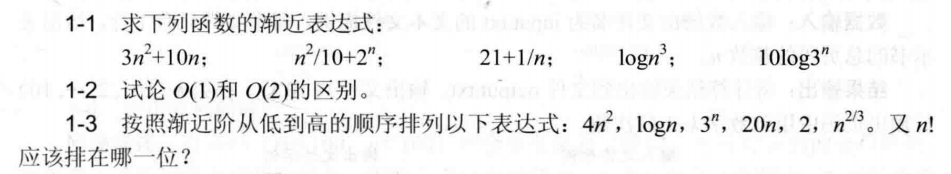
算法概论

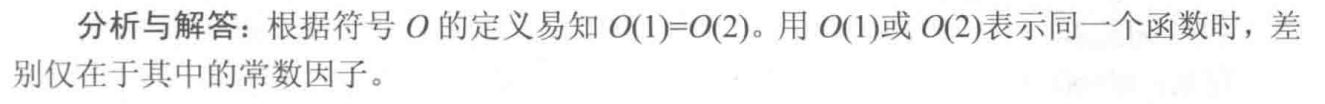
P7 算法分析题 1-3



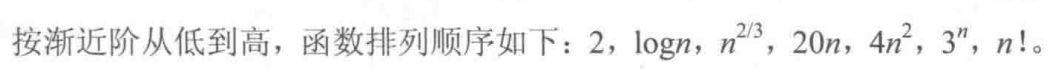
1-1

O(n2) O(2n) O(1) O(logn) O(n)

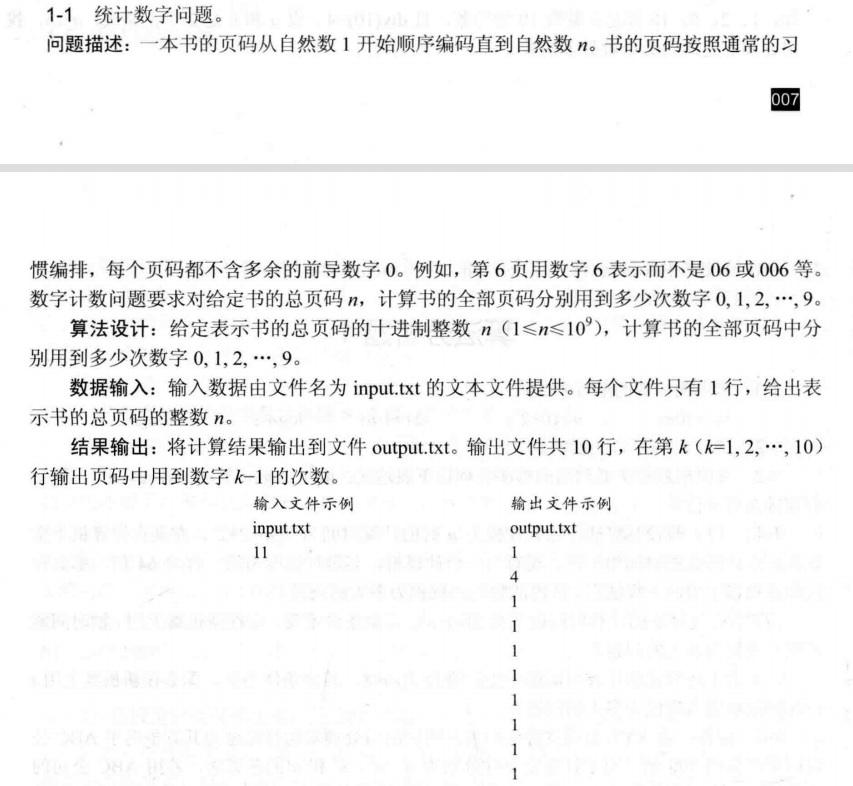
1-2

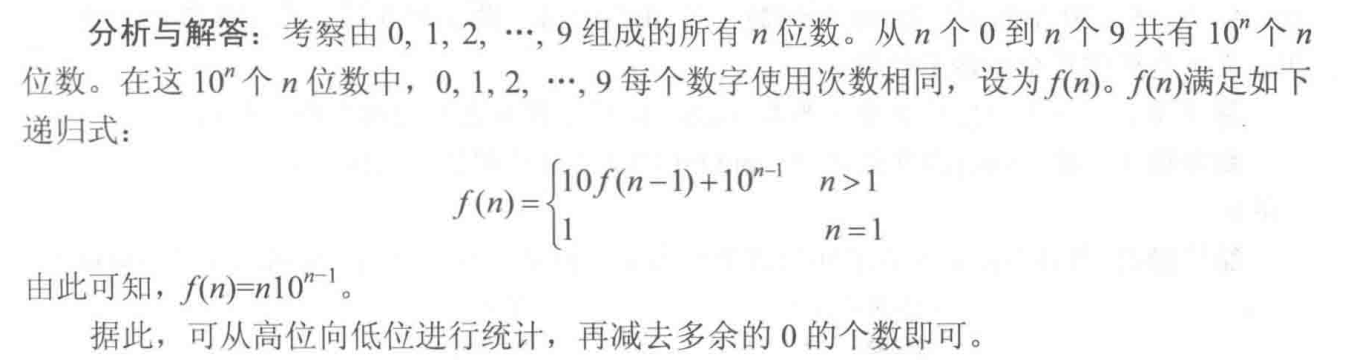


1-3



算法实现题：1.1 1.2 1.5





# 以5678为例

def solve(n):

msg = str(n)

# 数字n的长度

length = len(msg)

# 首位数字 p

p = int(msg[0])

# 公式处理 得到（000-999）\*（0，1，2，3，4）

for i in range(10):

c[i] += p \* (length - 1) \* int(pow(10, length - 2))

# 最高位处理 在最高位出现的（0，1，2，3，4）的次数还没统计进去

for i in range(p):

c[i] += int(pow(10, length - 1))

# 余项处理 得到位数t

t = int(pow(10, length - 1))

t = n%t

if t == 0:

c[p] += 1

return

lenT = len(str(t))

if lenT != length - 1: # 如果下一个开头是0的话(length - 1 - lenT)表示有几个0 (t + 1)表示有几行

c[0] += (length - 1 - lenT)\*(t + 1)

c[p] += 1 + t # 把5000-5678中出现的5加上

return solve(t)

