算法分析

算法设计与算法分析

算法就是一套固定的指令，可以重复使用。

算法的要素

变量、操作、条件、循环、列表、函数、类

算法设计

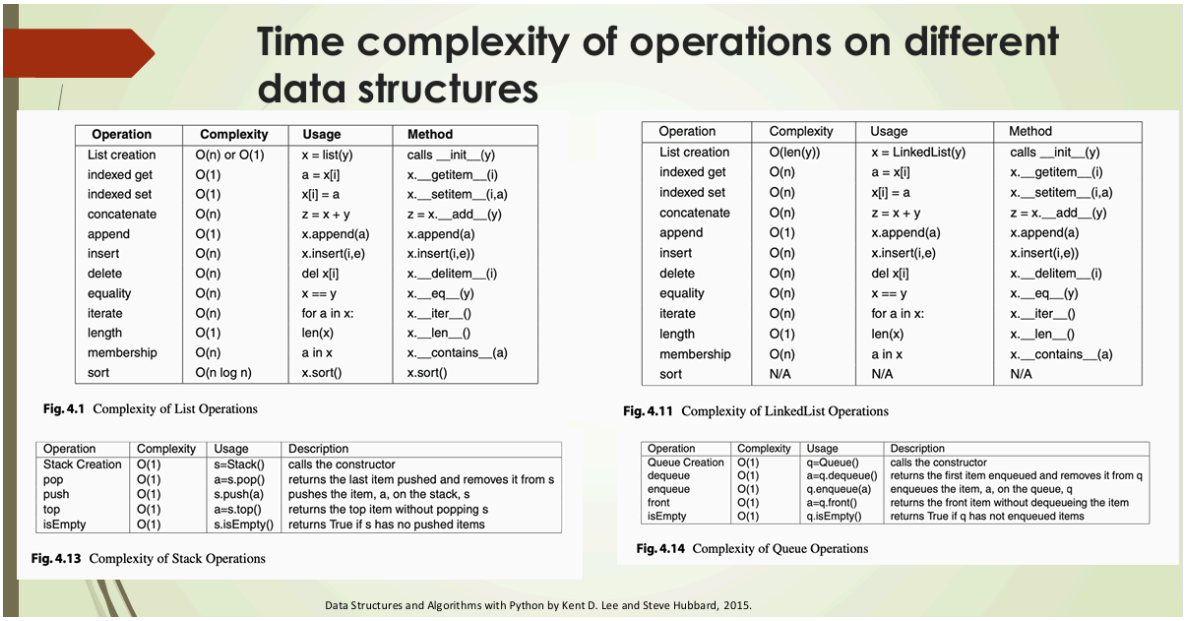
算法设计需要保证所用时间在合理的范围内，并且追求可扩展性（能够在面对更大规模的数据时加快解决问题的速度）。可扩展性的是通过所谓的算法效率度量的。算法运行所需要的时间取决于计算机本身的性能和差异，但是算法效率与之无关，这里我们采用所谓的渐进分析（Asymptotic Analysis）来度量算法本身的运行效率

算法效率

比起具体的时间，我们更关心算法运行了多少步骤。通常来说，循环（for或者while）自带n次运算，然后内部的每次操作/判断都需要n次。循环之外的操作基本上是一行即一次执行。由此可以算出一个算法最多和最少要执行作多少步骤。渐进分析有三种符号：大O符号，Ω符号和Θ符号，分别表示：时间复杂度的上界、下界和平均表现，当然我们最常用的还是大O符号。

O-Notation规则

我们都知道时间复杂度是关于n的函数，我们只取其最高阶项并忽略常数。求和的时候只取高阶项，作积或复合的时候则都保留。比较常见的大O符号有：
$O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), O(2^n), O(n!)$，复杂度也依次升高。下图也展示了同样的操作不同的数据结构的时间复杂度，可以看出对特定的操作，可能会有特别管用的数据结构：



复杂度

复杂度不仅有时间复杂度，还有空间复杂度。有些算法在运行过程中会创建临时的列表等，占用空间，因此对某些问题也必须考虑空间复杂度是否在可控范围内，比如嵌入式系统或移动应用程序，但是在大气中一般不需要太关注。递归操作占用不少空间复杂度。哈希表通过牺牲空间复杂度来换取了更低的时间复杂度

优化策略

提高算法效率最好的方式当然是直接优化算法，比如避免或减少循环的嵌套，或者其他时间复杂度比较高的操作（如切片、计数、求和等）。

如前所述，使用合适的数据结构也很重要，综合问题特征、存储限制、顺序要求等等，具体而言，我们可以利用的是：

数组可以利用指标实现快速处理，并且其储存是连续的；

链表：高效插入和删除；

哈希表：快速查找、插入和删除；

树：分层数据表示，高效检索和排序；

堆：优先队列，高效获取最大值和最小值；

图：表示实体之间的关系；

栈：后进先出操作；

队列：先进先出操作

当然除此之外也有一些好用的算法套路/思想，比较常用的：

暴力算法；

分治策略；

贪心算法；

动态规划；

回溯。

例题提供了一种比较常用的思想：

滑动窗口。比起遍历数组，通过只考虑两个标记的移动来避免对显然不必要考虑的情况的计算。

哈希表

哈希是什么？哈希就是不管你的输入是什么，我都把它变成哈希值（通常是整数），这样我查找的时候就不用一个个对比名字是啥，而是可以直接跳到对应的位置查询。其要求是不会撞车（即两个不同的输入哈希成同一个数），所以我们通常需要加入判断，如果哈希值已经被占用了，通常是+1；然后如果我们给的整数集不够大，比如10^2却储存了80个数，此时效率不够，需要扩大哈希值的范围。

所以哈希表的特点就是元素不可直接修改，但是查找很快，插入、删除也很方便，对大数据尤其管用，并且还有安全性（不可逆）。缺点就是有撞车风险、哈希函数不可逆、不便于排序、占用空间大等等。

在python中集合和字典用的都是哈希表

Operation	Complexity	Usage	Description
Set Creation	O(1)	s=set([iterable])	Calls the set constructor to create a set. Iterable is an optional initial contents in which case we have O(n) complexity.
Set Creation	O(1)	s=frozenset([iterable])	Calls the frozenset constructor for immutable set objects to create a frozenset object.
Cardinality	O(1)	len(s)	The number of elements in *s* is returned.
Membership	O(1)	e in s	Returns True if *e* is in *s* and False otherwise.
non-Membership	O(1)	e not in s	Returns True if *e* is not in *s* and False otherwise.
Disjoint	O(n)	s.isdisjoint(t)	Returns True if *s* and *t* share no elements, and False otherwise.
Subset	O(n)	s.issubset(t)	Returns True if *s* is a subset of *t*, and False otherwise.
Superset	O(n)	s.issuperset(t)	Returns True if *s* is a superset of *t* and False otherwise.
Union	O(n)	s.union(t)	Returns a new set which contains all elements in *s* and *t*.
Intersection	O(n)	s.intersection(t)	Returns a new set which contains only the elements in both *s* and *t*.
Set Difference	O(n)	s.difference(t)	Returns a new set which contains the elements of *s* that are not in *t*.
Symmetric Difference	O(n)	s.symmetric_difference(t)	Returns a new set which contains *s.difference(t).union(t.difference(s))*
Set Copy	O(n)	s.copy()	Returns a shallow copy of *s*.

