**《算法设计与分析》实验报告**

计151 郭悟清

**实验一、递归与分治**

1. 实验内容
2. 利用分治算法，编程实现循环赛日程表安排问题，并进行时间复杂性分析；

（2）利用分治算法，编程实现最近点对问题，并进行时间复杂性分析。

1. 实验目的

（1）深刻理解并掌握“分治算法”的设计思想；

（2）提高应用“分治算法”设计技能；

（3）理解这样一个观点：用递归方法编写的问题解决程序具有结构清晰，可读性强等优点，且递归算法的设计比非递归算法的设计往往要容易一些，所以当问题本身是递归定义的，或者问题所涉及到的数据结构是递归定义的，或者是问题的解决方法是递归形式的时候，往往采用递归算法来解决。

3.程序清单

（1）

# -\*- coding: utf-8 -\*-

\_\_author\_\_ = 'Victor'

import numpy as np

import time

#分治法解决循环赛日程表安排问题

def match(k):

time\_start = time.time()

for i in range(1,k):

half=2\*\*i

#左下角的子表中项为左上角子表对应项加2\*\*i

for row in range(half):

for col in range(half):

a[row+half][col]=a[row][col]+half

# 右上角的子表等于左下角子表

for row in range(half):

for col in range(half):

a[row][col+half]=a[row+half][col]

# 右下角的子表等于左上角的子表

for row in range(half):

for col in range(half):

a[row+half][col+half]=a[row][col]

time\_end = time.time()

print "当k = %d、n = %d 时，矩阵如下，运行时间为：%fs" % (k, 2\*\*k, time\_end - time\_start)

np.set\_printoptions(threshold='nan')

print a

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

for k in range(3,6):

n = 2\*\*k

a = np.zeros((n,n))

a[0][0] = 1

a[1][1] = 1

a[1][0] = 2

a[0][1] = 2

match(k)

（2）

# -\*- coding: utf-8 -\*-

\_\_author\_\_ = 'Victor'

import numpy as np

import time, random, math

import pylab as pl

#最近点对问题，分治法和蛮力法两种方法求解

#计算两点的距离

def calDis(seq):

dis=math.sqrt((seq[0][0]-seq[1][0])\*\*2+(seq[0][1]-seq[1][1])\*\*2)

return dis

#生成器：生成横跨跨两个点集的候选点

def candidateDot(u,right,dis,med\_x):

cnt=0

#遍历right（已按横坐标升序排序）。若横坐标小于med\_x-dis则进入下一次循环；若横坐标大于med\_x+dis则跳出循环；若点的纵坐标好是否落在在[u[1]-dis,u[1]+dis]，则返回这个点

for v in right:

if v[0]<med\_x-dis:

continue

if v[0]>med\_x+dis:

break

if v[1]>=u[1]-dis and v[1]<=u[1]+dis:

yield v

#求出横跨两个部分的点的最小距离

def combine(left,right,resMin,med\_x):

dis=resMin[1]

minDis=resMin[1]

pair=resMin[0]

for u in left:

if u[0]<med\_x-dis:

continue

for v in candidateDot(u,right,dis,med\_x):

dis=calDis([u,v])

if dis<minDis:

minDis=dis

pair=[u,v]

return [pair,minDis]

#分治求解

def divide(seq):

#求序列元素数量

n=len(seq)

#按点的纵坐标升序排序

seq=sorted(seq)

#递归开始进行

if n<=1:

return None,float('inf')

elif n==2:

return [seq,calDis(seq)]

else:

half=int(len(seq)/2)

med\_x=(seq[half][0]+seq[-half-1][0])/2

left=seq[:half]

resLeft=divide(left)

right=seq[half:]

resRight=divide(right)

#获取两集合中距离最短的点对

if resLeft[1]<resRight[1]:

resMin=combine(left,right,resLeft,med\_x)

else:

resMin=combine(left,right,resRight,med\_x)

pair=resMin[0]

minDis=resMin[1]

return [pair,minDis]

#蛮力法

def calDirect(seq):

minDis=float('inf')

pair=[]

for i in range(len(seq)):