# 存储0到9数字的次数

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

n = 8

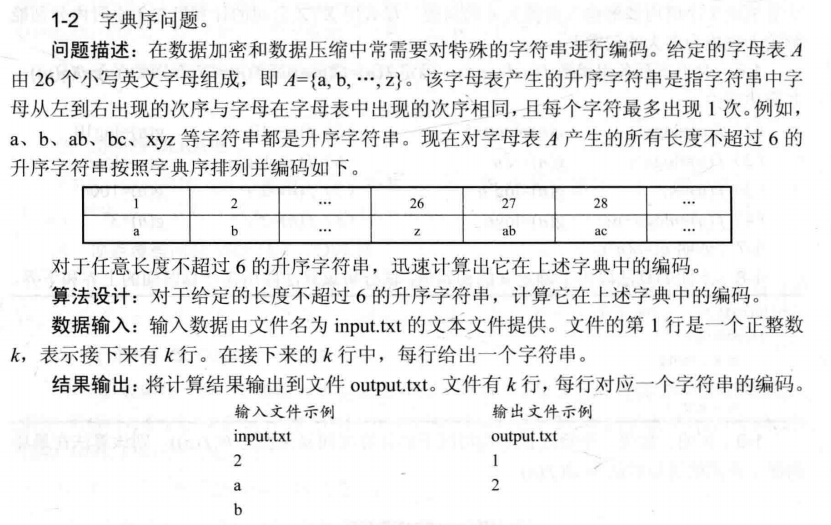
solve(n)

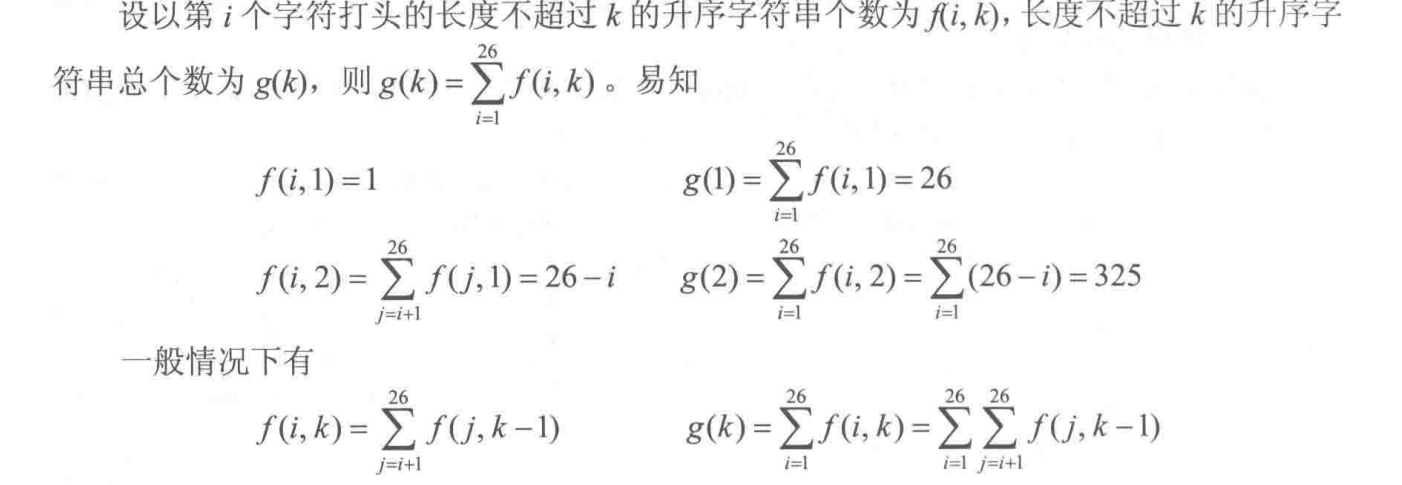
# 去零 前导0个数分别为10， 110，1110 ，即个长度n下是10\*\*i

for i in range(0, len(str(n))):

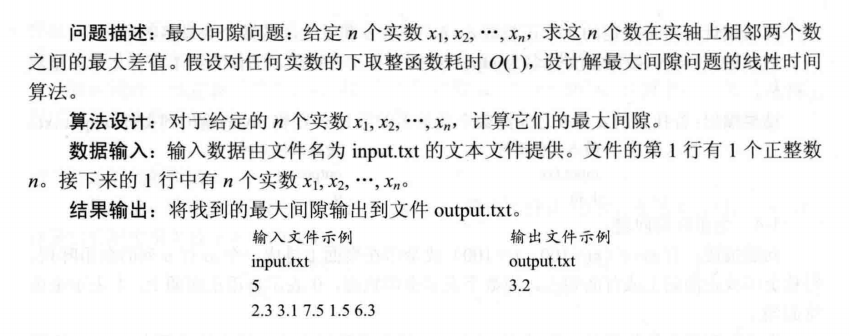
c[0] -= pow(10, i)

print(c)





1.5 最大间隙问题



将范围分为n+1个桶，最小值在第一个桶，最大值在最后一个桶，至少有一个空桶

最大间隙 一定出现在空桶左边最大和右边最小中。

def maximumGap(nums):

if len(nums) < 2:

return 0

max\_val, min\_val = max(nums), min(nums)

if max\_val == min\_val:

return 0

n = len(nums)

size = (max\_val - min\_val) / (n - 1)

bucket = [[None, None] for \_ in range(n + 1)] # 保存每个桶中的最大最小值

for num in nums: # 更新桶中数字的最大最小值

b = bucket[int((num - min\_val) // size)]

b[0] = min(b[0], num) if b[0] else num

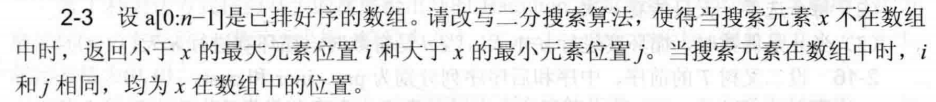
b[1] = max(b[1], num) if b[1] else num

bucket = [b for b in bucket if b[0] is not None] # 去掉空桶

return max(bucket[i][0] - bucket[i - 1][1] for i in range(1, len(bucket))) # 找到相邻最大的值

1. **分治法**

算法分析题 2.3 2.5 2.7 2.13 2.10



def bs(a, x):

L = 0

R = len(a)-1

while L<=R:

m = (L + R) / 2

if a[m] == x:

return m, m

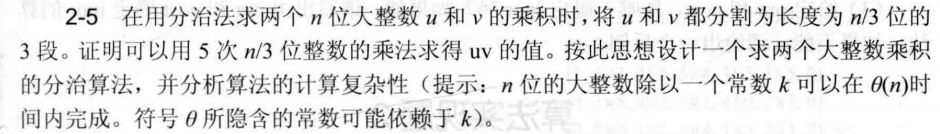
elif a[m]<x:

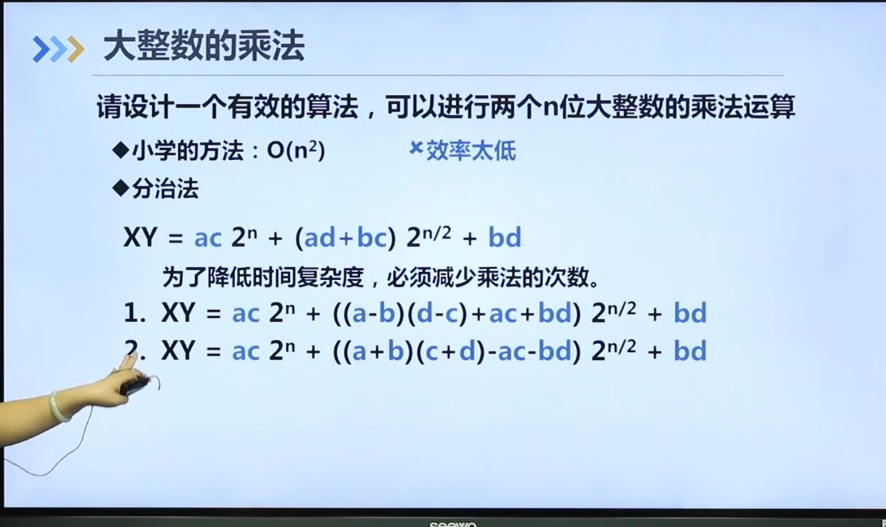
L = L + 1

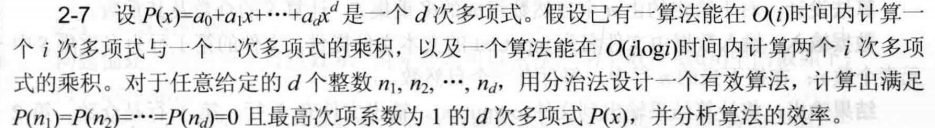
else:

R = R - 1

return R, L



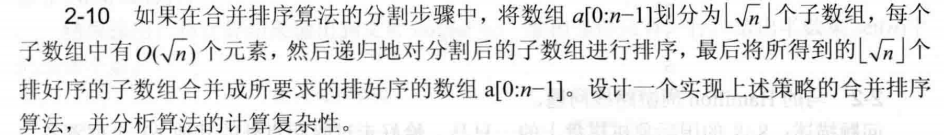




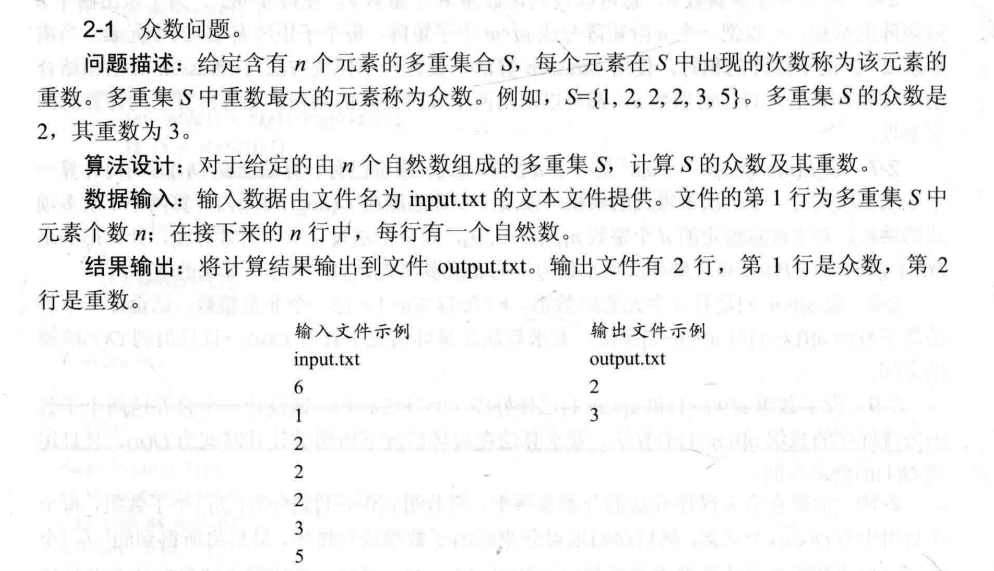
IMG_256



2.10



算法实现题 2.1 2.5



res = 0

\_max = 0

t = -1

n = 0

for x in a:

if x==t:

n += 1

else:

if \_max<max:

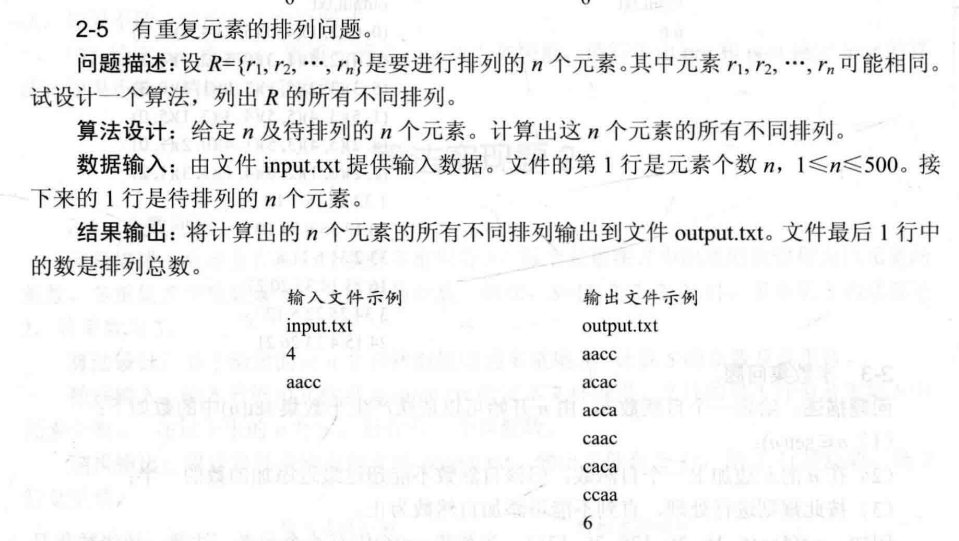
res = t

t = n

n = 1

if \_max<max:

res = t



res = set()

l = len(nums)

def pl(k):

if k>=l-1:

res.add(tuple(nums))

for i in range(k,l):

nums[k],nums[i] = nums[i], nums[k]

pl(k+1)

