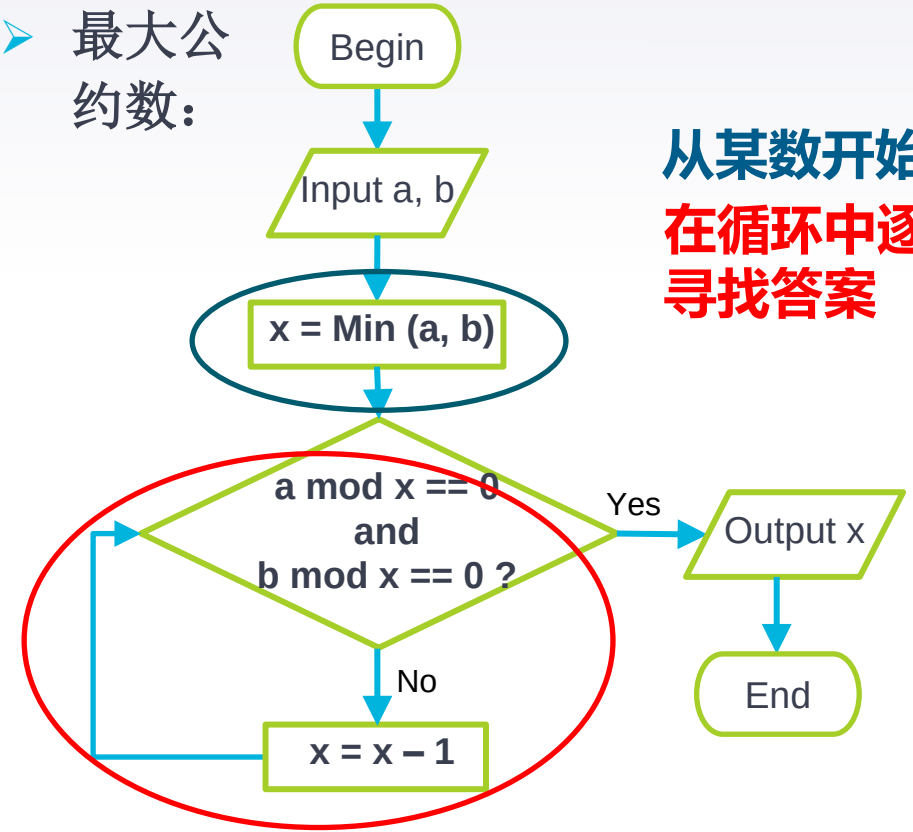
程式中常见的算法: Searching and Sorting

刘绍瑜
拉曼大学资讯与通讯科技学院
Liew Soung Yue
Faculty of Information and Communication Technology
Universiti Tunku Abdul Rahman

