# 算法设计与分析——Python版实验指导书

## 实验1算法概述实验(2学时)

**一、 实验目的**

1. 明确算法的研究内容和研究目标；

2. 能够按照算法设计的一般步骤完成实际问题的算法设计；

3. 掌握算法分析方法，主要包括时间复杂度分析和空间复杂度分析；

4. 给定具体算法，能够分析算法渐进意义下的复杂度。

**二、 实验环境**

1. Windows操作系统或Linux操作系统；

2. Python3.x；

3. PyCharm或Sublime或Jupyter Notebook。

**三、 实验内容(任选一个)**

1. 排序问题： 给定一个无序序列，采用以下排序方法分别对序列升序排序。

(1) 冒泡排序；

(2) 堆排序；

(3) 插入排序；

(4) 选择排序。

2. 统计数字问题：一本书的页码从自然数1开始顺序编码直到自然数n。书的页码按照通常的习惯编排，每个页码都不含多余的前导数字0。例如第6页用6表示而不是06或006。数字统计问题要求对给定书的总页码，计算出书的全部页码中分别用到多少次数字0,1,2,3,…9。

3. 字典序问题： 在数据加密和数据压缩中需要对特殊的字符串进行编码。给定的字母表由26个小写字母组成。该字母表产生的升序字符串是指字符串中字母从左到右出现的次序与字母在字母表中出现的次序相同，且每个字符最多出现1次。例如，a,b,ab,bc,xyz等都是升序字符串。现在对字母表中产生的所有长度不超过6的升序字符串按照字典序排列并编码如下：

12…262728…

ab…zabac…

对于任意长度不超过6的升序字符中，迅速计算出它在上述字典中的编码。

任务： 对于给定的长度不超过6的升序字符串，编程计算它在上述字典中的编码。

**四、 实验步骤**

1. 认真审阅题目，明确题目的已知条件和求解的目标；

2. 问题建模；

3. 算法设计；

4. 编码实现(用Python语言)；

5. 测试数据；

6. 程序运行结果；

7. 分析实验结果是否符合预期，如果不符合，分析可能的原因；

8. 算法分析；

9. 实验总结： 总结实验中遇到的问题及解决方法。

**五、 实验报告**

1. 封皮。

2. 正文：

(1) 实验目的；

(2) 实验平台；

(3) 实验内容；

(4) 算法设计(问题分析、建模、算法描述)；

(5) 算法源码；

(6) 测试数据；

(7) 程序运行结果(要求： 截图说明算法运行的结果)；

(8) 算法分析(分析算法的时间复杂度和空间复杂度)；

(9) 实验总结(实验中遇到的问题、解决方法、实验收获等)。

**六、参考代码**

1.排序问题

（1）冒泡排序：

def bubble\_sort(nums):

for i in range(len(nums) - 1):

swap\_flag = False # 改进后的冒泡，设置一个交换标志位

for j in range(len(nums) - i - 1):

if nums[j] > nums[j + 1]:

nums[j], nums[j + 1] = nums[j + 1], nums[j]

swap\_flag = True

if not swap\_flag:

return nums #若没有元素交换，则表示已经有序

return nums

（2）堆排序

#堆的调整

def heap\_adjust(L, start, end):

temp = L[start]

i = start

j = 2 \* i

while j <= end:

if (j < end) and (L[j] < L[j + 1]):

j += 1

if temp < L[j]:

L[i] = L[j]

i = j

j = 2 \* i

else:

break

L[i] = temp

import math

def heap\_sort(L):

L\_length = len(L) - 1

first\_sort\_count = math.floor(L\_length / 2)

for i in range(first\_sort\_count):

heap\_adjust(L, first\_sort\_count - i, L\_length)

for i in range(L\_length - 1):

L = swap\_param(L, 1, L\_length - i)

heap\_adjust(L, 1, L\_length - i - 1)

return [L[i] for i in range(1, len(L))]

（3）插入排序

def insert\_sort(data):

for i in range(1, len(data)):

value = data[i] # 插入值

j = i-1 # 记录插入位置

while j >= 0:

if data[j] > value:

data[j+1] = data[j]

else :

break

j -= 1

data[j+1] = value#j+1是插入的位置

return data

（4）选择排序

def selection\_sort(arr):

# 第一层for表示循环选择的遍数

for i in range(len(arr) - 1):

# 将起始元素设为最小元素

min\_index = i

# 第二层for表示最小元素和后面的元素逐个比较

for j in range(i + 1, len(arr)):

if arr[j] < arr[min\_index]:

# 如果当前元素比最小元素小，则把当前元素角标记为最小元素角标

min\_index = j

# 查找一遍后将最小元素与起始元素互换

arr[min\_index] , arr[i] = arr[i] , arr[min\_index]

return arr

2.字典序问题

def dfs(i, k):

sum = 0

if k == 1:

return 1

for j in range(i + 1, 27):

sum += dfs(j, k - 1)

return sum

def fun(k):

sum = 0

for i in range(1, 27):

sum += dfs(i, k)

return sum

def main():

try:

while True:

s = input()

ans, l, temp = 1, len(s), 0

for i in range(1, l):

ans += fun(i)

for i in range(l):

num = ord(s[i]) - ord('a') + 1

len2 = l - i

for j in range(temp + 1, num):

ans += dfs(j, len2)

temp = num

print(ans)

except EOFError:

pass

3.统计数字问题

def count\_num2(page\_num):

page\_list = range(1,page\_num + 1)

result = [0 for i in range(10)]

for page in page\_list:

page = str(page)

for i in range(10):

temp = page.count(str(i))

result[i] += temp

return result

## **实验2贪心算法实践**

**一、 实验目的**

1. 了解贪心算法思想及基本原理；

2. 掌握使用贪心算法求解问题的一般特征；

3. 能够针对实际问题，能够正确选择贪心策略；

4. 能够针对选择的贪心策略，证明算法的正确性；

5. 能够根据贪心策略，正确编写代码；

6. 能够正确分析算法的时间复杂度和空间复杂度。

**二、 实验环境**

1. Windows操作系统或Linux操作系统；

2. Python3.x；

3. PyCharm或Sublime或Jupyter Notebook。

**三、 实验内容(任选一个)**

1. 最优合并问题： 给定k个排好序的序列s1,s2,…,sk，用2路合并算法将这k个序列合并成一个序列。假设所采用的2路合并算法合并两个长度分别为m和n的序列需要m+n-1次比较。试设计一个算法确定合并这个序列的最优合并顺序，使所需要的总比较次数最少。