Fig. 5.4 Set and Frozen Set Operations

these operations are all efficient with a

Operation	List	Set
`L.append` / `S.add`	$O(1)$	$O(1)$
`in`	$O(n)$	$O(1)$
`L.remove` / `S.remove`	$O(n)$	$O(1)$

Operation	Complexity	Usage	Description
Dictionary Creation	O(1)	d = {[iterable]}	Calls the constructor to create a dictionary. Iterable is an optional initial contents in which case it is O(n) complexity.
Size	O(1)	len(d)	The number of key/value pairs in the dictionary.
Membership	O(1)	k in d	Returns True if *k* is a key in *d* and False otherwise.
non-Membership	O(1)	k not in d	Returns True if *k* is not a key in *d* and False otherwise.
Add	O(1)	d[k] = v	Adds *(k,v)* as a key/value pair in *d*.
Lookup	O(1)	d[k]	Returns the value associated with the key, *k*. A *KeyError* exception is raised if *k* is not in *d*.
Lookup	O(1)	d.get(k[,default])	Returns *v* for the key/value pair *(k,v)*. If *k* is not in *d* returns *default* or *None* if not specified.
Remove Key/-Value Pair	O(1)	del d[k]	Removes the *(k,v)* key value pair from *d*. Raises *KeyError* if *k* is not in *d*.
Items	O(1)	d.items()	Returns a view of the key/value pairs in *d*. The view updates as *d* changes.
Keys	O(1)	d.keys()	Returns a view of the keys in *d*. The view updates as *d* changes.
Values	O(1)	d.values()	Returns a view of the values in *d*. The view updates as *d* changes.
Pop	O(1)	d.pop(k)	Returns the value associated with key *k* and deletes the item. Raises *KeyError* if *k* is not in *d*.
Pop Item	O(1)	d.popitem()	Return an abritrary key/value pair, *(k,v)*, from *d*.
Set Default	O(1)	d.setdefault(k[, default])	Sets *k* as a key in *d* and maps *k* to *default* or *None* if not specified.
Update	O(n)	d.update(e)	Updates the dictionary, *d*, with the contents of dictionary *e*.
Clear	O(1)	d.clear()	Removes all key/value pairs from *d*.
Dictionary Copy	O(n)	d.copy()	Returns a shallow copy of *d*.

Fig. 5.6 Dictionary Operations

手搓哈希表（不考！）

考试不会让我们真的搓一个类出来。所以我们先来看一下类当中常用的记号：

Syntax	**Meaning**
_variable/object	the variable is meant for internal use only, e.g. from M import * does not import objects whose name starts with an underscore
class_	avoiding naming conflicts with Python keywords and built-in names
__attribute	triggers name mangling in Python classes, Class._ClassName__attribute
__attribute/method __	Python attributes and methods
-	a temporary or throwaway variable

undefined

```
# %% define a class 这里作为定义一个类的例子，面向对象编程
class student:  类别：学生
    def __init__(self,name = [],age =[],gender = []):  #初始化，包含变量（对象自身
 (self)，变量的属性)
        self.name = [name]  #基本语法
        self.age = [age]
        self.gender = [gender]
    def __str__(self):  #直接调用对象要输出什么文本（string)
        return f'{self.name}({self.age},{self.gender})'  #自行定义
    def __gender(self,gender):  #内部变量
        self.gender.append(gender)
    def myfunc(self,i):  #随便你想定义什么函数
        print('Hello my name is ' + self.name[i])
    def __add(self,name,age):
        self.name.append(name)
        self.age.append(age)
# %%
test = student('LiHongTao',21,'M')
# %%
test.myfunc()

# %%
test._student__add('JiangYe',20)

# %%
test._student__gender('M')
# %%
test.myfunc(1)

# %%

# %%
[_+5 for _ in range(5)]

# %% Hash table  下面正式进入哈希表的构建
class hashtable:
    def __init__(self,slots=[],data=[]):  #初始化包含（self，插槽集合和数据集合)
        if len(slots)!=len(data):  #如果能供储存的插槽数要等于数据集合的大小
            raise Exception('Error:length mismatch of slots and data')
        self.size = 10  #设置大小
        self.slots = [None]*self.size #构建给定大小的插槽集合
        self.data = [None]*self.size  #构建给定大小的数据集合
        if len(slots)>=1:  #确保size不是0或是负数
            if len(slots)>self.size:  #如果slot长度大于size
                self.size = len(slots)+1  #自适应size
                self.slots = [None]*self.size
```

```
                self.data = [None]*self.size
            for i,j in zip(slots,data):  #把数据缝合起来加入哈希表
                hashtable.__add(i,self.slots,j,self.data)
    def __add(key,slots,value,data):  #插入元素
        idx = hash(key) % len(slots)  #哈希函数给出的通常是很大的整数，为了保证其值为我们想要的插槽而不会真的很大，我们取余
        while slots[idx] != None:  #如果撞车了，重复以下操作直到不撞车
            if slots[idx] == key:  #如果插槽已有，但是内容一样就不用理他
                return False
            idx = (idx+1)%len(slots) # was missing!  否则+1试试
        slots[idx] = key  #赋值
        data[idx] = value  #赋值
        return True
    def __rehash(oldslots,newslots,olddata,newdata):  #重哈希是中间步骤，目的是在改变了插槽大小之后重新把数据录入
        for x,y in zip(oldslots,olddata):  #旧数据缝合
            if x!=None:
                hashtable.__add(x,newslots,y,newdata)  #到新的插槽里重新添加
        return newdata, newslots  #返回新的哈希值
    def add(self,key,value):  #加入元素的完整代码
        if hashtable.__add(key,self.slots,value,self.data):  #运行插入元素，如果正常插入（注意__add操作完之后返回的是布尔型数值）
            load = len(set(self.slots))/self.size  #如果正常插入则计算负载
            if load>=0.75:  #如果太满了，干扰运行效率，扩容然后重哈希
                self.slots,self.data = hashtable.__rehash(self.slots,
[None]*2*self.size,self.data,[None]*2*self.size)
                self.size = len(self.slots)
    def __remove(key,slots,data):  #移除元素
        idx = hash(key) % len(slots)   #计算哈希值
        while slots[idx] != None:  #运行直到目标被删除
            if slots[idx] == key:  #如果检测到目标则删除（赋值None）
                slots[idx] = None
                data[idx] = None
                return True
            idx = (idx+1)%len(slots)
            # idx = idx+1 # equivalent to line 74
        return False
    def remove(self,key):  #移除元素完整代码
        if hashtable.__remove(key,self.slots,self.data):  #移除元素
            load = len(set(self.slots))/self.size  #如果正常移除则计算负载
            if load <= 0.25:  #如果太空了，浪费空间，则缩容然后重哈希
                self.slots,self.data = hashtable.__rehash(self.slots,
[None]*self.size/2,self.data,[None]*self.size/2)
                self.size = len(self.slots)
    def __contains__(self,key):
        idx = hash(key) % len(self.slots)
        while self.slots[idx] != None:
            if self.slots[idx] == key:
```

```
                return True
            idx = (idx+1)%len(self.slots)
            # idx = idx+1 # equivalent to line 74
        return False
    def __getitem__(self,key):
        idx = hash(key) % len(self.slots) # this is necessary to tranlste the
key to index in slots and data!
        return self.data[idx]
    def __setitem__(self,key,value):
        self.add(key,value)
# The indentation was wrong!!!!
# def __contains__(self,key):
#     idx = hash(key) % len(self.slots)
#     while self.slots[idx] != None:
#         if self.slots[idx] == key:
#             return True
#         idx = (idx+1)%len(self.slots)
#     return False
# def __getitem__(self,key):
#     return self.data[key]
# def __setitem__(self,key,value):
#     self.add(key,value)
# %%
HT = hashtable([1,2],['a','b'])
HT.add(3,'c')

HT.add(555,'d')
# %%
3 in HT
# %%
HT.remove(3)
# %%
HT[2]



# %% demostration about the issue of the while loop
def test(i):
    s = [_ for _ in range(10)]
    while s[i] != None:
        if s[i] == 'a':
            return True
        i = (i+1)%10
        print(i)
    return False

# %%
```

```
test(0)
# %%
```

排序算法

排序就是把数组中的元素按照大小顺序排列，通常我们默认是升序，我们接下来会介绍9种排序算法及其实现：

Method	Core idea
Bubble Sort	Repeatedly swap adjacent elements if out of order
Selection Sort	Repeatedly select the minimum and place it in correct position
Insertion Sort	Build the sorted portion one element at a time, $O(n)$ if already sorted
Merge Sort	Divide&conquer, split&merge
Quick Sort	Divide&conquer, partition around a pivot. Good pivot choice important!
Heap Sort	Use heap data structure
Counting Sort	Based on counts of elements, k - range of values
Radix Sort	Sort digits one by one (use counting sort internally)
Bucket Sort	Distribute into buckets, sort each, then merge

Algorithm	Best Case	Average Case	Worst Case	Space	Stable?	
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Yes	simple
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	No	
Insertion Sort	O(n)	O(n²)	O(n²)	O(1)	Yes	
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	Intermediate
Quick Sort	O(n log n)	O(n log n)	O(n²)	O(log n)	No	
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No	
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)	Yes	Advanced
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)	Yes	
Bucket Sort	O(n+k)	O(n+k)	O(n²)	O(n+k)	Yes	

- Stable sorting: preserves relative order of equal elements (e.g., Merge Sort, Insertion Sort, Counting Sort).
- Unstable sorting: may change the order of equal elements (e.g., Quick Sort, Heap Sort, Selection Sort).

