The bLaCk book of DP

Compiled by WA-c

Version 0.1

January 2009

Foreward

- Don't know what's DP? I'm afraid, this document is not for you at the moment. Read Topcoder tutorial. Do solve some easy/easy-medium DP problems. When you feel comfortable at those, reading this will be meaningful & fun!

- There are 60 problem related to dynamic programming in this compilation of problem and analysis. The number of problems of each category is as following:

Standard DP -	5
Knapsack -	9
1D State DP (linear DP) -	6
Interval Based DP (MCM Type) -	10
Large state DP (Optimisation) -	14
BFS in DP state-space (State BFS) -	4
Row by row DP -	3
Tree based DP –	5
Geometry based DP -	2
DP with convex hulls -	2
Total -	60

- All the problems in this book is sorted by (difficulty, fun) (difficulty first, tie-breaking by fun) in ascending order. All problems come with analyis and solution code. The difficulty and fun value is written beside each problem name.

- Most of the problems are indeed hard, they require insight & ingenuity. So the deserve some time. Don't spoil this opportunity to think about these nice problems by seeing the solutions too early. Work sample cases by hand. Try over and over again. Don't lose hope. Be open minded.

- As the problems are from various USACO contests, so chances are high that they are on PKU, SPOJ etc online judge. Google by contest name, problem name to find them. If you can't find a problem, then bad luck ☹. Generate some random testdata, generate output with sample solution and test them with your solution. Don't lose your enthusiasm ☺.

- In future all the test-datas will come along this document.

- From next edition, this file will contain a table of content. And also, the analyses will be at the end of the book.

NOTE: Bitmask DP (also known as profile DP in some countries) and DP with datastructures (BST, BIT)

not included. (I hope, I'll be able to include some in future).

All credit goes to the problem and analysis authors. Problem author/source, the name of the analysis author are shown at their respective places.

STANDARD DP

1. Apple Catching [Hal Burch, 2004] (17, 31) NOV 04 QUAL

It is a little known fact that cows love apples. Farmer John has two apple trees (which are conveniently numbered 1 and 2) in his field, each full of apples. Bessie cannot reach the apples when they are on the tree, so she must wait for them to fall. However, she must catch them in the air since the apples bruise when they hit the ground (and no one wants to eat bruised apples). Bessie is a quick eater, so an apple she does catch is eaten in just a few seconds.

Each minute, one of the two apple trees drops an apple. Bessie, having much practice, can catch an apple if she is standing under a tree from which one falls. While Bessie can walk between the two trees quickly (in much less than a minute), she can stand under only one tree at any time. Moreover, cows do not get a lot of exercise, so she is not willing to walk back and forth between the trees endlessly (and, because of this indolence, thus misses some apples).

Apples fall (one each minute) for T (1 <= T <= 1,000) minutes. Bessie is willing to walk back and forth at most W (1 <= W <= 30) times. Given which tree will drop an apple each minute, determine the maximum number of apples which Bessie can catch. Bessie starts at tree 1.

PROBLEM NAME: bcatch

INPUT FORMAT:

* Line 1: Two space separated integers: T and W

* Lines 2..T+1: 1 or 2: the tree that will drop an apple each minute.

SAMPLE INPUT (file bcatch.in):

```
7 2
2
1
1
2
2
1
1
```

INPUT DETAILS:

Seven apples fall - one from tree 2, then two in a row from tree 1, then two in a row from tree 2, then two in a row from tree 1. Bessie is willing to walk from one tree to the other twice.

OUTPUT FORMAT:

* Line 1: The maximum number of apples Bessie can catch without
        walking more than W times.

SAMPLE OUTPUT (file bcatch.out):

6

OUTPUT DETAILS:

Bessie can catch six apples by staying under tree 1 until the first two
have dropped, then moving to tree 2 for the next two, then returning back
to tree 1 for the final two.

Analysis: Apple Catching by Bruce Merry

The bcatch problem can be solved with dynamic programming. At any point in time, the only information from the past that affects what Bessie can do in the future is the number of times she crossed (note that the number of crossings also tells us which tree Bessie is currently under). So for each minute, we compute the maximum number of apples she could catch for each possible number of crossings. For a particular minute and number of crossings, there are only two possibilities:
1. Bessie has stood still for the last minute.
2. Bessie crossed from the other tree during the last minute.

For each case, we can look up how many apples she would have caught in the previous state and choose the larger of the two. Then we add one if Bessie is standing under the right tree to catch an apple.
Running time: O(TW)
Memory: O(W), since we only need to keep the information about the current and previous minutes.
Chinese student Kevin Ma's solution was particularly brief:

```
#include<stdio.h>

#define MAX(a,b) a>b?a:b

int main () {
    int t, w, i, j;
    int cat[35][2]={0}, apple[1000]={0};

    FILE *fp = fopen("bcatch.in","r");

    fscanf(fp,"%d %d\n",&t,&w);

    for(i = 0; i < t; i++) {
        fscanf(fp,"%d\n",&apple[i]);
        apple[i]--;

        for(j = 0; j <= w; j++)  {
            cat[j][apple[i]]++;
            if(j > 0 && cat[j-1][1-apple[i]] + 1 >
at[j][apple[i]])
                cat[j][apple[i]] = cat[j-1][1-apple[i]] + 1;
        }
    }

    fp=fopen("bcatch.out","w+");
    fprintf(fp,"%d\n", MAX(cat[w][0], cat[w][1]));
```

```
    return 0;
}
```

2. Books [BulgarianOI, 2005] (18, 28)

A sequence of N (1 <= N <= 200) books on a bookshelf contains respectively s_1, s_2, ..., s_N pages (1 <= s_i <= 10,000,000). These books must be digitized by K employees (1 <= K <= 100). The first employee will process 0 or more of the books starting at s1; the second employee will process 0 or more of the next books in sequence and so on. Each employee is paid d_1 dollars per page for her first processed book, d_2 dollars per page for her second processed book, ..., d_i dollars per page (0 <= d_i <= 10,000,000) for her i-th processed book.

Write a program to determine the minimum possible cost to process all the books.

PROBLEM NAME: books

INPUT FORMAT:

* Line 1: Two space-separated integers, N and K

* Line 2: N space-separated integers, respectively s1, s2, ..., sN

* Line 3: N space-separated integers, respectively d1, d2, ..., dN

SAMPLE INPUT (file books.in):

```
6 3
50 100 60 5 6 30
1 2 3 4 5 6
```

OUTPUT FORMAT:

* Line 1: A single line on the standard output, the minimal cost the employer has to pay to process all the books.

SAMPLE OUTPUT (file books.out):

```
339
```

Analysis: Books by Richard Peng

The state is best[i,p], which is the minimum cost of employing p people to organize the first i books. And our transition is based on how many books the current employer is processing:

best[i,p]=min(best[j][p-1]+cost(j+1,i))

Where cost(j+1,i) is simply s_{j+1} *d_1 + s_{j+2} * d_2 + ... + s_i * $d_{\_i-j}$, which we can precompute in advance in $O(n^2)$ time for each j.

Since there are O(nk) states and our transition is O(n), the algorithm is $O(n^2k)$.

A sample code from mahbub follows; note that to make it possible in O(N) memory the loop is run from backward.

```
#include<stdio.h>

#define I64d "%lld"
#define LL long long

LL page[201],cost[201],ans[201],w[201][201];
int i,j,k,N,K;

int main()
{
        freopen("books.in","r",stdin);
        freopen("books.out","w",stdout);

        scanf("%d%d", &N,&K);

        for(i=1;i<=N;i++) scanf(I64d,&page[i]);
        for(i=1;i<=N;i++) scanf(I64d,&cost[i]);

        for(i=1;i<=N;i++)
        {
                w[i][i]=page[i]*cost[1];
                for(j=i+1,k=2;j<=N;j++,k++)
                        w[i][j]=w[i][j-1]+page[j]*cost[k];
        }

        for(i=0;i<=N;i++) ans[i]=-1;

        ans[0]=0;
        for(k=1;k<=K;k++)
        {
                for(i=N;i>=1;i--)
                        for(j=0;j<i;j++)
```

```
                                if(ans[j]!=-1)
                                {
                                        if(ans[i]==-1 ||
ans[j]+w[j+1][i]<ans[i])
                                                ans[i]=ans[j]+w[j+1][i];
                                }
        }

        printf(I64d "\n",ans[N]);

        return 0;
}
```

Neal Wu writes:

It is also possible to compute the costs to produce books inside the loop for the DP, as in the following solution:

