计算思维通识教育

Computational Thinking

第4章 用计算思维求解问题

主讲人：曹轶臻

联系方式：caoyizhen@cuc.edu.cn

Programming is all about data structures and algorithms.
Data structures are used to hold data while algorithms are
used to solve the problem using that data.

CONTENTS

计算机与网络空间安全学院
School of Computer and Cyber Sciences

计算思维通识教育
Computational Thinking

02

查找特定的数据

计算机与网络空间安全学院
School of Computer and Cyber Sciences

2-1 问题概述

查找过程就是在给定的一列数据中寻找指定的数据及该数据在数列中的位置。

在 Python 中，可以用in运算符查找列表中的元素。

```
>>> 5 in [1,3,5,7,9]
True
>>> 'he' in 'hello world'
True
>>> '100' in [10,100,1000]
False
```

这很容易写，但其实有一个底层的操作替我们完成了这个工作。

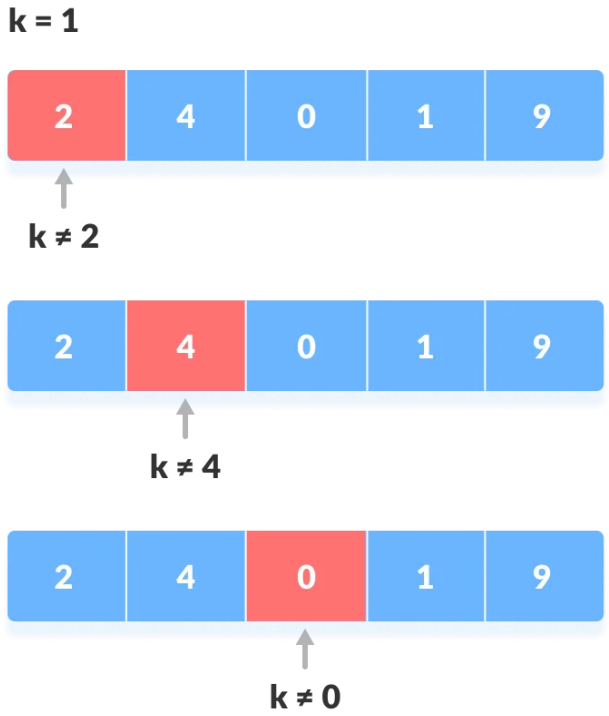
怎么实现in的功能？也就是如何实现序列中数据的查找？

2-1 顺序查找 (Linear Search)

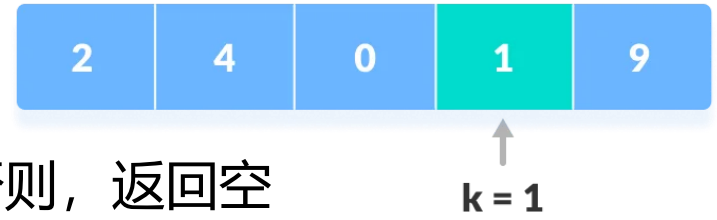
问题：在下面的列表中搜索k = 1的元素：


```
def find_num(n, arr):
    for i in range(0, len(arr)):
        if n == arr[i]:
            return i
    return None


my_array = [2, 4, 0, 1, 9]
index = find_num(int(input()), my_array)
if index is not None:
    print(f'number found at index {index}!')
else:
    print(f'number not found')
```

1. 从第一个元素开始，比较K与每个元素X



2. 如果X=K（找到了！），则返回其索引位置



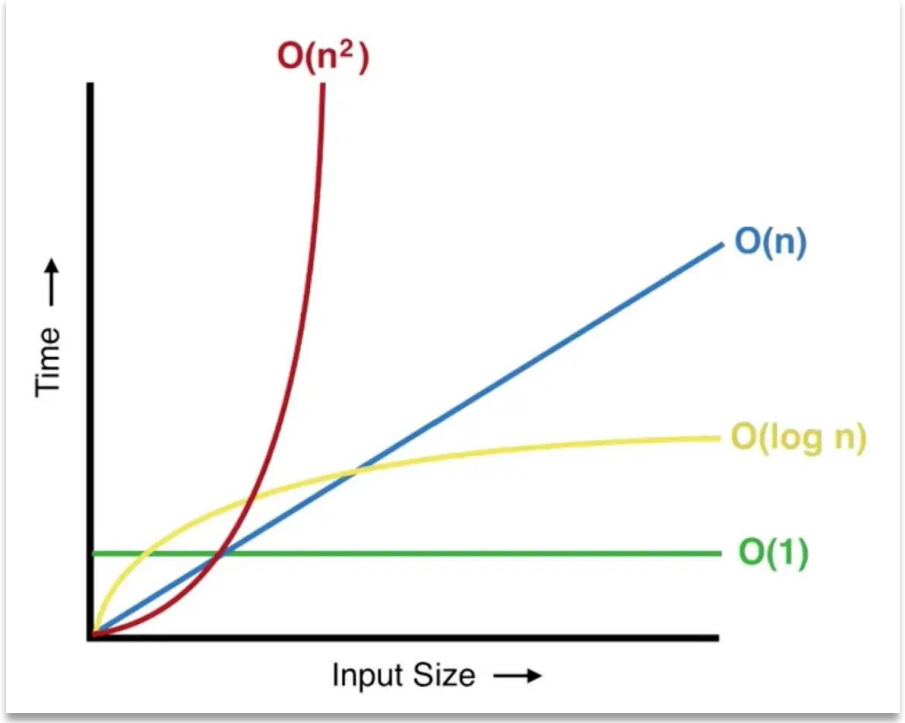
3. 否则，返回空

2-1 顺序查找 (Linear Search)