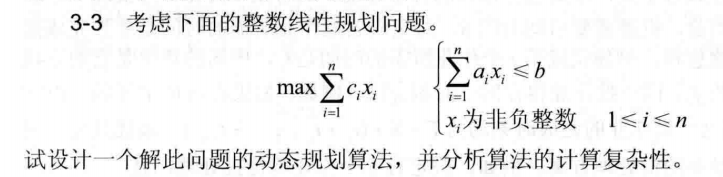
nums[k],nums[i] = nums[i], nums[k]

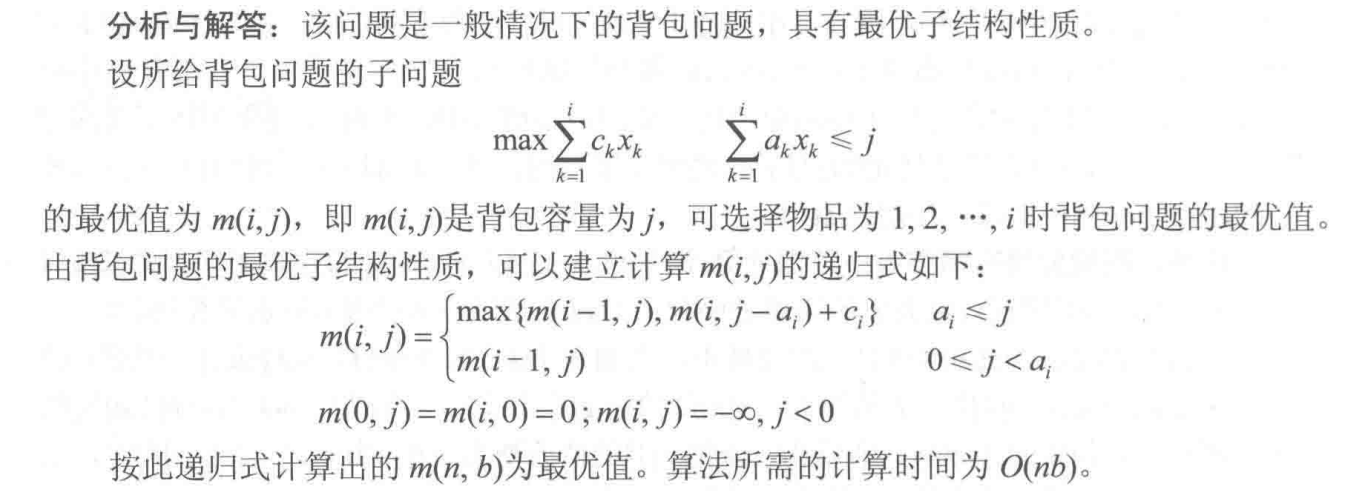
pl(0)

return list(res)

1. **动态规划**

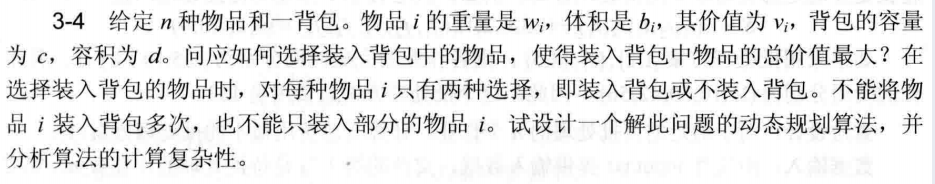
算法分析题 3.3 3.4





c是价值，a是重量

b是承载量



容量c是可以放下的最大重量

同0-1背包相类似，只是多了一个限制，我们可以使用三维数组进行处理

dp[i][m][n] = max(dp[i-1][m-w][n-b]+V[i], dp[i-1][m][n])

我们注意到 每次计算只与前一状态i-1相关，因此可以对存储空间进行优化，成二维数组

dp[m][n] = max(dp[m-w][n-b]+V[i], dp[m][n])

但是当我们从前往后遍历时，dp已经被赋值为这一状态的值，会导致错误

因为每次比较都是和[m-w][n-b]的值相关，从前往后的话，你遍历到后面的时候，用的[m-w][n-b]的值就是错的，所以从后往前，这样用的[m-w][n-b]还没被遍历到，还是上一状态的值

因此 需要从后往前遍历至放的下物体i的条件为止

W = [3,2,6,7,1,4,9,5]

B = [6,2,4,6,7,3,8,5]

V = [6,3,5,8,3,1,6,9]

#Count = [3,5,1,9,3,5,6,8]

c = 20

d = 25

l = len(W)

dp = [[0] \* (d+1) for i in range(0,c+1)]

for i in range(0,l):

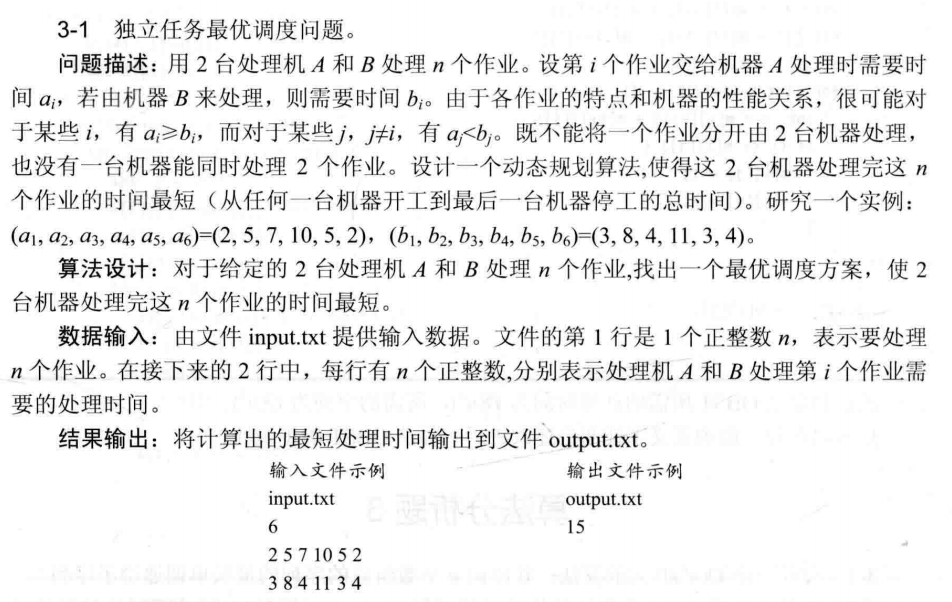
for m in range(c, W[i]-1, -1):

for n in range(d, B[i]-1, -1):#逆序遍历

dp[m][n] = max(dp[m][n], dp[m-W[i]][n-B[i]] + V[i])

print(dp[c][d])