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

dis=calDis([seq[i],seq[j]])

if dis <minDis:

minDis=dis

pair=[seq[i],seq[j]]

return [pair,minDis]

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

seq=[(random.uniform(0,10),random.uniform(0,10)) for x in range(30)]

for e in seq:

pl.plot(e[0], e[1], 'bo')

list = []

time\_start = time.time()

list = divide(seq)

time\_end = time.time()

print "距离最近的点对为：%s和%s，它们之间的距离是%f" % (list[0][0], list[0][1], list[1])

print "分治法求解时间：%fs" % (time\_end - time\_start)

time\_start = time.time()

list2 = calDirect(seq)

time\_end = time.time()

print "蛮力法求解的时间：%fs" % (time\_end - time\_start)

pl.plot(list[0][0][0], list[0][0][1], 'ro')

pl.plot(list[0][1][0], list[0][1][1], 'ro')

x1 = [list[0][0][0], list[0][1][0]]

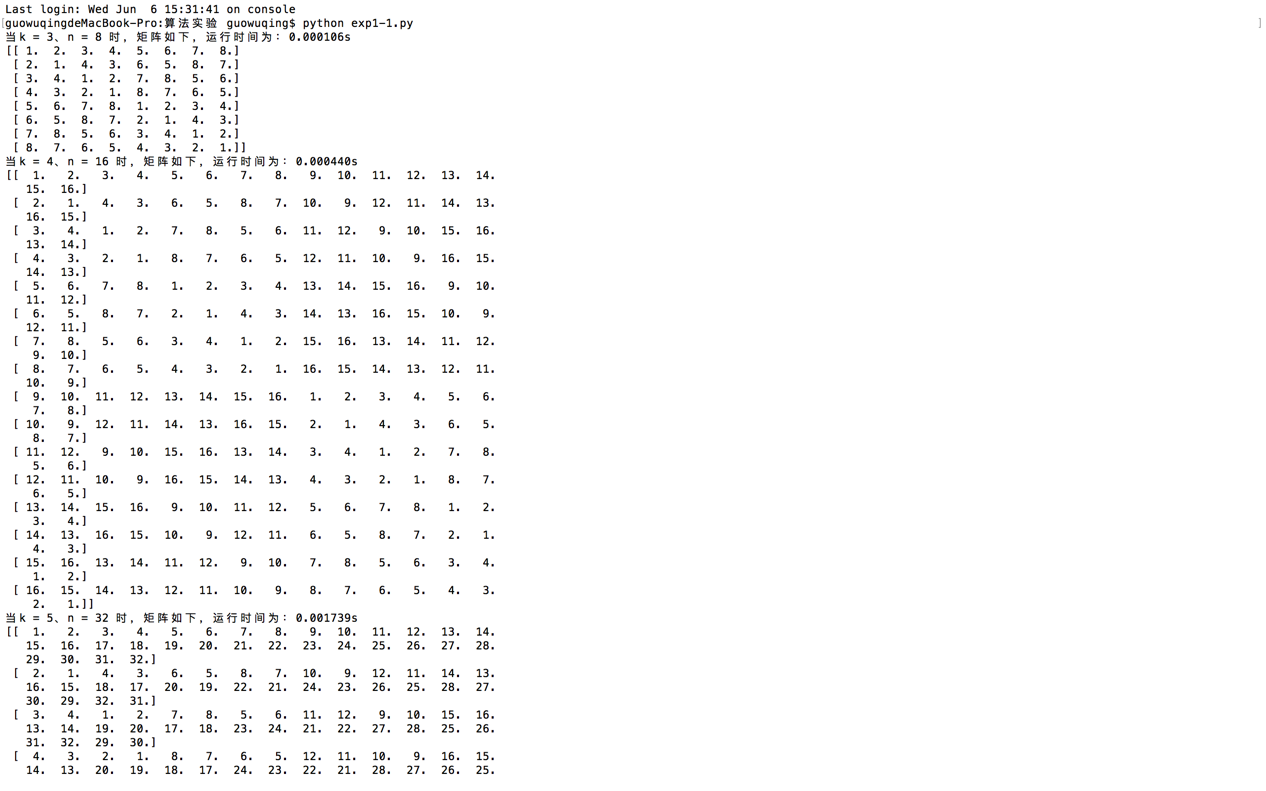
x2 = [list[0][0][1], list[0][1][1]]