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

FILE *fin = fopen ("books.in", "r");
FILE *fout = fopen ("books.out", "w");

const long long LLINF = (long long) 1e18;
const int MAXN = 205, MAXK = 105;

int N, K;
int pages [MAXN], costs [MAXN];
long long best [MAXN][MAXK];

int main ()
{
    fscanf (fin, "%d %d", &N, &K);

    for (int i = 0; i < N; i++)
        fscanf (fin, "%d", pages + i);

    for (int i = 0; i < N; i++)
        fscanf (fin, "%d", costs + i);

    memset (best, 63, sizeof (best));

    best [0][0] = 0;

    for (int i = 0; i < N; i++)
        for (int j = 0; j < K; j++)
            if (best [i][j] < LLINF)
            {
                long long sum = 0;

                for (int k = i, count = 0; k <= N; k++, count++)
                {
                    best [k][j + 1] <?= best [i][j] + sum;
                    sum += (long long) pages [k] * costs [count];
                }
            }
```

```
    fprintf (fout, "%lld\n", best [N][K]);

    return 0;
}
```

3. Cow Pie Treasures [Kolstad, 2006] (23, 27) OCT 06 QUAL

The cows have been busily baking pies that contain gold coins! Each pie has some number N_i (1 <= N_i <= 25) of gold coins and is neatly labeled on its crust with the number of gold coins inside.

The cows have placed the pies very precisely in an array of R rows and C columns (1 <= R <= C <= 100) out in the pasture. You have been placed in the pasture at the location (R=1,C=1) and have gathered the gold coins in that pie. You must make your way to the other side of the pasture, moving one column closer to the end point (which is location (R,C)) with each move you make. As you step to the new column, you may stay on the same row or change your row by no more than 1 (i.e., from (r,c) you can move to (r-1,c+1), (r, c+1), or (r+1,c+1). Of course, you would not want to leave the field and fail to get some gold coins, and you must end up at (R,C).

Given a map of the pasture, what is the greatest number of gold coins you can gather?

By way of example, consider this field of gold-bearing cow pies:

```
start-> 6 5 3 7 9 2 7
        2 4 3 5 6 8 6
        4 9 9 9 1 5 8 <-end
```

Here's one path:

```
start-> 6 5 3 7 9 2 7
         \
        2 4 3 5 6 8 6
           \   / \
        4 9 9-9 1 5-8 <-end
```

The path above yields 6+4+9+9+6+5+8 = 47 gold coins. The path below is even better and yields 50 coins, which is the best possible:

```
start-> 6 5 3 7 9 2 7
         \
        2 4 3 5 6-8 6
           \   /   \
        4 9 9-9 1 5 8 <-end
```

PROBLEM NAME: pie1

INPUT FORMAT:

* Line 1: Two space-separated integers: R and C

* Lines 2..R+1: Each line contains C space-separated small integers in
        the obvious order

SAMPLE INPUT (file pie1.in):

```
3 7
6 5 3 7 9 2 7
2 4 3 5 6 8 6
4 9 9 9 1 5 8
```

OUTPUT FORMAT:

* Line 1: An integer (which fits easily into 32 bits) that is the
        maximum number of gold coins that can be gathered

SAMPLE OUTPUT (file pie1.out):

```
50
```

USACO OCT06 2006 Problem 'pie1' Analysis

by Rob Kolstad

This is a variant of the 'numtri' problem from the USACO training pages. The trick is to avoid the use of recursion, which ends up in a combinatorial explosion since it recalculates various answers over and over again. The atractiveness is that the solution could be just like the solution to the 'skate' problem, with a depth-first search; below is a Pascal version of such a search from one of the competitors with added commands by Coach Rob:

```
procedure find (r , c, total:integer);
begin
    total := total + a[r,c];            { add in this square }
    if (r=r) and (c=c) then begin       { are we done? }
        if max<total then max:=total;   { tally maximum sum if better }
        exit;                           { exit = 'return' in C/C++/Java }
    end;
    if (r>1) and (c<c) then             { legal to search at r-1,c+1? }
        find(r-1,c+1,total);            { check next column right }
    if (c<c) then                       { legal to search at r,c+1? }
        find(r,c+1,total);              { check }
    if (r<r) and (c<c) then             { legal for r+1,c+1? }
        find(r+1,c+1,total);            { check }
end;
```

Unfortunately, the recursive calls end up repeatedly touching the same squares (and the number of recursions is just astonishingly large).

What to do?

One way is simply to work the problem backwards, touching each square only a very few times. The best result possible for column C-1 (the next-to-last one) is entirely dependent on the values in column C-1 and C. Inductively, this takes one all the back to column 1, where the answer pops out of the R=1,C=1 element.

This is a sort of 'Dynamic Programming', though perhaps is not as illuminating for the general case as one would hope. The complete JAVA solution below is from USA's Justin Hsu and implements the backwards-working algorithm (read on later for other interesting solutions). Comments are by Coach Rob:

```
import java.io.*;
import java.util.*;

class pie1 {
    public static void main (String [] args) throws IOException {

    BufferedReader in = new BufferedReader (new FileReader ("pie1.in"));
    PrintWriter out = new PrintWriter (new BufferedWriter new
FileWriter("pie1.out")));
    StringTokenizer st;

    int[][] max = new int[102][102];
    int[][] field = new int[102][102];

    st = new StringTokenizer(in.readLine());
```

```
    int R = Integer.parseInt(st.nextToken());   /* problem parameters */
    int C = Integer.parseInt(st.nextToken());

    /* Read the pie values: */
    for(int r = 0; r <= R+1; r++) {
        if(r != 0 && r <= R)
            st = new StringTokenizer(in.readLine());
        for(int c = 0; c <= C+1; c++) {
            max[r][c] = 0;
                        /* don't read into inappropriate squares: */
            if(r == 0 || c == 0) continue;
            if(r > R || c > C) continue;
            field[r][c] = Integer.parseInt(st.nextToken());
        }
    }

    max[R][C] = field[R][C];
    /* working backwards from column C-1: */
    for(int c = C-1; c >= 0; c--) {
        for(int r = 1; r <= R; r++) {   /* for each row */
                        /* don't tabulate inappropriate squares: */
            if(!((r <= c) && (c-r <= C-R))) continue;
            /* calculate best that square r,c can be: */
            max[r][c] = field[r][c] + Max3 (max[r-1][c+1], max[r][c+1],
max[r+1][c+1]);
        }
    }

    out.println(max[1][1]);
    in.close();
    out.close();
    System.exit(0);
  }

  public static int Max3(int a, int b, int c) {
         return Max(Max(a, b),c);
  }

  public static int Max(int a, int b) {
         if(a > b) return a;
         else return b;
  }
}
```

But clever folks can create a forward-looking dynamic programming solution that is amazing in its brevity and cleverness. Here is South Africa coach Bruce Merry's C++ solution that reverses the calculations of the program above so that the answer pops out of R,C instead of 1,1:

```
#include <fstream>
#include <algorithm>

using namespace std;

#define MAXR 105
#define MAXC 105

int main() {
```

```
    int coins[MAXR][MAXC] = {{0}};
    int dp[MAXR][MAXC] = {{0}};
    int R, C;
    ifstream in("pie1.in");

    in >> R >> C;
    /* read in all the coin values: */
    for (int i = 1; i <= R; i++)
        for (int j = 1; j <= C; j++)
            in >> coins[i][j];
    /* dynamic programming: */
    for (int j = 1; j <= C; j++)
        for (int i = 1; i <= j && i <= R; i++)
            dp[i][j] = coins[i][j] + (dp[i - 1][j - 1] >? dp[i][j - 1] >?
dp[i + 1][j - 1]);

    ofstream out("pie1.out");
    out << dp[R][C] << "\n";
    return 0;
}
```

4. Numbering the Cows [Romanian Olympiad, 2001] (31, 34) Camp 07 D4

Farmer John wants to assign a (not necessarily distinct) integer in the range 1..N (3 <= N <= 1,000) to each of his N cows for identification purposes. Thus, he lined up all his cows in order and came up with the following rule: the i-th cow in line must be assigned an integer that is less than the integers assigned to cows i+2 and i+3.

FJ wants to know the total number of possible assignments he might make. If cow i+2 does not exist, then cow i does not have the constraint that its ID number must be less than that of cow i+2. Likewise if cow i+3 does not exist.

Since Farmer John is not too good at math and might be confused by large numbers, output the answer modulo M (1 <= M <= 10,000,000).

PROBLEM NAME: cownum

INPUT FORMAT:

Two space-separated integers: N and M.

SAMPLE INPUT (file cownum.in):

3 10000

OUTPUT FORMAT:

A single line containing the number of assignments modulo M.

SAMPLE OUTPUT (file cownum.out):

9

OUTPUT DETAILS:

There are 9 valid assignments: (1,1,2), (1,1,3), (1,2,2), (1,2,3), (1,3,2), (1,3,3), (2,1,3), (2,2,3), (2,3,3).

Analysis: Numbering the Cows by Md. Mahbubul Hasan

The state of the DP is not that obvious.

While solving this problem by myself the first idea that came to mind is the obvious one, O(N^5) that is sliding window. We need 4 information in a state in this algorithm, [at which position are we (n)][number at n-1 th position][number at n-2 th state][number at n-3 th state].

If we think a bit we can reduce this to O(N^4). In that case the state will be:
[at which position are we (n)][minimum of number at n-1 th and n-2 th position][minimum of number at n-2 and n-3 th position]. (Why it is a valid state
information? - I am leaving this question as exercise for the readers)

But from here we cant see any obvious improvement. So we step back and return to drawing board. In worst case, N = 1000. So it is quite obvious that the expected solution should be O(N^2). How can we represent a state by only two variables? One obvious variable is the position number. And what should be
the other one? We must use the limit. Ok, lets start with this type of state:

[position][the highest limit of the ID].

And if we think a bit then we can realize that this will lead to O(N^2) solution. This is how it can be cracked down:

Say we will label the cows from rear. That means we are going to assign cow number from n'th cow.

[i][j] = number of ways we can label j cows by id 1 to i.

1st obvious choice is we can label j cows using id 1 to i-1. ([i-1][j])
2nd choice is we label last two cows by i. ([i-1][j-2])
3rd choice we can label last one i, and other j-1 less than i. ([i-1][j-1])
4th choice, we can label last one x (ranging from 1 to i-1) and second last one i. (sum x = 0 to i-2 cnt[x][j-2])

So our recurrence is:
[i][j] = [i-1][j] + [i-1][j-2] + [i-1][j-1] + sum (x = 0 to j-2) [i][j-2]

This leads to our desired O(N^2) algorithm. A sample code is given below which
implements the above idea:

(Note that in our recurrence we still have a linear loop. To get rid of it i used another array scnt[][] that keeps information about the sum. There is certainly a way not using two different array. We can just simply switch our dp state to sum state. That means in the following code we could have just used scnt[][] to solve the problem!)

