杨博海：

第一题：本程序采用全面搜索的方法，找出最精确的三个根。其中 x[0]为函数自变量，y[0]为函数值，使 abs(y[0])成为极小值的自变量 x[0]即为方程的根。x[i] (i>0)用于存放根的值。lasty 是上一次算出的函数值，当 abs(y[0])<abs(lasty)时，随着 x[0]的增大，abs(y[0])越来越小，只要对 y[0]进行跟踪，算出使 abs(y[0])最小的 x[0]值，再存入 x[i]，然后再等待下一次的 abs(y[0])<abs(lasty)，并计算出所有的根。程序如下：

```
const
  nn=3;nm=2;     {nn 表示方程的最高次项的指数，nm 表示计算时精确到小数点后的位数}
var
  da,x,y:array[0..nn] of real;   {da[i]：x 的 i 次项系数}
  b,i,g:integer;
  j,long,lasty,m,fs,fe:real;
  a:longint;
  f:boolean;
begin
  fs:=-100;fe:=100;m:=1/exp(nm*ln(10));f:=false; {fs：根的下限，fe：根的上限}
  for a:=0 to nn do begin                    { m：1/10^nm ，f：表示 y[0]是否正在减小 }
    read(da[nn-a]);
  end;
  for a:=trunc(fs/m)-1 to trunc(fe/m) do begin
    x[0]:=a*m;y[0]:=0;j:=1;                  {按顺序产生 x[0]}
    for b:=0 to nn do begin
      y[0]:=y[0]+da[b]*j;                    {计算出相应的 y[0]}
      j:=j*x[0];
    end;
    if (abs(y[0])<abs(lasty)) and (not f) then begin
      i:=i+1;
        f:=true
    end;
    if (abs(y[0])>abs(lasty)) and f then begin
      x[i]:=x[0]-m;                          {将上次的 x[0]值存入 x[i]}
      f:=false;
      if g=nn then a:=trunc(fe/m);           {判断是否已经计算出所有的根}
    end;
        lasty:=y[0];
  end;
  for a:=1 to nn do
    writeln(x[a]:0:nm);                      {输出结果}
end.
```

总结：此程序结构简单，对根的位置没有要求，但由于采用了全面搜索的算法，效率较低，当精度要求很高时，其速度远不及二分法。

第二题：

递归算法：

本程序用递归的方法进行计算，首先建立递推关系式：将一整数 num 按题意分成 k 个整数之和的分法个数 M 满足如下关系：（简写为 M(num,k,large)，其中 large 为将 num 分成 k 分之后的最大整数，在这里 large=num-k+1）

1　当 large=1，num=k 时，M(num,k,large)=1。

2　当 large=1，num≠k 时，M(num,k,large)=0。

3　当 k=1，num<=large 时，M(num,k,large)=1。

4　当 k=1，num>k 时，M(num,k,large)=0。

5　当 k>1，large>1 时，M(num,k,large)=M(num-large,k-1,MAX(num-k-large+2,large))+M(num,k,large-1)。

优化：用递推关系式递归可以进行大幅度优化，而计数法递归却做不到这一点。在本题中当 k 值较大时(如 k>5)，有些 M 值被计算上千遍甚至上万遍，如果每个需要计算的 M 值只需算一遍，那么其速度会成倍提高。建立数组存储是最好的方法，但由于内存有限，本程序只建立了部分 M 值的数组。尽管如此，优化后程序的效率最大可为原先的 960 倍(num=200，k=6)。