算法实现题 3.1 3.2 3.3 3.6 3.17



n = 6

a = [0,2,5,7,10,5,2]

b = [0,3,8,4,11,3,4]

sa = 0

# p[i][k] 表示最优的完成k个任务 A花费i时间的条件下，在B上还需要p[i][k]时间 总共需要max(p[i][k],i)的时间

# i表示条件，并不是一定都用来做任务了

对于前n个任务来说，时间最多到全部由A完成的sa，数组里面的值表示B将部分任务交给A后，B还需要的时间

p = [[0 for \_ in range(n+1)] for \_ in range(30)]

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

sa += a[k]

for i in range(sa+1): # 在A当前最多花费时间sa

p[i][k] = p[i][k-1] + b[k] # 把任务k交给B需要的时间

# 如果这个时候 A所花费的时间可以完成任务k

if i >= a[k]:

p[i][k] = p[i][k] if p[i][k] < p[i-a[k]][k-1] else p[i-a[k]][k-1] #选择是否给A做

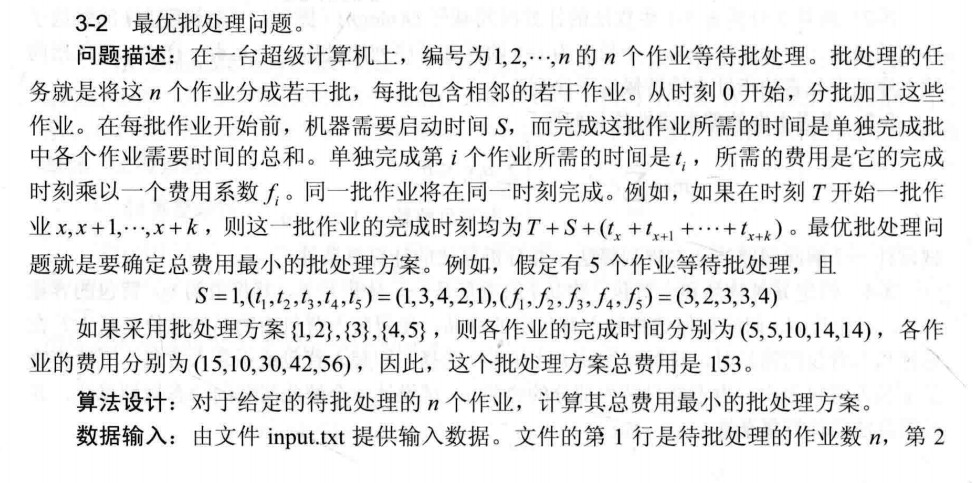
如果给A做，B花费的时间就是A只有i-a[k]的时间时，B完成k-1个任务的时间

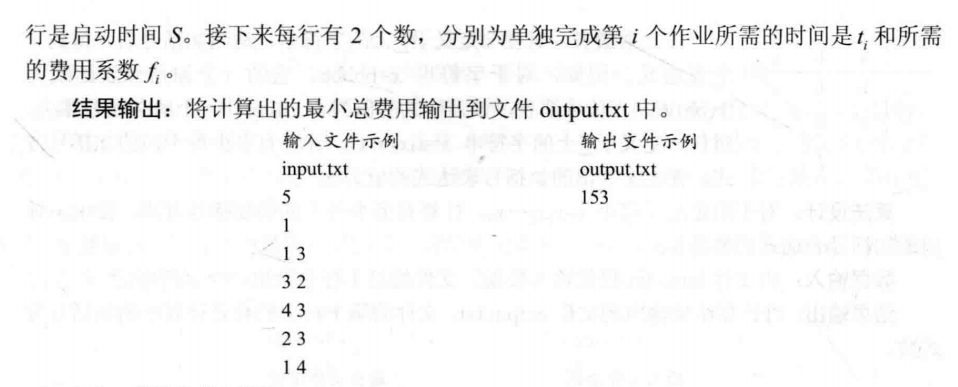
\_min = 1000

for i in range(sa):

\_min = min(\_min, max(i, p[i][n]))

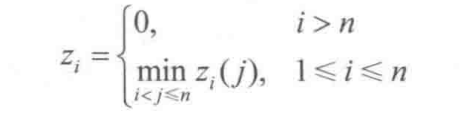
print(\_min)





作业只能顺序完成，Zi是指从i-n作业需要的代价，从后往前遍历所有的i，求解每个子问题，

确定新的i所在的第一批应该是到哪里结束。



n = 5

s = 1

t = [0, 1, 3, 4, 2, 1]

w = [0, 3, 2, 3, 3, 4]

st = [0 for \_ in range(n+2)]

sw = [0 for \_ in range(n+2)]

z = [999 for \_ in range(n+2)]

z[n+1] = 0

for i in range(n, 0, -1): # 从后往前累计时间/权重

st[i] = st[i+1] + t[i]

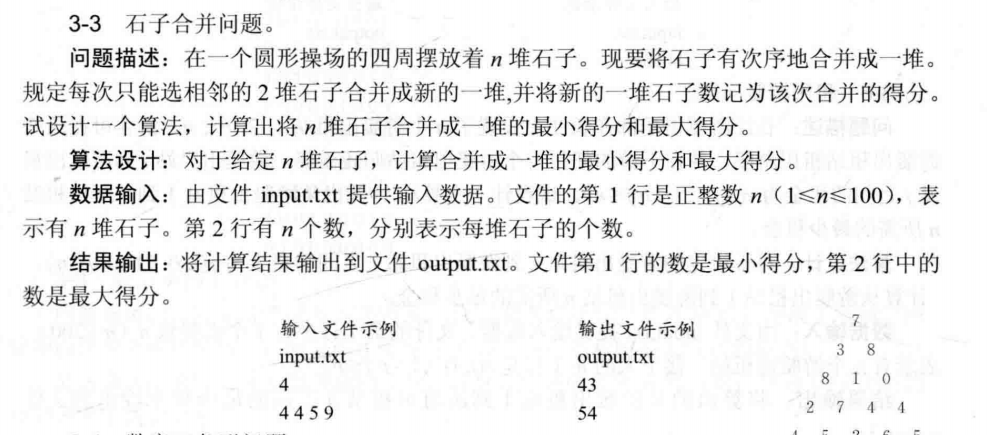
sw[i] = sw[i+1] + w[i]

for i in range(n, 0, -1): # 从后往前遍历

for j in range(i+1, n+2): # 遍历划分点

z[i] = min(z[i], z[j]+sw[i]\*(st[i]+s-st[j]))

print(z[1])



