[笔记][算法图解]

编程

---

1. 算法简介

1.1 引言

算法是一组完成任务的指令。

1.2 二分查找

仅当列表是有序的时候，二分查找才管用。

线性时间 `linear time`
对数时间 `log 时间`

```
# Python 3
def binary_search(array, item):
    """返回 item 在 array 中的下标，没找到返回 None。"""
    low = 0
    high = len(array) - 1

    while low <= high:
        mid = (low + high) // 2
        guess = array[mid]
        if guess == item:
            return mid
        elif guess > item:
            high = mid - 1
        else:
            low = mid + 1

    return None


if __name__ == '__main__':
    array = [1, 2, 3, 4, 5, 6, 7, 8]
    item = 7
    print(binary_search(array, item))

"""
6
"""
```

1.3 大 O 表示法

大 `O` 表示法 指出了算法有多快，它让你能够比较操作数，它指出了算法运行时间的增速。

> 【注】
> 个人理解：大 `O` 表示法指的是在最坏的情况下，算法涉及到的基本运算的数量级，忽略了系数和常数。

从快到慢的大 `O` 运行时间

- `O(log n)`：注意都是以 `2` 为底的对数，比如二分查找。
- `O(n)`：线性时间，比如简单查找。
- `O(n * log n)`：快速排序
- `O(n2)`：选择排序
- `O(n!)`：旅行商问题

旅行商问题：一个旅行商要去 `5` 个城市，需要保证总距离最短，最坏要算 `5!` 次。

---

2. 选择排序

2.1 内存的工作原理

计算机就像是很多抽屉的集合体，每个抽屉都有地址，可以存放东西。

2.2 数组和链表

数组中的元素占据连续的内存空间，并且同一个数组中的元素类型相同。
如果加入了一个元素之后放不下了，就要新开辟一块内存并转移。
可以一次性多开辟几块内存，但是这样会造成资源浪费，当放不下了还是需要转移。

链表中的元素可以存储在内存的任何地方。
链表的每个元素都存储了下一个元素的地址。
从而使一系列随机的内存地址串在一起。
只要有足够的内存空间，就能为链表分配内存。
链表的优势在于插入元素方面。

如果需要选择性的读取某一个元素，链表的效率很低，因为你不知道它在哪里，只能从头开始读取。
而数组则知道其中每一个元素的地址，在随机读取元素时，数组的效率很高。

数组中元素的位置称为**索引**。

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)`

> 【注】
> 这里链表的删除指的是删除第一个或最后一个元素，通常我们都知道这些元素的地址。所以是 `O(1)`。

有两种访问方式：

- **随机访问**
- **顺序访问**

很多情况都要求能够随机访问，所以数组比链表用的更多。

数组链表：由数组组成的链表。比如 `Facebook` 可以用数组链表存储用户名。
数组链表的查找速度比数组慢，插入速度比数组快。
数组链表的查找速度比链表快，插入速度与链表相当。

2.3 选择排序

```python
def find_smallest(arr):
    """返回数组中最小值的索引。"""
    smallest = arr[0]
    smallest_i = 0

    for i in range(1, len(arr)):
        if arr[i] < smallest:
            smallest = arr[i]
            smallest_i = i

    return smallest_i


def selection_sort(arr):
    """对数组进行排序。"""
    result = []

    for i in range(len(arr)):
        smallest_i = find_smallest(arr)
        result.append(arr.pop(smallest_i))

    return result


if __name__ == '__main__':
    arr = [1, 4, 2, 7, 5, 8, 6, 3]
    result = selection_sort(arr)
    print(result)
```

```
"""
[1, 2, 3, 4, 5, 6, 7, 8]
"""
```

---

3. 递归

伪代码是对手头问题的简要描述，看着像代码，但其实更接近自然语言。

3.1 递归

http://stackoverflow.com/a/72694/139117
如果使用循环，程序的性能可能更高；如果使用递归，程序可能更容易理解。
如何选择要看什么对你来说更重要。

3.2 基线条件和递归条件

每个递归函数都有两部分：

- 基线条件 `base case`：指的是**函数不再调用自己**，从而避免形成无限循环。
- 递归条件 `recursive case`：指的是**函数调用自己。**

3.3 栈

栈 `stack` 的两种操作：**压入**（插入），**弹出**（删除并读取）

调用函数时，会给该函数调用分配一块内存，然后计算机将函数调用所涉及的所有变量的值存储到内存中。
计算机使用栈来表示这些内存块。
调用另一个函数时，当前函数暂停并处于未完成状态。

调用栈 `call stack`：用于存储多个函数的变量。

递归函数也使用调用栈，在一个函数调用中不能访问另一个的同名变量。
调用栈包含了未完成的函数调用。

> 【注】
> 或者说调用栈包含了**未返回的**函数调用。

递归函数的代价
每个函数调用都要占用一定的内存，因为调用栈存储了每次函数调用的信息。
解决方案

- 使用尾递归
- 使用循环

3.4 小结

- 递归指的是 调用自己的函数。
- 每个递归函数都有两个条件：基线条件 和 递归条件。
- 栈有两种操作：压入 和 弹出。
- 所有函数调用都进入 调用栈。
- 调用栈很高的话，会占用大量内存。
- 如果出现无限递归，每个程序可以使用的调用栈空间是有限的，最终会导致栈溢出 `stack overflow`。

---

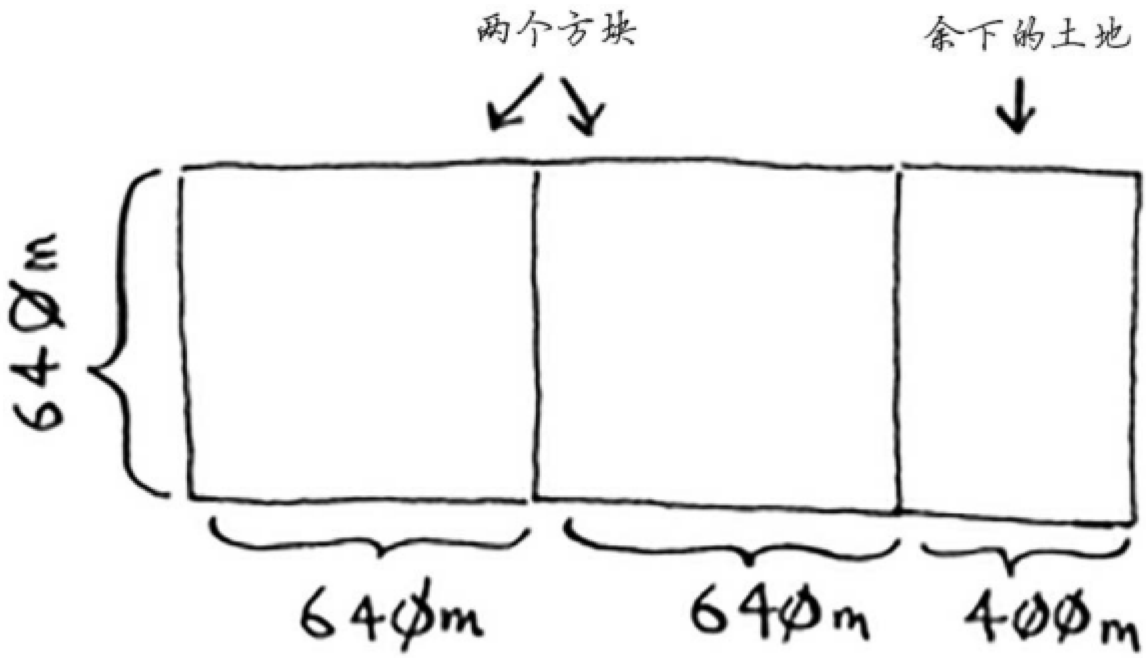
4. 快速排序

分而治之 `divide and conquer`，缩写是 `D & C`。

4.1 分而治之

使用 `D&C` 解决问题的两个步骤：

- 找出 基线条件，这种条件必须尽可能简单
- 不断地将问题 分解（或者说 缩小规模），直到符合基线条件



适用于这小块地的最大方块，也是适用于整块地的最大方块

> **欧几里得算法**
> https://www.khanacademy.org/computing/computer-science/ryptography/modarithmetic/a/the-euclidean-algorithm

`D&C` 不是算法，而是一种解决问题的思路。

提示：编写涉及数组的递归函数时，基线条件 通常是数组为空或只包含一个元素。

`Haskell` 等函数式编程语言没有循环，只能使用递归来编写这样的函数。

递归版本的 `sum`

```
def recursive_sum(arr):
    """使用递归对数组求和。"""
    if not arr:
        return 0
    else:
        return arr[0] + recursive_sum(arr[1:])


if __name__ == '__main__':
    arr = [1, 2, 3]
    print(recursive_sum(arr))

"""
```

```
6
"""
```

计算列表中包含的元素数。

```python
def count_list(li):
    """计算列表中包含的元素个数。"""
    if not li:
        return 0
    else:
        return 1 + count_list(li[1:])


if __name__ == '__main__':
    li = [1, 2, 3]
    print(count_list(li))

"""
3
"""
```