冒泡排序

很简单的想法，跑i趟，每趟确定一个最大元素。具体来说，在每一趟里，有一个指针j，如果所指比右边小，则指针后移，如果比右边大，则指针后移，同时交换。

```
def BubbleSort(arr):
    n = len(arr)  #获取长度
    for i in range(n):  #遍历n回
        swapped = False  #定义布尔型变量
        for j in range(0,n-i-1):  #对前n-i-1个数
            if arr[j] > arr[j+1]:  #如果前面比后面大
                arr[j],arr[j+1] = arr[j+1],arr[j]  #则前后交换
                swapped = True
        if (swapped == False):  #如果值是False则结束运行。因为当已经排好序了，到某一次循环中没有需要交换的（也就是排完了），swapped不会变成True
            break
```

选择排序

更简单了，就是找个最小的放在最左边，然后在剩下的里面找最小的，依此类推。

```
# %% Selection Sort

def SelectionSort(arr):
    n = len(arr)  #获取长度
    for i in range(n-1):  #遍历数据
        min_idx = i  #记最小值为第i个数
        for j in range(i+1,n):  #从i开始遍历
            if arr[j]<arr[min_idx]:  #查找最小值
                min_idx = j

        arr[i],arr[min_idx] = arr[min_idx],arr[i]  #找到i到n之中的最小值，将其与i交换
```

插入排序

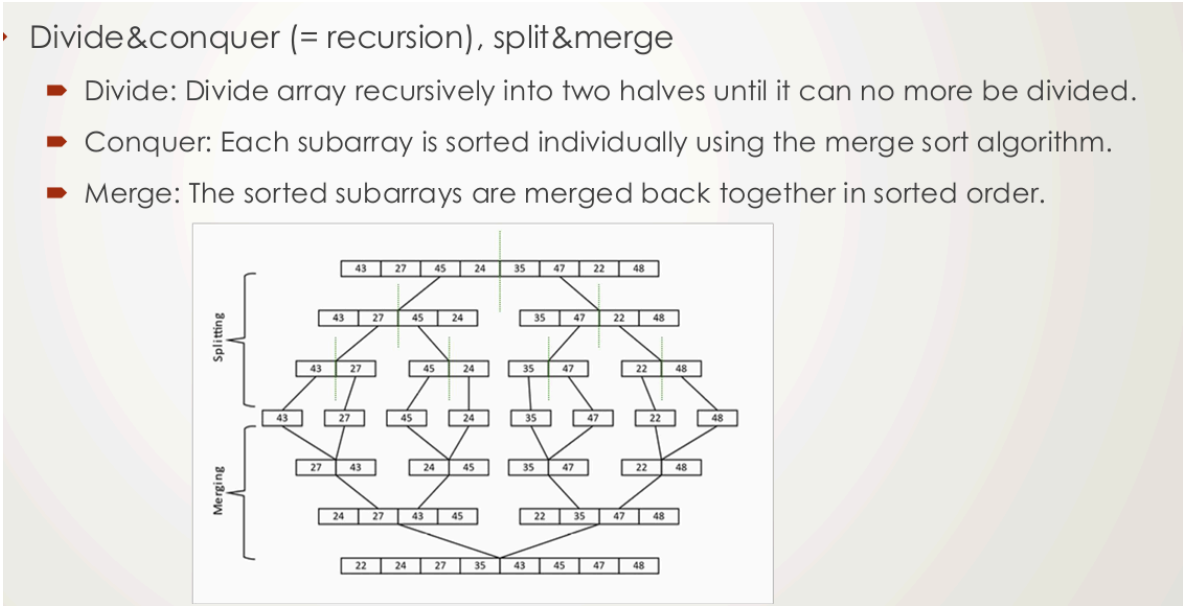
我们先排前i个，拍好之后看i+1个大还是小，依次往左挪到合适的位置。

```
# %% Insertion Sort

def InsertionSort(arr):
    for i in range(1,len(arr)):  #遍历数据
        key = arr[i]  #保存第i个数
```

```
        j = i-1  #取i左边的数
        while j >= 0 and key < arr[j]:  #如果第i个数比第j个数小
            arr[j+1] = arr[j]  #则将第j个数往右挪
            j -= 1  #j往左走
        arr[j+1] = key  #最后，第i个数比第j个数大，说明位置到了，把第i个数存进来。
```

归并排序

这个稍微复杂一点了，时间复杂度也降到了$O(n\log n)$。这个是典型的分治思想



分容易（$O(\log n)$），分到最后成单个元素自然不需要排了，合成比较麻烦，主要的时间复杂度在这（$O(n)$）。简单来说就是在要合成的两个数组里面分别塞一个指针，然后创建一个新的数组，比较两个指针大小，小的塞进去，指针+1，直到某一个空了，然后由于剩下的也是排好序的，所以直接填满就行。

```
#主程序
def MergeSort(arr):
    if len(arr)<=1:  #如果数组长度为1，则自动排好了
        return arr
    mid = len(arr)//2  #取中间位置
    left_half = MergeSort(arr[:mid])  #将左右两边分别取，然后递归自身（因为我们知道这个函数能够找到最优的结果，所以左半右半也可以直接递归得到最优的）
    right_half = MergeSort(arr[mid:])
    return Merge(left_half,right_half)  #将结果归并

#归并代码
def Merge(left,right):
```

```
    sorted_array = []  #新建空数组来加入排好的元素
    i = j = 0  #两个指标皆从0开始
    while i<len(left) and j<len(right):  #i负责左，j负责右，重复操作直到其中一个填完了
        if left[i] <= right[j]:  #如果左边比右边小，则填入左边，然后i+1
            sorted_array.append(left[i])
            i += 1
        else:                    #反之亦然
            sorted_array.append(right[j])
            j += 1
    sorted_array.extend(left[i:])  #剩下的部分直接减下来贴后面
    sorted_array.extend(right[j:])
    return sorted_array  #输出排好的数组
```

当然这意味着我们生成的结果不在原地，而是另一个数组了，为此我们在作业中实现了原地排序的算法。显然二分不会导致非原地，所以问题出在如何合并。操作也不是很麻烦，也要新建数组，只不过原来是把要合并的抄写到新的数组，现在是把要合并复制下来，然后抄写回原数组。

快速排序

快速排序的思想也是分治。先选一个元素，然后比它小的放左边，大的放右边，它放中间；然后对两边再选元素分，最后就排好序了。为了防止数组本身有一定的顺序，随机往往比较好，随机完我们把它取到最右边，然后左右各设一个指标，先跑左边，遇到小的不动，遇到大的跟它交换，然后从右边跑，大的不动，小的跟它交换，依此类推，直到左右指标重合。这里显然也分原地与否，但是实际上的思路没有变化。非原地占用空间为$O(n)$，原地则为$O(\log n)$。

```
import random

#非原地的算法
def QuickSort(arr):
    if len(arr)<=1:  #如果数组大小为1，自然已经排好了
        return arr
    pivot_index = random.randint(0,len(arr)-1)  #随机取一个主元
    pivot = arr[pivot_index]
    left = [x for i,x in enumerate(arr) if x <= pivot and i != pivot_index]  #把所有小于等于主元而index非主元的放左边
    right = [x for i,x in enumerate(arr) if x > pivot and i != pivot_index]  #大于的放右边

    return QuickSort(left)+[pivot]+QuickSort(right)  #然后对左右分别递归之后合起来


#原地的算法

#主程序
```