29/4/2023

搜索（Searching）

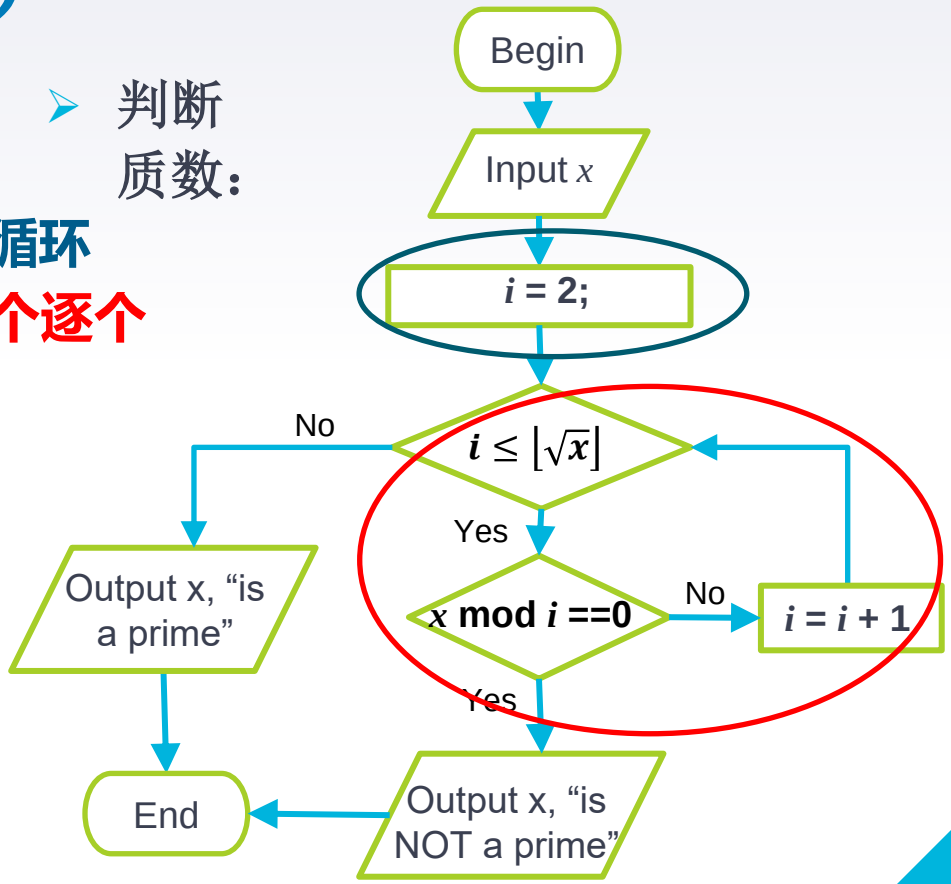
- 最大公约数：



从某数开始循环

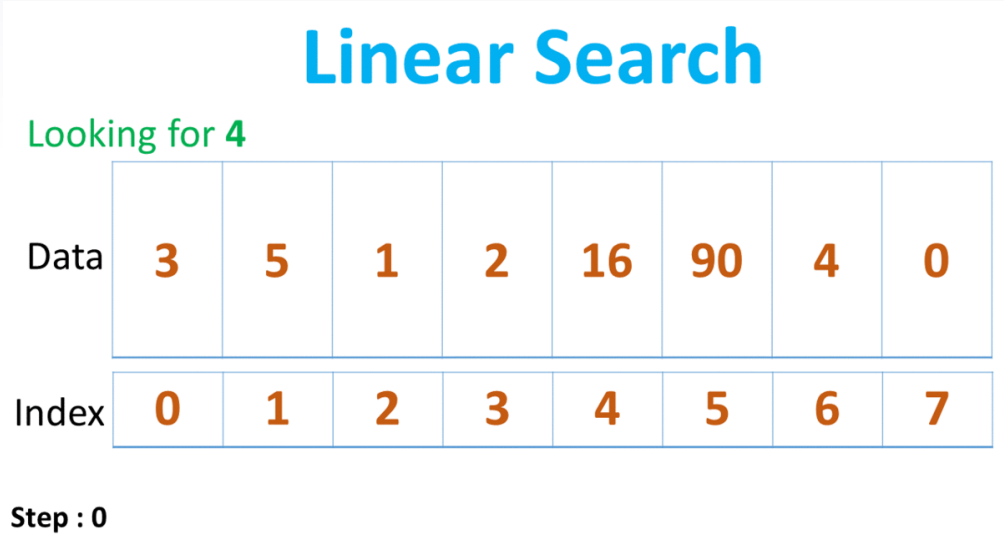
在循环中逐个逐个寻找答案

- 判断质数：



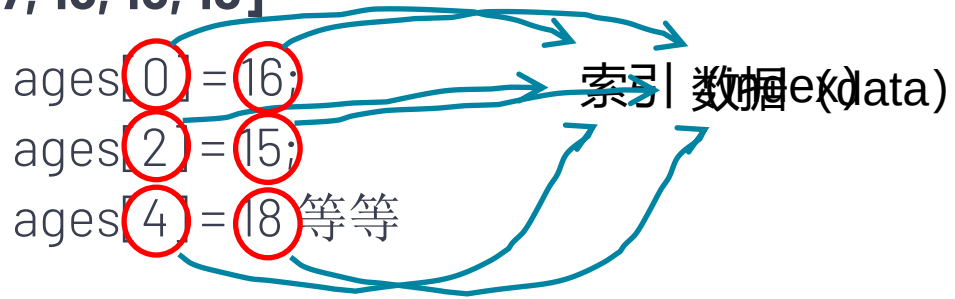
顺序/线性搜索（Linear Search）

- 是一种简单的搜索算法。
- 基本思想是从列表的开头（或末端）开始逐个比较元素，直到找到目标元素或搜索完整个列表（list）为止。
- 优点：实现简单、适用于小型列表以及在无序列表中进行搜索。



数组（Array）

- 数组（Array，也称为“阵列”）是一种非常基础和重要的数据结构（Data Structure）。
- 它可被用来存储多个相同数据类型的元素（element）。
- 这些元素按顺序排列在一起，每个元素都有一个对应的索引（index），可以通过索引来访问和操作数组中的元素。注意，索引从 0 开始。
- 例如，我们可以创建一个包含五个学生年龄的数组：
  ages = [16, 17, 15, 16, 18]
- 则



例子（线性搜索）：

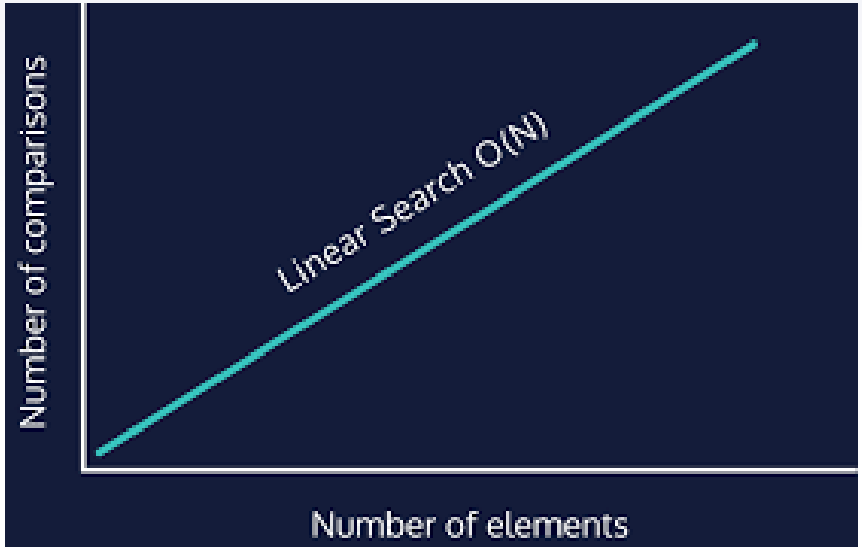
- 以下是全班20名男生的高度：**172, 168, 160, 166, 170,175, 162, 159, 168, 169, 171, 166, 171, 168, 170, 165, 165, 173, 160, 166**
- 试写一代码，找出有多少位男生的高度位170或以上。

- 解答（伪代码）：

```
heights = [172, 168, 160, 166, 170, 175, 162, 159, 168, 169, 171, 166, 171, 168, 170, 165, 165, 173, 160, 166]
count = 0
For i = 0 to 19 do
If heights[i] >= 170 then  count = count + 1
End For
print(" The number of boys whose height is 170 or above is：  ", count)
```

