

《算法图解》学习笔记

下面是常见数组和链表操作的运行时间。

	数组	链表
读取	$O(1)$	$O(n)$
插入	$O(n)$	$O(1)$
删除	$O(n)$	$O(1)$

编写递归函数时，必须告诉它何时停止递归。正因为如此，每个递归函数都有两部分：基线条件（base case）和递归条件（recursive case）。递归条件指的是函数调用自己，而基线条件则指的是函数不再调用自己，从而避免形成无限循环。

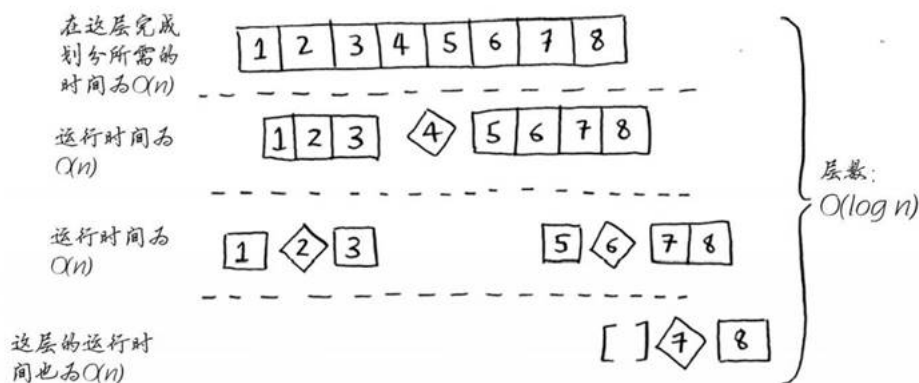
我们来给函数countdown添加基线条件。

```
def countdown(i):  
    print i  
    if i <= 0:  ← 基线条件  
        return  
    else:  ← 递归条件  
        countdown(i-1)
```

虽然快排和归并排序的算法复杂度相同 $O(n \log n)$ ，但快排的常量 c 更小，因此快排更快。

快排在最优情况下，栈长为 $O(\log n)$ 。完成一次划分需要遍历一遍[即 $O(n)$]

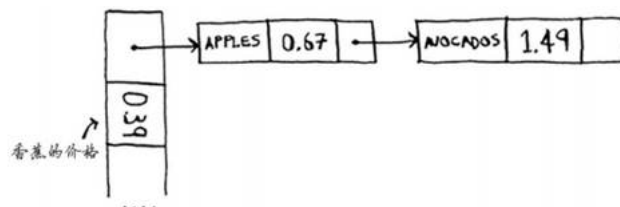
因此，完成每层所需的时间都为 $O(n)$ 。



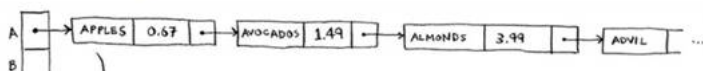
如果用专业术语来表达的话，我们会说，散列函数“将输入映射到数字”。你可能认为散列函数输出的数字没什么规律，但其实散列函数必须满足一些要求。

- ❑ 它必须是一致的。例如，假设你输入apple时得到的是4，那么每次输入apple时，得到的都必须为4。如果不是这样，散列表将毫无用处。
- ❑ 它应将不同的输入映射到不同的数字。例如，如果一个散列函数不管输入是什么都返回1，它就不是好的散列函数。最理想的情况是，将不同的输入映射到不同的数字。

不好，这个位置已经存储了苹果的价格！怎么办？这种情况被称为冲突（collision）：给两个键分配的位置相同。这是个问题。如果你将鳄梨的价格存储到这个位置，将覆盖苹果的价格，以后再查询苹果的价格时，得到的将是鳄梨的价格！冲突很糟糕，必须要避免。处理冲突的方式很多，最简单的办法如下：如果两个键映射到了同一个位置，就在这个位置存储一个链表。



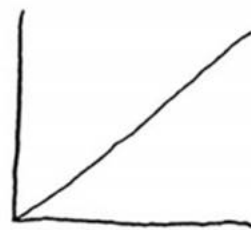
在这个例子中，apple和avocado映射到了同一个位置，因此在这个位置存储一个链表。在需要查询香蕉的价格时，速度依然很快。但在需要查询苹果的价格时，速度要慢些：你必须在相应的链表中找到apple。如果这个链表很短，也没什么大不了——只需搜索三四个元素。但是，假设你工作的杂货店只销售名称以字母A打头的商品。



- ❑ 散列函数很重要。前面的散列函数将所有的键都映射到一个位置，而最理想的情况是，散列函数将键均匀地映射到散列表的不同位置。

- ❑ 如果散列表存储的链表很长，散列表的速度将急剧下降。然而，如果使用的散列函数很好，这些链表就不会很长！

散列函数很重要，好的散列函数很少导致冲突。那么，如何选择好的散列函数呢？这将在下



$O(n)$

线性时间
(简单查找)

二分查找的速度更快，所需时间为对数时间。



$O(\log n)$

对数时间
(二分查找)

在散列表中查找所花费的时间为常量时间。



$O(1)$

常量时间
(散列表)

在最糟情况下，散列表所有操作的运行时间都为 $O(n)$ ——线性时间，这真的很慢。我们来将散列表同数组和链表比较一下。

	散列表 (平均 情况)	散列表 (最糟 情况)	数组	链表
查找	$O(1)$	$O(n)$	$O(1)$	$O(n)$
插入	$O(1)$	$O(n)$	$O(n)$	$O(1)$
删除	$O(1)$	$O(n)$	$O(n)$	$O(1)$

在平均情况下，散列表的查找（获取给定索引处的值）速度与数组一样快，而插入和删除速度与链表一样快，因此它兼具两者的优点！但在最糟情况下，散列表的各种操作的速度都很慢。因此，在使用散列表时，避开最糟情况至关重要。为此，需要避免冲突。而要避免冲突，需要有：

- ❑ 较低的填装因子；
- ❑ 良好的散列函数。

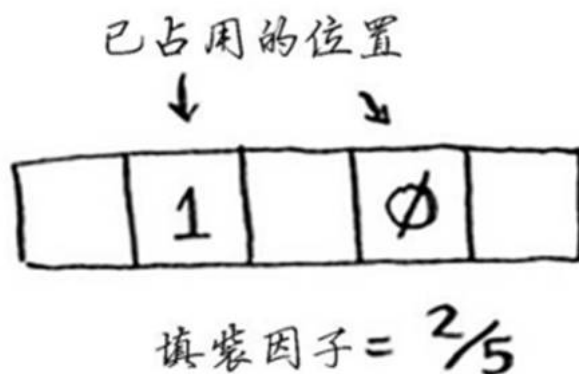
5.4.1 填装因子

散列表的填装因子很容易计算。

散列表包含的元素数

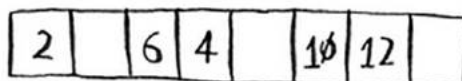
位置总数

散列表使用数组来存储数据，因此你需要计算数组中被占用的位置数。例如，下述散列表的填装因子为 $2/5$ ，即0.4。

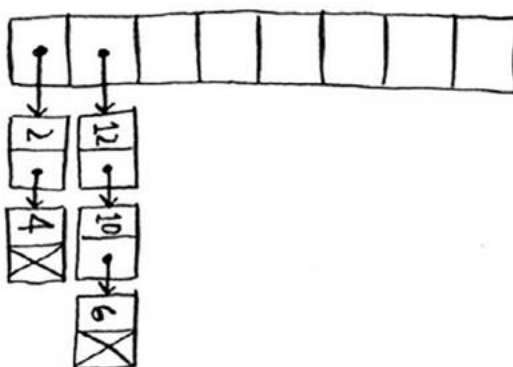


填装因子越低，发生冲突的可能性越小，散列表的性能越高。一个不错的经验规则是：一旦填装因子/负载因子大于0.7，就调整散列表的长度。

良好的散列函数让数组中的值呈均匀分布。



糟糕的散列函数让值扎堆，导致大量的冲突。



SHA函数是良好的散列函数

- ❑ 你可以结合散列函数和数组来创建散列表。
- ❑ 冲突很糟糕，你应使用可以最大限度减少冲突的散列函数。
- ❑ 散列表的查找、插入和删除速度都非常快。
- ❑ 散列表适合用于模拟映射关系。
- ❑ 一旦填充因子超过0.7，就该调整散列表的长度。
- ❑ 散列表可用于缓存数据（例如，在Web服务器上）。
- ❑ 散列表非常适合用于防止重复。