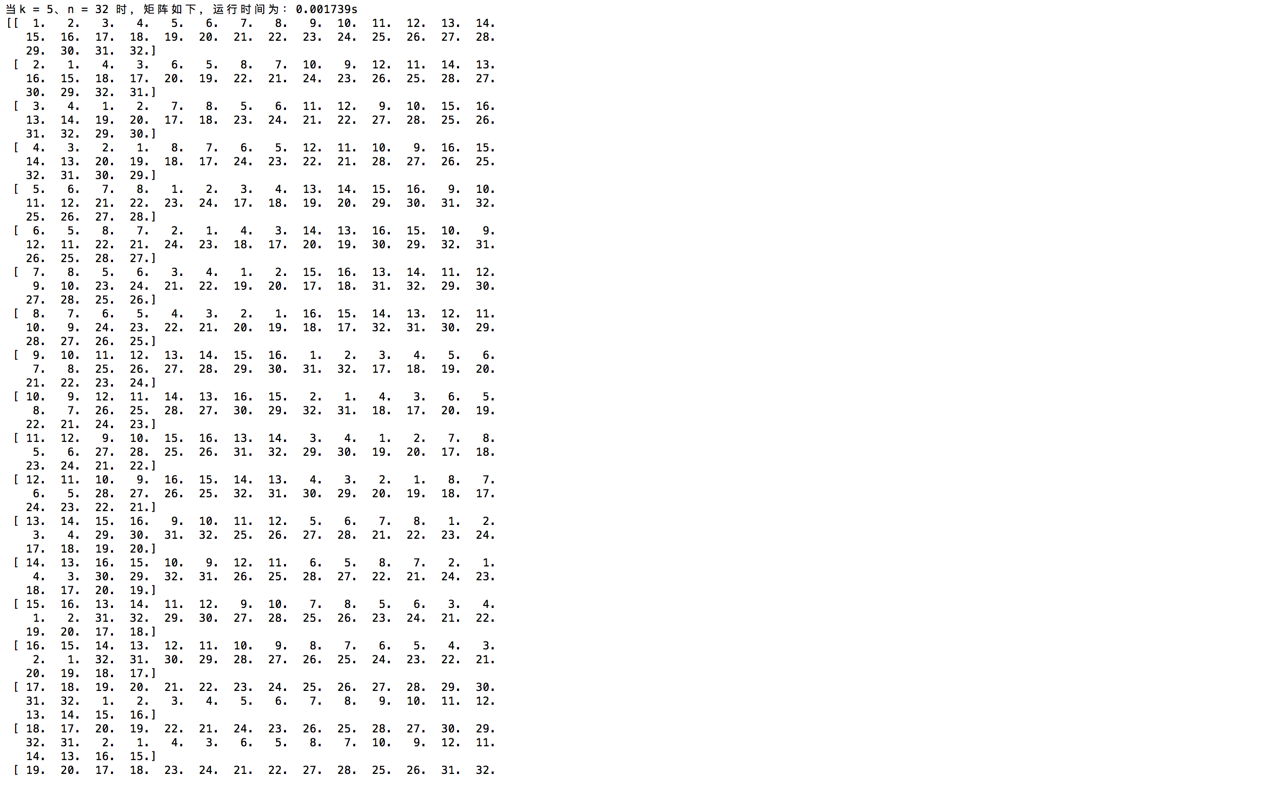
pl.plot(x1, x2, 'r')