找到列表中最大的数。

```python
def find_max(arr):
    if len(arr) == 1:
        return arr[0]
    elif len(arr) == 2:
        return arr[0] if arr[0] > arr[1] else arr[1]

    first = arr[0]
    second = find_max(arr[1:])
    return first if first > second else second


if __name__ == '__main__':
    arr = [1, 7, 4, 8]
    print(find_max(arr))

"""
8
"""
```

4.2 快速排序

`C` 语言标准库中的 `qsort` 用的就是快速排序。

> 【注】`quick sort`

归纳证明 与 `D&C` 协同发挥作用。

快速排序

- 基线条件：
  - 空数组或者只包含一个元素的数组，直接返回即可，根本不用排序。
  - 包含两个元素的数组，直接排序。
- 递归条件：
  - 对于包含三个及以上的元素的数组，选择一个 基准值 `pivot`。
  - 分区 `partitioning` ：将数组与基准值进行比较，得到两个子数组。小于基准值的元素和大于基准值的元素。
  - 对这两个子数组进行快速排序。

快速排序代码

```
def qsort(arr):
    if len(arr) < 2:
        return arr

    pivot, arr = arr[0], arr[1:]
    smaller = []
    bigger = []

    for i in arr:
        if i <= pivot:
            smaller.append(i)
        else:
            bigger.append(i)

    return qsort(smaller) + [pivot] + qsort(bigger)


if __name__ == '__main__':
    arr = [1, 4, 2, 7, 5, 3, 8, 9, 6]
    print(qsort(arr))
"""
[1, 2, 3, 4, 5, 6, 7, 8, 9]
"""
```

> 【注】
> 书中的代码使用了列表推导式
> `less = [i for i in array[1:] if i <= pivot]`
> 比较清晰，但是这样需要遍历两次数组

4.3 再谈大 O 表示法

合并排序 `merge sort`，时间复杂度为 `O(n log n)`
快速排序 `quick sort`，最坏情况 `O(n2)`，平均情况 `O(n log n)`

大 `O` 表示法实际上忽略了常量 `c`，它是算法所需的固定时间量。
很多情况下，常量没有什么影响。
但是对于快速排序和合并排序，常量影响很大。
快速排序的常量比合并排序小，如果运行时间都为 `O(n log n)` 的话，快速排序速度更快，而快速排序遇到平均情况的可能性更大。

最佳情况/平均情况：
快速排序调用栈的高度为 `O(log n)`。
每层需要的时间为 `O(n)`。
该算法的最佳运行时间/平均运行时间为 `O(n log n)`。
如果能每次都 随机选择 基准值，那么平均运行时间就是 `O(n log n)`。

4.4 小结

- `D&C` 将问题逐步分解。使用 `D&C` 处理列表时，基线条件很可能是空数组或只包含一个元素的数组。
- 实现快速排序时，请随机地选择用作基准值的元素。快速排序的平均运行时间为 `O(n log n)`。
- 大 `O` 表示法中的常量有时候事关重大，这就是快速排序比合并排序快的原因所在。
- 比较简单查找和二分查找时，常量几乎无关紧要，因为列表很长时， `O(log n)` 的速度比 `O(n)` 快得多。

---

5. 散列表

5.1 散列函数

散列函数

- 将输入映射到数字
- 即无论你给它什么数据，它都还你一个数字

散列函数的要求

- 它必须是一致的。同样的输入，会得到同样的结果。
- 它应该将不同的输入映射到不同的数字。

5.2 应用案例

散列表的应用

1. 模拟映射关系
2. 防止重复
3. 缓存/记住数据，以免服务器再通过处理来生成它们

查找
`DNS` 解析 `DNS resolution`：将网址映射到 `IP` 地址。
散列表是提供 `DNS` 解析这种功能的方式之一。

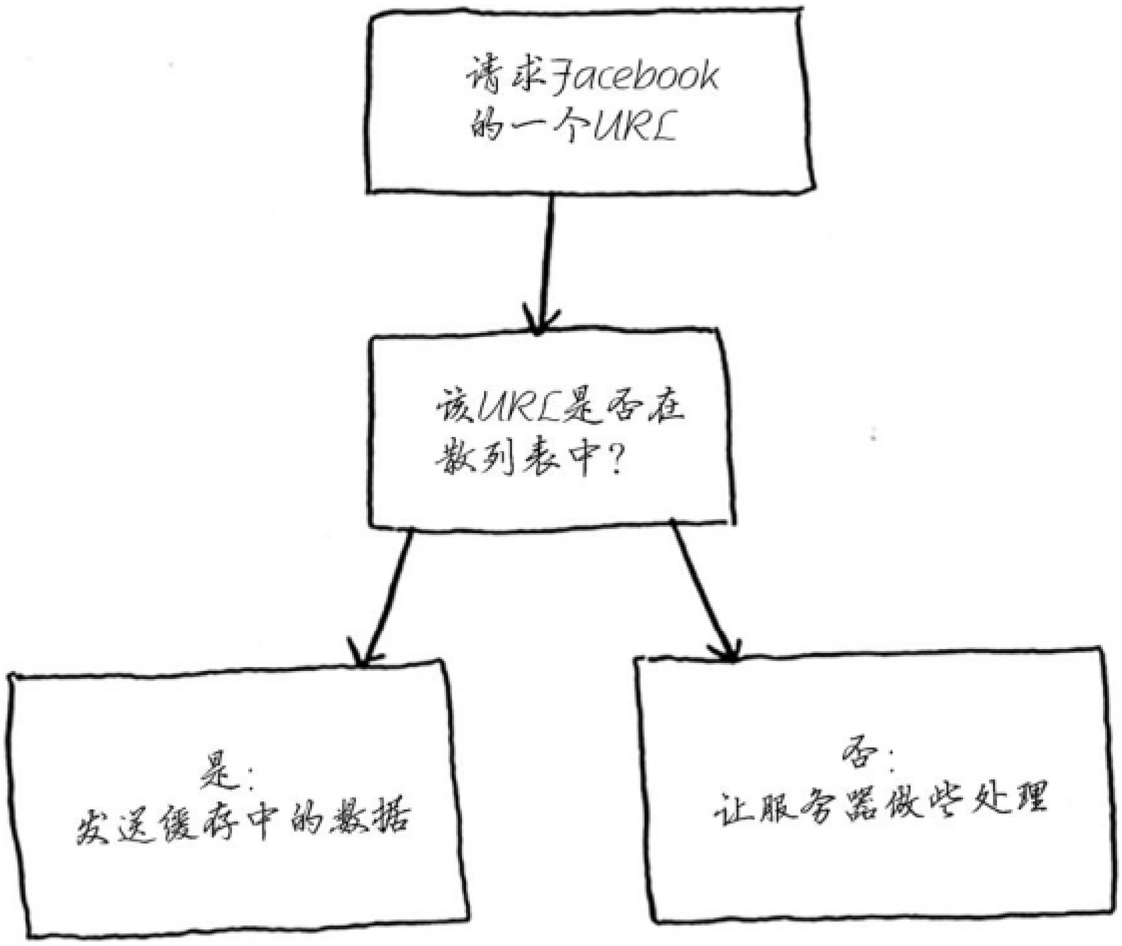
防止重复
用散列表来检查是否重复，速度非常快。

缓存
网站将数据记住，而不再重新计算。
比如 `Facebook` 将主页缓存起来，一旦用户注销，就把缓存的主页展示给用户，从而不用请求服务器。

缓存的优点

- 用户能够更快地看到网页
- 减少服务器压力

缓存的数据存储在散列表中。
服务器需要将页面 `URL` 映射到页面数据。



5.3 冲突

冲突 `collision`：在散列表中，给两个键分配的位置相同。

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。

散列函数很重要。最理想的情况：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。但是如果使用的散列函数很好，这些链表就不会很长。

5.4 性能

平均情况下，散列表执行各种操作的时间为 `O(1)`，称为 常量时间。

	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)

平均情况下的散列表兼具数组和链表的优点。
使用散列表时，避开最糟情况至关重要。
要避免冲突，就要有：

- 较低的填装因子
- 良好的散列函数

填装因子 等于**散列表包含的元素数 / 位置总数**

一旦填装因子开始增大，你就需要在散列表中添加位置，这被称为**调整长度** `resizing`

调整长度通常是将数组的长度增加一倍。
经验规则：一般填装因子大于 `0.7` ，就调整散列表的长度。
调整长度的开销很大，所以你不想频繁的这么做，但是考虑到调整长度需要的时间，散列表操作所需的事件也为 `O(1)` 。