这里写图片描述

sum = [[0 for \_ in range(n)] for \_ in range(n)] # sum[i][j]表示i到j石子的个数

dp = [[0 for \_ in range(n)] for \_ in range(n)] # dp[i][j]表示取i到j的石子的最小代价

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

sum[i][i] = w[i]

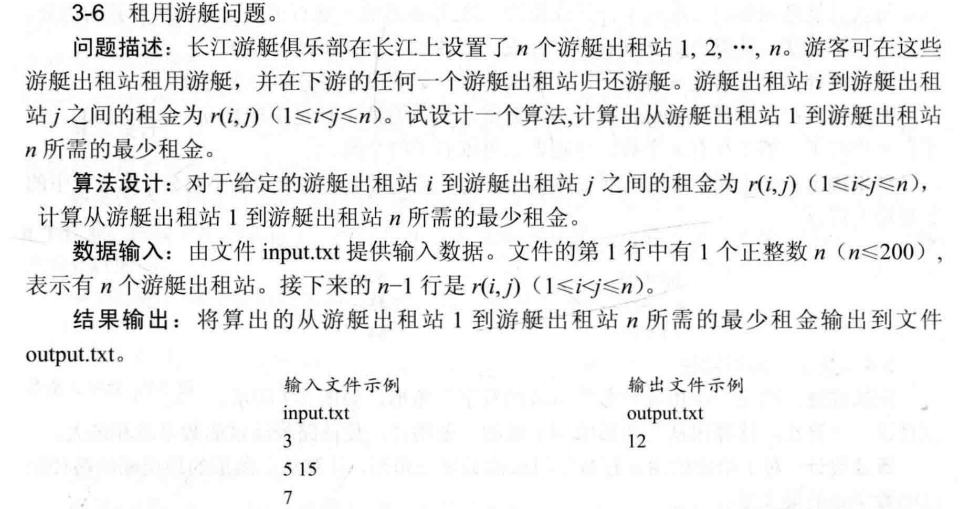
dp[i][i] = w[i]

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

sum[i][j] = sum[i][j-1]+w[j]

dp[i][j] = min([dp[i][k]+dp[k+1][j] for k in range(i, j)]) + sum[i][j] # 在i到j之间找最小分割点 即左右两边需要的代价之和最小的点，再加上本次合并的代价，得到dp[i][j]

return dp[0][n-1]

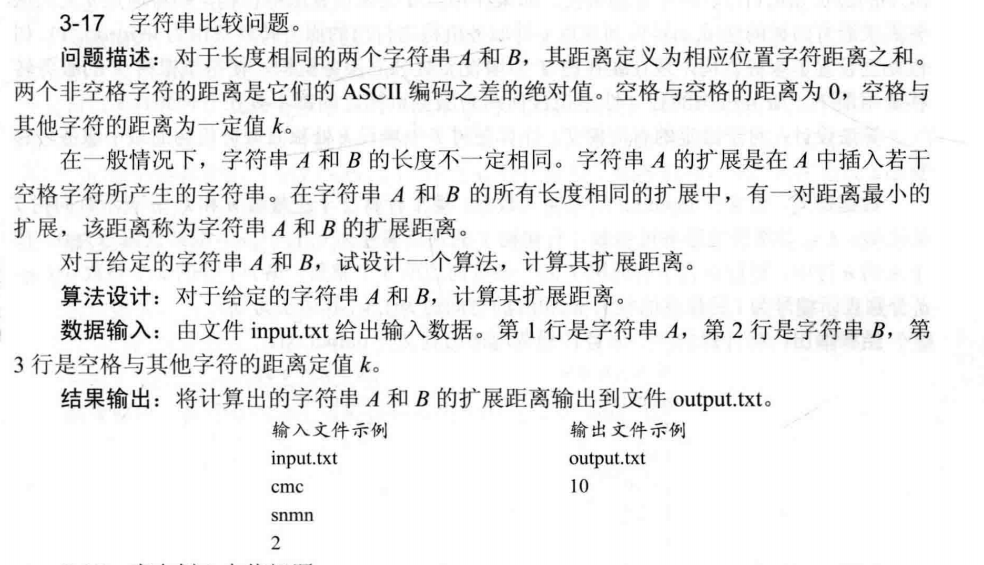


dp = [0 for \_ in range(n)]

for I in range(n):

dp[i] = min([dp[j]+r[i][j] for j in range(i)])

return dp[n-1]



dis = 0

la = len(a)

lb = len(b)

l = max(la,lb)

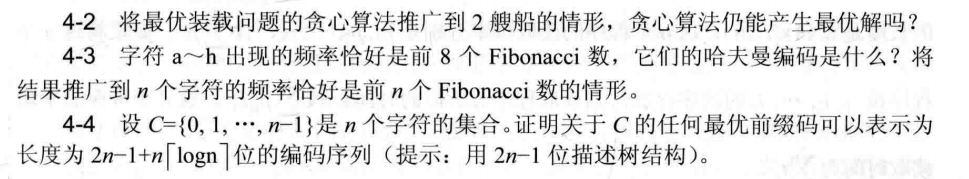
for I in range(l):

dis += ord(a[i])-ord(b[i]) if i<la and i<lb else k

return dis

1. 贪心算法

算法分析题 4.1 4.2 4.4



4-2:

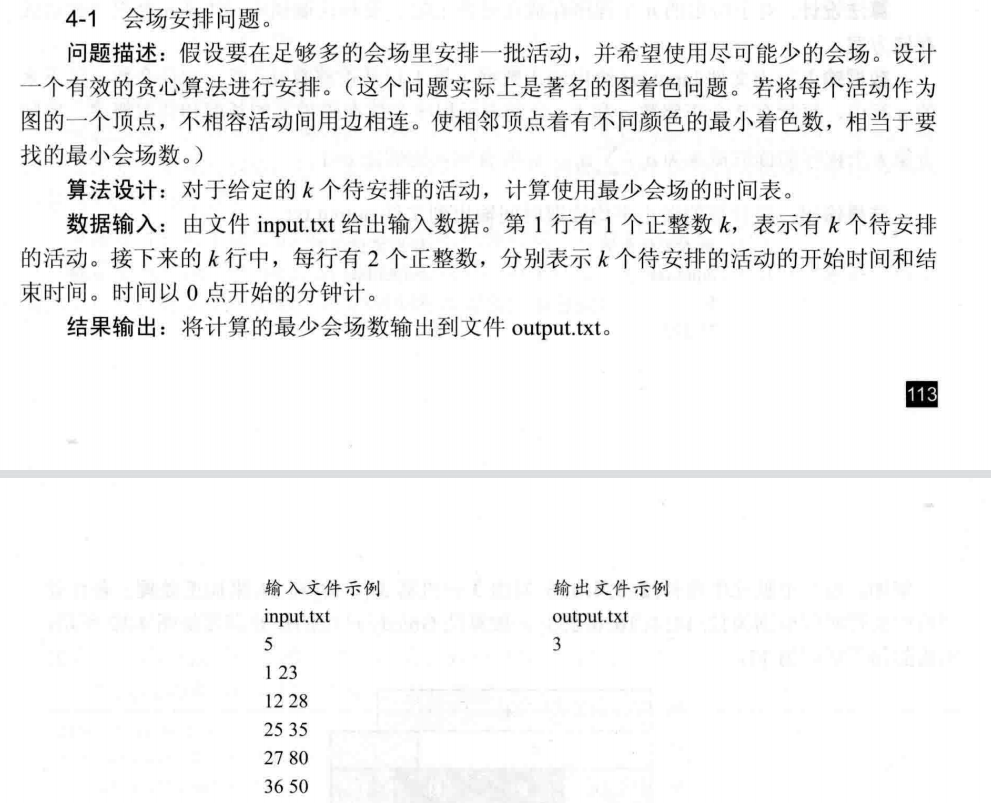
不能，因为从小到大装货物，可能导致空间的浪费

4-3：



4-4

算法实现题 4.1 4.6 4.9



按照开始时间从小到大排序

end = [] # 记录每个会议室的结束时间

for i in range(n):

f = 0

end.sort() # 按结束时间从小到大排

for j in range(len(end)): # 遍历所有会议室

if a[0]>=end[j]: # 如果当前活动开始时间大于结束时间

f = 1

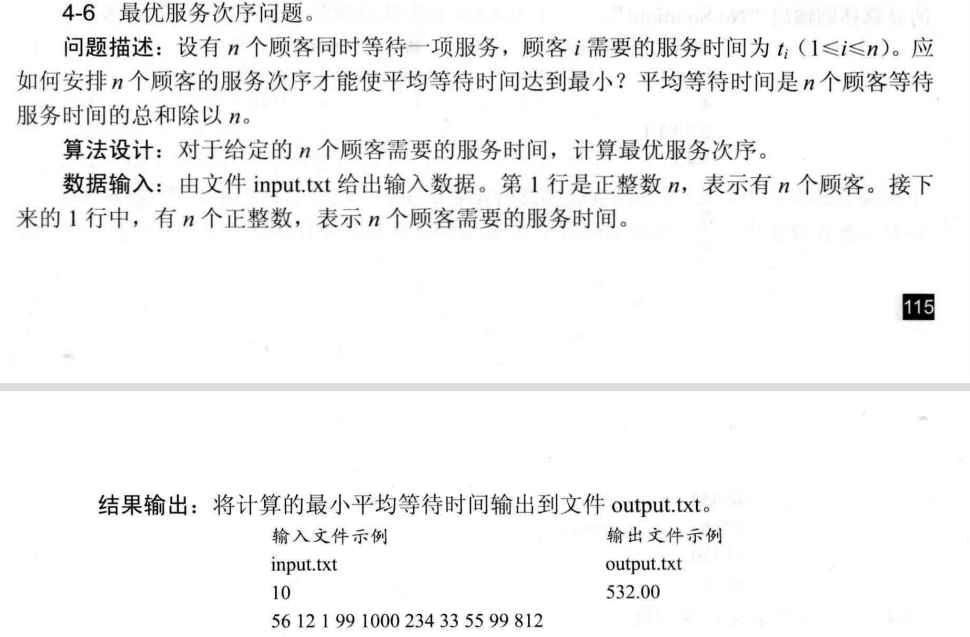
end[j] = a[1] # 更新会议室结束时间

break

if f==0:

end.append(a[1])

return len(end)



因为要求平均等待时间最小，等价于总服务时间最小，不妨先将每个顾客需要的服务时间从小到大排序，然后以此顺序进行服务，

times.sort()

wait = 0

t = 0

for i in range(n):

wait += t

t += times[i]

return wait



直接贪心，每次走尽可能远的路

print('请输入汽车加满油可行驶的距离n和加油站k，中间用空格隔开：')

n, k = map(int, input().split())

print('请输入每个加油站之间的距离，空格隔开')

d = list(map(int, input().split()))

c = 0 # 初始化加油次数

now = n # 初始化当前油量

for i in range(k + 1): # 开始旅途~

if now < d[i]: # 判断当前油量是否可以到达下一个加油站

c += 1 # 如果无法到达，则需要加油。加油次数+1

now = n # 重新初始化油量

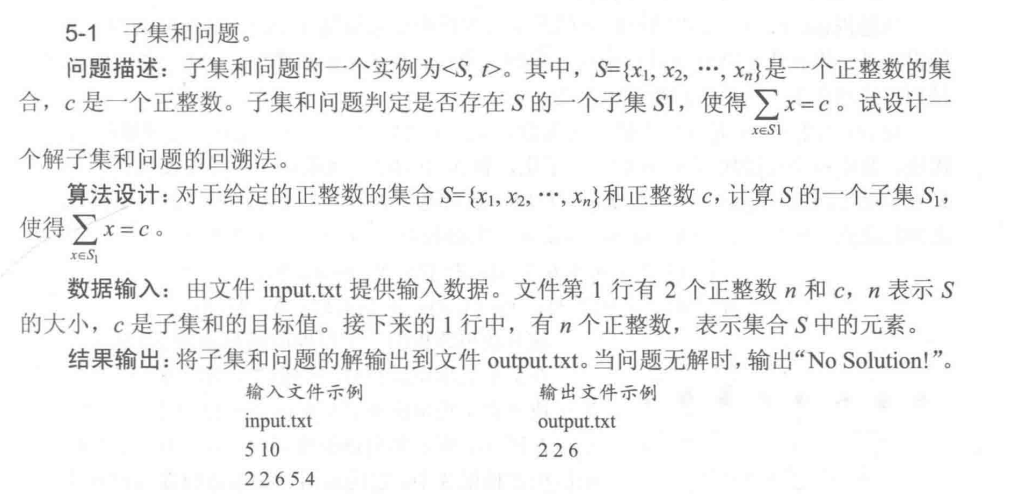
now -= d[i] # 减去到达下一站所需油量

if now < 0: # 判断此时当前油量

c = 'No Solution' # 当前油量为负，说明无法到达下一站，输出无解

break # 不能再愉快的旅行了

print(c) # 输出结果



w = [2, 2, 6, 5, 4] n = 5 c = 10

r = sum(w) # 剩余的元素的和 cw = 0 # 当前的总和

x = [0 for \_ in range(n)] bestx = [0 for \_ in range(n)] # 放入集合中的元素位置

bestw = 0 # 上一步得到的总和

def backtrack(i, cw, r, bestw):

if i >= n: # 如果遍历到头

for j in range(n):

bestx[j] = x[j]

bestw = cw

return bestw == c

r -= w[i] # 把i放入待选集合

if cw+w[i] <= c: # 如果放了i后不会超过c 则放i试试看

x[i] = 1

cw += w[i]

if backtrack(i+1, cw, r, bestw):

return True

cw -= w[i]