```
#include<stdio.h>

int i,j,N,M,cnt[1001][1001],scnt[1001][1001];

int main()
{
        freopen("cownum.in","r",stdin);
        freopen("cownum.out","w",stdout);

        scanf("%d%d",&N,&M);

/*Base Case*/
        for(i=1;i<=N;i++) cnt[0][i]=scnt[0][i]=0;
        for(i=0;i<=N;i++) {cnt[i][0]=1; scnt[i][0]=i+1;}
        for(i=1;i<=N;i++) {cnt[i][1]=i; scnt[i][1]=scnt[i-1][1]+i;}
/*Base Case*/

        for(j=2;j<=N;j++)
                for(i=1;i<=N;i++)
                {
                        cnt[i][j]=(cnt[i-1][j]+cnt[i-1][j-2]+cnt[i-1][j-
1]+scnt[i-2][j-2])%M;
                        scnt[i][j]=(scnt[i-1][j]+cnt[i][j])%M;
                }

        printf("%d\n",cnt[N][N]);

        return 0;
}
```

It is also possible to solve this problem with O(N) memory, as in Neal Wu's solution:

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

FILE *fin = fopen ("cownum.in", "r");
FILE *fout = fopen ("cownum.out", "w");

const int MAXN = 1005;

int N, M;
int dp [MAXN], dp2 [MAXN];
int sum [MAXN], sum2 [MAXN], sum3 [MAXN];

inline int mod (int a)
{
    while (a >= M) a -= M;
    return a;
}
```

```
int main ()
{
    fscanf (fin, "%d %d", &N, &M);

    for (int i = 0; i < N; i++)
    {
        dp [i] = i % M;
        sum [i] = (i + 1) % M;
        sum2 [i] = (i * (i + 1) >> 1) % M;
    }

    for (int i = 2; i <= N; i++)
    {
        for (int j = 1; j <= N; j++)
        {
            dp2 [j] = mod (dp2 [j - 1] + dp [j - 1] + sum [j - 1]);
            sum3 [j] = mod (sum3 [j - 1] + dp2 [j]);
        }

        memcpy (dp, dp2, sizeof (dp));
        memcpy (sum, sum2, sizeof (sum));
        memcpy (sum2, sum3, sizeof (sum));
    }

    fprintf (fout, "%d\n", dp [N]);

    return 0;
}
```

5. Cow Highway [COCI Contest 6, 2008]

FJ's N (1 <= N <= 100,000) cows are each making trips on the highway. The highway has several stops, where each stop is the location of an entrance and an exit. Each cow has a particular route it needs to take; cow i wishes to enter the highway at A_i and exit at B_i (1 <= A_i, B_i <= 1,000,000,000). No two cows will have either the same entrances or the same exits.

Rising prices have prompted the government to charge drivers. In particular, upon entering the highway each cow is given a ticket stating the entrance used. When the cow leaves the highway at location Y with a ticket containing the number X, she must pay a fee of |X - Y|.

The cows have realized that by exchanging entrance cards, they may be able to reduce their total cost. However, they do not want a cow to have the same location for her entrance card and exit, since that would arouse suspicion. Thus, the cows would like you to help them determine the lowest possible total cost while keeping entrances and exits distinct.

PROBLEM NAME: highway

INPUT FORMAT:

* Line 1: The single integer N.

* Lines 2..N+1: Line i+1 contains the two space-separated integers A_i and B_i.

SAMPLE INPUT (file highway.in):

```
3
5 5
6 7
8 8
```

INPUT DETAILS:

There are 3 cows: the first cow is entering and exiting at location 5; the second cow is entering at location 6 and exiting at location 7; the third cow is entering and exiting at location 8.

OUTPUT FORMAT:

* Line 1: A single integer representing the minimum possible total cost for the cows.

SAMPLE OUTPUT (file highway.out):

```
5
```

```
OUTPUT DETAILS:

Swap entrance cards so that the first cow has the card with 6, the second
cow has the card with 8, and the third cow has the card with 5. The total
cost is then |6 - 5| + |8 - 7| + |5 - 8| = 1 + 1 + 3 = 5.
```

Analysis: Cow Highway by Official Solution by Igor Canadi

Let us first solve the problem without the constraint that a driver may not use his ticket at the same exit where the ticket was issued. Because the tickets may be exchanged arbitrarily, any driver can obtain any ticket. It is easy to see that the optimal solution is to sort the tickets, sort the drivers by their desired exits, and give the drivers tickets in order.

What if a driver gets a ticket issued at the same exit he needs to use? The best action would be for him to swap his ticket with one of his colleagues next to him in the sorted sequence, which is always possible. But what if, as in the second example test case, two drivers want to exchange tickets with the same driver? Then we need to allow them to exchange tickets not only with immediately adjacent drivers, but also those 2 indices away in the sorted sequence. From this analysis we obtain a dynamic programming solution.

Let dp[n] be the cost of the optimal distribution of tickets for all drivers up to driver n (the drivers are sorted by their exits). Let distribution(n, k) be the cheapest distribution of tickets in positions n, n−1, ..., n−k to drivers in positions n, n-1, ... n-k. There are (k+1)! possible distributions, but because k will be at most 2, we can check every one and select the distribution which results in the least total cost.

The DP relation is:

dp[n] = min{ dp[n−k−1] + distribution(n, k) for 0<=k<=2 }

The solution is dp[N].

Code for official solution:

```
#include <algorithm>
#include <cstdio>

#define MAX 100000

using namespace std;

typedef long long llint;

const llint inf = 1000000000000000LL;

llint calc( int a, int b ) {
   if( a > b ) return a - b;
   if( b > a ) return b - a;
   return inf;
}

int n;
int a[MAX], b[MAX];
```

```
int perm[3];
llint dp[MAX+1];

int main( void ) {
   scanf( "%d", &n );
   for( int i = 0; i < n; ++i ) scanf( "%d%d", &a[i], &b[i] );

   sort( a, a+n );
   sort( b, b+n );

   dp[n] = 0;

   for( int i = n-1; i >= 0; --i ) {
      dp[i] = inf;

      for( int k = 1; k <= 3 && i+k <= n; ++k ) {
         for( int j = 0; j < k; ++j ) perm[j] = j;

         do {
            llint cost = 0;

            for( int j = 0; j < k; ++j )
               cost += calc( a[i+j], b[i+perm[j]] );

            dp[i] = min( dp[i], cost + dp[i+k] );
         } while( next_permutation( perm, perm+k ) );
      }
   }

   printf( "%lld\n", dp[0] );

   return 0;
}
```

Knapsack DP

1. Cow Cash [Traditional, 2003] (13, 16) OCT 07 GOLD

The cows have not only created their own government but they have chosen to create their own money system. In their own rebellious way, they are curious about values of coinage. Traditionally, coins come in values like 1, 5, 10, 20 or 25, 50, and 100 units, sometimes with a 2 unit coin thrown in for good measure.

The cows want to know how many different ways it is possible to dispense a certain amount of money using various coin systems. For instance, using a system with values {1, 2, 5, 10, ...} it is possible to create 18 units several different ways, including: 18x1, 9x2, 8x2+2x1, 3x5+2+1, and many others.

Write a program to compute how many ways to construct a given amount of money N (1 <= N <= 10,000) using V (1 <= V <= 25) coins. It is guaranteed that the total will fit into both a signed 'long long' integer (C/C++), 'Int64' (Pascal), and 'long' integers in Java.