良好的散列函数：`SHA` 函数。

5.5 小结

你可以使用散列函数和数组来创建散列表。

---

6. 广度优先搜索

广度优先搜索 `breadth-first search` ，缩写 `BFS`
能够让你找出 两样东西之间的最短距离

6.1 图简介

解决最短路径问题 `shortest-path problem` 的算法被称为**广度优先搜索**。

6.2 图是什么

图由 节点 `node` 和 边 `edge` 组成。
两个直接相连的节点被称为 邻居。

6.3 广度优先搜索

广度优先搜索是一种用于图的查找算法，用于解答两种问题：

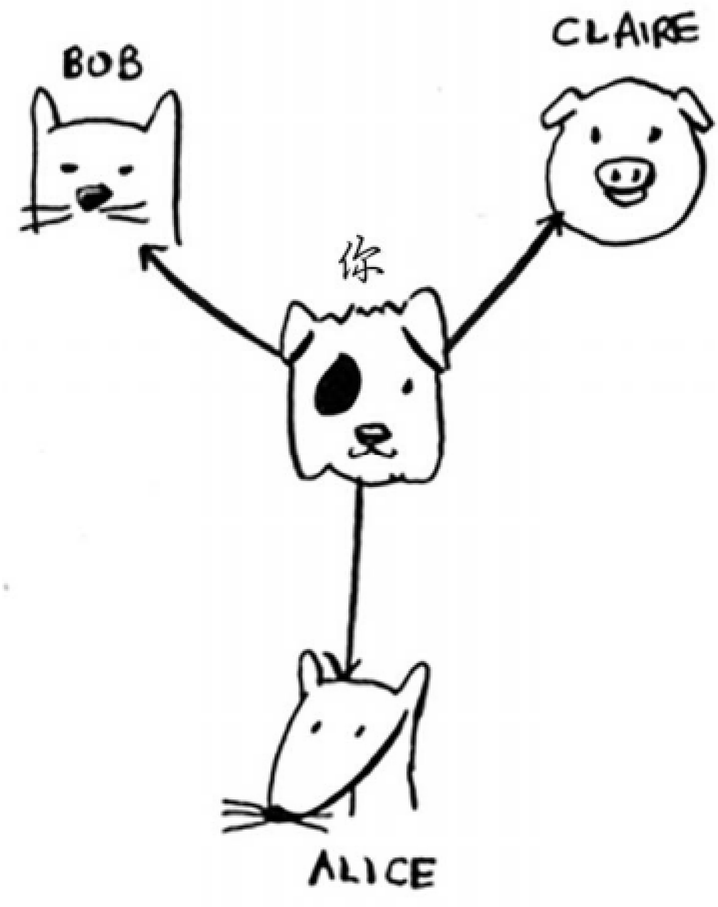
- 从节点 `A` 出发，有前往节点 `B` 的路径吗？
- 从节点 `A` 出发，前往节点 `B` 的哪条路径最短？

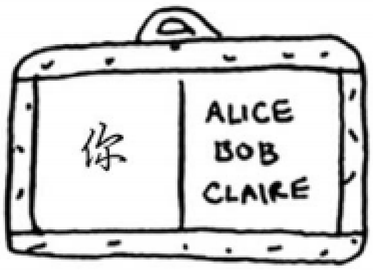
在广度优先搜索的执行过程中，搜索范围从起点开始逐渐向外延伸，即先检查一度关系，再检查二度关系。
你也可以这样看，一度关系在二度关系之前加入查找名单。
广度优先搜索不仅查找从 `A` 到 `B` 的路径，而且找到的是最短的路径。
注意，只有按添加顺序查找时，才能实现这样的目的。
有一个可实现这种目的的数据结构，那就是**队列** `queue`。

队列 `queue` 只支持两种操作：入队 和 出队。
队列是一种**先进先出** `First In First Out，FIFO` 的数据结构。
而栈是一种**后进先出** `Last In First Out，LIFO` 的数据结构。

6.4 实现图

可以使用散列表来表示图。

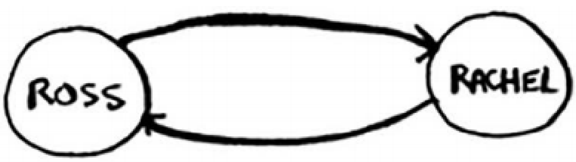





```
graph = {}
graph['you'] = ['alice', 'bob', 'claire']
```

散列表是无序的，因此添加键-值对的顺序无关紧要。

有向图 `directed graph` ，其中的关系是单向的。有向图的边是箭头。
无向图 `undirected graph` ，没有箭头，直接相连的节点互为邻居。无向图的边没有箭头，其中的关系是双向的。
下面两个图是等价的：



有向图

无向图

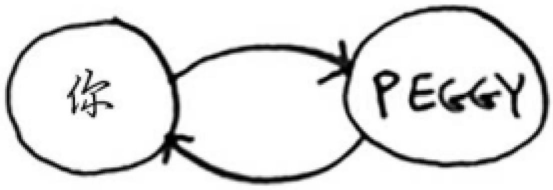
6.5 实现算法

更新队列时，我使用术语“**入队**”和“**出队**”，但你也可能遇到术语“**压入**”和“**弹出**”。
压入大致相当于入队，而弹出大致相当于出队。

使用函数 `collections.deque` 来创建一个双端队列。

> 【注】
> 来自百度百科：
> `double-ended queue`
> 双端队列是指允许两端都可以进行入队和出队操作的队列，其元素的逻辑结构仍是线性结构。将队列的两端分别称为前端和后端，两端都可以入队和出队。

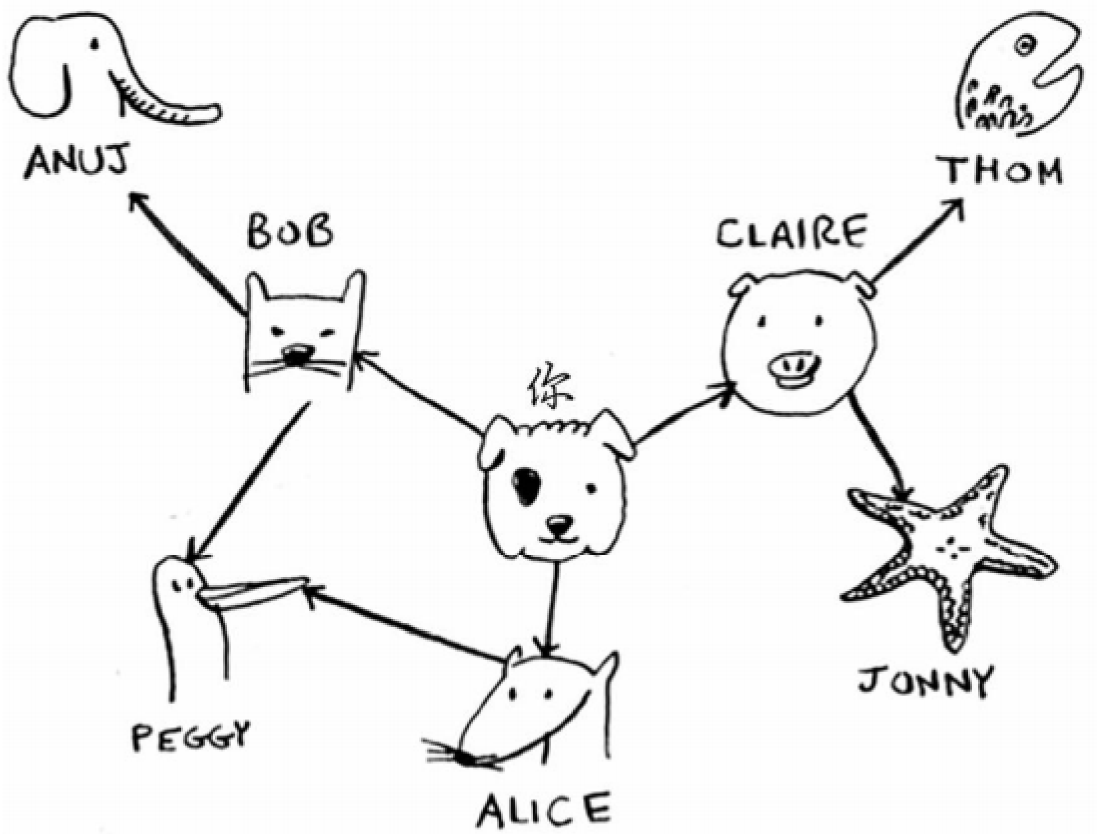
检查完一个人后，应将其标记为已检查，且不再检查他。
如果不这样做，就可能会导致无限循环。