2. 程序存储问题： 设有n 个程序{1,2,…,n }要存放在长度为L的磁带上。程序i存放在磁带上的长度是li1≤i≤n。程序存储问题要求确定这n 个程序在磁带上的一个存储方案，使得能够在磁带上存储尽可能多的程序。对于给定的n个程序存放在磁带上的长度，编程计算磁带上最多可以存储的程序数。

3. 最优服务次序问题： 设有n个顾客同时等待一项服务。顾客i需要的服务时间为ti,1≤i≤n。共有s处可以提供此服务。应如何安排n个顾客的服务次序才能使平均等待时间达到最小平均等待时间是n个顾客等待服务时间的总和除以n。

**四、 实验步骤**

1. 认真审阅题目，明确题目的已知条件和求解的目标；

2. 问题建模；

3. 算法设计；

4. 编码实现(用Python语言)；

5. 测试数据；

6. 程序运行结果；

7. 分析实验结果是否符合预期，如果不符合，分析可能的原因；

8. 算法分析；

9. 实验总结： 总结实验中遇到的问题及解决方法。

**五、 实验报告**

1. 封皮

2. 正文：

(1) 实验目的；

(2) 实验平台；

(3) 实验内容；

(4) 算法设计(问题分析、建模、算法描述)；

(5) 算法源码；

(6) 测试数据；

(7) 程序运行结果(要求： 截图说明算法运行的结果)；

(8) 算法分析(分析算法的时间复杂度和空间复杂度)；

(9) 实验总结(实验中遇到的问题、解决方法、实验收获等)。

**六、参考代码**

1. 最优合并问题：from heapq import heappop,heapify,heappush

class HeapNode:

def \_\_init\_\_(self, char, freq):

self.char = char

self.freq = freq

def \_\_lt\_\_(self, other):

return self.freq < other.freq

def \_\_eq\_\_(self, other):

if(other == None):

return False

if(not isinstance(other, HeapNode)):

return False

return self.freq == other.freq

#创建最初极小堆

def create\_heap(frequency, data\_set):

if(len(frequency) != len(data\_set)):

raise Exception('数据和标签不匹配!')

nodes = []#极小堆

for i in range(len(data\_set)):

heappush(nodes,HeapNode(data\_set[i],frequency[i]) )

return nodes

# 求最少比较次数

import copy

def best\_merge(nodes):

global min\_count

tree\_nodes = nodes.copy()

while len(tree\_nodes) > 1:

new\_left = heappop(tree\_nodes)

new\_right = heappop(tree\_nodes)

min\_count = min\_count + new\_left.freq + new\_right.freq - 1

new\_node = HeapNode(None, (new\_left.freq + new\_right.freq))

heappush(tree\_nodes,new\_node)

if \_\_name\_\_ == '\_\_main\_\_':

min\_count = 0

data\_set = ['s1','s2','s3','s4']

frequency = [5,12,11,2]

nodes = create\_heap(frequency,data\_set)#创建初始堆

best\_merge(nodes)#求最少比较次数

print("最少比较次数为：",min\_count)

2. 程序存储问题：

def programm\_store():

global n,l,file\_len

file\_len.sort()

temp\_len = 0

result\_num = 0

for i in range(n):

if temp\_len + file\_len[i]<= l:

temp\_len += file\_len[i]

result\_num += 1

return result\_num

3. 最优服务次序问题：

def programm\_store():

global n,s,s\_time

s\_time.sort()

time=[0]\*n

sum\_time=[0]\*n

i,j=0,0

while i<n:

time[j]+= s\_time[i]

sum\_time[j]+=time[j]

i+=1

j+=1

if j==s:

j=0

k,t=0,0

while k<n:

t+=sum\_time[k]

k+=1

return t//n

## **实验3分治算法实践**

一、 实验目的

1. 了解分治策略算法思想及基本原理；

2. 掌握使用分治算法求解问题的一般特征；

3. 掌握分解、治理的方法；

4. 能够针对实际问题，能够正确的分解、治理，设计分治算法；

5. 能够正确分析算法的时间复杂度和空间复杂度。

二、 实验环境

1. Windows操作系统或Linux操作系统；

2. Python3.x；

3. PyCharm或Sublime或Jupyter Notebook。

三、 实验内容(任选一个)

1. 大整数乘法问题： 给定两个n位的大整数A、B，求A与B的乘积。

2. 最小值问题： 求n个元素的最小值。

3. 棋盘覆盖问题： 给定一个2k×2k的棋盘(如图32所示k=2的一种棋盘)，有一个特殊棋格，拥有一个特殊棋格的棋盘称为特殊棋盘。现要用四种L型骨牌(如图33所示)覆盖特殊棋盘上除特殊棋格外的全部棋格，不能重叠，找出覆盖方案。

4. 幂乘问题： 给定实数a和自然数n，求an。

四、 实验步骤

1. 认真审阅题目，明确题目的已知条件和求解的目标；

2. 问题建模；

3. 算法设计；

4. 编码实现(用Python语言)；

5. 测试数据；

6. 程序运行结果；

7. 分析实验结果是否符合预期，如果不符合，分析可能的原因；

8. 算法分析；

9. 实验总结： 总结实验中遇到的问题及解决方法。

五、 实验报告

1. 封皮

2. 正文：

(1) 实验目的；

(2) 实验平台；

(3) 实验内容；

(4) 算法设计(问题分析、建模、算法描述)；

(5) 算法源码；

(6) 测试数据；

(7) 程序运行结果(要求： 截图说明算法运行的结果)；

(8) 算法分析(分析算法的时间复杂度和空间复杂度)；

(9) 实验总结(实验中遇到的问题、解决方法、实验收获等)。

六、参考代码

1. 大整数乘法问题： from math import log2,ceil

#将给定的整数高位补0，补充够2^n位

def add\_zero(str1,real\_len,max\_len):

add\_zero\_len = max\_len - real\_len

return "0" \* add\_zero\_len + str1

def kara(n1,n2):

if n1 < 10 or n2 < 10:

return n1 \* n2

n1\_str = str(n1)

n2\_str = str(n2)

n1\_len = len(n1\_str)

n2\_len = len(n2\_str)

real\_len = max(n1\_len,n2\_len)

max\_len = 2 \*\* ceil(log2(real\_len))

mid\_len = max\_len >> 1

n1\_add\_zero = add\_zero(n1\_str,real\_len,max\_len)

n2\_add\_zero = add\_zero(n2\_str,real\_len,max\_len)

a1 = int(n1\_add\_zero[:mid\_len])

a2 = int(n1\_add\_zero[mid\_len:])

