算法分析与复杂性理论：

大作业

李梓铉

学号 3118103163

指导老师: 张选平教授

2019 年 5 月 10 日

1 复杂度问题

1.1 题目

已知 T(n) $\leq \begin{cases} c_1 & n \leq 1 \\ \max(T(i)+T(n-i-1))+c_2 n & n>1 \end{cases}$,(1 <= i <= n − 1) c_1, c_2, 为正常数，请证明 $T(n)=\Theta(n^2)$。

1.2 证明

令 T(n) 对应最坏情况的时间则

$$T(n)=\max(T(i)+T(n-i-1))+c_2 n \tag{1}$$

假设对于任何 k<n, 总有 $T(k) \leq ck^2$ 其中 c 为常数，显然 k=1 时成立。将归纳假设带入 (1) 式可以得到:

$$T(n) \leq c\max(i^2+(n-i-1)^2))+c_2 n \tag{2}$$

显然在 [1,n-1] 上，i=1 或 n-1 时取最大值 $(n-2)^2+1$ 于是有:

$$T(n) \leq cn^2+c(5-4n)+c_2 n \leq cn^2 \tag{3}$$

因为 n>=2, 所以只要 c 足够大，第二个等号就成立。因此有 $T(n)$ <= cn^2. 根据归纳法，k 成立则 n 成立，且 k=1 时成立。所以对于所有的 n 都有 $T(n)$ <= cn^2. 得证

2 凸包问题

2.1 题目

给定平面上 n 个点，从中找出一个最小点集，使该点集所组成的凸多边形包围所有的 n 个点。用分治法编写一个求解凸包问题的算法。

2.2 算法设计

首先想一个可行的暴力算法，依次取点集 n 中所有可能的边 C_n^2, 每取一条边，遍历其他所有点判断是否都在该边的上方或下方。满足条件即为凸包的一条边。该算法复杂度易得是 $O(n^3)$。使用分治法，将点集按横坐标顺序从左到右排序（nlogn）均分为每个含有五个点的点集。可知每个子集只有限次搜索即可得到子集的凸集，找到所有子集的凸集的时间复杂度为 $O(n)$。此时再用一个巧妙地方法将子集凸包合并起来即可。相邻凸包的左边凸包的最右点与右边凸包的最左点作为初始点对，不妨先选左边凸包为调整对象。在点集中依横坐标减小顺序向左搜索纵坐标大于最右点的点，直到该点连线与左边凸包相切（每判断一次相切至需要子凸包的点数 n' 次），同理再调整右边凸包的点，直到找到上方的公共切线。下方切线同理。显然点数有限，每两个凸包合并时间复杂度即为 O(n)。将合并的凸包与该凸包右边的子凸包继续合并，就可得到原点集的凸包，合并的时间复杂度为 $O(n^2)$。因此算法复杂度为 $nlogn + O(n) + O(n^2) = O(n^2)$。

算法的主要过程总结如下:

1. 输入：包含 n 个点的点集
2. 将点集根据横坐标顺序排序
3. 按顺序每五个点作为一个子点集进行划分
4. 在每个子点集中使用暴力枚举得到子集的凸包
5. 寻找相邻子集凸包的上下两条切线连接的点
6. 合并所有子集的凸包
7. 输出: 点集的凸包

2.3 测试实例

使用 python 语言实现了一个个数为 9 的点集的求解，但包含了算法中所有的关键步骤如暴力求解部分，寻找切线部分，和合并部分都在。如果求解大于 10 的个数的点集还做不到完全的自动化，还需要修改，但由于是算法作业，在验证算法可行性上已经足够。下面是测试结果:

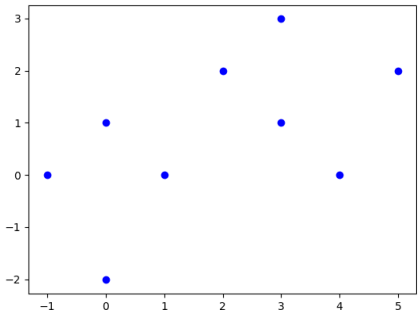

图 1: 原始点集

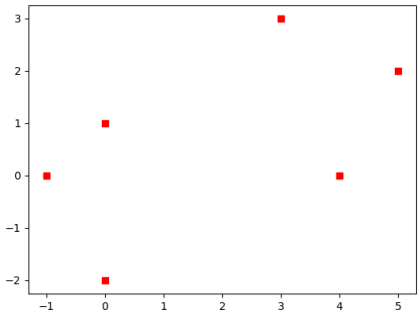

图 2: 求解得到的凸包

2.4 源代码

实验源代码如下.

```
# Input : First Polygon  : {(2, 2), (3, 3), (5, 2), (4, 0), (3, 1)}
#         Second Polygon : {(-1, 0), (0, 1), (1, 0), (0, -2)}.
# Output : Upper Tangent - line joining (0,1) and (3,3)
#          Lower Tangent - line joining (0,-2) and (4,0)
import math
import matplotlib.pyplot as plt

def sign_ret(num):
    if(num==0):
        return 0
    if(num > 0):
        return 1
    else:
        return -1

def split(li):
    li.sort(key=takefirst)
    li_temp = [[]]
    num0 = math.floor(len(li) / 5)
    for i in range(num0):
        li_temp[i] = li[i:i + 5]
```

```
    if len(li) - 5 * num0 != 0:
        li_temp.append(li[5 * num0:len(lists)])
    return li_temp

def takefirst(elem):
    return elem[0]
def takesecond(elem):
    return elem[1]

def find_upper_tangent(a0,b0):
    a=[]
    b=[]
    for i in range(len(a0)):
        if takesecond(a0[i]) >= takesecond(a0[len(a0)-1]):
            a.append(a0[i])

    for i in range(len(b0)):
        if takesecond(b0[i]) >= takesecond(b0[0]):
            b.append(b0[i])

    inc_find=0;
    # a 是左边的凸集， b 是右边的凸集
    n1 = len(a)-1
    n2 = len(a)-2

    n3 = 0
    n4 = 1

    a_cross = 1
    b_cross = 1
    p1 = a[n1]
    p2 = a[n2]
    p3 = b[n3]
    p4 = b[n4]
    while inc_find==0:
        while a_cross==1:
            p2 = a[n2]
            p_temp = a[n2-1]
            if (p_temp[0]-p3[0])*(p2[1]-p3[1])/(p2[0]-p3[0]) >= p_temp[1]-p3[1]:
                n1 = n2
                n2 = n1 - 1
                p1 = a[n1]
                p2 = a[n2]
                a_cross=0
            else:
                n2 = n2-1
            print('across=0')

        while b_cross==1:
            p4 = b[n4]
            p_temp = b[n4+1]
            if (p_temp[0]-p1[0])*(p4[1]-p1[1])/(p4[0]-p1[0]) >= p_temp[1]-p1[1]:
                n3 = n4
                n4 = n3+1
                p3 = b[n3]
                p4 = b[n4]
                p_temp = a[n2]
                b_cross = 0
                if (p_temp[0] - p3[0]) * (p2[1] - p3[1]) / (p2[0] - p3[0]) >= p_temp[1] - p3[1]:
                    inc_find = 1
                else:
                    a_cross=0
            else:
                n4=n4+1
    return(p1,p3)

def find_lower_tangent(a0,b0):
    a=[]
    b=[]
    for i in range(len(a0)):
```

```
93          if takesecond(a0[i]) <= takesecond(a0[len(a0)-1]):
                a.append(a0[i])
95
        for i in range(len(b0)):
97          if takesecond(b0[i]) <= takesecond(b0[0]):
                b.append(b0[i])
99
        inc_find=0;
101     # a 是左边的凸集， b 是右边的凸集
        n1 = len(a)-1
103     n2 = len(a)-2

105     n3 = 0
        n4 = 1
107
        a_cross = 1
109     b_cross = 1
        p1 = a[n1]
111     p2 = a[n2]
        p3 = b[n3]
113     p4 = b[n4]
        while inc_find==0:
115         while a_cross==1:

117             p2 = a[n2]
                p_temp = a[n2-1]
119
                if (p_temp[0]-p3[0])*(p2[1]-p3[1])/(p2[0]-p3[0])<=p_temp[1]-p3[1]:
121                 n1 = n2
                    n2 = n1 - 1
123                 p1 = a[n1]
                    p2 = a[n2]