运行时间
广度优先搜索的运行时间为 `O(人数 + 边数)`，这通常写作 `O(V + E)`，其中 `V` 为顶点（`vertice`）数，`E` 为边数

- 每条边都访问一次
- 每个人都要添加到需要检查的队列

广度优先搜索代码

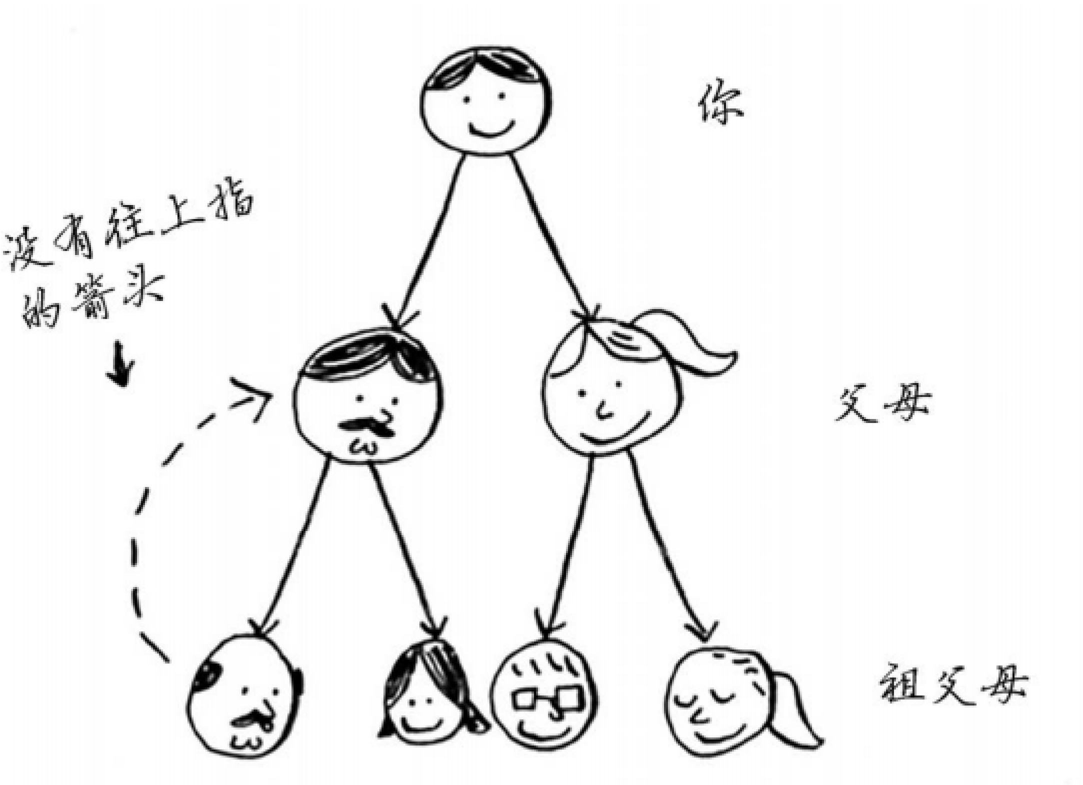


需要寻找芒果供应商（名字以 `'m'` 结尾的人）

```
from collections import deque

# 构建图
graph = {}
graph['you'] = ['alice', 'bob', 'claire']
graph['bob'] = ['anuj', 'peggy']
graph['alice'] = ['peggy']
graph['claire'] = ['thom', 'jonny']
graph['anuj'] = []
graph['peggy'] = []
graph['thom'] = []
graph['jonny'] = []


def is_mango_seller(person):
    """判断一个人是否为芒果供应商。"""
    # return True if person[-1] == 'm' else False
    # 更好的写法:
    return person[-1] == 'm'


def search_mango_seller(name):
    """寻找一个人的关系网中是否有芒果供应商。"""
    q = deque(graph[name])
    searched = []

    while q:
```

```python
            person = q.popleft()
            if person not in searched:
                if is_mango_seller(person):
                    return person
                else:
                    q.extend(graph[person])
                    searched.append(person)

        return None


    if __name__ == '__main__':
        print(search_mango_seller('you'))
```

拓扑排序：根据图创建一个有序列表
树：是一种特殊的图，其中没有往后指的边。



树是图的子集，所以树都是图，但图不一定是树。

6.6 小结

- 你需要按加入顺序检查搜索列表中的人，否则找到的就不是最短路径，因此**搜索列表必须是队列。**
- **对于检查过的人，务必不要再去检查**，否则可能导致无限循环。

---

7. 狄克斯特拉算法

狄克斯特拉算法 `Dijkstra's algorithm`

7.1 使用狄克斯特拉算法

狄克斯特拉算法找出**总权重最小的路径。**

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。

7.2 术语

狄克斯特拉算法用于每条边都有关联数字的图，这些数字称为**权重**（ `weight` ）。
带权重的图称为**加权图**（ `weighted graph` ），不带权重的图称为**非加权图**（ `unweighted graph` ）。

要计算非加权图中的最短路径，可使用**广度优先搜索。**
要计算加权图中的最短路径，可使用**狄克斯特拉算法。**

无向图意味着两个节点彼此指向对方，其实就是环！
在无向图中，每条边都是一个环。狄克斯特拉算法只适用于**有向无环图**（ `directed acyclic graph` ， `DAG` ）。

7.3 换钢琴

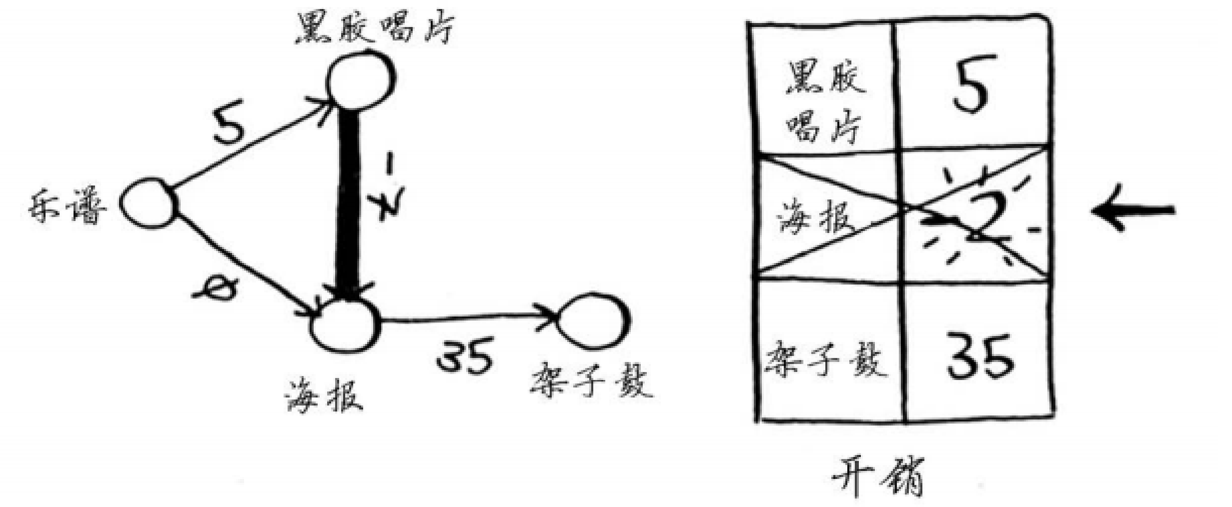
这就是狄克斯特拉算法背后的关键理念： 找出图中最便宜的节点，并确保没有到该节点的更便宜的路径！

- 创建一个表格，在其中列出每个节点的开销
- 还要添加表示**父节点**的列

通过**沿父节点回溯**，便得到了完整的交换路径。

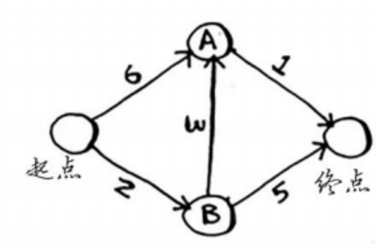
7.4 负权边

如果有负权边，就不能使用狄克斯特拉算法。
海报节点已处理过，这里却更新了它的开销。这是一个危险信号。
节点一旦被处理，就意味着没有前往该节点的更便宜途径，但你刚才却找到了前往海报节点的更便宜途径！

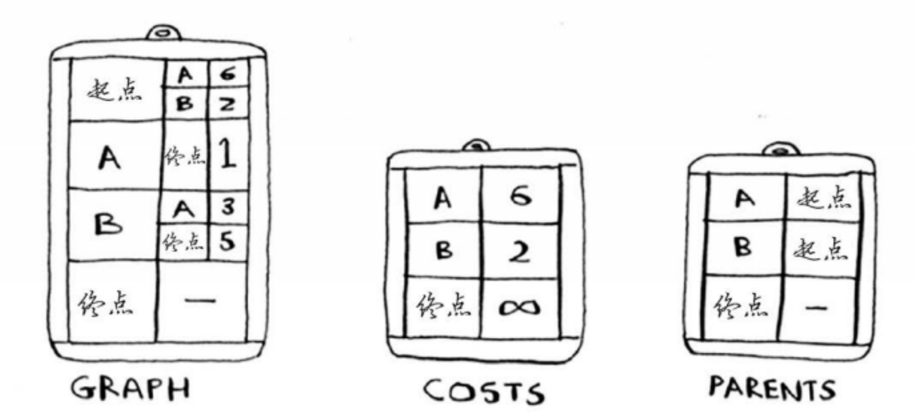


不能将狄克斯特拉算法用于包含负权边的图。
在包含负权边的图中，要找出最短路径，可使用另一种算法——**贝尔曼-福德算法**（`Bellman-Ford algorithm`）。

