# 算法基础

## 1.算法概念

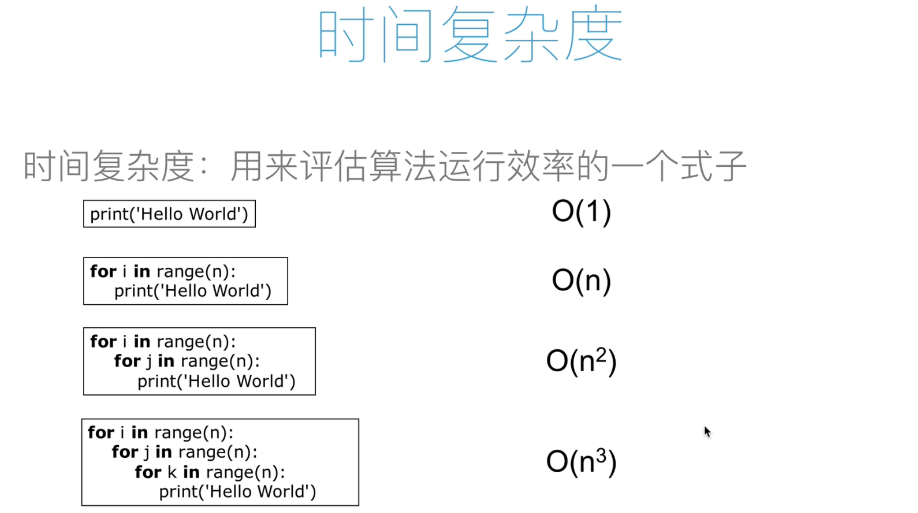
算法(Algorithm)：一个计算过程，解决问题的方法

Niklaus Wirth：“程序=数据结构+算法”

## 2.时间复杂度

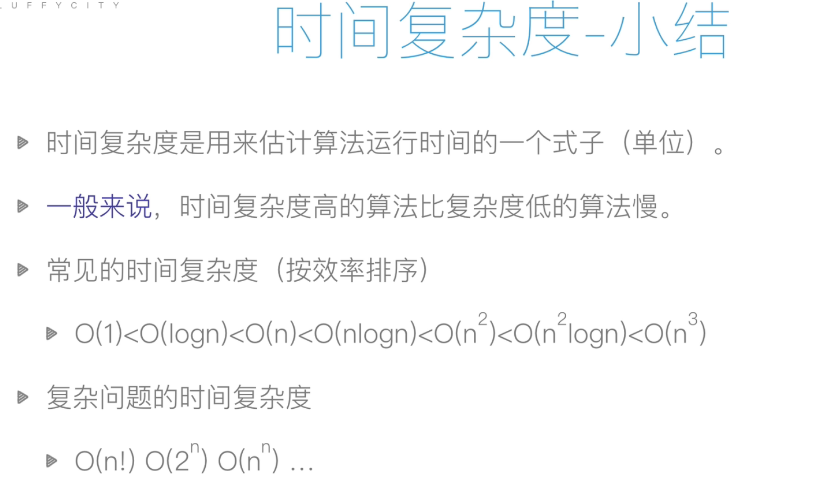
*for* i *in* range(n):  
 print("Hello World")

