问题，实例和算法

- 考虑计算问题时，需要区分三个概念
  - 一个问题 W：是一个需要解决的需求 (需用计算求解)；例如，求实数的平方根，找出大于整数 m 的第一个素数等等
  - 问题 W 的一个实例 w：是问题 W 的一个具体例子；例如：求 2.0 的平方根，找出大于 10^{10} 的第一个素数等等
  - 解决问题 W 的一个算法 A：是对“能求解问题 W 的所有实例的一个计算过程”的严格描述；对 W 的实例 w，实施算法 A 描述的计算过程就得到所需结果；例如，平方根算法能求出 2.0 的平方根，求素数算法能得到大于 10^{10} 的第一个素数
- 对于解决某个具体计算问题 W 的具体算法 A
  - 需关注 A 在计算中的存储开销、时间开销
  - 算法 A 应该能处理问题 W 的任何一个实例
    - 具体实例的规模不同 ➔ 计算时的实际开销与实例的规模有关

算法的复杂性 (1)

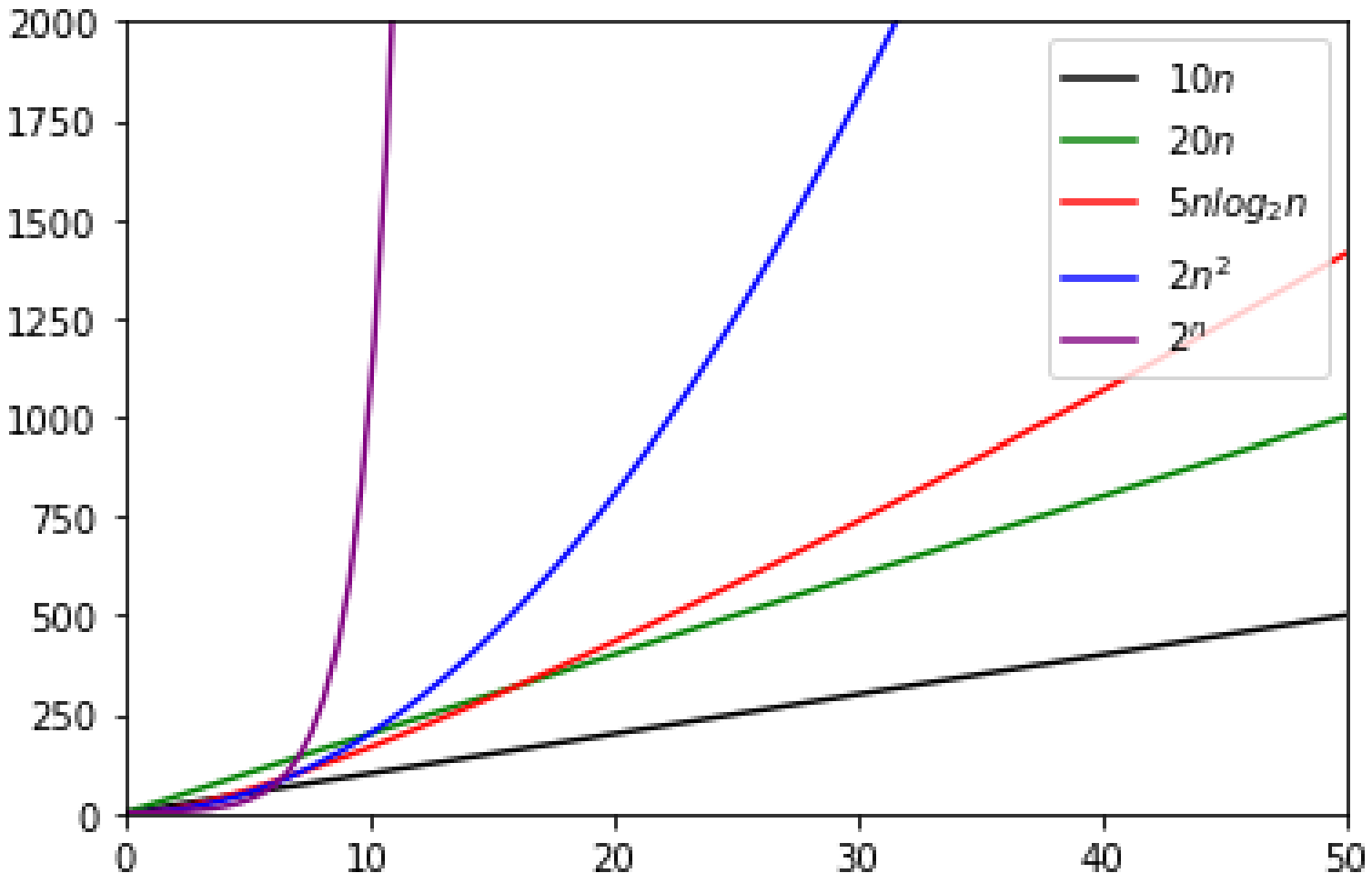
- 不同实际机器的操作执行速度、存储单元大小都可能不同 ……
  - 应该对算法性质做抽象的理论分析和研究，排除具体机器的非普遍性特征，从而反映出问题的共性和本质
- 针对现实中的计算问题，为了抽象地研究算法性质，需假定
  - 计算中的数据存储有一种基本单元，每个单元只能保存固定的一点有限数据 (一个单位的空间)
  - 计算中的一次基本操作要消耗一个单位的时间
- 为反映问题实例的具体规模对计算代价的影响，一般把一个算法的计算 (时间和空间) 代价定义为关于实例规模的函数
  - 可能有一些问题，其时间 (或空间) 开销函数随规模扩大增长得很快，另一些问题的开销函数增长较慢
  - 用函数关系反映计算的性质，实际上是描述一种增长趋势。这两个函数关系称为算法的时间代价和空间代价

算法的复杂性 (2)