# 到这里意味着i放不进去 或者 放进去是不行的

if cw+r > bestw: # 如果不放i 结果可以比之前不行的更大

x[i] = 0

if backtrack(i+1, cw, r, bestw): # 不放i遍历

return True

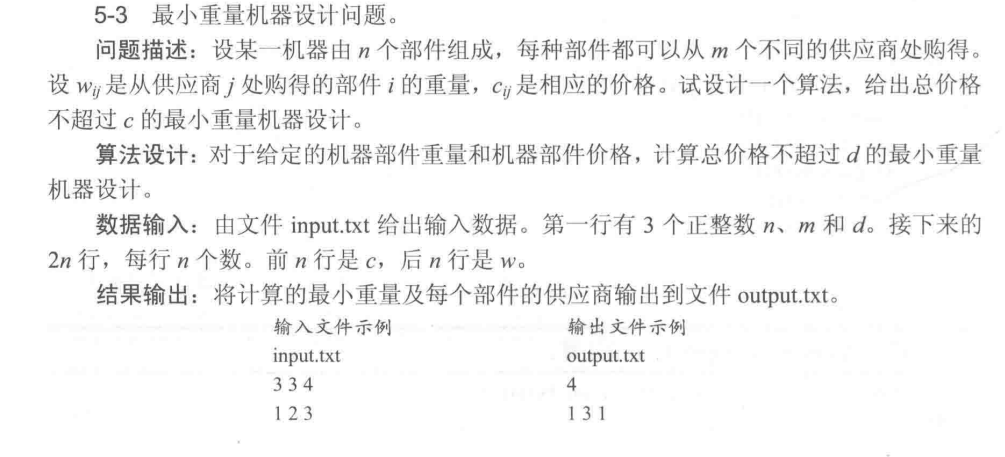
return False

if backtrack(0, cw, r, bestw):

for i in range(n):

if bestx[i] == 1:

print(w[i])



n = 3 # 部件个数 m = 3 # 供应商个数 d = 4 # 最大价格

c = [[1, 2, 3], [3, 2, 1], [2, 2, 2]] # c[i][j]表示从j处买i部件的价格

w = [[1, 2, 3], [3, 2, 1], [2, 2, 2]] # w[i][j]表示从j处买i部件的重量

cw = 0 # 现有重量

cp = 0 # 现有价格

bestw = 99999999

x = [0 for \_ in range(n)] # 每个部件选择哪个供应商

bestx = [0 for \_ in range(n)]

def backtrack(i, cw, cp, bestw):

if i >= n: # 所有部件都确定好了

if cw < bestw: # 是否更优

bestw = cw

for j in range(n):

bestx[j] = x[j]

return True

found = False

for j in range(m): # 遍历每个供应商

x[i] = j

cw += w[i][j]

cp += c[i][j]

if cp <= d and cw < bestw: # 是否符合约束

if backtrack(i+1, cw, cp, bestw):

found = True

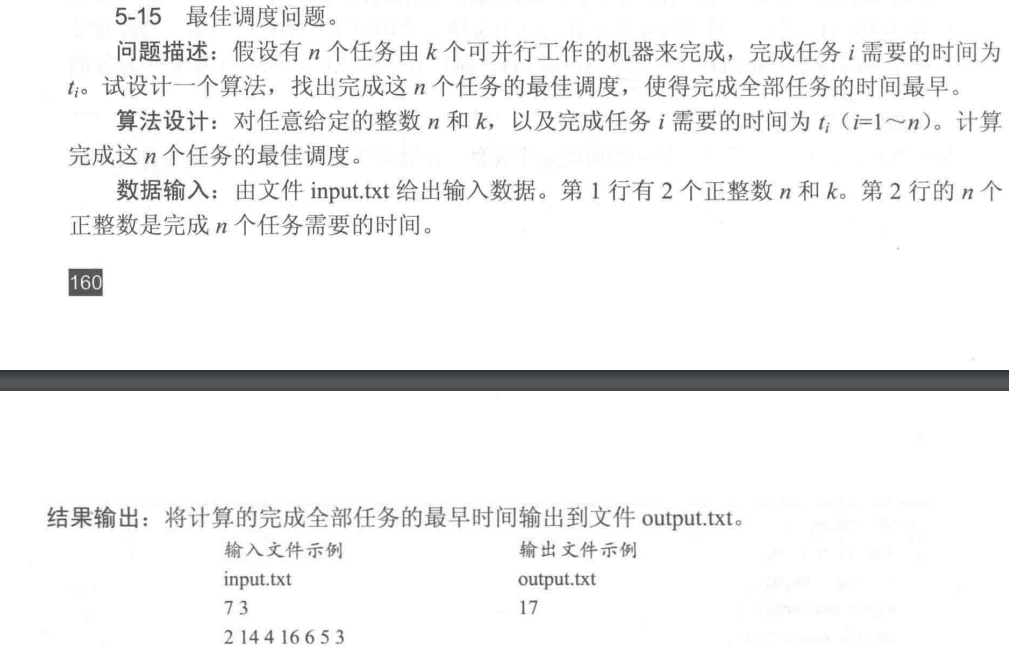
cw -= w[i][j]

cp -= c[i][j]

return found

if backtrack(0, cw, cp, bestw):

print(bestw, bestx)



n = 7

k = 3

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

tk = [0 for \_ in range(k)]

# 遍历所有可能性

def backtrack(i, best):

if i >= n: # 全部选择完毕

best = min(best, max(tk))

return best

for j in range(k): # 为部件i选择供应商

tk[j] += t[i]

if tk[j] < best: # 剪枝

best = backtrack(i+1, best)

tk[j] -= t[i]

return best

print(backtrack(0, 9999999))