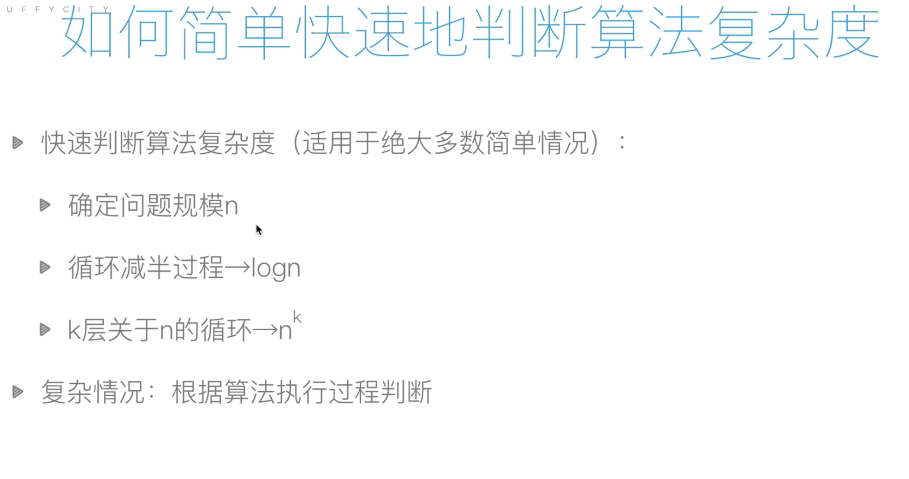
range 中的n ，表示一个问题规模，问题规模的大小影响问题的运行时间。



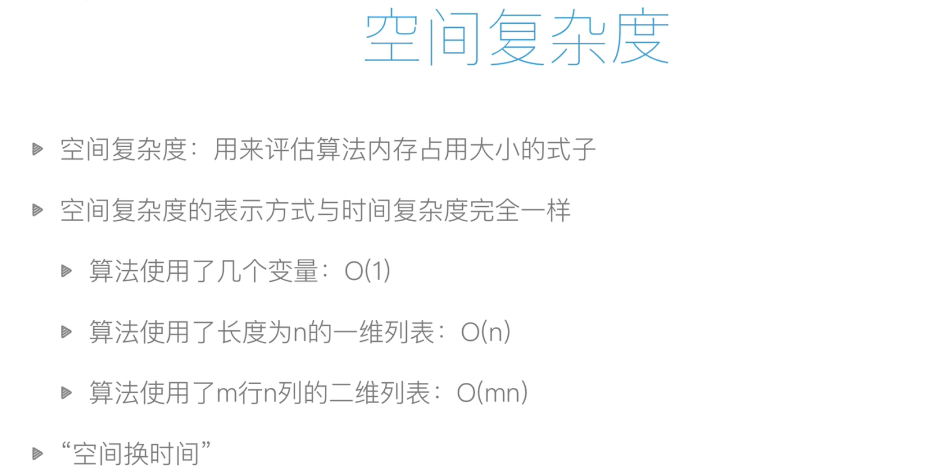
其中的1 n n方 n立方 只是表示一个单位，一个规模的单位







## 3.空间复杂度



## 4.递归

递归的两个特点：

1.调用自身

2.结束条件

汉诺塔：n表示盘子，a，b，c表示柱子,顺序意义是，盘子从 a 经过 b 到c  
*def* hanoi(*n*,*a*,*b*,*c*):  
 *if n* > 0:  
 hanoi(*n*-1,*a*,*c*,*b*)  
 print("moving from %s to %s"%(*a*,*c*))  
 hanoi(*n*-1,*b*,*a*,*c*)  
  
  
hanoi(100,"A","B","C")

## 5.查找

查找：在一些数据元素中，通过一定的方法找出与给定关键字相同对的数据元素的过程

列表查找（线性表查找）：从列表中查找指定元素

输入：列表、待查找元素

输出：元素小标（未找到元素时一般返回None或-1）

内置列表查找函数：index()

### 1.顺序查找（Linear Search）

顺序查找：也叫线性查找，从列表第一个元素开始，顺序进行搜索，知道找到元素或搜索到列表最后一个元素为止。

# li 表示列表；val表示待查找的元素  
# 此时的时间复杂度为：O(n)  
*def* linear\_search(*li*,*val*):  
 *for* ind,v *in* enumerate(*li*):  
 *if* v == *val*:  
 *return* ind  
 *else*:  
 *return None*

### 2.二分查找介绍

二分查找：又叫折半查找，从有序列表的初始候选区li[0:n]开始，通过对待查找的值与候选区中间值的比较，可以使候选区减少一半。

### 3.二分查找代码实现

*def* binary\_search(*li*,*val*):  
 left = 0  
 right = len(*li*) -1  
 *while* left <= right: #候选区还有值  
 mid = (left + right) //2 #mid left right 都表示列表的小标索引  
 *if li*[mid] == *val*:  
 *return* mid  
 *elif li*[mid] > *val*: # 待查找的值在mid左侧  
 right = mid-1  
 *else*: # li[mid] <val 待查找的值在mid右侧  
 left = mid + 1  
  
 *else*: #没有找到，就是不符合 left 不小于等于 right  
 *return None*

二分查找时间复杂度：O(logn)