“散列表达到一定饱和度时，大量元素拥挤在相同下标位置，形成很长的链表，对后续插入操作和查询操作的性能都有很大影响。这时就需要扩容。HashMap的负载因子默认值为0.75。

扩容步骤：1、创建一个长度为原数组的2倍的新数组；2、遍历原数组，重新Hash到新数组中。

扩容后，原本拥挤的散列表重新变得稀疏。Java的ThreadLocal使用开放寻址法应对哈希冲突，JDK1.8之前，HashMap使用链表法，之后使用红黑树。”

—— 《漫画算法：小灰的算法之旅》

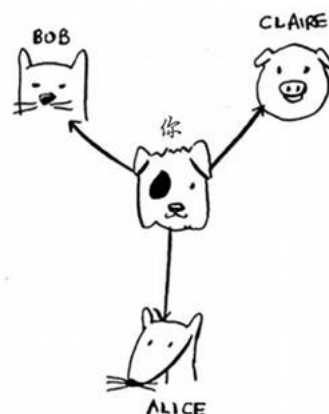
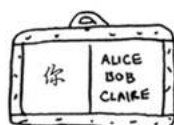
6 BFS

6.4 实现图

首先，需要使用代码来实现图。图由多个节点组成。

每个节点都与邻近节点相连，如果表示类似于“你→Bob”这样的关系呢？好在你知道的一种结构让你能够表示这种关系，它就是散列表！

记住，散列表让你能够将键映射到值。在这里，你要将节点映射到其所有邻居。



表示这种映射关系的Python代码如下。

```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
```

首先，创建一个队列。在Python中，可使用函数deque来创建一个双端队列。

```
from collections import deque
search_queue = deque()  # 创建一个队列
search_queue += graph["you"]  # 将你的邻居都加入到这个搜索队列中
```

别忘了，graph["you"]是一个数组，其中包含你的所有邻居，如["alice", "bob", "claire"]。这些邻居都将加入到搜索队列中。

```
def search(name):
    search_queue = deque()
    search_queue += graph[name]
    searched = []  # 这个数组用于记录检查过的人
    while search_queue:
        person = search_queue.popleft()
        if not person in searched:  # 仅当这个人没检查过时才检查
            if person_is_seller(person):
                print person + " is a mango seller!"
                return True
            else:
                search_queue += graph[person]
                searched.append(person)  # 将这个人标记为检查过
    return False

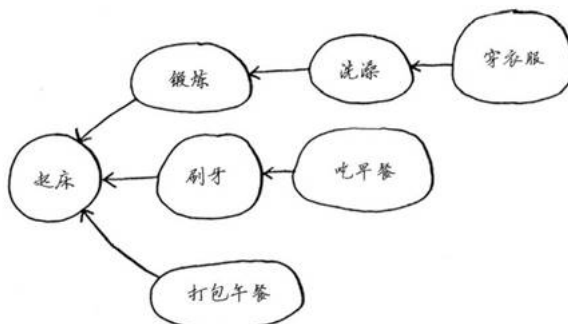
search("you")
```

searched即为入队标记

如果你在你的整个人际关系网中搜索芒果销售商，就意味着你将沿每条边前行（记住，边是从一个人到另一个人的箭头或连接），因此运行时间至少为 $O(\text{边数})$ 。

你还使用了一个队列，其中包含要检查的每个人。将一个人添加到队列需要的时间是固定的，即为 $O(1)$ ，因此对每个人都这样做需要的总时间为 $O(\text{人数})$ 。所以，广度优先搜索的运行时间为 $O(\text{人数} + \text{边数})$ ，这通常写作 $O(V + E)$ ，其中 V 为顶点（vertex）数， E 为边数。

6.4 下面是一个更大的图，请根据它创建一个可行的列表。



从某种程度上说，这种列表是有序的。如果任务A依赖于任务B，在列表中任务A就必须在任务B后面。这被称为拓扑排序，使用它可根据图创建一个有序列表。假设你正在规划一场婚礼，并有一个很大的图，其中充斥着需要做的事情，但却不知道要从哪里开始。这时就可使用拓扑排序来创建一个有序的任务列表。

这里重述一下，狄克斯特拉算法包含4个步骤。

- (1) 找出最便宜的节点，即可在最短时间内前往的节点。
- (2) 对于该节点的邻居，检查是否有前往它们的更短路径，如果有，就更新其开销。
- (3) 重复这个过程，直到对图中的每个节点都这样做了。
- (4) 计算最终路径。（下一节再介绍！）

狄克斯特拉算法只适用于有向无环图（directed acyclic graph, DAG）。

如果有负权边，就不能使用狄克斯特拉算法。

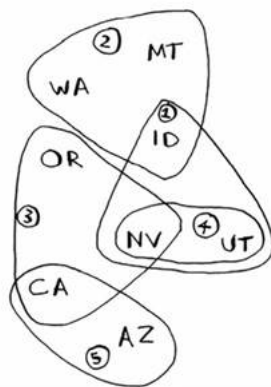
狄克斯特拉算法背后的关键理念：找出图中最便宜的节点，并确保没有到该节点的更便宜的路径！

7.6 小结

- 广度优先搜索用于在非加权图中查找最短路径。
- 狄克斯特拉算法用于在加权图中查找最短路径。
- 仅当权重为正时狄克斯特拉算法才管用。
- 如果图中包含负权边，请使用贝尔曼-福德算法。

8 贪心算法

每个广播台都覆盖特定的区域，不同广播台的覆盖区域可能重叠。



如何找出覆盖全美50个州的最小广播台集合呢？听起来很容易，但其实非常难。具体方法如下。

(1) 列出每个可能的广播台集合，这被称为幂集（power set）。可能的子集有 2^n 个。

(2) 在这些集合中，选出覆盖全美50个州的最小集合，时间复杂度为 $O(2^n)$ 。

近似算法

贪婪算法可化解危机！使用下面的贪婪算法可得到非常接近的解。

(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。

这是一种近似算法（approximation algorithm）。在获得精确解需要的时间太长时，可使用近似算法。判断近似算法优劣的标准如下：

- 速度有多快；
- 得到的近似解与最优解的接近程度。

贪婪算法是不错的选择，它们不仅简单，而且通常运行速度很快。在这个例子中，贪婪算法的运行时间为 $O(n^2)$ ，其中 n 为广播台数量。

具体实现：

1. 准备工作

出于简化考虑，这里假设要覆盖的州没有那么多，广播台也没有那么多。

首先，创建一个列表，其中包含要覆盖的州。

```
states_needed = set(["mt", "wa", "or", "id", "nv", "ut",  
"ca", "az"])  ◀..... 你传入一个数组，它被转换为集合
```

我使用集合来表示要覆盖的州。集合类似于列表，只是同样的元素只能出现一次，即集合不能包含重复的元素。例如，假设你有如下列表。

```
>>> arr = [1, 2, 2, 3, 3, 3]
```

