Chapter 5

Practices for Greedy Algorithms

Greedy algorithms are used to solve optimization problems through a sequence of stages. At each stage greedy algorithms make the locally optimal choice in order to find a globally optimal solution. For some problems, greedy algorithms can yield a globally optimal solution; but for some problems, such as the traveling salesman problem (TSP), they can't.

This chapter organizes practices for greedy algorithms as follows.

1. Practices for Greedy Algorithms
2. Greedy-Choices based on Sorted Data
3. Greedy Algorithms used with other methods to solve P-Problems

5.1 Practices for Greedy Algorithms

Greedy algorithms are used to solve optimization problems through a sequence of steps, and make the choice that looks best at each step. There are some famous greedy algorithms, such as Prim's algorithm and Kruskal's algorithm used to find a minimum spanning tree for a weighted undirected graph; Dijkstra's algorithm used to get single-source shortest paths between nodes in a graph; and Huffman coding.

There are two properties for optimization problems that can be solved by greedy algorithms.

1. Optimal substructures: Optimal solutions to problems consist of a sequence of their optimal solutions to subproblems (Necessity).

2. The property for greedy-choices: Global optimal solutions to problems can be gotten by making a sequence of local optimal (greedy) choices (Feasibility).

The following two experiments are practices for greedy algorithms.

【5.1.1 Pass-Muraille】

In modern day magic shows, passing through walls is very popular in which a magician performer passes through several walls in a predesigned stage show. The wall-passer (Pass-Muraille) has a limited wall-passing energy to pass through at most k walls in each wall-passing show. The walls are placed on a grid-like area. An example is shown in Figure 1, where the land is viewed from above. All the walls have unit widths, but different lengths. You may assume

that no grid cell belongs to two or more walls. A spectator chooses a column of the grid. Our wall-passer starts from the upper side of the grid and walks along the entire column, passing through every wall in his way to get to the lower side of the grid. If he faces more than *k* walls when he tries to walk along a column, he would fail presenting a good show. For example, in the wall configuration shown in Figure 1, a wall-passer with *k* = 3 can pass from the upper side to the lower side choosing any column except column 6.

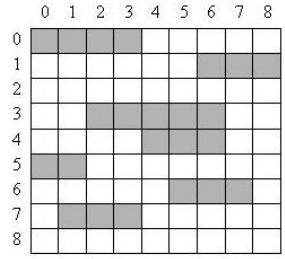

Figure 1. Shaded cells represent the walls.

Figure5.1-1

Given a wall-passer with a given energy and a show stage, we want to remove the minimum number of walls from the stage so that our performer can pass through all the walls at any column chosen by spectators.

Input

The first line of the input file contains a single integer *t* (1≤*t*≤10), the number of test cases, followed by the input data for each test case. The first line of each test case contains two integers *n* (1≤*n*≤100), the number of walls, and *k* (0≤*k*≤100), the maximum number of walls that the wall-passer can pass through, respectively. After the first line, there are *n* lines each containing two (*x*, *y*) pairs representing coordinates of the two endpoints of a wall. Coordinates are non-negative integers less than or equal to 100. The upper-left of the grid is assumed to have coordinates (0, 0). The second sample test case below corresponds to the land given in Figure 1.

Output

There should be one line per test case containing an integer number which is the minimum number of walls to be removed such that the wall-passer can pass through walls starting from any column on the upper side.

Sample Input	Sample Output
2 3 1 2 0 4 0 0 1 1 1 1 2 2 2 7 3 0 0 3 0 6 1 8 1 2 3 6 3	1 1

4 4 6 4 0 5 1 5 5 6 7 6 1 7 3 7	

Hint

Walls are parallel to X.

Source: ACM Tehran 2002 Preliminary

IDs for Online Judges: POJ 1230 , ZOJ 1375

Analysis

All columns are scanned from left to right. Removing the minimum number of walls from the stage must guarantee removing the minimum number of walls in scanned columns. Therefore, the optimal solution to the problem consists of its optimal solutions to subproblems. The key to the problem is its greedy-choice.

Suppose there are *D* walls in the current column. If *D*≤*K*, we needn't remove any wall; and if *D*>*K*, *D*-*K* walls must be removed. The greedy-choice is as follows. For walls in the current column, the longest *D*-*K* walls in unscanned columns are removed. Obviously, the greedy-choice removes minimum number of walls.

Program

```
#include<iostream>
using namespace std;
int t,n,k,x,y,x1,y2,max_x,max_y,sum_s=0;        //t: number of test cases; n: number of walls; k: at most k walls can be passed through; x,y,x1,y2: Coordinate; max_x,max_y: maximal row and column Coordinate; sum_s: the minimum number of removed walls
int map[105][105];
int main()
{
   scanf("%d",&t);                          // number of test cases
   while(t--)                               // all test cases are processed
   {
      memset(map,0,sizeof(map));
      max_x=0;                             //Initialization
      max_y=0;
      sum_s=0;
      scanf("%d %d",&n,&k);
      for (int i=1;i<=n;i++)
      {
         scanf("%d %d %d %d",&x,&y,&x1,&y2);
         if (x>max_x)max_x=x;
         if (x1>max_x)max_x=x1;
         if(y>max_y)max_y=y;
         if (x<x1)
```

```
          {
             for (int j=x;j<=x1;j++) map[j][y]=i;
          }
         else
         {
             for (int j=x1;j<=x;j++) map[j][y]=i;
         }
       }
      for (int i=0;i<=max_x;i++)             //scan from left to right
      {
               int tem=0;           //calculate the number of walls in the i-th
column
               for (int j=0;j<=max_y;j++)  if (map[i][j]>0) tem++;
               int offset=tem-k;
               if (offset>0)             // some walls are romoved
               {
                   sum_s+=offset;
                   while(offset--)
                   {
                         int max_s=0,max_bh;
                         for (int k=0;k<=max_y;k++)  //search
                         {
                              if (map[i][k]>0)
                  //calculate length of wall in unscanned columns
                              {
                                   int tem_s=0;
                                   for (int z=i+1;z<=max_x;z++)
                                     if (map[z][k]==map[i][k]) tem_s++;
                                         else  break;
                                   if (max_s<tem_s) //record
                                   {
                                      max_s=tem_s; max_bh=k;
                                   }
                              }
                         }
                           for (int a=i;a<=i+max_s;a++) map[a][max_bh]=0;    //
  some walls are removed
                   }
               }
      }
      printf("%d\n",sum_s);   //output the result
   }
   return 0;
}
```

【5.1.2 Tian Ji -- The Horse Racing】

Here is a famous story in Chinese history.

That was about 2300 years ago. General Tian Ji was a high official in the country Qi. He likes to play horse racing with the king and others.

Both of Tian and the king have three horses in different classes, namely, regular, plus, and super. The rule is to have three rounds in a match; each of the horses must be used in one round. The winner of a single round takes two hundred silver dollars from the loser.

Being the most powerful man in the country, the king has so nice horses that in each class his horse is better than Tian's. As a result, each time the king takes six hundred silver dollars from Tian.

Tian Ji was not happy about that, until he met Sun Bin, one of the most famous generals in Chinese history. Using a little trick due to Sun, Tian Ji brought home two hundred silver dollars and such a grace in the next match.

It was a rather simple trick. Using his regular class horse race against the super class from the king, they will certainly lose that round. But then his plus beat the king's regular, and his super beat the king's plus. What a simple trick. And how do you think of Tian Ji, the high ranked official in China?

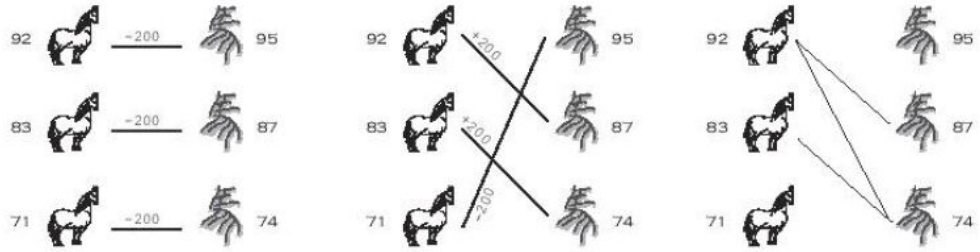

Figure5.1-2

Were Tian Ji lives in nowadays, he will certainly laugh at himself. Even more, were he sitting in the ACM contest right now, he may discover that the horse racing problem can be simply viewed as finding the maximum matching in a bipartite graph. Draw Tian's horses on one side, and the king's horses on the other. Whenever one of Tian's horses can beat one from the king, we draw an edge between them, meaning we wish to establish this pair. Then, the problem of winning as many rounds as possible is just to find the maximum matching in this graph. If there are ties, the problem becomes more complicated, he needs to assign weights 0, 1, or -1 to all the possible edges, and find a maximum weighted perfect matching...