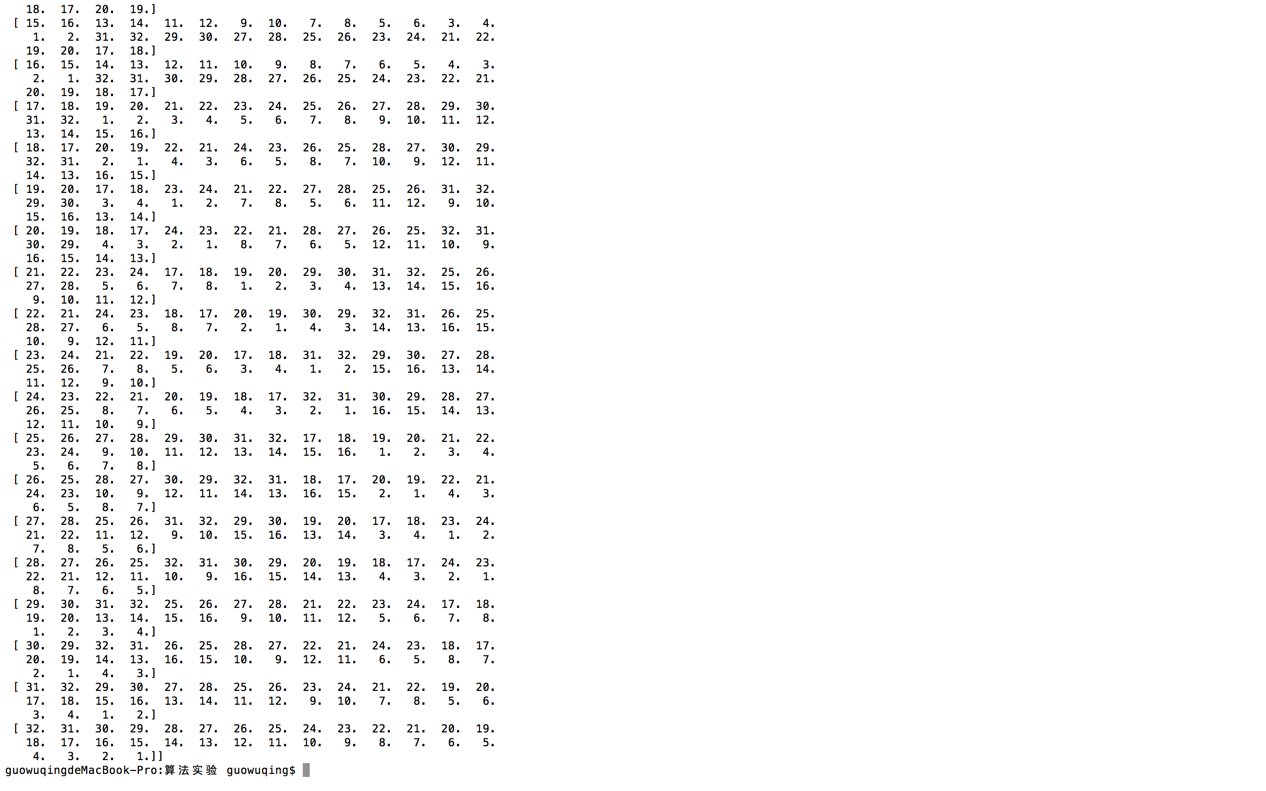
pl.show()