并且你将其转换为集合。

```
>>> set(arr)  
set([1, 2, 3])
```

在这个集合中，1、2和3都只出现一次。

$[1, 2, 2, 3, 3, 3] \rightarrow \text{转换为集合} \rightarrow (1, 2, 3)$
集合

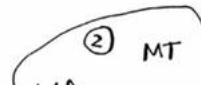
还需要有可供选择的广播台清单，我选择使用散列表来表示它。

```
stations = {}  
stations["kone"] = set(["id", "nv", "ut"])  
stations["ktwo"] = set(["wa", "id", "mt"])  
stations["kthree"] = set(["or", "nv", "ca"])  
stations["kfour"] = set(["nv", "ut"])  
stations["kfive"] = set(["ca", "az"])
```

其中的键为广播台的名称，值为广播台覆盖的州。在该示例中，广播台kone覆盖了爱达荷州、内达华州和犹他州。所有的值都是集合。你马上将看到，使用集合来表示一切可以简化工作。

最后，需要使用一个集合来存储最终选择的广播台。

```
final_stations = set()
```

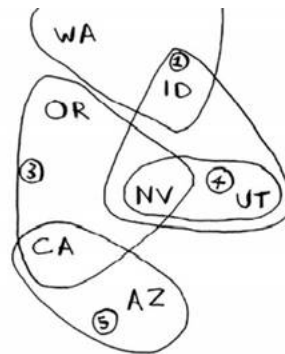


2. 计算答案

接下来需要计算要使用哪些广播台。根据右边的示意图，你能确定应使用哪些广播台吗？

正确的解可能有多个。你需要遍历所有的广播台，从中选择覆盖了最多的未覆盖州的广播台。我将这个广播台存储在 `best_station` 中。

```
best_station = None
states_covered = set()
for station, states_for_station in stations.items():
```



124 第8章 贪婪算法

`states_covered` 是一个集合，包含该广播台覆盖的所有未覆盖的州。for 循环迭代每个广播台，并确定它是否是最佳的广播台。下面来看看这个 for 循环的循环体。

```
covered = states_needed & states_for_station
if len(covered) > len(states_covered):  <----- 你没见过的语法！它计算交集
    best_station = station
    states_covered = covered
```

其中有一行代码看起来很有趣。

```
covered = states_needed & states_for_station
```

这行代码用于计算交集

4. 回到代码

回到前面的示例。

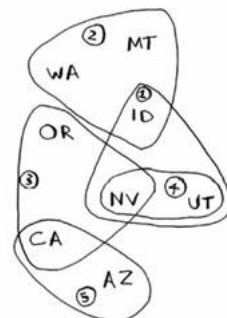
下面的代码计算交集。

```
covered = states_needed & states_for_station
```

`covered` 是一个集合，包含同时出现在 `states_needed` 和 `states_for_station` 中的州；换言之，它包含当前广播台覆盖的一系列还未覆盖的州！接下来，你检查该广播台覆盖的州是否比 `best_station` 多。

```
if len(covered) > len(states_covered):
    best_station = station
    states_covered = covered
```

如果是这样的，就将 `best_station` 设置为当前广播台。最后，你在 for 循环结束后将 `best_station` 添加到最终的广播台列表中。



```
final_stations.add(best_station)
```

你还需更新states_needed。由于该广播电台覆盖了一些州，因此不用再覆盖这些州。

```
states_needed -= states_covered
```

你不断地循环，直到states_needed为空。这个循环的完整代码如下。

```
while states_needed:
    best_station = None
    states_covered = set()
    for station, states in stations.items():
        covered = states_needed & states
        if len(covered) > len(states_covered):
            best_station = station
            states_covered = covered
```

```
states_needed -= states_covered
final_stations.add(best_station)
```

最后，你打印final_stations，结果类似于下面这样。

```
>>> print final_stations
set(['ktwo', 'kthree', 'kone', 'kfive'])
```

结果符合你的预期吗？选择的广播电台可能是2、3、4和5，而不是预期的1、2、3和5。下

```
>>> fruits = set(["avocado", "tomato", "banana"])
>>> vegetables = set(["beets", "carrots", "tomato"])
>>> fruits | vegetables  <----- 并集
set(["avocado", "beets", "carrots", "tomato", "banana"])
>>> fruits & vegetables  <----- 交集
set(["tomato"])
>>> fruits - vegetables  <----- 差集
set(["avocado", "banana"])
>>> vegetables - fruits  <----- 你觉得这行代码是做什么的呢？
```

城市数

1 → 1条路线

2 → 2个可能的出发城市 × 每个出发城市1条可能的路线 = 2条可能的路线

3 → 3个可能的出发城市 × 每个出发城市2条可能的路线 = 6条可能的路线

4 → 4个可能的出发城市 × 每个出发城市6条可能的路线 = 24条可能的路线

5 → 5个可能的出发城市 × 每个出发城市24条可能的路线 = 120条可能的路线

涉及6个城市时，可能的路线有多少条呢？如果你说720条，那就对了。7个城市为5040条，8个城市为40320条。

这被称为阶乘函数（factorial function），第3章介绍过。 $5! = 120$ 。假设有10个城市，可能的路线有多少条呢？ $10! = 3\,628\,800$ 。换句话说，涉及10个城市时，需要计算的可能路线超过300万条。正如你看到的，可能的路线数增加得非常快！因此，如果涉及的城市很多，根本就无法找出旅行商问题的正确解。

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。

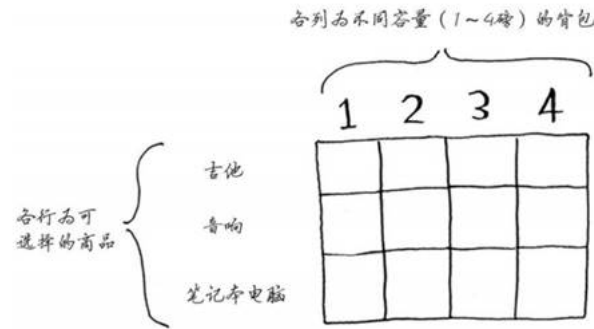
NP完全问题的简单定义是，以难解著称的问题，如旅行商问题和集合覆盖问题。很多非常 聪明的人都认为，根本不可能编写出可快速解决这些问题的算法。

但如果要找出经由指定几个点的的最短路径，就是旅行商问题——NP完全问题。简言之，没办法判断问题是不是NP完全问题，但还是有一些蛛丝马迹可循的。

- ❑ 元素较少时算法的运行速度非常快，但随着元素数量的增加，速度会变得非常慢。
- ❑ 涉及“所有组合”的问题通常是NP完全问题。
- ❑ 不能将问题分成小问题，必须考虑各种可能的情况。这可能是NP完全问题。
- ❑ 如果问题涉及序列（如旅行商问题中的城市序列）且难以解决，它可能就是NP完全问题。
- ❑ 如果问题涉及集合（如广播台集合）且难以解决，它可能就是NP完全问题。
- ❑ 如果问题可转换为集合覆盖问题或旅行商问题，那它肯定是NP完全问题。

9 动态规划

9.1 背包问题



网格的各行为商品，各列为不同容量（1~4磅）的背包。所有这些列你都需要，因为它们将帮助你计算子背包的价值。

网格最初是空的。你将填充其中的每个单元格，网格填满后，就找到了问题的答案！你一定单元格中的值就是要优化的值（maxValue）

第一个单元格表示背包的容量为1磅。吉他的重量也是1磅，这意味着它能装入背包！因此这个单元格包含吉他，价值为1500美元。

下面来开始填充网格。

	1	2	3	4
吉他 (G)	\$1500 G			
音响				
笔记本电脑				

与这个单元格一样，每个单元格都将包含当前可装入背包的所有商品。

来看下一个单元格。这个单元格表示背包的容量为2磅，完全能够装下吉他！

	1	2	3	4
吉他 (G)	\$1500 G	\$1500 G		
音响				
笔记本电脑				

2. 音响行

我们来填充下一行——音响行。你现在处于第二行，可偷的商品有吉他和音响。在每一行，可偷的商品都为当前行的商品以及之前各行的商品。因此，当前你还不能偷笔记本电脑，而只能偷音响和吉他。我们先来看第一个单元格，它表示容量为1磅的背包。在此之前，可装入1磅背包的商品的最大价值为1500美元。