125                 a_cross=0
                else:
127                 n2=n2-1
                print('across=0')
129         while b_cross==1:
                p4 = b[n4]
131             p_temp = b[n4+1]
                if (p_temp[0]-p1[0])*(p4[1]-p1[1])/(p4[0]-p1[0])<=p_temp[1]-p1[1]:
133                 n3 = n4
                    n4 = n3+1
135                 p3 = b[n3]
                    p4 = b[n4]
137                 p_temp = a[n2]
                    b_cross = 0
139                 if (p_temp[0] - p3[0]) * (p2[1] - p3[1]) / (p2[0] - p3[0]) <= p_temp[1] - p3[1]:
                        inc_find = 1
141                 else:
                        a_cross=0
143             else:
                    n4 = n4+1
145
        return(p1,p3)
147
    def merge_convex(a0,b0):
149     pu = find_upper_tangent(a0,b0)
        pl = find_lower_tangent(a0,b0)
151     convex0 =[]
        for i in range(len(a0)):
153         if (takefirst(a0[i]) <= takefirst(pu[0]) and takesecond(a0[i]) >= takesecond(a0[0]) ) or (takefirst(a0[i]) <= takefirst(pl[0]) and
                takesecond(a0[i]) <= takesecond(a0[0]) )  :
                convex0.append(a0[i])
155     for i in range(len(b0)):
            if (takefirst(b0[i]) >= takefirst(pu[1]) and takesecond(b0[i]) >= takesecond(b0[0]) ) or (takefirst(b0[i]) >= takefirst(pl[1]) and
                takesecond(b0[i]) <= takesecond(b0[0]) )  :
157             convex0.append(b0[i])
        return(convex0)
159
    def brutehull(li):
161     hull=[]
```

```
    hull0=[]

    for i in range(len(li)):
        for j in range(i+1,len(li)):
            p1=li[i]
            p2=li[j]
            pos = 0
            neg = 0
            for k in range(len(li)):
                p3=li[k]
                if (p2[0] - p1[0]) * (p3[1] - p1[1]) - (p3[0] - p1[0]) * (p2[1] - p1[1])>=0:
                    pos = pos+1
                if (p2[0] - p1[0]) * (p3[1] - p1[1]) - (p3[0] - p1[0]) * (p2[1] - p1[1])<=0:
                    neg = neg+1
            if pos == len(li) or neg == len(li):
                hull0.append(p1)
                hull0.append(p2)

    for i in hull0:
        if i not in hull:
            hull.append(i)

    li.sort(key=takefirst)
    return hull



lists = [(2, 2), (3, 3), (5, 2), (4, 0), (3, 1),(-1, 0), (0, 1), (1, 0), (0, -2)]

split_li=split(lists)

hull1=brutehull(split_li[0])


testlist1=[(2, 2), (3, 3), (5, 2), (4, 0), (3, 1)]
testlist2=[(-1, 0), (0, 1), (1, 0), (0, -2)]
testlist1.sort(key=takefirst)
testlist2.sort(key=takefirst)
convex= merge_convex(testlist2,testlist1)

print(find_upper_tangent(testlist2,testlist1))
print(find_lower_tangent(testlist2,testlist1))

plt.figure(1)
x=[]
y=[]
for i in range(len(convex)):
    x.append(takefirst(convex[i]))
    y.append(takesecond(convex[i]))
plt.scatter(x,y,color='red',marker='s')
x=[]
y=[]
plt.figure(2)
for i in range(len(testlist1)):
    x.append(takefirst(testlist1[i]))
    y.append(takesecond(testlist1[i]))
plt.scatter(x,y,color='blue')
x=[]
y=[]
for i in range(len(testlist2)):
    x.append(takefirst(testlist2[i]))
    y.append(takesecond(testlist2[i]))
plt.scatter(x,y,color='blue')

plt.show()
print('finished')
```