However, the horse racing problem is a very special case of bipartite matching. The graph is decided by the speed of the horses -- a vertex of higher speed always beat a vertex of lower speed. In this case, the weighted bipartite matching algorithm is a too advanced tool to deal with the problem.

In this problem, you are asked to write a program to solve this special case of matching problem.

Input

The input consists of up to 50 test cases. Each case starts with a positive integer n ($n \leq 1000$) on the first line, which is the number of horses on each side. The next n integers on the second line are the speeds of Tian's horses. Then the next n integers on the third line are the speeds of the king's horses. The input ends with a line that has a single `0' after the last test case.

Output

For each input case, output a line containing a single number, which is the maximum money Tian Ji will get, in silver dollars.

Sample Input	Sample Output
3 92 83 71 95 87 74 2 20 20 20 20 2 20 19 22 18 0	200 0 0

Source: ACM Shanghai 2004

IDs for Online Judges: POJ 2287 , ZOJ 2397 , UVA 3266

Analysis

The problem can be solved by several different methods. Maximum matching in a bipartite graph or dynamic programming can be used to solve the problem, but using greedy algorithm to solve the problem is simple and efficient. The greedy algorithm is as follow.

First, the speeds of Tian's horses and the speeds of the king's horses are sorted in ascending order respectively. Suppose the sequence for speeds of Tian's horses in ascending order is $A=a_1 \ldots a_n$; and the sequence for the speeds of the king's horses are sorted in ascending order is $B=b_1 \ldots b_n$.

Second, greedy-choices are as follow.

1. If Tian's slowest horse is faster than the king's slowest horse, that is, $a_1 > b_1$; then Tian's slowest horse races against the king's slowest horse, that is, a_1 is compared with b_1. Because b_1 is less than any elements in A and the king's slowest horse can be defeated by any Tian’s horse, it is suitable that the king's slowest horse is defeated by Tian's slowest horse.

2. If Tian's slowest horse is slower than the king's slowest horse, that is, $a_1 < b_1$; then Tian's slowest horse races against the king's fastest horse, that is, a_1 is compared with b_n. Because a_1 is less than any elements in B and Tian's slowest horse can be defeated by any king's horse, it is suitable that Tian's slowest horse is defeated by the king's fastest horse.

3. If Tian's fastest horse is faster than the king's fastest horse, that

is, $a_n>b_n$; then Tian's fastest horse races against the king's fastest horse, that is, a_n is compared with b_n. Because a_n is larger than any elements in B and Tian's fastest horse can defeat any king's horse, it is suitable that Tian's fastest horse defeats the king's fastest horse.

4. If Tian's fastest horse is slower than the king's fastest horse, that is, $a_n<b_n$; then Tian's slowest horse races against the king's fastest horse, that is, a_1 is compared with b_n. Because b_n is larger than any elements in A and the king's fastest horse can defeat any Tian's horse, it is suitable that the king's fastest horse defeats Tian's slowest horse.

5. If ($a_1==b_1$) and ($a_n>b_n$), then it is suitable that Tian's fastest horse races against the king's fastest horse, that is, a_n is compared with b_n.

6. If ($a_n==b_n$), then there exist an optimal solution that a_1 is compared with b_n.

The above process repeats until the horse racing ends. Tian's fastest or slowest horse races against the king's fastest or slowest horse each time based on above greedy-choices. Optimal solutions to subproblems constitute the global optimal solution to the problem.

Program

```
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
int a[1010],b[1010];                    //Speeds of Tian's horses and the king's horses
int main()
{
    int n;
    while(scanf("%d",&n),n)     //number of Tian's horses(the king's horses)
    {
        for(int i=1; i<=n; i++)  scanf("%d",&a[i]);   // Input speeds of Tian's horses
         for(int i=1; i<=n; i++)  scanf("%d",&b[i]);   // Input speeds of the king's horses
        sort(a+1,a+1+n);                          //Sorting speeds in ascending order
        sort(b+1,b+1+n);
        int tl=1,tr=n,ql=1,qr=n;           //Initialization
        int sum=0;
        while(tl<=tr)                    // the horse racing doesn't end
        {
              if(a[tl]<b[ql])       // Tian's slowest horse is slower than the king's slowest horse
            {
                qr--;tl++;sum=sum-200;
```

```
            }
            else if(a[tl]==b[ql])          // Speeds of two slowest horses are
same
            {
                while(tl<=tr&&ql<=qr)
                {
                     if(a[tr]>b[qr]) //Tian's fastest horse is faster than the
king's fastest horse
                    {
                        sum+=200;tr--;qr--;
                    }
                     else      // Tian's slowest horse races against the king's
fastest horse
                    {
                        if(a[tl]<b[qr])  sum-=200;
                        tl++;qr--; break;
                    }
                }
            }
             else   // Tian's slowest horse is faster than the king's slowest
horse
            {
                tl++;ql++;sum=sum+200;
            }
        }
        printf("%d\n",sum);             //Output the result
    }
    return 0;
}
```

5.2 Greedy-Choices based on Sorted Data

The key to a greedy algorithm is its greedy-choices. Sometimes the greedy-choices must be based on sorted data. First, data are sorted. Then greedy-choices are made based on sorted data.

【5.2.1 Shoemaker's Problem】

Shoemaker has N jobs (orders from customers) which he must make. Shoemaker can work on only one job in each day. For each i_{th} job, it is known the integer T_i ($1 \le T_i \le 1000$), the time in days it takes the shoemaker to finish the job. For each day of delay before starting to work for the i_{th} job, shoemaker must pay a fine of S_i ($1 \le S_i \le 10000$) cents. Your task is to help the shoemaker, writing a programm to find the sequence of jobs with minimal total fine.

Input

The input begins with a single positive integer on a line by itself indicating the number of the cases following, each of them as described below. This line is followed by a blank line, and there is also a blank line between two consecutive inputs.

First line of input contains an integer N ($1 \le N \le 1000$). The next N lines each contain two numbers: the time and fine of each task in order.

Output

For each test case, the output must follow the description below. The outputs of two consecutive cases will be separated by a blank line.

You programm should print the sequence of jobs with minimal fine. Each job should be represented by its number in input. All integers should be placed on only one output line and separated by one space. If multiple solutions are possible, print the first lexicographically.

Sample Input	Sample Output
1 4 3 4 1 1000 2 2 5 5	2 1 3 4

Source: Second Programming Contest of Alex Gevak , 2000

ID for Online Judge: UVA 10026

Analysis

"For each day of delay before starting to work for the i_{th} job, shoemaker must pay a fine of S_i cents" means "For each day of delay after starting to work for the i_{th} job, shoemaker must pay a fine of S_i/T_i cents". S_i/T_i is the measurement of influence for the i_{th} job, $1 \le i \le n$. Therefore, in order to pay minimal fine, the job whose measurement of influence is higher must be finished earlier. The greedy algorithm is as follow.

The metric for jobs is their measurement of influence. The n jobs are sorted using their measurement of influence as the first key (in ascending order), and numbers of jobs as the second key (in descending order). The sorted sequence is the sequence of jobs with minimal fine.

Program

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

```
const int maxN=1010;            // the upper limit of the number of jobs
struct job
{
    double a;                   // measurement of influence for the job
    int num;                    // the number of a job
} p[maxN];                     // the sequence of jobs with minimal fine
int n;
void init()
{
    double a1,a2;
    scanf("%d",&n);             // n jobs
    for (int i=1;i<=n;i++)        // the time and fine of each task
    {
        scanf("%lf%lf",&a1,&a2);
        p[i].a=a2/a1;p[i].num=i;     // Calculate Si/Ti, and record the number
    }
}
bool cmp(job x,job y)            // sort two jobs using their measurement of
influence as the first key (in ascending order), and numbers of jobs as the
second key (in descending order)
{
    if ((x.a>y.a)||((x.a==y.a)&&(x.num<y.num))) return true;
    return false;
}
void work()
{
    sort(p+1,p+n+1,cmp);          // sort n jobs using their measurement of
influence as the first key (in ascending order), and numbers of jobs as the
second key (in descending order)
    for (int i=1;i<n;i++) printf("%d ",p[i].num);        // Output the result
    printf("%d\n",p[n].num);
}
int main()
{
    int t;
    scanf("%d",&t);                              //the number of test case
    for (int i=1;i<=t;i++)                         // deal with each test case
    {
        if (i>1) printf("\n");
        init();
        work();
    }
    return 0;
}
```

【5.2.2 Add All】

Yup!! The problem name reflects your task; just add a set of numbers. But you may feel yourselves condescended, to write a C/C++ program just to add a set of numbers. Such a problem will simply question your erudition. So, let's add some flavor of ingenuity to it.

