2016 级算法第 次上机解题报告

15081070 张雨任

一、总结

本次上机主要考查贪心等简单的算法知识以及动态规划、最大流、二分图匹配等较难算法知识。

二、解题报告

A. Beihang Collegiate Pronunciation Contest 2017（签到题）

思路分析

本题考察字符串匹配问题。

算法分析

可以考虑使用 String 类的 substr 方法， str.substr (int pos, int len) 代表取从 str [pos] 开始的长度为 len 的子串，因此，线性遍历输入的字符串，查看从该字符开始的子串是否与目标串相等即可。时间复杂度为 O(n)。

参考代码

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
#include <string>
#define ll long long
using namespace std;

int n;
string s;

int main() {
```

```
    while(~scanf("%d",&n)) {
        cin>>s;
        for(int i=0;i<n;i++) {
            if(s.substr(i,7)=="AlvinZH") {
                printf("hg, shg, awsl!\n");
            }
            else if(s.substr(i,10)=="ModricWang") {
                printf("1080Ti!, wyr, silver!!!\n");
            }
            else if(s.substr(i,6)=="Bamboo") {
                printf("this is 51's father\n");
            }
            else if(s.substr(i,11)=="ConnorZhong") {
                printf("I am so weak\n");
            }
            else if(s.substr(i,4)=="BCPC") {
                printf("I want to join in!\n");
            }
        }
    }
}
```

B. Bamboo&APTX4844 魔发药水

思路分析

本题考察背包问题，先使用 01 背包，求解仅使用绿色能量的最大生发效果、以及此时已占用的瓶子的重量，再利用瓶子剩余重量使用分数背包。

算法分析

首先尽量装入整块的绿色能量，每种只有一粒，因此使用 01 背包的方法。本题的重点在于实现 01 背包后，计算瓶子剩余可供装入液体的重量，剩余重量是计算分数背包的前提。考虑使用 01 背包的变形来解决此问题。01 背包是给定多个物品的 weight 和 value，给定背包的承重上限，求能装入的 value 的最大值。这里要求承重的最大值，也就是能装入 weight 的最大值，因此可以把此处的 value 当作 weight，即可求出装入 value 最大值时，weight 的最大值，也就是绿色能量占用的重量。

接下来，对剩余的重量装入时光泉水，由于时光泉水是液体，可以拆分装入，因此使用分数背包的方法。

分数背包属于贪心算法，本质上就是优先选择“投入产出比”大的时光泉水，直到装满瓶子。“投入产出比”就是单位时光泉水重量所能提供的生发效果，由于时光泉水是液体，并且瓶子中剩余重量所能装入的时光泉水的重量是固定的，因此装入液体的平均“投入产出比”越大，总的生发效果就越大。所以优先选择“投入产出比”大的时光泉水。

所以算出每个时光泉水的“投入产出比”，优先选择大者，直到装满剩余的所有重量。

01 背包的时间复杂度为 O(KS) ，分数背包时间复杂度为 O(M)。

参考代码

```
#include <cstdio>

#include <cstring>
#include <algorithm>
using namespace std;

struct Wtr{
    double w;
    double v;
    double r;
```

```
}wtr[100007];

struct Gr{
    double w;
    double v;
}gr[100007];

double a1[100007];
double a2[100007];

bool cmp(Wtr a, Wtr b){
    return a.r>b.r;
}

int main() {
    int M,K,S;
    while(~scanf("%d%d%d",&M,&K,&S)) {
        memset(a1,0,sizeof(a1));
        memset(a2,0,sizeof(a2));
        for(int i=0;i<M;i++) {
            scanf("%lf%lf",&wtr[i].w,&wtr[i].v);
            wtr[i].r=wtr[i].v/wtr[i].w;
        }

        for(int i=0;i<K;i++) {
            scanf("%lf%lf",&gr[i].w,&gr[i].v);
        }
        for(int i=0;i<K;i++) {
            for(int j=S;j>=gr[i].w;j--) {
                if(a1[j-(int)gr[i].w]+gr[i].v>a1[j]) {
                    a1[j] = a1[j-(int)gr[i].w]+gr[i].v;
                    a2[j] = a2[j-(int)gr[i].w]+gr[i].w;
                }
            }
        }
        int i=0;
        double res=a1[S];
        double lf=S-a2[S];
        sort(wtr,wtr+M,cmp);
        while(lf && i<M){
            if(wtr[i].w<=lf){
```

