A - Betygsättning

Här gäller det bara att göra det som står. Ett smidigt sätt att implementera är att gå igenom betygen från `E` till `A`, och för varje betyg kolla ifall man inte uppfyller nästa betygs krav. I så fall skriver vi ut betyget som svar, annars går vi vidare.

I vissa språk kan det bli problem med då man dividerar med 2, eftersom det avrundas nedåt till närmsta heltal. Om man ska kolla ifall $y > b/2$ kan det därför vara säkrare att kolla $2y > b$.

Exempellösning i python:

```
A,C,E = map(int, input().split())
a,c,e = map(int, input().split())

if c*2 < C:
   print('E')
elif c != C:
   print('D')
elif a*2 < A:
   print('C')
elif a != A:
   print('B')
else:
   print('A')
```

B - Volleybollmatchen

Här gäller det bara att simulera matchen. Vi håller koll på respektive lags poäng och går igenom poängen en för en. Ifall vi ser bokstaven `A` lägger vi till ett på Algoritmikernas poäng, och ifall vi ser bokstaven `B` lägger vi till ett på Bäverbusarnas poäng. Så fort vi uppdaterat poängen kollar vi ifall något av lagen vunnit settet, vilket görs genom att kolla ifall något lags poäng är större eller lika med winScore och dessutom minst två poäng mer än motståndarlaget. Värdet på variablen winScore ändrar vi beroende på vilket set som spelas just nu. Vi håller koll på hur många set respektive lag vunnit, och när något lag nått 2 vunna set skriver vi ut resultatet.

Exempellösning i c++:

```
#include <iostream>

using namespace std;

int main(){
```

```
    cin.sync_with_stdio(false); cin.tie(0);

    int n;
    cin>>n;
    int aScore = 0;
    int bScore = 0;
    int aSet = 0;
    int bSet = 0;
    for(int i = 0; i<n;i++){
        char c;
        cin>>c;

        if(c=='A') aScore++;
        if(c=='B') bScore++;

        int winScore;
        if(aSet+bSet==2) winScore = 15;
        else winScore = 25;

        if(aScore>=winScore&&aScore>=2+bScore){
            aSet++;
            aScore=0;
            bScore=0;
        }
        if(bScore>=winScore&&bScore>=2+aScore){
            bSet++;
            aScore=0;
            bScore=0;
        }
    }

    cout<<aSet<<"␣"<<bSet<<endl;
}
```

C - Skolavslutningen

Subtask 3 ($N, M \leq 50$)

Man kan se uppställningen som en graf där varje elev är en nod och en kant mellan två elever betyder att dessa måste ha samma hattfärg. Det enda kravet på hattfärg blir då att alla elever som tillhör samma komponent måste ha samma hattfärg, så man kan maximera antalet unika hattfärger genom att tilldela varje komponent en unik färg. Svaret blir alltså antalet komponenter i grafen. För

att räkna antal komponenter i en oriktad graf kan vi helt enkelt köra en DFS från en godtycklig nod, markera alla noder vi nådde som "besökta", välja en ny nod som inte redan är besökt att börja ifrån och upprepa. Antal gånger vi upprepar algoritmen är hur många komponenter grafen har.

För att bygga grafen kan man först lägga till kanter mellan alla noder i samma kolumn och sedan iterera över alla par av elever och lägga till kanter för de som tillhör samma klass. Tidskomplexitet för att bygga grafen blir $\mathcal{O}((NM)^2)$. Att räkna antalet komponenter kan göras med djupet/bredden först sökning i $\mathcal{O}(V + E)$, där V är antalet noder i grafen och E är antalet kanter. Den totala tidskomplexiteten blir $\mathcal{O}((NM)^2)$.

Subtask 4 ($N, M \leq 700$)

För 100 poäng behöver man bygga grafen på ett mer optimerat sätt. Exempelvis kan man istället för att lägga till kanter mellan alla par av elever i samma klass endast lägga till kanter från en av eleverna till de resterande. Detta kan man göra genom att för varje klass hålla koll på vilken nod som motsvarar den första eleven och för de resterande eleverna lägga till en kant mellan denna och den första. Om man ansluter noder i samma kolumn på liknande sätt (till exempel ansluter alla noder till första radens nod) blir tidskomplexiteten $\mathcal{O}(NM)$.