Addition operation requires cost now, and the cost is the summation of those two to be added. So, to add 1 and 10, you need a cost of 11. If you want to add 1, 2 and 3. There are several ways –

1 + 2 = 3, cost = 3 3 + 3 = 6, cost = 6 Total = 9	1 + 3 = 4, cost = 4 2 + 4 = 6, cost = 6 Total = 10	2 + 3 = 5, cost = 5 1 + 5 = 6, cost = 6 Total = 11

I hope you have understood already your mission, to add a set of integers so that the cost is minimal.

Input

Each test case will start with a positive number, N ($2 \le N \le 5000$) followed by N positive integers (all are less than 100000). Input is terminated by a case where the value of N is zero. This case should not be processed.

Output

For each case print the minimum total cost of addition in a single line.

Sample Input	Sample Output
3 1 2 3 4 1 2 3 4 0	9 19

Source: UVa Regional Warmup Contest 2005

ID for Online Judge: UVA 10954

Analysis

Initially there is a set of n positive numbers. Each time two positive numbers are deleted from the set, and the sum of the two numbers is added into the set. The process repeats n-1 times. The final sum is the total cost of addition. The problem requires to calculate the minimum total cost of addition.

Obviously, in order to get the minimum total cost of addition, the greedy-choice is to select two minimal positive numbers each time. Therefore a min heap is suitable to represent the set.

Program

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
```

```
#include<cmath>
#include<cstring>
#include<algorithm>
using namespace std;
const int maxN=5010;                //the upper limit of the size of the set
int n,a[maxN];                      // n: the size of the heap, a[]: min heap
void sift(int i)                      // the subtree with root i is adjusted as
a min heap
{
    a[0]=a[i];
    int k=i<<1;
    while (k<=n)
    {
        if ((k<n)&&(a[k]>a[k+1])) k++;
        if (a[0]>a[k]) { a[i]=a[k];i=k;k=i<<1;} else k=n+1;
    }
    a[i]=a[0];
}
void work()     //Calculate and output the result
{
    for (int i=n >> 1;i;i--) sift(i);                  // set up a min heap
    long long ans=0;
    while (n!=1)
    {
        swap(a[1],a[n--]);
        sift(1);                               // adjust the heap
        a[1]+=a[n+1];
        ans+=a[1];
        sift(1);                              // adjust the heap
    }
    cout << ans << endl;                       //Output the result
}
int main()
{
    while (scanf("%d",&n),n)
    {
        for (int i=1;i<=n;i++) scanf("%d",&a[i]);   // Input n positive numbers
        work();                     // calculate and the minimum total cost of
addition
    }
    return 0;
}
```

【5.2.3 Wooden Sticks】

There is a pile of n wooden sticks. The length and weight of each stick are known in advance. The sticks are to be processed by a woodworking machine in one by one fashion. It needs some time, called setup time, for the machine to prepare processing a stick. The setup times are associated with cleaning operations and changing tools and shapes in the machine. The setup times of the woodworking machine are given as follows:

(a) The setup time for the first wooden stick is 1 minute.

(b) Right after processing a stick of length l and weight w , the machine will need no setup time for a stick of length l' and weight w' if $l \le l'$ and $w \le w'$. Otherwise, it will need 1 minute for setup.

You are to find the minimum setup time to process a given pile of n wooden sticks. For example, if you have five sticks whose pairs of length and weight are (9 , 4) , (2 , 5) , (1 , 2) , (5 , 3) , and (4 , 1) , then the minimum setup time should be 2 minutes since there is a sequence of pairs (4 , 1) , (5 , 3) , (9 , 4) , (1 , 2) , (2 , 5) .

Input

The input consists of T test cases. The number of test cases (T) is given in the first line of the input file. Each test case consists of two lines: The first line has an integer n , 1□□n □□5000, that represents the number of wooden sticks in the test case, and the second line contains $2n$ positive integers l_1, w_1, l_2, w_2,……, l_n, w_n, each of magnitude at most 10000 , where l_i and w_i are the length and weight of the i th wooden stick, respectively. The $2n$ integers are delimited by one or more spaces.

Output

The output should contain the minimum setup time in minutes, one per line.

Sample Input	Sample Output
3 5 4 9 5 2 2 1 3 5 1 4 3 2 2 1 1 2 2 3 1 3 2 2 3 1	2 1 3

Source: ACM Taejon 2001

IDs for Online Judges: POJ 1065 , ZOJ 1025 , UVA 2322

Analysis

Right after processing a stick of length l and weight w , the machine will need no setup time for a stick of length l' and weight w' if $l \le l'$ and $w \le w'$. Otherwise, it will need 1 minute for setup. In order to reduce setup time, the greedy-choice is as follows.

For unprocessed sticks, the stick with minimal length is selected first. If there are more than one stick with minimal length, the stick with minimal

weight is selected.

First all sticks are sorted. A stick is represented as (l, w), where l is its length, and w is its weight. Sticks are sorted using l as the first key and w as the second key. That is, ($l1$,$w1$)>($l2$,$w2$), if $l1$>$l2$ || ($l1$==$l2$ && $w1$>$w2$).

After sorting sticks, the greedy-choice is processed as follows.

Initially, setup time c=0, stick 0 is as the last processed stick cur. Then following steps repeat.

Step 1: All sticks in the sequence after stick cur are set as processed. And the last stick is as stick cur. The machine will need no setup time for these sticks;

Step 2: setup time c++;

Step 3: Search the first unprocessed stick in the sequence. If there is no unprocessed stick, the output the minimum setup time; else set the unprocessed stick as processed and stick cur, return to step 1.

Program

```
#include <iostream>
using namespace std;
const int N = 5000;
struct node{                                  // Struct of stick
      node& operator=(node &n){
              l=n.l, w=n.w, isUsed=n.isUsed;    //the length, weight, flag that is processed or not for stick n
            return *this;
    }
    bool operator>(node &n){                  //compare sticks
        return l>n.l || (l==n.l && w>n.w);
    }
    void swap(node &n){                      //exchange sticks
        node tmp=*this;
        *this=n;
        n=tmp;
    }
    int l, w;
    bool isUsed;
}A[N];                                    //sequence of sticks A[ ]
int main()
{
    int t, n, i, j, k;
    cin >> t;                             //number of test cases
    for(i=0;i<t;i++){                      // test cases are processed one by one
```