```
            lf-=wtr[i].w;
            res+=wtr[i].v;
            i++;
        }
        else{
            res+=lf*wtr[i].r;
            break;
        }
    }
    printf("%.1f\n",res);
  }
}
```

C. Bamboo 和"Coco"

思路分析

本题考察动态规划，考察了动态规划的正推和反推。

算法分析

给定一个序列 data[0..n-1]代表思念值，每个思念值为 data[i]的亡灵对应一个花瓣数 a[i]，“保证若某个亡灵要比它邻近（前或后）的亡灵的思念值高，则其获得花瓣也要更多”。那么要保证若 data[i..j]单调递增，则 a[i..j]也单调递增；同样的，若 data[i..j]单调递减，则 a[i..j]也单调递减。

对于每个亡灵，最少拥有一个花瓣，为了保证亡灵 data[i]能够拿到最小且合法的情况，分别考虑单调递增和单调递减的情况：

首先考虑单调递增的情况。如果 data[i]>data[i-1]，则必须满足 a[i]>a[i-1]，为了使得 a[i]最小，且 a[i]是整数，所以 a[i]=a[i-1]+1，当不满足递增关系的时候，则把 a[i]置为 1。，因为该亡灵不需要比前一个亡灵拥有更多花瓣，因此赋予最小值即可。

同理，还要考虑单调递减的情况，正向的单调递减等价于反向的单调递增。因此从后往前，重复上述操作：如果 data[i]>data[i]+1，则必须满足 a[i]>a[i+1], 为了使得 a[i]最小，且 a[i]是整数，所以 a[i]=a[i+1]+1。同样，当不满足递减关系的时候，则把 a[i] 置为 1，因为该亡灵不需要比后一个亡灵拥有更多花瓣，因此赋予最小值即可。

因此每个 data[i]对应两个 a[i]的值，由于 a[i]要同时满足和 a[i-1]以及 a[i+1]的大小关系，因此每个 a[i]取较大值。最后累加所有的 a[i]即可。由于每个 a [i]取两个值，因此使用两个数组标记 a[0..n-1]更方便。

时间复杂度为 O(n)。

参考代码

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

int data[1000007];
```

```
int up[1000007];
int down[1000007];
int ans,n;

int main() {
    while(~scanf("%d",&n)) {
        ans=0;
        for(int i=0;i<n;i++) scanf("%d",&data[i]);
        up[0]=1;
        down[n-1]=1;
        for(int i=1;i<n;i++) {
            if(data[i]>data[i-1]) up[i]=up[i-1]+1;
            else up[i]=1;
        }
        for(int i=n-2;i>=0;i--) {
            if(data[i]>data[i+1]) down[i]=down[i+1]+1;
            else down[i]=1;
        }
        for(int i=0;i<n;i++) ans+=max(up[i],down[i]);
        printf("%d\n",ans);
    }
}
```

D. AlvinZH 的学霸养成记 III

思路分析

本题考查概率 DP。

算法分析

设 E[i]表示从班里从 i-1 个学霸变成 i 个学霸的期望天数，那么显然，若想使班里从 1 个学霸到 n 个学霸，计算天数的期望，需要计算 E[2]+E[3]+…+E[n]。

