算法

1.算法基础

1. 算法的定义和特征

1.1定义

- 一个计算过程,解决问题的方法
- 解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制
- 能够对一定规范的输入，在有限时间内获得所要求的输出
- 如果一个算法有缺陷，或不适合于某个问题，执行这个算法将不会解决这个问题
- 不同的算法可能用不同的时间、空间或效率来完成同样的任务。一个算法的优劣可以用空间复杂度与时间复杂度来衡量

1.2特征

- 有穷性：算法的有穷性是指算法必须能在执行有限个步骤之后终止；
- 确切性：算法的每一步骤必须有确切的定义；
- 输入项：一个算法有0个或多个输入，以刻画运算对象的初始情况，所谓0个输入是指算法本身定出了初始条件；
- 输出项：一个算法有一个或多个输出，以反映对输入数据加工后的结果，没有输出的算法是毫无意义的；
- 可行性：算法中执行的任何计算步骤都是可以被分解为基本的可执行的操作步，即每个计算步都可以在有限时间内完成（也称之为有效性）。

2. 算法设计的要求

- 确定性: 指的是算法至少应该有输入,输出和加工处理无歧义性，能正确反映问题的需求,能够得到问题的正确答案。确定性大体分为四个层次:

    1.算法程序无语法错误;

    2.算法程序对于合法的输入产生满足要求的输出;

    3.对于非法输入能够产生满足规格的说明;

    4.算法程序对于故意刁难的测试输入都有满足要求的输出结果。

- 可读性: 程序便于阅读,理解交流。
- 健壮性: 当输入数据不合法时,算法也能作出相关处理，而不是产生异常,崩溃或者莫名其妙的结果。
- 时间效率高和存储量低

3. 算法的时间复杂度

3.1 定义

- 在进行算法分析时，语句总的执行次数T(n)是关于问题规模n的函数，进而分析T(n)随n的变化情况并确定T(n)的数量级
- 算法的时间复杂度，也就是算法的时间量度，记作：T(n}=0(f(n))

- 它表示随问题规模n的增大，算法执行时间的埔长率和 f(n)的埔长率相同，称作算法的渐近时间复杂度，简称为时间复杂度。其中f(n)是问题规横n的某个函数

3.2 求解算法的时间复杂度的具体步骤

- 找出算法中的基本语句 (算法中执行次数最多的那条语句就是基本语句，通常是最内层循环的循环体)
- 计算基本语句的执行次数的数量级(只需计算基本语句执行次数的数量级，这就意味着只要保证基本语句执行次数的函数中的最高次幂正确即可，可以忽略所有低次幂和最高次幂的系数。这样能够简化算法分析，并且使注意力集中在最重要的一点上：增长率)
- 用大O记号表示算法的时间性能(将基本语句执行次数的数量级放入大O记号中)

3.3 推导大o阶的基本的推导方法：

- 用常数1取代运行时间中的所有加法常数。
- 在修改后的运行次数函数中，只保留最高阶项。
- 如果最高阶项存在且不是1,则去除与这个项相乘的常数。

简单的说，就是保留求出次数的最高次幂，并且把系数去掉。 如T(n)=n2+n+1 =O(n2)

```
#复杂度O(1)
print("this is wd")

#复杂度O(n)
for i in range(n):
    print(i)

#复杂度O(n2)
for i in range(n):
    for j in range(n):
        print(j)

#复杂度O(n3)
for i in range(n):
    for j in range(n):
        for k in range(n):
            print('wd')

#复杂度O(log2n)
while n > 1:
    print(n)
    n = n // 2
```

常见的复杂度按效率排序：O(1)<O(logn)<O(n)<O(nlogn)<O(n2)<O(2nlogn)<O(n2)

4 算法空间复杂度

空间复杂度(Space Complexity)是对一个算法在运行过程中临时占用存储空间大小的量度(三个方面)

- 包括存储算法本身所占用的存储空间(存储算法本身所占用的存储空间与算法书写的长短成正比，要压缩这方面的存储空间，就必须编写出较短的算法)
- 算法的输入输出数据所占用的存储空间(算法的输入输出数据所占用的存储空间是由要解决的问题决定的，是通过参数表由调用函数传递而来的，它不随本算法的不同而改变)