```
        cin >> n;
        for(j=0;j<n;j++){             //Input length, weight foe all sticks
            cin >> A[j].l >> A[j].w;
            A[j].isUsed=false;
        }
        for(j=1;j<n;j++)             //Sorting A
            for(k=1;k<=n-j;k++)
                if(A[k-1] > A[k])
                    A[k-1].swap(A[k]);
         node cur = A[0];            // stick 0 is as the last processed stick
cur
        A[0].isUsed=true;
        int c=0;                   //Initialize setup time
        while(true){
             for(j=1;j<n;j++)       //set sticks whose lengths and weights are
  larger than the current stick as processed
              if(A[j].isUsed==false)
                if(A[j].l >= cur.l && A[j].w >= cur.w){
                    A[j].isUsed=true;
                    cur = A[j];
                }
            c++;                   //setup time+1
               for(j=1;j<n;j++) if(A[j].isUsed==false){     //Search the first
unprocessed stick
                cur = A[j];
                A[j].isUsed=true;
                break;
            }
            if(j==n) break;           //all sticks are processed
        }
        cout << c << endl;            // output the minimum setup time
    }
    return 0;
}
```

【5.2.4 Radar Installation】

Assume the coasting is an infinite straight line. Land is in one side of coasting, sea in the other. Each small island is a point locating in the sea side. And any radar installation, locating on the coasting, can only cover *d* distance, so an island in the sea can be covered by a radius installation, if the distance between them is at most *d*.

We use Cartesian coordinate system, defining the coasting is the x-axis. The sea side is above x-axis, and the land side below. Given the position of

each island in the sea, and given the distance of the coverage of the radar installation, your task is to write a program to find the minimal number of radar installations to cover all the islands. Note that the position of an island is represented by its x-y coordinates.

Input

The input consists of several test cases. The first line of each case contains two integers *n* (1≤*n*≤1000) and *d*, where *n* is the number of islands in the sea and *d* is the distance of coverage of the radar installation. This is followed by *n* lines each containing two integers representing the coordinate of the position of each island. Then a blank line follows to separate the cases. The input is terminated by a line containing pair of zeros.

Output

For each test case output one line consisting of the test case number followed by the minimal number of radar installations needed. "-1" installation means no solution for that case.

Sample Input	Sample Output
3 2 1 2 -3 1 2 1 1 2 0 2 0 0	Case 1: 2 Case 2: 1

Source: ACM Beijing 2002

IDs for Online Judge: POJ 1328 , ZOJ 1360 , UVA 2519

Analysis

Each small island is represented as a segment on the coasting. If a rader locates on the segment, the island can be covered by the radar. Suppose the Cartesian coordinate for the island is (*x*, *y*). If a rader locates on the coasting from (*x*-*h*, 0) to (*x*+*h*, 0), where $h=\sqrt{d^2-y^2}$, the island can be covered. Therefor the island is represented as a segment from (*x*-*h*, 0) to (*x*+*h*, 0). It can be showed as Figure 5.2-2.

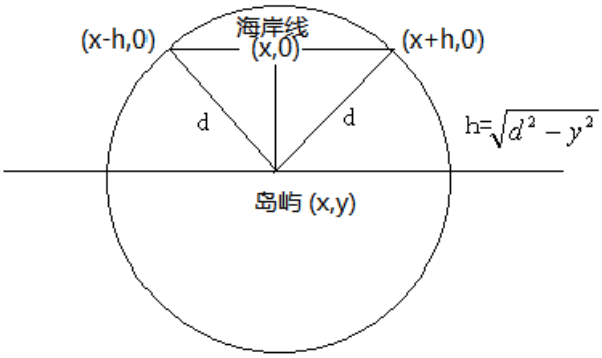

Figure5.2-2

Suppose there are *n* islands. First *n* islands are represented as *n* segments. Second, right endpoints are as the first key(in ascending order), left endpoints are as second key (in ascending order), and the *n* segments are sorted. Finally all sorted segments are scanned one by one. If the current segment isn't covered by a radar, a radar locates at the right endpoint for the segment.

Program

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cmath>
using namespace std;
const int maxn = 1010;  //the upper limit of number of segments
struct tt {
        double l,r;     // left, right pointer
} p[maxn];   // the sequence of segments, where the i-th island is represented as segment [p[i].l, p[i].r]
int n,d;   //n: number of islands, any radar covers d distance
bool flag;
void init( ) {     //Input positions of islands, and calculate corresponding segments
      flag = true;
      int i;
      double x,y;
      for(i = 1 ; i <= n ; ++i){
            scanf("%lf%lf",&x,&y);
            if(d < y){   // if d<y, no solution
                flag = false;
            }
            double h = sqrt(d*d - y*y);
            p[i].l = x - h;
            p[i].r = x + h;
```

```
          }
}
bool cmp (tt a, tt b){  //compare segment a and segment b
      if( b.r - a.r > 10e-7){
                  return true;
        }
      if(abs(a.r - b.r) < 10e-7 && ( b.l - a.l > 10e-7)) {
            return true;
        }
      return false;
}
void work( ) { //Calculate and output the minimal number of radar installations
needed
if( d == -1){ printf("-1\n");  return ;  }
sort(p+1,p+1+n,cmp);  // Sorting segments
int ans = 0;   // Initialize the minimal number of radar installations needed
double last = -10000.0; Initialize the position of radar installation
int i;
for(i = 1 ; i <= n ; ++i){         // search segemnts one by one
      if(p[i].l <= last){   //there is a radar on the segment
              if(p[i].r <= last){
                        last = p[i].r;
                }
      continue;
      }
      ans++;    // a radar is installed on the right endpoint
      last = p[i].r;
}
printf("%d\n",ans);  //Output
}
int main(){
int counter = 1;
while(scanf("%d%d",&n,&d)!=EOF,n||d){ // Input test cases
    printf("Case %d: ",counter++);  // the number of test case
    init();
    if(!flag){
          printf("-1\n");
      }else{
          work();
      }
    }
return 0;
}
```

5.3 Greedy Algorithms used with other Methods to solve P-Problems

In the real world, problems that we try to solve can be classified into two classes:

(1) P-Problems: P-Problems are polynomially solvable problems. That is, a P-Problem can be solved by an algorithm whose running time is bounded by a polynomial.

(2) NP-Complete Problems: NP-Complete Problems can not be solved in polynomial time.

In this section, practices for greedy algorithms used with other methods to solve P-Problems are showed.

【5.3.1 Color a Tree】

Bob is very interested in the data structure of a tree. A tree is a directed graph in which a special node is singled out, called the "root" of the tree, and there is a unique path from the root to each of the other nodes.

Bob intends to color all the nodes of a tree with a pen. A tree has N nodes, these nodes are numbered 1, 2, ..., N. Suppose coloring a node takes 1 unit of time, and after finishing coloring one node, he is allowed to color another. Additionally, he is allowed to color a node only when its father node has been colored. Obviously, Bob is only allowed to color the root in the first try.

Each node has a "coloring cost factor", C_i. The coloring cost of each node depends both on C_i and the time at which Bob finishes the coloring of this node. At the beginning, the time is set to 0. If the finishing time of coloring node i is F_i, then the coloring cost of node i is C_i*F_i.

For example, a tree with five nodes is shown in Figure 5.3-1. The coloring cost factors of each node are 1, 2, 1, 2 and 4. Bob can color the tree in the order 1, 3, 5, 2, 4, with the minimum total coloring cost of 33.

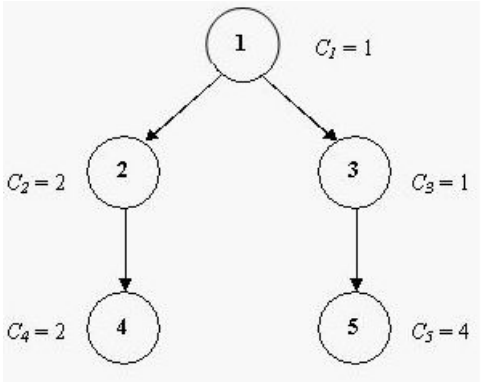

Figure5.3-1

Given a tree and the coloring cost factor of each node, please help Bob to

find the minimum possible total coloring cost for coloring all the nodes.

Input

The input consists of several test cases. The first line of each case contains two integers N and R ($1 \le N \le 1000$, $1 \le R \le N$), where N is the number of nodes in the tree and R is the node number of the root node. The second line contains N integers, the i-th of which is C_i ($1 \le C_i \le 500$), the coloring cost factor of node i. Each of the next N-1 lines contains two space-separated node numbers V_1 and V_2, which are the endpoints of an edge in the tree, denoting that V_1 is the father node of V_2. No edge will be listed twice, and all edges will be listed.

A test case of $N = 0$ and $R = 0$ indicates the end of input, and should not be processed.

Output

For each test case, output a line containing the minimum total coloring cost required for Bob to color all the nodes.

Sample Input	Sample Output
5 1 1 2 1 2 4 1 2 1 3 2 4 3 5 0 0	33

Source: ACM Beijing 2004

IDs for Online Judge: POJ 2054 , ZOJ 2215 , UVA 3138

Analysis

For each node, the coloring cost is based on its coloring cost factor and the time at which Bob finishes coloring it. The coloring cost factor for each node is given. The key to the problem is determining the sequence coloring nodes.

Because Bob is allowed to color a node only when the node's father has been colored, the pointer pointing to its fater for each node should be set up when edges are input. A DFS is used to calculate pointers pointing to its father for each node.

The sequence coloring nodes can be regarded as a merger process. For a father-child relationship (k, x), node x can be colored only after its father k is colored. If there are several children, the sequence coloring nodes should be determined.

Suppose $now[i]$ is the average for coloring cost factors for nodes which are mergered into node i, $cnt[i]$ is the number of nodes which are mergered into node i. Initially $now[i]$= the coloring cost factor for node i, $cnt[i]=1(1 \le i \le n)$.

