研究生算法课课堂笔记

上课日期：2014.11.04　　第(3)节课

组长：韩硕　　**组员：黄高乐**

组员：何相腾　　**组员：周亚峰**

注意：请提交 Word 格式文档。

本节课主要对第二次课后作业中的习题进行了讲解。

作业中需要注意的一些问题

1. 对于需要给出算法的题目，要证明算法的正确性。证明不能只是简单地一两句话带过，应当给出清晰明了的证明过程。但也不宜过长。
2. 举反例时，应当尽量避免位于边缘状态的反例。

题目一

首先对事件 $x_1,x_2,\ldots x_n$ 按从小到大排序。然后顺序枚举 x_i，尽可能去匹配 t_j+e_j（即结束时间）最早的转账区间。若匹配过程中存在 x_i 没有找到匹配的区间，那么不存在完备匹配。算法伪代码如下:

```
sort x[1…n] with ascending order
for i from 1 to n
    find the smallest and unmatched t[j]+e[j] which satisfies |t[j]-x[i]|<=e[j]
    if not found legal t[j] then
        return NO_ANSWER
    end if
    match x[i] to t[j]
end for
```

可以使用“替换”的思路来证明该算法是正确的。首先，该算法保证了每个 x_i 所匹配的 t_j 满足|t[j]-x[i]|<=e[j]且是一一映射的关系，那么当该算法能够找到解时，这个解一定是合法的。假设存在一个合法解 S 但该算法未找到。令 $x_1,x_2,\ldots x_i$ 是该算法找到的匹配与合法解 S 中相同的连续最长的部分。对于 x_{i+1}，S 中匹配到了 t_a。不妨设该算法对于 x_{i+1} 匹配到了 t_b（显然该算法一定能找到一个 t_b 作为匹配，因为由 S 知至少 t_a 对于 x_i 是可以匹配的），t_b 在 S 中的匹配是 x_j（$x_j >= x_{i+1}$）。那么有：

$$t_a-e_a <= x_{i+1} <= x_j <= t_b+e_b <= t_a+e_a$$

那么在 S 中可以通过交换 x_{i+1} 和 x_j 的匹配来构造另一个合法解 S’。而 S’中从 x_1 一直到 x_{i+1} 的匹配与该算法都是一致的，原假设矛盾。

由伪代码的描述可知，排序部分需要 O(n*logn)，两重循环寻找匹配部分需要 O(n^2)，因此整个算法的时间复杂度为 O(n^2)。

注意到每次寻找最早的转账区间进行匹配时，如果对于每个 x_i 都去在转账区间中进行线型查找，那么将会存在大量重复计算。可以使用小根堆对这个过程进

行优化。先将区间按照 t_j-e_j 排序，枚举到 x_i 时，将 t_j-e_j<=x_i 的区间的 t_j+e_j（结束时间）压入堆中。然后弹出最小的 t_j+e_j 进行匹配。复杂度降到 O(n*logn)算法伪代码如下：

```
suppose heap is a Min-Heap sorted by t[i]+e[i]
sort t[1…n] by t[i]-e[i] with non-descending order
sort x[1…n] with non-descending order
j:=1
for i from 1 to n
    while j<=n and t[j]-e[j]<=x[i]
        heap.push(t[j]+e[j])
        j:=j+1
    end while
    if heap.isEmpty() then
        return NO_ANSWER
    end if
    T:=heap.pop()
    if T.endtime<x[i]
        return NO_ANSWER
    end if
    match x[i] to T
end for
```

题目二

这样的树一定存在。对于图 G 中的任何一个环，删掉环中边权最小的边（若有多个权重最小的边，任选一个即可），满足图中任意顶点对 u-v 的最佳可达瓶颈值不变。按此性质，直到将图 G 中所有的环消除，即得到符合要求的树 T。这一过程与删边法求图的最大生成树是一致的。事实上，图 G 的所有最大生成树 T 都满足题意要求。证明过程可以利用最大生成树的环性质。

最小生成树的环性质是指：对于原图中的任意一个环，环上的最大边不可能全在同一个最小生成树上。类似地，最大生成树的环性质是：对于原图中的任意一个环，环上的最小边不可能全在同一个最大生成树上。