- 顺序查找没有难度和技术含量
- 顺序查找绝不仅限于对数字、字符的查找，也适用于其它对象的查找和匹配。

- 最好的情况是数组的第1个元素就是我们要找的元素，这样可以直接找到；
- 最坏的情况是到最后一个元素时才找到或者没有找到要找的元素；
- 因此其平均时间复杂度随规模增大而线性增大

在元素并不多的很多情况下，运用还是很广泛的。
因为没有必要为了有限数量的元素使用复杂的算法。

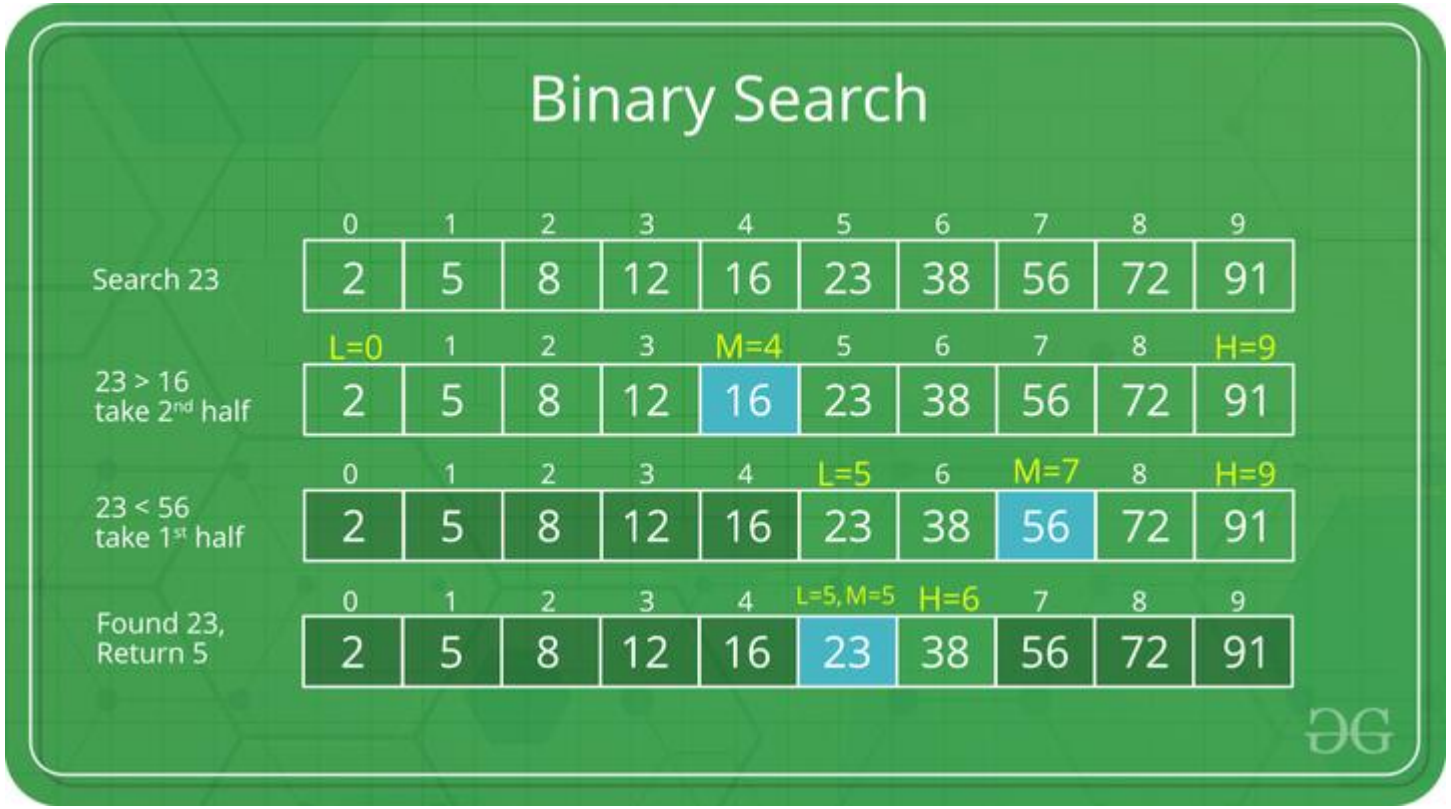


在计算机科学中，大O表示法表示程序的执行时间或占用空间随输入数据规模增长而增长的趋势。

2-2 二分查找 (Binary Search)

猜数字游戏

从1-1000之间随便想一个数字，让对方猜，给出数字大了或小了的提示，看谁用最短的次数来猜出数字。



最多只需要10次就可以猜中！

即使数字范围扩大到1-42亿，也只需要32次就可以猜中结果，是不是很快?