那么计算每个期望 E[i]的值，需要计算从 i-1 个学霸变成 i 个学霸的概率 P0。那么就变成一个数学问题了。

从 n 个人中，挑出 2 个人去自习，总共有 C(2,n)种选择方法。在计算 E[i]时，班里学霸有 i-1 人，学渣有 n-i+1 人，那么当且仅当挑出一学霸一学渣时，才有可能达成从 i-1 个学霸变成 i 个学霸。i-1 人挑出 1 学霸，有 C(1,i-1)种挑法。n-i+1 人挑出 1 学渣，有 C(1,n+1-i)种挑法。因此从 n 个人中，挑出 2 个人去自习，有 C(1,i-1)* C(1,n+1-i)/ C(2,n)的可能性调到一学霸一学渣的组合。那么再乘以转换成两学霸的概率 p，则有 P0= p*C(1,i-1)* C(1,n+1-i)/ C(2,n)，化简得到 P0=2*p*(i-1)*(n-i+1)/(n*(n-1))。

从 i-1 个学霸变成 i 个学霸的概率 P0，那么达成此目的的期望天数 E[i]=1/P0。

同理可以算出 E[2...n]，算出 sum(E[i]), i=2…n 即可。

时间复杂度 O(n)。

参考代码

```
#include <cstdio>

#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
#define ll long long
using namespace std;

double ans,E,p,n,P0;
int N;
```

```
int main() {
    while(~scanf("%d",&N)) {
        for(int i=0;i<N;i++) {
            scanf("%lf%lf",&n,&p);
            ans=0.0;
            for(int i=2;i<=n;i++) {
                P0=2.0*p*(i-1)*(n-i+1)/(n*(n-1));
                E=1/P0;
                ans+=E;
            }
            printf("%.3f\n",ans);
        }
    }
}
```

E.

思路分析

本题考查最大二分匹配问题，可以使用匈牙利算法解决。

算法分析

首先给这道题的“匹配”下个定义。“匹配”就是我方某随从 j 可以在他不死亡的情况下，杀死对方的随从 i，在这样的条件下，有 r[j][i]=1。那么同时遍历我方随从和对方随从，根据我方随从不同的属性值计算是否“匹配”。

属性值为 0，则匹配成立当且仅当我方随从杀死对方，且不被对方杀死。因此，需要满足我方攻击力大于等于对方生命值，且我方生命值大于对方攻击力。

属性值为 1，则匹配成立当且仅当我方随从杀死对方，因为我方“免疫”，不会受到对方伤害，没有死亡的风险。因此，需要满足我方攻击力大于等于对方生命值。

属性值为 2，则匹配成立当且仅当我方随从不被对方杀死，因为我方“剧毒”，可以直接杀死对方，只要保证不被对方杀死即可。因此，需要满足我方生命值大于对方攻击力。

属性值为 3，则匹配恒成立，因为我方可直接杀死对方，且不会被对方杀死。

这样就得到了从 j 到 i 的二分图映射 r[j][i]，若为 1，则 j 到 i 为匹配。

接下来，使用匈牙利算法解决最大二分匹配问题。

匈牙利算法本质上与最大流 Edmonds-Kart 算法类似，主要思路是寻找增广路径。定义增广路径为：若 P 是图 G 中一条连通两个未匹配顶点的路径，并且已匹配和待匹配的边在 P 上交替出现，则称 P 为相对于 M 的一条增广路径。

为什么寻找增广路径的方法可以解决最大二分匹配问题？

由增广路径的性质，增广路径的第一条边和最后一条边都是未匹配边，所以增广路径中的已匹配边总是比未匹配边多一条，所以如果我们放弃一条增广路径中的匹配边，选取未匹配边作为已匹配边，则匹配的数量就会增加。匈牙利算法就是在不断寻找增广路径，如果找不到增广路径，就说明达到了最大匹配。

使用两个数组来标记是否已匹配。cx[i]=j 代表与 x 集合（起点集合）元素 i 匹配的是 j，cx[i]=-1 代表 x 集合（起点集合）元素 i 还未匹配。同理，cy[j]=i 代表与 y 集合（终点集合）元素 j 匹配的是 i，cy[j]=-1 代表 y 集合（终点集合）元素 j 还未匹配。