接下来的两个单元格的情况与此相同。在这些单元格中，背包的容量分别为2磅和3磅，而以前的最大价值为1500美元。

	1	2	3	4
吉他 (G)	\$1500 G	\$1500 G	\$1500 G	\$1500 G
音响	↓ \$1500 G	↓ \$1500 G	↓ \$1500 G	
笔记本电脑				

由于这些背包装不下音响，因此最大价值保持不变。

背包容量为4磅呢？终于能够装下音响了！原来的最大价值为1500美元，但如果在背包中装入音响而不是吉他，价值将为3000美元！因此还是偷音响吧。

你更新了最大价值！如果背包的容量为4磅，就能装入价值至少3000美元的商品。在这个网格中，你逐步地更新最大价值。

	1	2	3	4	
吉他 (G)	\$1500 G	\$1500 G	\$1500 G	\$1500 G	← 以前的最大价值
音响 (S)	\$1500 G	\$1500 G	\$1500 G	\$3000 S	← 最新的最大价值
笔记本电脑					← 最终解

3. 笔记本电脑行

下面以同样的方式处理笔记本电脑。笔记本电脑重3磅，没法将其装入容量为1磅或2磅的背包，因此前两个单元格的最大价值还是1500美元。

	1	2	3	4
吉他 (G)	\$1500 G	\$1500 G	\$1500 G	\$1500 G
音响 (S)	\$1500 G	\$1500 G	\$1500 G	\$3000 S
笔记本电脑	\$1500 G	\$1500 G		

对于容量为3磅的背包，原来的最大价值为1500美元，但现在你可选择盗窃价值2000美元的笔记本电脑而不是吉他，这样新的最大价值将为2000美元！

对于容量为4磅的背包，情况很有趣。这是非常重要的部分。当前的最大价值为3000美元，你可不偷音响，而偷笔记本电脑，但它只值2000美元。

$$\begin{array}{ccc} \$3000 & \text{vs} & \$2000 \\ \text{音响} & & \text{笔记本电脑} \end{array}$$

价值没有原来高。但等一等，笔记本电脑的重量只有3磅，背包还有1磅的容量没用！

$$\begin{array}{ccc} \$3000 & \text{vs} & \left(\$2000 + \frac{???}{\text{余下的1磅容量}} \right) \\ \text{音响} & & \text{笔记本电脑} \end{array}$$

在1磅的容量中，可装入的商品的最大价值是多少呢？你之前计算过。

1磅容量可装入商品的最大价值 →

	1	2	3	4
吉他 (G)	\$1500	\$1500	\$1500	\$1500
音响 (S)	\$1500	\$1500	\$1500	\$3000
笔记本电脑 (L)	\$1500	\$1500	\$2000	

根据之前计算的最大价值可知，在1磅的容量中可装入吉他，价值1500美元。因此，你需要做如下比较。

$$\begin{array}{ccc} \$3000 & \text{vs} & \left(\$2000 + \$1500 \right) \\ \text{音响} & & \text{笔记本电脑} \end{array}$$

你可能始终心存疑惑：为何计算小背包可装入的商品的最大价值呢？但愿你现在明白了其中的原因！余下了空间时，你可根据这些子问题的答案来确定余下的空间可装入哪些商品。笔记本电脑和吉他的总价值为3500美元，因此偷它们是更好的选择。

最终的网格类似于下面这样。

	1	2	3	4
吉他 (G)	\$1500	\$1500	\$1500	\$1500
音响 (S)	\$1500	\$1500	\$1500	\$3000
笔记本电脑 (L)	\$1500	\$1500	\$2000	\$3500

↑
最终答案

答案如下：将吉他和笔记本电脑装入背包时价值最高，为3500美元。

你可能认为，计算最后一个单元格的价值时，我使用了不同的公式。那是因为填充之前的单元格时，我故意避开了一些复杂的因素。其实，计算每个单元格的价值时，使用的公式都相同。这个公式如下。

$$\text{CELL}[i][j] = \begin{cases} 1. \text{上一个单元格的值 (即 CELL}[i-1][j]\text{的值)} \\ \text{VS} \\ 2. \text{当前商品的价值 + 剩余空间的价值} \end{cases}$$

\uparrow
 $\text{CELL}[i-1][j - \text{当前商品的重量}]$

你可以使用这个公式来计算每个单元格的价值，最终的网格将与前一个网格相同。现在你明白了为何要求解子问题吧？你可以合并两个子问题的解来得到更大问题的解。

9.2.1 再增加一件商品将如何呢

这意味着背包容量为4磅时，你最多可偷价值3500美元的商品。但这是以前的情况，下面再添表示iPhone的行。

	1	2	3	4
吉他 (G)	\$1500 G	\$1500 G	\$1500 G	\$1500 G
音响 (S)	\$1500 G	\$1500 G	\$1500 G	\$3000 S
笔记本电脑 (L)	\$1500 G	\$1500 G	\$2000 L	\$3500 LG
iPhone				

↑
新的答案

最大价值可能发生变化！请尝试填充这个新增的行，再接着往下读。

我们从第一个单元格开始。iPhone可装入容量为1磅的背包。之前的最大价值为1500美元，但iPhone价值2000美元，因此该偷iPhone而不是吉他。

在下一个单元格中，你可装入iPhone和吉他。

\$1500 G	\$1500 G	\$1500 G	\$1500 G
\$1500 G	\$1500 G	\$1500 G	\$3000 S
\$1500 G	\$1500 G	\$2000 L	\$3500 LG
\$2000 I	\$3500 IG		

对于第三个单元格，也没有比装入iPhone和吉他更好的选择了。

对于最后一个单元格，情况比较有趣。当前的最大价值为3500美元，但你可偷iPhone，这将余下3磅的容量。

$$\begin{array}{c} \$3500 \\ \text{笔记本电脑+吉他} \end{array} \quad \text{vs} \quad \left(\begin{array}{c} \$2000 \\ \text{IPHONE} \end{array} + \begin{array}{c} ??? \\ \text{3磅容量的最大价值} \end{array} \right)$$

3磅容量的最大价值为2000美元！再加上iPhone价值2000美元，总价值为4000美元。新的最大价值诞生了！

最终的网格如下：

\$1500 G	\$1500 G	\$1500 G	\$1500 G
\$1500 G	\$1500 G	\$1500 G	\$3000 S
\$1500 G	\$1500 G	\$2000 L	\$3500 LG
\$3500 I	\$3500 IG	\$3500 IG	\$4000 IL

↑
新答案

问题：沿着一列往下走时，最大价值有可能降低吗？

	1	2	3	4
	\$1500	\$1500	\$1500	\$1500
往下走时最大价值有可能下降吗？ ↓	∅	∅	∅	\$3000

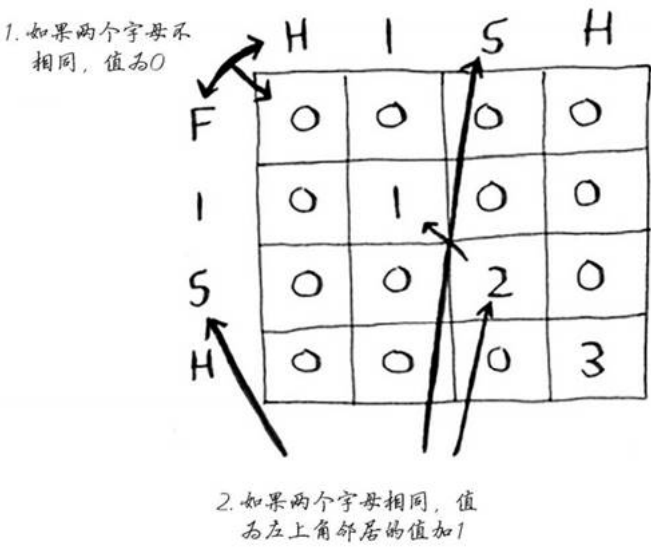
请找出这个问题的答案，再接着往下读。

答案：不可能。每次迭代时，你都存储当前的最大价值。最大价值不可能比以前低！

动态规划功能强大，它能够解决子问题并使用这些答案来解决大问题。但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。

