注意看师兄给我的三本书。

算法图解
剑指offer
我自己在网上找的教程

本部分内容对算法部分进行总结。分为下面三大块：一般算法、机器学习算法、深度学习算法。

一般算法

算法的一般知识：算法复杂度O

1. **divide and conquer(二分法)**
   复杂度：$O(logn)$
2. **选择排序**
   计算机内存犹如一大堆抽屉。

- 基数数据结构：数组（一块连续内存空间，方便访问和查找（支持随机访问，$O(1)$），顺序插入和删除$O(n)$）和链表（离散空间，便于插入和删除（$O(1)$），顺序访问和查找$O(n)$）
- **选择排序**算法：一次找到最大/小的数，再从剩下的$n-1$个数中再找到最大/小，直到最后一个数字；复杂度$O(n^2)$.
- **优缺点**：使用灵巧，适合多种数据结构，但是效率不高
  代码示例： 过于简单，不写。

概念进阶

递归-优雅的解决方案

能用递归的一般可以用循环实现，递归在于优雅，循环可能性能上占优（使用递归，其数据全部叠加在栈中，对内存要求很高），看你喜欢哪种。

- 如果要提高效率，办法1：转换成循环；办法2：使用**尾递归**。
- 写递归函数时候注意基线条件(函数不再调用自己的条件)和递归条件（函数调用自己的条件），以避免程序陷入死循环。
  $note$:

1） 《算法图解》详细讲解了递归的**调用栈**的过程，记住这个；
2） 栈的两种操作：压入和弹出；
3） 所有的函数都进入调用栈；

快速排序

快速排序的隐形思维是“分而治之”，就是divide and conquer。这里的divide不是狭义的分割（如列表），而是一种缩小规模的抽象方法。上述的递归、二分法，其基本策略就是D&C。在《算法图解》中，**回忆分田和数组加法问题。**
快速排序的要点：
1） 基本思路

- 分区（基准值，左边比基准值小，右边比基准值大）
  2） 选择基准值和复杂度（原断点：code，困了）

```
def quicksort(array):
    if len(array) < 2:
        return array
    else:
        pivot = array[0]
        less = [i for i in array[1:] if i <=pivot]  #这个语句要学会！很简单
        greater = [ i for i in array[1:] if i >pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
print(quicksort([10,5,2,3]))
```

有下面两点需要注意：

1. 大O表示法的常量问题：常量，即单位运行时间，合并排序（ merge sort）和快速排序从大O表示而言一样，但是常量是前者大，所以后者为优；
2. 平均情况和最糟情况：明显，对一个顺序数列且使用第一个元素作为基准值，复杂度为$O(n*n)$，这是最糟情况；若是取中位数作为基准值，复杂度为$O(n*logn)$，这是最优情况。这里要注意，复杂度计算的时候是每层复杂度*层数，基准值直接决定层数的多少，此外，快速排序的平均情况就是最优情况（假设数列随机排列）。

散列表

哇！这个数据结构有点儿有趣！

其内秉的逻辑是这样的：将输入，比如“苹果”，通过一个函数，这里成为散列映射函数（满足函数一一对应的定义），将这个输入直接映射到计算机的储存地址。散列表也被称作散列映射、映射、字典（python中，用dict表示）和关联数组。
因为python中，我几乎没有用到过字典，这里特意写下字典的文法：
book = dict() %创建一个名为book的空字典
book["apple"] = 0.67 %字典由键和值组成，这个例子中，键为apple，行为0.67
book["milk"] = 1.02
1） 散列表的应用
手机里的通讯录，名字和手机号码，就是一个例子！对于大海捞针式的**搜索**，就用散列表，即使是高效的二分法，也无法比拟散列表的优势，比如**网址和IP的对应。**
散列表的还有两个用途：**防止重复**和**用作缓存。**举个例子，下面的code：

```
voted = { }
def check_voter(name):
    if voted.get(name):
        print( " kick them out!  " ) #不好意思，我不厚道地笑了，哈哈哈
    else:
        voted[name] = True
        print( " let them vote! " )
```