4.运行结果

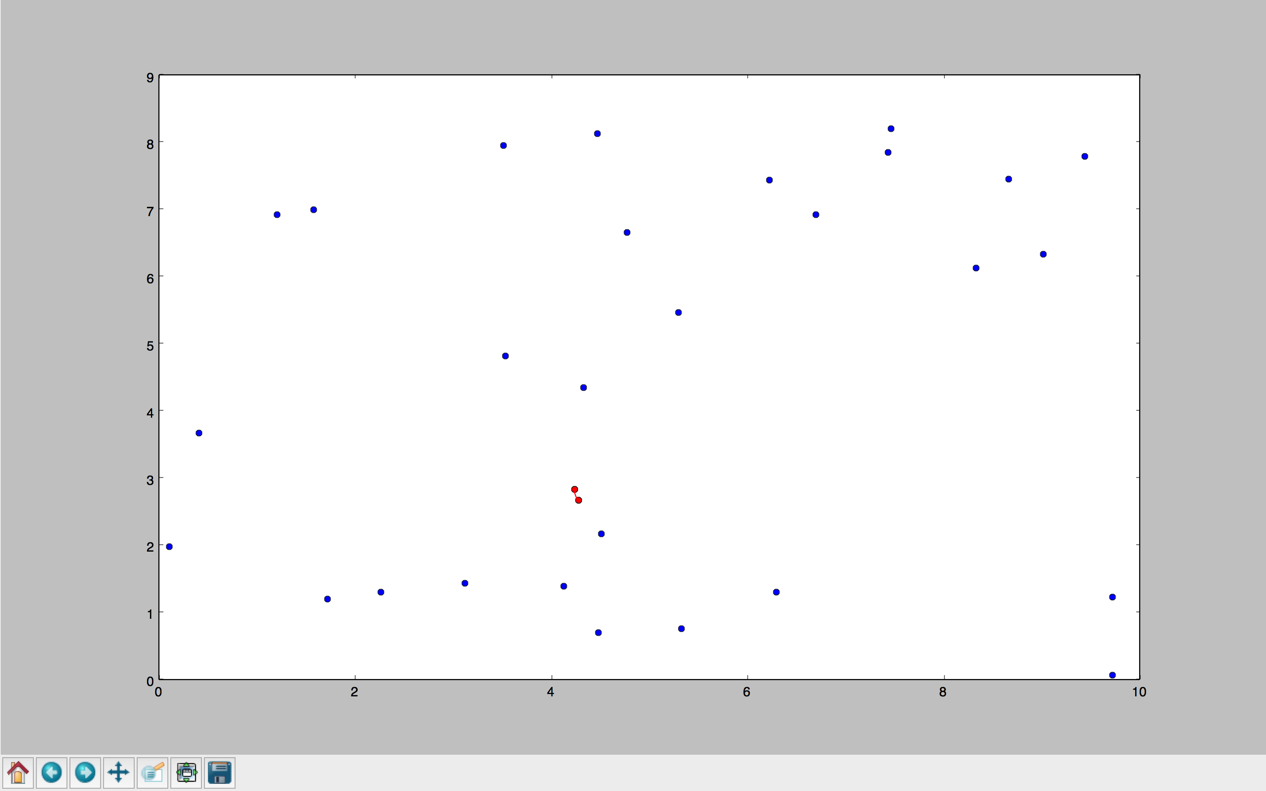
实验一截图如下：



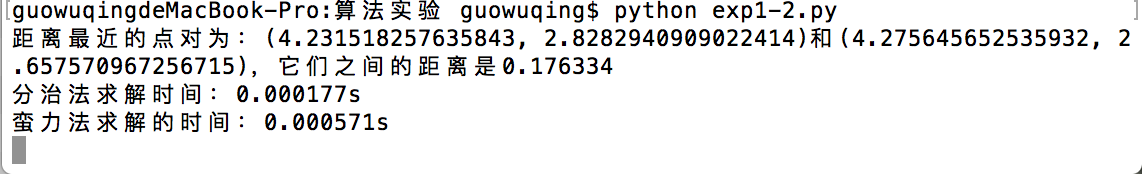




实验二截图如下：



最近点对用红色标注，其他点是蓝色，并且最近点对之间用红色线段连接。



5.分析与思考

分治法将一个难以直接解决的大问题划分成一些规模较小的子问题，分别求解各个子问题的，再合并子问题的解得到原问题的解。

在实验一中，运用分治法解决循环赛问题的时间复杂度为O（n），最近点对的时间复杂度为O（nlogn）。

**实验二 动态规划**

1.实验内容

1. 利用动态规划算法编程求解TSP问题，并进行时间复杂性分析；

（2） 利用动态规划算法编程求解0-1背包问题，并进行时间复杂性分析。

2.实验目的

1. 熟练掌握动态规划思想及教材中相关经典算法。
2. 使用动态规划法编程，求解0/1背包问题和TSP问题。
3. 程序清单

（1）

# -\*- coding: utf-8 -\*-

\_\_author\_\_ = 'Victor'

import numpy as np

from itertools import combinations

#动态规划法求解TSP问题

def Tsp(arc, n, v):

d = [[0 for col in range(2\*\*(n-1))] for row in range(n)]

