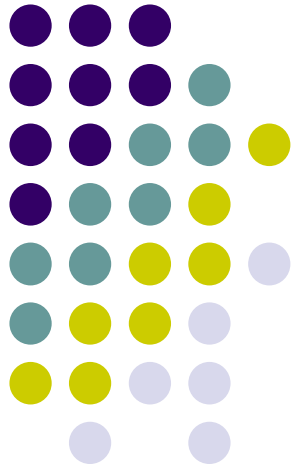
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

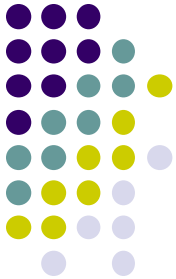
第10章
Algorithms

申丽萍

lpshen@sjtu.edu.cn

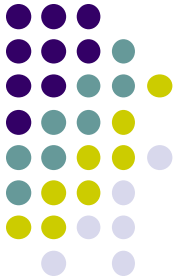


第10章 模块化编程



- 模块化程序设计
- 函数
- 自顶向下设计
- Python模块

查找问题

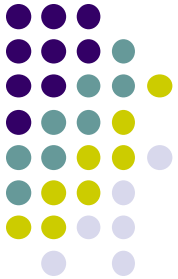


- 问题:在一个列表中查找某个值.

```
def search(x, nums):
    # nums为一数值列表, x是一个数
    # 如果找到,返回x在列表中的位置
    # 否则返回-1
```

- Python本身就提供有关的功能:
  - 判断: `x in nums`
  - 求位置: `nums.index(x)`

策略一:线性查找

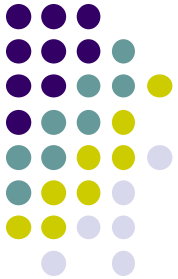


- 逐个检查列表中的数据.

```
def search(x, nums):
    for i in range(len(nums)):
        if nums[i] == x:
            return i
    return -1
```

- 特点:
  - 适合无序数据列表
  - 不适合大数据量
    - 使列表有序后,线性查找算法可略加改进.(How?)

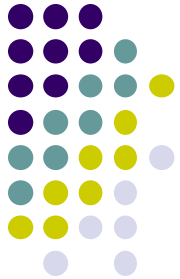
策略二:两分查找



- 猜数游戏:可取的策略?
- 两分查找:每次查看有序列表的中点数据,根据情况接着查找较大一半或较小一半.

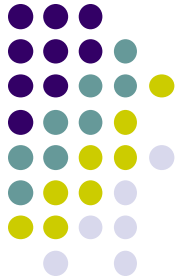
```
def search(x, nums):
    low, high = 0, len(nums) - 1
    while low <= high:
        mid = (low + high) / 2
        if x == nums[mid]:
            return mid
        elif x < nums[mid]:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

算法的优劣比较



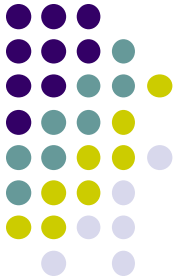
- 经验分析
  - 根据电脑上的运行时间比较
- 抽象分析
  - 分析算法解题所耗"步数"(时间).
  - 步数又与问题难度相关
    - 查找问题中,问题难度用数据量*n*衡量

查找算法的比较



- 策略一
  - 步数与n成正比
  - 称为线性时间算法
- 策略二
  - 步数与$\log_2 n$成正比
  - 称为对数时间算法
- 猜数游戏中:若数的范围是1~1000000,则
  - 策略一:平均要猜50万次才能猜对
    - 最坏1百万次,最好1次
  - 策略二:最坏也只需猜20次

递归定义



- 两分查找算法的另一表述:

  算法binarySearch:在nums[low]~nums[high]中查找x