（如果用顺序查找，最坏的情况下，得猜42亿次！）

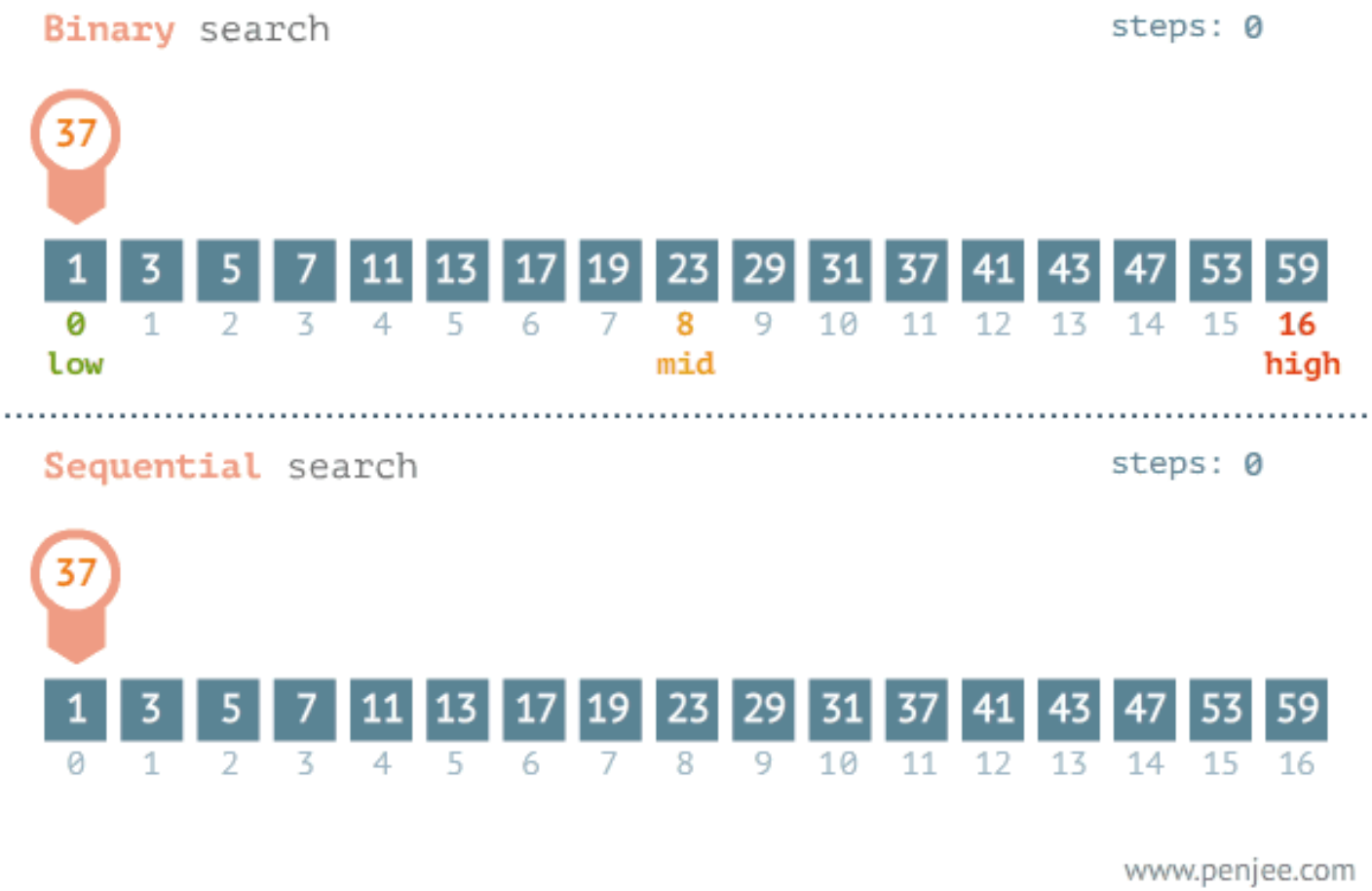
2-2 二分查找（Binary Search）

算法思想

二分查找
（折半查找）

VS

顺序查找



查找数字1，
各需要多少步？

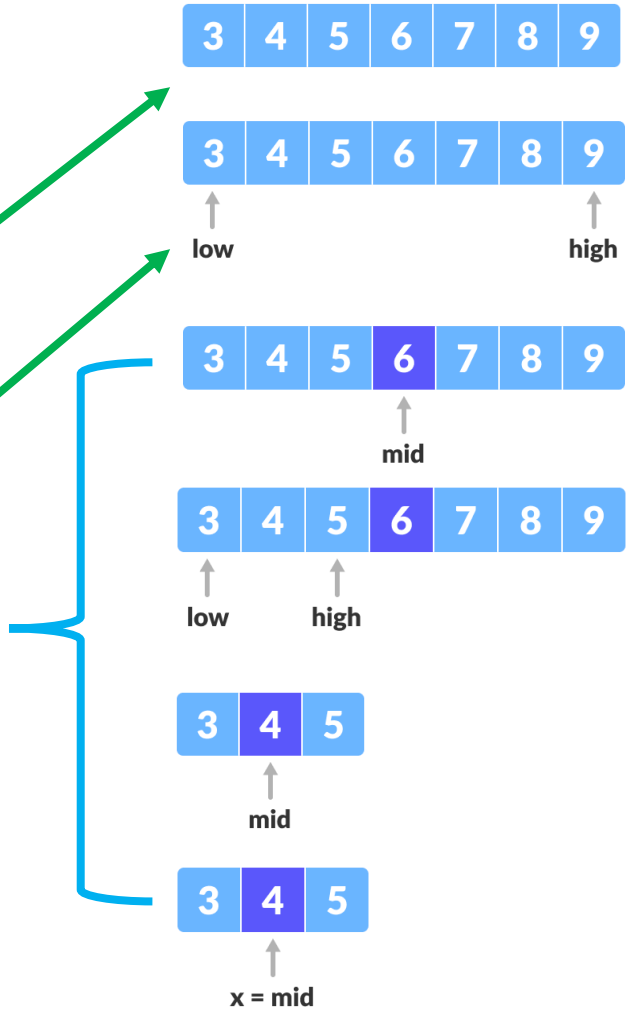
查找数字23，
各需要多少步？

图源：https://blog.penjee.com/binary-vs-linear-search-animated-gifs/

2-2 二分查找 (Binary Search)