- 所采用的方法类似于数学分析中有关无穷大的讨论
  - 忽略具体算法的常量特征 (和低阶项)，只考虑其量级 (例如，认为 $2\times n^2$ 和 $100\times n^2$ 属于同一量级)
- 算法的复杂性度量：在处理规模为 n 的实例时
  - 所需存储空间的量级，称为算法的空间复杂性
  - 所需要的计算时间的量级，称为算法的时间复杂性
- 例如，如果一个算法的时间代价随着实例规模的增长而线性增长 (成比例增长) ➔ 这个算法具有线性的时间复杂性

  例如，求一个表中的元素之和或平均值
- 常见复杂性：常量 — O(1)；对数 — O(log n)；线性 — O(n)；O(n log n)；平方 — $O(n^2)$；立方 — $O(n^3)$；指数 — $O(2^n)$
  - O 表示法：描述算法复杂性的上界

不同复杂性的增长


2000
1750
1500
1250
1000
750
500
250
0
0
10
20
30
40
50
10n
20n
5nlog2n
2n2
2n

算法复杂性的考量

- 用同一个算法解决问题，同样规模实例的计算开销可能不同
  - 求一组数据的平均值，同样大小的组开销差不多
  - 在一组数据里找第一个小于 0 的项，情况不同
- 几种可能有用的考虑 (以时间为例)，对规模为 n 的数据：
  - ① 考虑完成算法 A 最少需要多少时间
  - ② 考虑完成算法 A 最多需要多少时间
  - ③ 考虑完成算法 A 平均需要多少时间

➢ ① 价值较小，没提供太多有用信息 (最乐观的估计用处不大)

➢ ② 是一种保证，某时间内工作一定完成 (最坏情况时间复杂性)

➢ ③ 是对算法 A 的全面评价，但没有保证；且依赖于实例中的实际分布，较难确定 (平均时间复杂性)

Python 表操作的时间复杂度

check https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt https://wiki.python.org/moin/TimeComplexity for more

Lists:

Operation	Example	Complexity Class	Notes
Index	l[i]	O(1)	
Store	l[i] = 0	O(1)	
Length	len(l)	O(1)	
Append	l.append(5)	O(1)	mostly: ICS-46 covers details
Pop	l.pop()	O(1)	same as l.pop(-1), popping at end
Clear	l.clear()	O(1)	similar to l = []
Slice	l[a:b]	O(b-a)	l[1:5]:O(1)/l[:]:O(len(l)-0)=O(N)
Extend	l.extend(...)	O(len(...))	depends only on len of extension
Construction	list(...)	O(len(...))	depends on length of ... iterable
check ==, !=	l1 == l2	O(N)	
Insert	l[a:b] = ...	O(N)	
Delete	del l[i]	O(N)	depends on i; O(N) in worst case
Containment	x in/not in l	O(N)	linearly searches list
Copy	l.copy()	O(N)	Same as l[:] which is O(N)
Remove	l.remove(...)	O(N)	
Pop	l.pop(i)	O(N)	O(N-i): l.pop(0):O(N) (see above)
Extreme value	min(l)/max(l)	O(N)	linearly searches list for value
Reverse	l.reverse()	O(N)	
Iteration	for v in l:	O(N)	Worst: no return/break in loop
Sort	l.sort()	O(N Log N)	key/reverse mostly doesn't change
Multiply	k*l	O(k N)	5*l is O(N): len(l)*l is O(N**2)

Tuples support all operations that do not mutate the data structure (and they have the same complexity classes).

序列编程实例 1：再看筛法求素数

- 新的需求：基于筛法求前 n 个素数
  - 按照前面的实现，需要首先构建能包含前 n 个素数的整数表，再执行筛法
    - 问题：多大的表才足够大？素数的分布？
  - 新思路：用一个 (动态) 表记录已求得的素数，并利用该表不断地找到更大的素数，直到得到 n 个素数为止
    - 把已有的素数作为筛子，只有通过所有已有素数筛选的自然数/奇数才是素数 (筛法的思想)

序列编程实例 1：再看筛法求素数

```
def primes_n(n):
    def is_prime(cand, plist):
        '''谓词函数，利用表 plist 里的已知素数，判断 cand 是否是素数'''
        for p in plist:
            if cand % p == 0: return False
            if p * p > cand:  return True

    plist = [2]                          # 初始素数表
    num = 1                              # 已有素数的个数
    cand = 3                             # 候选数
    while num < n:
        if is_prime(cand, plist):        # cand 是找到的新素数
            plist.append(cand)           # 加入新素数
            num += 1                     # 更新素数个数
        cand += 2                        # 下一个奇数是新的候选数

    return plist
```

序列编程实例 2：Josephus 问题

- 问题：假设 n 个人围成一圈，依次编号为 1~n。现要求从第 k 个人开始报数，报到第 m 个数的人从圈里退出。然后，从下一个人继续 (重新) 报数，并采用同样规则让人出列，一直到所有人都退出。要求按顺序输出各出列人的编号
- 实现一：(模拟报数过程 + 定长数组)
  - 基本思路：用一个自然数表 people 来记录 n 个人，且将 people 看作是元素个数固定的数据组
    - 在操作中，只修改表中元素的值 (出列即置 0)，不改变表的结构 (类似于，n 把椅子摆了一圈，人退出后椅子不动)
  - 算法过程：对当前 people，反复数 m 个元素 (期间需跳过值为 0 的元素)，期间遇到表的末端则转回表头继续
- 实现二：利用 Python 表对象的动态特性模拟报数过程，将退出的人 (的编号) 直接从表中删除 (演示)