After node x is colored, it is merged into node k, $now[k]=\frac{now[k]*cnt[k]+now[x]*cnt[x]}{cnt[k]+cnt[x]}$, and $cnt[k]=cnt[k]+cnt[x]$. Such a merge process is performed n-1 times. In each time, the crietia for the merger is selecting a uncolored node whose *now* value is maximal. Obviously it is a greedy strategy. The implementation process is as follows.

n-1 merger processes are run:

Selecting an unmerged node k (isn't the root) whose *now* value is maximal;

Setting the merger mark for node k;

Determining the sequence for coloring node k and its fater f;

Searching node f which is the nearest for k and isn't mergered based on the father pointer for k, and adjusting $now[f]$ and $cnt[f]$;

From the root, based on the coloring sequence, calculating the minimum total coloring cost required for Bob to color all the nodes.

$$ans=\sum_{i=1}^{n} i*\text{the coloring cost factor for node } i \text{ in the coloring sequence .}$$

Program

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

const int maxN=1100;      // the upper limit for the number of nodes
int root,n,fa[maxN],l[maxN],next[maxN],cnt[maxN],c[maxN],e[maxN][maxN];
// root: the root of the tree; n: the number of nodes; fa[ ]: each node's
  father; next[ ]: the coloring sequence, where the node is colored after node
  x is colored is node next[x]; cnt[ ]: the number of mergered nodes for each
  node; c[ ]: coloring cost factor; e[ ][ ]: the adjacency matrix for the tree
double now[maxN];     //now[ ]: coloring costs for nodes after merger
void init()             // Input coloring cost factors for n nodes and edges,
and construct e[ ][ ]
{
    int x,y;
    memset(e,0,sizeof(e));
    for (int i=1;i<=n;i++) scanf("%d",&c[i]);
```

```
    for (int i=1;i<n;i++) { scanf("%d%d",&x,&y);e[x][++e[x][0]]=y;e[y][++e[y]
[0]]=x;}
}
void dfs(int x)               //calculating the pointer pointing to its father
    for each node
{
    int y;
    for (int i=1;i<=e[x][0];i++)          // for each child of x, setting its
father pointer x
    {
        y=e[x][i];
        if (fa[y]==0) { fa[y]=x;dfs(y);}
    }
}
void addedge(int x,int y)   //determine the coloring sequence for x and y, that
is, y is colored after x is colored
{
    while (next[x]) x=next[x];
    next[x]=y;
}

void work()              //calculate and output the minimum total coloring cost
{
    memset(fa,0,sizeof(fa));             //initialization
    fa[root]=-1;
  dfs(root);     // Traverse the tree whose root is root, and determine father-
  children relationships
    for (int i=1;i<=n;i++) now[i]=c[i];    // initialization
    bool flag[maxN];                  // marks for mergering nodes
    int k,f;
    double max;
    memset(flag,1,sizeof(flag));  memset(next,0,sizeof(next));
    for (int i=1;i<=n;i++) cnt[i]=1;
    for (int i=1;i<n;i++)               // n-1 merger processes
    {
        max=0;    // Selecting an unmerged node k (isn't the root) whose now
value is maximal
        for   (int   j=1;j<=n;j++)   if   ((j!=root)&&(flag[j])&&(max<now[j]))
{ max=now[j];k=j;}          f=fa[k];addedge(f,k);   // Determining the sequence
for coloring node k and its fater f;
        while (!flag[f]) f=fa[f];           // Searching node f which is the
nearest for k and isn't mergered based on the pointer pointing to the father
for k, that is the father node for k after merger
        flag[k]=false;               // Set the merger mark for node k
```

```
        now[f]=(now[f]*cnt[f]+now[k]*cnt[k])/(cnt[f]+cnt[k]);      // adjusting
now[f]
        cnt[f]+=cnt[k];             // adjusting cnt[f]
    }
    int p=root,ans=0;         // calculate minimum total coloring cost
    for (int i=1;i<=n;i++)
    {
        ans+=i*c[p];p=next[p];
    }
    printf("%d\n",ans);       // output the minimum total coloring cost
}
int main()
{
    while (scanf("%d%d",&n,&root),n+root)    //Input
    {
        init();
        work();              // calculate and output the minimum total coloring
cost
    }
    return 0;
}
```

【5.3.2 Copying Books 】

Before the invention of book-printing, it was very hard to make a copy of a book. All the contents had to be re-written by hand by so called scribers. The scriber had been given a book and after several months he finished its copy. One of the most famous scribers lived in the 15th century and his name was Xaverius Endricus Remius Ontius Xendrianus (Xerox). Anyway, the work was very annoying and boring. And the only way to speed it up was to hire more scribers.

Once upon a time, there was a theater ensemble that wanted to play famous Antique Tragedies. The scripts of these plays were divided into many books and actors needed more copies of them, of course. So they hired many scribers to make copies of these books. Imagine you have m books (numbered 1, 2 ... m) that may have different number of pages (p_1, p_2, ……, p_m) and you want to make one copy of each of them. Your task is to divide these books among k scribes, $k \le m$. Each book can be assigned to a single scriber only, and every scriber must get a continuous sequence of books. That means, there exists an increasing succession of numbers $0=b_0<b_1<b_2<\ldots<b_{k-1}\le b_k=m$ such that i-th scriber gets a sequence of books with numbers between $b_{i-1}+1$ and b_i. The time needed to make a copy of all the books is determined by the scriber who was assigned the most work. Therefore, our goal is to minimize the maximum number of pages assigned to a single scriber. Your task is to find the optimal assignment.

Input

The input consists of N cases. The first line of the input contains only positive integer N. Then follow the cases. Each case consists of exactly two lines. At the first line, there are two integers m and k, $1 \le k \le m \le 500$. At the second line, there are integers p_1, p_2, ... p_m separated by spaces. All these values are positive and less than 10000000.

Output

For each case, print exactly one line. The line must contain the input succession p_1, p_2, ... p_m divided into exactly k parts such that the maximum sum of a single part should be as small as possible. Use the slash character ('/') to separate the parts. There must be exactly one space character between any two successive numbers and between the number and the slash.

If there is more than one solution, print the one that minimizes the work assigned to the first scriber, then to the second scriber etc. But each scriber must be assigned at least one book.

Sample Input	Sample Output
2 9 3 100 200 300 400 500 600 700 800 900 5 4 100 100 100 100 100	100 200 300 400 500 / 600 700 / 800 900 100 / 100 / 100 / 100 100

Source: ACM Central European Regional Contest 1998

IDs for Online Judge: POJ 1505 , ZOJ 2002 , UVA 714

Analysis

Binary search can be used to solve the problem. If the current maximum number of pages assigned to a single scriber x is feasible, we can reduce it; otherwise, we can increase it.

The key to the problem is to determine whether the current maximum number of pages assigned to a single scriber x is feasible or not. Because numbers of pages assigned to scribers are increasing from left to right, the greedy strategy is as follows. From back to front every book is scanned, the crietia that the current book can be assigned to the current scriber is that after the book is assigned to the scriber the sum of numbers of pages assigned to the scriber isn't more than x and every remainder scriber can be assigned at least one book. If the current book meets the crietia, the book is assigned to the current scriber, else the book is assigned to a new scriber, and the new scriber becomes the current scriber. A slash character ('/') is used to separate the two scribers' work.

Obviously, if k scribers can't finish copys for m books, then the current maximum number of pages assigned to a single scriber x isn't feasible; else the

current maximum number of pages assigned to a single scriber *x* is feasible.

Binary search is used to find the minimal maximum number of pages assigned to a single scriber *min*. The above greedy algorithm is used to find the optimal assignment.

Program

```
#include<iostream>
#include<cstdlib>
#include<cstdio>
#include<cmath>
#include<cstring>
#include<algorithm>
using namespace std;
const int maxN=510;           //the upper limit of the number of books
int n,m,a[maxN];              //n books, m scribers, a[]:the sequence of books
long long sum;                // sum of pages
bool flag[maxN];              // flag to separate books


void init()                    //Input the current test case
{
    sum=0;
    scanf("%d%d",&n,&m);    // Input numbers of books and scribers
    for (int i=1;i<=n;i++)       // Input the numbers of pages, and sum
    {
        scanf("%d",&a[i]);sum+=a[i];
    }
}

bool judge(long long lmt)          // determine whether the current maximum number of pages assigned to a single scriber lmt is feasible or not
{    // determine whether the ith book needs a new scriber or not: isn't more than lmt, every remainder scriber can be assigned at least one book
// from back to front
    memset(flag,0,sizeof(flag));
    int cnt=m;                        // start from the mth scriber
    long long now=0;                  //number of pages for the current scriber
    for (int i=n;i;i--)                   // scan books
    {
        if ((now+a[i]>lmt)||(i<cnt))           // large than lmt, or every remainder scriber can't be assigned at least one book
        {
```