3 动态规划算法

3.1 问题

设计一个动态规划算法，找到字符串 T[1..n] 中前向和后项相同的最长连续子串。前向子串和后项子串不能重叠。例如:

1. 输入“ALGORITHM”，算法返回空串;
2. 输入“RCURSION”，算法返回“R”;
3. 输入“REDIVIDE”，算法返回“EDI”。

3.2 算法设计

最差情况下要找到最大子串显然所有字符都要比较至少一次才能确保找到最大字符串，考虑动态规划的算法思路，比较字符是否相同显然是相关子问题被重复计算了很多次。如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。所以我用一个表来记录所有已解的子问题的答案。先将字符串反向排列作为一个二维矩阵的纵向指标，再将原字符串作为二维矩阵的横向指标。再开始填写表，如果横纵指标相同，且字符不是翻转前后的同一个字符，那么该空填为 1，否则填为 0，完整的填写需要 n^2 次，图 3 给出了该部分的示意图。

	R	E	D	I	V	I	D	E
E	0	1	0	0	0	0	0	0
D	0	0	1	0	0	0	0	0
I	0	0	0	1	0	0	0	0
V	0	0	0	0	0	0	0	0
I	0	0	0	0	0	1	0	0
D	0	0	0	0	0	0	1	0
E	0	0	0	0	0	0	0	1
R	0	0	0	0	0	0	0	0

图 3: 表格示意图

然后将矩阵从左下角到右上角对角遍历循环输出便可得到所有可能的对称子串，如图 4 所示。找出其中 1 个数最多的子串即为前向后向相同的最长连续子串，至多需要 2n-1 次。所以算法复杂度为 $O(n^2)$

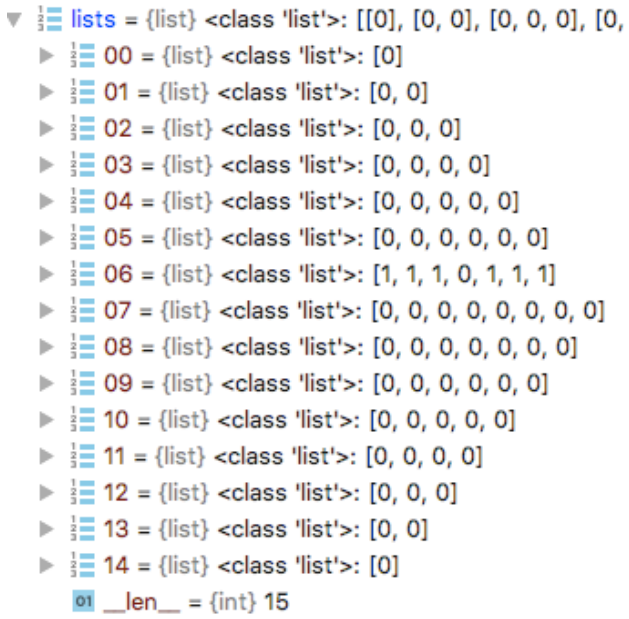

图 4: 对角遍历输出

算法的主要过程可以总结如下：

1. 输入：字符串 T[1..n]ϕ
2. 得到字符串反向排列
3. 填写比较表格
4. 对角遍历输出所有可能的对称子串，找到对称最长连续子串对应的字母对应的位置
5. 输出: 最长连续子串

3.3 测试示例

以题目给的三个例子，输入“ALGORITHM”，算法返回空串; 输入“RCURSION”，算法返回“R”; 输入“REDIVIDE”，算法返回“EDI”，作为测试，并且测试很多其他例子，都没有问题。下面是测试结果。

```
/usr/local/bin/python3 /Users/StarkLee/PycharmProjects/algorithm-practice/LCS_DYNAMIC_PROGRAMMING.py
[]
['R']
['E', 'D', 'I']

Process finished with exit code 0
```

图 5: 返回的最长连续子串

3.4 源代码

实验源代码如下.

```
1
import numpy as np
3 import math


5
def string_reverse(string):
7     return string[::-1]

9 def LCS_str(s1):
      num1=len(s1)
11    lists = [[] for i in range(num1*2-1)]
      arr = [[0 for i in range(num1)] for j in range(num1)]
13    arnp= np.asarray(arr)
      temp = [i for i in range(num1)]
15    temp0 = np.asarray(temp)
      s2 = string_reverse(s1)
17    cnt =[0 for i in range(num1*2-1)];
      ntemp=num1;
19
      for i in range(len(s2)):
21        for j in range(len(s1)):
              if s1[i] == s2[j] and i != num1 - j -1:
23                arr[i][j] = 1
              else:
25                arr[i][j] = 0

27    for i in range(num1*2-1):
          if i < num1 :
29            for j in range(i+1):
                  #print ([num1-1-temp0[i]+j],[j])
31                lists[i].append (arr[num1-1-temp0[i]+j][j])
          else:
33            ntemp=ntemp-1;
              for j in range(ntemp):
35                lists[i].append (arr[j][j+1+i-num1])

37    for i in range(num1 * 2 - 1):
          cnt[i]=lists[i].count(1)
39
      num4 = max (cnt)
41    i=cnt.index(num4)
      cnt2 =[0 for i in range(num4)]; #存储重复字段
43
      ntemp = num1;
45    num3=0;
      if i < num1 :
47        for j in range(i+1):
              if arr[num1-1-temp0[i]+j][j] == 1:
49                cnt2[num3] = (s1[num1-1-temp0[i]+j])
                  num3=num3+1
51    else:
          ntemp=ntemp-i+num1-1;
53        for j in range(ntemp):
              if arr[j][j+1+i-num1] == 1:
55                cnt2 [num3] = (s1[j])
                  num3 = num3+1
57
      print (cnt2[0:math.floor(num4/2)])
59


61            # if arr[i][num1-j-1] == arr[i+1][num1-j]:
              #       cnt[i] = cnt[i]+1
63 LCS_str('ALGORITHM')
   LCS_str('RECURSION')
65 LCS_str('REDIVIDE')
```