9.3最长公共子串

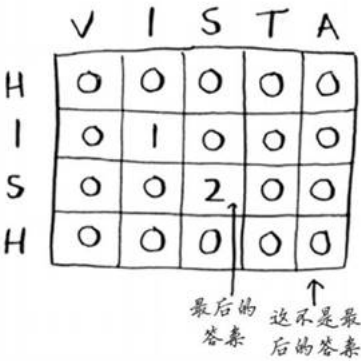
我使用下面的公式来计算每个单元格的值。



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
```

查找单词hish和vista的最长公共子串时，网格如下。



需要注意的一点是，这个问题的最终答案并不在最后一个单元格中！对于前面的背包问题，最终答案总是在最后的单元格中。但对于最长公共子串问题，答案为网格中最大的数字——它可能并不位于最后的单元格中。

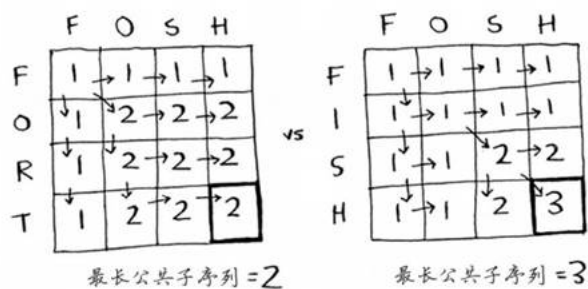
我们回到最初的问题：哪个单词与hish更像？hish和fish的最长公共子串包含三个字母，而hish和vista的最长公共子串包含两个字母。

因此Alex很可能原本要输入的是fish。

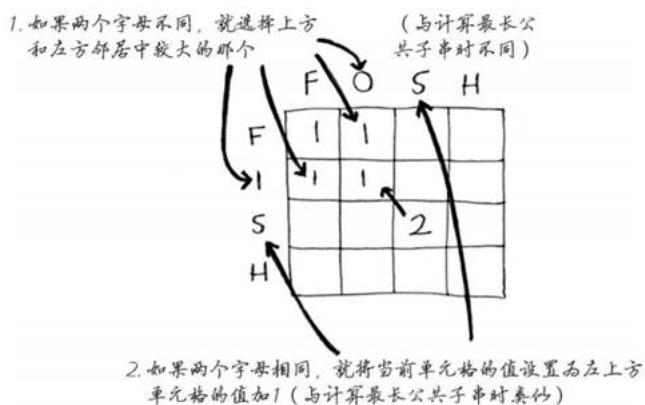
9.3.4 最长公共子序列

9.3.5 最长公共子序列之解决方案

最终的网格如下。



下面是填写各个单元格时使用的公式。



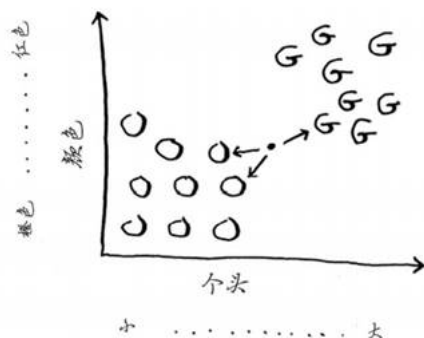
伪代码如下。