b1 = int(n2\_add\_zero[:mid\_len])

b2 = int(n2\_add\_zero[mid\_len:])

u = kara(a1,b1)

v = kara(a1-a2,b2-b1)

w = kara(a2,b2)

return u \* 10 \*\* max\_len +(u + v + w) \* 10 \*\* mid\_len + w

2. 最小值问题：

def n\_min(a\_list,low,high):

if low >=high:

return a\_list[low]

mid = (low + high)//2

left\_min = n\_min(a\_list,low,mid)

right\_min = n\_min(a\_list,mid+1,high)

if left\_min < right\_min:

return left\_min

else:

return right\_min

3. 棋盘覆盖问题：

def chess(tr,tc,pr,pc,size):#tr,tc：棋盘左上角的位置，即棋盘位置。pr,pc:特殊方格的位置，size为棋盘大小。

global mark

global table

if size==1:

return

mark = mark + 1

count = mark

half=size//2

if pr<tr+half and pc<tc+half:

chess(tr,tc,pr,pc,half)

else:

table[tr+half-1][tc+half-1]=count

chess(tr,tc,tr+half-1,tc+half-1,half)

if pr<tr+half and pc>=tc+half:

chess(tr,tc+half,pr,pc,half)

else:

table[tr+half-1][tc+half]=count

chess(tr,tc+half,tr+half-1,tc+half,half)

if pr>=tr+half and pc<tc+half:

chess(tr+half,tc,pr,pc,half)

else:

table[tr+half][tc+half-1]=count

chess(tr+half,tc,tr+half,tc+half-1,half)

if pr>=tr+half and pc>=tc+half:

chess(tr+half,tc+half,pr,pc,half)

else:

table[tr+half][tc+half]=count

chess(tr+half,tc+half,tr+half,tc+half,half)

4. 幂乘问题：

def power(a,n):

if a == 1:

return 1

if a == 0:

return 0

if n == 1:

return a

if n % 2 == 0:

b = power(a,n//2)

return b\*b

else:

b = power(a,(n-1)//2)

return b \* b \* a

## **实验4动态规划算法实践**

**一、 实验目的**

1. 了解动态规划算法思想；

2. 掌握算法的基本要素及解题步骤；

3. 能够针对实际问题，能够按照动态规划解题步骤，分析问题；

4. 能够正确的编码、实现动态规划算法；

5. 能够正确分析算法的时间复杂度和空间复杂度。

**二、 实验环境**

1. Windows操作系统或Linux操作系统；

2. Python3.x；

3. PyCharm或Sublime或Jupyter Notebook。

**三、 实验内容(任选一个)**

1. 矩阵连乘问题；

2. 最长公共子序列问题；

3. 0-1背包问题；

4. 最优二叉搜索树。

**四、 实验步骤**

1. 认真审阅题目，明确题目的已知条件和求解的目标；

2. 问题建模；

3. 算法设计；

4. 编码实现(用Python语言)；

5. 测试数据；

6. 程序运行结果；

7. 分析实验结果是否符合预期，如果不符合，分析可能的原因；

8. 算法分析；

9. 实验总结： 总结实验中遇到的问题及解决方法。

**五、 实验报告**

1. 封皮

2. 正文：

(1) 实验目的；

(2) 实验平台；

(3) 实验内容；

(4) 算法设计(问题分析、建模、算法描述)；

(5) 算法源码；

(6) 测试数据；

(7) 程序运行结果(要求： 截图说明算法运行的结果)；

(8) 算法分析(分析算法的时间复杂度和空间复杂度)；

(9) 实验总结(实验中遇到的问题、解决方法、实验收获等)。

**六、参考代码**

1. 矩阵连乘问题：

import numpy as np

#求最优值 并记录相关信息

def MatrixChain(p,n):

# 存储最优值

m=np.zeros((n+1,n+1))#牺牲第0行和第0列

#存储最优决策

s=np.zeros((n+1,n+1))#牺牲第0行和第0列

#单个矩阵连乘的次数

for i in range(n+1):

m[i][i]=0

s[i][i]=0

#r 表示子问题的规模，即连乘的矩阵个数，从两个矩阵开始，规模逐步放大到n

for r in range(2,n+1):

#控制规模为r的子问题个数，共n-r+1个,0...

for i in range(1,n-r+2):#range产生的范围是左闭右开的区间

j=i+r-1#当前子问题为Ai...Aj

#计算第一种决策对应的乘法次数，从Ai处分开的决策

m[i][j]=m[i+1][j]+p[i-1]\*p[i]\*p[j]#m的下标从0开始

s[i][j]=i#记录当前决策i+1

#计算i+1处分开的决策、i+2处分开的决策，..,一直到j-1处的决策的乘法次数，取最小值及对应的决策

for k in range(i+1,j):

t=m[i][k]+m[k+1][j]+p[i-1]\*p[k]\*p[j]

if t<m[i][j]:

m[i][j]=t

s[i][j]=k

return m,s

#res = []

def Traceback(i,j,s):

global res

if i==j:

res.append('A'+str(i))

else:

res.append('(')

Traceback(i, int(s[i][j]),s)

Traceback(int(s[i][j]+1), j,s)

res.append(')')

if \_\_name\_\_ == "\_\_main\_\_":

arr=[[3,2],[2,5],[5,10],[10,2],[2,3]]

n=len(arr)

res = []

# 处理矩阵的行和列

p=[]

for i in range(n):

if i==0:

p.append(arr[0][0])

p.append(arr[0][1])

else:

p.append(arr[i][1])

m,s = MatrixChain(p,n)

Traceback(1,n,s)

print(''.join(res))

2. 最长公共子序列问题

def LCS(A,B):

n = len(A)

m = len(B)

A.insert(0,'0')

B.insert(0,'0')

# 二维表c存放公共子序列的长度

c = [ ([0]\*(m+1)) for i in range(n+1) ]

# 二维表s存放公共子序列的长度步进

b = [ ([0]\*(m+1)) for i in range(n+1) ]

for i in range (0,n+1):

for j in range (0,m+1):

if (i==0 or j==0):

c[i][j] = 0

elif A[i] == B[j]:

c[i][j] = ( c[i-1][j-1] + 1 )

b[i][j] = 0

elif c[i-1][j] >= c[i][j-1]:

c[i][j] = c[i-1][j]

b[i][j] = 1

else:

c[i][j] = c[i][j-1]

b[i][j] = -1

return c,b

def printLCS(s,A,i,j):

global res

if ( i == 0 or j == 0):

return 0

if s[i][j] == 0:

printLCS(s,A,i-1,j-1)

res.append(A[i])

elif s[i][j] == 1:

printLCS(s,A,i-1,j)

else:

printLCS(s,A,i,j-1)

if \_\_name\_\_=="\_\_main\_\_":