#d[0] = v

for i in range(1, n):

d[i][0] = arc[i][0]

for j in range(1, 2\*\*(n-1)):

for i in range(1, n):

if i not in v[j]:

lst = []

for k in v[j]:

vj = list(v[j])

vj.remove(k)

num = v.index(vj)

sum = arc[i][k] + d[k][num]

lst.append(sum)

if len(lst) != 0:

d[i][j] = min(lst)

else:

d[i][j] = 0

lst = []

for k in v[2\*\*(n-1)-1]:

vj = list(v[2\*\*(n-1)-1])

vj.remove(k)

num = v.index(vj)

sum = arc[0][k] + d[k][num]

lst.append(sum)

d[0][2\*\*(n-1)-1] = min(lst)

print d

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

#n = int(raw\_input("请输入矩阵的n值:"))

n = 4

#arc = np.zeros((n,n))

'''

print("请输入矩阵每行的元素，以空格间隔，每行结束以回车间隔")

i = 0

while i < n:

raw = raw\_input().split(" ")

raw\_int = [int(j) for j in raw]

arc[i] = raw\_int

i += 1

'''

arc = [[0, 3, 6, 7], [5, 0, 2, 3], [6, 4, 0, 2], [3, 7, 5, 0]]

v = []

v.append([])

for i in range(1, n):

vi = [c for c in combinations(range(1, n), i)]

for j in vi:

v.append(list(j))

#v.append([c for c in combinations(range(1, n), i)])

Tsp(arc, n, v)

（2）

# -\*- coding: utf-8 -\*-

\_\_author\_\_ = 'Victor'

import numpy as np

#动态规划法求解 0/1 背包问题

def KnapSack(w, v, n, c):

i = 0

V = np.zeros((n+1, c+1))

for j in range(c+1):

V[0][j] = 0

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

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

V[i][j] = V[i-1][j]

if j>= w[i-1] and V[i][j] < V[i-1][j-w[i-1]] + v[i-1]:

V[i][j] = V[i-1][j-w[i-1]] + v[i-1]

return V

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

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

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

n = 5

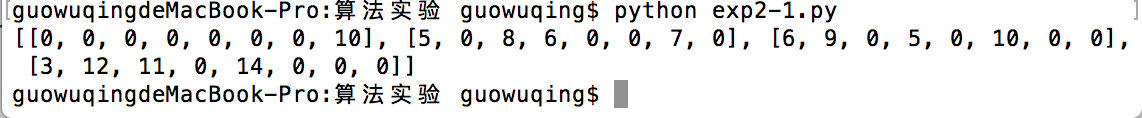
C = 10

result = KnapSack(w, v, n, C)

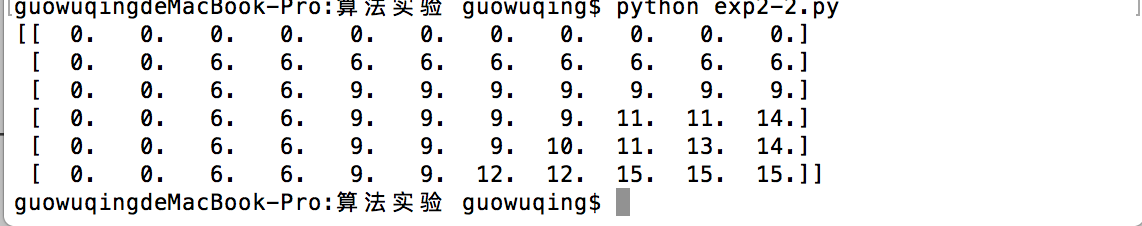
print result

1. 运行结果

实验一的结果矩阵如下：



实验二的截图如下：



5.分析与思考

动态规划法是一种自底向上的算法，所有动态规划问题都需要确定动态规划函数和填表。

实验一需要对定点集合的每个子集进行操作，所有时间复杂度是O（2\*\*n）；实验二的时间复杂度是O（n\*C）。

**实验三 贪心算法**

1.实验内容

（1）TSP问题最近邻点策略，并对算法进行时间复杂性分析。

（2）实现多机调度问题，并对算法进行时间复杂性分析。

2.实验目的

（1）TSP问题最近邻点策略，并对算法进行时间复杂性分析。

（2）实现多机调度问题，并对算法进行时间复杂性分析。

3.程序清单

（1）

# -\*- coding: utf-8 -\*-

\_\_author\_\_ = 'Victor'

import numpy as np

import sys

#贪心法求解Tsp问题最近临点策略

def Tsp(start, arc):