线性搜索的缺点

- 缺点：效率较低，时间复杂度（Time Complexity）为 O(N)，其中 N 是列表的长度。
- 例：某人心中有一数，介于 200和 1000 之间，请猜出此数。
- 不论从200 起，逐一逐一的顺序试，或是从1000倒算，逐一逐一逆序试，虽然可以找到答案，效率都不高。
- 是否有更有效率的搜寻方法？



二分搜索（Binary Search）

- 是一种在有序数组中，以一分为二、不断切割数组的搜索算法。
- 基本思想是，假设数组中的元素是按升序排列的，算法从数组的中间位置开始搜索；即使不能立即找到要找的元素，也可以把数组一分为二，缩小一半的范围，让接下来的搜索更快捷。
- 重复以上过程，直到找到目标元素或确定目标元素不存在为止。
- 优点：二分搜索比线性搜索更有效率、可以更快完成任务。

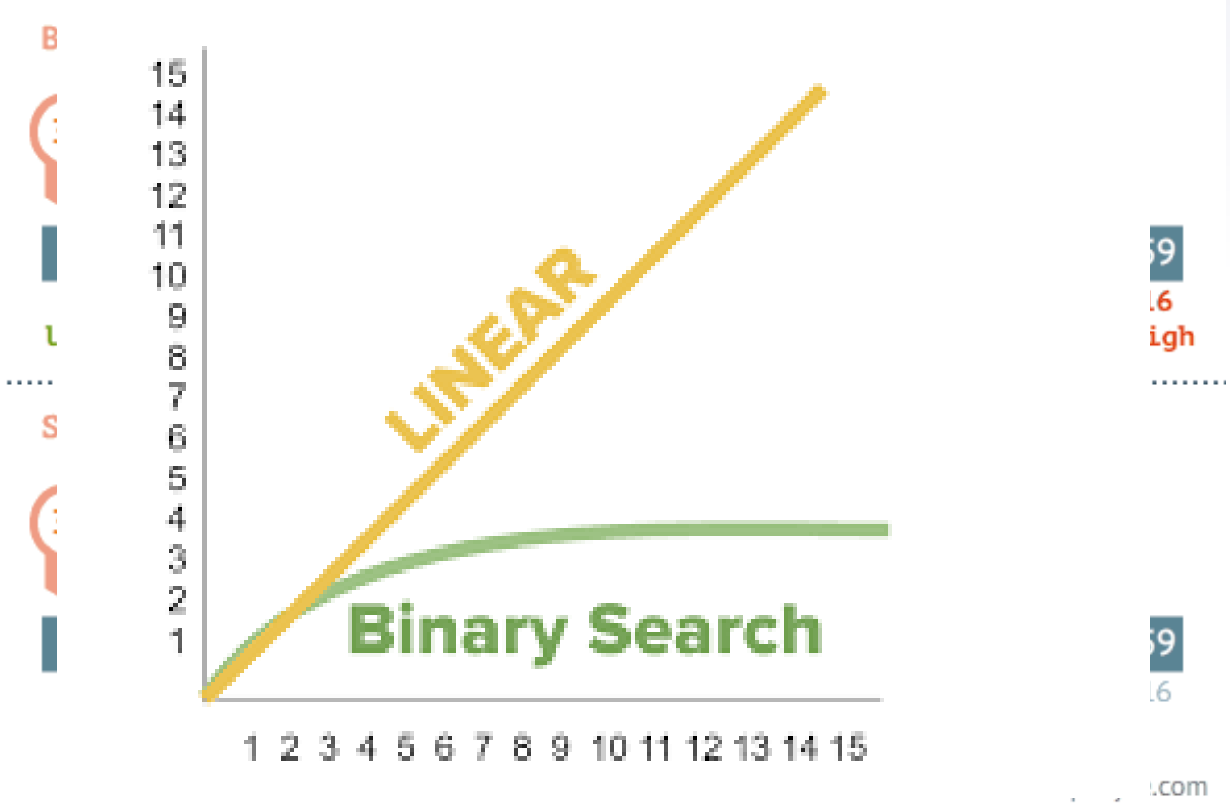
二分搜索 – 例一

- 回到之前的例子，某人心中有一数，介于 200和 1000 之间，请猜出此数。（假设此数为650 。）

1. 由于猜的人不知此数，于是可从 $\left\lfloor \frac{200+1000}{2} \right\rfloor$ = 600开始猜。
2. 若被告知此数大于600，范围缩小至[601, 1000]；接下来可猜 $\left\lfloor \frac{601+1000}{2} \right\rfloor$ = 800。
3. 若被告知此数小于800，范围缩小至[601, 799]；接下来可猜 $\left\lfloor \frac{601+799}{2} \right\rfloor$ = 700。
4. 若被告知此数小于700，范围缩小至[601, 699]；接下来可猜 $\left\lfloor \frac{601+699}{2} \right\rfloor$ = 650。
5. 猜到，结束。

- 二分搜索的时间复杂度（Time Complexity）为 O(log N)。以上述例子，最多只要猜 $\lceil \log_2(1000 - 200 + 1) \rceil = 10$ 次，就一定可以猜到。

二分搜索与线性搜索的比较



二分搜索 – 例二

- 又如之前的例子，若20名男生的高度已经排序好如下：
  159, 160, 160, 162, 165, 165, 166, 166, 166, 168, 168, 168, 169, 170, 170, 171, 171, 172, 173, 175
- 试写一代码，找出有多少位男生的高度位170或以上。
- 解题思路：找出第一个大于等于170数字的位置，如
  159, 160, 160, 162, 165, 165, 166, 166, 166, 168, 168, 168, 169, 170, 170, 171, 171, 172, 173, 175
- 在此数字之后所有数字都是大于等于170
- 我们可利用定义区间 [low, high+)，再不断缩小此区间直到 low 成为第一个大于等于170的数字。