```
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])
```

本章到这里就结束了！它绝对是本书最难理解的一章。动态规划都有哪些实际应用呢？

- ❑ 生物学家根据最长公共序列来确定DNA链的相似性，进而判断度两种动物或疾病有多相似。最长公共序列还被用来寻找多发性硬化症治疗方案。
- ❑ 你使用过诸如git diff等命令吗？它们指出两个文件的差异，也是使用动态规划实现的。
- ❑ 前面讨论了字符串的相似程度。编辑距离（levenshtein distance）指出了两个字符串的相似程度，也是使用动态规划计算得到的。编辑距离算法的用途很多，从拼写检查到判断用户上传的资料是否是盗版，都在其中。
- ❑ 你使用过诸如Microsoft Word等具有断字功能的应用程序吗？它们如何确定在什么地方断字以确保行长一致呢？使用动态规划！

如果判断这个水果是橙子还是柚子呢？一种办法是看它的邻居。来看看离它最近的三个邻居。



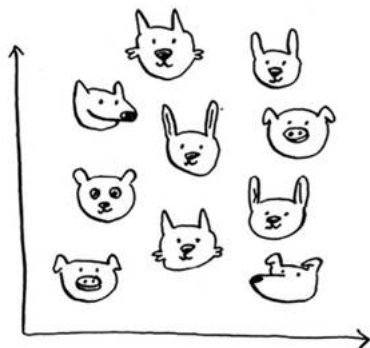
在这三个邻居中，橙子比柚子多，因此这个水果很可能是橙子。祝贺你，你刚才就是使用K最近邻 (k-nearest neighbours, KNN) 算法进行了分类！这个算法非常简单。



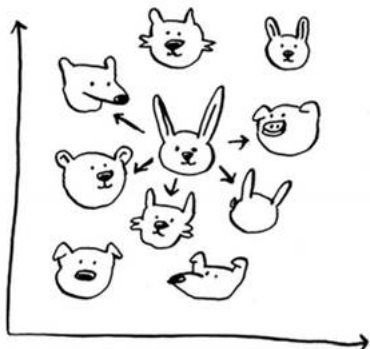
10.2 创建推荐系统

假设你是Netflix，要为用户创建一个电影推荐系统。从本质上说，这类似于前面的水果问题！

你可以将所有用户都放入一个图表中。

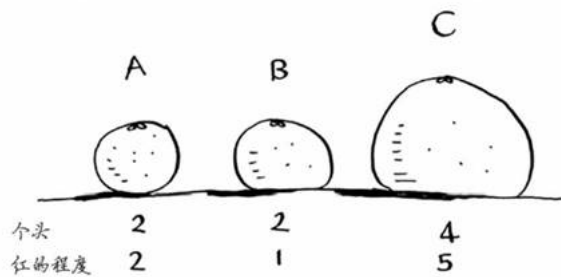


这些用户在图表中的位置取决于其喜好，因此喜好相似的用户距离较近。假设你要向Priyanka推荐电影，可以找出五位与他最接近的用户。

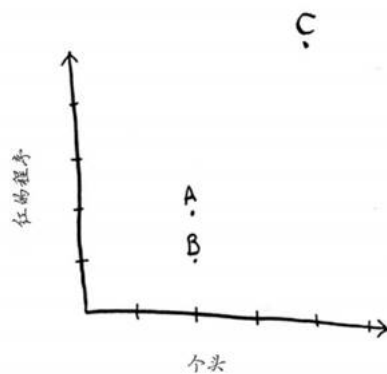


10.2.1 特征抽取

在前面的水果示例中，你根据个头和颜色来比较水果，换言之，你比较的特征是个头和颜色。现在假设有三个水果，你可抽取它们的特征。



再根据这些特征绘图。






从上图可知，水果A和B比较像。下面来度量它们有多像。要计算两点的距离，可使用毕达哥拉斯公式。

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$


例如，A和B的距离如下。

$$\begin{aligned} & \sqrt{(2-2)^2 + (2-1)^2} \\ &= \sqrt{0+1} \\ &= \sqrt{1} \\ &= 1 \end{aligned}$$

			
	PRIYANKA	JUSTIN	MORPHEUS
喜剧片	3	4	2
动作片	4	3	5
生活片	4	5	1
恐怖片	1	1	3
爱情片	4	5	1

Priyanka和Justin都喜欢爱情片且都讨厌恐怖片。Morpheus喜欢动作片，但讨厌爱情片（他讨厌好好的动作电影毁于浪漫的桥段）。前面判断水果是橙子还是柚子时，每种水果都用2个数字表示，你还记得吗？在这里，每位用户都用5个数字表示。

 $\rightarrow (2, 2)$

 $\rightarrow (3, 4, 4, 1, 4)$

在数学家看来，这里计算的是五维（而不是二维）空间中的距离，但计算公式不变。

$$\sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2 + (c_1 - c_2)^2 + (d_1 - d_2)^2 + (e_1 - e_2)^2}$$

假设你要预测Priyanka会给电影*Pitch Perfect*打多少分。Justin、JC、Joey、Lance和Chris都给它打了多少分呢？

JUSTIN : 5
JC : 4
JOEY : 4
LANCE : 5
CHRIS : 3

你求这些人打的分的平均值，结果为4.2。这就是回归（regression）。你将使用KNN来做两项基本工作——分类和回归：

- 分类就是编组；
- 回归就是预测结果（如一个数字）。

回归很有用。假设你在伯克利开个小小的面包店，每天都做新鲜面包，需要根据如下一组特征预测当天该烤多少条面包：

- 天气指数1~5（1表示天气很糟，5表示天气非常好）；
- 是不是周末或节假日（周末或节假日为1，否则为0）；
- 有没有活动（1表示有，0表示没有）。

你还有一些历史数据，记录了在各种不同的日子里售出的面包数量。



$$\boxed{\text{A.}} (5, 1, \emptyset) = \underset{\text{条}}{300} \quad \boxed{\text{B.}} (3, 1, 1) = \underset{\text{条}}{225}$$

$$\boxed{\text{C.}} (1, 1, \emptyset) = \underset{\text{条}}{75} \quad \boxed{\text{D.}} (4, \emptyset, 1) = \underset{\text{条}}{200}$$

$$\boxed{\text{E.}} (4, \emptyset, \emptyset) = \underset{\text{条}}{150} \quad \boxed{\text{F.}} (2, \emptyset, \emptyset) = \underset{\text{条}}{50}$$

今天是周末，天气不错。根据这些数据，预测你今天能售出多少条面包呢？我们来使用KNN算法，其中的K为4。首先，找出与今天最接近的4个邻居。

$$(4, 1, \emptyset) = ?$$

距离如下，因此最近的邻居为A、B、D和E。

A. 1 ←

B. 2 ←

C. 9

D. 2 ←

E. 1 ←

F. 5

将这些天售出的面包数平均，结果为218.75。这就是你今天要烤的面包数！

余弦相似度

前面计算两位用户的距离时，使用的都是距离公式。还有更合适的公式吗？在实际工作中，经常使用**余弦相似度**（cosine similarity）。假设有两位品味类似的用户，但其中一位打分时更保守。他们都很喜欢Manmohan Desai的电影*Amar Akbar Anthony*，但Paul给了5星，而Rowan只给4星。如果你使用距离公式，这两位用户可能不是邻居，虽然他们的品味非常接近。

余弦相似度不计算两个矢量的距离，而比较它们的角度，因此更适合处理前面所说的情况。本书不讨论余弦相似度，但如果你要使用KNN，就一定要研究研究它！

10.2.3 挑选合适的特征

为推荐电影，你让用户指出他对各类电影的喜好程度。如果你是让用户给一系列小猫图片打分呢？在这种情况下，你找出的是对小猫图片的欣赏品味类似的用户。对电影推荐系统来说，这很可能是一个糟糕的推荐引擎，因为你选择的特征与电影欣赏品味没多大关系。



又假设你只让用户给《玩具总动员》《玩具总动员2》和《玩具总动员3》打分。这将难以让用户的电影欣赏品味显现出来！使用KNN时，挑选合适的特征进行比较至关重要。所谓合适的特征，就是：

- 与要推荐的电影紧密相关的特征；
- 不偏不倚的特征（例如，如果只让用户给喜剧片打分，就无法判断他们是否喜欢动作片）。

你认为评分是不错的电影推荐指标吗？我给*The Wire*的评分可能比*House Hunters*高，但实际上我观看*House Hunters*的时间更长。该如何改进Netflix的推荐系统呢？