#start为开始和结束的节点；节点数字从1-5

way = []

way.append(start)

for e in range(len(arc)):

m = arc[start][:]

for n in way:

m[n] = sys.maxint

next = m.index(min(m))

way.append(next)

start = next

return way

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

arc = [[sys.maxint, 3, 3, 2, 6],

[3, sys.maxint, 7, 3, 2],

[3, 7, sys.maxint, 2, 5],

[2, 3, 2, sys.maxint, 3],

[6, 2, 5, 3, sys.maxint]]

way = Tsp(0, arc)

#计算路径长度

length = 0

for i in range(0, 5):

length += arc[way[i]][way[i+1]]

#way中的节点数字是从0-4的，对way列表中每个元素加一，对应节点1-5

way = [i+1 for i in way]

print "路径为:", way

print "路径长度为:", length

（2）

# -\*- coding: utf-8 -\*-

\_\_author\_\_ = 'Victor'

import numpy as np

import sys

#贪心法求解多机调度问题

def MultiMachine(work\_time, n, d, m):

s = []#存储m台机器处理的作业

timeList = sorted(work\_time.items(), key = lambda item:item[1], reverse = True)

print "按作业处理时间排序后的结果:", timeList

for w in range(0, m):

s.append([timeList[w][0]])

d.append(timeList[w][1])

for w in range(m, n):

j = d.index(min(d))

s[j].append(timeList[w][0])

d[j] += timeList[w][1]

print "每个机器处理的作业序号:", s

print "每个机器处理所有作业的总时长:",d

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

n = 7

work = [1, 2, 3, 4, 5, 6, 7]

t = [2, 14, 4, 16, 6, 5, 3]

wt = zip(work, t)

work\_time = dict((w, t) for w, t in wt)

print "作业序号和处理时间的对应关系:", work\_time

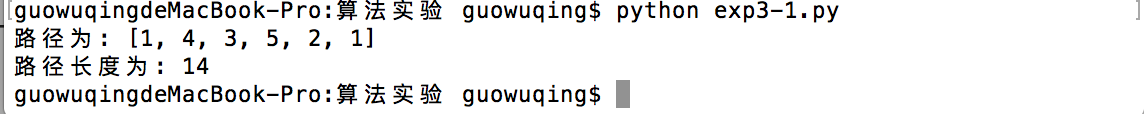
m = 3

d = []#存储m台机器空闲时间

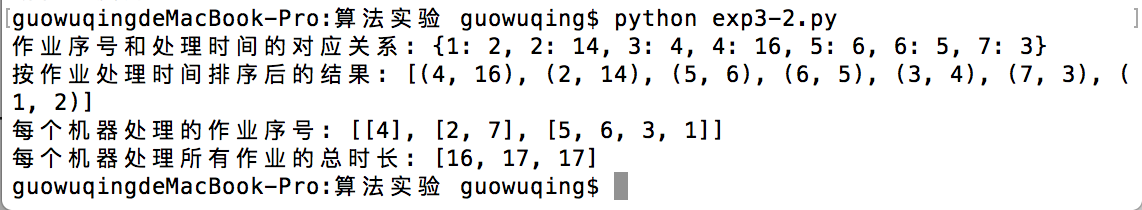
MultiMachine(work\_time, n, d, m)