PROBLEM NAME: money

INPUT FORMAT:

* Line 1: Two space-separated integers: V and N

* Lines 2..V+1: Each line contains an integer that is an available coin value

SAMPLE INPUT (file money.in):

```
3 10
1
2
5
```

OUTPUT FORMAT:

* Line 1: A single line containing the total number of ways to construct N money units using V coins

SAMPLE OUTPUT (file money.out):

```
10
```

Analysis: Cow Cash by Valeriy Skobelev

For this problem, as we try to find the number of combinations of coins that sum up to a certain amount. We search through every possible combination of coins using a recursive routine that keeps track of previously solved subproblems (combinations of coins that sum up to a different, smaller amount) in a 2 dimensional array. One dimension is used to look up the amount of money the coins sums up to, and the second dimension is used to keep track of what subset of coins are used for the sub problem.

It is important to account for the fact that the order of the coins is irrelevant, this is accomplished by restricting the recursive calls of the search to using either the last used coin (to allow for multiple coins of the same type) or a subset coins that have not already been used.

```
#include <stdio.h>
int v, n;
int coins[25];
long long num_ways[25][10001];

/* search for how many combinations of coins which have
  an index >= lowest_coin, that sum up to amount */
long long search(int amount, int lowest_coin) {
        if (amount < 0) return 0;
        if (num_ways[lowest_coin][amount] != -1)
                return num_ways[lowest_coin][amount];
        long long total = 0;
        for (int i = lowest_coin; i < v; i++) {
                total += search(amount - coins[i], i);
        }
        num_ways[lowest_coin][amount] = total;
        return total;
}

int main() {
        freopen("money.in", "r", stdin);
        freopen("money.out", "w", stdout);

        scanf("%d%d", &v, &n);
        for (int i = 0; i < v; i++)
                scanf("%d", &coins[i]);
        for (int j = 0; j < v; j++) { /* initialize the subproblem lookup
array */
                for (int i = 1; i <= n; i++)
                        num_ways[j][i] = -1;
                num_ways[j][0] = 1;
        }
        printf("%lld\n", search(n, 0));
        return 0;
```

```
}
```

This is a DP problem that can be computed recursively the only thing you should take care of is integer overflows so you can use c++ long long or java long or BigInteger class. Given that the amount needed is V that uses K coins. You can compute the number of ways using all the first K-1 coins except the last one or you can compute it with all the K coins.

Compute(v,k)= the number of ways using all the first K-1 coins except the last one+ compute it with all the K coins

Compute(v,k)= compute(v,k-1) /*all but the last one*/ + compute(v-coins[k],k) /*using the last one then the remaining value is v-coins[k]*/

Here is wahab1 java solution using java BigInteger class

```
/*
ID: wahab1
PROG: money
LANG: JAVA
*/

import java.math.*;
import java.io.*;
import java.util.*;

public class money
{

static  int coins[]=new int[25];
static  BigInteger val[][]=new BigInteger[25][10001];
        public static BigInteger compute(int v,int k)
        {
                if(k==0)
                        return v%coins[k]==0?BigInteger.ONE:BigInteger.ZERO;
                if(v==0)
                        return BigInteger.ONE;
                if(v<0)
                        return BigInteger.ZERO;
                if(val[k][v]!=null)
                        return val[k][v];
                return val[k][v]=compute(v,k-1).add(compute(v-coins[k],k));
        }


        public static void main(String[] args) throws Exception
        {
                FileReader in=new FileReader("money.in");
                PrintWriter out=new PrintWriter(new FileWriter("money.out"));
                StreamTokenizer st=new StreamTokenizer(in);
                int n,v;
                st.nextToken();
                n=(int)st.nval;
                st.nextToken();
                v=(int)st.nval;
                for(int i=0;i<n;i++)
                {
                        st.nextToken();
                        coins[i]=(int)st.nval;
```

```
                }
                out.println(compute(v,n-1));
                out.close();

        }

}
```

Neal Wu writes:

The following is a simple and fast iterative solution that uses only O(N) memory:

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

FILE *fout = fopen ("money.out", "w");
FILE *fin = fopen ("money.in", "r");

const int MAXN = 10005;

int V, N, C;
long long nways [MAXN];

int main ()
{
    nways [0] = 1;

    fscanf (fin, "%d %d", &V, &N);

    while (V--)
    {
        fscanf (fin, "%d", &C);

        for (int i = 0; i + C <= N; i++)
            nways [i + C] += nways [i];
    }

    fprintf (fout, "%lld\n", nways [N]);

    return 0;
}
```

2. Hay Packing [Traditional, 2005] (20, 28)

Cows do love their hay. Their latest challenge is storing hay bales in a certain bin in the barn so they'll have hay throughout FJ's holiday. They know the bin will hold a maximum volume M (10 <= M <= 5,000) of hay. They have H (1 <= H <= 200) indivisible hay bales, each with volume V_i (1 <= V_i <= 500).

They're keen to know: what is the maximum total volume of hay they can stash into the bin? They can choose as many or as few bales as they wish.

PROBLEM NAME: packhay

INPUT FORMAT:

* Line 1: Two space-separated integers: M and H

* Lines 2..H+1: Line i contains a single integer: V_i

SAMPLE INPUT (file packhay.in):

```
21 5
2
4
6
8
10
```

OUTPUT FORMAT:

* Line 1: The maximum amount of hay that can actually be stored in the bin

SAMPLE OUTPUT (file packhay.out):

```
20
```

OUTPUT DETAILS:

All the hay bale sizes are even, so it's unlikely that 21 can be achieved

Analysis: Hay Packing by Richard Peng

This is a very typical knapsack problem with the state being the number of bales already considered and the list of possible sums. The DP values stored in the states are 0/1 which indicates whether the state is possible.

To see if a state of [bale,total] is possible, we check whether either one of the state of [bale-1,total] or [bale-1,total-size[bale]] as these represents the choice of whether to use the current bale. This runs in O(VH) time and uses O(VH) memory.

We can reduce the memory consumption to O(V) using rotating arrays, or simply updating the array backwards (in reducing totals).

Here is a solution from Mahbub:

```
#include<stdio.h>
int i,n,V,H,dp[6000],j;
int main()
{
        freopen("packhay.in","r",stdin);
        freopen("packhay.out","w",stdout);

        scanf("%d%d",&V,&H);
        for(dp[0]=1,j=1;j<=H;j++)
        {
                scanf("%d",&n);
                for(i=V-n;i>=0;i--) dp[i+n]|=dp[i];
        }
        for(i=V;i>=0;i--) if(dp[i]) break;
        printf("%d\n",i);
        return 0;
}
```

Neal Wu writes:

It is also possible to obtain a simple (and fast!) solution using bitsets:

```
#include <cstdio>
#include <bitset>
using namespace std;

FILE *fin = fopen ("packhay.in", "r");
FILE *fout = fopen ("packhay.out", "w");

const int MAX = 6000;

int V, H, S;
```

```
bitset <MAX> good;

int main ()
{
    good [0] = 1;

    fscanf (fin, "%d %d", &V, &H);

    while (H--)
    {
        fscanf (fin, "%d", &S);
        good |= good << S;
    }

    while (!good [V])
        V--;

    fprintf (fout, "%d\n", V);

    return 0;
}
```

3. Feed Factories [Tim Abbott, 2003] (26, 30) Camp 04

Farmer John has discovered that villainous feed salesmen have been ripping him off by charging him more than the market price of M cowbucks per pound of feed! In order to prevent this from ever being a problem again, he plans to purchase enough feed factories to be entirely self-sufficient and never have to purchase from a salesman. Farmer John currently uses K pounds of feed (1 <= K <= 100) per day on his farm.

He has surveyed a group of N (2 <= N <= 100) feed factories, determining for each the daily productivity P_i (1 <= P_i <= 100) and a daily maintenance cost C_i (1 <= C_i <= 100). FJ is a business-savvy guy, and he wants to make his business as efficient as possible. FJ measures the efficiency of a business by the formula (total profits) / (total productivity). Since he will sell all his feed at the market price of M (1 <= M <= 100) cowbucks/pound, the total profit is

```
M * sum(P_i) - sum(C_i)
```

Hence the efficiency is

```
M - (sum(C_i) / sum(P_i))
```

(we don't subtract the M*K cost of the feed FJ uses because he would have to pay for it anyway).

FJ only produces R (2 <= R <= 100,000) cowbucks per day in profit from his farm, and so he will not accept expenditures greater than that in case the cow feed industry fails.

Help him determine the maximum efficiency his new business can achieve.

PROBLEM NAME: cowfact

INPUT FORMAT:

* Line 1: Four space-separated integers: N, K, and M, and R

* Lines 2..N+1: Line i describes factor i with two space-separated integers: C_i and P_i

SAMPLE INPUT (file cowfact.in):