```
mid = (low + high) / 2
if low > high
    x 不在nums中
elif x < nums[mid]
    在nums[low]~nums[mid-1]中查找x
else
    在nums[mid+1]~nums[high]中查找x
```

- 大问题的子问题仍是同样形式的问题,故仍用解决大问题的算法来解决子问题.

递归定义的特征

- 递归定义完全是合法的,数学里有很多递归定义的对象.如阶乘:

$$n! = \begin{cases} 1, & \text{当} n = 0; \\ n * (n-1)!, & n>0 \end{cases}$$

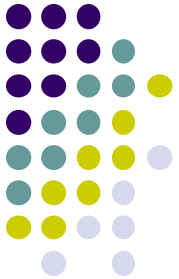
  - 这不是循环定义.
- 递归定义的特征:
  - 有奠基情形,这种情形无需递归;
  - 每次递归都是针对较小情形;
  - 递归链最后终止于奠基情形.

Python递归函数

- 例如:计算阶乘的递归函数

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

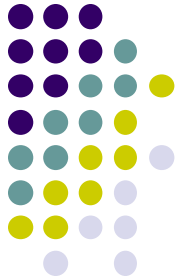
递归查找算法



- 两分查找的递归版本:

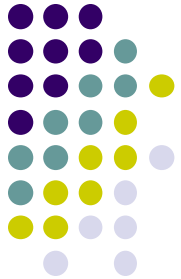
```
def recBinSearch(x, nums, low, high):
    if low > high:
        return -1
    mid = (low + high) / 2
    if x == nums[mid]
        return mid
    elif x < nums[mid]:
        return recBinSearch(x, nums, low, mid-1)
    else:
        return recBinSearch(x, nums, mid+1, high)
def search(x, nums):
    return recBinSearch(x, nums, 0, len(nums)-1)
```

递归vs迭代



- 递归算法
  - 设计容易
  - 易读
  - 效率略低 (stack overflow!)
- 迭代算法:用循环
  - 设计困难
    - 有的问题没有直观的迭代算法
  - 效率高

排序问题

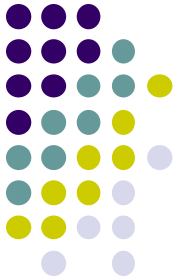


- 给定数据列表,将其数据重新排列,形成一个有序的(递增)列表.
- 回顾:Python列表类型提供了方法

```
<list>.sort()
```

  - 我们要学的是如何实现这个方法,而不是学会用现成的.

朴素策略:选择排序

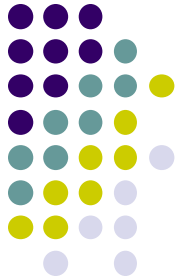


- 每次从剩下的数据中选择最小值输出.
  - 求列表中最小值的算法:参考前面的max算法

```
def selSort(nums):
    n = len(nums)
    for bottom in range(n-1):
        # 求nums[bottom]..nums[n-1]间的最小值
        mp = bottom                      # 初始bottom为迄今最小
        for i in range(bottom+1,n): # 考虑其他值
            if nums[i] < nums[mp]:
                mp = i                   # 新的迄今最小值
        nums[bottom], nums[mp] = nums[mp], nums[bottom]
```

  - 大数据量时效率低.

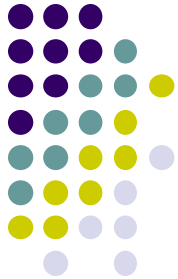
分而治之:归并排序



- 数据分成两组或更多组,分别对各组排序,再把已有序的各组归并(merge)成完整有序列表.
- 归并:比较两组各自的第一个数据,小者输出,由该组的下一个数据顶上来继续比较.
  - 当某组没有数据,则将另一组整个输出.