7.5 实现



要编写解决这个问题的代码，需要三个散列表。




```
# 创建图
```

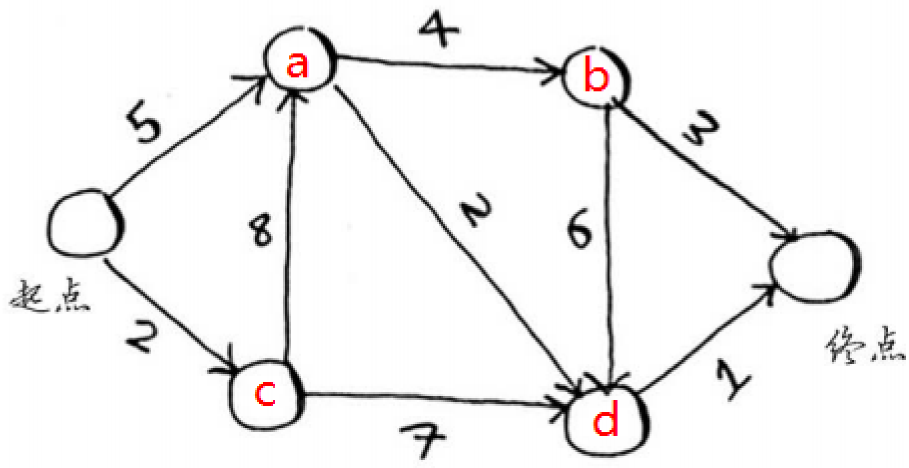
```python
graph = {}
# start
graph['start'] = {}
graph['start']['a'] = 6
graph['start']['b'] = 2
# A
graph['a'] = {}
graph['a']['end'] = 1
# B
graph['b'] = {}
graph['b']['a'] = 3
graph['b']['end'] = 5
# end
graph['end'] = {}   # 终点没有任何邻居

# 创建开销的散列表
inf = float('inf')
costs = {}
costs['a'] = 6
costs['b'] = 2
costs['end'] = inf

# 创建父节点散列表
parents = {}
parents['a'] = 'start'
parents['b'] = 'start'
parents['end'] = None

# 记录处理过的节点
processed = []


def find_lowest_cost_node(costs):
    """找到还未被处理过的最低开销节点。"""
    lowest_cost = 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


# 狄克斯特拉算法
while True:
    # 1. 寻找最小开销节点，获得最小开销
    lowest_cost_node = find_lowest_cost_node(costs)
    if lowest_cost_node is None:
        break
```

```python
        lowest_cost = costs[lowest_cost_node]
        # 2. 找到最小开销节点的邻居
        neighbors = graph[lowest_cost_node]
        # 3. 计算到每个邻居的开销
        for neighbor, neighbor_cost in neighbors.items():
            to_neighbor_cost = lowest_cost + neighbor_cost
            # 4. 如果到邻居的开销比开销表中存储的开销低，
            #     更新最低开销，并设置邻居的父节点为当前最小开销节点
            if to_neighbor_cost < costs[neighbor]:
                costs[neighbor] = to_neighbor_cost
                parents[neighbor] = lowest_cost_node
        # 5. 追加已处理节点列表
        processed.append(lowest_cost_node)


def print_lowest_cost_path(parents):
    path = ['end']
    while path[0] != 'start':
        for k, v in parents.items():
            if path[0] == k:
                path.insert(0, v)
    print(' >>> '.join(path))


print_lowest_cost_path(parents)

"""
start >>> b >>> a >>> end
"""
```

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。

7.7 作业补充



1. 最短路径是哪一条？
2. 总权重为多少？

```
"""
此处采用算法动画图解 APP 的方式，
构建开销表的时候除了起点，
其余节点开销都设置为正无穷，
起点的开销为 0。
"""
import time

# 构建图
graph = {}
# start
graph['start'] = {}
graph['start']['a'] = 5
graph['start']['c'] = 2
# a
graph['a'] = {}
graph['a']['b'] = 4
graph['a']['d'] = 2
# b
graph['b'] = {}
graph['b']['end'] = 3
graph['b']['d'] = 6
# c
graph['c'] = {}
graph['c']['a'] = 8
graph['c']['d'] = 7
# d
graph['d'] = {}
graph['d']['end'] = 1
# end
graph['end'] = {}

# 构建开销表
inf = float('inf')
costs = {}.fromkeys(['start', 'a', 'b', 'c', 'd', 'end'], inf)
```

```python
costs['start'] = 0

# 构建父节点表
parents = {}.fromkeys(['a', 'b', 'c', 'd', 'end'])


def get_lowest(costs, processed):
    """返回开销表中开销最低的节点和最低开销。"""
    lowest_node = None
    lowest_cost = inf
    for node, cost in costs.items():
        if node in processed:
            continue
        if cost < lowest_cost:
            lowest_cost = cost
            lowest_node = node
    return lowest_node, lowest_cost


def dijkstra():
    """使用狄克斯特拉算法，返回路径和最低开销。"""
    # 记录已处理节点
    processed = []

    while True:
        # 找到开销表中开销最低的节点和最低开销
        l_node, l_cost = get_lowest(costs, processed)
        if l_node is None:
            break
        # 遍历该节点的邻居，计算并更新开销
        neighbors = graph[l_node]
        for neighbor, neighbor_cost in neighbors.items():
            new_cost = l_cost + neighbor_cost
            if new_cost < costs[neighbor]:
                costs[neighbor] = new_cost
                parents[neighbor] = l_node
        # 追加已处理节点
        processed.append(l_node)

    # 最低开销路径
    path_li = ['end']
    while path_li[0] != 'start':
        for child, parent in parents.items():
            if child == path_li[0]:
                path_li.insert(0, parent)
    path = ' >>> '.join(path_li)
    path_cost = costs['end']
    return path, path_cost


if __name__ == '__main__':
    path, path_cost = dijkstra()
```

```
    print('最低开销路径为: %s' % path)
    print('最低开销为: %d' % path_cost)
"""
最低开销路径为: start >>> a >>> d >>> end
最低开销为: 8
"""
```

---

8. 贪婪算法

8.1 教室调度问题

贪婪算法的优点：简单易行！
贪婪算法很简单：每步都采取最优的做法。
用专业术语说，就是你**每步都选择局部最优解**。

8.2 背包问题

从这个示例你得到了如下启示：在有些情况下，完美是优秀的敌人。有时候，你只需找到一个能够大致解决问题的算法，此时贪婪算法正好可派上用场，因为它们实现起来很容易，得到的结果又与正确结果相当接近。

8.3 集合覆盖问题

列出每个可能的广播台集合，这被称为幂集 `power set`。
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。
在这个例子中，贪婪算法的运行时间为 `O(n2)`，其中 `n` 为广播台数量。

```
# 集合覆盖问题
# 需要覆盖的州
states_needed = set(['mt', 'wa', 'or', 'id', 'nv', 'ut', 'ca', 'az'])

# 广播台清单
stations = {}
stations['kone'] = set(['id', 'nv', 'ut'])
stations['ktwo'] = set(['wa', 'id', 'mt'])
stations['kthree'] = set(['or', 'nv', 'ca'])
stations['kfour'] = set(['nv', 'ut'])
```

```python
stations['kfive'] = set(['ca', 'az'])

# 最终选择的广播台
final_stations = set()

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)

print(final_stations)
"""
{'kthree', 'kone', 'ktwo', 'kfive'}
"""
```

8.4 NP 完全问题

`NP` 完全问题的简单定义是：以难解著称的问题。
比如旅行商问题要考虑 `n!` 种情况

如何判断 `NP` 完全问题？

- 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。

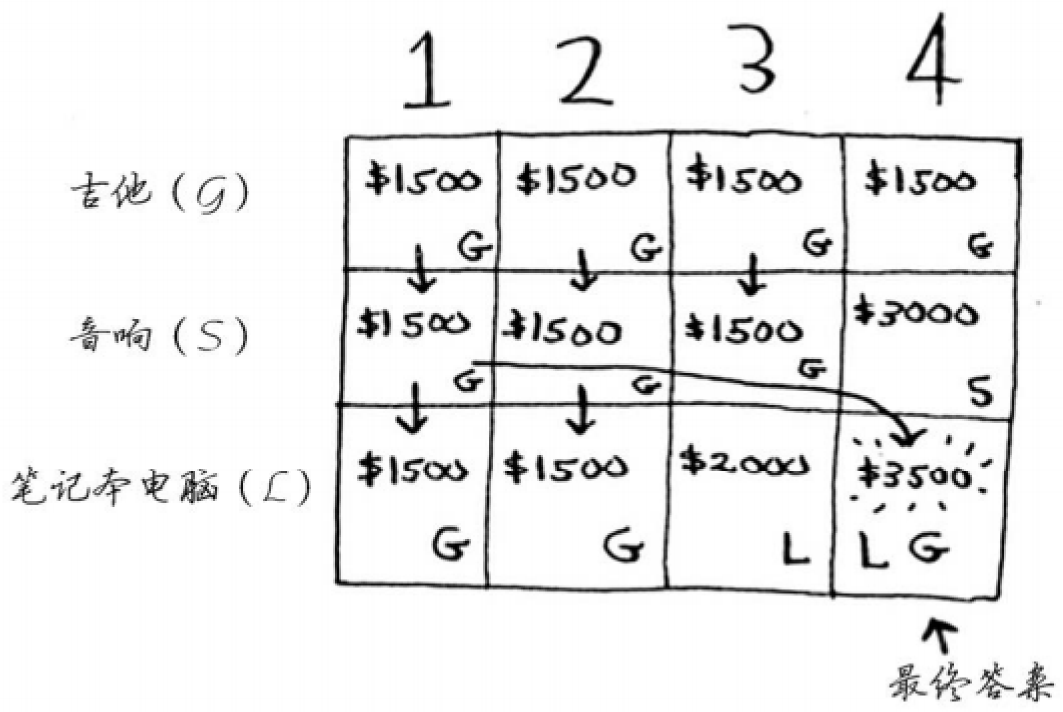
8.5 小结

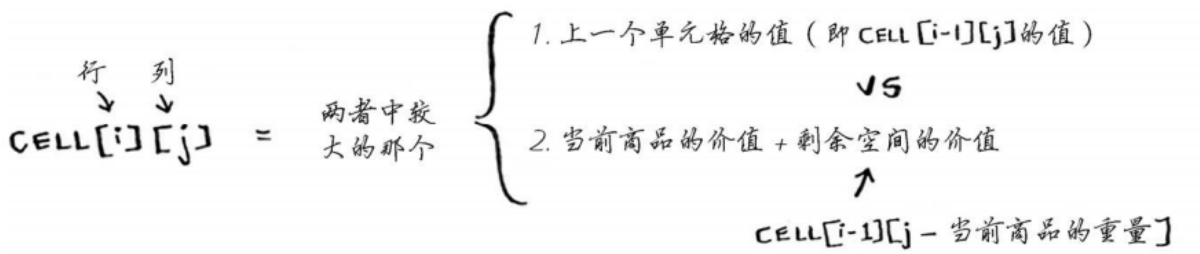
- 贪婪算法寻找局部最优解，企图以这种方式获得全局最优解。
- 对于 `NP` 完全问题，还没有找到快速解决方案。
- 面临 `NP` 完全问题时，最佳的做法是使用近似算法。
- 贪婪算法易于实现、运行速度快，是不错的近似算法。