### 4.二分查找和线性查找比较

*import* time  
  
*def* cal\_time(*func*):  
 *def* wrapper(*\**args,*\*\**kwargs):  
 t1 = time.time()  
 result = func(\**args*,\*\**kwargs*)  
 t2 = time.time()  
 print("%s running time: %s secs."%(func.\_\_name\_\_,t2-t1))  
 *return* result  
 *return* wrapper  
  
@cal\_time  
*def* linear\_search(*li*,*val*):  
 *for* ind,v *in* enumerate(*li*):  
 *if* v == *val*:  
 *return* ind  
 *else*:  
 *return None*@cal\_time  
*def* binary\_search(*li*,*val*):  
 left = 0  
 right = len(*li*) -1  
 *while* left <= right: #候选区还有值  
 mid = (left + right) //2 #mid left right 都表示列表的小标索引  
 *if li*[mid] == *val*:  
 *return* mid  
 *elif li*[mid] > *val*: # 待查找的值在mid左侧  
 right = mid-1  
 *else*: # li[mid] <val 待查找的值在mid右侧  
 left = mid + 1  
  
 *else*: #没有找到，就是不符合 left 不小于等于 right  
 *return None*li = list(range(100000000))  
linear\_search(li,3338900)  
binary\_search(li,3338900)

result：

linear\_search running time: 0.3142251968383789 secs.  
binary\_search running time: 0.0 secs.

结果可以得到：线性查找比二分查找耗时长得多。但是，二分查找弊端，必须是有序列表；

例如：232 如果是按照线性查找某个数字，那可能要查找40多亿次，但是2分查找的话就最多需要32次就可以找到。

如果是一个无序列表，那如果要进行二分查找，那必须先排序，但是排序的时间复杂度就要大于O(n),如果需要一直查找，就可以先排序。

## 6.排序

### 1.列表排序

排序：将一组“无序”的记录序列调整为“有序”的记录序列

列表排序：将无序列表变为有序列表

输入：列表

输出：有序列表

升序与降序

内置排序函数：sort()

### 2.常见排序算法



### 3.冒泡排序（Bubble Sort）

列表每两个相邻的输，如果前面比后面打，则交换这个数。

一趟排序完成后，则无序区减少一个数，有序区增加一个数。

代码关键点：趟、无序区范围

*def* bubble\_sort(*li*):  
 *for* i *in* range(len(*li*)-1): # 第i趟，从0开始  
 *for* j *in* range(len(*li*)-i-1): #列表内自循环，相当于列表里面的元素进行比较时的次数  
 *if li*[j] > *li*[j+1]: # 升序 > ,降序 <  
 *li*[j],*li*[j+1] = *li*[j+1],*li*[j] # 两个值进行交换  
 print(*li*)  
  
  
li = [9,8,7,1,2,3,4,5,6]  
bubble\_sort(li)  
运行结果：  
 [8, 7, 1, 2, 3, 4, 5, 6, 9]  
 [7, 1, 2, 3, 4, 5, 6, 8, 9]  
 [1, 2, 3, 4, 5, 6, 7, 8, 9]  
 [1, 2, 3, 4, 5, 6, 7, 8, 9]  
 [1, 2, 3, 4, 5, 6, 7, 8, 9]  
 [1, 2, 3, 4, 5, 6, 7, 8, 9]  
 [1, 2, 3, 4, 5, 6, 7, 8, 9]  
 [1, 2, 3, 4, 5, 6, 7, 8, 9]

时间复杂度：O(n2) 。可以发现，在第四趟有，无序区其实已经排好了序，因此，后面的循环是没有必要的。下面进行改进。

**冒泡改进：**

*def* bubble\_sort(*li*):  
 *for* i *in* range(len(*li*)-1): # 第i趟，从0开始  
 exchange = *False  
 for* j *in* range(len(*li*)-i-1): #列表内自循环，相当于列表里面的元素进行比较时的次数  
 *if li*[j] > *li*[j+1]: # 升序 > ,降序 <  
 *li*[j],*li*[j+1] = *li*[j+1],*li*[j] # 两个值进行交换  
 exchange = *True* print(*li*)  
 *if not* exchange:  
 *return*li = [9,8,7,1,2,3,4,5,6]  
bubble\_sort(li)  
运行结果：  
 [8, 7, 1, 2, 3, 4, 5, 6, 9]  
 [7, 1, 2, 3, 4, 5, 6, 8, 9]  
 [1, 2, 3, 4, 5, 6, 7, 8, 9]  
 [1, 2, 3, 4, 5, 6, 7, 8, 9]