对于任意一个最大生成树 T，均满足任意 u-v 路径的最佳瓶颈值与原图 G 一致。证明过程如下：假设存在一个顶点对 u-v 在 T 中的最佳瓶颈值比原图小。记 u-v 在 T 中的路径为 P，P 中的最小边集为{e}。在 G 中需要有一路径 P’，该路径上所有的边的边权均大于{e}。那么 P 与 P’中包含的边一定存在环，即 u-v 在 P 和 P’的并集上存在至少两条不相交路径。{e}是这个环上的最小边集。由最大生成树的环性质（环上的最小边不可能全在同一个最大生成树上）可知，{e}中不可能所有的边都在 T 上，即 T 不是最大生成树，这与假设矛盾。因此所有最大生成树 T 均满足题意要求。

因此可以使用 Kruskal、Prim 等算法求得图 G 的最大生成树 T。给出 Kruskal 求最大生成树的伪代码如下：

将连通图 G 的 m 条边按权重递减的顺序排序。

初始化边集 T=Ø
为 G 中的每一个顶点创建一个集合
While 所有顶点不属于同一个集合
    e = 当前尚未考虑的边中权重最大的边
    u=e 的左端点
    v=e 的右端点
    If u,v 属于不同的集合
        将 u,v 所在的集合合并
        将 e 添加入 T
    End If
End While
用集合 T 中的边构造出的树就是我们要求的生成树

题目三

可以通过 dijkstra 算法来求解从起点到终点最快的路，只需要在求每个点到起点的最短距离时同时记录该点在最短路上的的前驱顶点即可。伪代码如下：

suppose S to be the set of visited vertice,
d(u) records the earlist reach time of u in S, and p(u) records the last vertex on the fastest path to u.
initialize S:={s}, d(s):=0
while (t is not in S)
    for each v not in S
        calculate temp(v) = min{f_e(d(u)), u is in S}
    end for
    select v with the minimal temp(v) and update d(v), p(v)
    put v into S
end while
suppose P is the fastest path from s to t.
u:=t
while (u <> s)
    v := p(u)
    put edge v-u into P
    u:=v
end while
return d(t) and P

该算法的正确性可用数学归纳法证明。当|S|=1 时，S 只包含起点 s，d(s)=0 显然正确。假设当|S|=k>=1 时算法是正确的，需要证明当 S 加入点 v 使得|S|=k+1 时仍然保持正确性。对于 S 中的任一顶点 x，记 P(x)为从起点 s 到达 x 的最快路径。记 u-v 是从 s 到 v 最快路径 P(v)上的最后一条边。需要证明对于任意一条其他从 s 到 v 的路径 P，抵达 v 的时间都不早于 d(v)。对于这样的 P，记它第一个离开 S 的顶点为 b，b 在 P 上的前驱顶点为 a。在算法的第 k+1 次迭代时，满足 temp(b)>=temp(v)，即到达 b 的时间不会早于到 v。由于图中不存在“开始的迟而到达的更早”的情况，经过 P 到达 v 的时间不会早于 d(v)。伪代码所示的算法

的时间复杂度为 O(n^2)，可以使用堆优化至 O(m*log(n))。

课上对该问题有两个扩展：

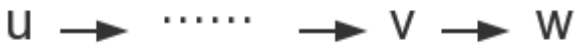
扩展一：如果允许在时间上向后旅行（但仍然满足不存在“后发先至”的情况），怎么解？

这种情况相当于图中存在负边，如果形成了负环，那么可以在负环中一直走下去。如果不存在负环可以使用 Bellman-Ford 算法求解。给出一个使用队列优化的 Bellman-Ford 算法（搞过竞赛的同学可能更习惯叫它 SPFA）。最坏情况时间复杂度为 O(N*M)。