---

9. 动态规划

9.1 背包问题





9.2 背包问题 FAQ

问题：沿着一列往下走时，最大价值有可能降低吗？
答案：不可能。每次迭代时，你都存储当前的最大价值。

问题：行的排列顺序发生变化时，答案会随之变化吗？
答案：答案没有变化。也就是说，各行的排列顺序无关紧要。

问题：可以逐列而不是逐行填充网格吗？
答案：就这个问题而言，这没有任何影响，但对于其他问题，可能有影响。

问题：增加一件更小的商品将如何呢？
答案：由于项链的加入，你需要考虑的粒度更细，因此必须调整网格。

	0.5	1	1.5	2	2.5	3	3.5	4
吉他								
音响								
笔记本电脑								
项链								

问题：可以偷商品的一部分吗？
答案：答案是没法处理。使用动态规划时，要么考虑拿走整件商品，要么考虑不拿，而没法判断该不该拿走商品的一部分。但使用贪婪算法可轻松地处理这种情况！

问题：旅游行程最优化。去伦敦度假，假期两天。

名胜	时间	评分
威斯敏斯特教堂	0.5天	7
环球剧场	0.5天	6
英国国家美术馆	1天	9
大英博物馆	2天	9
圣保罗大教堂	0.5天	8

	1/2	1	1 1/2	2
威斯敏斯特教堂				
环球剧场				
英国国家美术馆				
大英博物馆				
圣保罗大教堂				

答案：

	1/2	1	1 1/2	2
W 威斯敏斯特教堂	7 W	7 W	7 W	7 W
H 环球剧场	7 W	13 W/H	7 W/H	7 W/H
Y 英国国家美术馆	7 W	13 W/H	16 W/Y	22 W/H/Y
D 大英博物馆	7 W	13 W/H	16 W/Y	22 W/H/Y
S 圣保罗大教堂	8 S	15 W/S	21 W/H/S	24 W/Y/S

问题：如何处理相互依赖的情况？假设你还想去巴黎，因此在前述清单中又添加了几项。
回答：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。

问题：计算最终的解时会涉及两个以上的子背包吗？
回答：据动态规划算法的设计，最多只需合并两个子背包，即根本不会涉及两个以上的子背包。

问题：最优解可能导致背包没装满吗
回答：完全可能。

练习
假设你要去野营。你有一个容量为6磅的背包，需要决定该携带下面的哪些东西。其中
每样东西都有相应的价值，价值越大意味着越重要：

- 水（重3磅，价值10）；

- 书（重1磅，价值3）
- 食物（重2磅，价值9）；
- 夹克（重2磅，价值5）；
- 相机（重1磅，价值6）。

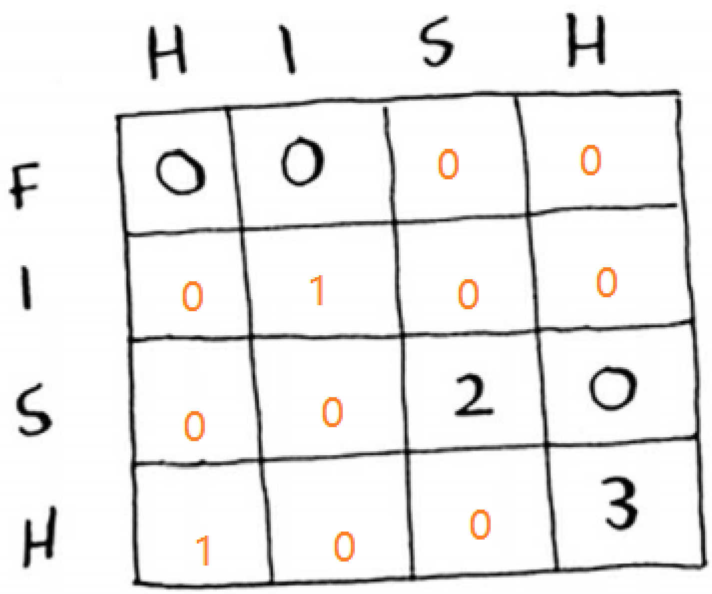
请问携带哪些东西时价值最高？

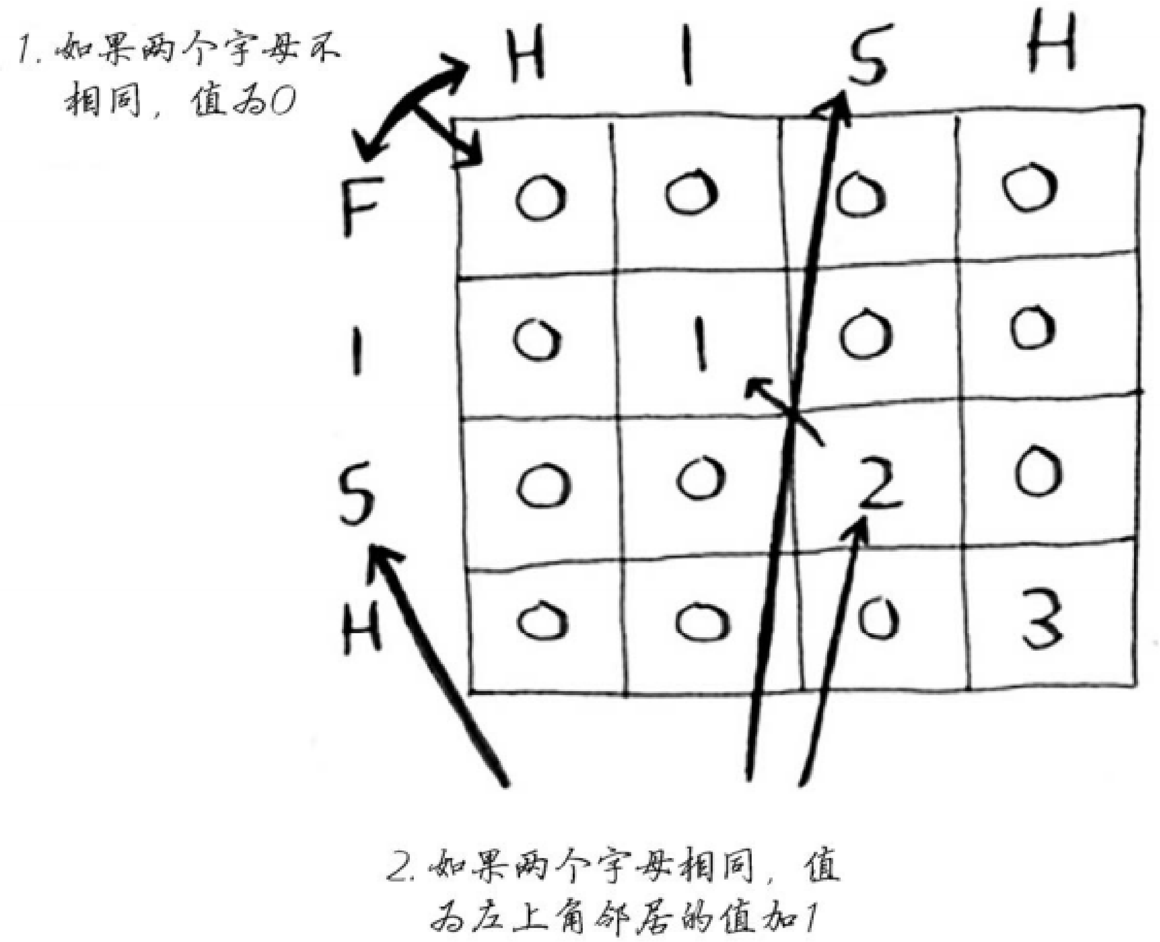
9.3 最长公共子串

- 动态规划可帮助你在给定约束条件下找到最优解。
- 在问题可分解为彼此独立且离散的子问题时，就可使用动态规划来解决。
- 每种动态规划解决方案都涉及网格。
- 单元格中的值通常就是你要优化的值。
- 每个单元格都是一个子问题，因此你应考虑如何将问题分成子问题，这有助于你找出网格的坐标轴。