算法实现

```
# Binary Search in python
def binarySearch(array, x, low, high):
    # Repeat until the pointers low and high meet each other
    while low <= high:
        mid = low + (high - low)//2
        if array[mid] == x:
            return mid
        elif array[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1

array = [3, 4, 5, 6, 7, 8, 9]
x = 4
result = binarySearch(array, x, 0, len(array)-1)
if result != -1:
    print("Element is present at index " + str(result))
else:
    print("Not found")
```



1. 初始数组，令变量x=4

2. 分别在最低和最高位置设置两个指针low(0)和high(6)

3. 找到数组的中间元素索引 mid，mid=(low+high)/2

4. 如果x==arr[mid]，则返回mid。否则，将要查找的元素与mid指向的元素进行比较

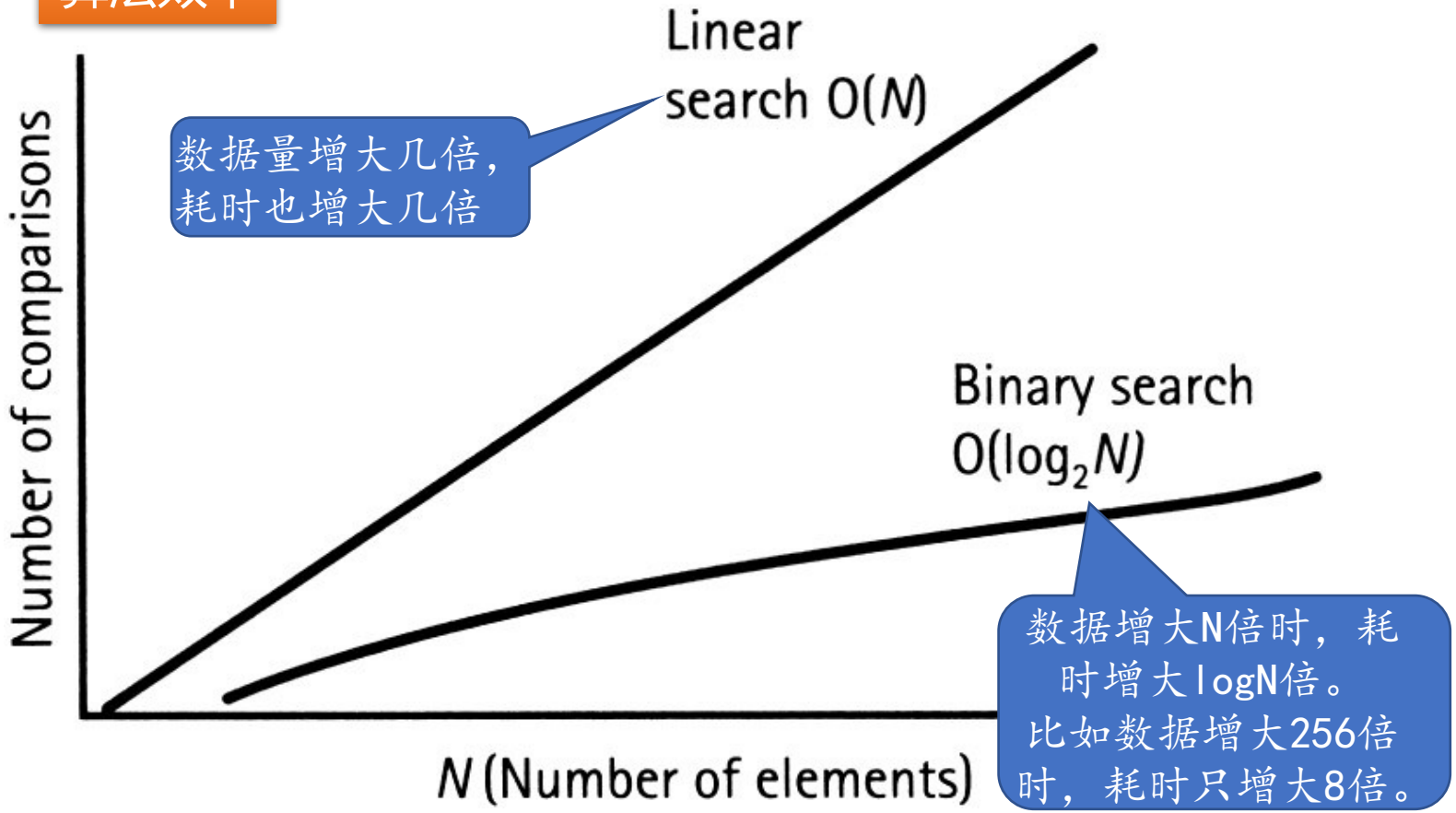
5. 如果x>中间元素，比较x和mid右边元素的中间元素

6. 否则，将x与mid左侧元素的中间元素进行比较

7. 重复步骤 3 到 6，直到low遇到high

2-2 二分查找（Binary Search）

算法效率



顺序查找与二分查找的时间复杂度

二分查找比较次数少、查找速度快、平均性能好

但是待查表必须为**有序数组**（若为无序数组，分成两份查找无意义，排序本身也耗费时间）

03

快速得到有序的数据

问题概述 3-1

选择排序 3-2

快速排序 3-3

If data is sorted then the search for information became easy and efficient.