```
            now=a[i];cnt--;flag[i]=true;    // add a new scriber
            if (cnt==0) return false;       //need more scribers, for lmt
        }
        else now+=a[i];             // accumulation
    }
    return true;                        // lmt is feasible
}

void work()                  //calculate and output the solution to the current
test case
{
    long long l=0,r=sum,mid;              // initial interval [l, sum], middle
pointer mid
    for (int i=1;i<=n;i++) if (l<a[i]) l=a[i];       // the maximal number of
pages in these books
    while (l!=r)                           //Binary search in [l, r]
    {
        mid=(l+r)>>1;                     // middle pointer mid
        if (judge(mid)) r=mid;else l=mid+1;  // if mid is feasible, left sub-
interval; else right sub-interval
    }
    judge(l);                 //calculate

    for(int i=1;i<=n;i++)                   // output
    {
        printf("%d",a[i]);
        if (i<n) printf(" ");
        if (flag[i]) printf("/ ");
    }
    printf("\n");
}
int main()
{
    int t;
    scanf("%d",&t);            // the number of test cases
    for (int i=1;i<=t;i++)        // deal with every test case
    {
        init();                // the ith test case
        work();                //calculate and output the solution to the ith
test case
    }
    return 0;
}
```

5.4 Problems

【5.4.1 Stripies】

Our chemical biologists have invented a new very useful form of life called stripies (in fact, they were first called in Russian - polosatiki, but the scientists had to invent an English name to apply for an international patent). The stripies are transparent amorphous amebiform creatures that live in flat colonies in a jelly-like nutrient medium. Most of the time the stripies are moving. When two of them collide a new stripie appears instead of them. Long observations made by our scientists enabled them to establish that the weight of the new stripie isn't equal to the sum of weights of two disappeared stripies that collided; nevertheless, they soon learned that when two stripies of weights m_1 and m_2 collide the weight of resulting stripie equals to 2*sqrt(m_1*m_2). Our chemical biologists are very anxious to know to what limits can decrease the total weight of a given colony of stripies.

You are to write a program that will help them to answer this question. You may assume that 3 or more stipies never collide together.

Input

The first line of the input contains one integer N ($1 \le N \le 100$) - the number of stripies in a colony. Each of next N lines contains one integer ranging from 1 to 10000 - the weight of the corresponding stripie.

Output

The output must contain one line with the minimal possible total weight of colony with the accuracy of three decimal digits after the point.

Sample Input	Sample Output
3 72 30 50	120.00

Source: ACM Northeastern Europe 2001, Northern Subregion

IDs for Online Judge: POJ 1862 , ZOJ 1543 , Ural 1161

Hint

Suppose weights of n stripies are m_1, m_2, …, m_n, respectively. After n-1 collisions the total weight of the colony is as follows.

$$W = 2^{n-1}((m_1 m_2)^{\frac{1}{2^{n-1}}} m_3^{\frac{1}{2^{n-2}}} \cdots m_n^{\frac{1}{2}}).$$

Obviously, if m_1, m_2, …, m_n are sorted in ascending order, the total weight of the colony W is minimal.

【5.4.2 The Product of Digits】

Your task is to find the minimal positive integer number Q so that the product of digits of Q is exactly equal to N.

Input

The input contains the single integer number N ($0 \le N \le 10^9$).

Output

Your program should print to the output the only number Q. If such a number does not exist print −1.

Sample Input	Sample Output
10	25

Source: USU Local Contest 1999

IDs for Online Judge: Ural 1014

Hint

The crietia for factorization of N is to produce factors as big as possible.

There are two special cases: If N==0, then Q=0; and if N==1 then Q=1.

Otherwise the greedy strategy is used as follows. N is factorized from 9 to 2. First, factors 9 are produced as more as possible; second, factors 8 are produced as more as possible; ……; and so on. If the final result for the factorization is not 1, then there is no solution; else Q is the positive integer that factors are listed from small to large.

【5.4.3 Democracy in Danger】

In one of the countries of Caribbean basin all decisions were accepted by the simple majority of votes at the general meeting of citizens (fortunately, there were no lots of them). One of the local parties, aspiring to come to power as lawfully as possible, got its way in putting into effect some reform of the election system. The main argument was that the population of the island recently had increased and it was to longer easy to hold general meetings.

The essence of the reform is as follows. From the moment of its coming into effect all the citizens were divided into K (may be not equal) groups. Votes on every question were to be held then in each group, moreover, the group was said to vote “for” if more than half of the group had voted “for”, otherwise it was said to vote “against”. After the voting in each group a number of group that had voted “for” and “against” was calculated. The answer to the question was positive if the number of groups that had voted “for” was greater than the half of the general number of groups.

At first the inhabitants of the island accepted this system with pleasure. But when the first delights dispersed, some negative properties became obvious. It appeared that supporters of the party, that had introduced this system, could influence upon formation of groups of voters. Due to this they had an opportunity to put into effect some decisions without a majority of voters

“for” it.

Let’s consider three groups of voters, containing 5, 5 and 7 persons, respectively. Then it is enough for the party to have only three supporters in each of the first two groups. So it would be able to put into effect a decision with the help of only six votes “for” instead of nine, that would be necessary in the case of general votes.

You are to write a program, which would determine according to the given partition of the electors the minimal number of supporters of the party, sufficient for putting into effect of any decision, with some distribution of those supporters among the groups.

Input

In the first line an only odd integer K – a quantity of groups – is written ($1 \le K \le 101$). In the second line there are written K odd integers, separated with a space. Those numbers define a number of voters in each group. The population of the island does not exceeds 9999 persons.

Output

You should write a minimal quantity of supporters of the party, that can put into effect any decision.

Sample Input	Sample Output
3 5 7 5	6

Source: Autumn School Contest 2000

IDs for Online Judge: Ural 1025

Hint

K groups are sorted in ascending order of numbers of voters in groups. There are K groups. Therefore the party needs $\left\lfloor \frac{K}{2} \right\rfloor$ +1 groups voting “for”. If there are n voters in a group, the party needs $\left\lfloor \frac{n}{2} \right\rfloor$ +1 supporters in the group. Therefore the minimal quantity of supporters of the part is there are just over half supporters for the party in the first $\left\lfloor \frac{K}{2} \right\rfloor$ +1 groups.

【5.4.4 Box of Bricks】

Little Bob likes playing with his box of bricks. He puts the bricks one upon another and builds stacks of different height. "Look, I've built a wall!", he tells his older sister Alice. "Nah, you should make all stacks the same height. Then you would have a real wall.", she retorts. After a little con-

sideration, Bob sees that she is right. So he sets out to rearrange the bricks, one by one, such that all stacks are the same height afterwards. But since Bob is lazy he wants to do this with the minimum number of bricks moved. Can you help?

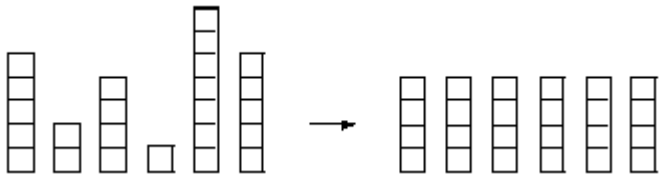

Figure5.4-1

Input

The input consists of several data sets. Each set begins with a line containing the number n of stacks Bob has built. The next line contains n numbers, the heights h_i of the n stacks. You may assume $1 \le n \le 50$ and $1 \le h_i \le 100$.

The total number of bricks will be divisible by the number of stacks. Thus, it is always possible to rearrange the bricks such that all stacks have the same height.

The input is terminated by a set starting with $n = 0$. This set should not be processed.

Output

For each set, first print the number of the set, as shown in the sample output. Then print the line "The minimum number of moves is k.", where k is the minimum number of bricks that have to be moved in order to make all the stacks the same height.

Output a blank line after each set.

Sample Input	Sample Output
6 5 2 4 1 7 5 0	Set #1 The minimum number of moves is 5.

Source: ACM Southwestern European Regional Contest 1997

IDs for Online Judge: POJ 1477 , ZOJ 1251 , UVA 591

Hint

Suppose the average value $avg = \frac{\sum_{i=1}^{n} h_i}{n}$. That is, avg is the heights of the n stacks after bricks are moved.

The crietia that bricks in the ith stack should be moved is as follows. If $h_i > avg$, then $h_i - avg$ bricks should be moved in the stack. Therefore the minimum

number of bricks that have to be moved is $ans=\sum_{i=1}^{n}(h_i - avg | h_i > avg)$.

【5.4.5 Minimal coverage】

Given several segments of line (int the X axis) with coordinates [L_i, R_i]. You are to choose the minimal amount of them, such they would completely cover the segment [0, M].

Input

The first line is the number of test cases, followed by a blank line.

Each test case in the input should contains an integer M($1 \le M \le 5000$), followed by pairs "L_i R_i"($|L_i|$, $|R_i| \le 50000$, $i \le 100000$), each on a separate line. Each test case of input is terminated by pair "0 0".

Each test case will be separated by a single line.

Output

For each test case, in the first line of output your programm should print the minimal number of line segments which can cover segment [0, M]. In the following lines, the coordinates of segments, sorted by their left end (L_i), should be printed in the same format as in the input. Pair "0 0" should not be printed. If [0, M] can not be covered by given line segments, your programm should print "0"(without quotes).