- 算法在运行过程中临时占用的存储空间(算法在运行过程中临时占用的存储空间随算法的不同而异，有的算法只需要占用少量的临时工作单元，而且不随问题规模的大小而改变，这种算法是节省存储的算法；有的算法需要占用的临时工作单元数与解决问题的规模n有关，它随着n的增大而增大，当n较大时，将占用较多的存储单元)

具体表述

- 当一个算法的空间复杂度为一个常量，即不随被处理数据量n的大小而改变时，可表示为O(1)；
- 当一个算法的空间复杂度与以2为底的n的对数成正比时，可表示为0(log2n)；
- 当一个算法的空间复杂度与n成线性比例关系时，可表示为0(n).若形参为数组，则只需要为它分配一个存储由实参传送来的一个地址指针的空间，即一个机器字长空间；若形参为引用方式，则也只需要为其分配存储一个地址的空间，用它来存储对应实参变量的地址，以便由系统自动引用实参变量。

2.递归 、二分查找

1. 递归

1.1 递归的特点

- **调用自身**
- **结束条件**

判断下列函数是否是递归

```
def func(x):
    print(x)
    func(x-1)

#没有结束条件   --> 递归出口
def func(x):
    if x > 0:
        print (x)
        func(x +1)
```

```
# 递归调用之前对x进行 打印
def func(x):
    if x > 0:
        print(x)
        func(x-1)
func(3)

# 输出结果 3,2,1
#递归调用之后进行调用
def func(x):
    if x > 0:
        func(x-1)
        print(x)
func(3)

# 输出结果 1,2,3
```

1.2 递归实例 -- 斐波那契问题

斐波那契数列:

斐波那契数列数列从第3项开始,每一项都等于前两项之和。 例子:数列 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 .....

求解斐波那契第 n 项

- **使用递归**

  时间复杂度 O(2^n)

```
#使用递归,但是注意此时最后的return 语句不是尾递归
def func(n):
    if n==1 or n==2:
        return 1
    else:
        return func(n-1) + func(n-2)
```

  使用递归慢的原因: 由于需要多次的重复计算,导致很慢

- **使用递归的改进方法 :**

  通过记录值解决重复计算的问题,但是时间复杂度没有改变

```
def func(n):
    #创建一个列表,此时列表中的元素 未被赋值前是-1 ,当赋值后变为正数
    lst = [-1 for i in range(n+1)]
    def fib(n):
        if  n == 0 or n == 1:
            return 1
        elif lst[n] >= 0:
            return lst[n]
        else:
            v = fib(n-1) + fib(n-2)
            #只是记录一下,下次再用的时候直接扔出去
            lst[n] = v
            return lst[n]
    return fib(n)
```

- **使用列表,将后一项添加到列表中,return 列表最后一项**

  时间复杂度O(n)

```
def func(n):
    res = [1,1]
    for i in range(2,n+1):
        res.append(res[-1] + res[-2])
        return res[-1]
```

  上述方式的缺点: 虽然时间复杂度为O(n) ,但是创造了列表 ,空间复杂度增加

- **使用3变量的方式**

  ,时间复杂度为 O(n) ,但是省空间

```
def func(n):
    if n == 0 or n == 1:
        return 1
    a = 1
    b = 1
    c = 0  #c可以是任何值

    for i in range(2,n+1):
        c = a + b
        a = b
        b = c
    return c
```

更优化版:

```
def func(n):
    a = 1
    b = 1
    c = 1
    for i in range(2,n+1):
        c = a + b
        a = b
        b = c
    return c
```

1.3 递归实例2 - 汉诺塔问题

问题描述:

- 假设有三个命名为 A B C 的塔座， 在塔座A上插有n个直径大小不相同，由小到大编号为1， 2， 3， …， n的圆盘，要求将A座上的圆盘移至塔座C
- 并按同样的顺序叠排

圆盘移动规则:

- 每次只能移动一个圆盘
- 圆盘可以插在任意一个塔座上
- 任何时刻都不能将一个较大的圆盘放在一个较小的圆盘上

问题简化 :