二分搜索 – 例二（解）

heights = [159, 160, 160, 162, 165, 165, 166, 166, 166, 168, 168, 168, 169, 170, 170, 171, 171, 172, 173, 175]

low = 0 // low 是左边界，包括此数，初始值为 0

high+ = length(heights) // high+ 是右边界，但不包括该数，初始值为列表长度

While low < high+ **do**

mid = ⌊(low + high+) / 2⌋ // 计算中间索引值，必须取整数

If heights[mid] < 170 **Then** low = mid + 1 // 如果中间值小于170，则查找右半部分

Else high+ = mid // 如果中间值大于等于170，则查找左半部分

End While

count = length(heights) - low // 计算大于等于170的数字的个数

print count // 输出计数器的值，即大于等于170的数字的个数

二分搜索 – 例二（推演）

low = 0, high+ = 20, mid = 10

[159, 160, 160, 162, 165, 165, 166, 166, 166, 168, 168, 168, 169, 170, 170, 171, 171, 172, 173, 175 +]

low = 11, high+ = 20, mid = 15

[159, 160, 160, 162, 165, 165, 166, 166, 166, 168, 168, 168, 169, 170, 170, 171, 171, 172, 173, 175 +]

low = 11, high+ = 15, mid = 13

[159, 160, 160, 162, 165, 165, 166, 166, 166, 168, 168, 168, 169, 170, 170, 171, 171, 172, 173, 175]

low = 11, high+ = 13, mid = 12

[159, 160, 160, 162, 165, 165, 166, 166, 166, 168, 168, 168, 169, 170, 170, 171, 171, 172, 173, 175]

low = 13, high+ = 13, end of the while loop

[159, 160, 160, 162, 165, 165, 166, 166, 166, 168, 168, 168, 169, 170, 170, 171, 171, 172, 173, 175]

The answer is then 20 – 13 = 7

练习一（位置安插）

- 给定有一个有序数组（sorted array）：
  Num = [2, 4, 4, 9, 13, 21, 24, 24, 24, 29, 32, 34, 45, 45, 49, 50]
- 输入一数 x。把 x 安插进这数组中，并保留有序的特性；同时若 x 的值已经出现过，则 x 会安插在相同数字的最后一位。请问 x 在此数组中的位置为何？其他数字的位置有何影响？
- 伪代码：

  Num = [2, 4, 4, 9, 13, 21, 24, 24, 24, 29, 32, 34, 45, 45, 49, 50]

  Input x

  ……

解题思路（位置安插）

- 解题思路一：
  - x 应该被安插在第一个大于 x 的数字之前。
- 解题思路二（线性搜索）：
  - 若是用线性搜索，只要逐个逐个，从第 0 个至第 length(Num)－1 个数字查找，当碰到第一个大于 x 的数字，就可以找到相应安插 x 的位置。
- 解题思路三（二分搜索）：
  - 利用二分搜索，可以更快速的找到第一个大于 x 的数字。
- 解题思路四：
  - 找到 x 的相应位置，安插前，之后的数字先要顺移一位，然后才能安插 x。

位置安插 – 解（线性搜索）

Num = [2, 4, 4, 9, 13, 21, 24, 24, 24, 29, 32, 34, 45, 45, 49, 50]

Input x； N = length(Num) // 原来数组的长度

If x >= Num[N – 1] Then Num[N] = x // 若 x 大于等于数组中所有数字，则把 x 放在末尾

Else For i = 0 to N – 1 do

If Num[i] > x, Then For j = N to i +1 do // x 的位置已找到

Num[j] = Num[j-1] //先顺移后面的数字

End For

Num[i] = x // 再把 x 安插在 i 的位置上

break // x 的位置已找到，跳出 i 的 For loop

附加练习（位置安插的二分搜索解）

- 如果要求用二分搜索来安插 **x** 的位置，伪代码该如何写？

搜索算法的应用 – 求函数之根

- 给定一函数 $f(x)$。当 $x = r$ 时，$f(r) = 0$，则 r 称为$f(x)$的根（root）。
- 例：若 $f(x) = x^2 - 2x - 3$，则$f(x)$有二根，分别为 –1 及 3，因

  $f(-1) = （-1）^2 - 2(-1) - 3 = 0;$

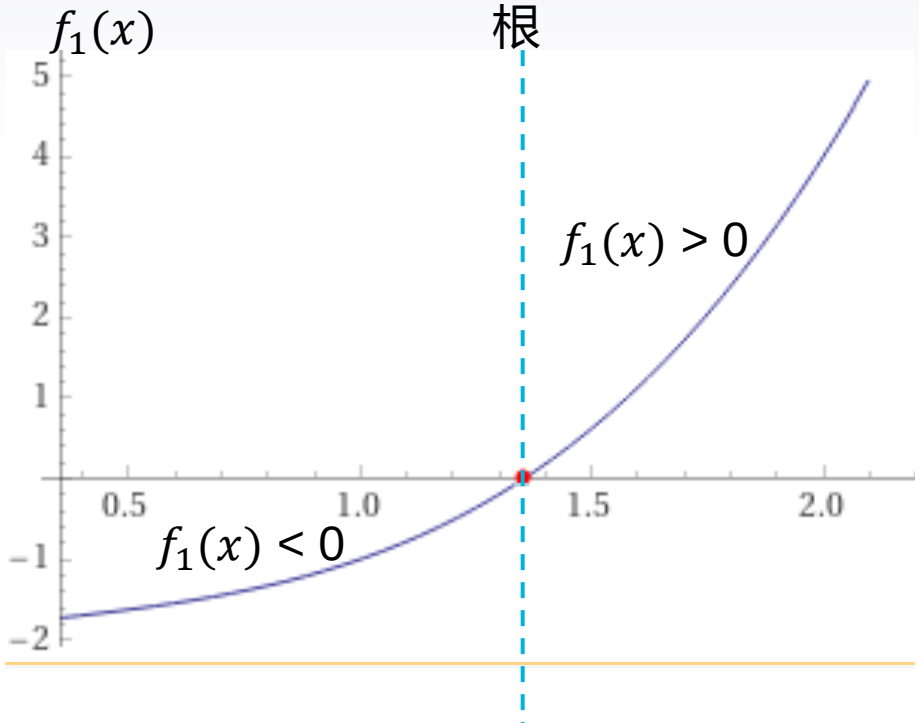
  $f(3) = 3^2 - 2 \times 3 - 3 = 0$
- 若要用人力进行计算，可通过因式分解、或由以下公式轻松求得上述一元二次方程式的根：