Print a blank line between the outputs for two consecutive test cases.

Sample Input	Sample Output
2 1 -1 0 -5 -3 2 5 0 0 1 -1 0 0 1 0 0	0 1 0 1

Source: USU Internal Contest March'2004

IDs for Online Judge: UVA 10020 , Ural 1303

Hint

All segments are sorted in ascending order of left ends as the first key, and right ends as the second key (($L_i \le L_{i+1}$|| ((L_i == L_{i+1})&&(R_i< R_{i+1})), $1 \le i \le$the number of segments -1).

The crietia selecting segments is selecting a segment whose right end is the farthest among segemnts whose left ends are covered.

The greedy algorithm is as follows.

Suppose *now* is the end position that the current segment covers; *len* is the farthest position that a segment *k* whose left end is covered can reach. Initially *ans=now=len=*0.

Every segment in the sorted sequence is analyzed one by one:

if ((*L_i*≤*now*)&&(*len*< *R_i*)) { *len*= *R_i*; *k=i*;}

if ((*L_{i+1}* >*now*) &&(*now*<*len*)) { *now=len*; segment *k* is as a new covered segment; }

if (*now*≥*m*) output the result and exit;

If *now*<*m* after all segments are analyzed, then [0, *M*] can not be covered by given segments.

【5.4.6 Annoying painting tool】

Maybe you wonder what an annoying painting tool is? First of all, the painting tool we speak of supports only black and white. Therefore, a picture consists of a rectangular area of pixels, which are either black or white. Second, there is only one operation how to change the colour of pixels:

Select a rectangular area of ***r*** rows and ***c*** columns of pixels, which is completely inside the picture. As a result of the operation, each pixel inside the selected rectangle changes its colour (from black to white, or from white to black).

Initially, all pixels are white. To create a picture, the operation described above can be applied several times. Can you paint a certain picture which you have in mind?

Input

The input contains several test cases. Each test case starts with one line containing four integers ***n***, ***m***, ***r*** and ***c***. (1 ≤ *r* ≤ *n* ≤ 100, 1 ≤ *c* ≤ *m* ≤ 100), The following ***n*** lines each describe one row of pixels of the painting you want to create. The ***i***th line consists of ***m*** characters describing the desired pixel values of the ***i***th row in the finished painting ('0' indicates white, '1' indicates black).

The last test case is followed by a line containing four zeros.

Output

For each test case, print the minimum number of operations needed to create the painting, or -1 if it is impossible.

Sample Input	Sample Output
3 3 1 1 010 101 010 4 3 2 1	4 6 -1

011 110 011 110 3 4 2 2 0110 0111 0000 0 0 0 0	

Source: Ulm Local 2007

IDs for Online Judge: POJ 3363

Hint

The first thing to realise is that in an optimal solution, the painting operation is never applied more than once at the same position. Also, it doesn't matter in which order the operations are done, therefore we can do the painting operations from top to bottom, and from left to right.

Using these ideas we can check easily if a painting operation at some position is required or not. Since the pixel in the top left corner of a selected area for the painting operation will not be changed by later operations, we just check if it already has the required colour. If its colour still needs to be changed, we have to apply the painting operation.

After we have applied all the painting operations, we need to check the pixels in the rightmost *m*-*c* columns and bottom *n*-*r* rows if they have their required colour. If one of these pixels doesn't have its required colour, it is impossible to create the painting.

Since the size of the picture is at most 100 × 100, a naive implementation with O(*n*^4) runs in time. There exists an optimal solution which runs in O(*n***m*). The idea is to store how many operations have been applied with the top left corner in one of the first i rows and j columns. With this stored data it is possible to answer in constant time how many operations covering a pixel have been applied.

【5.4.7 Troublemakers】

Every school class has its troublemakers - those kids who can make the teacher's life miserable. On his own, a troublemaker is manageable, but when you put certain pairs of troublemakers together in the same room, teaching a class becomes very hard. There are ***n*** kids in Mrs. Shaida's math class, and there are ***m*** pairs of troublemakers among them. The situation has gotten so bad that Mrs. Shaida has decided to split the class into two classes. Help her do it in such a way that the number of troublemaker pairs is reduced by at least a half.

输入：

The first line of input gives the number of cases, ***N***. ***N*** test cases follow. Each one starts with a line containing ***n*** (0≤***n***≤100) and ***m*** (0<***m***<5000). The next ***m*** lines will contain a pair of integers ***u*** and ***v*** meaning that when kids ***u*** and ***v*** are in the same room, they make a troublemaker pair. Kids are numbered from 1 to ***n***.

输出：

For each test case, output one line containing "Case #***x***:" followed by ***L*** - the number of kids who will be moved to a different class (in a different room). The next line should list those kids. The total number of troublemaker pairs in the two rooms must be at most ***m***/2. If that is impossible, print "Impossible." instead of ***L*** and an empty line afterwards.

Sample Input	Sample Output
2 4 3 1 2 2 3 3 4 4 6 1 2 1 3 1 4 2 3 2 4 3 4	Case #1: 3 1 3 4 Case #2: 2 1 2

Source: Abednego's Graph Lovers' Contest, 2006

IDs for Online Judge: UVA 10982

Hint

A graph is used to represent the problem, where kids in Mrs. Shaida's math class are represented as vertices, and there are edges between each pair of troublemakers. Mrs. Shaida splits the class into two classes, $s[0]$ and $s[1]$, where the number of kids in $s[1]$ is less than the number of kids in $s[0]$.

The method that Mrs. Shaida splits the class into two classes is as follows.

For kid i ($1 \le i \le n$), numbers of kids among kid 1 to kid i-1 who constitute pair of troublemakers with kid i in $s[0]$ and $s[1]$ are calculated. If such number in $s[1]$ is less than such number in $s[0]$, kid i is moved to $s[1]$, else kid i stays in $s[0]$.

The greedy algorithm is as follows.

For (i=1; $i \le n$; i++)

Calculate numbers of vertices which connect with vertice i in $s[0]$ and $s[1]$ from vertice 1 to vertice i-1;

if (the number of such vertices in *s*[1] < the number of such vertices in *s*[0])

vertice *i* is moved to *s*[1];

Finally vertices in *s*[1] corresponds to kids moved to a different class (in a different room).

【5.4.8 Constructing BST】

BST (Binary Search Tree) is an efficient data structure for searching. In a BST all the elements of the left sub-tree are smaller and those of right sub-tree are greater than the root. A typical example of BST is as figure 5.4-2.

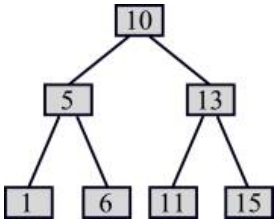

Figure5.4-2

Normally, we construct BST by successively inserting an element. In that case, the ordering of elements has great impact on the structure of the tree. Look at the following cases:

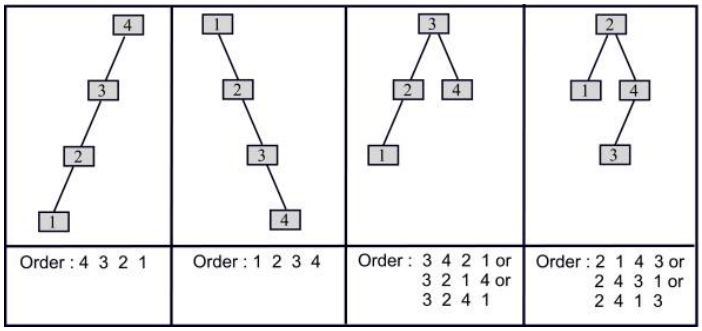

Figure5.4-3

In this problem, you have to find the order of 1 to *N* integers such that the BST constructed by them has height of at most *H*. The height of a BST is defined by the following relation

1. BST having no node has height 0.
2. Otherwise, it is equal to the maximum of the height of the left sub-tree and right sub-tree plus 1.

Again, several orderings can satisfy the criterion. In that case we prefer the sequence where smaller numbers come first. For example, for *N* = 4, *H* = 3 we want the sequence 1 3 2 4 rather than 2 1 4 3 or 3 2 1 4.

Input

Each test case starts with two positive integers *N* (1≤*N*≤10000) and *H* (1≤*H*≤30). Input is terminated by *N* = 0, *H* = 0. This case should not be processed. There can be at most 30 test cases.

Output

Output of each test case should consist of a line starting with 'Case #: ' where # is the test case number. It should be followed by the sequence of *N* integers in the same line. There must not be any trailing space at the end of the line. If it is not possible to construct such tree then print "Impossible.". (without the quotes).

Sample Input	Sample Output
4 3 4 1 6 3 0 0	Case 1: 1 3 2 4 Case 2: Impossible. Case 3: 3 1 2 5 4 6

Source: ACM ICPC World Finals Warmup 1 , 2005

IDs for Online Judge: UVA 10821

Hint

The problem requires to output the Pre-order Traversal of a BST. Because in the sequence smaller numbers come first, the number for the root is as smaller as possible.

A BST with the height of at most *H* is constructed by the order of 1 to *N* integers. The number of nodes in its left subtree and right subtree is no more than 2^{H-1}-1. The crietia for the number of the root is as follows.

If the right subtree is a full subtree, the number for the root is *N*-(2^{H-1}-1); else the number for the root is 1.

Then the problem is transferred a BST with the height of at most *H*-1 is constructed by the order of 1 to *root*-1 integers, the BST is as the left subtree; and a BST with the height of at most *H*-1 is constructed by the order of *root*+1 to *N*, the BST is as the right subtree.

Obviously the greedy algorithm is a recursive algorithm.

Program

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
#include<cmath>
#include<cstring>
#include<algorithm>
using namespace std;
const int maxN=31;
int n,h,cnt,s[maxN];//s[i]: the number of nodes for a full binary treeth height
i
//function work is an inorder traversal for a BST with height at most h, nodes
from l to r
void work(int l,int r,int h)
{
```

```
    int m=max(l,r-s[h-1]);  //the number of root is as small as possible
    printf("%d",m);
    if (++cnt<n) printf(" ");
    if (l<m) work(l,m-1,h-1);   // recursion for the left subtree
    if (r>m) work(m+1,r,h-1);   // recursion for the right subtree
}
int main()
{
    for (int i=0;i<=30;i++) s[i]=(1<<i)-1;
    int t=0;
    while (scanf("%d%d",&n,&h),n+h)
    {
        cnt=0;
        printf("Case %d: ",++t);
        if (s[h]<n) printf("Impossible.");else work(1,n,h);
        printf("\n");
    }
    return 0;
}
```