Här är en exempellösning i c++:

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

//För varje par av klasser i och j
//är adj[i][j] = true ifall det går en
//kant mellan klasss i och klass j i grafen
bool adj[26][26];

bool visited[26];

int N, M, K;

void dfs(int _class) {
   if (visited[_class])
      return;
   visited[_class] = true;

   for (int nc = 0; nc < K; nc++) {
      if (adj[_class][nc]) {
```

```
            dfs(nc);
        }
    }
}

int main() {
    cin >> N >> M >> K;

    vector<string> rows(N);
    for (int r = 0; r < N; r++) {
        cin >> rows[r];
    }

    for (int c = 0; c < M; c++) {
        int cl0 = rows[0][c] - 'A';
        for (int r = 0; r < N; r++) {
            int cl1 = rows[r][c] - 'A';
            adj[cl0][cl1] = true;
            adj[cl1][cl0] = true;
        }
    }

    //Räkna komponenter:
    int numComponents = 0;
    for (int i = 0; i < K; i++) {
        if (!visited[i]) {
            numComponents++;
            dfs(i);
        }
    }

    cout << numComponents << endl;
}
```

D - Eldberget

Ifall $k = 0$ är uppgiften att helt enkelt hitta kortaste vägen i en graf, där en ruta (som inte har eld på sig) i rutnätet är en nod och en kant går mellan intilliggande rutor. Standardalgoritmen för att hitta kortaste vägen mellan ett par av punkter i en graf är "bredden-först-sökning", BFS, och den fungerar alldeles utmärkt här. Algoritmen går ut på att först hitta alla noder på avstånd 1 till startnoden (alla dess grannar), sedan alla noder på avstånd 2 (alla grannar till noderna på avstånd 1), sedan alla noder på avstånd 3 (alla grannar till noderna på avstånd 2 som vi inte redan gett ett avstånd till), och så vidare.

Man kommer alltså att ”processa” alla noder i ordning efter deras avstånd till startnoden. Ett smidigt sätt att implementera detta på är att ha en kö med alla noder som ska processas. När vi processar en nod så lägger vi in dess grannar (de som inte redan processats) längst bak i kön, tillsammans med de avståndet från startnoden de är på. När vi kommer till slutnoden (rutan längst ned i högra hörnet) så skriver vi bara ut avståndet vi är på.

När $k > 0$ kan vi fortfarande göra BFS, men vi måste vara lite smartare. Man kan tänka att vi kan vara vid en viss nod på k olika sätt, beroende på hur många eldar vi hittills gått igenom. Vi kan se det som att vi gör k stycken kopior av varje nod, en för varje antal eldar man gått genom innan, och gör en BFS på den nya grafen vi får. Man kan även tänka på det som att vi lägger till en till dimension till rutnätet. För en ruta utan eld på så säger vi att den har kanter som går till alla dess grannar i samma lager i det 3-dimensionella rutnätet, och för en ruta med eld så går det kanter till alla grannar i lagret ovanför (eftersom man gått igenom en mer eld).

Nu kör vi BFS:en som vanligt på den här grafen. När vi processar en nod med eld som är på översta lagret (d.v.s har redan gått genom k stycken eldar) så ignorerar vi den, då vi inte kan fortsätta gå någonstans därifrån. Ifall vi aldrig i BFS kommer till slutnoden är svaret -1;

Exempellösning i c++:

```
#include <iostream>
#include <vector>
#include <queue>
#include <tuple>

using namespace std;

int main(){
    cin.sync_with_stdio(false);
    int r,c,k;
    cin>>r>>c>>k;
    vector<vector<bool>> fire(r,vector<bool>(c));
    for(int i = 0; i<r; i++)
        for(int j = 0; j<c; j++){
            char c;
            cin>>c;
            fire[i][j] = c=='#';
        }
    if(r+c<=k){
        cout<<r-1+c-1<<endl;
        return 0;
    }

    queue<tuple<int,int,int,int>> q;
```

```
    q.emplace(0,0,0,0);

    //Det här är den 3-dimensionella matrisen där vi hå
        ller koll på om vi redan processat en nod eller
        inte
    vector<vector<vector<bool>>> seen(r,vector<vector<
        bool>>(c,vector<bool>(r+c)));
    while(q.size()>0){ //Så länge det finns noder att
        processa
        int x,y,fires,distance;
        tie(x,y,fires,distance) = q.front();
        q.pop();
        if(x<0||x>=r||y<0||y>=c) continue; //Ifall vi är
            utanför rutnätet, ignorera denna och gå
           vidare

        fires += fire[x][y];

        if(fires>k||seen[x][y][fires]) continue; //Ifall
            vi har gått genom för många eldar eller
           redan har processat rutan, ignorera denna och
            gå vidare

        seen[x][y][fires] = true;

        if(x==r-1&&y==c-1){ //Ifall vi är vid mål
            cout<<distance<<endl;
            return 0;
        }