A = ['z', 'x', 'y', 'x', 'y', 'z']

B = ['x', 'y', 'y', 'z', 'x']

res = []

n = len(A)

m = len(B)

c,s = LCS(A,B)

printLCS(s,A,n,m)

print(res)

3. 0-1背包问题：

import numpy as np

def knapsack(w,v,W):#return max v

w.insert(0,0)#前0件要用

v.insert(0,0)#前0件要用

n = len(w)

c=np.zeros((n,W+1),dtype=np.int32)#下标从零开始

for i in range(1,n):

for j in range(1,W+1):

if w[i]<=j:

c[i][j]=max(c[i-1][j-w[i]]+v[i],c[i-1][j])

else:

c[i][j]=c[i-1][j]

x = [0 for i in range(n)]

j = W

for i in range(n-1, 0, -1):

if c[i][j] > c[i - 1][j]:

x[i - 1] = 1

j -= w[i - 1]

return c[n-1][W],x

if \_\_name\_\_ =="\_\_main\_\_":

w =[2,2,6,5,4]

v = [3,6,5,4,6]

w\_most = 10

bestp,x =knapsack(w,v,w\_most)

print('最大价值为：',bestp)

print('背包中所装物品为:',x)

4. 最优二叉搜索树：

def optimal\_bst(p,q,n):

c=[[0 for j in range(n+1)]for i in range(n+2)]

w=[[0 for j in range(n+1)]for i in range(n+2)]

root=[[0 for j in range(n+1)]for i in range(n+1)]

for i in range(n+2):

c[i][i-1]=q[i-1]

w[i][i-1]=q[i-1]

for l in range(1,n+1):

for i in range(1,n-l+2):

j=i+l-1

c[i][j]=float("inf")

w[i][j]=w[i][j-1]+p[j]+q[j]

for r in range(i,j+1):

t=c[i][r-1]+c[r+1][j]+w[i][j]

if t<c[i][j]:

c[i][j]=t

root[i][j]=r

return c,root

def BestSolution(root,i,j,s):

if(i<=j):

print("s"+str(i)+":s"+str(j)+"的根为："+s[root[i][j]-1])

BestSolution(root,i, root[i][j]-1,s)

BestSolution(root,root[i][j]+1,j,s)

if \_\_name\_\_=="\_\_main\_\_":

p=[0,0.15,0.1,0.05,0.1,0.2]

q=[0.05,0.1,0.05,0.05,0.05,0.1]

s="abcde"

n=len(s)

c,root=optimal\_bst(p,q,n)

BestSolution(root,1,n,s)

## **实验5回溯法实践**

**一、 实验目的**

1. 了解回溯算法思想；

2. 掌握回溯法的算法框架；

3. 能够针对实际问题，按照回溯法算法框架，分析问题的解空间、解空间的组织结构、搜索的约束条件、限界条件；

4. 能够正确编码；

5. 能够正确分析算法的时间复杂度和空间复杂度。

**二、 实验环境**

1. Windows操作系统或Linux操作系统；

2. Python3.x；

3. PyCharm或Sublime或Jupyter Notebook。

**三、 实验内容(任选一个)**

1. 最大团问题；

2. n皇后问题；

3. 最小重量机器设计问题。

**四、 实验步骤**

1. 认真审阅题目，明确题目的已知条件和求解的目标；

2. 问题建模；

3. 算法设计；

4. 编码实现(用Python语言)；

5. 测试数据；

6. 程序运行结果；

7. 分析实验结果是否符合预期，如果不符合，分析可能的原因；

8. 算法分析；

9. 实验总结： 总结实验中遇到的问题及解决方法。

**五、 实验报告**

1. 封皮。

2. 正文：

(1) 实验目的；

(2) 实验平台；

(3) 实验内容；

(4) 算法设计(问题分析、建模、算法描述)；

(5) 算法源码；

(6) 测试数据；

(7) 程序运行结果(要求： 截图说明算法运行的结果)；

(8) 算法分析(分析算法的时间复杂度和空间复杂度)；

(9) 实验总结(实验中遇到的问题、解决方法、实验收获等)。

六、参考代码

1. 最大团问题：

def place(t):

global x

global a

OK=True

for j in range(t):

if x[j] and a[t][j]==0:

OK=False

break

return OK

def backtrack(t):

global cn,bestn,n,bestx,x

if (t > n):

bestx = x[:]

bestn=cn

return

if place(t-1):

x[t-1]=1

cn += 1

backtrack(t+1)

cn -= 1

if (cn+n-t>bestn):

x[t-1]=0

backtrack(t+1)

if \_\_name\_\_ =="\_\_main\_\_":

a = [[0,1,1,0,0],[1,0,1,1,1],[1,1,0,1,1],[0,1,1,0,1],[0,1,1,1,0]]

n = len(a)

x=[i for i in range(n)]

bestx =None

bestn = cn = 0

backtrack(1)

print("最大团顶点个数：", bestn)

print("最大团为：", bestx)

2. n皇后问题：基于满n叉树

def Ok(k):

global x,place\_answers

for j in range(1,k):

if (x[k] == x[j] or (abs(k-j) ==abs(x[k]-x[j]))):

return False

return True

def backtrack(t):

global x,n,sum1

if (t>n):

sum1 += 1

place\_answers.append(x[:])

else:

for i in range(1,n+1):

x[t]=i

if (Ok(t)):

backtrack(t+1)

if \_\_name\_\_ =="\_\_main\_\_":

n = 4

place\_answers = []

sum1 = 0

x = [0 for i in range(n+1)]

backtrack(1)

for i in range(len(place\_answers)):

print(place\_answers [i])

print("共有："+str(sum1)+"种放置方案")

3. 最小重量机器设计问题：

def Backtrack(t):

global bestw,cw,bestx,x,COST,cc,w,c,Total\_cost

if(t>n):

Total\_cost = cc

bestw = cw

bestx = x[:]

return

for j in range(1,m+1):

x[t]=j

if(cc+c[t][j]<=COST and cw+w[t][j]<bestw):

cc+=c[t][j]

cw+=w[t][j]

Backtrack(t+1)

cc-=c[t][j]

cw-=w[t][j]

if \_\_name\_\_ =="\_\_main\_\_":

import sys

COST = 7

w = [[0,0,0,0],[0,1,2,3],[0,3,2,1],[0,2,3,2]]

c = [[0,0,0,0],[0,1,2,3],[0,5,4,2],[0,3,1,2]]

n = 3

bestw = sys.maxsize

cw = 0

cc = 0

Total\_cost = 0

m = 3

bestx = [0 for i in range(n+1)]

x = [0 for i in range(n+1)]