对于后者，书本上用了FACEBOOK网页为例子。试想象，你打入一个网址（向facebook服务器发出请求），服务器需要做一些处理（花时间），生成一个网页给你。网页生成可能需要搜集你的个人信息，为你定制网页，进而花费时间比较长，这是你不愿意等待的。这时，如果服务器已经经过你上次操作信息为你提前做好网页，记住网页内容，这样当你发出请求时，它可以直接把这个网页给你（这个网页其实就是缓存），这样就快了。

```
cache = { }
def get_page(url):
    if cache[url]:
        return cache[url] #当catch[url]有缓存数据，则返回这个数据；
    else:
        data = get_data_from_server(url)
        cache[url] = data
        return cache[url] #当catch[url]没有缓存数据，则先把这个urld的网页数据
```

注意上面的code是可以更精简的，这里不做了。

2) 散列表注意点

散列表的性能和散列映射函数很相关！映射函数可以简单的按照首字母排序，也可以是SHA函数。前者，是糟糕的函数，比如apple，banana（这两个位置的确不一样），但是加入avocado，就和apple**冲突**了。这是，可以在这个位置引入一个链表，如果冲突不严重，速度下降可以接受，但是如果函数糟糕，冲突严重，考虑链表的$O(n)$查找，恐怖如斯！

书本中引入一个“**填装因子**”表述预期冲突程度，定义为：散列表元素个数/可以安排的位置数。**填装因子为1且散列函数优使得满射，则效果最优。**如果填装因子过大（一般设定**经验界限是0.7**，这个值下，发生冲突的可能性比较低），需要对数组进行“调整长度”。

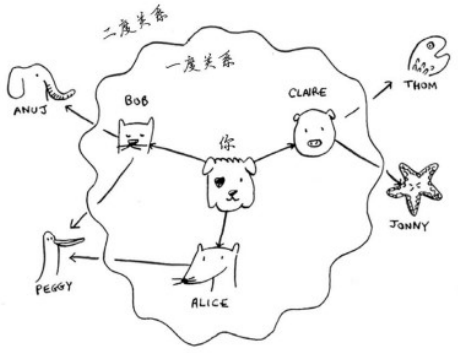
散列函数可以千变万化（自己也可以定义吗？），一个良好的函数是可以尽量让元素在这个函数映射下均匀分布。

算法进阶

下面介绍：广度优先搜索

广度优先搜索

这里出现“**图**”的概念，但是不管怎样，最让我惊讶的是**队列+散列表**竟然能够完成图查找算法。想起来大学时候傻得冲天的自己还在循环套循环，亏得没有发表，否则会成为自己科研的耻辱（掩面）。



图示：一个有向图，边上有箭头

我们先从一个问题出发，看看光度优先搜索能够完成什么任务（答案：**有无路径和最短路径**，关于最短路径，指经过的节点最少，要区分加权图中的**最快路径**）。

我（上图的你）上我的脸书找香蕉供应商。我想一度关系优于二度关系，越高度关系越次。广度优先搜索会先搜索我的一度关系朋友，依次向外辐射，直到找到供应商或者找遍所有朋友（如脸书只能找到二度关系朋友）。

队列：先进先出；对比栈：先进后出。按顺序加入队列，先将全部一度关系朋友加入队列，再后队列后依次加入二度朋友，这样就可以实现低度关系的人优先查找。

下面用散列表构造图，如此优雅的数据结构！图的各个**节点**之间的关系（**边**）会被自动识别！

```
'''
广度优先算法
'''
graph = {}
graph["you"] = ["alice","bob","claire"] #即you键的值是一个list
graph["bob"] = ["anuj","peggy"]
graph["alice"] = ["peggy"] #注意和bob好友有重复，后面编程时候要注意
graph["claire"] = ["thon", "jonny","tom"]
graph["anuj"] = []
graph["peggy"] = []
graph["thon"] = []
graph["jonny"] = []
graph["tom"] = []

from collections import deque

def search(name):
    search_queue = deque()
    search_queue += graph["you"]
    searched = []
    while search_queue:
        person = search_queue.popleft()
        if not person in searched: #互为好友情况下会造成循环
            if person_is_seller(person):
                print(person + " is a mango seller ")
                return True
            else:
                search_queue += graph[person]
                searched.append(person)
    return False
def person_is_seller(name):
    return name[-1] == 'm'
search("you")
```