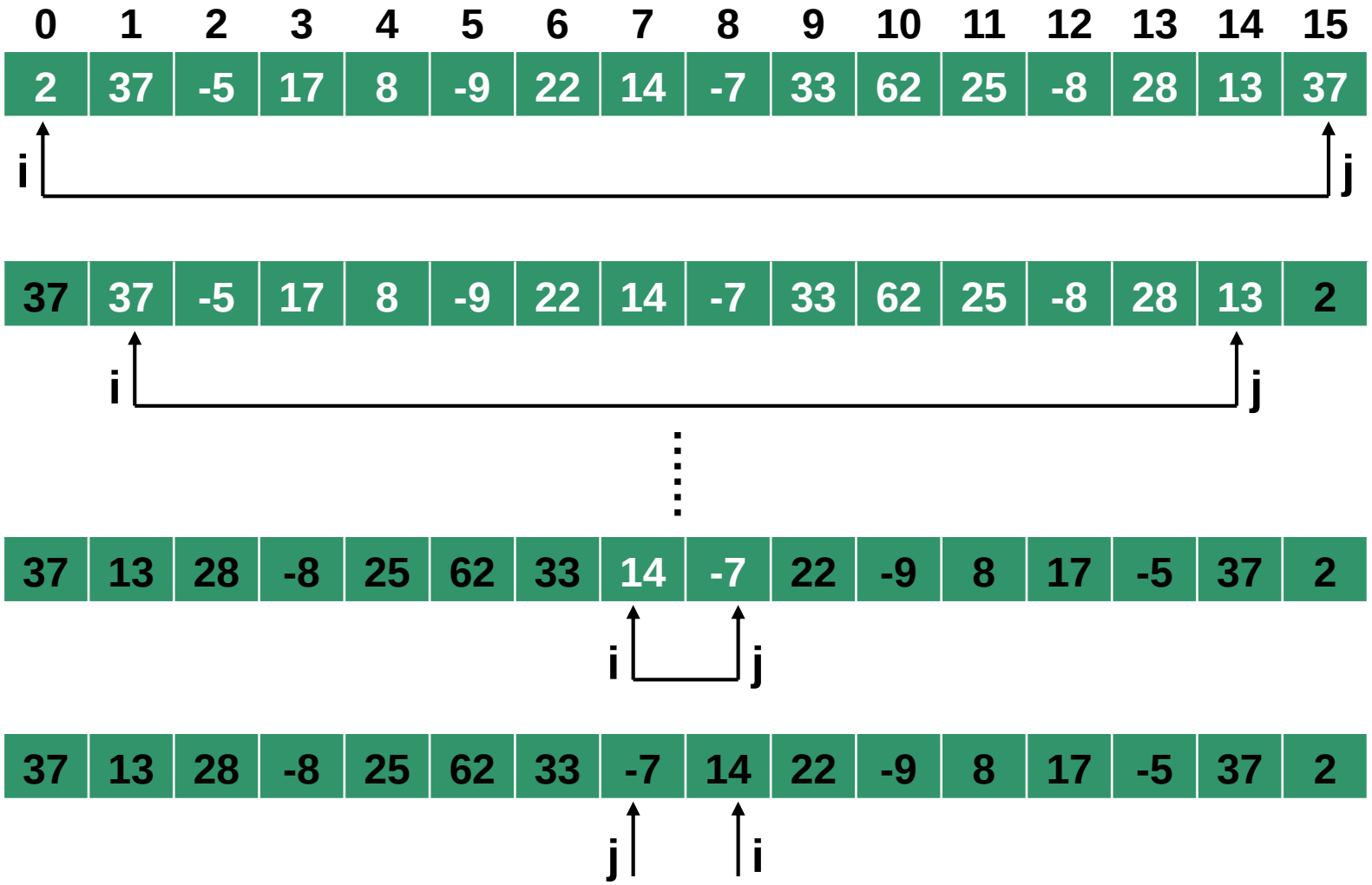
序列编程实例 3：反转表中元素的函数 (1)

- 要求：定义功能函数 rev，其参数是一个表，其功能与序列的 reverse 操作类似

```
def rev(lst): …          # 反转实参表里的元素
```

- 算法思路：基于循环 (for or while)，逐对交换表中前后对应位置的元素
  - 用一对下标变量 (正、负下标) 指向表中需要反转的一对元素
  - 交换当前下标指向的一对元素的值 (via 平行赋值)
  - 修改两个下标变量的值，使之相向而行，指向下一对元素并继续
  - 直到两个变量重合 (即只剩一个元素未处理)，或者交错 (即处理完所有元素)
- 自行完成 rev 函数定义 (递归、非递归均可)

序列编程实例 3：反转表中元素的函数 (2)