增广路径的寻找可以用递归来描述：“从点 i 出发的增广路径”，首先连向一个在原匹配中没有与点 i 配对的点 j，因此保证增广路径的第一条边是未匹配边（即 cx[i]=-1）。如果点 j 在原匹配中没有与任何点配对，则它就是这条增广路径的终点；反之，如果点 j 已与点 k 配对，那么这条增广路径就是从 i 到 j，再从 j 到 k，再加上“从点 k 出发的增广路径”。并且，这条从 k 出发的增广路径中不能与前半部分的增广路径有重复的点。这样，就能保证已匹配边和待匹配边在增广路径中交替出现。

假设 x、y 集合各有 n 个点。判断“匹配”的时间复杂度为 O(n^2)。匈牙利算法的时间复杂度为 O(nm)。遍历 x 集合的每个点，时间复杂度为 O(n)。假设 x、y 集合产生 m 个边，所以每个点寻找到的增广路径最长有 m 个边，时间复杂度为 O(m)。所以此题匈牙利算法的时间复杂度为 O(nm)。

参考代码（匈牙利算法）

```
#include<iostream>
#include<cstdio>
#include<cstring>
using namespace std;

const int maxn=1007;
int r[maxn][maxn];
int visit[maxn];
int cx[maxn],cy[maxn];
int n,eA,eL,ans;

struct Me {
    int A;
    int L;
    int P;
}m[1007];

int DFS(int u) {
    int v;
    for(v=1;v<=1000;v++) {
        if(r[u][v] && !visit[v]) {
            visit[v]=1;
            if(cy[v]==-1 || DFS(cy[v])) {
                cy[v]=u;
                cx[u]=v;
```

```
                return 1;
            }
        }
    }
    return 0;
}

int Hungarian() {
    int cnt=0;
    for(int i=1;i<=1000;i++) {
        if(cx[i]==-1) {
            memset(visit,0,sizeof(visit));
            cnt+=DFS(i);
        }
    }
    return cnt;
}

int main() {
    while(~scanf("%d",&n)) {
        memset(r,0,sizeof(r));
        memset(cy,-1,sizeof(cy));
        memset(cx,-1,sizeof(cx));
        for(int i=1;i<=n;i++) {
            scanf("%d%d%d",&m[i].A,&m[i].L,&m[i].P);
        }
        for(int i=1;i<=n;i++) {
            scanf("%d%d",&eA,&eL);
            for(int j=1;j<=n;j++) {
                if(m[j].P==0) {
                    if(m[j].A>=eL && m[j].L>eA) {
                        r[j][i]=1;
                    }
                }
                else if(m[j].P==1) {
                    if(m[j].A>=eL) {
                        r[j][i]=1;
                    }
                }
                else if(m[j].P==2) {
                    if(m[j].L>eA) {
```

```
                    r[j][i]=1;
                }
            }
            else if(m[j].P==3) {
                r[j][i]=1;
            }
        }
    }
    ans=Hungarian();
    if(ans==n) printf("YES\n");
    else printf("NO\n");
}
}
```

F. ModricWang 的水系法术

思路分析

本题考查最大流的知识点，可以通过 Edmonds-Karp 算法或 Dinic 算法解决。

要额外注意，这道题中，水可以往两个方向流，因此输入的相当于是一个无向图，所以初始状态下，点 a 到 b 的容量等于点 b 到 a 的容量，即 r[a][b]=r[b][a]=c。

算法分析

Edmonds-Karp 算法就是使用 BFS 寻找增广路径的 Ford-Fulkerson 方法。在 Ford-Fulkerson 方法的每次迭代中，寻找某条增广路径 p，只要存在增广路径，就在该路径 p 上选取最短的流作为 delta，然后使用 delta 来对流 f 进行增加，获得一个大小为 f+delta 的流。然后对于增广路径上每条边的流进行更新：如果残存边是原来网络中的一条边，则增加 delta 流量，否则缩减 delta 流量。当不再有增广路径时，流 f 就是最大流。