```
5 54 7 100
10 10
20 18
```

30 25
40 22
50 30

OUTPUT FORMAT:

* Line 1: The positive integer that is the truncated product of 1,000
  and the maximum efficiency of FJ's business with a minimum
  total productivity of K if that value is positive, or -1 if
  the maximum efficiency is negative.

SAMPLE OUTPUT (file cowfact.out):

5666

OUTPUT DETAILS:

FJ takes the first 4 factories, for a total productivity of 75 and
a total cost of 100, so the efficiency is 7 - 4/3 = 5.66666... Note
that we cannot buy the first three factories and the 5th, which
would give a slightly higher efficiency, due to FJ's limited value
of R. He cannot succeed using only the first three factories because
K is too big.

Analysis: Feed Factories by Richard Peng

The hard part of the problem is the two bounding conditions on minimum amount produced and maximum cost allowed. So there isn't much choice than to DP on those values.

Our state can be the machines already considered and the total cost and we try
to maximize amount produced at each cost since that's what's necessary to satisfy the amount produced condition and the higher the amount the higher the
productivity.

The total number of stats is N*(sum(p_i)) which can be at most 10^6, so this is best done iteratively with a queue tracking the reachable states. Again, we can just update on the same array each time if we process the transitions in decreasing order of cost.

Here is the sample solution:

```
#include<stdio.h>
#include<utility>
using namespace std;

pair<int,int> F[102];
int ans,N,K,M,R,maxL,i,j;
int dp[100000];

int main()
{
        freopen("cowfact.in","r",stdin);
        freopen("cowfact.out","w",stdout);

        scanf("%d%d%d%d",&N,&K,&M,&R);

        for(maxL=0,i=1;i<=N;i++)
        {
                scanf("%d%d",&F[i].first,&F[i].second);
                maxL+=F[i].second;
        }

        dp[0]=0; for(i=1;i<=maxL;i++) dp[i]=-1;

        for(i=1;i<=N;i++)
        {
                for(j=maxL-F[i].second;j>=0;j--)
```

```
                        if(dp[j]!=-1)
                        {
                                if(
                                   dp[j+F[i].second]==-1 ||
                                   dp[j+F[i].second]>dp[j]+F[i].first
                                  )
                                        dp[j+F[i].second] = dp[j]+F[i].first;
                        }
        }

        ans=-1;
        for(i=K;i<=maxL;i++)
        {
                if(dp[i]==-1 || dp[i]>R) continue;
                if(ans==-1) {ans=i; continue;}
                if(dp[ans]*i > dp[i]*ans) ans=i;
        }

        if(ans==-1) printf("-1\n");
        else printf("%d\n",(1000*(M*ans-dp[ans]))/ans);

        return 0;
}
```

4. Contest Digests [Ilham Kurnia, Russ Cox, 2002] (27, 34) Camp 05 D1

Farmer John runs his own Olympiad, the USACOW, to keep his cows on their hooves. FJ's Olympiad is a bit like the USACO contests with N tasks (1 <= N <= 10), each with T (1 <= T <= 10) test cases, with each test case worth a certain number of points P_i (1 <= P_i <= 400).

At the end of each contest, FJ posts the results on the barn wall, using a digest format like the USACO scoring lines, with '*'s marking correct test cases, '.'s marking incorrect ones, and a total number of points.

Bessie entered FJ's last contest and remembers N, the total number of points for each test case, and her total score. She does not, however, remember which cases she got correct and incorrect. Help her find a possible digest line (test cases marked correct and incorrect) for her score.

If there are multiple possible score lines, find the one that is lexicographically first when '*'s come before '.'s. Some solution will exist for each input set.

PROBLEM NAME: digest

INPUT FORMAT:

* Line 1: Two space-separated integers: N and S

* Lines 2..N+1: Many space-separated integers: T_i, ... Each line describes the test case points for a single task with a number of space-separated integers. The first integer on the line, T_i, is the number of test cases. The remaining T integers are the point values for the test cases for that problem.

SAMPLE INPUT (file digest.in):

```
2 850
3 100 200 200
2 250 300
```

OUTPUT FORMAT:

* Lines 1..N: Bessie's results in the usual contest digest form: each task should be on its own line, represented by a sequence of '*'s and '.'s. Line i reflects the results for task i.

SAMPLE OUTPUT (file digest.out):

```
**.
**
```

OUTPUT DETAILS:

Bessie can earn a score of 850 by getting every test case correct except for task 1, test case 3. (Bessie could also get every test case correct except for task 1, test case 2, but that score line -- *.* ** -- comes lexicographically later.)

Analysis: Contest Digests by Richard Peng

We basically want to find the lexicographical minimum subset (when considered as a binary number) that has a certain sum.

This is again a knapsack problem with the state being the test case currently being considered and the current sum. Since the sum can be as large as 10*10*400=40,000, we need to maintain 100*40,000 states, which is easily doable.

Doing this iteratively is quite messy as we need to compute the trace. So memorization is an easier way out as we can search in lexicographical order and
output the first answer we encounter. Generally memorization is a bit slower but in this case, a lot of the states aren't reached will be processed in the iterative case and memorization lets us quit earlier, so it usually runs faster.

Here is a sample solution by Mahbub:

```
#include<stdio.h>

int C,N,S,done;
int point[11][11];
int P[11];
char memo[11][11][40000];
int u[11][11];

void DP(int at,int pos,int taken)
{
        int i,j;

        if(taken==S)
        {
                for(i=1;i<=N;i++)
                {
                        for(j=0;j<P[i];j++)
                        {
                                if(u[i][j]) printf("*"); else printf(".");
                        }
                        printf("\n");
                }
```

```
                done=1;
                return;
        }

        if(at>N) return;
        if(pos==P[at]) {DP(at+1,0,taken); return;}
        if(memo[at][pos][taken]) return;
        memo[at][pos][taken]=1;

        if(taken+point[at][pos]<=S)
        {
                u[at][pos]=1;
                DP(at,pos+1,taken+point[at][pos]);
                if(done) return;
                u[at][pos]=0;
        }

        DP(at,pos+1,taken);
}

int main()
{
        int i,j;

        freopen("digest.in","r",stdin);
        freopen("digest.out","w",stdout);

        scanf("%d%d",&N,&S);
        for(i=1;i<=N;i++)
        {
                scanf("%d",&P[i]);
                for(j=0;j<P[i];j++) scanf("%d",&point[i][j]);
        }

        DP(1,0,0);

        return 0;
}
```

5. Space Elevator [Coaches , 2004] (30, 38) MAR 05 GOLD

The cows are going to space! They plan to achieve orbit by building a sort of space elevator: a giant tower of blocks. They have K (1 <= K <= 400) different types of blocks with which to build the tower. Each block of type i has height h_i (1 <= h_i <= 100) and is available in quantity c_i (1 <= c_i <= 10). Due to possible damage caused by cosmic rays, no part of a block of type i can exceed a maximum altitude a_i (1 <= a_i <= 40000).

Help the cows build the tallest space elevator possible by stacking blocks on top of each other according to the rules.

PROBLEM NAME: elevator

INPUT FORMAT:

* Line 1: A single integer: K

* Lines 2..K+1: Line i+1 describes block type i with three space-separated integers: h_i, a_i, and c_i

SAMPLE INPUT (file elevator.in):

```
3
7 40 3
5 23 8
2 52 6
```

OUTPUT FORMAT:

* Line 1: A single integer H, the maximum height of a tower that can be built

SAMPLE OUTPUT (file elevator.out):

```
48
```

OUTPUT DETAILS:

From the bottom: 3 blocks of type 2, below 3 of type 1, below 6 of type 3. Stacking 4 blocks of type 2 and 3 of type 1 is not legal, since the top of the last type 1 block would exceed height 40.

Analysis: Space Elevator by Md. Mahbubul Hasan and Neal Wu

It is a traditional DP problem.

Here is a sample solution:

```
#include<stdio.h>
#include<algorithm>
using namespace std;

#define max(A,B) ((A) > (B) ? (A) : (B))

struct BLOCK
{
        int h,a,c;
}block[500];

bool operator<(BLOCK A,BLOCK B)
{
        if(A.a < B.a) return 1;
        return 0;
}

int dp[40001];

int main()
{
        freopen("elevator.in","r",stdin);
        freopen("elevator.out","w",stdout);

        int n,sofar,i,j,k;

        scanf("%d",&n);

        for(i=0;i<n;i++)
                scanf("%d%d%d",&block[i].h,&block[i].a,&block[i].c);

        sort(block,block+n);

        dp[0]=1;
        sofar=0;

        for(i=0;i<n;i++)
        {
                for(j=sofar;j>=0;j--) if(dp[j])
                {
```