【5.4.9 Gone Fishing】

John is going on a fishing trip. He has h hours available ($1 \le h \le 16$), and there are n lakes in the area ($2 \le n \le 25$) all reachable along a single, one-way road. John starts at lake 1, but he can finish at any lake he wants. He can only travel from one lake to the next one, but he does not have to stop at any lake unless he wishes to. For each $i = 1, \ldots, n - 1$, the number of 5-minute intervals it takes to travel from lake i to lake $i + 1$ is denoted t_i ($0 < t_i \le 192$). For example, $t_3 = 4$ means that it takes 20 minutes to travel from lake 3 to lake 4. To help plan his fishing trip, John has gathered some information about the lakes. For each lake i, the number of fish expected to be caught in the initial 5 minutes, denoted f_i ($f_i \ge 0$), is known. Each 5 minutes of fishing decreases the number of fish expected to be caught in the next 5-minute interval by a constant rate of d_i ($d_i \ge 0$). If the number of fish expected to be caught in an interval is less than or equal to d_i, there will be no more fish left in the lake in the next interval. To simplify the planning, John assumes that no one else will be fishing at the lakes to affect the number of fish he expects to catch.

Write a program to help John plan his fishing trip to maximize the number of fish expected to be caught. The number of minutes spent at each lake must be a multiple of 5.

Input

You will be given a number of cases in the input. Each case starts with a line containing n. This is followed by a line containing h. Next, there is a

line of n integers specifying f_i ($1 \le i \le n$), then a line of n integers d_i ($1 \le i \le n$), and finally, a line of n - 1 integers t_i ($1 \le i \le n$ - 1). Input is terminated by a case in which n = 0.

Output

For each test case, print the number of minutes spent at each lake, separated by commas, for the plan achieving the maximum number of fish expected to be caught (you should print the entire plan on one line even if it exceeds 80 characters). This is followed by a line containing the number of fish expected.

If multiple plans exist, choose the one that spends as long as possible at lake 1, even if no fish are expected to be caught in some intervals. If there is still a tie, choose the one that spends as long as possible at lake 2, and so on. Insert a blank line between cases.

Sample Input	Sample Output
2 1 10 1 2 5 2 4 4 10 15 20 17 0 3 4 3 1 2 3 4 4 10 15 50 30 0 3 4 3 1 2 3 0	45, 5 Number of fish expected: 31 240, 0, 0, 0 Number of fish expected: 480 115, 10, 50, 35 Number of fish expected: 724

Source: ACM East Central North America 1999

IDs for Online Judge: POJ 1042 , UVA 757

Hint

Obviously, in the solution there is no turning back. That is, in John's fishing trip, if John fish at a lake and leave the lake, he cann't be back to the lake.

Suppose John finishes the trip at lake *ed*. How can we calculate the maximum number of fish expected to be caught at lake *ed*?

The crietia selecting a lake is as follows.

If the time is allowed, the lake in which there are maximum number of fish is selected. The greedy algorithm is as follows.

Initially, for lake i, the number of fish expected to be caught $f2[i]$ is the number of fish expected to be caught in the initial 5 minutes f_i, the time that John fishes at the lake $tt[i]$ is 0, $1 \le i \le ed$. The time that John can fish is $h2=h-\sum_{i=1}^{ed} t_i$, for there is no turning back in his fishing trip. The current number of fish to be caught $now=0$.

Then, for each terminal ed, repeat the following steps until $h2 \le 0$:

Search a lake p in which there are maximum number of fish, that is, $f2[p]=\max_{1 \le i \le ed}\{f2[i]\}$;

$h2-=5$;

$tt[p]+=5$;

$now+=f2[p]$;

the number of fish to be caught in lake p is adjusted $f2[p]$ =max($f2[p]-d_p$, 0);

Finally, if ($ans<now$), then $ans=now$, and $ans\_tt[\ ]$ is adjusted;

Obviously, after every terminal ed is enumerated ($1 \le ed \le n$), the number of minutes spent at each lake, and the plan achieving the maximum number of fish expected to be caught can be gotten. That is, ans is the maximum number of fish expected to be caught, and $ans\_tt[\ ]$ is the number of minutes spent at each lake.

Program

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
#include<cmath>
#include<cstring>
#include<algorithm>
using namespace std;
const int maxN=30;
int n,h,f[maxN],d[maxN],t[maxN];
int f2[maxN],tt[maxN],ans,ans_tt[maxN]; //f2 is the same as f, tt is the time that John fishes at the lake; ans_tt is the tt when ans is maximal
void init()
{
// Initialization
    ans=-1;
    memset(t,0,sizeof(t));
    memset(f,0,sizeof(f));
    memset(d,0,sizeof(d));
    memset(ans_tt,0,sizeof(ans_tt));
//Input
    scanf("%d",&h);h*=60;//h is transferred into minutes
```

```
    for (int i=1;i<=n;i++) scanf("%d",&f[i]);
    for (int i=1;i<=n;i++) scanf("%d",&d[i]);
    for (int i=1;i<n;i++) { scanf("%d",&t[i]);t[i]+=t[i-1];}
}
//function work : calculate the maximum number of fish expected to be caught at lake ed
void work(int ed)
{
    memcpy(f2,f,sizeof(f));
    memset(tt,0,sizeof(tt));
    int now=0,h2=h; //now: the current number of fish to be caught; h2: the time that John can fish
    h2-=t[ed-1]*5;  // the number of minutes spent from lake 1 to lake ed
    f2[0]=-1;
    while (h2>0)    // each while correspond to a five-minute-fishing
    {
        int p=0;
        h2-=5;  // spend 5 minutes
        for (int i=1;i<=ed;i++) // search a lake p in which there are maximum number of fish
            if (f2[p]<f2[i]) p=i;
        tt[p]+=5;
        now+=f2[p];f2[p]=max(f2[p]-d[p],0); // accumulation
    }
    if (ans<now) { ans=now;memcpy(ans_tt,tt,sizeof(tt));}
}
//output the result
void print()
{
    for (int i=1;i<=n;i++)
    {
        printf("%d",ans_tt[i]);
        if (i<n) printf(", ");
    }
    printf("\nNumber of fish expected: %d\n\n",ans);
}
int main()
{
    while (scanf("%d",&n),n)
    {
        init();
        for (int i=1;i<=n;i++)  // every terminal is enumerated
            work(i);
        print();
```

```
    }
    return 0;
}
```