时间复杂度为 O(VE^2)，其中 V 代表点数，E 代表边数。每次用 delta 更新 f 的时间复杂度为 O(E)，因为一条增广路径最多有 E 条边要遍历。由《算法导论》的定理 26.8 可知，如果 Edmonds-Kart 算法运行在源结点为 s 汇点为 t 的流网络 G=(V, E)上，则该算法所执行的流量递增操作的总次数为 O(VE)。因此，Edmonds-Kart 算法的时间复杂度为 O(VE^2)。

Dinic 算法是上机之后学到的，推荐一个博客，讲解得很清楚：http://www.cnblogs.com/LUO77/p/6115057.html

Dinic 算法的 AC 代码附在文末。

参考代码一（Edmonds-Kart）

```
#include <iostream>

#include <queue>
#include <algorithm>
#include <cstring>
#include <cstdio>
#define ll long long
using namespace std;
const int maxn=1007;

const int inf=0x3f3f3f3f;
```

```
int N,M;
int r[maxn][maxn];
int pre[maxn];
bool vis[maxn];

bool BFS(int s,int t) {
    queue<int> que;
    memset(pre,-1,sizeof(pre));
    memset(vis,false,sizeof(vis));
    pre[s]=s;
    vis[s]=true;
    que.push(s);
    int p;
    while(!que.empty()) {
        p=que.front();
        que.pop();
        for(int i=1;i<=M;++i) {
            if(r[p][i]>0 && !vis[i]) {
                pre[i]=p;
                vis[i]=true;
                if(i==t) return true;
                que.push(i);
            }
        }
    }
    return false;
}

int EK(int s,int t) {
    int maxflow=0,d;
    while(BFS(s,t)) {//O(VE)
        d=inf;
        for(int i=t;i!=s;i=pre[i]) {
            d=min(d,r[pre[i]][i]);
        }
        for(int i=t;i!=s;i=pre[i]) {
            r[pre[i]][i]-=d;
            r[i][pre[i]]+=d;
        }
        maxflow+=d;
    }
```

```
    return maxflow;
}

int main() {
    while(~scanf("%d%d",&M,&N)) {
        memset(r,0,sizeof(r));
        int s,e,c;
        for(int i=0;i<N;++i) {
            scanf("%d%d%d",&s,&e,&c);
            r[s][e]+=c;
            r[e][s]+=c;
        }
        printf("%d\n",EK(1,M));
    }
    return 0;
}
```

参考代码二（Dinic）

```
#include <cstdio>
#include <queue>
#include <cstring>
using namespace std;

const int inf=0x7fffffff;
int M,N;
int level[1007];
int a,b,c;

struct Dinic {
    int c;
    int f;
}edge[1007][1007];

bool dinic_bfs() {
    queue<int> q;
    memset(level,0,sizeof(level));
    q.push(1);
    level[1]=1;
    int u,v;
```

```
    while(!q.empty()) {
        u=q.front();
        q.pop();
        for(v=1;v<=N;v++) {
            if(!level[v] && edge[u][v].c>edge[u][v].f) {
                level[v]=level[u]+1;
                q.push(v);
            }
        }
    }
    return level[N]!=0;
}

int dinic_dfs(int u,int cnt) {
    int rt=cnt;
    int v,t;
    if(u==N) return cnt;
    for(v=1;v<=N && rt;v++) {
        if(level[u]+1==level[v]) {
            if(edge[u][v].c>edge[u][v].f) {
                t=dinic_dfs(v,min(rt,edge[u][v].c-edge[u][v].f));
                edge[u][v].f+=t;
                edge[v][u].f-=t;
                rt-=t;
            }
        }
    }
    return cnt-rt;
}
int dinic() {
    int sum=0;
    int cnt=0;
    while(dinic_bfs()) {
        while(cnt=dinic_dfs(1,inf)) sum+=cnt;
    }
    return sum;
}

int main() {
    while(~scanf("%d%d",&N,&M)) {
        memset(edge,0,sizeof(edge));
```

```
        while(M--) {
            scanf("%d%d%d",&a,&b,&c);
            edge[a][b].c+=c;
            edge[b][a].c+=c;
        }
        printf("%d\n",dinic());
    }
}
```