4 贪心算法

4.1 问题

假定 J=1,2,3,...,n 是 n 个等待在同一台机器上加工的作业集合，每个作业所需要的加工时间都为 1 个时间单位，且每个作业 i 都有一个最迟完成时间 di。如果一个作业 i 能够在 di 之前完成，就可获得收益 pi>0; 反之，则不获得收益。请设计一个贪心的算法求该问题的一个最佳安排方案使得收益最大，并证明你所设计的贪心策略的最优性。

4.2 算法设计

贪心算法在每一部做出局部最优的选择，并寄希望于这样的选择可以得到全局最优。对于题目，如果在截止时间后完成，任务称为延迟，如果在截止时间前完成，称为提前的。对于任意一个安排方案都可以以提前优先的形式，即将提前的任务都放在延迟任务之前，并将提前任务然截止时间单调递增的顺序排列。如果存在一个任务集合，有调度方案使得所有任务都不延迟，则称该任务是独立的。原问题及转换为寻找最优独立提前子任务集。较为显然对于任意的有 n 个元素的独立任务集合有，t 在 [1...n] 时，集合中截止时间小于 t 的任务数量一定小于 t（因为大于的话一定会发生延迟）。因此贪心算法设计思路就是，优先选择收益最大的作业，并每选择一个作业后验证更新之后的任务集合仍然满足独立条件。每次验证独立性的时间复杂度为 O(nlogn)，又需要进行 n 次独立性验证，因此时间复杂度为 $O(n^2logn)$

算法的主要过程可以总结如下：

1. 输入：任务集 $D = x_1, x_2, ..., x_m$，
2. 对任务集根据奖励价值由高到低进行排列
3. for i = 1,2,...,m do
4. 对于集合 fin 增加任务集中的第 i 个元素
5. 检验集合 fin 的独立性
6. if 不满足独立性，删除掉该元素
7. endif
8. endfor
9. 输出: 最优的安排方案

最优性证明：如果 S 是一个给定截止时间的单位时间任务集合，$\mathcal{I}$ 是所有独立任务集合的集合，则对应的系统 $S, \mathcal{I}$ 可以证明是一个拟阵：假设 B 和 A 是独立的任务集合，且 $|B| > |A|$. 取 k 是满足 $N_t(B) \leqslant N_t(A)$ 最大的 t。因为 $N_n(B) = |B|$ 且 $N_n(A) = |A|$ 但 $|B| > |A|$。因此对任意 j：$k+1 \leq j \leq n$, 必然有 k<n 且 $N_j(B) > N_j(A)$, 因此 B 比 A 包含更多截止时间为 k+1 的任务。令 ai 为 B-A 中截止

时间为 k+1 的任务，令 $A' = A \cup \{a_i\}$ ，和之前证明独立一样，较为显然对于任意的有 n 个元素的独立任务集合有，t 在 [1...n] 时，集合中截止时间小于 t 的任务数量一定小于 t（因为大于的话一定会发生延迟），易得 A' 必然也是独立的。而对于 $0 \leq k \leq n$ 有 $N_t(A') = N_t(A) \leqslant t$. 所以 A' 独立，所以 $S, \mathcal{I}$ 是一个拟阵。而拟阵具有贪心选择性质，即如果某个元素初始不是最优的选择，那么在随后也不会选入最优集合，这保证了删除掉元素的正确性，而由于拟阵的最优子结构性质，寻找 fin 的最大权重独立子集并将它扩展，总体效果就是找到原集合的一个最大权重子集。