广度优先算法的时间复杂度为：$O(V+E)$。其中E为边数，V（vertice）为顶点数。另外，因为涉及到一些队列操作，需要自己默写代码，基础有些渣，只能笨方法上

了。

狄克斯特拉算法

不好意思，我看成狄拉克算法了，正统物理生，我骄傲，哈哈。**这个算法的学习，让我明白应该用最好的状态学习任何的东西，不要想着要多次重复就在状态不好的时候看，这样只会增加误解，浪费时间。**

时间过了三四个礼拜，我现在再来看，还是狄拉克算法，真的没脸见人了。来，是狄-克-斯-特-拉-算法。

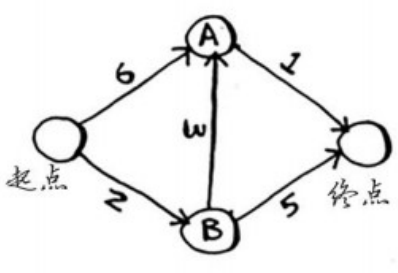
这次我们仍然是对**图**的研究，只不过这次是**加权图**，应用的场景可以是最短时间换车方案的选择。也可以认为是节点之间的路径具有长度。我想到了一个损招，如果按照最小的路径为单位，把具有大路径的节点之间假想成多个节点，且这些假象节点之间的路径加起来就是原两节点之间的路径，然后套用广度优先搜索的算法，不也可以做吗?

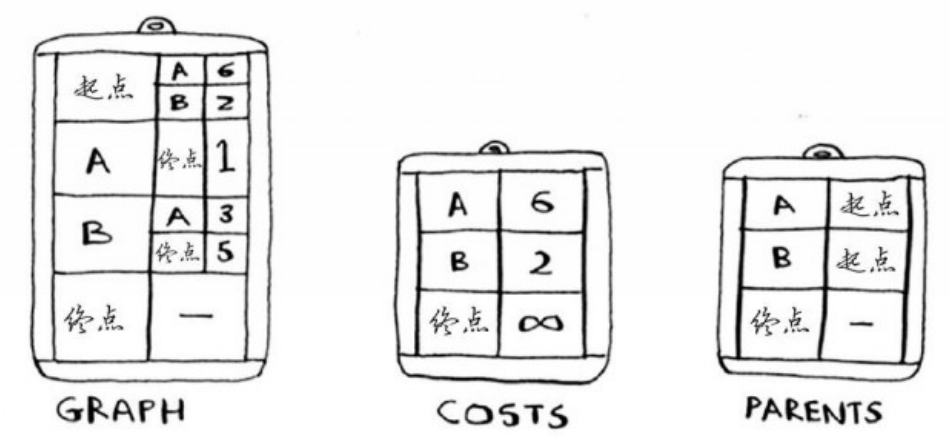
当然可以，我的方法是思想偷懒了，而且对于不能拆成整数的情况下，就不适用了。我们需要更加聪明的方法。

注意，该方法不适用于有向无环图；也不适用与出现负权重的图，这时候应该用“**贝尔曼-福德算法（Bellman-Ford algorithm）**”。原因在于1和3中的黑体部分，导致原先的节点不会被更新。