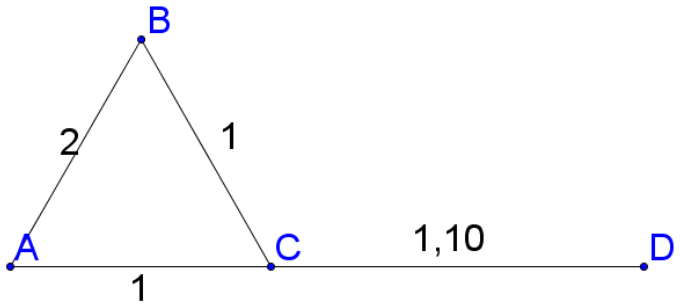
```
d(u) records the earliest reach time of u in S, p(u) records the last vertex on the
fastest path to u. Q is a queue which contains the updated vertices
initialize d(s):=0, and d(u):=+oo (when u<>s)
push s into Q
while Q is not empty
    u:= the front vertex in Q
    pop out the front vertex in Q
    for each u's neighbor vertex v
        if f_e(d(u))<d(v) then
            d(v):= f_e(d(u))
            p(v):=u
            if v is not in Q
                push v into the tail of Q
            end if
        end if
    end for
end while
suppose P is the fastest path from s to t.
u:=t
while (u <> s)
    v := p(u)
    put edge v-u into P
    u:=v
end while
return d(t) and P
```

扩展二：如果存在“后发先至”的情况，怎么解？

这样就变成了 NP 难的问题，只有列举所以可能的路径，找到最优解。注意使用 DP 也是不可解的，因为后发先至的情况下不存在最优子结构，不符合 DP 的要求。下图中 u, w 为起始点，v 为中间点。即使 u->v 不是最短路，u->w 也可能是最短路。



课上有同学尝试使用类似 dijkstra 的思路求解，是错误的，黄高乐同学给出了一个反例。



起点为 A，终点为 D。按照 dijkstra 算法，首先从 A 开始，找到 AC=1，AB=2。接着从 C 开始，找到 CB=1，不更新 AB，找到 CD=10，即 AD=11。接着从 B 开始，找到 BC=1，不更新 BC。最后找到 D，结束。

实际上，因为允许后发先至，可能存在从 B 经 C 到 D，只需要耗时 BC+CD=1+1，即从 B 后到 C，反而先到 D。所以算法是错误的。

如果函数 $f_e(t)$的值域中值的个数是有限的，可以先对所有的值进行离散化处理，然后可以采用 Bellman-Ford 算法求解。原算法中的 d(u)修改为 d(u,t)，表示时间 t 时能否到达顶点 u。最坏情况下时间复杂度为 O(N*T*M)，T 为所有时间点（即值域中的所有的值）的个数。不过仍为伪多项式时间。

题目四

(a)

对每个敏感进程的结束时间按照从小到大排序。然后按序枚举每个敏感进程，若当前进程未被任何一个 status_check 覆盖，那么在该进程的结束时间运行一次新的 status_check。时间复杂度为 O(n*logn)。

该算法找到的 status_check 集合显然能够将所有的敏感进程覆盖，现在需要证明这个集合是最小的。记该算法找到的 status_check 集合为 S，对于任意一个其他的合法解 S'，从初始时间到 S 中的第 k 个 status_check 运行这个时间区间内，S'也至少需要 k 个 status_check 覆盖。我们采用数学归纳法来证明其正确性。

首先，S'中的第一个 status_check 不晚于 S 中的第一个 status_check，否则第一个结束的进程将无法被覆盖。

假设初始时间到 S 中第 k 个 status_check 运行这个时间区间，S'中同样也需要 k 个 status_check 来覆盖。那么从初始时间到 S 中第 k+1 个 status_check 运行这个时间区间内，由于 S 中第 k+1 个 status_check 覆盖到的进程 p 与从初始时间到 S 中第 k 个 status_check 运行这个区间是相离的，那么进程 p 在 S'中至少需要一个 status_check 来覆盖。因此 S'中的第 k+1 个 status_check 不能晚于进程 p 的结束时间，即 S 中第 k+1 个 status_check 所在时间。因此对于从初始时间到 S 中第 k+1 个 status_check 运行这个时间区间，S'也至少需要 k+1 个 status_check。

(b)