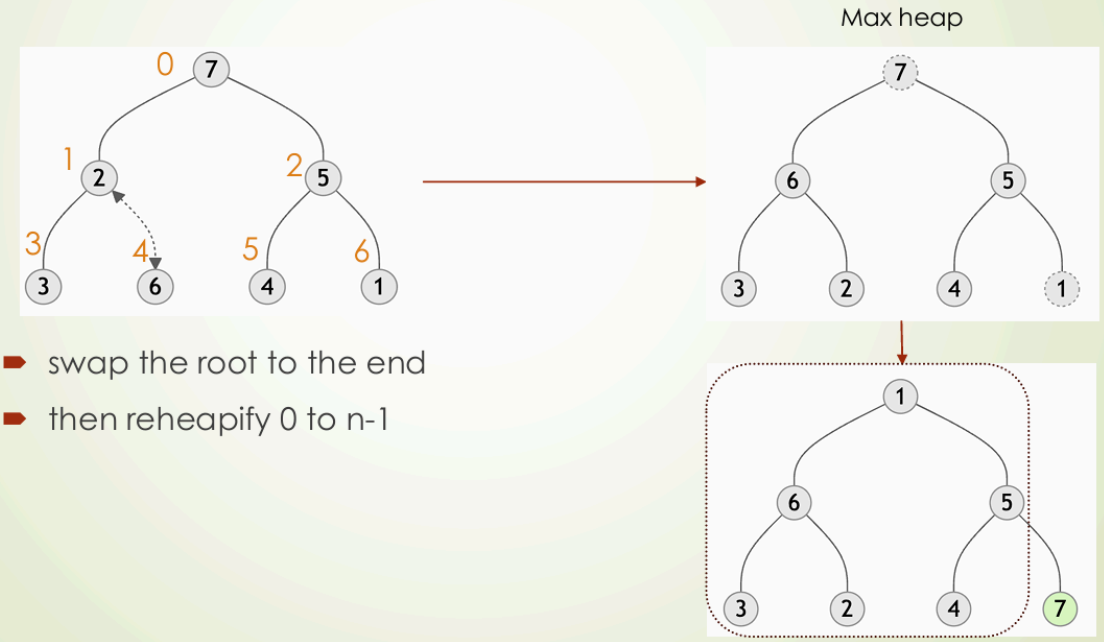
```
def QuickSort(arr,low,high):
    if low<high:
        pivot_index = randomized_partition(arr,low, high)  #随机取一个作为主元
        QuickSort(arr,low,pivot_index-1)  #对左右分别递归
        QuickSort(arr,pivot_index+1,high)

#随机到的主元移到最右边
def randomized_partition(arr,low, high):
    pivot_index = random.randint(low,high)
    arr[pivot_index],arr[high] = arr[high],arr[pivot_index]
    return partition(arr, low, high)  #然后对单一边排

def partition(arr,low,high):
    pivot = arr[high]
    i = low - 1  #i为最左边
    for j in range(low,high): #j从最左边遍历到最右边
        if arr[j] <= pivot:  #如果第j个比主元小，则i+1并交换i j
            i += 1
            arr[i],arr[j] = arr[j],arr[i]
    arr[i+1],arr[high] = arr[high],arr[i+1]  #j遍历完之后将最右边的与i+1交换
    return i+1  #返回此时的i+1
```

堆排序

堆排序是将数组视为堆（一种二叉树），然后先对其进行排序，每个节点跟下面的比，如果下面大则交换，直到从下到上依次变大。接着将其中某个叶节点跟顶部交换，顶部那个拿出去，然后再排序得到第二大的顶部，依此类推。

Intermediate algorithms $O(n \log n)$
Heap Sort: How does it work?

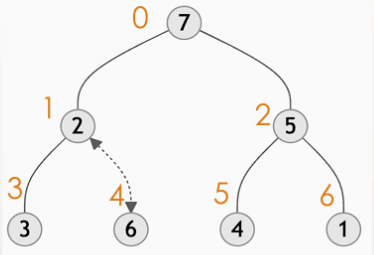
- Create a heap data structure



- swap the root to the end
- then reheapify 0 to n-1

Intermediate algorithms $O(n \log n)$
Heap Sort: How to find the last parent node?

- In a binary heap stored as an array with **n** elements:
  - Parent of node **i** is at **(i - 1) // 2**
  - Children of node **i** are at **2*i + 1** and **2*i + 2**
- all nodes from index **n // 2** to **n - 1** are leaf nodes (they have no children).
- Last parent node is at index **n // 2 - 1**



arr = [7,2,5,3,6,4,1]

```
#建最大堆
def heapify(arr,n,i):
    largest = i #给定父母节点i
    left = 2*i+1  #其子节点分别为2i+1 +2
    right = 2*i+2

    #如果子节点比父母节点大，则重新定largest
    if left < n and arr[left] >arr[largest]:
        largest = left
    if right < n and arr[right] >arr[largest]:
        largest = right

    #如果最大值不是父母节点i，说明需要把下面的换上来
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]  #交换
        heapify(arr,n,largest)  #交换之后重新判断/排，直到最大的就是i


#主程序
def HeapSort(arr):
    n = len(arr) #获取数组长度

    #建最大堆
    for i in range(n//2-1,-1,-1):  #对每一个父母节点
        heapify(arr,n,i)  #保证其比子节点大

    for i in range(n-1,0,-1):  #建立完最大堆之后开始剪枝
        arr[0],arr[i] = arr[i],arr[0]  #把最大的放到最右边
        heapify(arr,i,0)  #然后重新建最大堆。
```

计数（Counting）排序

很简单，但是对数据类型有要求（纯排数，没有其他附加的信息）。我们就遍历一次，数每个元素出现了几遍，储存到对应的序号中。比如如果数字是0-10，那就直接创建一个大小为11的数组储存；如果比较大。比如80-90，那还是生成大小为11的数组，只是输出的时候对应的序号要加上最小元素80.

```
def CountingSort(arr):
    if not arr:
        return []
    max_val = max(arr)  #取最大最小值
    min_val = min(arr)
    range_of_elements = max_val-min_val+1  #计算目标区间
```

```
    count = [0]*range_of_elements  #设置range个插槽用来存放对应的数
    for num in arr:  #遍历所有数，如果在其中则+1
        count[num-min_val] += 1

    for i in range(1,len(count)):
        count[i] += count[i-1]

    #生成空位
    output = [0] * len(arr)
    for num in reversed(arr):  #遍历一遍区间内所有的num
        count[num-min_val] -= 1  #每次count-1
        output[count[num-min_val]] = num  #对应的位置上插入num
    return output   #输出
```