程序如下：

```
const
  mn=200;mk=6;ml=110;sk=2;  { mn：数组中 num 的上限，ml：数组中 large 的上限}
  opendata=true;    {opendata：优化开关(true 为打开)，mk：数组中 k 的上限，sk：数组中 k 的下限}
type
  data=array[1..mk,1..ml] of longint;  {建立数组}
var
  n,m:integer;
  da:array[1..mn] of ^data;          {建立数组}
function work(num,k,large:integer):longint;              {递归函数}
var
  temp:longint;
  max:integer;
  f:boolean;
begin
  f:=(num<=mn) and (num>0) and (k<=mk) and (k>=sk-1) and (large<=ml) and (large>0) and opendata;
                    {判断 num，k，large 的值是否在存储范围之内和是否打开优化功能}
  if f and (da[num]^[k][large]>-1) then                          {判断该值是否已经计算过}
    temp:=da[num]^[k][large]
  else                                      {以下是算法的核心部分}
    if large<1 then temp:=0
    else
      if large=1 then
        if num=k then temp:=1 else temp:=0
      else
        if k=1 then
          if large>=num then temp:=1 else temp:=0
        else begin
          max:=num-k-large+2;
          if max>large then max:=large;
```

```
        temp:=work(num-large,k-1,max)+work(num,k,large-1);
      end;
  work:=temp;
  if f then da[num]^[k][large]:=temp;                    {将算好的值储存}
end;
begin                                          {主程序}
  if opendata then                                  {判断是否打开优化功能}
    for n:=1 to mn do begin
      new(da[n]);                                  {数组初始化}
      FillChar(da[n]^, SizeOf(da[n]^), 255);              {将每一个数初始化为-1}
    end;
  readln(n,m);
  writeln(work(n,m,n-m+1));                          {计算并打印}
end.
```

总结：该程序是单纯的递归+优化，结构简单。这种优化方法适用于每一个以递推关系式为基础递归，其效果十分明显，且容易编写，但是所需内存巨大，甚至超过了动态规划，并且在空间上无法优化。如果内存足够大，优化后的递归计算次数要小与动态规划，但由于递归本身的效率不高，如果本题中的 num，k 值不是非常大，动态规划的效率仍然大于优化后的递归。

动态规划算法：

设 N(num,k,large)为将一整数 num 按题意分成最大数为 large 的 k 个整数之和的分法个数，由刚才的递推关系式⑤可得 M(num,k,large)=N(num,k,large)+M(num,k,large-1)=∑N(num,k,i) (1≦i≦large)，N(num,k,large)=M(num-large,k-1,MAX(num-k-large+2,large))。

不难看出 M(num,k,large)=∑M(num-i,k-1,MAX(num-k-i+2,i)) (1≦i≦large)

所以由 M(1~num,k,1~large)即可算出 M(1~num,k+1,1~large)。

但是这种算法的效率并不是最高的，为了进一步减少运算量，可以把对最大数(large)的讨论改为对最小数(small)的讨论，因为把 num 分为 k 份的 k 个整数中最大数的范围是 num div k +1 ~ num-k+1，最小数的范围是 1 ~ num div k，当 num 不变时，随着 k 的增大，最小数的范围将远小于最大数。这样可以进一步减少运算量。

设 S(num,k,small)为将一整数 num 按题意分成最小数不小于 small 的 k 个整数之和的分法个数。

设 T(num,k,small)为将一整数 num 按题意分成最小数为 small 的 k 个整数之和的分法个数。

S(num,k,small)=T(num,k,small)+S(num,k,small+1)= ∑T(num,k,i) (small≦i≦num div k ，1≦small≦num div k)，T(num,k,small)=S(num-small,k-1,small)

S(num,k,small)= ∑S(num-i,k-1,i) (small≦i≦num div k)

将一整数 num 按题意分成 k 个整数之和的分法个数为 S(num,k,1)

程序如下：

```
const
  max=200;                        {max：num 的最大值}
type
  long200=array[1..max] of longint;        {建立数组}
var
  data:array[1..2,1..max] of ^long200       {建立数组 data[num,k]^[small]};
  a,b,c,d,e,f,g,h,i,j,k,n:integer;             {n：num  k：k}
```

```
begin
  readln(n,k);
if (n<=max) and (k<=n) then              {判断输入数据是否正确}
begin
  for a:=1 to 2 do
    for b:=1 to 200 do
      new(data[a][b]);                   {数组初始化}
  for a:=1 to 200 do
    for b:=1 to a do
      data[1][a]^[b]:=1;                 {S(a,1,b)=1 ,(b≦a)}
  i:=1;j:=2;        {以下是算法的核心部分，在计算时两个数组将交替使用，用 i，j 的交换来实现}
  for a:=2 to k do                   {计算 S(1~num,2,1~small) ~ S(1~num,k,1~small)}
  begin
      i:=(a mod 2)+1;                          {交换 i，j }
    j:=((a+1) mod 2)+1;
    for b:=a to n do
    begin
      h:=b div a;                        {h：把 b 分成 a 份后的最小数的最大值}
      for c:=1 to h do
        data[j][b]^[c]:=data[i][b-c]^[c];                {计算 T(b,a,c)}
      for c:=h-1 downto 1 do
        data[j][b]^[c]:=data[j][b]^[c]+data[j][b]^[c+1];    {计算 S(b,a,c)}
    end;
  end;
  writeln(data[j][n]^[1]);     {输出结果}
end;
  writeln;
end.
```