- 将 n - 1 个圆盘 看成一个整体,将其从A经过C移动到B
- 将第 n个盘子从A移动到C
- 将开始的n - 1 个盘子从B 经过A移动到C

代码书写

- 汉诺塔移动次数的递推式 h(x) = 2h(x-1) +1

```
def hanoi(n,A,B,C):
    '''
    :param n: 需要从A移到C 的盘子总数
    :param A: 移动盘子的起始柱子
    :param B: 移动过程中经过的柱子
    :param C: 移动结束的柱子
    '''
    if n > 0 :
        hanoi(n-1,A,C,B)              #将起始A柱子上的n-1个盘子通过C移动到B
        print("%s -> %s" %(A,C))      #将底层的盘子从A放到C
        hanoi(n-1,B,A,C)              #将B上的n-1 个盘子经由A移动到C
```

1.4 题目练习

问题描述:

- 一段有n个台阶组成的楼梯，小明从楼梯的最底层向最高处前进，
- 他可以选择一次迈一级台阶或者一次迈两级台阶。问：他有多少种不同的走法?

2. 列表查找 - 二分查找

列表查找 :从列表中查找指定元素

- 输入: 列表,带查找元素
- 输出: 元素下标或未查找到的元素

顺序查找:从列表的第一个元素开始,顺序进行搜索,直到找到为止

```
# 顺序查找代码,时间复杂度为O(n)
def linear_search(data,value):
    '''
    :param data: 查找的目标列表
    :param value:目标查找元素
    :return:
    '''
    for i in range(data):
        if data[i] == value:
            return i
    return
```

二分查找: 从有序列表的候选区data[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

```
def binary_search(lst,val):
    left = 0
    right = len(lst) -1
    while left <= right:
        mid = (left + right) // 2
        if val == lst[mid]:
            return mid
        elif val < lst[mid]:
            right = mid -1
        else:
            left = mid +1
    return
```

递归版本的二分查找

```
def bin_search(lst,val,left,right):
    if left <= right:
        mid = (left + right) // 2
        if lst[mid] == val:
            return mid
        elif lst[mid] > val:
            return bin_search(lst,val,left,mid-1)
        else:
            return bin_search(lst,val,mid+1,right)
    else:
        return

# 上述return属于尾递归 ：  编译器会优化成迭代 但是Python没有针对尾递归进行优化
```

3.LB排序

1. 冒泡排序

时间复杂度 : O(n2)

原理:

- 比较相邻的元素，如果后一个比前一个大就进行交换
- 对每一对相邻元素做同样的工作，完成以后，最后的元素会是最大的数，这里可以理解为走了一趟；
- 针对所有的元素重复以上的步骤，除了最后一个

冒泡排序代码

```
import random
from cal_time import cal_time

@cal_time
def bubble_sort(lst):
    for i in range(len(lst)-1):  # i 表示要进行冒泡比较的趟数
        # 第i趟的无序区位置 [0,n-1-i]
        for j in range(len(lst) -i -1):
            if lst[j] > lst[j+1]:
```

```
                #下一个数比当前的大就进行交换
                lst[j],lst[j+1] = lst[j+1],lst[j]

# 测试:
lst = list(range(10000))
random.shuffle(lst)
bubble_sort(lst)
#bubble_sort running time: 7.759883880615234 secs
```

代码做一次优化:

```
@cal_time
def bubble_sort(lst):
    for i in range(len(lst)-1):  # i 表示要进行冒泡比较的趟数
        # 第i趟的无序区位置 [0,n-1-i]
        exchange = False
        for j in range(len(lst) -i -1):
            if lst[j] > lst[j+1]:
                #下一个数比当前的大就进行交换
                lst[j],lst[j+1] = lst[j+1],lst[j]
                exchange = True
        if not exchange:
            return
```

对于时间复杂度的具体讨论:

- 最好情况 :此时的列表已将排好序,此时无序进行交换 时间复杂度为 O(n)
- 平均情况 :此时的列表为正常的无序状态, 时间复杂度为O(n^2)
- 最坏情况 :此时列表是倒序,每次都需要进行交换 时间复杂度为O(n^2)