加一个exchange标志位的理解：意思就是说，在进行排序的过程中前后有数字的位置进行交换，那么，就把exchange=true , 后面的判断语句就不执行。当9,8,7,6都排序好了以后，就会发现，后面的循环冒泡排序其实是没有进行数字的变动的，那此过程中 exchange 就是false，那么，判断语句就会被执行。

提示：每次循环都是遍历列表，对每一个数字进行前后比较。

### 4.选择排序

一趟排序记录最小的数，放到第一个位置

再一趟排序记录列表无序区最小的数，放到第二个位置

…..

算法关键点：有序区和无序区、无序区最小数的位置

# 简单版，不推荐  
*def* select\_sort\_simple(*li*):  
 li\_new = []  
 *for* i *in* range(len(*li*)): # 时间复杂度 O(n)  
 min\_val = min(*li*) # 时间复杂度 O(n)  
 li\_new.append(min\_val)  
 *li*.remove(min\_val) # 时间复杂度 O(n)  
 *return* li\_new  
  
# 进阶版  
*def* select\_sort(*li*):  
 *for* i *in* range(len(*li*)-1): # i 是第几趟  
 min\_loc = i #记录最小值的位置，用于交换  
 *for* j *in* range(i+1,len(*li*)):  
 *if li*[j] < *li*[min\_loc]:  
 min\_loc = j  
 *li*[i],*li*[min\_loc] = *li*[min\_loc],*li*[i]  
 print(*li*)  
  
  
li = [3,5,7,8,4,2,1]  
print(li)  
# print(select\_sort\_simple(li))  
select\_sort(li)

结果：  
[3, 5, 7, 8, 4, 2, 1]  
[1, 5, 7, 8, 4, 2, 3]  
[1, 2, 7, 8, 4, 5, 3]  
[1, 2, 3, 8, 4, 5, 7]  
[1, 2, 3, 4, 8, 5, 7]  
[1, 2, 3, 4, 5, 8, 7]  
[1, 2, 3, 4, 5, 7, 8]

总结：选择排序是，先找到列表中最小的数位置，然后和第一个位置的数进行交换。然后把已经交换的数作为有序区，然后再从无序区进行交换，然后，把无序区里面最小的数的位置和无序区第一个位置进行交换。依次往下。

### 5.插入排序

插入排序就像打纸牌一样

初始时手里（有序区）只有一张牌，每次（从无序区）摸一张牌，插入到手里已有牌的正确位置（大小排序）。

# 时间复杂度 O(n2)  
*def* insert\_sort(*li*):  
 *for* i *in* range(1,len(*li*)): # i 表示摸到牌的下标  
 temp = *li*[i]  
 j = i - 1 # j指的是手里的牌的下标  
 *while* j >=0 *and li*[j] > temp : # 循环主要是找位置  
 *li*[j+1] = *li*[j]  
 j -= 1  
 *li*[j+1] = temp  
 print(*li*)  
  
  
li = [5,2,3,7,6,9,8]  
print(li)  
insert\_sort(li)  
print(li)

## 7.快速排序

### 1.快速排序



*def* partition(*li*,*left*,*right*):  
 tmp = *li*[*left*]  
 *while left* < *right*:  
 *while left*<*right and li*[*right*] >= tmp: # 从右边找比tmp小的数  
 *right* -= 1 # 往左移动一步  
 *li*[*left*] = *li*[*right*] # 把右边的值写到左边空位上  
 print("right==>",*li*)  
 *while left* < *right and li*[*left*] <= tmp:  
 *left* += 1  
 *li*[*right*] = *li*[*left*] # 把左边的值写到右边空位上  
 print("left==>",*li*)  
 *li*[*left*] = tmp # 把tmp 归位  
 *return left*# 快速排序框架  
*def* quick\_sort(*data*,*left*,*right*):  
 *if left* < *right*: # 至少2个元素  
 mid = partition(*data*,*left*,*right*)  
 quick\_sort(*data*,*left*,mid - 1)  
 quick\_sort(*data*,mid + 1,*right*)  
  
li = [5,7,4,6,3,1,2,9,8]  
print(li)  
quick\_sort(li,0,len(li)-1)  
print(li)