D算法的核心算法：

1. 找出全局最便宜的节点；**在D中，没有比不花费任何费用的节点更近了**；
2. 计算**经过**该节点，前往它的邻居的开销；如果某些邻居有更便宜的开销，更新其开销。所以这一步算的一直都是从起点到这些邻居的开销；
3. 重复1、2两步，直到图中**剩余的**所有的节点都遍历了；
4. 计算最终的路径。






```
'''
我们需要三个散列表，分别记录原始图、起点到其他节点的开销以及节点-父节点图，后两个
python用dict提供散列表，例子：
{'avocado': 1.49, 'apple': 0.67, 'milk': 1.49}
'''
#原始图，见上图2号（从左到右）；
graph = {}
graph ["start"]= {}
graph["start"]["a"] = 6
graph["start"]["b"] = 2
graph["a"] = { }
graph["a"]["fin"] = 1
graph["b"] = {}
graph["b"]["a"] = 3
graph["b"]["fin"] = 5
graph["fin"] = {}
#下面创建开销表格
infinity = float("inf")
costs = {}
costs["a"] = 6
costs["b"] = 2
costs["fin"] = infinity
#下面为父节点图创建
parents = {}
parents ["a"] = "start"
parents["b"] = "start"
parents["fin"] = None
#另一个数组存放已经处理过的节点
processed = [ ]
#下面开始正式的代码
```

```python

#下面定义过程1 ，找到没被遍历过的最小开销的节点
def find_lowest_cost_node(costs):
    lowest_cost = float("inf")
    lowest_cost_node = None
    for node in costs:
        cost = costs[node]
        if (cost < lowest_cost) and (node not in processed):
            lowest_cost = cost
            lowest_cost_node = node
    return lowest_cost_node

node = find_lowest_cost_node(costs) #过程1
#因为涉及到散列表里面嵌散列表，写变量的时候注意具体指的啥。
while node is not None: #如果存在满足条件的最小开销节点
    cost = costs[node]
    neighbors = graph[node]
    for n in neighbors.keys(): #现在n指邻居节点
        new_cost = cost+neighbors[n]
        if costs[n] > new_cost:
            costs[n] = new_cost
            parents[n] = node
    processed.append(node)
    node = find_lowest_cost_node(costs)
#看一下结果会是什么。注意这两个散列表是一直都在变化的，最后结果包含了最优结果。
print(parents)
print(costs)

#下面把路径和使用的时间数打印出来，注意list的reverse的应用，这是地址上直接改的，
def print_path(parents):
    path_re = ["fin"]
    while True:
        next_parents = parents[path_re[-1]]
        path_re.append(next_parents)
        if next_parents == "start":
            break
    for item in reversed(path_re):
        print(item,end='')
        if item != "fin":
            print('->',end='')

```

```
71
72 print_path(parents)
73
74 print("\n","The optimal path needs",costs["fin"],'minutes.')
```

为了让路径输出更具有可读性，就写了print_path，原先书本里是没有的。

贪婪算法

涉及到NP概念，突然觉得高大上了不少。
贪婪算法的核心思想在于：**每一步都走最优解（局部最优解），则可以得到最终的全局最优解。**很显然，这样做是并不能次次奏效，因为逻辑上就存在问题，但是对于一些“找最优解只能遍历且遍历的计算成本极大”的情况下，贪婪算法算是比较优秀了。
毕竟我们不可能对最优解都在意，很多时候，有一个近似的结果就可以了。
在书本中，举例“教室调度问题”，就希望把最早下课的课排上，后续同样操作；在“背包问题”中，对于体积一定的背包，小偷先拿价值最高的，而后拿能放下书包的价值最高的东西，这种“贪婪”算法肯定是不行的。让我来想，应该把每件东西折算成“单位体积价值”，再放入包中，才是合理的。**（下面是我的想法）从这里，是不是可以发现贪婪算法成功与否的决定性因素：最优化目标和条件不重合。**
书本中的“集合覆盖”问题让我更加在意。（注意，集合覆盖问题，是一类问题，如果懂得建模，你可以将教练选择足球运动员的问题抽象为集合覆盖问题）。
问题描述：全美有非常多个广播台（假设n个），各个广播台有自己的覆盖州，我们想广播使得全美国人都能够听到，但是广播台需要付费（假设费用一样），我们希望能够找到一个包含五十个州的最小广播数目集合。
按照集合理论，n个元素有2^n个集合，我们需要遍历这些集合，找到满足上述条件的集合。
用贪婪算法：