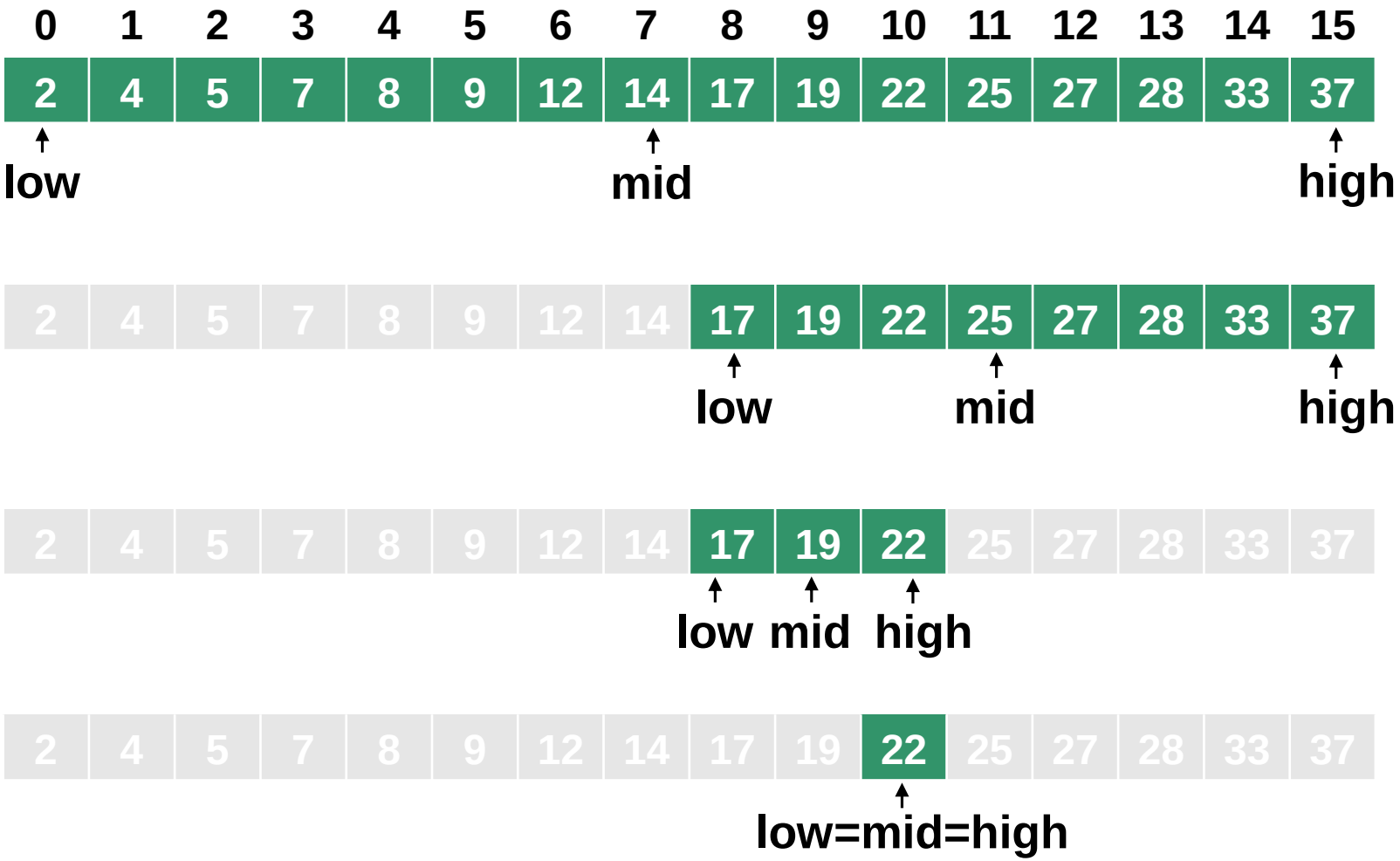
（类似的思路，也可以用于实现数据组的划分 partition）

序列编程实例 4：有序数据组的二分查找 (1)

- 查找：在数据组中查找满足某种条件的元素 (目标值)
- 无序数据组：顺序查找是基本的查找方法
  - 用循环顺序检查每一个元素，直至找到目标值或者检查完数据组 (线性时间复杂度)
- 有序数据组：存在更高效的算法，如二分查找
  - 属于分治算法 (divide and conquer)：分而治之
  - 设数据组中元素递增排列，现要查找某个元素 x 是否存在
    - 首先，比较 x 和数据组中的中间元素 m
      - 如果 x = m，则查找成功并终止
      - 如果 x < m，则在 m 之前的元素继续查找
      - 如果 x > m，则在 m 之后的元素继续查找
    - 直至查找成功，或所查找数据组为空 (查找不成功)

序列编程实例 4：有序数据组的二分查找 (2)

- 二分查找示例：有序数组中查找目标值 22



序列编程实例 5：朴素贪心法介绍

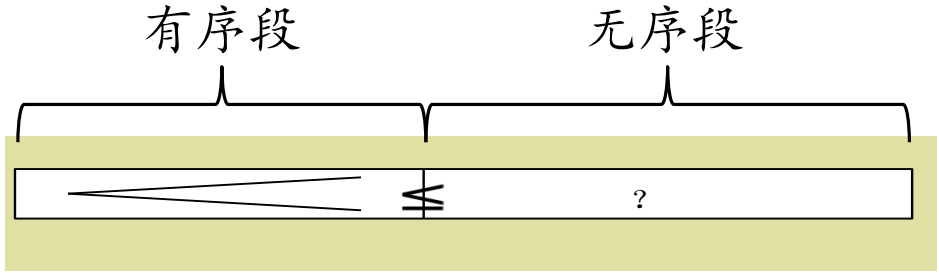
- 贪心策略的基本思想：
  - 将整个问题的解决切分成不同的决策阶段 (局部问题)
  - 在每个阶段，根据当前的局部状态做最优决策 (当前最好的选择，即贪心选择)
  - 通过一系列的贪心选择，逐步得到完整的解
- 贪心策略符合人们的日常思维方式，简单直观，通用性强，应用非常广泛 (但没有固定的算法框架)
- 贪心策略的使用要素：
  - 要求问题满足贪心选择性质：所求问题的整体最优解可通过一系列局部最优的选择达到
    - 理论证明通常比较复杂；实际应用时，一般只要求每次局部的决策可以依赖之前作出的决策，但不依赖于之后要作的决策
  - 要求问题满足最优子结构性质 (此处略，算法类课程介绍)