3-1 问题概述

排序是数据处理中最常见的问题之一

排序算法（sorting algorithm）用于按特定顺序排列数组/列表的元素。

估计表明超过 25% 的计算时间花在了排序上。因此，找到有效的排序算法的问题非常重要。

有很多种不同的排序算法

- 选择排序 Selection Sort
- 冒泡排序 Bubble Sort
- 插入排序 Insertion Sort
- 快速排序 Quick Sort
- ……

没有一种“最佳的”排序算法是适用于所有序列的。（因为它们的初始顺序和大小都不同）

3-2 选择排序（Selection Sort）

算法思路

假设一个待排序序列，其中包含n个数据元素（假定均为数字类型）。从头到尾对元素进行扫描，找出其中最小的元素，放在序列的最前面；在其余的数据元素上，将上面的过程重复执行多次，直到只剩余1个待排序元素，排序过程结束。

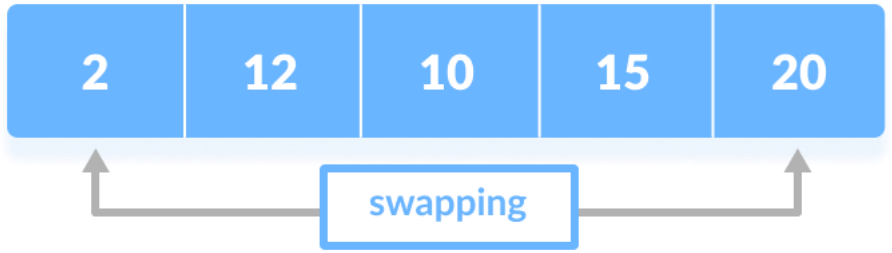
玩扑克牌拿到随机发放的牌，排序是乱的，我们一般会先把最大/最小的拿出来放在最左边，然后再把剩下的执行相同的操作，最后得到排好序的牌（升序或降序）。

3-2 选择排序 (Selection Sort)

算法过程

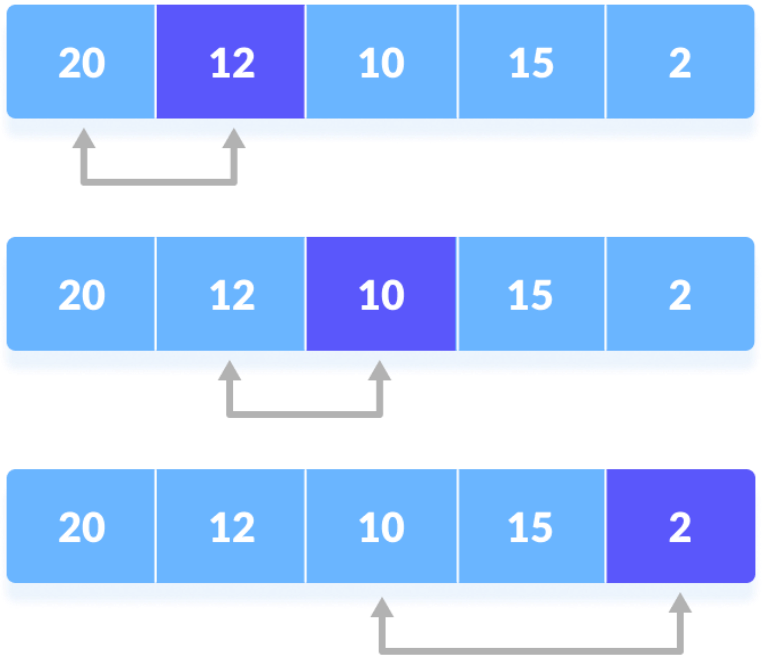
Step 1. 将第一个元素设为min(紫色表示)

Step 3. 把min被放在列表的最前端



Step 4. ???

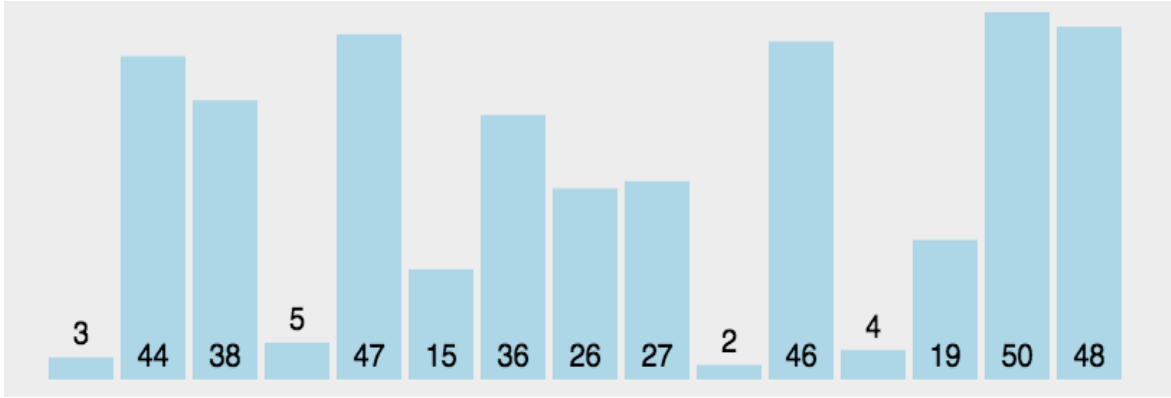
Step 2.
用min与第2个元素进行比较。
如果第2个元素小于min，则将min设为第2个元素，否则什么也不做。
接着用min与第3个元素进行比较，并设定min的值。
该过程一直持续到最后一个元素。