1. 选出这样一个广播台，即它覆盖了最多的未覆盖州。即便这个广播台覆盖了一些已覆盖
   的州，也没有关系。
2. 重复第一步，直到覆盖了所有的州。
   用集合的数据类型，能够很见到完成上面的算法。目前我不写了，**mark一下，后面有空再写下来。**

```
1 #python里面的集合
2 a = set()
```

```
a.add('a')
a.add('b')
a.add('a')
a.add(2)
print(a) #{2, 'b', 'a'}
thisset = set(("google","taobao","facebook"))
thisset.add("runoob")
print(thisset) #{'facebook', 'runoob', 'google', 'taobao'}
thisset.update("yahoo")
print(thisset) #结果令我惊讶，竟然是{'runoob', 'h', 'google', 'o', 'taobao',

Thisset = thisset#打印这一个Thisset和上面的记结果一样，但是
#Thisset = thisset.update("huahua")#这一行，运行print时候，出现None
print(Thisset)
Thisset.remove("y")#Thisset.discard("y")也可以，且如果没有括号内元素，不会报
print(Thisset)
Thisset.pop()#thisset.clear(),清空元素，使集合变量为空集；
print(Thisset)#{'h', 'google', 'o', 'taobao', 'a', 'facebook'}
a = {i for i in Thisset if i not in "oah"}
print(a) #{'facebook', 'google', 'taobao'}
#这里注意，python中的set,如a = set("abcdbcduji"),会把字符拆开来

print("facebook" in Thisset)#True
aa = set('abracadabra')
bb = set('alacazam')
print(aa)
print(bb)
print(aa-bb) #{'r', 'b', 'd'}#aa中的元素去掉bb中有的；
print(bb-aa) #{{'m', 'l', 'z'}} #bb中的元素去掉aa中有的；和上面的一个不一样的
print(aa|bb) #{'r', 'l', 'b', 'z', 'c', 'd', 'a', 'm'}
print(aa ^ bb) # 不同时包含于a和b的元素
```

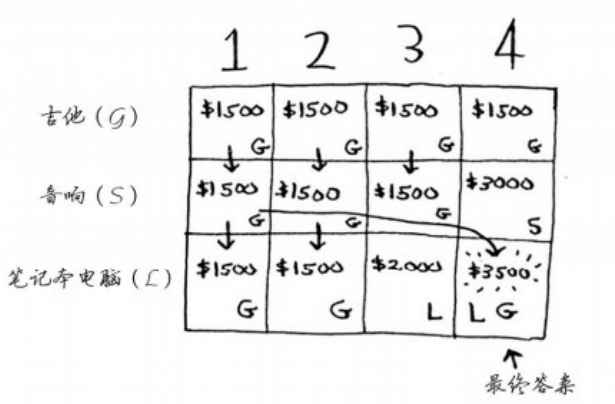
对于旅行者，两个城市，有两条路（这里假设A到B城和B到A城的路线不同），三个城市，则是$3 * 2! = 6$，四个城市为$4 * 6 = 24$，所以N个城市，就是$N!$。阶乘，爆炸性增长，全部遍历，对计算机是不小的挑战，甚至是不可能完成的任务。

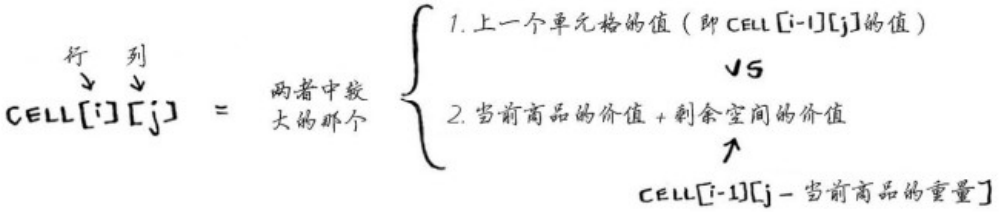
旅行商问题和集合覆盖问题有一些共同之处：你需要计算所有的解，并从中选出最小/最短的那个。这两个问题都属于NP完全问题。对于NP完全问题，我们不用费劲心思去想怎么解决了（因为只有遍历的方法），只要找到一个近似解就可以。所以**识别一个问题是不是NP完全是非常重要的。对于可以转换为“集合问题”和“旅行商问题”这种**