Backtrack(1)

print("设计方案为：" ,bestx)

print("最小重量为：" ,bestw)

print("所需费用为：" ,Total\_cost)

## **实验6分支限界算法实践**

**一、 实验目的**

1. 了解分支限界法思想和类型；

2. 掌握使用分支限界法求解问题的一般步骤；

3. 能够针对实际问题，分析正确问题的解空间、解空间的组织结构、搜索的约束条件、限界条件，优先级；

4. 能够正确使用队列、优先队列正确编码；

5. 能够正确分析算法的时间复杂度和空间复杂度。

**二、 实验环境**

1. Windows操作系统或Linux操作系统；

2. Python3.x；

3. PyCharm或Sublime或Jupyter Notebook。

**三、 实验内容(任选一个)**

1. 运动员最佳配对问题；

2. 旅行售货员问题；

3. 布线问题。

**四、 实验步骤**

1. 认真审阅题目，明确题目的已知条件和求解的目标；

2. 问题建模；

3. 算法设计；

4. 编码实现(用Python语言)；

5. 测试数据；

6. 程序运行结果；

7. 分析实验结果是否符合预期，如果不符合，分析可能的原因；

8. 算法分析；

9. 实验总结： 总结实验中遇到的问题及解决方法。

**五、 实验报告**

1. 封皮。

2. 正文：

(1) 实验目的；

(2) 实验平台；

(3) 实验内容；

(4) 算法设计(问题分析、建模、算法描述)；

(5) 算法源码；

(6) 测试数据；

(7) 程序运行结果(要求： 截图说明算法运行的结果)；

(8) 算法分析(分析算法的时间复杂度和空间复杂度)；

(9) 实验总结(实验中遇到的问题、解决方法、实验收获等)。

六、参考代码

1. 运动员最佳配对问题：（优先队列式分支限界法）

import heapq

class node():

def \_\_init\_\_(self,level,Sum,r,up,x):

self.level = level

self.Sum = Sum

self.r = r

self.up = up

self.x = x

def \_\_lt\_\_(self,other):

return self.up < other.up

def solve():

global bestx,bestv

heap = []

new\_node = node(1,0,0,0,[i for i in range(n+1)])

new\_node.r = sum(maxout)

new\_node.up =new\_node.r

for i in range(1,n+1):

new\_node.x[i] = i

heapq.heappush(heap,new\_node)

while ( len(heap) > 0 ):

current\_node = heapq.heappop(heap)

if current\_node.level > n:

if(current\_node.Sum > bestv):

bestx [:]= current\_node.x[:]

bestv = current\_node.Sum

else:

for i in range(current\_node.level,n+1):

level = current\_node.level

Sum = current\_node.Sum

r = current\_node.r

up =current\_node.up

current\_node.x[level],current\_node.x[i] =current\_node.x[i],current\_node.x[level]

Sum =Sum + p[level][current\_node.x[level]] \* q[current\_node.x[level]][level]

r = r - maxout[level]

up = Sum + r

if up > bestv:

heapq.heappush(heap,node(level + 1,Sum,r,up,current\_node.x[:]))

current\_node.x[level],current\_node.x[i] =current\_node.x[i],current\_node.x[level]

return bestx,bestv

if \_\_name\_\_=="\_\_main\_\_":

n = 3

p=[[0,0,0,0],[0,10,2,3],[0,2,3,4],[0,3,4,5]]

q=[[0,0,0,0],[0,2,2,2],[0,3,5,3],[0,4,5,1]]

bestx = [0 for i in range(n+1)]#记录最优解

bestv = 0

maxout = [0 for i in range(n+1)]#记录最优值

for i in range(1,n+1):

temp = 0

for j in range(1,n+1):

temp = max(temp,q[i][j] \* p[j][i])

maxout[i] = temp

bestx,bestv = solve()

print("最优解为:",bestx[1:])

print("最优值为:",bestv)

2. 旅行售货员问题：（队列式分支限界法）

import math

import queue

class Node:

def \_\_init\_\_(self,cl,level,x):#cl:当前路径长度，level：当前节点层次，g\_n：问题规模

self.cl = cl#当前路径长度

self.level = level#节点的层次

self.x = x#部分解

def traveling(a,start,g\_n):

que =queue.Queue()

node = Node(0,2,[i for i in range(g\_n+1)])#

que.put(node)

bestx =None#最大价值

bestl = NoEdge

while(not que.empty()):

current\_node = que.get()

level = current\_node.level

cl = current\_node.cl

if level == g\_n:#叶子表示找到了一个比当前解更好的一个解，记录之

if (a[current\_node.x[g\_n-1]][current\_node.x[g\_n]] != NoEdge and a[current\_node.x[g\_n]][1] != NoEdge and (cl + a[current\_node.x[g\_n-1]][current\_node.x[g\_n]] + a[current\_node.x[g\_n]][1] < bestl or bestl == NoEdge)):

bestx = current\_node.x[:]

bestl = cl + a[current\_node.x[g\_n-1]][current\_node.x[g\_n]] + a[current\_node.x[g\_n]][1]

else:

for j in range(level,g\_n+1):

if (a[current\_node.x[level-1]][current\_node.x[j]] != NoEdge and (cl < bestl or bestl == NoEdge)):

current\_node.x[level], current\_node.x[j] = current\_node.x[j], current\_node.x[level]

que.put(Node(cl + a[current\_node.x[level-1]][current\_node.x[level]],level+1,current\_node.x[:]))

current\_node.x[level], current\_node.x[j] = current\_node.x[j], current\_node.x[level]

return bestx,bestl

if \_\_name\_\_ == '\_\_main\_\_':

import sys

NoEdge = sys.maxsize

#a = [[NoEdge,NoEdge,NoEdge,NoEdge,NoEdge],[NoEdge,NoEdge,30,50,4],[NoEdge,30,NoEdge,5,15],[NoEdge,50,5,NoEdge,3],[NoEdge,4,15,3,NoEdge]]

a = [[NoEdge,NoEdge,NoEdge,NoEdge,NoEdge,NoEdge],[NoEdge,NoEdge,10,NoEdge,4,12],[NoEdge,10,NoEdge,15,8,5],[NoEdge,NoEdge,15,NoEdge,7,30],[NoEdge,4,8,7,NoEdge,6],[NoEdge,12,5,30,6,NoEdge]]

g\_n = len(a) - 1

bestx,bestl = traveling(a,1,g\_n)

print("最短路径长度为：", bestl)

print("最优旅行路线为：", bestx)