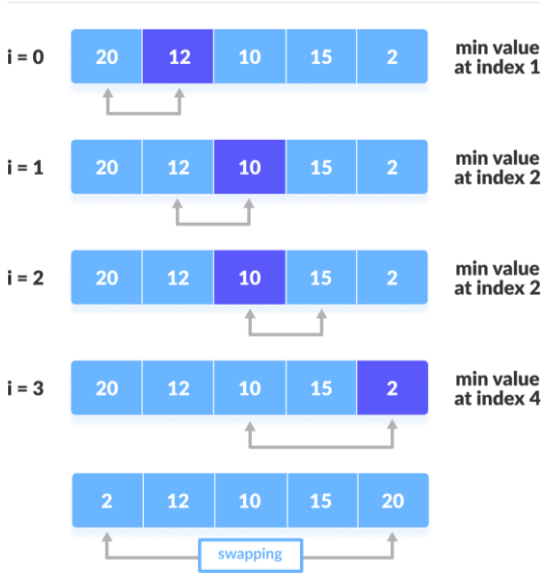


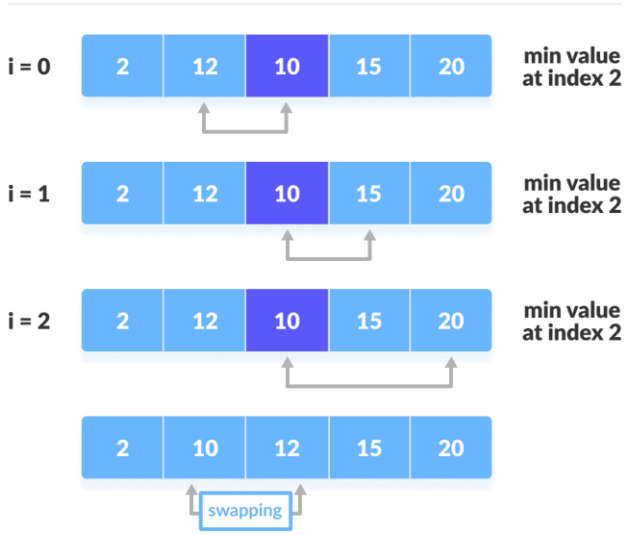
3-2 选择排序（Selection Sort）

算法过程

Step 4. 多次迭代（在未排序的元素上重复step1-step3），直到所有的元素都排在正确的位置上。







3-2 选择排序 (Selection Sort)

算法实现

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

伪代码

代码实现

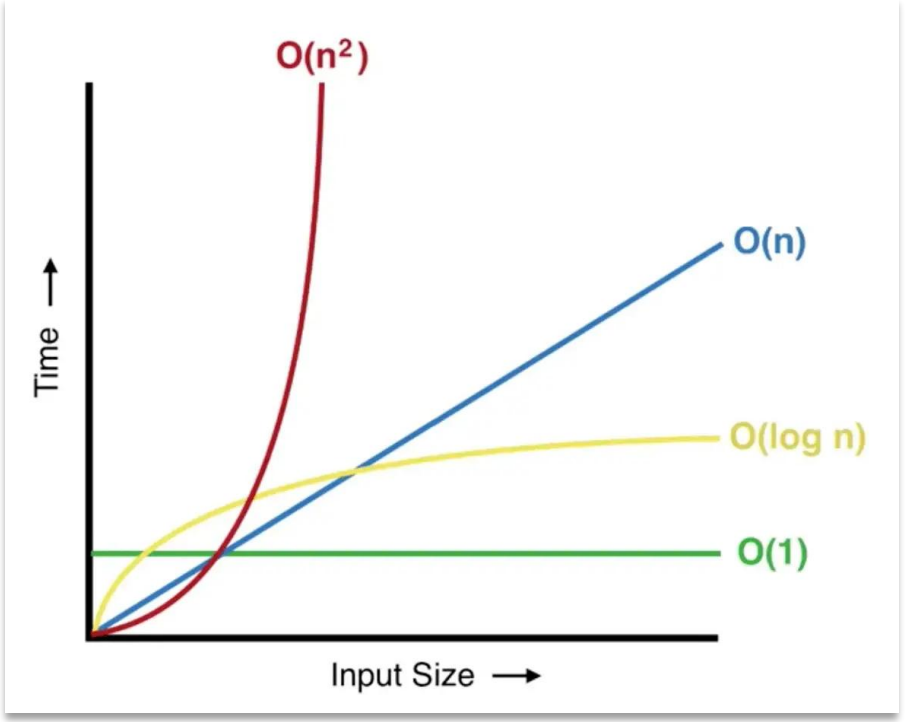
```
# Selection sort in Python
def selectionSort(array, size):
    for step in range(size):
        min_idx = step
        for i in range(step + 1, size):
            # select the minimum element in each loop
            if array[i] < array[min_idx]:
                min_idx = i
        # swap, put min at the correct position
        (array[step], array[min_idx]) = (array[min_idx], array[step])

data = [-2, 45, 0, 11, -9]
size = len(data)
selectionSort(data, size)
print('Sorted Array in Ascending Order:')
print(data)
```

效率高吗?