基数（Radix）排序

如果数字跨度很大怎么办呢？注意到我们人排的时候会按照数位排，所以借鉴这个想法，我们也按照数位排（具体操作是取余数%）。高位低位排都可以，这里我们用的是先从低位（个位）开始。创建0-9的10个数组，然后依次根据个位放到对应的数组中；然后按顺序提取出来，再看十位，放进去再从0到9取出来，最后肯定是高位大的在后面，高位相同的则是低位大的在后面，也就满足我们的要求了。

```
def CountingSort_for_radix(arr,exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for i in range(n):
        index = (arr[i]//exp)%10
        count[index] += 1
    for i in range(1,10):
        count[i] += count[i-1]
    for i in reversed(range(n)):
        index = (arr[i]//exp)%10
        output[count[index]-1] = arr[i]
        count[index] -= 1

    arr[:] = output

def _radix_sort_helper(arr):
    max_val = max(arr,default=0)
    exp = 1
    while max_val//exp >0:
```

```
        CountingSort_for_radix(arr,exp)
        exp *=10
    return arr

def RadixSort(arr):
    if not arr:
        return []
    negatives = [-x for x in arr if x<0]
    positives = [x for x in arr if x>=0]

    sorted_negatives = _radix_sort_helper(negatives)
    sorted_positives = _radix_sort_helper(positives)

    # xx = []
    # for x in reversed(sorted_negatives):
    #     xx.append(-x)

    return [-x for x in reversed(sorted_negatives)]+sorted_positives
```

桶排序

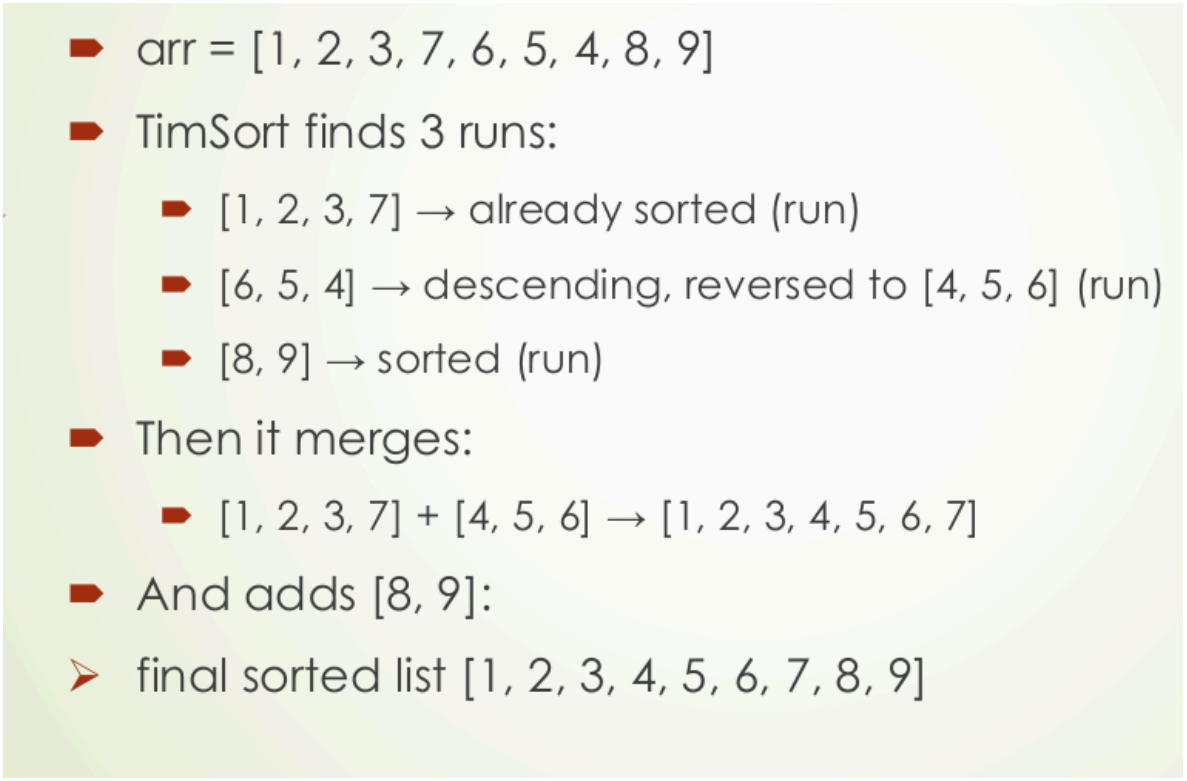
桶算法严格来说不能算单一的算法。思路就是按区间先把元素放到不同的数组里，然后在数组里使用其他算法排序，然后拼起来，比较适合均匀分布的数据（减少不必要的比较），对最坏的情况就跟没用桶算法差不多$O(n^2)$

```
def BucketSort(arr):
    if not arr:
        return arr

    bucket_count = len(arr)
    buckets = [[] for _ in range(bucket_count)]

    for num in arr:
        index = int(num*bucket_count)
        index = min(index, bucket_count-1)
        buckets[index].append(num)


    for i in range(bucket_count):
        buckets[i].sort()

    sorted_arr = []
    for bucket in buckets:
        sorted_arr.extend(bucket)
```

```
    return sorted_arr
```

Tim排序

Tim算法是Tim Peters专门为Python设计的算法，也就是默认的排序算法。它结合了归并排序和插入排序。具体做法是：先找一个已经排好的区间（Run），如果区间太短了就用插入排序强行弄长，依此类推搞出了多个排好的区间，然后使用归并排序来合并这些区间。具体的实现这里不要求，反正python里.sort就是。例子：



搜索与图

搜索问题就是我们有一个目标（比如说某个数），然后想要从数组（或是其他什么结构）里找到它的位置。

基础搜索

- Linear Search (equality comparison)
  - How it works: Scans each element one by one until the target is found.
  - Time complexity: Best:$O(1)$,Average: $O(n)$,Worst: $O(n)$
  - Space complexity: $O(1)$
  - Use case: Small or **unsorted** datasets. Multidimensional arrays.
- Binary Search (ordering comparison)
  - How it works: Divides the sorted dataset in half repeatedly to find the target.
  - Time complexity: Best: $O(1)$,Average: $O(\log n)$,Worst: $O(\log n)$
  - Space complexity: $O(1)$
  - Use case: Efficient for large, **sorted** datasets. Single dimensional arrays.

线性搜索

线性搜索是最简单的方法，遍历整个数组去对比。适用于少量的数据，支持多维数组

```
def LinearSearch(arr,target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

二分法搜索

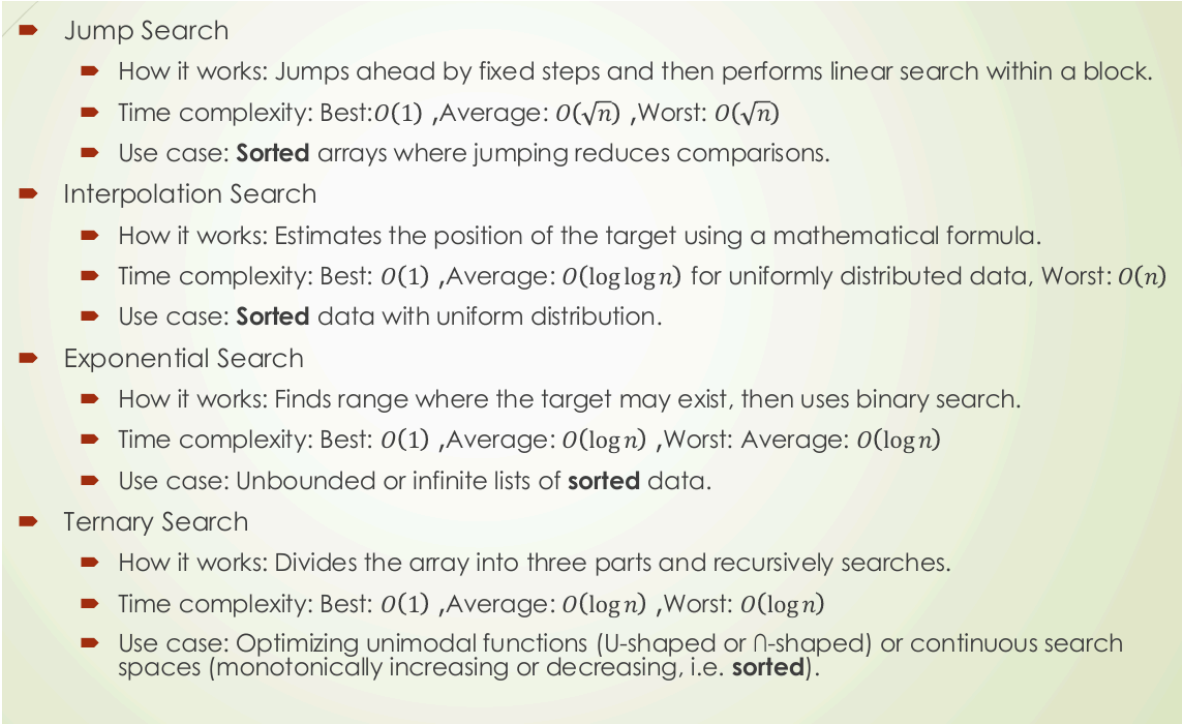
二分法适用于大量有序的数据，仅支持一维数组。首先选头尾，然后算出中间位置，跟目标比比大小，小了就在左边找，大了就在右边找，依此类推，递推和递归都能实现。