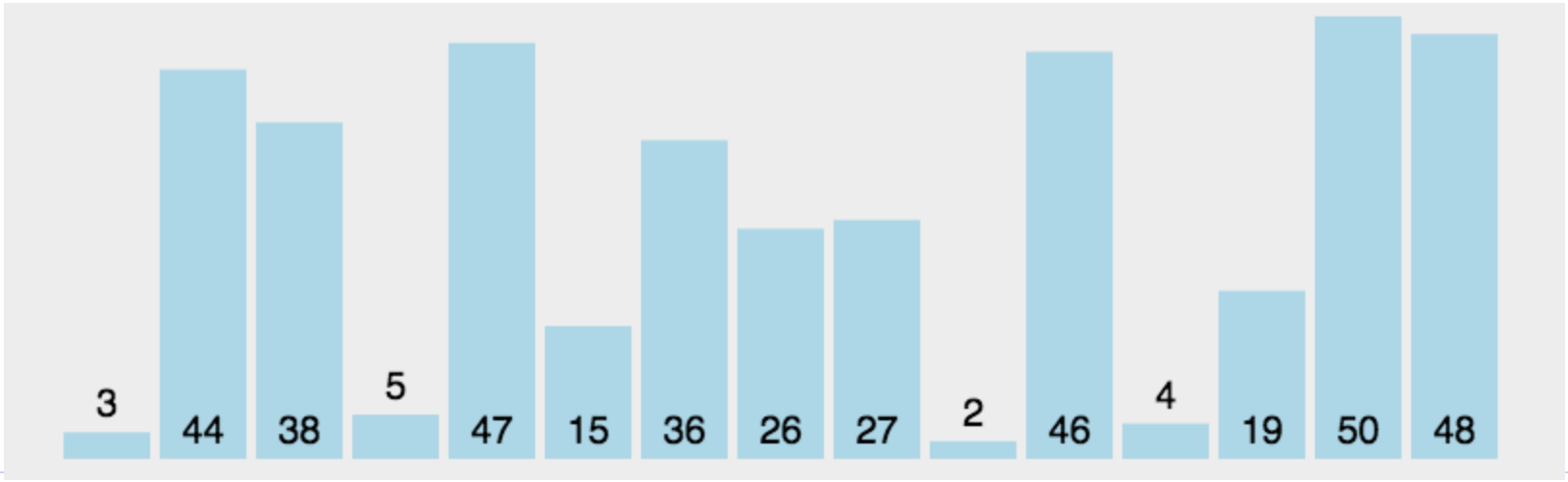
序列编程实例 5：朴素贪心法实例 - 1

- 付款问题：设若干种货币面额 $c1 > c2 > c3 > \cdots$，要用这些货币支付 k 元，问所需的最少货币数量
  - 贪心策略：每一次尽可能使用大面额的货币 (实现略)
- 背包问题：给定容量为 C 的背包，以及 n 种物品，其中物品 i 的重量是 w_i，价值为 v_i，且每种物品可以任意的拆分
  - 需求：应该如何选择装入背包的物品，使得背包所装入物品的总价值最大
  - 贪心策略：每次优先放单位重量价值 $\frac{v_i}{w_i}$ 最大的物品 (实现略)
  - ***c.f.* 0-1 背包问题**：不能拆分物品，每种物品只有装入或不装入两种选择
    - 不符合贪心选择性质，不能用贪心法解决
- 贪心策略中，一般会按照某种优先级排序处理，经常结合排序算法、优先队列等使用

序列编程实例 5：朴素贪心法实例-2

- **简单选择排序**：严格按照递增方式选出元素，之后再简单地顺序排列
  - 贪心策略：每次选最小元素
  - 排序工作在数组内部完成 (原位排序，尽可能少用辅助空间)

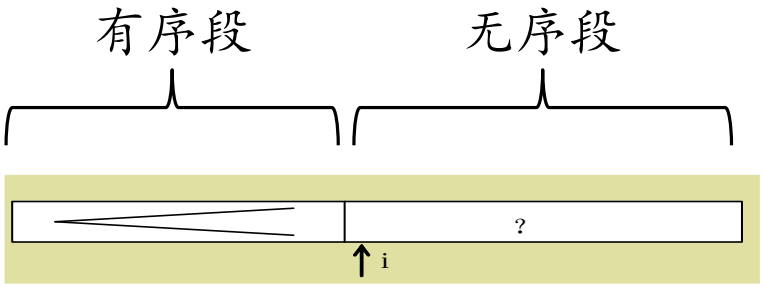


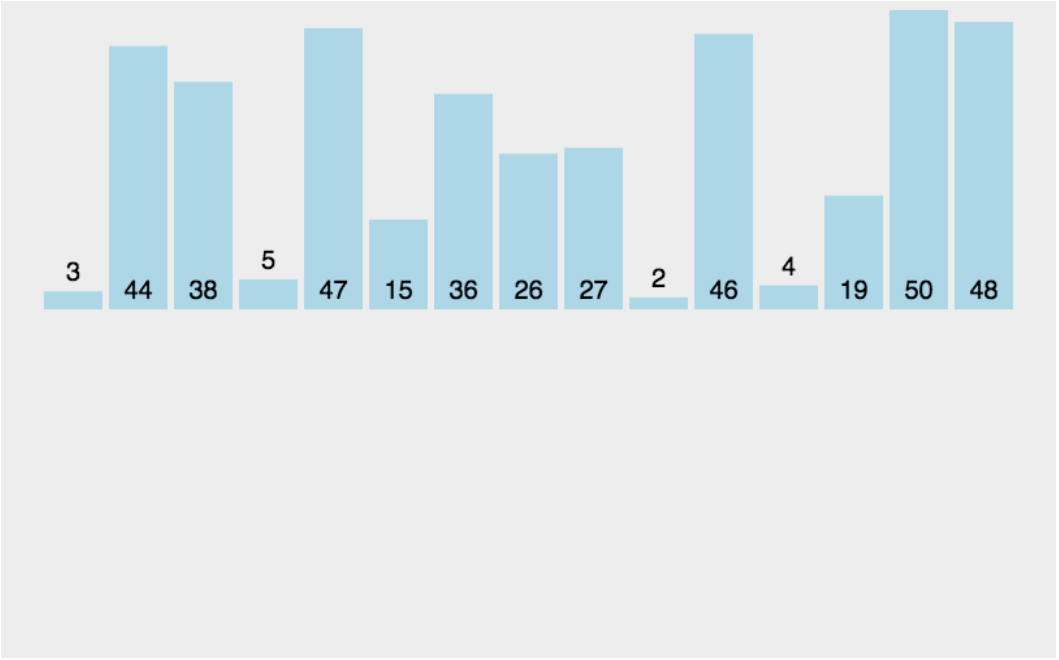


动图来自网络

序列编程实例 6：数据组的排序 (1)

- 基于比较的简单排序算法一：直接插入排序
  - In-place 原位排序
  - 将已排好序的数据放在数组的一侧 (如小下标的一侧)，另一侧是未排序的数据
  - 每次选择无序区间的首个元素，在有序区间内找到合适的位置插入