4.运行结果

实验一的截图如下：



实验二的截图如下：



5.分析与思考

贪心法把一个复杂问题分解为一系列较为简单的局部最优选择，每一步选择都是对当前解的一个扩展，直到获得问题的完整解。贪心法可以较快的获得问题近似最优解。

实验一共进行n-1次贪心选择，每次都需要查找符合条件的最短边，所有它的时间复杂度为O（n\*\*2）；实验二的时间复杂度为O（n\*m）。

**实验四 回溯法**

1.实验内容

1. 利用回溯法编程求解0-1背包问题，并对算法进行时间复杂性分析；
2. 利用回溯法编程求解八皇后问题，并对算法进行时间复杂性分析。

2.实验目的

1. 利用回溯法编程求解0-1背包问题，并对算法进行时间复杂性分析；
2. 利用回溯法编程求解八皇后问题，并对算法进行时间复杂性分析。

3.程序清单

（1）

# -\*- coding: utf-8 -\*-

\_\_author\_\_ = 'Victor'

import numpy as np

from itertools import product

#回溯法求解0/1背包问题

def KnapSack(w, v, n, c):

resultList = []

result = []

for i in product(range(2), repeat=n):

resultList.append(i)

resultList.reverse()

for i in resultList:

flag = True

for j in range(len(i)):

if i[j] == 1:

c -= w[j]

if c < 0:

flag = False

if flag == True:

result.append(i)

c = C

return result

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

w = [20, 15, 10]

v = [20, 30, 25]

n = 3

global C

C = 25

result = KnapSack(w, v, n, C)

print "解空间为:", result

value = []

for i in result:

va = 0

for j in range(len(i)):

if i[j] == 1:

va += v[j]

value.append(va)

print "解空间中每个背包的价值分别为:", value

（2）

# -\*- coding: utf-8 -\*-

\_\_author\_\_ = 'Victor'

import numpy as np

#回溯法求解八皇后问题

def place(x, k): #判断是否冲突

for i in range(1, k):

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

return False

return True

def queens(n):

k = 1 #设置初始皇后为第一个

x = [0 for row in range(n + 1)]# 设置x列表初始值为0

while k > 0:

x[k] = x[k] + 1 # 在当前列的下一列开始

while (x[k] <= n) and (not place(x, k)): # 不满足条件，继续搜索下一列位置

x[k] = x[k] + 1

if x[k] <= n:# 判断是否为最后一个，不是就执行下一行

if k == n:# 是最后一个皇后，退出

break

else: # 不是，则处理下一行皇后

k = k + 1 #执行下一行

x[k] = 0 #初始化，从第一列开始

else:#n列均不满足，回溯到上一行

x[k] = 0 #初始化列到第一列

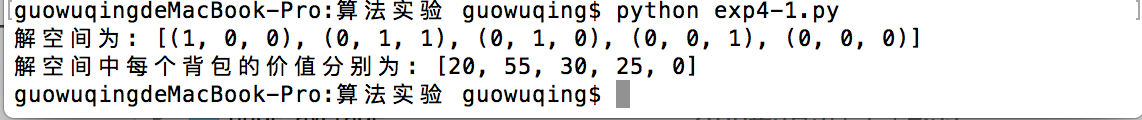
k = k - 1 #回溯到上一行

return x[1:] #返回1-8个皇后的位置

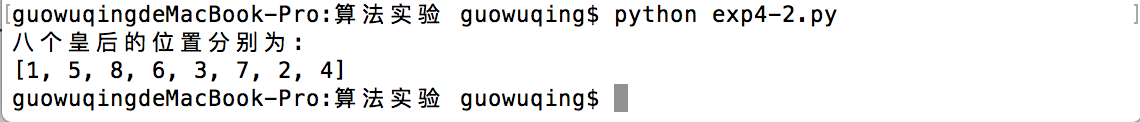
print(queens(8))

4.实验结果

实验一的截图如下：



实验二的截图如下：



5.分析与思考

回溯法在包含问题的所有可能解的解空间树中，从根节点出发，按照深度优先的策略进行搜索，若某一子树不满足问题的约束条件，则进行剪枝。它适合求解组合数较大的问题。

回溯法的时间复杂度在最坏情况下为指数阶，但算法通过剪枝可以大大减少搜索的子树，从而减少时间。