```
# Iteration
def BinarySearch(arr, target):
    low = 0
    high = len(arr)-1
    while low <= high:
        mid = (low+high)//2
        if arr[mid] == target:
            return mid
        elif arr[mid]<target:
            low = mid+1
        else:
            high = mid-1
```

```
    return -1

# %% recursion
def BinarySearch(arr, target, low, high):
    if low>high:
        return -1
    mid = (low+high)//2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return BinarySearch(arr,target,mid+1,high)
    else:
        return BinarySearch(arr,target,low,mid-1)
```

高级搜索

高级搜索全都仅支持有序数组。



三分搜索

原理跟二分差不多，就是从找中点变成了取三分点，效果嘛从$O(\log_2 n)$变成了$O(\log_3 n)$

```
def TernarySearch(arr, target, low, high):
    if low > high:
        return -1
```

```
    mid1 = low + (high-low)//3
    mid2 = high - (high-low)//3

    if arr[mid1] == target:
        return mid1
    if arr[mid2] == target:
        return mid2

    if arr[mid1] > target:
        return TernarySearch(arr,target,low,mid1-1)
    elif arr[mid2] < target:
        return TernarySearch(arr,target,mid2+1,high)
    else:
        return TernarySearch(arr,target,mid1+1,mid2-1)
```

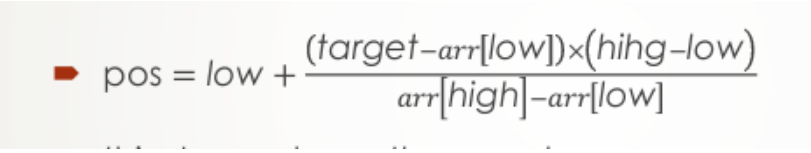
跳跃搜索

跳跃搜索要先根据数组大小找一个跳跃的长度（比如$\sqrt{n}$），然后在里面跳着找，找到目标所在区间之后使用搜索。这样时间复杂度就在两倍根号n了。

```
import math
def JumpSearch(arr,target):
    n = len(arr)
    step = int(math.sqrt(n))
    prev = 0
    while arr[min(step,n)-1]<target:
        # prev = current
        # current += step
        prev = step
        step += int(math.sqrt(n))
        print(step)
        if prev >= n:
            return -1

    for i in range(prev,min(step,n)):
        if arr[i] == target:
            return i
    return -1
```

插值搜索

插值搜索要先找两个头尾，然后利用插值估计目标位置，然后比较，小了在左边插值，大了在右边插值。有点像是翻版二分法，只不过不是简单除以2。插值公式：（其实是在利用线性插值估计target的位置，将target到low的数值差距转成指标差距）




```
# %% Interpolation search
def InterpolationSearch(arr,target):
    low = 0
    high = len(arr)-1

    while low<=high:
        pos = low + (target-arr[low])*(high-low)//(arr[high]-arr[low])

        if arr[pos] == target:
            return pos
        elif arr[pos] < target:
            low = pos+1
        else:
            high = pos-1
```

指数搜索

类似于跳跃搜索，只不过这次跳跃步长指数递增，所以对跨度很大的有序数组适用。然后在找到对应的区间之后用二分法。

```
# %% Exponential search
def BinarySearch(arr, target, low, high):
    if low>high:
        return -1
    mid = (low+high)//2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return BinarySearch(arr,target,mid+1,high)
    else:
        return BinarySearch(arr,target,low,mid-1)

def ExpoentialSearch(arr,target):
    if arr[0] == target:
```

```
        return 0
    i = 1
    while i < len(arr) and arr[i]<=target:
        i*=2
    return BinarySearch(arr,target,i//2,min(i,len(arr)-1))
```

图与搜索

树是有层次结构的，图则没有，只有节点之间的连接（Edges）。无向图的节点之间自然没有规定方向，有向图则有；加权图说明节点之间的连接被赋予的权重，无权图则只是简单的连接（或者认为权重全都相同）；循环图说明在途中包含环状结构，无环图则没有，注意有些图如果没有方向是循环图，但是加了方向之后就不是了，这被称为有向无环图（DAG）。

储存图的数据结构是字典，其键为节点，其值为一个列表，记录了跟相邻点的连接信息。图的格式如下：

```
class Graph:
    def __init__(self,nodes):
        self.nodes = nodes
        self.graph = {node:[] for node in nodes}
    def add_edge(self,a, b):
        self.graph[a].append(b)
        self.graph[b].append(a)

# %%
g = Graph([1,2,3,4,5])
g.add_edge(1,2)
g.add_edge(1,3)
g.add_edge(1,4)
g.add_edge(2,4)
g.add_edge(2,5)
g.add_edge(3,4)
g.add_edge(4,5)
```

我们可以基于图来实现一些搜索。

深度优先搜索

DFS，深度优先搜索，沿着连接一次遍历所有节点，直到没有连接，然后回头。用于寻路、迷宫等。可以用递推或者递归实现。（利用栈，前者显式地利用，后者隐式地利用）

```
# %% DFS
# iteration
def dfs(graph,start):
    visited = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            print(node)
            stack.extend(reversed(graph[node]))

# %% DFS: recursion
def dfs(graph,start,visited = None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)

    for neighor in graph[start]:
        if neighor not in visited:
            dfs(graph,neighor,visited)

graph = {'A':['B','C'],
         'B':['A','D'],
         'C':['C','E'],
         'D':['B'],
         'E':['C']}

#可视化

def dfs(graph,start,visited = None,path = None):
    if visited is None:
        visited = set()
    if path is None:
        path = []
    visited.add(start)
    path.append(start)

    for neighor in graph[start]:
        if neighor not in visited:
            dfs(graph,neighor,visited,path)
    return path

dfs_path = dfs(graph,'A')
G = nx.Graph()
```

```
for node, neighbors in graph.items():
    for neighbor in neighbors:
        G.add_edge(node, neighbor)

pos = nx.spring_layout(G)
nx.draw(G,pos,with_labels=True,node_size= 1000)
nx.draw_networkx_labels(G,pos)

for idx, node in enumerate(dfs_path):
    x,y = pos[node]
    plt.text(x,y+0.1, str(idx+1),fontsize = 12,color = 'red',ha = 'center')

plt.title('DFS Traversal Visualisation')
```

广度优先搜索

BFS，从一个节点出发，先搜索一个节点周边（即有连接）的节点，然后再找下一层。用于无权图最短路径寻找、社交网络等。通常使用递推而不推荐递归，因为对FIFO的队列，递推表现并不好。

```
# %% BFS
def bfs(graph,start):
    visited = set()
    queue = [start]
    visited.add(start)
    while queue:
        node = queue.pop(0)
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return visited
```

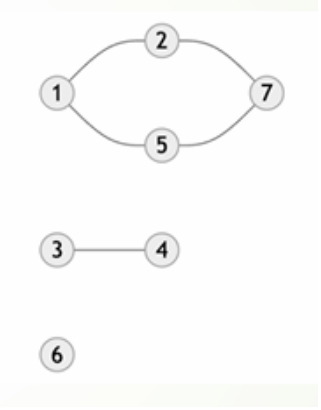
作为例子，寻找最短路径可以用BFS实现：