涉及完全组合的问题，那肯定是NP完全；对于涉及序列、难以分解成或者计算成本随元素数量激增的，则很有可能是NP完全。
对于前面的找最短距离，如果我们指定必须必须经过的城市，则就是NP完全了；再D算法中，不是NP完全。

动态规划

距离看完这本书只有50页了，我后面浏览了后面内容，发现实战部分少了很多。考虑到还有一些其他东西需要看，我今晚一口气看完。
我们说过，对于背包问题，贪婪算法可以给出一个近似且令人满意的结果，但是我们还能做得更好！用**动态规划**！
首先重复那个问题，核心在于一个约束条件（背包的称重量），一个优化目标（包内物品价值最高），我们应该怎么装?
动态规划的问题，关键点在于：**分解问题，使之可以分解成独立的可优化部分。**



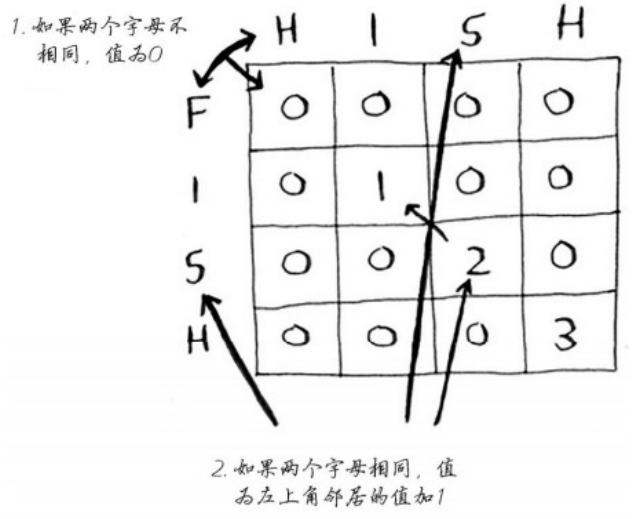


注意，如果你想投一个0.5磅的项链，这个格子就不是1磅，而是0.5磅了。这可以理解，但是如果我们的最小单位是0.3磅呢？那就0.3吧，不用管除不尽这种问题。比较大小时候会自动考虑到的。
背包问题的变式，可以是旅游问题（给定天数，数个旅游点，旅游点的评分），这里，如果只是想旅游更多的地点，用贪婪算法就可以了，但是，如果想最终的满意度（旅游地点的评分之和）最大，则可以用动态规划。同样的画格子的方法，十分有效。从旅游问题，我们想到，可能各个景点之间存在依存关系，比如同一个城市的旅游，就会缩小旅游时间，注意，**动态规划**不适用这种问题！我们注意：

> **仅当每个子问题都是离散的，即不依赖于其他子问题时，动态规划才管用。**

面对文字编辑，比如打字的时候，我“hish”，糟糕写错了！但是现在的输入法知道，一

般会自动排除错误的输入，比如会给出“fish”、“vista”等单词之间比较，那排在第一位的，是哪一个？这里也可以用动态规划的方法，一般我们称作这个问题为**最长公共子串**，更进一步的是**最长公共子序列**。下面看一个这两个问题的网格方法：