在挑选合适的特征方面，没有放之四海皆准的法则，你必须考虑到各种需要考虑的因素。

10.3 机器学习简介

KNN算法真的是很有用，堪称你进入神奇的机器学习领域的领路人！机器学习旨在让计算机更聪明。你见过一个机器学习的例子：创建推荐系统。下面再来看看其他一些例子。



10.3.1 OCR

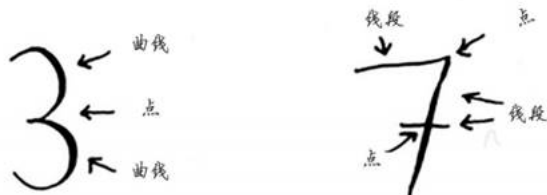
OCR指的是光学字符识别 (optical character recognition)，这意味着你可拍摄印刷页面的照片，计算机将自动识别出其中的文字。Google使用OCR来实现图书数字化。OCR是如何工作的呢？我们来看一个例子。请看下面的数字。

7

如何自动识别出这个数字是什么呢？可使用KNN。

- (1) 浏览大量的数字图像，将这些数字的特征提取出来。
- (2) 遇到新图像时，你提取该图像的特征，再找出它最近的邻居都是谁！

这与前面判断水果是橙子还是柚子时一样。一般而言，OCR算法提取线段、点和曲线等特征。



遇到新字符时，可从中提取同样的特征。

10.3.2 创建垃圾邮件过滤器

垃圾邮件过滤器使用一种简单算法——朴素贝叶斯分类器 (Naive Bayes classifier)，你首先需要使用一些数据对这个分类器进行训练。

主题	是不是垃圾邮件
"RESET YOUR PASSWORD"	不是
"YOU HAVE WON 1 MILLION DOLLARS"	是
"SEND ME YOUR PASSWORD"	是
"NIGERIAN PRINCE SENDS YOU 10 MILLION DOLLARS"	是
"HAPPY BIRTHDAY"	不是

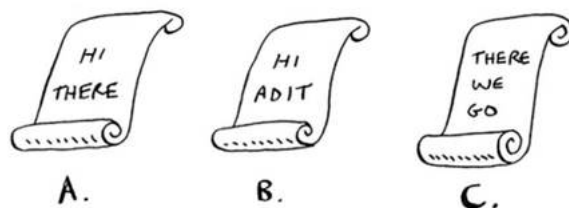
假设你收到一封主题为“collect your million dollars now!”的邮件，这是垃圾邮件吗？你可研究这个句子中的每个单词，看看它在垃圾邮件中出现的概率是多少。例如，使用这个非常简单的模型时，发现只有单词million在垃圾邮件中出现过。朴素贝叶斯分类器能计算出邮件为垃圾邮件的概率，其应用领域与KNN相似。

使用朴素贝叶斯分类器来对水果进行分类：假设有一个又大又红的水果，它是柚子的概率是多少呢？

	数组	二叉查找树
查找	$O(\log n)$	$O(\log n)$
插入	$O(n)$	$O(\log n)$
删除	$O(n)$	$O(\log n)$

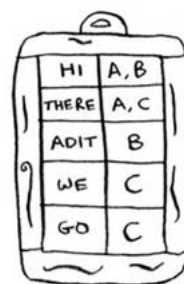
11.2 反向索引

这里非常简单地说说搜索引擎的工作原理。假设你有三个网页，内容如下。



我们根据这些内容创建一个散列表。

这个散列表的键为单词，值为包含指定单词的页面。现在假设有用户搜索hi，在这种情况下，搜索引擎需要检查哪些页面包含hi。



这是一种很有用的数据结构：一个散列表，将单词映射到包含它的页面。这种数据结构被称为反向索引（inverted index），常用于创建搜索引擎。如果你对搜索感兴趣，从反向索引着手研究是不错的选择。

11.3 傅里叶变换

绝妙、优雅且应用广泛的算法少之又少，傅里叶变换算是一个。Better Explained是一个杰出的网站，致力于以通俗易懂的语言阐释数学，它就傅里叶变换做了一个绝佳的比喻：给它一杯冰沙，它能告诉你其中包含哪些成分[摘自Kalid发表在Better Explained上的文章“An Interactive Guide to the Fourier Transform”，网址为<http://mng.bz/874X>]。换言之，给定一首歌曲，傅里叶变换能够将其中的各种频率分离出来。

这种理念虽然简单，应用却极其广泛。例如，如果能够将歌曲分解为不同的频率，就可强化你关心的部分，如强化低音并隐藏高音。傅里叶变换非常适合用于处理信号，可使用它来压缩音乐。为此，首先需要将音频文件分解为音符。傅里叶变换能够准确地指出各个音符对整个歌曲的贡献，让你能够将不重要的音符删除。这就是MP3格式的工作原理！

数字信号并非只有音乐一种类型。JPG也是一种压缩格式，也采用了刚才说的工作原理。傅里叶变换还被用来地震预测和DNA分析。

使用傅里叶变换可创建类似于Shazam这样的音乐识别软件。傅里叶变换的用途极其广泛，你遇到它的可能性极高！

11.4 并行算法

接下来的三个主题都与可扩展性和海量数据处理相关。我们身处一个处理器速度越来越快的时代，如果你要提高算法的速度，可等上几个月，届时计算机本身的速度就会更快。但这个时代已接近尾声，因此笔记本电脑和台式机转而采用多核处理器。为提高算法的速度，你需要让它们能够在多个内核中并行地执行！

来看一个简单的例子。在最佳情况下，排序算法的速度大致为 $O(n \log n)$ 。众所周知，对数组进行排序时，除非使用并行算法，否则运行时间不可能为 $O(n)$ ！对数组进行排序时，快速排序的并行版本所需的时间为 $O(n)$ 。

并行算法设计起来很难，要确保它们能够正确地工作并实现期望的速度提升也很难。有一点是确定的，那就是速度的提升并非线性的，因此即便你的笔记本电脑装备了两个而不是一个内核，算法的速度也不可能提高一倍，其中的原因有两个。

- ❑ **并行性管理开销。**假设你要对一个包含1000个元素的数组进行排序，如何在两个内核之间分配这项任务呢？如果让每个内核对其中500个元素进行排序，再将两个排好序的数组合并成一个有序数组，那么合并也是需要时间的。
- ❑ **负载均衡。**假设你需要完成10个任务，因此你给每个内核都分配5个任务。但分配给内核A的任务都很容易，10秒钟就完成了，而分配给内核B的任务都很难，1分钟才完成。这意味着有那么50秒，内核B在忙死忙活，而内核A却闲得很！你如何均匀地分配工作，让两个内核都一样忙呢？

要改善性能和可扩展性，并行算法可能是不错的选择！

11.5 MapReduce

有一种特殊的并行算法正越来越流行，它就是分布式算法。在并行算法只需两到四个内核时，完全可以在笔记本电脑上运行它，但如果需要数百个内核呢？在这种情况下，可让算法在多台计算机上运行。MapReduce是一种流行的分布式算法，你可通过流行的开源工具Apache Hadoop来使用它。

11.5.1 分布式算法为何很有用

假设你有一个数据库表，包含数十亿乃至数万亿行，需要对其执行复杂的SQL查询。在这种情况下，你不能使用MySQL，因为数据表的行数超过数十亿后，它处理起来将很吃力。相反，你需要通过Hadoop来使用MapReduce！

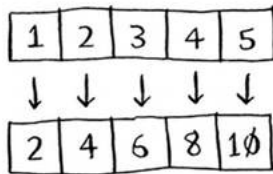
又假设你需要处理一个很长的清单，其中包含100万个职位，而每个职位处理起来需要10秒。如果使用一台计算机来处理，将耗时数月！如果使用100台计算机来处理，可能几天就能完工。

分布式算法非常适合用于在短时间内完成海量工作，其中的MapReduce基于两个简单的理念：映射（map）函数和归并（reduce）函数。

11.5.2 映射函数

映射函数很简单，它接受一个数组，并对其中的每个元素执行同样的处理。例如，下面的映射函数将数组的每个元素翻倍。

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = map(lambda x: 2 * x, arr1)
[2, 4, 6, 8, 10]
```