计算机科学家有时会开玩笑说，那就使用费曼算法（ Feynman algorithm ）。这个算法是以著名物理学家理查德·费曼命名的，其步骤如下。
(1) 将问题写下来。
(2) 好好思考。
(3) 将答案写下来。

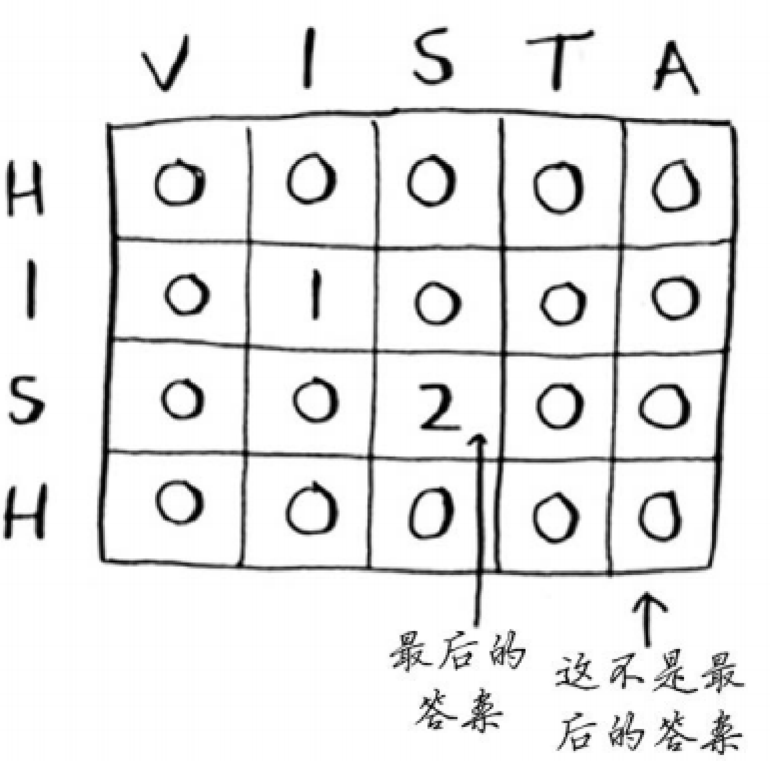
> 【注】：666！

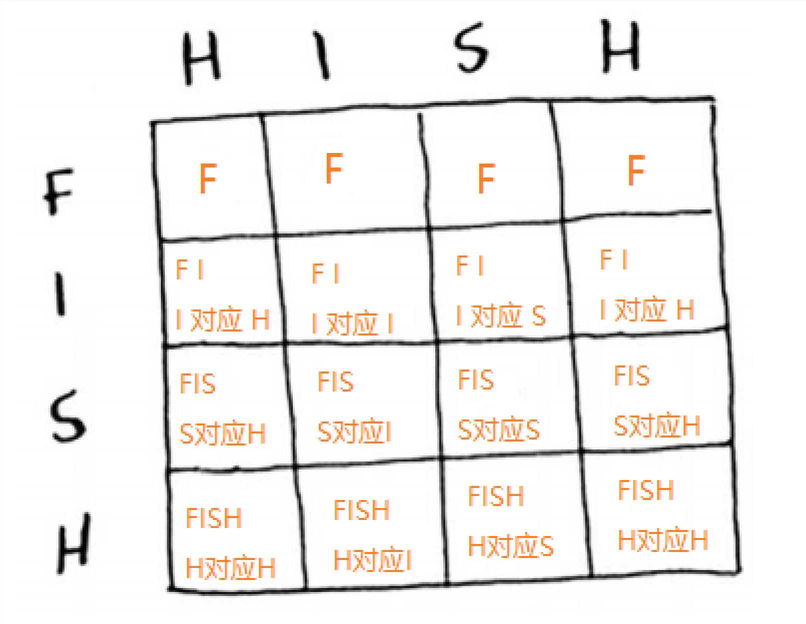




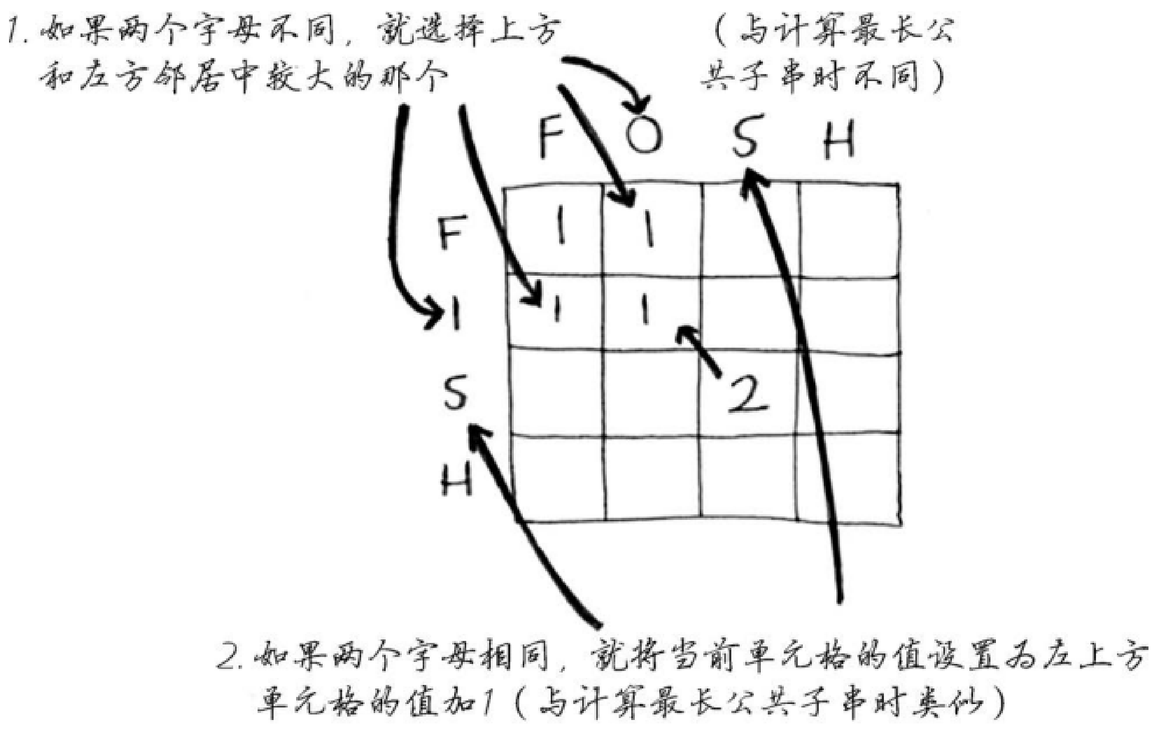
伪代码

```
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
```



> 【注】
> 我的想法如下图（理解的**不对**！）
>
> 
>

最长公共子序列



`伪代码`

```
if word_a[i] == word_b[j]:
    cell[i][j] = cell[i-1][j-1] + 1
else:
    cell[i][j] = max(cell[i-1][j], cell[i][j-1])
```

应用

- 编辑距离 `levenshtein distance`

9.4 小结

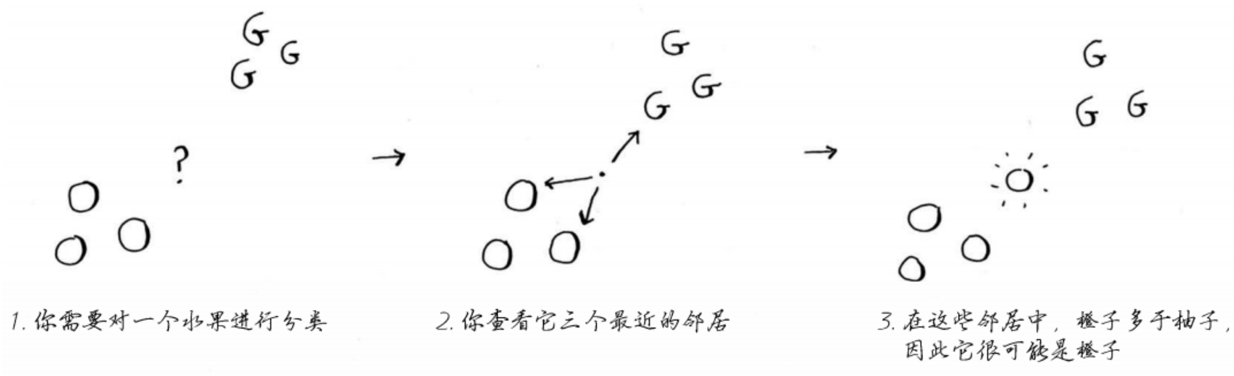
- 需要在给定约束条件下优化某种指标时，动态规划很有用。
- 问题可分解为离散子问题时，可使用动态规划来解决。
- 每种动态规划解决方案都涉及网格。
- 单元格中的值通常就是你要优化的值。
- 每个单元格都是一个子问题，因此你需要考虑如何将问题分解为子问题。
- 没有放之四海皆准的计算动态规划解决方案的公式。