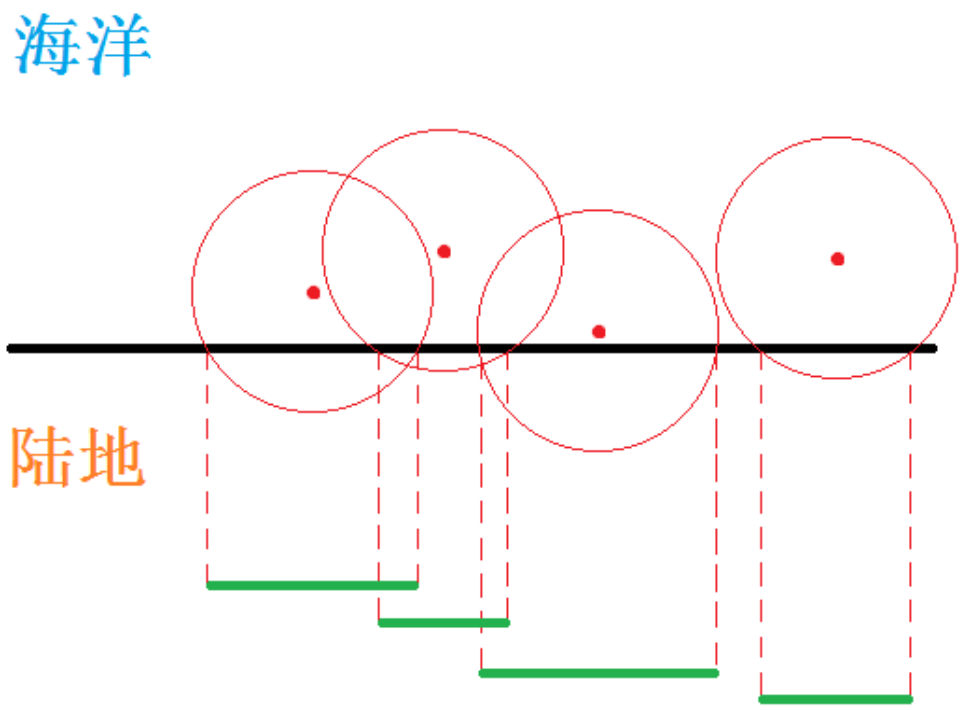
断言为真。$|S|>=k^*$是显然的，因为对于 k^*个不相交进程，至少需要 k^*个

status_check 来覆盖。对于(a)中给出的算法得到的 S，每个 status_check 的时间对应一个进程的结束时间，那么这些进程一定是两两不相交的。因为若存在两个进程是相交的，那么结束更晚的进程的结束时间处是不会新加入一个 status_check 的。而 k^*是最大的两两不相交进程的集合，则有 $k^*>=|S|$。综上可知 $k^*=|S|$。

扩展：

海面上有 n 个观察点，每个观察点到海岸线的垂直距离为 d1,d2,d3…dn，现在要在海岸上修建雷达基站，假设海岸线是一条直线，每个雷达基站可以接收到 D 范围内的观察点发来的信号。其中 D≥max(d1,d2,d3…dn)。要求最少修建多少个雷达基站，就能接收到所有观察点的信息。

首先考虑雷达基站的位置，要求将雷达基站建设在海岸上，但实际上建设在海岸线上可以最大程度地增加每个雷达基站所能监听到海面区域。因此这里我们考虑每个雷达基站都建立在海岸线的情况。对于每个观察点，它所能接受信号的范围是以该观察点为圆心，D 为半径所做的圆。该圆与海岸线相交的弦即是海岸线上能够监测到该观察点的区间。如下图所示。