```
                        for(k=1;k<=block[i].c;k++)
                        {
                                if(j+block[i].h*k>block[i].a) break;
                                dp[j+block[i].h*k]=1;
                                sofar=max(sofar,j+block[i].h*k);
                        }
                }
        }

        printf("%d\n",sofar);

        return 0;
}
```

Neal Wu writes:

First, to deal with the condition of the limits on altitudes for blocks, we note that it is always best to place blocks in increasing order of their maximum altitude. The reason for this is because if we had two blocks in the tower such that the block nearer the top had a lower maximum altitude than the block nearer the bottom, we could swap them without worsening our tower.

Thus, we first sort each block by its maximum altitude, and then go through the blocks in order, performing a typical knapsack algorithm. However, in order to obtain a dramatic speed increase, one can use bitsets, as in the following solution:

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

FILE *fin = fopen ("elevator.in", "r");
FILE *fout = fopen ("elevator.out", "w");

const int MAXK = 405;
const int MAX = 40005;

struct block
{
    int H, A, C;

    inline bool operator < (const block &right) const
    {
        return A < right.A;
    }
};

int K;
block tower [MAXK];
bitset <MAX> good, temp;

int main ()
{
    good [0] = 1;

    fscanf (fin, "%d", &K);

    for (int i = 0; i < K; i++)
```

```
    {
        fscanf (fin, "%d %d %d", &tower [i].H, &tower [i].A, &tower [i].C);
        tower [i].A++;
    }

    sort (tower, tower + K);

    for (int i = 0; i < K; i++)
    {
        temp = (temp.set () << (MAX - tower [i].A) >> (MAX - tower [i].A)) |
good;

        while (tower [i].C--)
            good |= good << tower [i].H;

        good &= temp;
    }

    int ans = MAX - 1;

    while (!good [ans])
        ans--;

    fprintf (fout, "%d\n", ans);

    return 0;
}
```

6. Cow Exhibition [Brian Jacokes, 2003] (30, 37)

"Fat and docile, big and dumb, they look so stupid, they aren't much fun..." - Cows with Guns by Dana Lyons

The cows want to prove to the public that they are both smart and fun. In order to do this, Bessie has organized an exhibition that will be put on by the cows. She has given each of the N (1 <= N <= 100) cows a thorough interview and determined two values for each cow: the cow's smartness S_i (-1000 <= S_i <= 1000) and the cow's funness F_i (-1000 <= F_i <= 1000).

Bessie must choose which cows she wants to bring to her exhibition. She believes that the total smartness TS of the group is the sum of the Si's and, likewise, the total funness TF of the group is the sum of the Fi's. Bessie wants to maximize the sum of TS and TF, but she also wants both of these values to be non-negative (since she must also show that the cows are well-rounded; a negative TS or negative TF would ruin this). Help Bessie maximize the sum of TS and TF without letting either of these values become negative.

PROBLEM NAME: smrtfun

INPUT FORMAT:

* Line 1: A single integer: N

* Lines 2..N+1: Line i+1 contains two space-separated integers: S_i and F_i

SAMPLE INPUT (file smrtfun.in):

```
5
-5 7
8 -6
6 -3
2 1
-8 -5
```

OUTPUT FORMAT:

* Line 1: One integer: the optimal sum of TS and TF such that both TS and TF are non-negative. If no subset of the cows has non-negative TS and non- negative TF, print 0.

SAMPLE OUTPUT (file smrtfun.out):

```
8

OUTPUT DETAILS:

Bessie chooses cows 1, 3, and 4, giving values of TS = -5+6+2 = 3 and TF
= 7-3+1 = 5, so 3+5 = 8.  Note that adding cow 2 would improve the value
of TS+TF to 10, but the new value of TF would be negative, so it is not
allowed.
```

Analysis: Cow Exhibition by Richard Peng

The problem asks us to optimize over two values, so we could DP on the first value while trying to maximize the second. So our DP state is [cow to be considered, sum of the smartness of all the cows so far] and we try to maximize the sum of fun values of the cows.

Also, we could restrict the fun values to be positive by considering the cows in decreasing order of funness so once a negative fun value is reached, it's impossible to get positive.

The transition is essentially the same as the version where 0/1s are stored in the states, except we need to keep two arrays and rotate on them instead of working on a single one as fun values can be negative (we also do case work based on the sign of the fun value in order to keep things in a single array, but that's also some added work).

Here is a sample solution:

```
#include <iostream>
#include <algorithm>
using namespace std;

struct cowtype{
        int v1,v2;
        bool operator < (const cowtype &o) const {return v1>o.v1;}
};

cowtype cow[100];
int n,lis[200000],lt,lt1,bes[2][200000],pre,cur,ans,huge;

int main(){
        int i,j,tem1;

freopen("smrtfun.in","r",stdin);
freopen("smrtfun.out","w",stdout);

        huge=-2000000000;
        cin>>n;

        for(i=0;i<n;i++)
                cin>>cow[i].v1>>cow[i].v2;

        sort(cow,cow+n);

        for(i=0;i<200000;i++)   bes[0][i]=bes[1][i]=huge;
```

```
        bes[0][0]=0;
        lis[0]=0;
        lt=1;

        for(i=0;i<n;i++){
                pre=i%2;
                cur=(i+1)%2;

                for(j=0;j<lt;j++) bes[cur][lis[j]]=bes[pre][lis[j]];

                for(lt1=lt,j=0;j<lt1;j++)
                        if((tem1=lis[j]+cow[i].v1)>=0){
                                if(bes[cur][tem1]==huge)
                                        lis[lt++]=tem1;
                                bes[cur][tem1]>?=bes[pre][lis[j]]+cow[i].v2;
                        }
        }

        for(ans=huge,i=0;i<lt;i++)
                if(bes[cur][lis[i]]>=0)
                        ans>?=bes[cur][lis[i]]+lis[i];

        cout<<ans<<endl;

        return 0;
}
```

7. Elite Eating [Bulgarian Winter 2002 via Nikolay Valtchanov, 2003] (31, 32) Dec 03 Green

Farmer John's 100 cows are conveniently branded with the integers 1..100. FJ has created an elite milking program where exactly N (1 <= N <= 25) cows with certain brands get to enter the barn first. The restriction on this elite group of cows is that the sum of the squares of the brands of each member of the group must be strictly less than a given integer S (1 <= S <= 10,000).

Determine the number of different groups of cows that can be selected for the elite milking program.

PROBLEM NAME: elite

INPUT FORMAT:

* Line 1: Two space-separated integers: N and S

SAMPLE INPUT (file elite.in):

```
3 30
```

OUTPUT FORMAT:

* Line 1: A single integer that is the number of different possible groups that can line up for elite eating.

SAMPLE OUTPUT (file elite.out):

```
4
```

OUTPUT DETAILS:

The sequences of length 3 with sum of squares < 30 are:
1 2 3
1 2 4
2 3 4
1 3 4

The sequence of brands 1 2 5 is not valid since 1 + 4 + 25 = 30 and is not strictly less than 30.

Analysis: Elite Eating by Richard Peng

The problem is asking for the number of ways of picking k out of the n 'knapsacks' so that the total is below a certain number.

So this can be done using typical knapsack with the state being the number of cows used, the 'weight' of the cow being considered and the sum.

Here is a solution:

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

#define MAXN 50

int pre,cur,lis[MAXN][11000],lt[MAXN],n,s;
long long cou[MAXN][11000],ans;

int main(){
        int i,j,j1,k,v,v1,tem;
freopen("elite.in","r",stdin);
freopen("elite.out","w",stdout);

        cin>>n>>s;
        memset(lt,0,sizeof(lt));
        lt[0]=1;
        lis[0][0]=0;
        cou[0][0]=1;

        for(i=1;(v=i*i)<s;i++)
                for(j=n-1,j1=n;j>=0;j--,j1--)
                        for(k=0;k<lt[j];k++)
                                if((v1=lis[j][k]+v)<s){
                                        if(cou[j1][v1]==0)
                                             lis[j1][lt[j1]++]=v1;

                                        cou[j1][v1]+=cou[j][lis[j][k]];
                                }
        for(i=ans=0;i<s;i++)    ans+=cou[n][i];
        cout<<ans<<endl;
        return 0;
```

}

8. Different dice [Hal Burch, 1998] (34, 31)

In a different galaxy far far away there are different kinds of dice, often with unusual numbers of sides -- anywhere from 2 to 16 sides. Each side can have from 1 to 32 spots. Sets of dice contain anywhere from 1 through 32 dice. These dice roll just like Earth-based dice, of course, and on any roll you might get any one of the numbers on each die.

Anthropologists digging through old ruins in the galaxy far far away have found sets of dice that have various different numbers of spots. Here's one set of two two-sided dice they found:

* {2,3} and {3,4}

and here's another:

* {1,2} and {4,5}

It's easy to see that the first set of dice yields sums of 5, 6, and 7. The second set also yields 5, 6, and 7. Furthermore, the probability of rolling a five is 1/4, rolling a six is 1/2, and rolling a a seven is 1/4 for both sets of dice.

Given two sets of dice, your program must decide if they yield the same sets of sums. Additionally, it must also decide if the two sets yield those sums with the same probabilities.

PROBLEM NAME: dice

INPUT FORMAT:

* Line 1: Two integers separated by a space, D1 and S1, representing the number of dice in the first set and the number of sides on each die.

* Lines 2..D1+1: S1 values that specify the spot values for a die. The values might or might not all be different.

* Line D1+2: Two integers separated by a space, D2 and S2. D2 is the number of dice in the second set, and S2 is the number of sides of each die.

* Lines D1+3..D1+D2+2: S2 values that specify the spot values for a die. The values might or might not all be different.

SAMPLE INPUT (file dice.in):