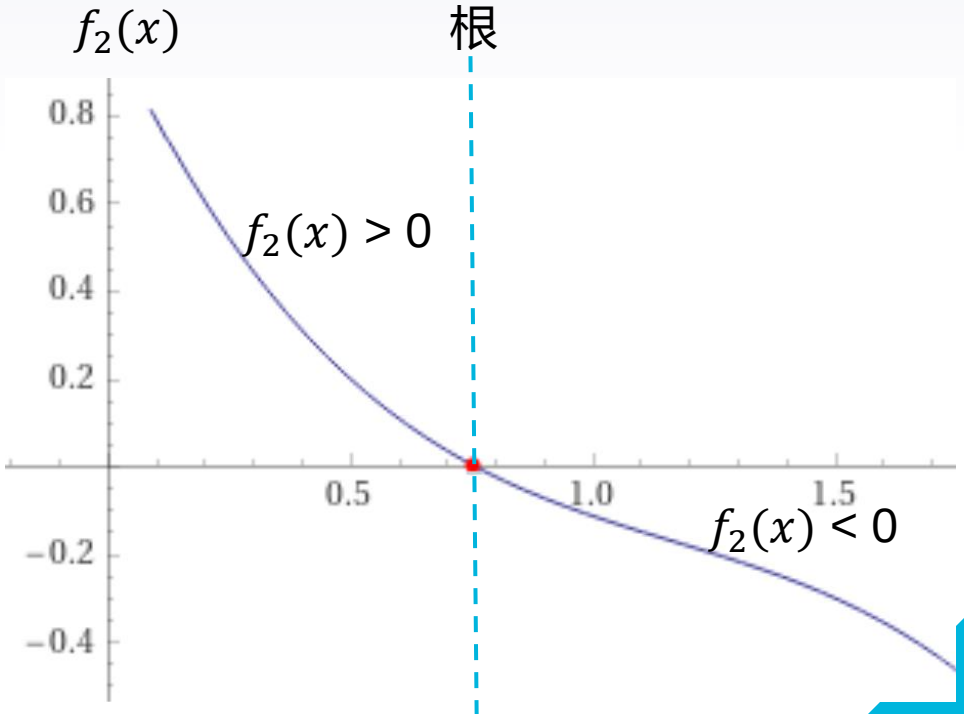
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- 但是，如何使用电脑程式来查找函数 $f(x)$ 的根？
- 比如：$f(x) = x^3 - x^2 + x - 4$，如何寻根？

简单根的特质

- 在一个简单根（**simple root**）的附近，函数 $f(x)$ 的值会发生 **由负数变为正数** 或者 **由正数变为负数** 的变化。





例子：查找函数之根

- 给定函数 $f(x) = x^3 - x^2 + x - 4$，且$f(x)$ 在 $x = -2$ 及 $x = 3$ 之间有一简单根。找出此根之值，并将结果近似至两位小数。
- 我们可以先检验函数在 $x = -2$ 及 $x = 3$ 的值

$$f(-2) = （-2）^3 - (-2)^2 + (-2) - 4 = -18 < 0$$

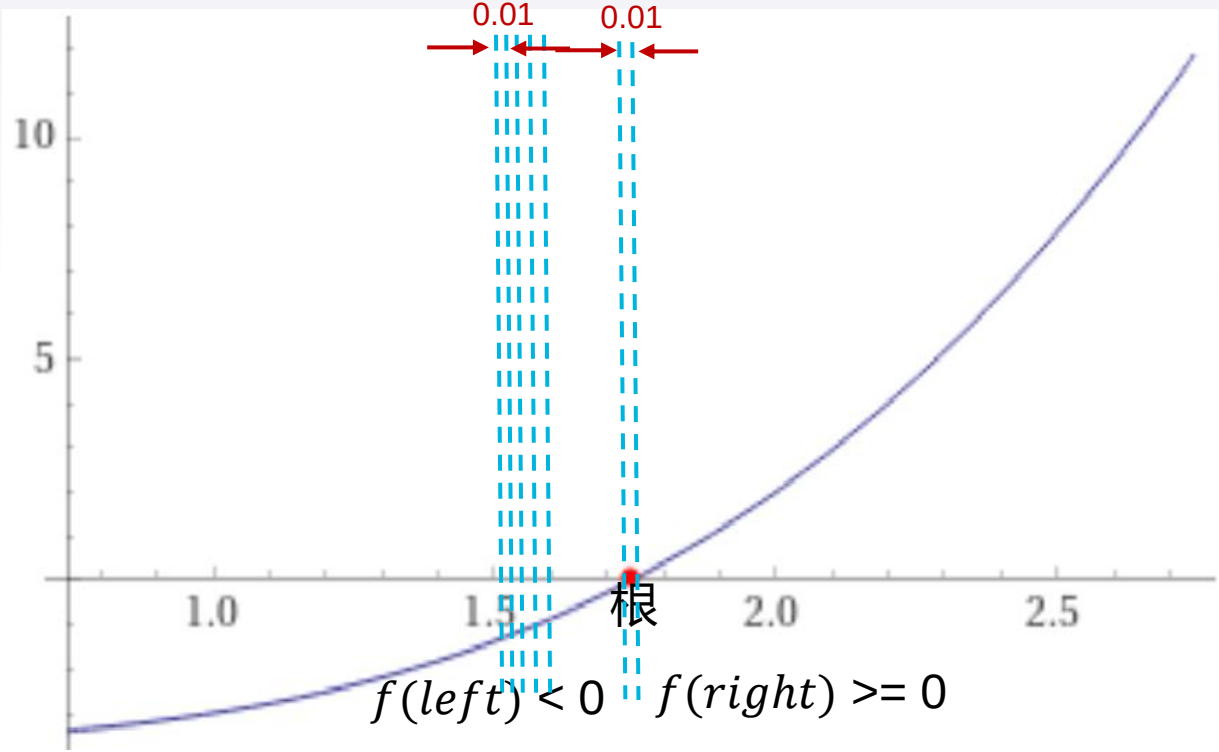
$$f(3) = （3）^3 - (3)^2 + (3) - 4 = 17 > 0$$