于是该题转化成了在海岸线上建最少的雷达，能够覆盖所有的区间。接下来的做法就与原题一致了。

题目五

1. 反例如下所示：

$A_1[2\times4]A_2[4\times1]A_3[1\times2]A_4[2\times2]A_5[2\times4]$

如果按照该贪心策略来做的话，会按以下顺序计算：

①A_2 做划分点，$p_0p_2p_5=A_{12}A_{345}=A_{12345}$，乘法代价为 $2\times1\times4=8$

②A_1 做划分点，$p_0p_1p_2=A_1A_2=A_{12}$，乘法代价为 $2\times4\times1=8$

③A_3 做划分点，$p_2p_3p_5=A_3A_{45}=A_{345}$，乘法代价为 $1\times2\times4=8$

④A_4 做划分点，$p_3p_4p_5=A_4A_5=A_{45}$，乘法代价为 $2\times2\times4=16$

总的代价为 8+8+8+16=40

但是如果我们按下面划分顺序计算：

①A_4 做划分点，$p_0p_4p_5=A_{1234}A_5=A_{12345}$ 乘法代价为 $2\times2\times4=16$

②A_3 做划分点，$p_0p_3p_4=A_{123}A_4=A_{1234}$，乘法代价为 $2\times2\times2=8$

③A_2 做划分点，$p_0p_2p_3=A_{12}A_3=A_{123}$，乘法代价为 $2\times1\times2=4$

④A_1 做划分点，$p_0p_1p_2=A_1A_2=A_{12}$，乘法代价为 $2\times4\times1=8$

此时用的总代价为 8+4+8+16=36，显然比用该贪心策略的代价小。因此该贪心算法不能对所有实例求出最优解。

2. 反例如下所示：

$A_1[2\times4]A_2[4\times2]A_3[2\times2]A_4[2\times1]A_5[1\times4]$

如果按照该贪心策略来做的话，会按以下顺序计算：

①$A_3A_4=A_{34}[2\times1]$，乘法代价为 $2\times2\times1=4$

②$A_{34}A_5=A_{345}[2\times4]$，乘法代价为 $2\times1\times4=8$

③$A_1A_2=A_{12}[2\times2]$，乘法代价为 $2\times4\times2=16$

④$A_{12}A_{345}=A_{12345}[2\times4]$，乘法代价为 $2\times2\times4=16$

总的代价为 4+8+16+16=44

但是如果我们按正常顺序计算：

①$A_1A_2=A_{12}[2\times2]$，乘法代价为 $2\times4\times2=16$

②$A_{12}A_3=A_{123}[2\times2]$，乘法代价为 $2\times2\times2=8$

③$A_{123}A_4=A_{1234}[2\times1]$，乘法代价为 $2\times2\times1=4$

④$A_{1234}A_5=A_{12345}[2\times4]$ 乘法代价为 $2\times1\times4=8$

此时用的总代价为 16+8+4+8=36，显然此顺序比用该贪心策略的代价小。因此该贪心算法不能对所有实例求出最优解。

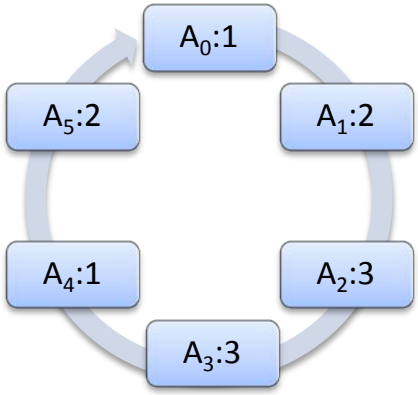
3. 这一道题目的反例很不好找。因为对于一个有 n 个数组的环来说，如果进行 n-1 次合并能得到一个最小合并次数的合并方法，那么肯定存在这样一条边，我们把这条边去掉后由原来的环形成的链拥有同样的合并方法也可以达到最小合并次数。因此这个贪心策略看起来非常像是正确的。但是如果在合并的某一过程中存在欺骗性的最小长度和相邻数组，我还是能找到反例的。分析过程如下所示：

(1) 我先找一个链结构的数组集$\{A_0,A_1,A_2\ldots A_n\}$，只有相邻的集合可以合并。假设这个链是 A_0 与 A_n 所构成的环最优的合并方式中没有用到的那条边去掉后得到的链。我先确定链的头部和尾部。所谓欺骗性，是指在环中，如果 A_0 跟 A_1、A_n 跟 A_{n-1} 分别合并就能得到正确最优解，但是 A_0 如果和 A_n 合并就不能得到最优解。为了满足欺骗性，$A_0+A_n=A_0+A_1=A_n+A_{n-1}$。