3-2 选择排序 (Selection Sort)

算法效率

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```



一个大小（规模）为n的列表，选择排序需要比较对比

n次、n-1次、n-2次……2次、1次

一共是n+(n-1)+(n-2)+...+2+1 = n*(n+1)/2次

即这个算法的时间复杂度是 $O(n^2)$

假如有10000个元素，需要执行上亿次指令！

3-3 快速排序（Quick Sort）

算法思路

基于分治法（Divide and Conquer）的排序算法。通过一趟排序将待排序记录分割成独立的两部分，其中一部分的数据均比另一部分数据小，则可以分别对这两部分数据递归地进行排序，以达到整个序列有序。

把一个复杂的问题**分解**成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接**求解**，原问题的解即子问题的解的**合并**。
这个技巧是很多高效算法的基础。

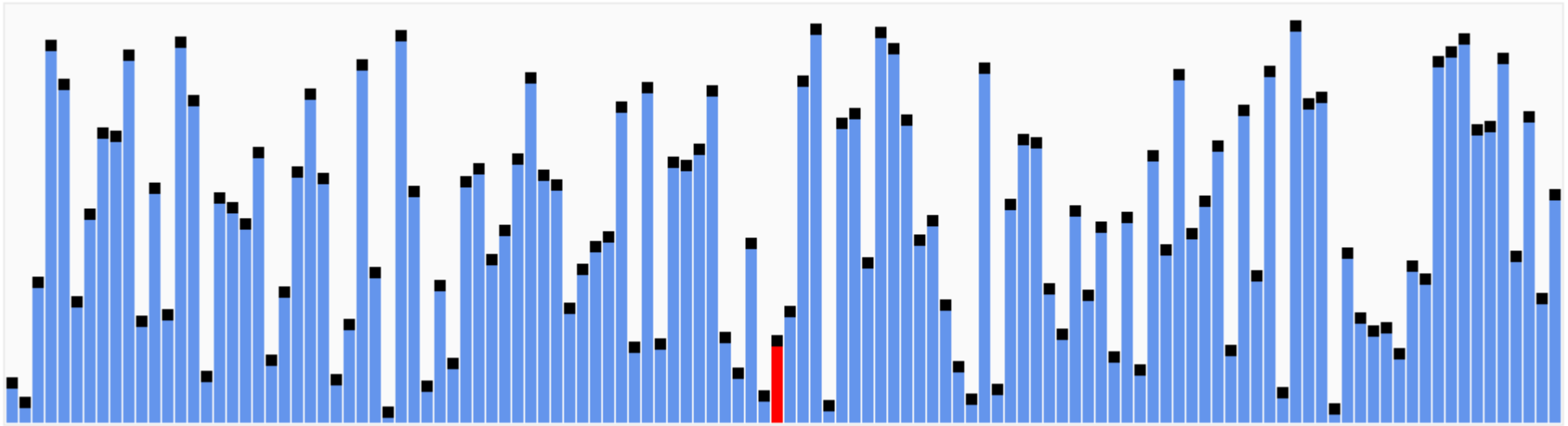
So it illustrates a basic concept of problem solving, which is divide and conquer, and it uses a computing technique known as recursion, to use – to get the computer to use the same method to – to solve the smaller problems as you have told it to solve the bigger problem.

-- Tony Hoare tells the story of how he came up with the idea for the Quicksort algorithm in 1960.

3-3 快速排序（Quick Sort）

算法过程

1. 先从数列中取出一个数作为基准数。(通常取第一个数或最后一个数或随机数)
2. 分区过程，将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边。
3. 再递归地对左右区间重复第二步，直到各区间只有一个数为止。
4. 最后，元素组合形成一个排序数组。



3-3 快速排序（Quick Sort）

算法过程

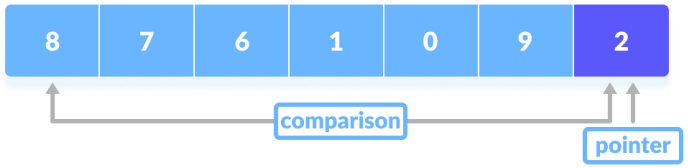
1. 选择基准数（pivot枢轴元素）

快速排序有不同的变体，pivot可从不同的位置选择。此例选择最右边的元素(2)为pivot。

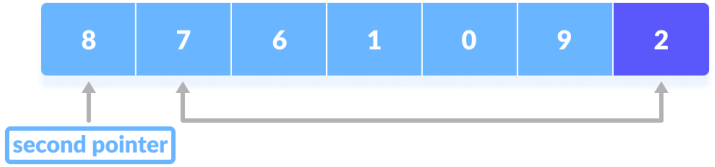
2.重新排列数组

列表的元素被重新排列，使得小于pivot的元素都放在左边，大于pivot的元素都放在右边。（分解）

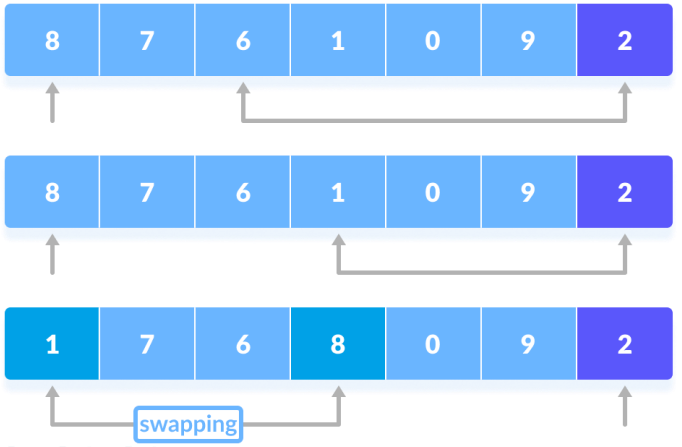
① 指针固定在pivot上。将pivot与第一个元素进行比较。



② 如果该元素大于pivot，则将第二指针指向该元素。



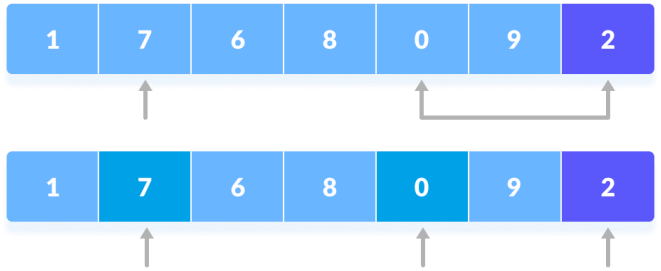
③ 将pivot与其他元素依次进行比较。如果到达小于pivot的元素，就将该元素与较早找到的较大元素交换。