由于函数 $f(x)$ 在 $x \in [-2, 3]$ 由负数转为正数，确认此区间有一根。

- 接下来，我们看看如何利用
  - 线性搜索
  - 二分搜索

  来找到这个根。

线性搜索 $f(x) = x^3 - x^2 + x - 4$



线性查找函数之根 – 解题思路

如果要在函数 $f(x)$ 中查找位于a 和 b之间的根，可以按照以下步骤使用

1. 将a和b之间的区间划分为长度为0.01子区间。
2. 从左到右依次搜索每个子区间，计算函数$f(x)$在该区间左端点（***left***）和右端点（***right***）的值，即$f(left)$及$f(right)$。
3. 如果两个值的符号相同，则根不在此，可推进到下一个子区间。
4. 如果两个值的符号不同，则在该子区间内存在一个根。
5. 使用中间点来估计根的位置，即 $mid = \frac{left+right}{2}$ 。
6. 若$f(left)$和$f(mid)$符号不同，则根的位置偏左，则 x = ***left*** 是为解；否则，根的位置在中间或偏右， x = ***right*** 是为解。

线性查找函数之根 – 伪代码

$f(x) = x^3 - x^2 + x - 4$

left = -2 // 初始化左端点

While *left* < 3 **do** // 设定程式运行的右边界

right = *left* + 0.01; // 计算子区间右端点

If $f(left)*f(right)$ <= 0 **Then** *mid* = (*left* + *right*) / 2 ;