```
2 6
1 2 3 4 5 6
1 2 4 3 4 1
3 4
1 4 3 2
5 3 2 1
1 1 1 1
```

OUTPUT FORMAT:

* Line 1: Two characters separated by a space. First character is `Y' if both sets of dice yield the same set of sums or `N' otherwise. The second character is `Y' if both sets of dice yield the same set of possible sums with the same set of probabilities of achieving those sums or `N' otherwise.

* Line 2: Two space-separated integers. The first integer indicates the number of possible sums the first set of dice could yield. The second integer indicates the number of different sums the second set of dice could yield.

SAMPLE OUTPUT (file dice.out):

```
N N
9 8
```

Analysis: Different dice by Richard Peng

It suffices find the number of ways of constructing k using each set of dice, which can be done by considering each dice in turn and track all the number of
ways of making i in count[i]. Then the state transition using a dice with k faces with a_1...a_k spots would be

count[i]=sum(j=1..i,c[i-a_j]), where each transition represents the case of the
jth face being chosen.

We need to process the is in reverse order to prevent updating using already updated values.

The numbers are fairly large, so the safest way is to use bignums, however doubles also seem to work quite well.

Here is the code:

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

double cou[2][10000],tot[2],tem;
int n,m,i,j,k,l,lis[40],can1,can2,t[2];

int main(){
freopen("dice.in","r",stdin);
freopen("dice.out","w",stdout);
        for(i=0;i<2;i++){
                tot[i]=1;
                for(j=1200;j>=0;j--)    cou[i][j]=0;
                cou[i][0]=1;
                cin>>n>>m;
                for(j=0;j<n;j++){
                        tem=m;
                        tot[i]*=tem;
                        for(k=0;k<m;k++)
                                cin>>lis[k];
                        for(k=1200;k>=0;k--)
                                for(cou[i][k]=0,l=0;l<m;l++)
```

```
                                        if(k>=lis[l])
                                                cou[i][k]+=cou[i][k-lis[l]];
                }
                for(t[i]=0,j=0;j<10000;j++){
                        t[i]+=((cou[i][j])!=0);
                }
        }
        for(can1=can2=1,i=0;(i<10000);i++){
                if(((cou[0][i]>0)&&(cou[1][i]==0))||
((cou[0][i]==0)&&(cou[1][i]>0)))
                        can1=0;
                if((cou[0][i]/tot[0])!=(cou[1][i]/tot[1]))
                        can2=0;
        }
        printf("%c %c\n%d %d\n",((can1)?'Y':'N'),((can2)?'Y':'N'),t[0],t[1]);
        return 0;
}
```

9. Jersey Politics [Coaches, 2004] (39, 39) Feb 05 GOLD

[
Memory Limit: 18MB, due to a change in the grading system since this problem was used.
]

In the newest census of Jersey Cows and Holstein Cows, Wisconsin cows have earned three stalls in the Barn of Representatives. The Jersey Cows currently control the state's redistricting committee. They want to partition the state into three equally sized voting districts such that the Jersey Cows are guaranteed to win elections in at least two of the districts.

Wisconsin has 3*K (1 <= K <= 60) cities of 1,000 cows, conveniently numbered 1..3*K, each with a known number (range: 0..1,000) of Jersey Cows. Find a way to partition the state into three districts, each with K cities, such that the Jersey Cows have the majority percentage in at least two of districts.

All supplied input datasets are solvable.

PROBLEM NAME: jpol

INPUT FORMAT:

* Line 1: A single integer: K

* Lines 2..3*K+1: Line i+1 contains a single integer that is city i's cow census: the number of cows in each city that are Jersey Cows.

SAMPLE INPUT (file jpol.in):

```
2
510
500
500
670
400
310
```

OUTPUT FORMAT:

* Lines 1..K: K lines that are the city numbers in district one, one per line

* Lines K+1..2K: K lines that are the city numbers in district two, one per line

* Lines 2K+1..3K: K lines that are the city numbers in district three, one per line

SAMPLE OUTPUT (file jpol.out):

```
1
2
3
6
5
4
```

OUTPUT DETAILS:

Other solutions might be possible. Note that "2 3" would NOT be a district won by the Jerseys, as they would be exactly half of the cows.

Analysis: Jersey Politics by Richard Peng, Md. Mahbubul Hasan, Spencer Liang

Spencer:

We start with the following recursive relation, where f(n, k, s) represents whether or not it is possible to form a district of cardinality k of sum s using the first n cities and v[n] is the number of Jersey Cows in city n.

```
f(n, k, s) = f(n-1, k, s) || f(n-1, k-1, s-v[n])
```

If X is the size of the largest city, then a simple implementation of the above equation runs in $O(K^3X)$ time and memory, taking both too much memory and too long.

We can reduce the space needed by a factor of K because only the values of f(n, *, *) are needed to calculate the values of f(n+1, *, *) and to cut down the running time, we use the following heuristics:
1) For a given k, we precompute min[k] and max[k] as the size of the smallest/largest possible district of k cities. Then we only need to compute the values of f(n, k, s) for those s from max(v[n], min[k]) <= s <= max[k] instead of from v[n] <= s <= 1000*K.
2) For a given n, we have max(1, n-K) as a lower bound for k because out of the n cities we have considered so far, the n-k cities not taken must be placed in district 2, but the maximum size for any district is K. So n-K <= k.

```
#include <stdio.h>
#include <iostream>
#define MAX 65
using namespace std;
#define FOR(i,a,b) for(int i=a;i<=b;i++)
#define FORD(i,a,b) for(int i=a;i>=b;i--)
typedef pair<int, int> PII;

int K; PII city[3*MAX];
int mini[MAX], maxi[MAX];
short dp[MAX][MAX*1000];
int used[3*MAX];

int main() {
    FILE *fin = fopen("jpol.in", "r"), *fout = fopen("jpol.out", "w");
```

```
    fscanf(fin, "%d", &K);
    FOR(i, 1, 3*K) {
        int v; fscanf(fin, "%d", &v);
        city[i] = PII(v, i);
    }
    sort(city+1, city+1+3*K, greater());

    FOR(i, 1, K) {
        maxi[i] = maxi[i-1]+city[i].first;
        mini[i] = mini[i-1]+city[2*K+1-i].first;
    }

    dp[0][0] = -1;
    FOR(i, 1, 2*K) {
        //i-K is a lower bound: the other district can have at most K cities
        int a = max(1, i-K), b = min(i, K);
        FORD(n, b, a) {
            //bounds on the smallest/largest possible districts of n cities
            int l = max(city[i].first, mini[n]), u = maxi[n];
            FOR(k, l, u)
                //avoid using the same city twice
                if (dp[n-1][k-city[i].first] && !dp[n][k])
                    dp[n][k] = i;
        }
    }

    FOR(c, 2*K+1, 3*K) {
        fprintf(fout, "%d\n", city[c].second);
        used[city[c].second] = true;
    }
    int goal = 500*K+1; while(!dp[K][goal]) goal++;
    FORD(k, K, 1) {
        int c = dp[k][goal];
        fprintf(fout, "%d\n", city[c].second);
        used[city[c].second] = true;
        goal -= city[c].first;
    }
    FOR(c, 1, 3*K) if (!used[c]) fprintf(fout, "%d\n", c);
    return 0;
}
```

Md. Mahbubul Hasan:
The problem asks us to divide the set of numbers into 3 groups such that the sum of at least 2 groups are more than 500*k.

After sorting the data we can easily discard the lower one third. So now if we
run DP over the rest set of data then the order would be approximately: 120*60*60*1000 where the state is: [how many city considered for the first group][What is the total vote of the selected city]. But the order is not that
much feasible. Adding some huristics the DP solution runs well.