(2) 当我们把 A_0、A_1 和 A_{n-1} 和 A_n 确定好后，我们为了让 A_0 跟 A_1、A_{n-1} 和 A_n 能正常合并，需要 $A_2>A_0$，$A_{n-2}>A_n$。因此我令 $A_0=1$，$A_n=2$，那么 $A_1=2$，$A_{n-1}=1$。再令 $A_2=3$，$A_3=A_{n-2}=3$。因此我得到的链结构是：1，2，3，3，

1，2。经过验证，由这六个长度的数组相互临接构成的环，正好是一个反例。

反例如下所示：



按照题中此人的贪心算法，以上序列的合并过程可以为：

① $A_0+A_5=A_{50}$:3

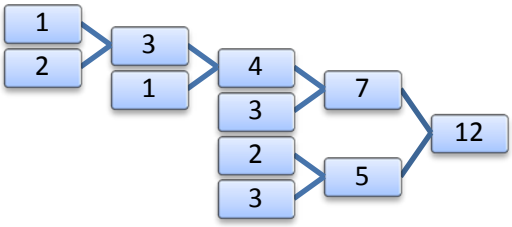
② $A_{50}+A_4=A_{450}$:4

③ $A_1+A_2=A_{12}$:5

④ $A_3+A_{450}=A_{3450}$:7

⑤ $A_{12}+A_{3450}=A_{123450}$:12

这种方法每一步都符合此贪心方法策略，最后算出的总代价为：31



(1+2)+(3+1)+(2+3)+(3+4)+(5+7) =31

然而，我能找到一种更好的合并方法，合并过程为：

① $A_0+A_1=A_{01}$:3

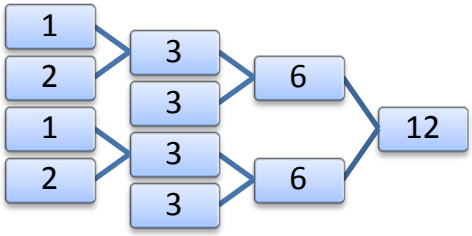
② $A_4+A_5=A_{45}$:3

③ $A_{01}+A_2=A_{012}$:6

④ $A_3+A_{45}=A_{345}$:6

⑤ $A_{012}+A_{345}=A_{012345}$:12

这种方法用的总代价仅为 30，比上面的方法少 1。



(1+2)+(1+2)+(3+3)+(3+3)+(6+6)=30

因此这种贪心算法不一定每次都对所有实例找到最优解。

正确的做法应该是先把 n 元环拆成长度为 2*n 的链，然后在链上做区间 DP。opt(i,j)表示区间 i 到 j 合并后所需的最小代价，那么:

opt(i,i)=0

opt(i,j)=min{opt(i,k)+opt(k+1,j)+sum(i,j)} (i<=k<j)

answer=min{opt(i,i+n-1)} (i+n-1<=2*n)

时间复杂度为 O(n^3)。迭代时需要注意子结构(i,j)的计算次序，搞不清楚的话可以使用“记忆化搜索”的方式。直接迭代方式的代码如下。

```
#include<cstdio>
#include<algorithm>
#define N 101
#define oo 0x3f3f3f3f
using namespace std;

int dp[N+N][N+N],sum[N];

int main()
{
    int t,n;

    scanf("%d",&t);
    while (t--)
    {
        scanf("%d",&n);
        for (int i=1;i<=n;i++)
        {
            scanf("%d",&sum[i]);
            sum[i+n]=sum[i];
        }

        sum[0]=0;
        for (int i=1;i<=n+n;i++)
            sum[i]+=sum[i-1];

        for (int i=1;i<=n+n;i++)
        {
            dp[i][i]=0;
            for (int j=i+1;j<n+n;j++)
                dp[i][j]=oo;
        }
        for (int j=1;j<=n+n;j++)
            for (int i=j-1;i>0&&j-i+1<=n;i--)
                for (int k=i;k<j;k++)
                    dp[i][j]=min(dp[i][j],dp[i][k]+dp[k+1][j]+sum[j]-sum[i-1]);

        int ans=oo;
        for (int i=1;i<=n;i++)
            ans=min(ans,dp[i][i+n-1]);

        printf("ans=%d\n",ans);
    }

    return 0;
}

```