If $f(left)*f(mid)$ < 0 **Then** x = *left*

Else x = *right*

End If; **break**

End If

left = *right* // 未找到根，往下一个子区间推进

End While

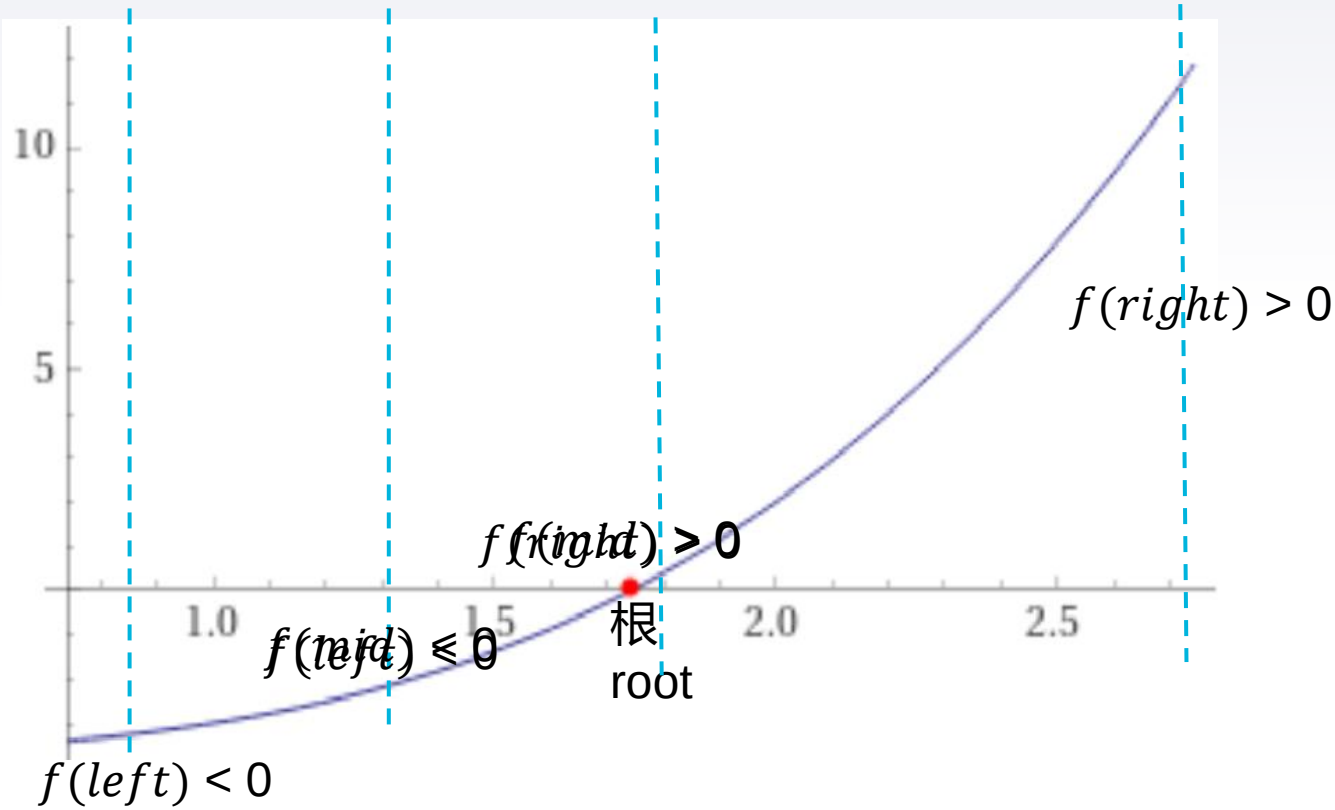
推演练习：$f(x) = x^3 - x^2 + x - 4$

为了节省时间，我们假设从 $x = -2$ 到 $x = 1.70$， $f(x)$的 值都是负数，没有转折，现在我们从 $x = 1.70$ 继续推演如下：

1. *left* = 1.70 ; *right* = 1.71
2. *left* = 1.71 ; *right* = 1.72
3. *left* = 1.72 ; *right* = 1.73
4. *left* = 1.73 ; *right* = 1.74
5. *left* = 1.74 ; *right* = 1.75
6. *mid* = 1.745
7. $f(1.74)^* f(1.745) < 0$
8. 所以这个根在 1.740 和 1.745 之间，取小数二位的近似值，则其解为 1.74

$f(1.70) = -0.277$
$f(1.71) = -0.214$
$f(1.72) = -0.150$
$f(1.73) = -0.085$
$f(1.74) = -0.020$
$f(1.75) = 0.047$
$f(1.745) = 0.014$

图解 $f(x) = x^3 - x^2 + x - 4$



查找函数之根 – 二分法

- 二分法（**Bisection Method**）是二分搜索的一个变例，用于寻找一个函数的实根（**real root**）。此方法基于连续函数的中间值定理，通过不断缩小包含实根的区间来逐步逼近实根。具体步骤如下：

1. 给定一个连续函数$f(x)$，在一个区间$[a, b]$内寻找$f(x) = 0$的根。
2. 选择一个中间点c，其中$c=(a+b)/2$，计算函数值$f(c)$。
3. 如果$f(c) = 0$，则c是实根，结束计算。
4. 如果$f(a)$和$f(c)$的符号不同，则实根必定在区间$[a, c]$内。将区间$[a, b]$缩小为区间$[a, c]$。
5. 如果$f(a)$和$f(c)$的符号相同，则实根必定在区间$[c, b]$内。将区间$[a, b]$缩小为区间$[c, b]$。
6. 重复步骤**2-5**，直到找到一个接近实根的解，或者找到一个满足预先设定的精度要求的解。

二分法查找函数之根 – 伪代码

$f(x) = x^3 - x^2 + x - 4$

left = -2 ; *right* = 3 // 初始化程式运行的左右边界

While round (*left*, 2) != round (*right*, 2) **do** // 设定精确度至小数 2 位

mid = (*left* + *right*) / 2; // 计算子区间的中点

If $f(mid)$ = 0, **Then** *left* = *mid*; **break**, **Endif**

If $f(left)*f(mid)$ < 0 **Then** *right* = *mid* // 缩小范围至左半部

Else *left* = *mid* // 否则，缩小范围至右半部

End If

End While

x = round (*left*, 2)

推演练习：$f(x) = x^3 - x^2 + x - 4$

1. *left* = -2 ; *right* = 3;
2. *left* = 0.5 ; *right* = 3;
3. *left* = 0.5 ; *right* = 1.75
4. *left* = 1.125 ; *right* = 1.75;
5. *left* = 1.438 ; *right* = 1.75;
6. *left* = 1.594 ; *right* = 1.75;
7. *left* = 1.672 ; *right* = 1.75
8. *left* = 1.711 ; *right* = 1.75;
9. *left* = 1.731 ; *right* = 1.75;
10. *left* = 1.741 ; *right* = 1.75;
11. *left* = 1.741 ; *right* = 1.746
12. round (*left*, 2) = round (*right*, 2) **Exit While** and x = round (*left*, 2) =

$f(3) = 17$
$f(-2) = -18$
$f(0.5) = -3.6$
$f(1.75) = 0.05$
$f(1.125) = -2.7$
$f(1.438) = -1.7$
$f(1.594) = -0.9$
$f(1.672) = -0.5$
$f(1.711) = -0.2$
$f(1.731) = -0.08$
$f(1.741) = -0.013$
$f(1.746) = 0.02$
$.744) = 0.007$

二分搜索的缺点

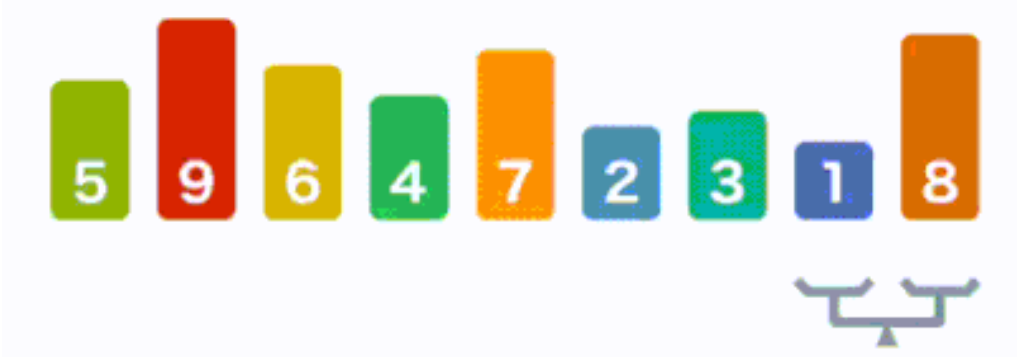
- **只适用于有序列表（ordered list）：**二分搜索只适用于有序列表，如果列表没有排序，需要先排序（sorting）才能使用。
- **数据量较小时不占优势：**在数据量较小的情况下，二分搜索并不一定比线性搜索更快。
- 排序的时间复杂度其实比线性搜索还要高！
  （问，那为什么还要排序？？）