3. 布线问题：

#队列式分支限界法

import queue

class Node:

def \_\_init\_\_(self,row,col):

self.row = row

self.col = col

def findpath(start,finish):

global grid,offset

pathLen = 0

if(start.row==finish.row) and (start.col==finish.col):#起点与终点相同，不用布线

pathLen = 0

# return

here = start

grid[start.row][start.col]=0

que = queue.Queue()

que.put(start)

while(True):

here = que.get()

for i in range(4):#沿着扩展结点的右、下、左、上四个方向扩展

nbr = Node(here.row + offset[i].row,here.col + offset[i].col)

#print("nbr.row = "+str(nbr.row))

#print("nbr.col =" +str(nbr.col))

if(grid[nbr.row][nbr.col] == -1):#如果这个方格还没有扩展

grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1

if((nbr.row == finish.row) and (nbr.col==finish.col)):

break #如果到达终点结束

que.put(nbr)#此邻结点放入队列

if((nbr.row==finish.row) and (nbr.col==finish.col)):

break #完成布线

if que.empty():

return

def build\_path(grid,start,finish):

global offset

pathlen=grid[finish.row][finish.col]

path = []

here=finish

for j in range(pathlen-1,-1,-1):

path.insert(0,here)

for i in range(4):#四个方向扩展

nbr = Node(here.row+offset[i].row,here.col+offset[i].col)

if (grid[nbr.row][nbr.col]==j):

break

here=nbr #往回推进

path.insert(0,start)

return path ,pathlen

## **实验7线性规划与网络流算法实践**

一、 实验目的

1. 了解线性规划问题的描述方式；

2. 掌握一般线性规划问题转化为约束标准型的方法；

3. 掌握求解约束标准型线性规划问题的单纯形算法；

4. 掌握求最大网络流的增广路算法；

5. 掌握最小费用最大流的消圈算法。

二、 实验环境

1. Windows操作系统或Linux操作系统；

2. Python3.x；

3. PyCharm或Sublime或Jupyter Notebook。

三、 实验内容(任选一个)

1. 方格取数问题： 在一个有m×n个方格的棋盘中，每个方格中有一个正整数。现要从方格中取数，使任意 2 个数所在方格没有公共边，且取出的数的总和最大。试设计一个满足要求的取数算法。对于给定的方格棋盘，按照取数要求编程找出总和最大的数。

输入：第 1 行有 2 个正整数 m 和 n，分别表示棋盘的行数和列数。接下来的 m 行，每行有 n 个正整数，表示棋盘方格中的数。

输入样例：

3 3

1 2 3

3 2 3

2 3 1

输出：程序运行结束时，将取数的最大总和输出

输出样例：

11

2. 飞行员配对方案问题： 英国皇家空军从沦陷国征募了大量外籍飞行员。由皇家空军派出的每一架飞机都需要配备在航行技能和语言上能互相配合的2 名飞行员，其中1 名是英国飞行员，另1名是外籍飞行员。在众多的飞行员中，每一名外籍飞行员都可以与其他若干名英国飞行员很好地配合。如何选择配对飞行的飞行员才能使一次派出最多的飞机。对于给定的外籍飞行员与英国飞行员的配合情况，试设计一个算法找出最佳飞行员配对方案，使皇家空军一次能派出最多的飞机。对于给定的外籍飞行员与英国飞行员的配合情况，编程找出一个最佳飞行员配对方案，使皇家空军一次能派出最多的飞机。

输入：

第 1 行有 2 个正整数 m 和 n。n 是皇家空军的飞行员总数(n<100)； m 是外籍飞行员数(m<=n)。外籍飞行员编号为 1~m； 英国飞行员编号为 m+1~n。

接下来每行有 2 个正整数 i 和 j，表示外籍飞行员 i 可以和英国飞行员 j 配合。最后以 2个-1 结束。

输入样例：

5 10

1 7

1 8

2 6

2 9

2 10

3 7

3 8

4 7

4 8

5 10

-1 -1

输出：

第 1 行是最佳飞行员配对方案一次能派出的最多的飞机数 M。接下来 M 行是最佳飞行员配对方案。每行有 2个正整数 i 和 j，表示在最佳飞行员配对方案中，飞行员 i 和飞行员 j 配对。如果所求的最佳飞行员配对方案不存在，则输出‘No Solution!’。

输出样例：

4

1 7

2 9

3 8

5 10

四、 实验步骤

1. 认真审阅题目，明确题目的已知条件和求解的目标；

2. 算法设计；

3. 编码实现(用Python语言)；

4. 测试数据；

5. 程序运行结果；

6. 分析实验结果是否符合预期，如果不符合，分析可能的原因；

7. 实验总结： 总结实验中遇到的问题及解决方法。

五、 实验报告

1. 封皮。

2. 正文：

(1) 实验目的；

(2) 实验平台；

(3) 实验内容

(4) 算法设计(问题分析、建模、算法描述)；

(5) 算法源码；

(6) 测试数据；

(7) 程序运行结果(要求： 截图说明算法运行的结果)；

(8) 实验总结(实验中遇到的问题、解决方法、实验收获等)；

六、参考代码

1. 方格取数问题：#方格取数问题

#（1）把相邻方格用黑白两种颜色染色

#（2）每个方格抽象成顶点，黑色顶点构成集合A，白色顶点构成集合B

#（3）相邻方格的顶点连接一条边，有A中顶点指向B中顶点，容量为1。

#（4）添加源点0,并从0号点向集合A中每一个顶点引一条边，容量为1

#（5）添加汇点，编号为m\*n+1，集合B中的每个顶点向汇点引一条有向边，容量为1

#（6）求该网络的最大流

#（7）根据最大流求最大总和

#增广路算法

import numpy as np

import queue

def maxFlow(ss, tt):

global maxflow,flow,s,t,k#流矩阵和网络中顶点个数

s = ss

t=tt

flow = np.zeros((k,k),dtype = int)#初始可行流为0流

maxflow = 0

while(pfs()):#是否存在可增广

augment(s, t)#沿增广路增流

def pfs():

global father,cap,k,s,t

MyQ = queue.Queue()#存储已标号未检查的点

father = [-1 for i in range(k)]

while(not MyQ.empty()):

MyQ.get()#队列清空

father[s] = s

#print("father="+str(father))