```
def merge(list1, list2, list3):
    while 当list1和list2两组都有数据:
        输出两组第一个数据的较小者至list3
        更新该组的第一个数据
    while 某组没有数据了:
        将另一组剩余数据输出至list3
```

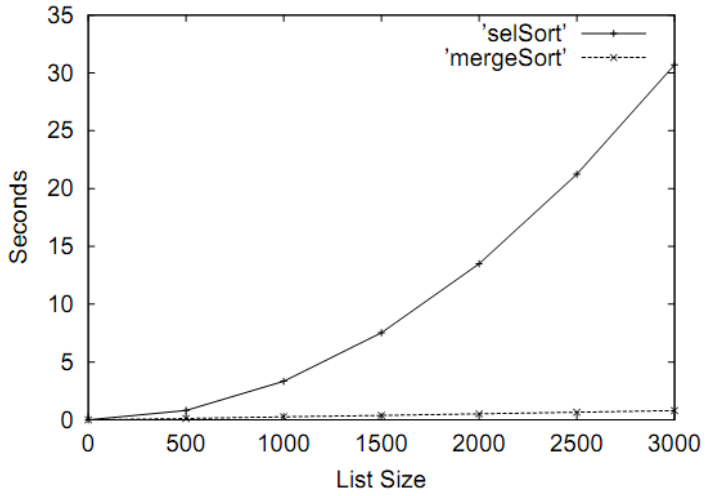
分而治之:归并排序(续)



- 问题:如何对各组排序?
- 似乎可以利用递归:对每一组再次应用分而治之的归并排序.因为满足递归的前提条件:
  - 奠基情形:组中只有一个数据时无需递归;
  - 每次分组都使列表变小,最终会到达奠基情形.

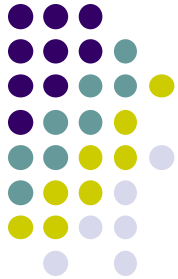
```
def mergeSort(nums):
    n = len(nums)
    if n > 1:
        m = n / 2
        nums1, nums2 = nums[:m], nums[m:]
        mergeSort(nums1)
        mergeSort(nums2)
        merge(nums1, nums2, nums)
```

排序算法的比较



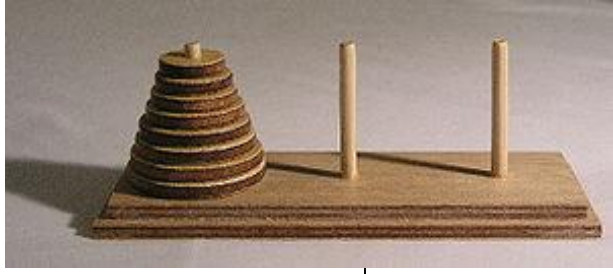
- 难度和列表大小*n*有关.
- 选择排序
  - 每次循环:从剩余数据中选择最小值,所需步数为剩余数据的个数
  - 总的步数: $n+(n-1)+...+1 = n(n+1)/2$
    - 称为n^2算法
- 归并排序
  - 作分组归并图示,可知每层归并都涉及*n*步,共有$\log_2 n$层,故需$n\log_2 n$步.
    - 称为*n*log*n*算法

可计算性与计算复杂性



- 问题可划分为:
  - 可计算的:存在确定的机械过程,一步一步地解决问题.
    - 可计算,而且能有效解决
    - 可计算,但难度太大,不能有效解决
  - 不可计算的:不存在明确的机械过程来求解该问题.
    - 不可解,不可判定

Hanoi塔问题

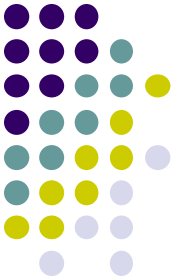


- 体现递归威力的经典问题!

```
def moveTower(n, source, dest, temp):
    if n == 1:
        print "Move disk from", source, "to", dest+"."
    else:
        moveTower(n-1, source, temp, dest)
        moveTower(1, source, dest, temp)
        moveTower(n-1, temp, dest, source)
```

- 难度:需要$2^n - 1$步!
  - 称为指数时间算法
  - 属于难解(*intractable*)问题.
  - 根据Hanoi塔的传说:有64个金盘.就算僧侣1秒移动一次,至少也要花 $2^{64}-1$秒,大约等于5850亿年.

停机问题



- 能否编一个终止性判定程序**HALT?**

  def HALT(prog,data)

  若**prog(data)**终止,则输出**True;**否则输出**False.**

- 是不可解(***unsolvable***)问题!
- 若存在**HALT,**则歌德巴赫猜想可以迎刃而解:

  def gc():

  n = 2

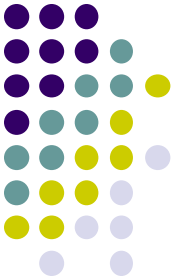
  while True:

  if 2*n 不是两个素数的和,则返回**False**

  n = n + 1

  - 然后运行**HALT(gc)**即可.

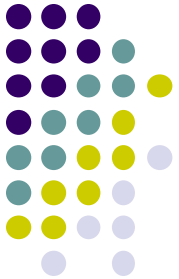
停机问题(续)



- 说HALT不存在只能通过严格证明:
  假设存在HALT(prog,data). 则编程序

```
def strange(p):
    result = HALT(p,p)
    if result == False:   #即p(p)不终止
        return
    else:
        while True: n = 1
```

运行strange(strange),结果如何?



End