        //Lägg till alla grannar i kön
        q.emplace(x,   y-1, fires, distance+1);
        q.emplace(x+1, y,   fires, distance+1);
        q.emplace(x,   y+1, fires, distance+1);
        q.emplace(x-1, y,   fires, distance+1);
    }

    //Ifall vi kommer hit kunde vi aldrig nå slutnoden
    cout<<"nej"<<endl;
}
```

E - Brevoptimering

Vi vill för varje person lista ut hur många brev som skickas till den per sekund, I_b. Låt W_{ab} beteckna andelen av sina brev person a skickar till person b. Värdet I_b kan vi då räkna ut som summan av $U_a \cdot W_{ab}$ för alla personer a som skickar något till person b. Om vi nu definerar en funktion `getOutput(v)` som ger person v:s output, kan vi räkna ut värdet på den den rekursivt, genom att anropa `getOutput(u)` för alla personer u som skickar något till v.

För att detta ska bli tillräckligt snabbt krävs dock en viktigt insikt: vi behöver inte räkna ut `getOutput(v)` mer än en gång för varje person v. Så fort vi kommit till slutet att `getOutput(v)` för en viss person v, sparar vi svaret i som `output[v]` i en global array `output` innan vi returnerar det. I början av funktionen `getOutput(v)` kollar vi nu ifall värdet redan är uträknat, och returnerar det i så fall direkt. Att komma ihåg värden för en rekursiv funktion för att slippa räkna ut dem igen kallas för "memoization". Med den här optimeringen blir tidskomplexiteten $O(N + S)$, eftersom vi kommer att gå igenom varje kant i grafen exakt en gång, och varje person exakt en gång.

Exempellösning i c++:

```
#include <iostream>
#include <vector>
#include <string>
#include <utility>

using namespace std;

vector<vector<pair<int,double>>> gettingLettersFrom;
//Här sparar vi för varje person v en lista av alla
   personer u den får brev från,
//tillsammans med ett flyttal som beskriver andelen av
    u:s brev som den skickar till v

vector<double> M;
vector<double> output;

double getOutput(int v){
   if(output[v]!=-1) { //Ifall output inte är -1 har
      vi redan räknat ut den
      return output[v];
   }
   if(gettingLettersFrom[v].size()==0) return M[v];

   double in = 0;
   for(int i = 0; i<(int)gettingLettersFrom[v].size();
       i++){
```

```
        int u = gettingLettersFrom[v][i].first;
        double share = gettingLettersFrom[v][i].second;

        in += getOutput(u)*share;
    }

    double out = min(in,M[v]);

    return output[v] = out; //spara värdet i output och
        returnera det
}

int main(){
    cin.sync_with_stdio(false);
    int n;
    cin>>n;
    gettingLettersFrom.resize(n);
    M.resize(n);
    output.resize(n,-1); //Vi sätter -1 som defaultvä
        rde innan det riktiga värdet är uträknat

    for(int i = 0; i<n;i++) {
        int k;
        cin>>M[i]>>k;
        for(int l = 0; l<k;l++) {
            int j;
            double w;
            cin>>j>>w;
            gettingLettersFrom[j-1].emplace_back(i,w
                /100.0);
        }
    }