MyQ.put(s)

while(not MyQ.empty()): #找可增广路

v=MyQ.get()

for j in range(k):#对与v连接的点j标号

if(cap[v][j]>0 and father[j]<0):

father[j]=v

if(j==t):

return True

MyQ.put(j)

return False

def augment(ss, tt):

global maxflow,flow,father,cap

v=tt

w=father[v]

d = 99999

while(True):

if(d > cap[w][v]):

d=cap[w][v]#确定增流量

v=w

w=father[v]

if( v == ss):

break

v=tt

w=father[v]

#沿增广路增流

while(True):

flow[w][v] += d

flow[v][w] -= d

cap[w][v] -= d

cap[v][w] += d

v=w

w=father[v]

if (v == ss):

break

maxflow += d

if \_\_name\_\_ =="\_\_main\_\_":

cap = [[0,1,0,1,0,1,0,1,0,1,0],[0,0,1,0,1,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,1],[0,0,1,0,0,0,1,0,0,0,0],

[0,0,0,0,0,0,0,0,0,0,1],[0,0,1,0,1,0,1,0,1,0,0],[0,0,0,0,0,0,0,0,0,0,1],[0,0,0,0,1,0,0,0,1,0,0],

[0,0,0,0,0,0,0,0,0,0,1],[0,0,0,0,0,0,0,1,0,1,0],[0,0,0,0,0,0,0,0,0,0,0]]

m = 3#方格行数

n = 3#方格列数

square = [[1,2,3],[3,2,3],[2,3,1]]#方格中的数据

k = m\*n+2#网络的顶点个数

father = np.zeros(k,dtype=int)

flow = np.zeros((k,k),dtype=int)#流量

s = 0#源点

t = 10#汇点

maxflow = 0#最大流量

maxFlow(s,t)

sum\_square = 0#最大总和

for i in range(1,k,2):

for j in range(2,k,2):

if flow[i][j] == 1:

sum\_square += square[(j-1)//n][(j-1)%n]

print("最大总和为：", sum\_square)

（2）飞行员配对方案问题：

# 飞行员配对问题

#（1）建立源点,编号为0。

#（2）建立集合A，A中的结点为外籍飞行员。

#（3）建立集合B，B中的结点为英国飞行员。

#（4）建立汇点，编号11。

#（5）源点0向集合A中的每一个结点引一条有向边，边的容量为1。

#（6）根据外籍飞行员 i 可以和英国飞行员 j 配合的情况，将集合A中的结点i向集合B中结点j引一条有向边，边的容量为1。

#（7）集合B中的每一个结点向汇点引一条有向边，边的容量为1。

#（8）从0流出发，求该网络的最大流，便能求出配对方案。

#增广路算法

import numpy as np

import queue

def maxFlow(ss, tt):

global maxflow,flow,n,s,t#流矩阵和网络中顶点个数

s = ss

t=tt

flow = np.zeros((n,n),dtype = int)#初始可行流为0流

maxflow = 0

while(pfs()):#是否存在可增广

augment(s, t)#沿增广路增流

def pfs():

global father,cap,n,s,t

MyQ = queue.Queue()#存储已标号未检查的点

father = [-1 for i in range(n)]

while(not MyQ.empty()):

MyQ.get()#队列清空

father[s] = s

#print("father="+str(father))

MyQ.put(s)

while(not MyQ.empty()): #找可增广路

v=MyQ.get()

for j in range(n):#对与v连接的点j标号

if(cap[v][j]>0 and father[j]<0):

father[j]=v

if(j==t):

return True

MyQ.put(j)

return False

def augment(ss, tt):

global maxflow,flow,father,cap

v=tt

w=father[v]

d = 99999

while(True):

if(d > cap[w][v]):

d=cap[w][v]#确定增流量

v=w

w=father[v]

if( v == ss):

break

v=tt

w=father[v]

#沿增广路增流

while(True):

flow[w][v] += d

flow[v][w] -= d

cap[w][v] -= d

cap[v][w] += d

v=w

w=father[v]

if (v == ss):

break

maxflow += d

if \_\_name\_\_ =="\_\_main\_\_":

cap = [[0,1,1,1,1,1,0,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0],[0,0,0,0,0,0,1,0,0,1,1,0],[0,0,0,0,0,0,0,1,1,0,0,0],

[0,0,0,0,0,0,0,1,1,0,0,0],[0,0,0,0,0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,0,0,0,0,1],[0,0,0,0,0,0,0,0,0,0,0,1],

[0,0,0,0,0,0,0,0,0,0,0,1],[0,0,0,0,0,0,0,0,0,0,0,1],[0,0,0,0,0,0,0,0,0,0,0,1],[0,0,0,0,0,0,0,0,0,0,0,0]]

n = len(cap)

father = np.zeros(n,dtype=int)

flow = np.zeros((n,n),dtype=int)

s = 0

t = 11

maxflow = 0

maxFlow(s,t)

print("最大流量为：", maxflow)

print("最大网络流为：", flow)

## **实验8随机化算法实践**

**一、 实验目的**

1. 了解随机化算法的类型及特征；

2. 掌握随机数发生器的线性同余法；

3. 掌握数值随机化算法随机实验的方法，估计值的精确度随算法的运行时间的增加而提高；

4. 掌握提高蒙特卡罗算法得到解为正确解的概率的方法、拉斯维加斯算法得到解的概率的方法；

5. 掌握舍伍德算法在确定性算法中引入的随机成分。

**二、 实验环境**

1. Windows操作系统或Linux操作系统；

2. Python3.x；

3. PyCharm或Sublime或Jupyter Notebook。

**三、 实验内容(任选一个)**

1. 估算pi的值、估算定积分；

2. 主元素问题、素数的测试；

3. 整数因子分解问题、n皇后问题；

4. 随机选择第k小元素、随机快速排序。

**四、 实验步骤**

1. 认真审阅题目，明确题目的已知条件和求解的目标；

2. 算法设计；

3. 编码实现(用Python语言)；

4. 测试数据；

5. 程序运行结果；

6. 分析实验结果是否符合预期，如果不符合，分析可能的原因；

7. 实验总结： 总结实验中遇到的问题及解决方法。

**五、 实验报告**

1. 封皮。

2. 正文：

(1) 实验目的；

(2) 实验平台；

(3) 实验内容；

(4) 算法设计(问题分析、建模、算法描述)；

(5) 算法源码；

(6) 测试数据；

(7) 程序运行结果(要求： 截图说明算法运行的结果)；

(8) 实验总结(实验中遇到的问题、解决方法、实验收获等)。

**六、参考代码**

1. 估算pi的值、估算定积分：

（1）估算pi的值

import random

def Darts(n):

