function to each item in the array
7. Reduce就是transform an array to a single item

CC150：

Arry和String总结

1. 字符串去重可以用set
2. 需要用什么代替某个字符的时候，可以用replace函数，也可以遍历的时候用append
3. 回文数的判断尽量用[::-1]，把其他形式的数据结构转成list就可以操作了
4. 半段选择字符的时候可以把字符串走两遍，str+str，这样无论怎么选择肯定都在里面
5. 括号问题可以考虑用栈解决
6. 子字符串问题可以截取和给定子字符串相同长度的字符串然后进行比较
7. 排好序的列表多考虑是否能用二分法
8. 二维列表（矩阵）都需要行和列,行是len(array)，列是len(array[0])
9. 要使用双指针时要先定义两个指针的其实位置，比如一个开头一个结尾，然后定义停止条件，一般是while left < right

Linked List总结：

1. 主要操作有遍历，头部添加元素，尾部添加元素，指定位置添加元素，删除节点，查看节点是否存在
2. Kth类问题的时候可以用到快慢针，快针以一个固定的遍历速度先走，具体过程是让两个指针都先指向头结点head，然后根据需求遍历，定义快慢针的时候记得把head的值分别赋给slow和fast，方便他们遍历
3. 主要操作就是.next
4. 判断空链表的方法就是node.next == None
5. 删除节点的方式是把.next赋给当前节点，下一个节点指向下下个节点
6. 有时候可以创建一个list来存放需要的元素，尤其是处理两个链表操作的时候，可能需要创建第三个空链表
7. 总是记得边界值情况，只有一个元素或者无元素的时候，如果给了head就是if head == None or head.next == None
8. 遍历链表的时候很多时候需要把head的值赋给新变量，以免到了Listnode尾部，再次扫描的时候就变成None.next了，而None没有next和val这些属性，会报错
9. 上一个节点pre的定义方式是pre.next = head

Stack和Queue总结：

1. Stack是LIFO，四个操作

Pop：删除头元素

Push：加入新元素在头部

Peek：返回最上面的元素

isEmpty：查看栈是否为空，是的话返回true

Tree和Graph总结：

1. BST有很多种定义，如果被问到记得和面试官确认是哪一种
2. 一般被问及tree的问题是，很多人都默认是bst，但其实不一定的，要和面试官确认
3. 所谓的balanced tree并不一定是两边完全一样的size，只要是能达到O(log N) time for insert 和find即可
4. 比较出名的平衡树有红黑树和AVL树
5. 树的遍历：

Inorder（left，root，right）

Preorder（root，left，right）

Postorder（left，right，root）

当遍历BST的时候，inorder是按照升序来访问节点的

Preorder中，root总是第一个被访问的节点

Postorder中，root总是最后一个被访问的

1. Minhead，最小堆，这是一个complete binary tree，按照从大到小的顺序排列，root是最小的
2. 求树的深度时记得+1，因为要加上根节点
3. 确定当前节点是否有右节点的时候就用if currNode.ritht
4. 树的问题大多都可以要么递归要么迭代，递归就是dfs

**OOD总结：**

1. 当被问到OOD问题时，首先要明确设计的这个东西是谁用，怎么用，有必要的话可以把六要素都问了：who，what，where，when，how，why
2. 分为六步走：
3. 消除模棱两可的地方，清楚需求
4. 定义核心元素：哪些东西是必备的
5. 分析关系：分析元素直接的关系，有没有继承，一对多，多对多

**递归和DP总结：**

1. 当遇到类似compute the nth, list the first n, to compute all, 这种类型的问题大多都可以用递归
2. 递归算法在空间上非常耗费，因为每一次递归调用都会在stack上加一个新layer，也就是说如果你的算法要调用depth of n，就至少需要O(n)memory，所以通常情况我们尽量iteratively的去实现递归算法
3. 通常求最大字段的四种方法

一．动态规划：O（n）

二．将给定序列分成长度相等的两段，分别求出这两段的最大字段和，O（n\*logn）

三．先计算字段和结果，储存到s[]中，即预处理，这样每次在计算a[i]到a[j]之间的数和的时候就等于s[j]-s[i]，复杂度为O(n^2)

四．枚举i和j，求i和a[i]到a[j]之间的和的最大值，时间复杂度O(n^3)

4. DP和递归的不同：首先递归肯定有两个特点，问题具有最优子结构，有大量重复子问题，而递归和DP不同的地方在于，这些重复的子问题，DP算法将其结果保存下来，等下一次又要计算该子问题的时候直接调用已计算好的，而递归不是这样，递归是一遍又一遍的计算这些重复的子问题，从而效率很低，能用DP的就别用递归

5. 初始化dp一定要写dp = [0]\*(len(num)),相当于把之前给的列表全部赋值为0