算法第五次作业

34.1

a)

独立集判定:

> {<G, V'>: G=(V, E)是一个无向图，V‘是V的子集；任意v1∈V'且v2∈V', E中不存在(v1, v2)，任意v3∈V', 都能找到v4 ∈ V', 使E中存在(v3, v4)}`
>
> 同样相当于{<G, V'>: G=(V, E)是一个无向图，V‘是V的子集；(V', E) 中顶点最大度数为当成二分图处理的最大}`

1. 验证:
   对于给出的V'中任何两个点，试图在E中找到对应的，有|V‘|**2 种可能。属于多项式时间判定的算法
2. 下列证明 独立集问题<=p团问题
   取G‘ = G(V', E), 不存在两个相邻顶点，使其满足以上，即不存在k = 2 的团
   可以规约为p团问题

故其为np-complete的

b)

设`judge(G, V')`能满足需求 `N(v)`为点的相邻节点的集合 `delete(V', v)`返回删除v，并且更新度的图

`delete_other(V', v)`,返回删除v相邻的所有节点，保留v的图

以上两种delete不改变原V‘

1. 建立无向图，统计无向图各店的度

2. 对于每个顶点：

   1. 顶点度为0，则点必定在最大独立集中
   2. 顶点度为1，N(v)只有一个点，v, N(v)不能同时存在集合中，把v加入集合，N(v)中点删除
   3. 删除了度为0、1的顶点，剩下的全是度大于二的节点

`TraverseAndCount(G)`用二分的方式遍历，求出需要求的团的大小

在算法运行前，先用`TraverseAndCount(G)`求出需要的x, 再带入GET_MIS递归 / 迭代求解

```
GET_MIS(G, V, x):
    for v in G.v:
        if judge(G, delete(V, v)):
            V = delete(V, v)
        else:
            V = delete_other(V', v)
```

这样能够保证求出的是最后的MIS

c)

度全为2...这怕不是很多个环?

从一个节点开始处理，选中该点，并把该点相连的所有的顶点(2个)删除，再找出度数为0的顶点

```
BINARY_FIND_MIS(G)
    S = {}  //使s为一个空集合
    for node in G.V:
        if node.marked:
            continue
        else:
            curnode = node
            while curnode.marked == false:
                // 标记这个节点
                curnode.marked = true
                S.add(curnode)          // 把此节点加入目标集合
                lastnode = curnode      // 标记旧节点
                for node in curnode.adaj:
                    if node.marked == false:
                        curnode = node
                if lastnode == curnode:
                // 没有标记新的节点,则此连通分量已经遍历完毕
                    break
```

34.4

a)

{<T, P, D, L>: T, P, D，T[i] 表示第i个任务耗时，P[i]表示第i个任务的利润，D[i]表示任务的完成时间，L表示实际任务调度的列表，}

b)

这我真不会了，狗狗我

c)

给出一个序列L, 按L的顺序调度，在多项式时间验证其是最佳调度方案

将a1, ... an 以 key = d (结束时间)排序

构建max-p[n][d[n]]， max-p[i][j] 表示j时间考虑i个任务的最大获利

从第一个任务到第n个任务，只要所有的调度都满足贪心选择特性，就可以说其满足

```
SEQ = queue()   // 作为结果队列，先使其为空队列
for i <- 1 upto n
    for j <- 1 upto d[i]
        // 获取不调度情况下的最大值，作为初始化的值
        max-p[i][j] = max-p[i - 1][min(j, d[i - 1])]
        max-p[i][j].last =
        if j < t[i]:
            // 不满足加入的条件，这任务是废的
            continue
        // 调度的最大值为是否调度中的最大值
        if_update = max-p[i - 1][j - t[i]] + p[i]       // 调度任务i后的值
        if if_update > max-p[i][j]
            max-p[i][j].last = max-p[i - 1][j - t[i]]
            max-p[i][j] = if_update
            max-p[i][j].append = True
```