第三题：

本程序采用动态规划的方法计算

设 ST(start,end)为题目所给的字符串中第 start 个字符到第 end 个字符所组成的新字符串。

D(ST(start,end),k)为将字符串 ST(start,end)分成 k 份后所含的最大单词数。（简写为 D(start,end,k)）

则 D(start,end,k+1)=MAX(D(start,i,k)+D(i+1,end,k)) (start≦i≦end-1)

所以用 D(1~start,start~end,1)即可求得 D(1,end,k)

本程序中求 D(1~start,start~end,1)的部分较复杂详细求法见注解。

程序如下：

```
type
  by200=record                       {建立数组 D}
    da:array[1..200,1..200] of byte;
  end;
  ds1=record                        {ds1：用于存放任务数据(单个任务)的数据类型}
       st:string;                    {字符串变}
    word:array[1..20] of string;          {单词数组}
```

```
    m:integer;                    {单词个数}
    k:integer;                    {分为 k 份}
  end;
  ds=record                     {ds：用于存放所有任务数据的数据类型}
    n:integer;                  {任务个数}
    da:array[1..6] of ds1;          {任务数据}
  end;
var
  i,j,max,a,a1,b,c,d,e,f:integer;
  dt,p:^by200;                  {建立数组 D}
  dat:array[1..3] of ^by200;          {建立数组 D}
  lo:ds;                        {lo: 用于存放任务数据}
  t:boolean;
  st:string;
  labels:array[1..200] of byte;         {labels：用于存放查找到的单词长度}
procedure loadfile(sn:string);          {读文件的过程}
var
  fi:text;
  t1,t2,t3,t4,t5,t6:integer;
  st1:string;
begin
  assign(fi,sn);
  reset(fi);
  readln(fi,lo.n);
  for t1:=1 to lo.n do
  begin
    readln(fi,t2,lo.da[t1].k);
    for t3:=1 to t2 do
    begin
      readln(fi,st1);
      lo.da[t1].st:=lo.da[t1].st+st1;
    end;
    readln(fi,t2);
    lo.da[t1].m:=t2;
    for t3:=1 to t2 do
       readln(fi,lo.da[t1].word[t3]);
  end;
  close(fi);
end;

begin
  writeln;
  readln(st);
  writeln('Open File : ',st);
```

```
 loadfile(st);
 for a:=1 to 3 do
  new(dat[a]);                          {建立数组 D}
 for a:=1 to lo.n do
 begin
  for i:=1 to lo.da[a].m-1 do
   for j:=i+1 to lo.da[a].m do
    if lo.da[a].word[i][0]>lo.da[a].word[j][0] then {首先对单词进行排序，使单词的长度由小到大排列。
为了保证程序算出的是最优值，长度较小的单词将优先处理}
      begin
       st:=lo.da[a].word[i];
       lo.da[a].word[i]:=lo.da[a].word[j];
       lo.da[a].word[j]:=st;
      end;
  for i:=1 to 200 do
   labels[i]:=255;
  f:=length(lo.da[a].st);               {f 是字符串的长度}
  for i:=1 to lo.da[a].m do
  begin
   c:=length(lo.da[a].word[i]);
   for j:=1 to f-c+1 do
    if (labels[j]=255) and (copy(lo.da[a].st,j,c)=lo.da[a].word[i]) then
     labels[j]:=c;   {labels[j]表示首字母占用字符串中第 j 个位置的单词的长度}
  end;
  for j:=1 to f do
  begin
   d:=0;                        {d 用于计算 D(i,j,1)}
   for i:=j downto 1 do
   begin
    if labels[i]<=j-i+1 then d:=d+1;        {如果单词的长度小于 i 到 j 的距离，
那么这个单词就包含在 ST(i,j)内，如果单词含在 ST(i,j)内，那么这个单词一定包含在 ST(1 ~ i-1,j)内}
     dat[1]^.da[i][j]:=d;
   end;
  end;                              {D(1~i,i+1~end,1) 计算完毕(1≤i≤end-1)}
  for a1:=1 to lo.da[a].k-1 do                  {计算 D(1,end,k)}
  begin
   if a1=1 then
   begin
    i:=1;
    j:=2;
   end
   else
   begin
    i:=2+(a1 mod 2);
```

```
      j:=2+((a1+1) mod 2);                    {通过 i,j 的交换来实现 D(1,end,2~k)的计算}
    end;
    for b:=1 to f-a1 do
     for c:=b+a1 to f do
     begin
       max:=0;
       for d:=b+a1 to c do
       begin
         e:=dat[i]^.da[b][d-1]+dat[1]^.da[d][c];
         if e>max then max:=e;                {计算 MAX(D(b,d,a)+D(d+1,c,a))}
       end;
       dat[j]^.da[b][c]:=max;
     end;
   end;
   if lo.da[a].k=1 then j:=1;
   writeln(dat[j]^.da[1][f]);
 end;
 writeln;
end.
```