排序（Sorting）

- 排序是电脑程式中常见的操作，可以用于各种场景，例如：
- 二分搜索，需要先对数组进行排序，以便能够多次地快速找到目标元素。
- 在一些竞赛或考试中，需要对参与者进行排名。例如，将学生成绩进行排序，以便能够排出从最高分到最低分的学生。
- 在数据分析（Data Analysis）中，我们经常需要对数据进行排序，以便更容易地发现其中的规律（patterns）和趋势（trends）。例如，
  - 我们可以对销售额（sales）进行排序，以便找到最畅销的产品或服务；
  - 或者对人口数量进行排序，以便找到人口最多或最少的城市或国家。

冒泡排序（Bubble Sort）

- 冒泡排序（**Bubble Sort**）是最基础的排序算法。
- 通过比较相邻的元素，冒泡排序算法不断比较相邻的两个元素，如有必要交换两数的位置，以达到总体排序的目的。
- 因为较小的元素会像气泡一样逐渐“浮”到序列的顶端，而较大的元素会逐渐“沉”到序列的底部，所以这种排序算法被称为“冒泡排序”。



冒泡排序的例子推演

- 这个例子，我们展示如何把大的数字“沉”到数组的底部，其原理和把小的数字“浮”到数组的顶部是一样的。初始数组：[5, 3, 8, 4, 2]
- 第一回合（**First iteration**）：
  [5, 3, 8, 4, 2] -> [3, 5, 8, 4, 2] -> [3, 5, 8, 4, 2] -> [3, 5, 4, 8, 2] -> [3, 5, 4, 2, 8]
- 第二回合（**Second iteration**）：
  [3, 5, 4, 2, 8] -> [3, 5, 4, 2, 8] -> [3, 4, 5, 2, 8] -> [3, 4, 2, 5, 8]
- 第三回合（**Third iteration**）：
  [3, 4, 2, 5, 8] -> [3, 4, 2, 5, 8] -> [3, 2, 4, 5, 8]
- 第四回合（**Fourth iteration**）：
  [3, 2, 4, 5, 8] -> [2, 3, 4, 5, 8]
- 已排好序的数组： [2, 3, 4, 5, 8]

冒泡排序的伪代码

```
Input A: the list of unsorted items
  N = length(A)   // 设 N 为此数组的长度
  For i = 1 to N - 1 do  // 总共要做 N – 1 个回合
    For j = 0 to N – i - 1 do // 第 i 个回合，需要做 N – i – 1 次的对比
      If A[j] > A[j+1] then
        swap(A[j], A[j+1])   // 若第 j 个数字大于第 j+1 个数字，则交换。
      End If
    End For
  End For
```

练习二（工作排程，Job Scheduling）

- 某公司有一位职员，每一次可执行系统里的一项工作。每一项工作的延迟（**delay**）时间是从工作输入系统开始计算、一直到该职员完成这项工作为止。
- 假设现在有10项工作同时被输入了系统，它们所需的执行时间分别为 **Jobs = [15, 5, 4, 9, 13, 3, 8, 10, 9, 5]**
- 如何安排这些工作，这名职员才可以最小化所有工作的平均延迟时间？试写一程式算出最优的排程以及最小的平均延迟时间。
- 伪代码：**Jobs = [15, 5, 4, 9, 13, 3, 8, 10, 9, 5]**

  .....

解题思路（工作排程）

- 解题思路一 （Jobs = [15, 5, 4, 9, 13, 3, 8, 10, 9, 5]）：
  - 若该职员先做时间长的工作，则此工作的执行时间将成为所有之后工作的延迟。
  - 例如，若是职员先做15分钟的工作，则其他所有 9 项工作都会遭受到至少15分钟的延迟时间。
  - 所以，显然在这题目上，先做时间短的工作会有好处！
- 解题思路二：
  - 先把 Jobs 里的工作排序，然后从时间最短的做到时间最长的。
  - 每项任务的延迟，是它之前任务执行的时间（等待时间）加上它本身的执行时间。即：$\mathbf{Delay}[i] = \sum_{j=0}^{i} \mathbf{Jobs}[j]$

工作排程 – 伪代码

```
Jobs = [15, 5, 4, 9, 13, 3, 8, 10, 9, 5]
Bubble_Sort (Jobs)   // 这一段可以改成任何的 Sorting Algorithm
N = length(Jobs);
Delay[0] = Jobs[0]          //初始化第0项工作的延迟时间
Total_Delay = Delay[0]      //初始化总延迟时间
For i = 1 to N-1 do
   Delay[i] = Delay[i-1]+ Jobs[i];
   Total_Delay = Total_Delay + Delay [i]
End For
Average_Delay = Total_Delay / N
```

工作排程 – 推演

Jobs = [15, 5, 4, 9, 13, 3, 8, 10, 9, 5]

- 经过排序后： **Jobs = [3, 4, 5, 5, 8, 9, 9, 10, 13, 15]**
- **Delay = [3, 7, 12, 17, 25, 34, 43, 53, 66, 81]**
- **Total_Delay = 3+7+12+25+34+43+53+66+81 = 341**
- **Average_Delay = Total_Delay / N = 341/10 = 34.1**