G. ModricWang 的撒币游戏

思路分析

本题考查概率 DP。

算法分析

建立 DP 模型。考虑引入数组 dp[0..m][0..n]，设 dp[i][j]表示第 i 次抛硬币时 n 个硬币中有 j 个向上的概率。

考虑最优撒币策略的定义。最优撒币策略就是尽量抛起向下的硬币，因为抛起向上的硬币可能会导致该硬币在投掷种变为向下，导致失去本应得到的这枚硬币。

每次抛起 k 个硬币，尽量使这 k 个硬币都是向下的状态，对于状态 dp[i][j]（第 i 次抛硬币时，n 个硬币中有 j 个向上、有 n-j 个硬币向下），有两种情况：

第 i 次抛起的硬币数量 k<=n-j，换句话说，此次抛起的都是向下的，以保证最优撒币策略。抛起的硬币总共 k 个，设向上的硬币有 t 个，那么有 1<=t<=k，从而导致此时向上的硬币增加了 t 个，在第 i+1 次抛起时，总的向上硬币个数为 j+t 个，因此 dp[i][j]转移到 dp[i+1][j+t]。抛起 k 个硬币有 t 个向上的概率服从二项分布，即 p=com(k,t)*(1/2)^k。所以有 dp[i+1][j+t]+=p*dp[i][j]。

第 i 次抛起的硬币数量 k>n-j，则抛起 k 个硬币不能保证所有都是向下的，因此抛起了 n-j 个向下的硬币和 k-n+j 个向上的硬币。同样地，抛起的硬币总共 k 个，设向上的硬币有 t 个，那么向下的硬币有 k-t 个（1<=t<=k）。 可以看到，这次硬币抛起，变化（增加或减少）了(k-t)-(k-n+j)=t-k+n-j 个硬币，因此在第 i+1 次抛起硬币时，总的向上的硬币个数为 j+(t-k+n-j)=t-k+n 个，因此 dp[i][j]转移到 dp[i+1][t-k+n]。同样，抛起 k 个硬币有 t 个向上的概率服从二项分布，即 p=com(k,t)*(1/2)^k。所以有 dp[i+1][t-k+n]+=p*dp[i][j]。

初始条件为 dp[0][0]，第 0 次抛起且没有硬币向上就是初始条件，概率为 100%，因此 dp[0][0]=1.0。

本题算法时间复杂度为 O(Tmnk)。计算组合数的时间复杂度为 O(n^2)。计算 2 的幂的时间复杂度为 O(n)。

参考代码

```
#include <cstdio>
```

```cpp
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
#define ll long long
using namespace std;

const int maxn=107;
int n,m,k,T;
double dp[maxn][maxn];
double c[maxn][maxn];
double power[maxn];
double ans;

int main() {
    for(int i=0;i<=maxn-3;i++) {
        c[i][0]=1.0;
    }
    for(int i=1;i<=maxn-3;i++) {
        for(int j=1;j<=i;j++) {
            c[i][j]=c[i-1][j]+c[i-1][j-1];
        }
    }
    power[0]=1.0;
    for(int i=1;i<=maxn-3;i++) {
        power[i]=power[i-1]*2.0;
    }
    scanf("%d",&T);
    while(T--) {
        scanf("%d%d%d",&n,&m,&k);
        memset(dp,0,sizeof(dp));
        dp[0][0]=1.0;
        for(int i=0;i<=m;i++) {
            for(int j=0;j<=n;j++) {
                for(int t=0;t<=k;t++) {
                    if(k<=n-j) {
                        dp[i+1][j+t]+=c[k][t]/power[k]*dp[i][j];
                    }
                    else {
                        dp[i+1][t-k+n]+=c[k][t]/power[k]*dp[i][j];
                    }
```

```
                }
            }
        }
        ans=0.0;
        for(int i=1;i<=n;i++) {
            ans+=(dp[m][i]*i);
        }
        printf("%.3f\n",ans);
    }
}
```