2. 选择排序

- 一趟遍历记录最小的数,放到第一个位置
- 再一趟遍历记录剩余列表中最小的数,继续放置

关键:

- 无序区
- 最小数的位置

```
def find_min(lst):
    min_num = lst[0]
    for i in range(1,len(lst)):
        if lst[i] < min_num:
            min_num = lst[i]
    return min_num

#获取最小数的位置
def find_min_pos(lst):
    min_pos = 0
    for j in range(1,len(lst)):
        if lst[j] < lst[min_pos]:
```

```
            min_pos = j
    return min_pos
```

代码 时间复杂度 O(n^2)

```
import  random
from cal_time import cal_time

@cal_time
def select_sort(lst):
    for i in range(len(lst) -1):
        #第 i趟无序区 [i,len(lst)-1]
        #找无序区最小数的位置,和无序区第一个数进行交换
        min_pos = i
        for j in range(i+1,len(lst)):
            if lst[j] < lst[min_pos]:
                min_pos = j
        lst [min_pos],lst[i] = lst[i],lst[min_pos]


li = list(range(10000))
random.shuffle(li)
select_sort(li)

#select_sort running time: 3.5867788791656494 secs.
```

3. 插入排序

- 将列表分为有序区和无序区两个部分,最初有序区只有一个元素
- 每次从无序区选择一个元素,插入到有序区的位置,知道无序区变空

```
import random
from cal_time import cal_time

@cal_time
def insert_sort(lst):
    for i in range(1,len(lst)):  # 从无序区摸排,表示第i趟,i表示摸到的牌
        tmp = lst[i]             # tmp表示摸到的牌的值
        #摸到的牌前一张,即有序区最后一张牌的下标为j
        j = i - 1
        while j >=0 and lst[j] > tmp: #只要往后挪就循环,2个条件都满足
            #如果j =-1 就停止挪,如果lst[j]小了就停止挪
            lst[j+1] = lst[j]
            j -=1
        #j位置在循环结束的时
        # 候要么是-1 要么是一个比tmp小的值的下标
        lst[j+1] = tmp
li = list(range(10000))
random.shuffle(li)
insert_sort(li)
```

4.NB排序

1. 快速排序

- 取一个元素p（第一个元素），使元素p归位；
- 列表被p分成两部分，左边都比p小，右边都比p大；
- 递归完成排序

1.1 快排代码书写

```
import random
from cal_time import cal_time

def _quick_sort(lst,left,right):
    if left < right:
        mid = partition(lst,left,right)
        _quick_sort(lst,left,mid-1)
        _quick_sort(lst,mid+1,right)

def partition(lst,left,right):
    tmp = lst[left]

    while left < right:
        while left < right and lst[right] >= tmp:
            right -=1
        #当这个while循环退出时,将右边比tmp大的数放到左边
        lst[left] = lst[right]
        #此时进入下边的while循环,从左边选出比tmp大的数去田右边刚留出的空
        while left < right and lst[left] <= tmp:
            left +=1
        #循环退出时候,将比tmp大的数放到右边
        lst[right] = lst[left]
    #循环退出
    lst[left] = tmp
    return left


@cal_time
def quick_sort(li):
    _quick_sort(li, 0, len(li)-1)

li = list(range(10000))
random.shuffle(li)

quick_sort(li)

#quick_sort running time: 0.021941661834716797 secs.
```

1.2 快排的时间复杂度 O(nlogn)

- **递归深度并不是很大**，因为他是分层的,每次大概少一半

1.3 快排的问题

- 最坏情况: 当所有的数是倒序 ,此时不是logn层,每次递归只让元素少一个,时间复杂度编程 O(n^2) (可以通过修改partition函数进行解决)

```
def partition(li, left, right):
    i = random.randint(left, right)
    li[i], li[left] = li[left], li[i]
    tmp = li[left]
    while left < right:
        while left < right and li[right] >= tmp:
            right -= 1
        li[left] = li[right]
        while left < right and li[left] <= tmp:
            left += 1
        li[right] = li[left]
    li[left] = tmp
    return left
```