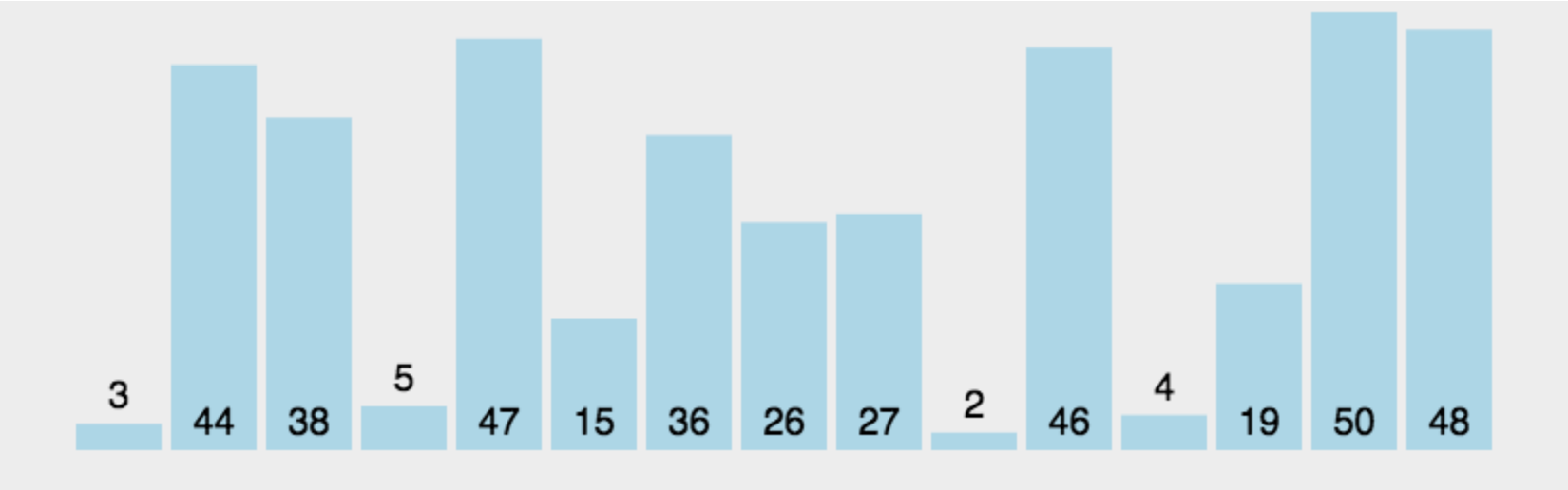




动图来自网络

序列编程实例 6：数据组的排序 (2)

- 基于比较的简单排序算法二：冒泡排序
  - 基本思想：多次走访数据组，每一次均从前往后、逐对比较相邻两个元素并排序 (从而使得小元素经过多次交换，慢慢“浮”到数列的前端)



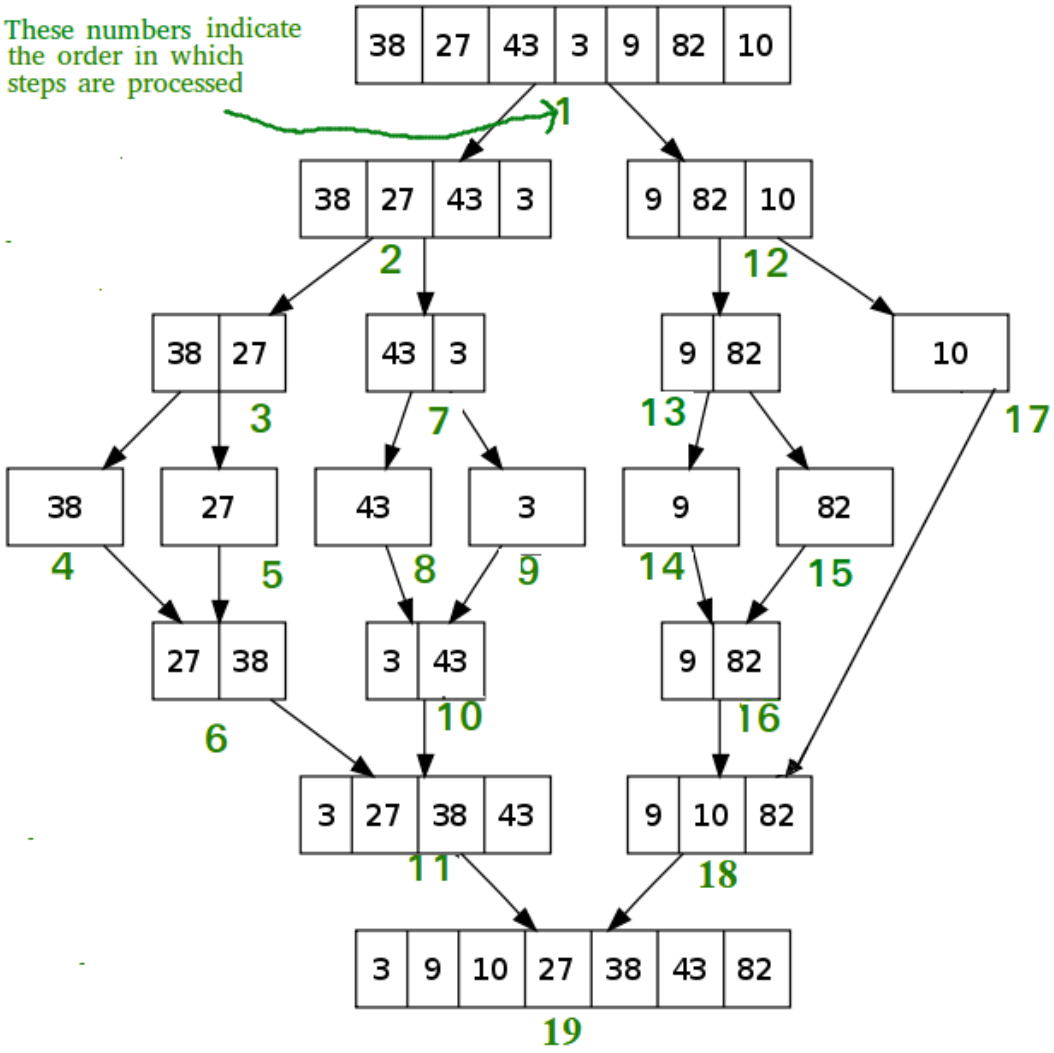
动图来自网络

序列编程实例 6：数据组的排序 (3)