```
# %% shortest path
def bfs_shortest_path(graph,start,goal):
    visited = set([start])
    parent = {start:None}
    queue= [start]
    while queue:
        node = queue.pop(0)
```

```
        if node == goal:
            break
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
                parent[neighbor] = node
    if goal not in parent:
        return None
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path
```

连通分量

连通分量（Component）指的是图中的极大连通子图，或者说，这个图可以分成几个互不连通的部分。



DFS和BFS都可以用于查找图中的分量。

```
# %% Find components

def find_components(graph):
    visited = set()
    components = []
    for node in graph:
        if node not in visited:
            component = []
            dfs(node,visited,graph,component)
            components.append(component)
    return components
```

```
def dfs(node,visited,graph,component):
    visited.add(node)
    component.append(node)
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(neighbor,visited,graph,component)

# %% Find components
def find_components(graph):
    visited = set()
    components = []
    for node in graph:
        if node not in visited:
            component = bfs(node,visited,graph)
            components.append(component)
    return components


def bfs(start_node,visited,graph):
    visited.add(start_node)
    queue = [start_node]
    component = [start_node]
    while queue:
        node = queue.pop(0)
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
                component.append(neighbor)
    return component
```

对比与延伸

Feature	Depth-First Search (DFS)	Breadth-First Search (BFS)
Traversal	Explores as far as possible along each branch	Explores all neighbors level by level
Data Structure Used	Stack (explicit or via recursion)	Queue
Memory Usage	Lower if no deep recursion	Higher due to storing all neighbors
Shortest Path Guarantee	Not guaranteed	Guaranteed in unweighted graphs
Cycle Detection	Yes	Yes
Suitable For	Topological sort, maze solving	Shortest path, level-order traversal
Time Complexity	$O(V+E)$	$O(V+E)$
Space Complexity	$O(V)$	$O(V)$

- Nodes (vertices) - V
- Edges - E

图在大气中的应用在近几年都有，如：简化化学模型、比较反应机制或者改进天气预报、捕捉气候模型。

搜索算法的性能从以下几个角度考虑：

数据结构：根据搜索算法选择

数据量：对大数据，结合数据特征选择算法，尽量避免线性搜索

时间复杂度：哈希表、二分法等表现比线性好得多

空间复杂度：DFS在深层递归时，由于需要大量调用栈，所以内存占用大。具体来说：每当DFS访问了一个节点，它会递归自身，直到没有相邻节点，然后回溯，重复。如果图非常深，则递归调用次数将会非常庞大。

贪心算法、动态规划

贪心算法

每次操作都寻找局部最优解，以寻求最终能找到全局最优解。

具体算法：从空解出发，每一步使用贪心原则找最佳选项，然后将其添加到解集里，重复步骤知道问题解决或者没有更多可用选项。

但是一直找局部最优解并不总是能达到全局最优解，可能真的就停在局部最优解了。

例子：零钱兑换问题，用最少的硬币换零钱。如用[1,3,4]换六块，贪心算法就是从大的开始选（假设相比于小的，大的总能优化硬币数量），最后选出[4,1,1]，但是这里的全局最优解是[3,3]。

```
# %% Greedy algorithms
def greedy_coin_change(cost,payment):
    target_change = int((payment-cost)*100)
    # target_change = (payment-cost)*100
    coins = [1, 2, 5, 10, 50]
    coins.sort(reverse=True)
    result = {}
    for coin in coins:
        count = target_change//coin
        # # alternative
        # count = 0
        # while target_change>=coin:
        #     target_change -= coin
        #     count +=1

        result[coin] = count
        target_change -= coin*count
        print(target_change)
        if target_change == 0:
            break
    return result
# %%
greedy_coin_change(9.32,10)
# %%
def greedy_coin_change(coins, target):
    coins.sort(reverse=True)
    result = []
    for coin in coins:
        while target>= coin:
            target -= coin
            result.append(coin)
    return result
coins = [1,2, 5, 10, 50]
target = round((10-9.32)*100)
greedy_coin_change(coins, target)
```

另一个例子是活动选择，给定一日内诸活动的时间区间，从中选出最大的子集，并且其中每个区间没有重叠。

```
# %% avitivity selection
courses = [(1, 4), (3, 5), (0, 6), (5, 7), (8, 9), (5, 9)]

def activity_selection(courses):
```

```
    courses.sort(key = lambda x: x[1])
    selected = []
    last_end_time = 0
    for start,end in courses:
        if start>=last_end_time:
            selected.append((start,end))
            last_end_time = end
    return selected
```

由于不能遍历每种组合所以有可能错过最优解。

所以我们需要动态规划。

动态规划

动态规划（DP）本身是在保证全局最优（当然，需要遍历）的情况下来减少重复计算的次数。具体来说是将大问题分解为有所重叠的小问题。

动态规划的流程：先从问题本身找到子问题的结构，然后寻找递推关系，保存结果，并从容易解决的最简情况往后推出结果。具体来说有两种，一是从高到低（Top-Down），使用递归直到最简情况；另一种是从低到高（Bottom-Up），从最简情况出发向上够建解直到我们需要的情况。

例1：斐波那契数列。这里是TD，即我们递归最开始不需要知道前一项/两项是什么，只消规定最初的状态然后一直分解问题递归就行。

```
def nth_fibonacci_memo(n,memo={}):
    for n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = nth_fibonacci_memo(n-1,memo)+nth_fibonacci_memo(n-2,memo)
    return memo[n]
```

例2：硬币兑换问题。这里是BU，先给出基本状态然后往上找。这里的两个遍历，即遍历硬币和金额哪个在外面都可以，这里是硬币在外，然后对每个金额，如果金额-当前硬币已经存在了，则取金额-当前硬币那个解加上当前硬币，然后这个解如果比原来的更优，则替换原来的解。由于可以保证最简单的情况是最优解，所以后续每个解都必定是最优解。

```
# %% bottom-up
# coin change

def DP_coin_change(coins,target):
```

```
    dp = [float('inf')]* (target+1) # list of minimum number of coins
    dp[0] = 0 # base case

    result = [[] for _ in range(target+1)]
    for coin in coins:
        for i in range(coin, target+1):
            if dp[i-coin]+1<dp[i]:
                dp[i] = dp[i-coin]+1
                result[i] = result[i-coin] + [coin]
    if dp[target] == float('inf'):
        return -1, []
    return dp[target],result[target]

coins = [1,3,4]
target = 6
# %% dict

def DP_coin_change_dict(coins,target):
    # dp[amount ('target of subproblem')] = (#_of_min_coins,combination_list)
    dp = {0:(0,[])}
    for coin in coins:
        for amount in range(coin,target+1):
            if amount-coin in dp:
                prev_count, prev_comb = dp[amount-coin]
                new_count = prev_count+1
                new_comb = prev_comb + [coin]
                if amount not in dp or new_count < dp[amount][0]:
                    dp[amount] = (new_count,new_comb)
    return dp, dp.get(target,[-1,[]])
```

例3：活动选择。这里增加了成本考虑。这里先按活动结束时间排序，然后考虑前i+1个活动的最优解dp[i]，dp[i]包含(最大活动数量，最小成本，活动索引)。得到dp[i]的方法：我们搜索在第索引i个活动开始之前结束最晚的活动l，所以可以有dp[i]=dp[l]+i；然后考虑如果不加入第索引i个活动，则结果继承自dp[i-1]，比较dp[l]+i与dp[i-1]哪个更优。