- 递归

1.4 另一种partition的写法

```
#将最后一个元素归为
def partition2(li, left, right):
    # 区域1: [left, i] 区域2: [i+1, j-1]
    i = left - 1 # 初识区域1和区域2都空  left = i此时就时有值的所以只能-1
    for j in range(left, right):
        if li[j] < li[right]: # 归到区域1
            i += 1
            li[j], li[i] = li[i], li[j]
    li[right], li[i+1] = li[i+1], li[right]
    return i+1
```

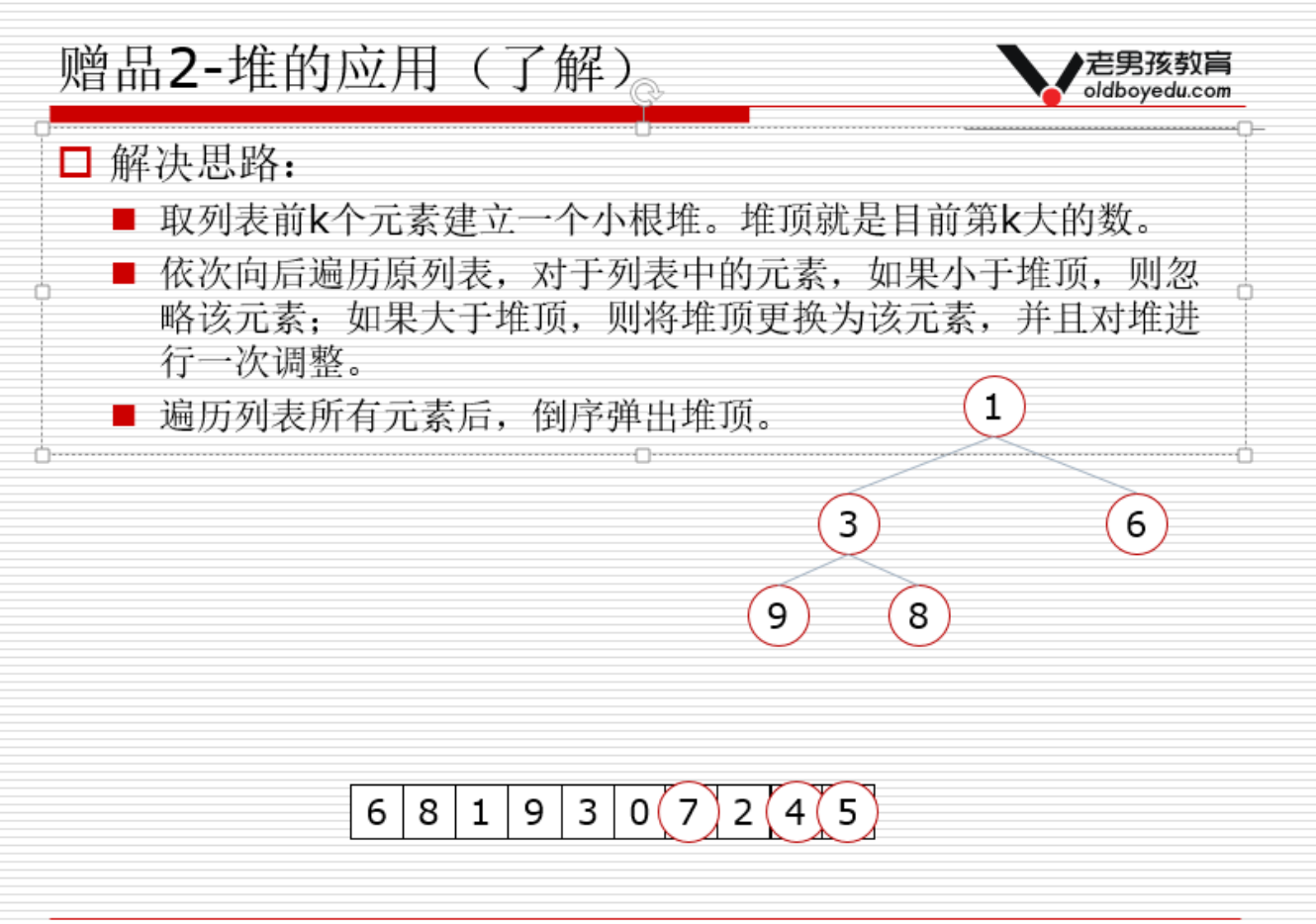
2. 堆排序

```
#堆排序的过程
建立堆
得到堆顶元素，为最大元素
去掉堆顶，将堆最后一个元素放到堆顶，此时可通过一次调整重新使堆有序。
堆顶元素为第二大元素。
重复步骤3，直到堆变空。
```

```
import random
from cal_time import cal_time

# O(log n)
def sift(li, low, high):
    # low 表示堆顶下标， high表示堆中最后一个元素下标
    tmp = li[low]
    i = low
    j = 2 * i + 1
```

```
    while j <= high:  # 第二种循环退出情况，没有孩子和tmp竞争i这个位置
        if j+1 <= high and li[j+1] > li[j]:  # 如果右孩子存在并且比左孩子大 j指向右孩子
            j += 1
        if li[j] > tmp:
            li[i] = li[j]
            i = j
            j = 2 * i + 1
        else:
            break  # 第一种循环退出情况，tmp比目前两个孩子都大
    li[i] = tmp

@cal_time
def heap_sort(li):
    # 1. 从列表构造堆 low的值和high的值
    n = len(li)
    for low in range(n//2-1, -1, -1):
        sift(li, low, n-1)
    # 2. 挨个出数 利用原来的空间存储下来的值，但是这些值不属于堆
    for high in range(n-1, -1, -1): # range(n-1,0,-1)
        li[high], li[0] = li[0], li[high] # 1.退休 2.棋子
        sift(li, 0, high-1) # 3.调整


li = list(range(100000))
random.shuffle(li)
heap_sort(li)
```