    for (int i = 0; i < n; i++) {
        if(getOutput(i)==M[i]) cout<<i+1<<"␣";
    }
    cout<<endl;
}
```

F - Matbeställning

Vi vill visa att följande giriga algoritm alltid ger bästa svaret: “Betrakta alla möjliga maträttsbyten som går att göra, ta det billigaste och utför det och upprepa sedan”. Vi menar att ett byte bara är möjligt ifall personen som byter

just nu har valt en rätt som någon annan också valt, och rätten som personen byter till har just nu ingen valt. Ifall detta inte är uppfyllt är ju bytet helt onödigt. Som vanlig med giriga algoritmer är det ganska intuitivt varför det fungerar, men vi ska här försöka visa det mer rigoröst.

Algoritmen kan bara gå fel ifall vi genom att ta den billigaste på något sätt förstör för en alternativ konfiguration som hade varit billigare. Det enda "negativa" med att göra detta byte är att personen vi byter och rätten vi byter till inte kan användas igen, så vi behöver bara betrakta de potentiella alternativa bytena som använder dessa, och se att kostnaden inte blir bättre. Säg att vi har fyra rätter a, b, c och d så att $c_a \leq c_b \leq c_c \leq c_d$, och säg att det billigaste valet just nu är att låta en person byta från en rätt b till rätt c. Att någon annan ska byta till rätt c, säg från en rätt a, och att personen som köpte b ska byta till någon annan rätt, säg d, kostar exakt $(c_c - c_a) + (c_d - c_b) = (c_d - c_a) + (c_c - c_b)$. Det är exakt lika mycket som det kostar att utföra bytet $b \to c$ och sedan $d \to a$. Vi behöver inte betrakta fallet där någon av c_a eller c_d ligger mellan c_b och c_c, eftersom då är inte $c_b \to c_c$ det billigaste Alltså kan det aldrig vara suboptimalt att göra detta billigaste byte $b \to c$.

Nu har vi direkt en lösning som kör i $O(km^2+n)$ tid, eller möjligtvis $O(km+n)$, som går ut på att k gånger leta upp det minsta bytet som går att utföra, utföra det och upprepa. För att få en $O(m \log m + n)$-lösning behövs lite fler insikter.

Vi tänker att vi vill skapa en lista över alla byten som görs när man följer algoritmen ovan, och sedan välja ut de k billigaste bland dessa. Den första insikten som behövs är att vi kan sortera alla rätter och gå igenom dem från dyrast till billigast, och ifall vi kommer till en dubbelbokad rätt parar vi ihop de överflödiga personerna där med den billigaste dyrare rätten. För att bevisa att detta alltid ger de byten vi vill ha använder vi igen argumentet om när det kan gå fel. Här kan det bara gå fel ifall vi genom att para ihop en person i med en rätt j den ska byta till förhindrar ett billigare byte från att ske. Men i kan ju inte ha ett billigare byte, eftersom den paras ihop med närmaste rätten, och j kan inte ha ett billigare byte, eftersom alla dubbelbokade rätter mellan i och j redan har parats ihop med en ledig rätt som ligger närmre sig själv i pris än i (annars skulle ju inte i vara ledig).

Den andra insikten som behövs är att vi kan implementera detta snabbt och smidigt genom att ha en stack som hela tiden innehåller alla "lediga" rätter (de som ingen valt från början och som vi inte än parat ihop), sorterad så att billigare rätter ligger högre upp i stacken. Så fort vi kommer till en ny ledig rätt bara lägger vi på den på stacken, och när vi kommer till en dubbelbokad rätt så tar vi och parar ihop alla personer som valt den rätten (förutom en) med de översta rätterna i stacken. Då de översta i stacken var de som lades till senast så kommer de vara exakt de billigaste rätterna som man kan byta till som inte redan någon bytt till.

Eftersom varje rätt bara läggs på i stacken och plockas bort från stacken maximalt en gång går lösningen linjärt i antalet rätter, bortsett steget att sortera

alla rätter, vilket är $O(m \log m)$. Även genomgången av personerna går i linjär tid. Tidskomplexiteten är alltså $O(m \log m + n)$. Här är en exempellösning i python:

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

meals = [[0,0] for i in range(m)]

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

for meal in choices:
   meals[meal-1][1] += 1

costIn = list(map(int, input().split()))
for i,cost in enumerate(costIn):
   meals[i][0] = cost

#Sortera i fallande prisordning
meals.sort()
meals.reverse()

#Vår stack där vi lägger rätter som inte parats ihop ä
   n
freeMeals = []

#Lista på alla "optimala" byten
swaps = []

for i in range(m):
   p = meals[i][1] #Antal personer som valt denna rätt
   c = meals[i][0] #Priset för denna rätt
   if p == 0: #Ifall ingen valt denna rätt, lägg till
      i stacken
      freeMeals.append(c)
   else:
      #Räkna med den här rätten som en av de k unika
      k -= 1

   #Så länge det finns saker i stacken och det finns
      personer,
   #para ihop de och lägg till bytet i swaps.
   while len(freeMeals)>0 and p>1:
      swaps.append(freeMeals.pop()-c)
      p -= 1

#Ifall det inte finns tillräckligt många byten att gö
```

```
    ra
if len(swaps) < k:
    print(-1)
else:
    #Välj ut de k billigaste bytena.
    swaps.sort()

    ans = 0;

    for i in range(k):
        ans+=swaps[i]
    print(ans)
```