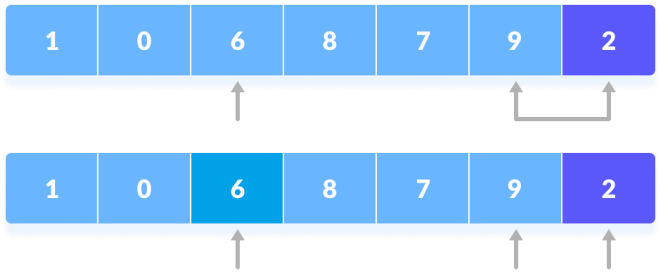
3-3 快速排序 (Quick Sort)

算法过程

④ 同样，重复该过程以将第二指针指向下一个更大的元素。并且，将它与另一个比pivot小的元素交换。



⑤ 该过程一直持续到到达倒数第二个元素。



⑥ 最后，pivot与第二指针位置元素交换。

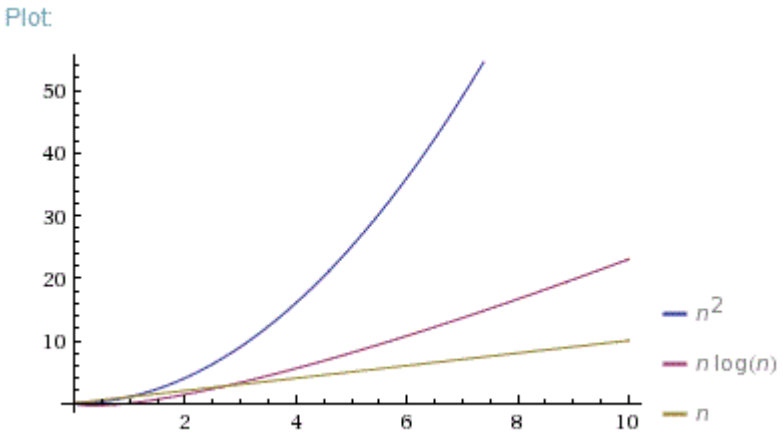
3. 再次分别为左右两个子部分选择pivot。并且，重复步骤2 。 (递归)

不断划分子序列，直到每个子序列都由单个元素组成。此时，数组已经排好序。

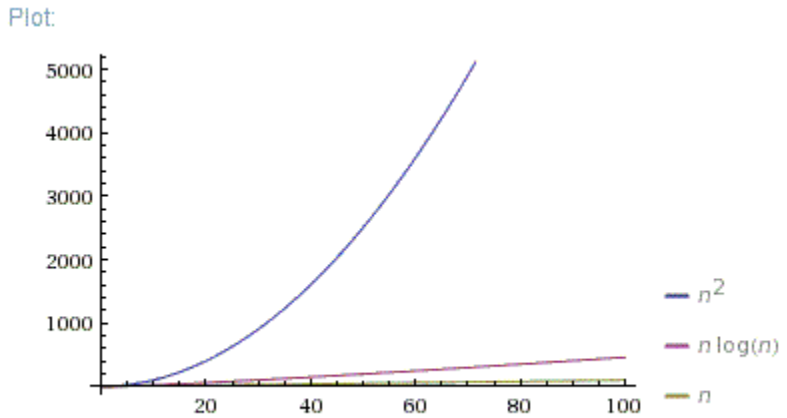
3-3 快速排序（Quick Sort）

算法效率

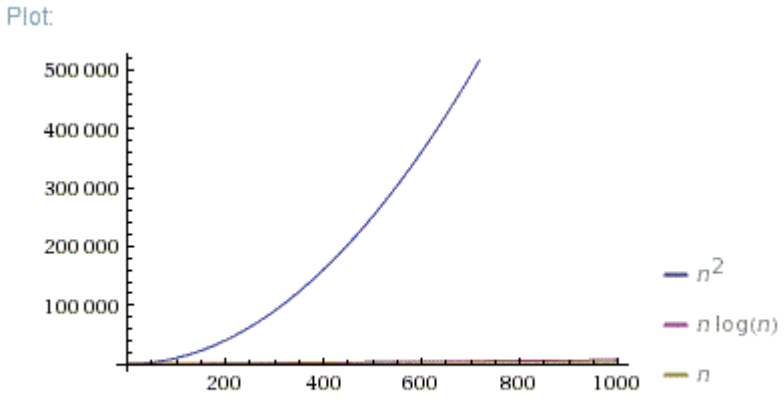
快速排序的时间复杂度是O(N*logN)



n = [0; 10]



n = [0; 100]



n = [0; 1000]

所以，快速排序的时间复杂度O(Nlog(N)) 比选择排序的时间复杂度$O(N^2)$ 好得多。它更接近 O(N)。

01 计算机求解复杂问题

02 查找特定的数据

03 快速得到有序的数据

- 如果一次排序后可以进行多次查找，那么排序的开销就可以摊薄。
- 但如果数据集经常变动，查找次数相对较少，那么可能还是直接用无序表加上顺序查找来得更经济。

CONTENTS

本次课程结束，请继续加油！

Computational Thinking

计算机与网络空间安全学院

School of Computer and Cyber Sciences