堆的应用




```
def topk(li, k):
    heap = li[0:k]
    for i in range(k // 2 - 1, -1, -1):
        sift(heap, i, k-1)
    for i in range(k, len(li)):
        if li[i] > heap[0]:
            heap[0] = li[i]
            sift(heap, 0, k - 1)
    for i in range(k - 1, -1, -1):
        heap[0], heap[i] = heap[i], heap[0]
        sift(heap, 0, i - 1)
```

3. 归并排序

4.two -sum

```
# [(0,2),(1,1),(2,3),(3,4)] 3
#
```

```
li = [1,3,6,9,13,18,21,26,33]
target = 31


def two_sum_1(li, target):
    for i in range(len(li)):
        for j in range(i+1, len(li)):
            if li[i] + li[j] == target:
                return i, j
    return -1,-1


def two_sum_2(li, target):
    def binary_search(li, val, low, high):
        while low <= high:  # 只要候选区有值
            mid = (low + high) // 2
            if val == li[mid]:
                return mid
            elif val < li[mid]:
                high = mid - 1
            else:  # val > li[mid]
                low = mid + 1
        return -1

    for i in range(len(li)):
        a = li[i]
        b = target - a
        j = binary_search(li, b, i+1, len(li)-1)
        if j >= 0:
            return i, j
    return -1,-1


# 有序列表 最优解法
# 3 sum 最优解法 = (1. 带原下标排序; 2. 一层for循环定一个, 然后用算法两边向中间找) O(n^2)
def two_sum_3(li, target):
    i = 0
    j = len(li)-1
    while i < j:
        x = li[i] + li[j]
        if x < target:
            i += 1
        elif x > target:
            j -= 1
        else:
            return i, j
    return -1,-1

# li = [1,3,6,9,13,18,21,26,33]
# target = 31
# {1:0, 3:1, 6:2, 9:3, 13:4, }
```

```
# 101.101001

# O(n) 需要空间复杂度
def two_sum_4(li, target):
    dic = {}
    for i in range(len(li)):
        a = li[i]
        b = target - a
        if b in dic:
            return dic[b], i
        else:
            dic[a] = i


print(two_sum_4(li, target))
```