第四题：

本程序采用动态规划计算，由于题目所给的数据结构复杂，为了便于处理数据和进行计算，程序中使用了大量的指针参与存储。

pci 为机场的数据类型，pcit 为城市。cpr(pci1,pci2)为从机场 pci1 直接到达机场 pci2 所需的最小费用（在程序中用函数 cpr 计算）。程序将出发城市和目的城市的铁路价格设为 0，这样从出发城市的任意一机场到目的城市的任意一机场的最少费用即为旅程所需费用。

动态规划思想：假设一机场 a、一机场集合 P，a 不属于 P，p_n 属于 P，cpr(a, p_{k1})最少。则在所有从 a 到 p_{k1} 的途径中直接到达的花费最少。若 MIN(cpr(a, p_{k2}), cpr(p_{k1}, p_{k2}))最小(k1≠k2)，则从 a 到 p_{k2} 的最小花费为 MIN(cpr(a, p_{k2}), cpr(p_{k1}, p_{k2}))。若 MIN(cpr(a, p_{k3}), cpr(p_{k2}, p_{k1}))最小(k2≠k3)，则从 a 到 p_{k3} 的最小花费为 MIN(cpr(a, p_{k3}), cpr(p_{k2}, p_{k3}))……只要这样计算下去，即可求得从 a 到 p_n 的最小花费。