用以上算法计算出是否满足，再带入L

```
sum_time = 0    // 已经花去的时间
for i <- 1 upto n:
    // 对于L 的所有任务
    if max-p[i][t[L[i]] + sum-time].append == false:
        // max-p[i][t[L[i]] + sum-time].append == false，则最优调度没有在这里调度这个任务
        return false
    // 增加总耗时
    sum-time += t[L[i]]
// 1 -- n，所有的任务都是被最优调度的
return true
```

d)

1. 将a1, ... an 以 key = d (结束时间)排序
2. 构建max-p[n][d[n]]， max-p[i][j] 表示j时间考虑i个任务的最大获利
   可以获得递推式
   `max-p[0][i] = 0`
   `max-p[0][0] = 0`
   若有任务t截止时间d[t], 则
   `max-p[i][j] = max(max-p[i - 1][j], max-p[i - 1][j - t[i]] + p[i])`

3. 在max-p中带上last字段，表示上一个对象; append对象表示是否添加，初始化为False

```
SEQ = queue()   // 作为结果队列，先使其为空队列
for i <- 1 upto n
    for j <- 1 upto d[i]
        // 获取不调度情况下的最大值，作为初始化的值
        max-p[i][j] = max-p[i - 1][min(j, d[i - 1])]
        max-p[i][j].last =
        if j < t[i]:
            // 不满足加入的条件，这任务是废的
            continue
        // 调度的最大值为是否调度中的最大值
        if_update = max-p[i - 1][j - t[i]] + p[i]       // 调度任务i后的值
        if if_update > max-p[i][j]
            max-p[i][j].last = max-p[i - 1][j - t[i]]
            max-p[i][j] = if_update
            max-p[i][j].append = True
```

35.1

1

取序列 L = {x1, x2, ..., xn}, s为序列的和. 希望找到<L, s>, 能找到S= {S1, S2, ..., Sk}, 使得L中项拼凑组合起来可以求得S，且S中任意两项和大于1.

1. 对于给定的一组S解，可以用一个O(n^2)的算法求出任意两项的和。
   对于给定的S，求得S1,...,Sk 是否只在{x1, x2, ..., xn}各个和出现一次，等价于在L中找出 和为S1, S2... Sn 的集合，这个问题等价于子集合问题。故其为NP-Complete的

2

S 为si之和，若S不为整数，显然至少upper_bound(S) 才能装的下，若S为整数，也要至少S个箱子。其定义等价于upper_bound(S)

3

1. 若si > 0.5切能够放入，放入的箱子不会不到半满
2. 若si < 0.5, 放入第一个不到半满的箱子，此时这个箱子只有si，si是最早的、唯一的非半满的箱子。则有如下讨论

   j > i, 若 sj < 0.5, sj 最晚会被放入si所在的箱子，箱子容量大于0.5则此事不再存在半满，小于0.5则等价于里面有一个s = si + sj重的箱子

故至多有一个半满

4

假设箱子数目到达upper_bound(2S)，且一个箱子最大容量为1，根据(3)结论，至多有一个箱子不到半满，则最小容量大于`(upper_bound(2S) -1 ) * 0.5 + 0.5 = 0.5upper_bound(2S` 即一个箱子接近空，剩下容量全部接近刚好不能插进这些箱子的值

Upper_bound(2S) >= 2S, 0.5Upper_bound(2S) >= S

又因为2S箱子最小容量大于Upper_bound(2S) 故放不下，不可能

5

我记得有个证明是17/10最优箱子之类的?

设最优需要K* 箱子，此算法用K个箱子

2) —> K* >= Upper_bound(S)

4) —> K <= Upper_bound(2S)

得出K/K* 约为2

6

```
// 让boxes为一个扩容数组
Boxes = Vector()
for object in Objects:
    append = false // 是否被加入，初始化为false
    for box in Boxes:
        if box + object < 1:
            // 可以被加入，则添加
            box.add(object)
            append = true
        if append == false:
            // 没有被加入，则在Boxes新加一个盒子容量相等的量
            Boxes.append(object)
```

复杂度:

对象数目N，对象和S

`for object in Objects`执行了N次

`for box in Boxes`每次执行时，有**2倍已求质量和**的上界(已经装箱的物体质量和的两倍)

可以求出一个模糊的上界O(N * S) —> O(N^2)