- 归并排序：分治策略的典型应用
  - 归并是一种典型的序列操作，用以把两个或多个有序序列合并成一个有序序列
  - (二路) 归并排序的基本过程：
    - 分解：若序列 s 中只有 0 个或 1 个数据，即已完成排序，直接返回 s；若 s 包含 n 个数据 (n≥2)，则把 s 分裂成两个子序列 s1 和 s2，其中 s1 包含 s 前一半的数据， s2 包含 s 后一半的数据 (显然属于非原地排序)
    - 解决子问题：递归地对 s1 和 s2 进行归并排序
    - 合并：把两个排序好的子序列 s1 和 s2 归并成有序序列
  - 实现时，既可以采用自顶向下的递归实现，也可以采用自底向上的非递归实现

序列编程实例 6：数据组的排序 (3)

- 图示归并排序的执行过程



表处理的典型操作：表的遍历

- 表遍历：
  - 顺序扫描和处理表里的全部/部分元素
  - 对表中每个元素执行同样的操作
- 遍历操作的类型：
  - 构造一个新表，其各元素是原表元素的操作结果，实现一种表变换，从原表做出一个结构相同的新表
  - 通过遍历表累积元素包含的信息，得到累积结果
  - 直接修改被操作的表，遍历的效果通过表的修改体现
  - 其他操作，例如选择一些元素 (或做变换后) 做成新表等

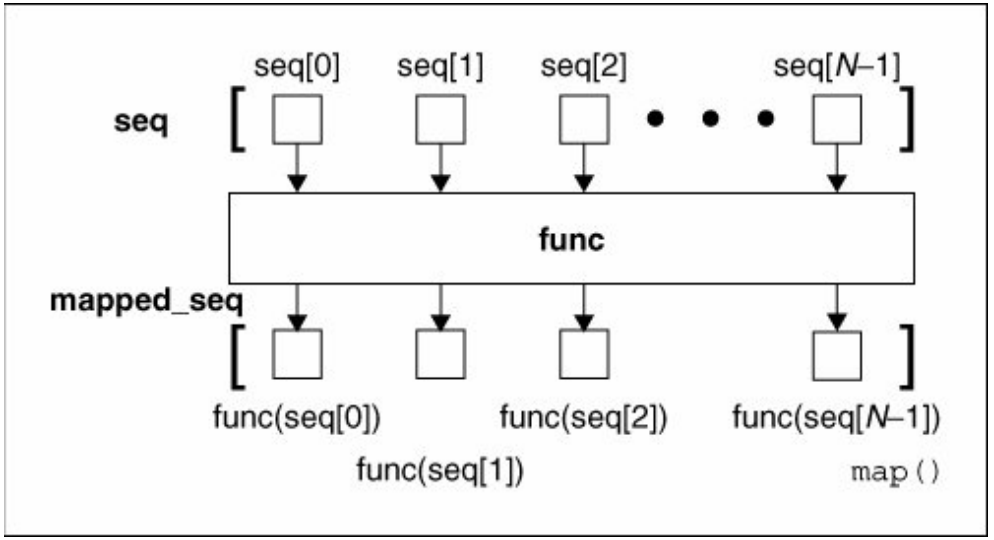
```
new_list = [ ]
for x in lst :
    new_list.append(f(x))
```

```
s = initValue
for x in lst :
    s = g(s, x)
```

```
for i in range(len(lst)):
    lst[i] = h(lst[i])
```

处理表的高阶函数 (1)

1. 基于一个表构造同样结构的新表




```
def map(func, list1):
    new_list = []
    for x in list1:
        new_list.append(func(x))
    return new_list
```


*图片来源：《Core Python Programming (2nd ed.)》

处理表的高阶函数 (1)

1. 基于一个表构造同样结构的新表

- 内置函数 **map(func, iterable, ...)** 提供类似功能，但得到一个迭代器，其中的元素是对参数 **iterable** 各个元素顺序调用 **func** 的结果

```
>>> lst = [1, 4, 5, 2]
>>> list(map(lambda x : x**2, lst))
[1, 16, 25, 4]
```

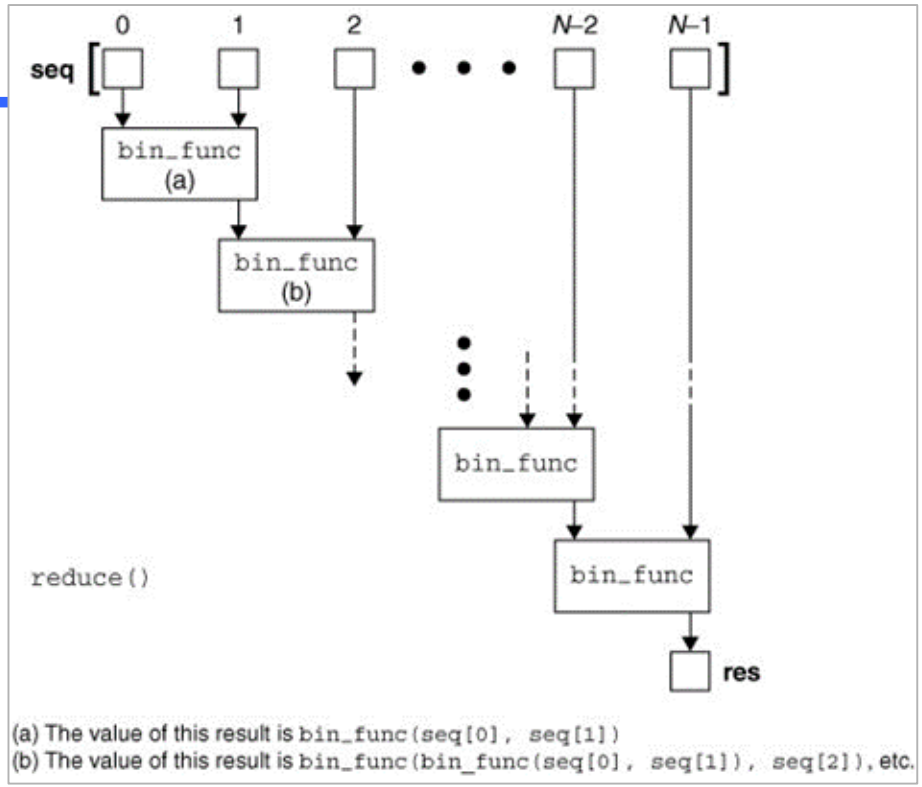
  - 当 **func** 的实参是多元函数时，需要用相应数量的 **iterable** 作为函数参数，执行时一旦某个迭代对象的值用完即结束

```
>>> list(map(lambda x, y: x + str(y), ["a","b"], [1, 2, 3]))
['a1', 'b2']
```