---

10. K 最近邻算法

10.1 橙子还是柚子

`K` 最近邻（`k-nearest neighbours, KNN`）算法

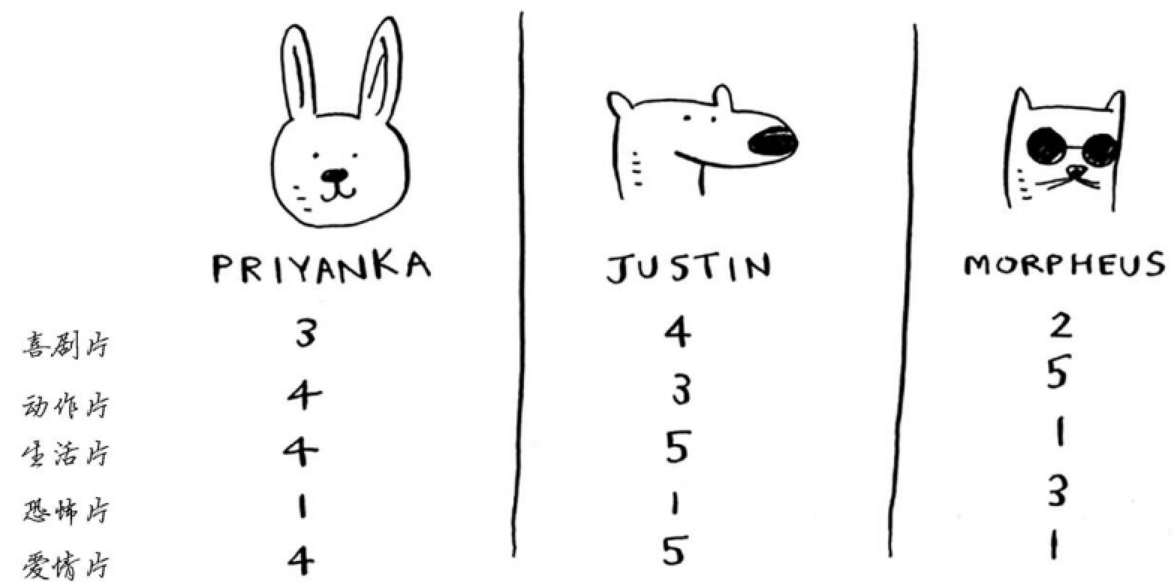


10.2 创建推荐系统

特征抽取
计算两者有多像，可以使用毕达哥拉斯公式（勾股定理）

$$\sqrt{(X_1-X_2)^2+(Y_1-Y_2)^2}$$

因此，你需要将每位用户都转换为一组坐标，就像前面对水果所做的那样。
在能够将用户放入图表后，你就可以计算他们之间的距离了。



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。

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

如果你是 `Netflix` 用户，`Netflix` 将不断提醒你：多给电影评分吧，你评论的电影越多，给你的推荐就越准确。现在你明白了其中的原因：你评论的电影越多，`Netflix` 就越能准确地判断出你与哪些用户类似。

- 如果一个用户评分偏低，导致两个用户虽然喜好相同，但是不算邻居，这时可以用**归一化** `nomralization`，计算平均分然后使平均分相同。
- 如果某个用户的评分很重要，那么可以给该用户加权重。

回归
预测某用户给电影打多少分。
可以选 `5` 个最近邻（注意 `5` 个可以随意取，所以叫 `K` 最近邻），看看他的邻居打多少分，然后求出平均值。
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）。

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。
在实际工作中，经常使用余弦相似度 `cosine similarity`。
余弦相似度不计算两个矢量的距离，而比较它们的角度。

`KNN` 特征抽取很重要：

- 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）。

`Netflix` 的用户数以百万计，前面创建推荐系统时只考虑了 `5` 个最近的邻居，这是太多还是太少了呢？
太少了，考虑的邻居太少可能出现偏差。
经验规则：`N` 位用户，考虑 `sqrt(N)` 个邻居。

10.3 机器学习简介

`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。

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！

一般而言，`OCR` 算法提取线段、点和曲线等特征。
与前面的水果示例相比，`OCR` 中的特征提取要复杂得多，但再复杂的技术也是基于 `KNN` 等简单理念的。这些理念也可用于语音识别和人脸识别。

`OCR` 的第一步是查看大量的数字图像并提取特征，这被称为**训练**（`training`）。大多数机器学习算法都包含训练的步骤：要让计算机完成任务，必须先训练它。

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`。
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。

使用机器学习来预测股票市场的涨跌真的很难。
在根据以往的数据来预测未来方面，没有万无一失的方法。未来很难预测，由于涉及的变数太多，这几乎是不可能完成的任务。

10.4 小结

- KNN用于**分类**和**回归**，需要考虑最近的邻居。
- **分类就是编组**。
- **回归就是预测结果**（如数字）。
- 特征抽取意味着将物品（如水果或用户）转换为一系列可比较的数字。
- 能否挑选合适的特征事关 `KNN` 算法的成败。

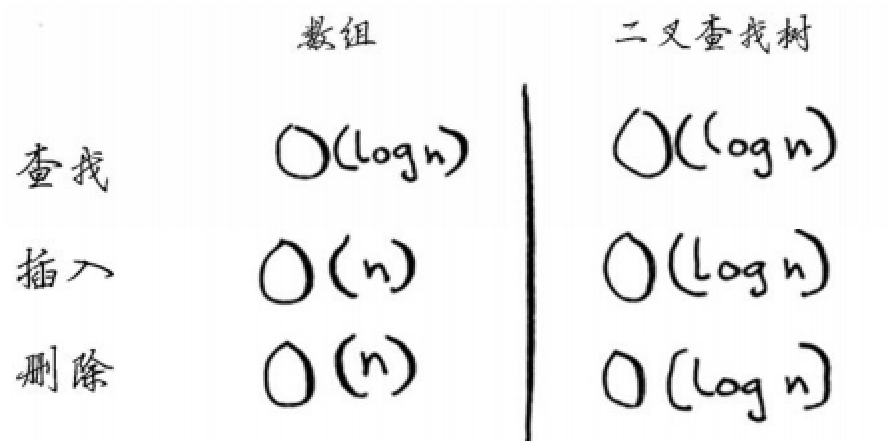
---

11. 接下来如何做

11.1 树

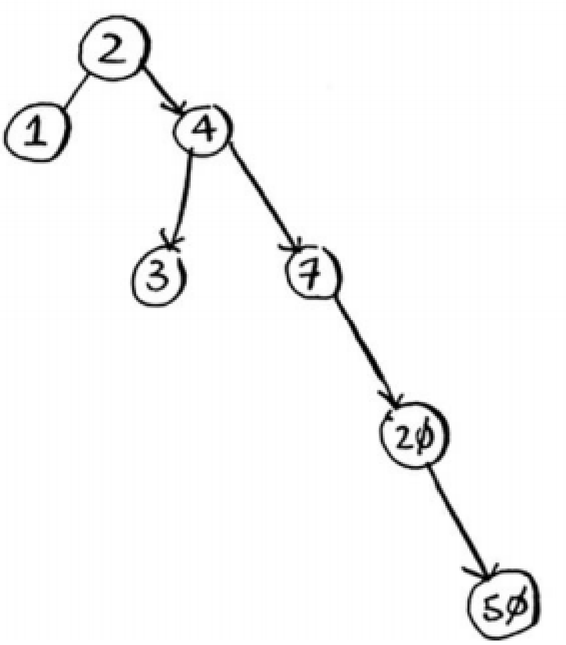
二叉查找树 `binary search tree`
对于其中的每个节点，**左子节点的值都比它小，右子节点的值都比它大**。
查找的时候，从根节点开始，如果大于根节点，往右找，如果小于根节点，往左找。
平均时间为 `O(log n)`，最坏时间为 `O(n)`。

有序数组最坏情况是 `O(log n)`。
但是二叉查找树的插入和删除速度快得多。



	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)

缺点：不能随机访问。
在二叉查找树处于平衡状态时，平均访问时间也为 `O(log n)`。
假如二叉树处于不平衡状态，则性能不佳。



也有一些处于平衡状态的特殊二叉查找树，如**红黑树。**
如果你对数据库或高级数据结构感兴趣，请研究如下数据结构：`B` 树，红黑树，堆，伸展树。

11.2 反向索引

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。

键是单词，值是包含单词的页面。

11.3 傅里叶变换

`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。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。

11.4 并行算法

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。

- 并行性管理开销。分配任务后合并也需要时间。
- 负载均衡。

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。**
`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。

> 【注】：
> 个人理解：**分布式算法**就是高级版的并行算法。

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。
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。
归并函数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。

11.6 布隆过滤器和 HyperLogLog

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。
布隆过滤器的优点在于占用的存储空间很少。

`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

安全散列算法 `secure hash algorithm` 缩写 `SHA` 函数。
给定一个字符串， `SHA` 返回其散列值。
对于每个不同的字符串， SHA生成的散列值都不同。
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。
你可使用 `SHA` 来判断两个文件是否相同。

`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。但你无法根据散列值推断出原始字符串。
`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。
最安全的密码散列函数是 `bcrypt`。

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。
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！
让攻击者无法通过比较散列值是否类似来破解密码。
希望散列函数是局部敏感的，可使用 `Simhash`。
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

`Diffie-Hellman` 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？

- 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。
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。
`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

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。
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！

11.11 结语

完成于 `2018.12.09 13:44`