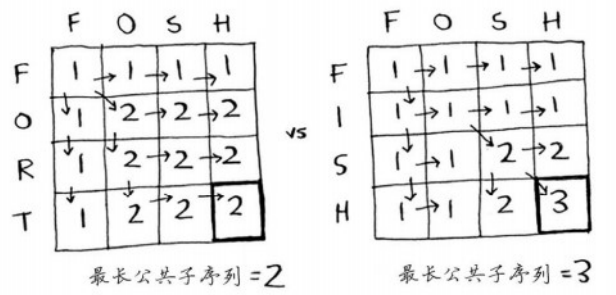


实现这个公式的伪代码类似于下面这样。

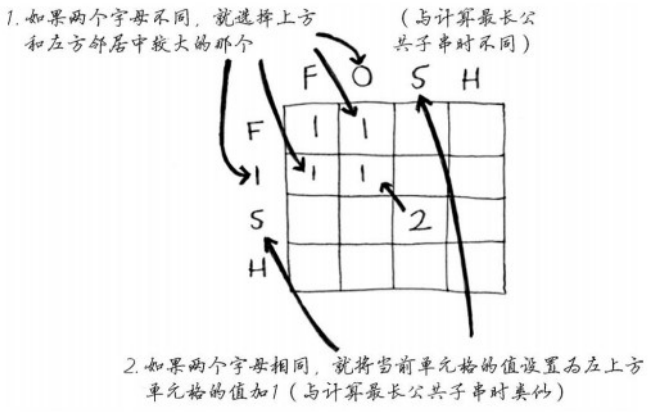
```
if word_a[i] == word_b[j]:      <-- 两个字母相同
  cell[i][j] = cell[i-1][j-1] + 1
else:                           <-- 两个字母不同
  cell[i][j] = 0
```

啊啊啊，好难看的截图...但是这里没有办法，这样子最直观！注意，上面的是最长公共**字串**。

最终的网格如下。



下面是填写各个单元格时使用的公式。




```
if word_a[i] == word_b[j]:      <-- 两个字母相同
  cell[i][j] = cell[i-1][j-1] + 1
else:                           <-- 两个字母不同
  cell[i][j] = max(cell[i-1][j], cell[i][j-1])
```

这上面是最长公共子序列。生物DNA找相似片段，也可以用动态规划呀！

K最邻近算法

在《机器学习》看过这个，因为想法比较简单，就没有上级操作。看了这本书的这个内容，我印象最深的，是**特征**的选择和**距离**的选择，前者是分类的最基本要素，后者决定分类的效率。
K最邻近算法在电影/音乐等有推荐功能的软件上应该用得不少。比如判断一个水果是哪个种类的。推荐电影时候，可以将用户看得多的电影或者直接让他们对各个类型电影喜爱程度打分的方式，判断这些用户中，哪些用户具有相似的电影品味，这样，如果在同一个群体中，有一个电影这些群体的人都喜欢，就可以直接推送给群体中还没看过这个电影的人。如果这个群体中人数过于多，则可以象征性地选取前几个和这个用户最像的用户的看片历史，再推送。一般不要每次都这么做，如果我们的推送精度不需要很高的话，可以直接把他们堪称一个人...当然，这样，是不是不太好，哈哈哈。
距离的选择，余弦相似度（不是普通的绝对距离，而是角度）

> 欧氏距离是最常见的距离度量，而余弦相似度则是最常见的相似度度量，很多的距离度量和相似度度量都是基于这两者的变形和衍生，所以下面重点比较下两者在衡量个体差异时实现方式和应用环境上的区别。欧氏距离能够体现个体数值特征的**绝对差异**，所以更多的**用于需要从维度的数值大小中体现差异的分析，如使用用户行为指标分析用户价值的相似度或差异**；而余弦相似度更多的是从方向上区分差异，而对绝对的数值不敏感，更多的用于**使用用户对内容评分来区分用户兴趣的相似度和差异，同时修正了用户间可能存在的度量标准不统一的问题**（因为余弦相似度对绝对数值不敏感）。

接下去该怎么做?

后面讲了：

1. 树，比如二叉树查找，还有一堆高级的数据结构：B树、红黑树、堆、伸展树等；
2. 反向索引：搜索引擎中常用；
3. 傅里叶变换：歌曲，频率分析，音乐识别软件，信号分析，也有地震预测，甚至DNA分析；
4. 并行算法：海量数据，分布式计算，SQL并不能对海量数据进行查询和索引；映射函数、归并函数、布隆过滤器（一种概率型数据结构）和 HyperLogLog。

面临海量数据且只要求答案八九不离十时，可考虑使用概率型算法！

5. 散列算法：SHA算法，比较文件，安全散列算法，密码安全问题
6. 局部敏感的散列算法
7. Diffie-Hellman 密钥交换：如何对消息进行加密；
8. 线性规划：**所有的图算法都可使用线性规划来实现。线性规划是一个宽泛得多的框架，图问题只是其中的一个子集。**这是多么诱人的算法！