arr2包含[2, 4, 6, 8, 10]：将数组arr1的每个元素都翻倍！将元素翻倍的速度非常快，但如果要执行的操作需要更长的时间呢？请看下面的伪代码。

```
>>> arr1 = # A list of URLs
>>> arr2 = map(download_page, arr1)
```

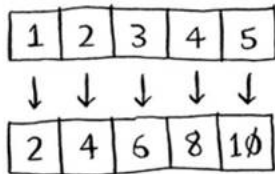
在这个示例中，你有一个URL清单，需要下载每个URL指向的页面并将这些内容存储在数组arr2中。对于每个URL，处理起来都可能需要几秒钟。如果总共有1000个URL，可能耗时几小时！

map对“数组”（总任务）的所有“元素”（子任务）分别做映射

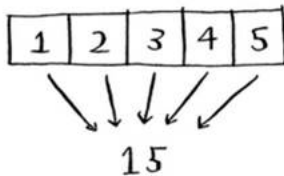
如果有100台计算机，而map能够自动将工作分配给这些计算机去完成就好了。这样就可同时下载100个页面，下载速度将快得多！这就是MapReduce中“映射”部分基于的理念。

11.5.3 归并函数

归并函数可能令人迷惑，其理念是将很多项归并为一项。映射是将一个数组转换为另一个数组。



而归并是将一个数组转换为一个元素。



下面是一个示例。

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> reduce(lambda x,y: x+y, arr1)
15
```

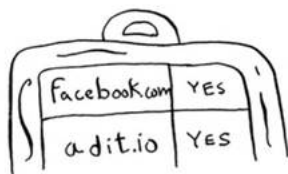
在这个示例中，你将数组中的所有元素相加：1 + 2 + 3 + 4 + 5 = 15！这里不深入介绍归并，网上有很多这方面的教程。

MapReduce使用这两个简单概念在多台计算机上执行数据查询。数据集很大，包含数十亿行时，使用MapReduce只需几分钟就可获得查询结果，而传统数据库可能要耗费数小时。

reduce将所有子任务映射后的结果汇总

11.6 布隆过滤器和 HyperLogLog

例如，Google可能有一个庞大的散列表，其中的键是已搜集的网页。



要判断是否已搜集adit.io，可在这个散列表中查找它。

adit.io → YES

adit.io是这个散列表中的一个键，这说明已搜集它。散列表的平均查找时间为 $O(1)$ ，即查找时间是固定的，非常好！

只是Google需要建立数万亿个网页的索引，因此这个散列表非常大，需要占用大量的存储空间。Reddit和bit.ly也面临着这样的问题。面临海量数据，你需要创造性的解决方案！

11.6.1 布隆过滤器

11.6.1 布隆过滤器

布隆过滤器提供了解决之道。布隆过滤器是一种概率型数据结构，它提供的答案有可能不对，但很可能是正确的。为判断网页以前是否已搜集，可不使用散列表，而使用布隆过滤器。使用散列表时，答案绝对可靠，而使用布隆过滤器时，答案却是很可能是正确的。

□ 可能出现错报的情况，即Google可能指出“这个网站已搜集”，但实际上并没有搜集。

□ 不可能出现漏报的情况，即如果布隆过滤器说“这个网站未搜集”，就肯定未搜集。

布隆过滤器的优点在于占用的存储空间很少。使用散列表时，必须存储Google搜集过的所有URL，但使用布隆过滤器时不用这样做。布隆过滤器非常适合用于不要求答案绝对准确的情况，前面所有的示例都是这样的。对bit.ly而言，这样说完全可行：“我们认为这个网站可能是恶意的，请倍加小心。”

11.6.2 HyperLogLog

HyperLogLog是一种类似于布隆过滤器的算法。如果Google要计算用户执行的不同搜索的数量，或者Amazon要计算当天用户浏览的不同商品的数量，要回答这些问题，需要耗用大量的空间！对Google来说，必须有一个日志，其中包含用户执行的不同搜索。有用户执行搜索时，Google必须判断该搜索是否包含在日志中：如果答案是否定的，就必须将其加入到日志中。即便只记录一天的搜索，这种日志也大得不得了！

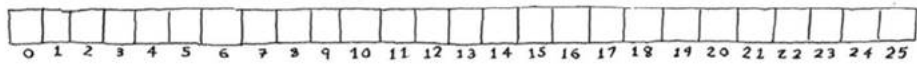
HyperLogLog近似地计算集合中不同的元素数，与布隆过滤器一样，它不能给出准确的答案，但也八九不离十，而占用的内存空间却少得多。

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！

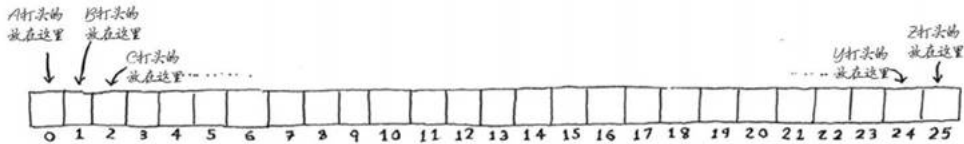
11.7 SHA算法

11.7 SHA 算法

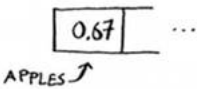
还记得第5章介绍的散列算法吗？我们回顾一下，假设你有一个键，需要将其相关联的值放到数组中。



你使用散列函数来确定应将这个值放在数组的什么地方。



你将值放在这个地方。



11.7.1 比较文件

另一种散列函数是安全散列算法（secure hash algorithm, SHA）函数。给定一个字符串，SHA 返回其散列值。

“hello” ⇒ 2cf24db...

这里的术语有点令人迷惑。SHA是一个散列函数，它生成一个散列值——一个较短的字符串。用于创建散列表的散列函数根据字符串生成数组索引，而SHA根据字符串生成另一个字符串。

对于每个不同的字符串，SHA生成的散列值都不同。

“hello” ⇒ 2cf24db...
“algorithm” ⇒ b1eb2ec..
“password” ⇒ 5e88489...

说 明

SHA 生成的散列值很长，这里截短了。

11.7.2 检查密码

SHA还让你能在不知道原始字符串的情况下对其进行比较。例如，假设Gmail遭到攻击，攻击者窃取了所有的密码！你的密码暴露了吗？没有，因为Google存储的并非密码，而是密码的SHA散列值！你输入密码时，Google计算其散列值，并将结果同其数据库中的散列值进行比较。



Google只是比较散列值，因此不必存储你的密码！SHA被广泛用于计算密码的散列值。这种散列算法是单向的。你可根据字符串计算出散列值。

`abc123 → 6ca13d`

但你无法根据散列值推断出原始字符串。

`? ← 6ca13d`

这意味着计算攻击者窃取了Gmail的SHA散列值，也无法据此推断出原始密码！你可将密码

SHA实际上是一系列算法：SHA-0、SHA-1、SHA-2和SHA-3。本书编写期间，SHA-0和SHA-1已被发现存在一些缺陷。如果你要使用SHA算法来计算密码的散列值，请使用SHA-2或SHA-3。当前，最安全的密码散列函数是bcrypt，但没有任何东西是万无一失的。

11.8 局部敏感的散列算法

SHA还有一个重要特征，那就是局部不敏感的。假设你有一个字符串，并计算了其散列值。

`dog → cd6357`

如果你修改其中的一个字符，再计算其散列值，结果将截然不同！

`dot → e392da`

这很好，让攻击者无法通过比较散列值是否类似来破解密码。

有时候，你希望结果相反，即希望散列函数是局部敏感的。在这种情况下，可使用Simhash。如果你对字符串做细微的修改，Simhash生成的散列值也只存在细微的差别。这让你能够通过比较散列值来判断两个字符串的相似程度，这很有用！

- ❑ Google使用Simhash来判断网页是否已搜集。
- ❑ 老师可以使用Simhash来判断学生的论文是否是从网上抄的。
- ❑ Scribd允许用户上传文档或图书，以便与人分享，但不希望用户上传有版权的内容！这个网站可使用Simhash来检查上传的内容是否与小说《哈利·波特》类似，如果类似，就自动拒绝。

需要检查两项内容的相似程度时，Simhash很有用。

11.9 Diffie-Hellman 密钥交换

这里有必要提一提Diffie-Hellman算法，它以优雅的方式解决了一个古老的问题：如何对消息进行加密，以便只有收件人才能看懂呢？

最简单的方式是设计一种加密算法，如将a转换为1，b转换为2，以此类推。这样，如果我给你发送消息“4,15,7”，你就可将其转换为“d,o,g”。但我们必须就加密算法达成一致，这种方式才可行。我们不能通过电子邮件来协商，因为可能有人拦截电子邮件，获悉加密算法，进而破译消息。即便通过会面来协商，这种加密算法也可能被猜出来——它并不复杂。因此，我们每天都得修改加密算法，但这样我们每天都得会面！

即便我们能够每天修改，像这样简单的加密算法也很容易使用蛮力攻击破解。假设我看到消

Diffie-Hellman算法解决了如下两个问题。

- 双方无需知道加密算法。他们不必会面协商要使用的加密算法。
- 要破解加密的消息比登天还难。

Diffie-Hellman使用两个密钥：公钥和私钥。顾名思义，公钥就是公开的，可将其发布到网站上，通过电子邮件发送给朋友，或使用其他任何方式来发布。你不必将它藏着掖着。有人要向你发送消息时，他使用公钥对其进行加密。加密后的消息只有使用私钥才能解密。只要只有你知道私钥，就只有你才能解密消息！

Diffie-Hellman算法及其替代者RSA依然被广泛使用。如果你对加密感兴趣，先着手研究Diffie-Hellman算法是不错的选择：它既优雅又不难理解。

11.10 线性规划

最好的东西留到最后介绍。线性规划是我知道的最酷的算法之一。

线性规划用于在给定约束条件下最大限度地改善指定的指标。例如，假设你所在的公司生产两种产品：衬衫和手提袋。衬衫每件利润2美元，需要消耗1米布料和5粒扣子；手提袋每个利润3美元，需要消耗2米布料和2粒扣子。你有11米布料和20粒扣子，为最大限度地提高利润，该生产多少件衬衫、多少个手提袋呢？

在这个例子中，目标是利润最大化，而约束条件是拥有的原材料数量。

再举一个例子。你是个政客，要尽可能多地获得支持票。你经过研究发现，平均而言，对于每张支持票，在旧金山需要付出1小时的劳动（宣传、研究等）和2美元的开销，而在芝加哥需要付出1.5小时的劳动和1美元的开销。在旧金山和芝加哥，你至少需要分别获得500和300张支持票。你有50天的时间，总预算为1500美元。请问你最多可从这两个地方获得多少支持票？

这里的目标是支持票数最大化，而约束条件是时间和预算。

你可能在想，本书花了很大的篇幅讨论最优化，这与线性规划有何关系？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。但愿你听到这一点后心潮澎湃！

线性规划使用Simplex算法，这个算法很复杂，因此本书没有介绍。如果你对最优化感兴趣，就研究研究线性规划吧！