将机场数据制成环状链表

程序如下：

```
type
  pci=^ci;                          {机场的数据类型}
  pcit=^cit;                        {机场的数据城市}
  ci=record                         {机场的数据类型}
    base:pcit;                      {指向机场所在的城市}
    x,y,minpr:real;                   {x,y：机场坐标，minpr 到达该机场所需最低费用}
    last,next:pci;                  {用于连成双向链表的指针}
  end;
  cit=record                        {机场的数据城市}
    num:integer;
    coli:real;                      {该城市中的铁路费用}
    pt:array[1..4] of pci;              {指向该城市中的的四个机场}
```

```
  end;
var
  ct:array[1..4,1..2] of real;
  city:array[1..100] of cit;                    {储存城市数据的数组}
  back,temp,point,start,endd:pci;
  min,copl,max,tel:real;
  fl:text;
  a,b,c,d,e,f,i,j,n,m,s:integer;
  sn:string;
function cpr(n1,n2:pci):real;          {计算从机场 pci1 直接到达机场 pci2 所需的最小费用的函数}
begin
  if n1^.base=n2^.base then
    if n1^.base^.coli<copl then cpr:= sqrt(sqr(n1^.x-n2^.x)+sqr(n1^.y-n2^.y))*n1^.base^.coli
    else cpr:= sqrt(sqr(n1^.x-n2^.x)+sqr(n1^.y-n2^.y))*copl
  else
  begin
    cpr:=sqrt(sqr(n1^.x-n2^.x)+sqr(n1^.y-n2^.y))*copl;
  end;
end;
begin
  readln(sn);
  assign(fl,sn);
  reset(fl);
  readln(fl,n);
  for a:=1 to n do
  begin
    read(fl,m,copl,s,e);  {m：城市个数，copl：飞机费用 s：起始城市编号，e：目的城市编号}
    for b:=1 to m do
    begin
      for c:=1 to 3 do
        read(fl,ct[c][1],ct[c][2]);                    {读取城市坐标}
      max:=0;
      for c:=1 to 3 do
      begin
        i:=1;j:=3;              {若 c=2，则 i=1，j=3}
        if c=1 then i:=2;           {若 c=1，则 i=2，j=3}
        if c=3 then j:=2;           {若 c=3，则 i=1，j=2}
        tel:=sqrt(sqr(ct[i][1]-ct[j][1])+sqr(ct[i][2]-ct[j][2]));    {城市中机场 i 与 j 之间的距离}
        if tel>max then
        begin
          max:=tel;          {求 tel 的最大值，因为机场 i，j，c 的位置可组成一个直角三角形，
使 tel 最大的机场 c 与机场 i 和 j 的夹角是直角，那么第 4 个机场就在机场 c 的对面}
          ct[4][1]:=ct[j][1]+ct[i][1]-ct[c][1];
          ct[4][2]:=ct[j][2]+ct[i][2]-ct[c][2];        {利用平移的方法求出第 4 个机场的坐标}
```

```
    end;
  end;
  read(fl,tel);                          {读取铁路价格}
  for c:=1 to 4 do
  begin
    new(temp);                           {初始化机场数据}
    temp^.x:=ct[c][1];
    temp^.y:=ct[c][2];
    temp^.base:=@city[b];
    city[b].pt[c]:=temp;
  end;
  city[b].num:=b;
  city[b].coli:=tel;
end;
city[s].coli:=0;
city[e].coli:=0;                    {将出发城市和目的城市的铁路价格设为 0}
start:=city[s].pt[1];
endd:=city[e].pt[1];                  {设定起始机场 start 和一个目的机场 endd}
start^.minpr:=0;                    {到达 start 的花费为 0}
temp:=city[m].pt[4];     {使制作完的链表首尾相接，成为环状链表，以便进行下一步的计算}
for b:=1 to m do         {制作机场数据链表}
  for c:=1 to 4 do
  begin
    temp^.next:=city[b].pt[c];
    city[b].pt[c]^.last:=temp;
    temp:=city[b].pt[c];
  end;
if start^.base=endd^.base then writeln(0)       {如果出发城市与目的城市相同，那么费用为 0}
else
begin
  point:=start;
  temp:=point^.next;
  while(temp<>point) do            {将除起始机场外的所有机场的最小费用设为无穷大}
  begin
    temp^.minpr:=1E38;
    temp:=temp^.next;
  end;
  while(point<>endd) do  {point 指向链表中已知最小费用的机场，若 point 指向目的机场则结束循环}
  begin
    temp:=point^.next;
    while(temp<>point) do          {更新各机场的最小费用}
    begin
      tel:=cpr(temp,point)+point^.minpr;
      if tel<temp^.minpr then temp^.minpr:=tel;
```

```
      temp:=temp^.next;
    end;
    temp:=point^.next;
    back:=point;
    min:=temp^.minpr;
    point:=temp;
    while(temp<>back) do          {寻找并用 point 指向下一个已知最小费用的机场}
    begin
      if temp^.minpr<min then
      begin
        min:=temp^.minpr;
        point:=temp;
      end;
      temp:=temp^.next;
    end;
    back^.next^.last:=back^.last;      {将原 point 指向的机场数据从链表中删除}
    back^.last^.next:=back^.next;
  end;
  writeln(endd^.minpr:0:2);          {输出结果}
 end;
end;
writeln;
readln;
end.
```