以上就是快速排序的方法。

理解：

1. 其中partition是关键，其中tmp要理解，这里是把第一个值作为一个比较的值，
2. 看第一个while语句，只要left(index) 小于right(index)那么就说明列表的长度大于1
3. 第二个while语句，left>right 并且 li[right] >= tmp ，表示只有当列表右边的值小于比较的值的时候，就退出第二个while循环，然后把右边的值，赋值比较的值。此时比较的值是被提取出来的，因此，比较的值的原始位置，就被右边小的值给赋值。相反，如果右边的值一直比左边大，那么就继续while循环，直到结束（特例：当列表只有[5,5]的时候，那么移动或者不移动就没有区别，然后直接走后面的值即可）
4. 第三个while循环的理解和第二个while的理解大同小异。
5. 最后，li[left] = tmp ,其实此时的left和right 已经碰头了，意思就是说整个列表左边的值一定小于等于比较的值，右边的值一定大于等于tmp的值，然后left(index)就是在最中间，把tmp在写进列表里面。
6. 然后以递归的方式，分别去对tmp(mid 最中间那个比较的值)左边的值和右边的值进行一个partition的方式去递归检索排序。

快速排序的效率：

快速排序的时间复杂度O(nlogn)

快速排序的问题：

最坏情况：[9,8,7,6,5,4,3,2,1] 时间复杂度相当于O(n2)

递归:如果递归的层数过多会报错。

*import* sys  
# 修改递归的最大深度  
sys.setrecursionlimit(1000000)

### 2.冒泡排序和快速排序运行比较

*import* time  
*import* random  
*import* copy  
# 计算函数运行时间的装饰器  
*def* cal\_time(*func*):  
 *def* wrapper(*\**args,*\*\**kwargs):  
 t1 = time.time()  
 result = func(\**args*,\*\**kwargs*)  
 t2 = time.time()  
 print("%s running time: %s secs."%(func.\_\_name\_\_,t2-t1))  
 *return* result  
 *return* wrapper  
  
@cal\_time  
*def* bubble\_sort(*li*):  
 *for* i *in* range(len(*li*)-1): # 第i趟，从0开始  
 exchange = *False  
 for* j *in* range(len(*li*)-i-1): #列表内自循环，相当于列表里面的元素进行比较时的次数  
 *if li*[j] > *li*[j+1]: # 升序 > ,降序 <  
 *li*[j],*li*[j+1] = *li*[j+1],*li*[j] # 两个值进行交换  
 exchange = *True  
 if not* exchange:  
 *return  
  
def* partition(*li*,*left*,*right*):  
 tmp = *li*[*left*]  
 *while left* < *right*:  
 *while left*<*right and li*[*right*] >= tmp: # 从右边找比tmp小的数  
 *right* -= 1 # 往左移动一步  
 *li*[*left*] = *li*[*right*] # 把右边的值写到左边空位上  
 *while left* < *right and li*[*left*] <= tmp:  
 *left* += 1  
 *li*[*right*] = *li*[*left*] # 把左边的值写到右边空位上  
 *li*[*left*] = tmp # 把tmp 归位  
 *return left*# 快速排序框架  
*def* \_quick\_sort(*data*,*left*,*right*):  
 *if left* < *right*: # 至少2个元素  
 mid = partition(*data*,*left*,*right*)  
 \_quick\_sort(*data*,*left*,mid - 1)  
 \_quick\_sort(*data*,mid + 1,*right*)  
  
@cal\_time  
*def* quick\_sort(*li*):  
 \_quick\_sort(*li*,0,len(*li*)-1)

li = list(range(10000))  
random.shuffle(li)  
li1 = copy.deepcopy(li)  
li2 = copy.deepcopy(li)  
quick\_sort(li1)  
bubble\_sort(li2)  
  
'''  
结果：  
quick\_sort running time: 0.04754304885864258 secs.  
bubble\_sort running time: 14.84272313117981 secs.  
'''

## 8.堆排序

### 1.堆排序前传-树与二叉树