处理表的高阶函数 (2)

2. 累积表中元素的信息

   - 归约函数：接受一个可迭代对象作为参数，返回单个结果




```
## 函数 bin_func 是二元操作函数
def reduce(bin_func, list1, init):
    acc = init
    for x in list1:
        acc = bin_func(acc, x)
    return acc
```


*图片来源：《Core Python Programming (2nd ed.)》

处理表的高阶函数 (2)

2. 累积 (归约) 表中元素的信息

- functools 模块中 reduce 函数提供类似归约功能

functools.**reduce**(*function*, *iterable*[, *initializer*])

Apply *function* of two arguments cumulatively to the items of *sequence*, from left to right, so as to reduce the sequence to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `((((1+2)+3)+4)+5)`. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *sequence*. If the optional *initializer* is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If *initializer* is not given and *sequence* contains only one item, the first item is returned.

```
>>> from functools import reduce
>>> def fact(n): # 基于 reduce 函数实现阶乘的计算
        return reduce(lambda a, b: a*b, range(1, n+1))

>>> fact(6)
720
```

处理表的高阶函数 (3)

3. 根据条件选择表里的一些元素

```
## 函数 pred 是谓词函数
def filter(pred, list1):
    res = []
    for x in list1:
        if pred(x):
            res.append(x)
    return res
```

- 内置函数 **filter(func, iterable)**

  提供类似功能，但结果为一个迭代器，其中参数 **func** 的实参应是谓词

```
>>> filter(lambda x : x >= 0, lst)
>>> list(filter(lambda x : x != 0, lst))
```

处理表的高阶函数 (4)

- 应用实例：计算前 n 个 Fibonacci 数中所有被 3 整除的数之和
  - 思路：
    - 基于 map 函数，构造包含前 n 个 Fib 数的序列/迭代器
    - 基于 filter 函数，选择出其中能被 3 整除的元素
    - 基于 reduce 函数，累积剩余元素之和

处理表的高阶函数 (4)

- 应用实例：计算前 n 个 Fibonacci 数中所有被 3 整除的数之和

```
def fib(n):
    if n < 1:
        return 0
    f1, f2 = 0, 1 # F_0, F_1
    k = 0
    while k < n:
        f1, f2 = f2, f2 + f1
        k += 1
    return f1

from functools import reduce
def fib_sum(n):
    return reduce(lambda x, y: x + y,                    # 累积的方式
                  filter(lambda x: x % 3 == 0,           # 过滤的方式
                         map(fib, list(range(n)))),      # 生成的方式
                  0)                                     # 累积的初值
```

- 计算过程中并没有实际构造出任何表
- 执行时数据的生成、传递、检查和求和都是单项进行（惰性求值 lazy evaluation）

处理表的高阶函数 (5)

- **map, filter, reduce** 等处理表的高阶函数实现了对表操作的分解
  - 整合 "对表的遍历" 以及 "对每个元素的具体操作"
- **Map-Reduce** 是函数式编程的重要概念
  - 函数式语言通常都会提供类似功能的高阶函数 (可能名称不同)
  - **Google** 的网络信息处理系统采用的核心技术
- ***Note***: 列表推导式 (或生成器表达式) 可部分替代 **map** 和 **filter** 函数，且更易阅读；内置函数 **sum** 可部分替代 **reduce** 函数

```
>>> list(map(fact, range(1, 7)))
[1, 2, 6, 24, 120, 720]
>>> [fact(n) for n in range(1, 7)]
[1, 2, 6, 24, 120, 720]
>>> list(map(fact, filter(lambda n: n%2, range(1, 7))))
[1, 6, 120]
>>> [fact(n) for n in range(1, 7) if n%2]
[1, 6, 120]
```

内置序列函数

Note：all, any, max, min, sum 都属于归约函数

all(iterable)	一个谓词，如果 **iterable** 中所有元素都是真，函数返回 **True**，否则返回 **False**
any(iterable)	一个谓词，如果 **iterable** 中存在值为真的元素，函数返回 **True**，否则返回 **False**
enumerate(iterable, start=0)	返回一个迭代器，它给出一个二元组的序列，元组中第一个元素是序号，第二个元素是 **iterable** 的相应元素。默认序号从 **0** 开始，可以通过 **start** 参数指定起始序号
max(iterable, *[, key, default]) **max(arg1, arg2, *args[, key])**	求出 **iterable** 中（或两个或更多实参 **arg*i*** 中）的最大值。可以通过 **key** 提供一个单参数的函数，要求将该函数作用于 **iterable** 的各元素，比较函数值的大小。还可以用 **default** 参数提供一个值，当 **iterable** 为空时返回这个值。如果存在多个最大值，函数返回遇到的第一个
min(iterable, *[, key, default]) **min(arg1, arg2, *args[, key])**	与 **max** 类似，但求出最小值而不是最大值
next(iterator[, default])	返回迭代器 **iterator** 的下一个元素。当迭代器已经没有元素时，如果调用提供了 **default** 值就返回它，否则就报告 **StopIteration** 异常
reversed(seq)	得到一个迭代器，按逆序给出 **seq** 的元素（要求 **seq** 必须是一个序列）
sorted(iterable[, key][, reverse])	返回对 **iterable** 中元素排序的表。可以通过 **key** 提供一个单参数函数，要求按该函数作用于各元素得到的值排序。默认按递增排序，可以用实参 **reverse=True** 要求按递减排序
sum(iterable[, start])	按顺序对**iterable**的元素求和。默认初始值为**0**，可以通过**start**提供初始值
zip(*iterables)	以任意多个可迭代对象作为参数，返回一个迭代器。迭代器给出的元素是元组，元组的元素是参数中对应位置的一组组元素。用完最短的可迭代对象时该迭代器结束。如果只有一个可迭代对象，迭代器给出单元素的元组。**zip()** 返回一个空迭代器。