4.3 测试示例

输入任务集合 d = [(70,4), (60,2), (50,4), (40,3), (30,1), (20,4), (10,6)] 第一项为奖励，第二项为截止时间。最优解将会舍弃 (30,1) 与 (20,4) 两个任务。

```
/usr/local/bin/python3 /Users/StarkLee/PycharmProjects/algorithm-practice/greedy.py
[(60, 2), (40, 3), (70, 4), (50, 4), (10, 6)]

Process finished with exit code 0
```

图 6: 最优任务安排

算法得到了最优解，实现验证。

4.4 源代码

实验使用的源代码如下.

```
import numpy as np
import math
import copy
def takefirst(elem):
    return elem[0]
def takesecond(elem):
    return elem[1]
d = [(70,4), (60,2), (50,4), (40,3), (30,1), (20,4), (10,6)] #第一项为奖励，第二项为截止时间

fin=[]
d.sort(key=takefirst,reverse=True)
for i in range(len(d)):
    fin.append(d[i])
    temp=copy.deepcopy(fin)
    temp.sort(key=takesecond)
    for j in range(len(temp)):
        if temp[j][1]<j+1:
            fin.pop() #pop 一定要带括号
            break

fin.sort(key=takesecond)
print (fin)
```

5 集合覆盖问题

5.1 问题

给定一个集合 X = $\{x_1, x_2, \ldots x_n\}$ 以及 X 的一个子集簇 F = $\{f_1, f_2, \ldots, f_n\}$，其中，$f_i \subseteq X$。求 F 的一个最小子集 C，使得 C 中的集合能够覆盖集合 X，即 ($U_{S \in C} S = X$), 完成以下两个问题

1) 已知图的顶点覆盖问题是 NPC，证明集合覆盖问题是 NP 难的;

2) 利用回溯法或分支界限法设计求解集合覆盖问题的算法。

5.2 证明

首先证明集合覆盖问题是一个 np 问题：显然，证书为一个 F 的一个子集，对于一个给定的子集验证是否能覆盖集合 X，只需要多项式时间，只需将元素挨个比较即可，即集合覆盖问题是一个 np 问题。

下面证明顶点覆盖问题可以规约到集合覆盖问题：对于一个顶点覆盖实例 G=(V,E). S=E, 对于 $\forall v_i \in V$, 与 vi 相连的边可以组成一个 E 的子边集 c_i, 和 C 对应。显然规约时间是多项式时间。

假设有一个大小至少为 k 的顶点覆盖 A，则 A 的每个顶点都可以对应一个集合 C 的元素组成集合 C'，因为 A 是顶点覆盖，因次覆盖了所有的边，所以 $\bigcup_{c \in C'} c = S$, 且 $|C'| \leq k$

假设有 C 的子集 C', 且 $|C'| \leq k$ 且 $\bigcup_{c \in C'} c = S$，因为 C'的每个元素代表了 G 中与某个顶点相连的边，因此 C' 元素对应的顶点组成集合 A，且一定可以覆盖 G 中全部的边，又因为 $|C'| \leq k$，因此 A 即为 G 的至多为 k 的顶点覆盖。

因此顶点覆盖问题可以在多项式时间规约到集合覆盖问题，又因为已知图的顶点覆盖问题是 NPC 问题，所以集合覆盖问题是 NP hard 的。

5.3 算法设计

使用分支界限法，将集合簇 F 中的元素按所含元素分类，写函数：首先含有 x_1 的作为一类，不妨以 x_1 类作为开始节点，每次选出类中的一个子集合，根据子集和中的元素 (如含有 x_1, x_2)，就排除不需要的类 (如 x_1, x_2 类)，递归调用自己，直到所有类均被排除，即找到了可行解。可行记录可行解对应的子集簇的元素个数。并在之后搜索中将大于当前最优解的节点直接排除。直到或节点表为空即遍历了 x_1 中的元素之后。即得到了集合 X 的覆盖。但是程序设计上，由于递归调用要求递归的任务数是不断减少的，而根据算法设计，递归的任务数是不断增加的，因此程序设计尝试了很久都没有实现。所以本节没有包含代码。但觉得 NPC 的问题程序设计还不如直接穷举简洁，而算法复杂度上实际差别是不大的。

6 结论与讨论

最后感谢张教授这学期的教学，让我收获很多。只是有些可惜课时太少，很希望未来算法分析这门课可以调整成全周的课程。顺颂时祺。