k = 0#记录落入四分之一园内的点数

for i in range(n):

x=random.random()#产生一个[0,1)之间的实数，赋给x

y=random.random()#产生一个[0,1)之间的实数，赋给y

if x \*\* 2 + y \*\* 2 <=1:

k +=1

return 4\*k/n

（2）估算定积分

import random

import math

def f(x):

return abs(math.sin(x))

def Definite\_integral(n):

k = 0#记录落入四分之一园内的点数

for i in range(n):

x=random.random()#产生一个[0,1)之间的实数，赋给x

y=random.random()#产生一个[0,1)之间的实数，赋给y

if y<f(x):

k +=1

return k/n

2. 主元素问题、素数的测试

（1）主元素问题

import random

import math

def majority(T): #判定主元素的蒙特卡罗算法

global p

n = len(T)

i=random.randint(0,n-1)#产生1~n之间的随机下标

x=T[i]# 随机选择元素

k=0

for j in range(n):

if T[j]==x:

k += 1

p = k/n

return p>0.5 #当 p>0.5 时，T含有主元素

def majorityMC(T,threshold):

#重复次调用算法majority

result1 = majority(T)

if result1:

return True

else:

k= int(math.ceil(math.log2(threshold)/math.log2(1-p)))

for i in range(1,k):

if (majority(T)):

return True

return False

（2）素数的测试

#试除法

import math

import random

def Prime(n):

m = math.floor(math.sqrt(n))

for i in range(2,m+1):

if n % i== 0:

return False

return True

#wilson定理

def WilsonP(n):

fac\_mod = 1

for i in range(2,n):

fac\_mod = (fac\_mod \* i) % n

if fac\_mod == n-1:

return True

else:

return False

#费马定理

def fermat\_prime(n):

power = n - 1

d = 1

a = random.randint(2,n)

while(power > 1):

if(power % 2 == 1):

d = d \* a % n

power =power//2 #整除

a = a \* a % n

if(a \* d % n == 1):

return True

else:

return False

#二次探测定理

def Secondary\_detection(n):

result = True

for x in range(2,n-1):

if(x \*\* 2 % n == 1):

result = False

break

return result

#二次探测定理+费马定理

def Miller\_Rabin1(n,k):

global a

for i in range(k):

a = random.randint(2,n-1)

result = power(n)

if not result:

return False

return True

3. 整数因子分解问题、n皇后问题：

（1）整数因子分解问题

# 整数因子分割——拉斯维加斯算法

import random

import math

#因子分割的确定性算法

def split(n):

k = math.floor(math.sqrt(n))

for i in range(2,k+1):

if (n % i == 0):

return i

return 1

#求a,b的最大公约数

def gcd(a,b):

if b == 0:

return a

else:

return gcd(b,a % b)

#因子分割的Pollard算法

def pollard(n):

x = random.randint(0,n-1)#随机整数

y = x

k = 2

i = 0

while (i<=64):#64为最大迭代次数

i += 1

x = (x \* x - 1) % n

d = gcd(y-x,n)#求n的非平凡因子xk-x

if d > 1 and d < n:

return d

if y==x:#特殊勤快处理

return n

if (i == k):

y =x

k \*= 2

（2）n皇后问题

import random

def Place(k):#判断能否在第k行放置第k个皇后(皇后从0开始编号)

global x

for j in range(k):#当前皇后和前面已经放置好的皇后是否同一斜线、是否同一列

if((abs(k - j) == abs(x[j] -x[k])) or (x[j] == x[k])):

return False

return True

def queensLV(n):

global x

k=0

count = 1#记录当前要放置的第k个皇后在第k行的有效位置

while((k < n) and (count > 0)):

count=0

y = []

for i in range(n):

x[k]=i

if(Place(k)):

y.append(i)#第k个皇后在第k行的有效位置存于y数组

count += 1

#从有效位置中随机选取一个位置放置第k个皇后

if(count>0):

x[k]=y[random.randint(0,count-1)]

k +=1

return (count>0)#count>0表示放置成功

4. 随机选择第k小元素、随机快速排序。

（1）随机选择第k小元素

import random

def RandPartition(R,left,right):# R：待查找元素序列 left：起始索引 right ：结束索引

i = left

j = right

randi = random.randint(left,right)

R[left],R[randi] = R[randi],R[left]

pivot=R[left]#用序列的第一个元素作为基准元素

while(i<j):#从序列的两端交替向中间扫描，直至i等于j为止

while(i<j and R[j]>=pivot): #pivot相当于在位置i上

j -= 1#从右向左扫描，查找第1个小于pivot的元素

if(i<j):#表示找到了小于基准元素的元素

R[i],R[j] = R[j],R[i]#交换R[i]和R[j]，交换后i执行加1操作

i +=1

while(i<j and R[i]<=pivot):#从左向右扫描，查找第1个大于pivot的元素

i += 1

if(i<j):#表示找到了大于基准元素的元素

R[i],R[j] = R[j],R[i]#交换R[i]和R[j]，交换后j执行减1操作

j -= 1

return j

# 随机选择第k小元素

def select(R,left,right,k):

if left >= right:

return R[left]

j = RandPartition(R,left,right)

count = j - left + 1

if count == k:

return R[j]

elif count < k:

return select(R,j + 1, right, k - count)

else:

return select(R,left, j, k)

（2）随机快速排序

import random

#随机选取基准元素进行划分

def RandPartition(R,left,right):# R：待排序元素 left：起始索引 right：结束索引

i = left

j = right

randi = random.randint(left,right)#随机选取一个位置

R[left],R[randi] = R[randi],R[left]#将随机选择的位置和第一个位置元素互换。

pivot=R[left]#用序列的第一个元素作为基准元素

while(i<j):#从序列的两端交替向中间扫描，直至i等于j为止

while(i<j and R[j]>=pivot): #pivot相当于在位置i上

j -= 1#从右向左扫描，查找第1个小于pivot的元素

if(i<j):#表示找到了小于基准元素的元素

R[i],R[j] = R[j],R[i]#交换R[i]和R[j]，交换后i执行加1操作

i +=1

while(i<j and R[i]<=pivot):#从左向右扫描，查找第1个大于pivot的元素

i += 1

if(i<j):#表示找到了大于基准元素的元素

R[i],R[j] = R[j],R[i]#交换R[i]和R[j]，交换后j执行减1操作

j -= 1

return j

# 快速排序函数

def quickSort(R,left,right):

if left < right:

j = RandPartition(R,left,right)

quickSort(R, left, j-1)

quickSort(R, j+1, right)