```
# %% activity selection

class Activity:
    def __init__(self, start, end, cost):
        self.start = start
        self.end = end
        self.cost = cost

def binary_search_latest(activities, index):
    low = 0
```

```
    high = index - 1
    while low <= high:
        mid = (low + high) // 2
        if activities[mid].end <= activities[index].start:
            if mid + 1 <= high and activities[mid + 1].end <=
activities[index].start:
                low = mid + 1
            else:
                return mid
        else:
            high = mid - 1
    return -1

def max_actitvity_min_cost(activities):
    activities.sort(key=lambda x: x.end)
    n = len(activities)
    # dp[i] = (count of activities, total cost, chosen activities)
    dp = {0: (1,activities[0].cost,[0])}

    for i in range(1,n):
        incl_count = 1
        incl_cost = activities[i].cost
        incl_list = [i]
        l  = binary_search_latest(activities,i)
        if l != -1:
            prev_count, prev_cost, prev_list = dp[l]
            incl_count += prev_count
            incl_cost += prev_cost
            incl_list = prev_list + [i]

        excl_count,excl_cost,excl_list = dp[i-1]

        if (incl_count>excl_count) or (incl_count==excl_count and
incl_cost<excl_cost):
            dp[i] = (incl_count,incl_cost,incl_list)
        else:
            dp[i] = (excl_count,excl_cost,excl_list)

    return dp[n-1]
    # dp[2] = best solutions using activites 0,1,2


activities = [
    Activity(1,4,5),
    Activity(1,3,4),
    Activity(3,5,6),
    Activity(0,6,5),
    Activity(0,2,3),
```

```
    Activity(5,7,7),
    Activity(5,8,11),
    Activity(7,9,2)
]
```

综上，我们可以对比贪心算法和动态规划

Feature	Greedy Algorithms	Dynamic Programming (DP)
Strategy	Make the best local choice at each step	Solve subproblems and build up to the final solution
Backtracking	No (once a choice is made, it's final)	Yes (considers multiple options and stores results)
Optimal Guarantee	Only if the problem has greedy-choice property	Always finds the optimal solution
Efficiency	Usually faster and simpler	Can be slower and more complex
Memory Usage	Low (often constant or linear)	Higher (stores solutions to subproblems)

有向图（不考！）

前面我们已经介绍过有向图了。具体实现就是加入连接的时候只加一边不加另一边，即（b-a）加入而（a-b）不加入。

拓扑

拓扑是将图的复杂结构线性化。为了找到拓扑，使用DFS是非常合适的：

```
# %% Topological ordering

class TopologicalSort:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in self.nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)
    def visit(self, node):
        if self.state[node] == 1: #如果正在搜索
            self.cycle = True  #说明是环
            return #下一个
        if self.state[node] == 2: #如果已经搜索完成了
            return  #下一个

        self.state[node] = 1  #如果尚未搜索，定义为正在搜索
        for next_node in self.graph[node]:  #对该节点的所有连接
            self.visit(next_node)  #递归
```

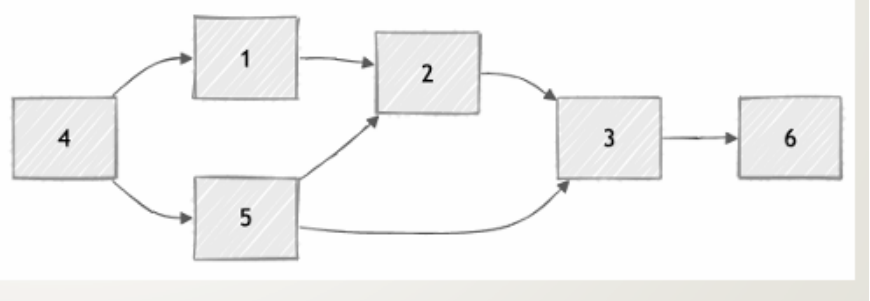
```
        self.state[node] = 2  #检索完了
        self.order.append(node)  #存储节点

    def create(self):
        self.state = {}
        for node in self.nodes:
            self.state[node] = 0 #赋予状态：未搜索

        self.order = []
        self.cycle = False

        for node in self.nodes:  #遍历节点
            if self.state[node] == 0:  #如果未搜索
                self.visit(node)  #搜索
        if self.cycle:  #如果有环
            return None  #返回空值
        else:  #否则
            self.order.reverse()  #由于是逆储存的，所以需要反转
            return self.order
```

对于无环图，其在拓扑上的顺序可能有不止一个（比如上面的代码就只提供力一种）。

无权有向图寻路

一个自然的问题就是从a到b有几种道路？最近的和最远的道路分别是什么？

数道路使用的是动态规划，比如查找1到3，我们可以先查找3之前的节点有哪些道路，依此类推

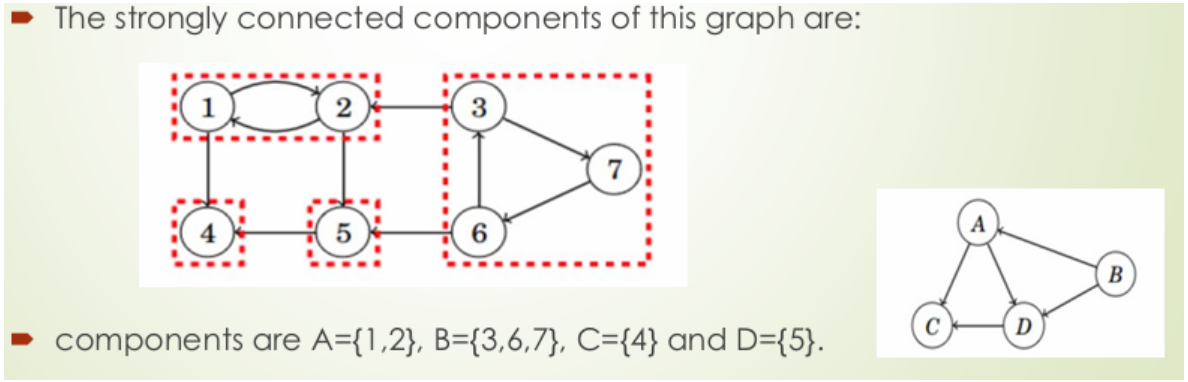



```
# %% DP
class CountPaths:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in self.nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)
```

```
    def count_from(self,node):
        if node in self.result:
            return self.result[node]
        path_count = 0
        for next_node in self.graph[node]:
            path_count +=self.count_from(next_node)

        self.result[node] = path_count
        return path_count
    def count_paths(self,x,y):
        self.result = {y:1}
        return self.count_from(x)
```

强连通

强连通：任意节点到任意节点都有道路。强连通分量（SCC）：



Kosaraju算法：

检索图中的强连接分量。由于逆图与原图有相同的强连通分量，所以我们可以：先在原图进行DFS，构建出所有节点的列表；然后将每条边反过来，再进行DFS

```
# %% Kosaraju for SCC identification
class Kosaraju:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in self.nodes}
        self.reverse = {node: [] for node in self.nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)
        self.reverse[b].append(a)

    def visit(self, node, phase): # Performs a depth-first search (DFS).
```

```
        if node in self.visited:
            return
        self.visited.add(node)

        if phase == 1:
            graph = self.graph
        if phase == 2:
            graph = self.reverse

        for next_node in graph[node]:
            self.visit(next_node, phase)

        if phase == 1:
            self.order.append(node)

    def count_components(self):
        self.visited = set()
        self.order = []

        for node in self.nodes:
            self.visit(node, 1)

        self.order.reverse()
        self.visited.clear()

        count = 0
        for node in self.order:
            if node not in self.visited:
                count += 1
                self.visit(node, 2)

        return count

k = Kosaraju([1, 2, 3, 4, 5, 6, 7])

k.add_edge(1, 2)
k.add_edge(1, 4)
k.add_edge(2, 1)
k.add_edge(2, 5)
k.add_edge(3, 2)
k.add_edge(3, 7)
k.add_edge(5, 4)
k.add_edge(6, 3)
k.add_edge(6, 5)
k.add_edge(7, 6)

print(k.count_components())
```

有向图

加权有向图

加入权重之后，寻路问题会更复杂一些。除此之外还有所谓的最大流问题。（因为不考我就懒得补完了，这些东西ppt里或者网上都有。。。）

最短路问题

最大流问题