Here is the code:

```
#include<stdio.h>
#include<algorithm>
```

```
#include<utility>
#include<functional>
using namespace std;

typedef pair<int,int> PII;

#define max(A,B) ((A) < (B) ? (B) : (A))
#define min(A,B) ((A) > (B) ? (B) : (A))

PII C[200];
int sum[200000];
int dp[62][60000];
int p[200];

int main()
{
        freopen("jpol.in","r",stdin);
        freopen("jpol.out","w",stdout);

        int n,i,j,k;
        int mincost,maxcost,maxL,xx;
        int need,covered,more;

        scanf("%d",&n);

        n*=3;
        for(i=0;i<n;i++)
        {
                scanf("%d",&C[i].first);
                C[i].second=i+1;
        }

        sort(C,C+n,greater<PII>());

        for(i=2*n/3;i<n;i++) {printf("%d\n",C[i].second); p[C[i].second]=2;}

        n=n/3*2;

        for(i=n-1;i>=0;i--) sum[i]=sum[i+1]+C[i].first;

        for(i=0;i<n;i++)
                if(!dp[1][C[i].first])
                        dp[1][C[i].first]=i+1;

        xx=n/2;
        need=500*xx+1;
        covered=1000*xx;

        for(i=2;i<=xx;i++)
        {
                more=xx-i+1;
                maxL=n-more;
                mincost=max(need-(sum[0]-sum[more]),0);
                maxcost=min(sum[0]-sum[i],covered-sum[maxL]);

                for(j=mincost;j<=maxcost;j++)
                {
```

```
                        if(!dp[i-1][j]) continue;

                        for(k=dp[i-1][j];k<=maxL;k++)
                        {
                                if(j + sum[k]-sum[k+more] < need) break;

                                if(dp[i][j+C[k].first]==0 ||
dp[i][j+C[k].first]>k+1)
                                        dp[i][j+C[k].first]=k+1;
                        }
                }
        }

        for(i=need;;i++) if(dp[xx][i]) break;

        while(i!=0)
        {
                p[C[dp[xx][i]-1].second]=1;
                i-=C[dp[xx--][i]-1].first;
        }

        for(i=1;i<=n/2*3;i++) if(p[i]==1) printf("%d\n",i);
        for(i=1;i<=n/2*3;i++) if(!p[i]) printf("%d\n",i);

        return 0;
}
```

Richard Peng writes:

Note that our state space is going to be fairly sparse when we haven't considered very many cows. So memorization is the way to go and we can track of
the states we've visited using hashing.

Also, we can add heuristics to reduce the state size even further. It's possible to reduce the total number of states to be considered to around 60,000, which easily run in time.

Another 'cheating' way to do this is by randomization. Note that the ideal case
is when we can divide all the cities so the number in the districts are exactly
the same. So we can define the 'chaos' level as the sum of squares of differences of the districts. We randomly organize the cities into groups, pick
a random pair of cities and swap their districts with high probability if it reduces the overall 'chaos' and low probability otherwise. This works surprisingly well in most cases and is a valid strategy of 'hacking' knapsack problems. Feel free to try to making data that breaks it.

Here is the solution using hashing:

```
#include <stdio.h>
#include <stdlib.h>
```

```
int k,n,val[180][2],k1,statet;

int exectime();

int comp(const void *pa, const void *pb) {
        int *a=(int*)pa; int *b=(int*)pb;
        if (a[0]==b[0]) return(0);
        if (a[0]>b[0]) return(-1);
        else return(1);
}

void datain(){
        FILE *f_in;
        int i;
        f_in=fopen("jpol.in","r");
        fscanf(f_in,"%d",&k);
        n=k*3;
        for (i=0;i<n;i++){
                fscanf(f_in,"%d",&val[i][0]);
                val[i][1]=i;
        }
        fclose(f_in);
        qsort(val,n,sizeof(int[2]),comp);
}

int
listt,hashsize,sum[120],max[120][61],sumback[121],low,high,rec[120],ans[120],
found;
short list[2600000][3];

void search(int pos,int used,int s){
        int p,i;
        if (found)      return;
        if (!((used<=k)&&(pos-used<=k)&&(s+max[pos][k-
used]>=low)&&(sum[pos-1]-s+max[pos][k-pos+used]>=low))) return;

        p=((pos*1288927+used*pos*212327+s*3301)/100)%hashsize;
        while (list[p][0]!=-1){
                if
((list[p][0]==pos)&&(list[p][1]==used)&&(list[p][2]==s))        return;
                p++;
                if (p==hashsize)        p=0;
        }
        list[p][0]=pos; list[p][1]=used;                list[p][2]=s;

        statet++;
if (statet%100000==0)   printf("%d\n",statet);
        if (pos==k1){
                if  ((used==k)&&(s>=low)&&(s<=high)){
                        found=1;
                        for (i=0;i<k1;i++)      ans[i]=rec[i];
                }
                return;
        }
        rec[pos]=0;
        search(pos+1,used,s);
        rec[pos]=1;
```

```
        search(pos+1,used+1,s+val[pos][0]);
}

void process(){
        int i,j;
        listt=0;        hashsize=2000003;
        k1=k*2;

//precomputation
        high=0;
        for (i=0;i<k1;i++)      high+=val[i][0];
        low=500*k+1;
        high=high-low;
        sumback[0]=0;
        for (i=1;i<=k1;i++)     sumback[i]+=sumback[i-1]+val[k1-i][0];
        for (i=0;i<k1;i++){
                for (j=0;j<=k;j++)      max[i][j]=0;
                for (j=1;((j<=k)&&(i+j-1<k1));j++)
max[i][j]=max[i][j-1]+val[i+j-1][0];
                for (j=1;j<=k;j++)      if (max[i][j]==0)
max[i][j]=max[i][j-1];
        }
        sum[0]=val[0][0];
        for (i=1;i<k1;i++)      sum[i]=sum[i-1]+val[i][0];
        for (i=0;i<hashsize;i++)        list[i][0]=-1;
        found=0;        statet=0;
        search(0,0,0);
}

void output(){
        FILE *f_out;
        int p,i,use[180],pre,a;
        for (i=0;i<k1;i++){
                if (ans[i])     use[i]=1;
                else use[i]=2;
        }
        for (i=k1;i<n;i++)      use[i]=3;
        f_out=fopen("jpol.out","w");
        for (i=0;i<n;i++)       if (use[i]==1)
fprintf(f_out,"%d\n",val[i][1]+1);
        for (i=0;i<n;i++)       if (use[i]==2)
fprintf(f_out,"%d\n",val[i][1]+1);
        for (i=0;i<n;i++)       if (use[i]==3)
fprintf(f_out,"%d\n",val[i][1]+1);
        fclose(f_out);
printf("%d\n",found);
        for (i=0;i<n;i++)       if (use[i]==1)  printf("%d\n",val[i][1]+1);
        for (i=0;i<n;i++)       if (use[i]==2)  printf("%d\n",val[i][1]+1);
        for (i=0;i<n;i++)       if (use[i]==3)  printf("%d\n",val[i][1]+1);
}

main(){
        datain();
        process();
        output();
        return(0);
}
```

1D state DP

1. The County Fair [Brian Dean, 2006] (20, 32) OPEN 06 Gold

Every year, Farmer John loves to attend the county fair. The fair has N booths (1 <= N <= 400), and each booth i is planning to give away a fabulous prize at a particular time P(i) (0 <= P(i) <= 1,000,000,000) during the day. He has heard about this and would like to collect as many fabulous prizes as possible to share with the cows. He would like to show up at a maximum possible number of booths at the exact times the prizes are going to be awarded.

FJ investigated and has determined the time T(i,j) (always in range 1..1,000,000) that it takes him to walk from booth i to booth j. The county fair's unusual layout means that perhaps FJ could travel from booth i to booth j by a faster route if he were to visit intermediate booths along the way. Being a poor map reader, Farmer John never considers taking such routes -- he will only walk from booth i to booth j in the event that he can actually collect a fabulous prize at booth j, and he never visits intermediate booths along the way. Furthermore, T(i,j) might not have the same value as T(j,i) owing to FJ's slow walking up hills.

Farmer John starts at booth #1 at time 0. Help him collect as many fabulous prizes as possible.

PROBLEM NAME: cfair

INPUT FORMAT:

* Line 1: A single integer: N.

* Lines 2..1+N: Line i+1 contains a single integer: P(i).

* Lines 2+N..1+N+N^2: These N^2 lines each contain a single integer T(i,j) for each pair (i,j) of booths. The first N of these lines respectively contain T(1,1), T(1,2), ..., T(1,N). The next N lines contain T(2,1), T(2,2), ..., T(2,N), and so on. Each T(i,j) value is in the range 1...1,000,000 except for the

diagonals T(1,1), T(2,2), ..., T(N,N), which have the value zero.

SAMPLE INPUT (file cfair.in):

```
4
13
9
19
3
0
10
20
3
4
0
11
2
1
15
0
12
5
5
13
0
```

INPUT DETAILS:

There are 4 booths. Booth #1 is giving away a prize at time 13, booth #2 at time 9, booth #3 at time 19, and booth #4 at time 3.

OUTPUT FORMAT:

* Line 1: A single integer, containing the maximum number of prizes Farmer John can acquire.

SAMPLE OUTPUT (file cfair.out):

```
3
```

OUTPUT DETAILS:

Farmer John first walks to booth #4 and arrives at time 3, just in time to receive the fabulous prize there. He them walks to booth #2 (always walking directly, never using intermediate booths!) and arrives at time 8, so after waiting 1 unit of time he receives the fabulous prize there. Finally, he walks back to booth #1, arrives at time 13, and collects his third fabulous prize.

USACO April 2006 Problem 'cfair' Analysis

by Bruce Merry

This is a straight-forward dynamic programming problem, that should be bread and butter to a gold competitor. Sort the events by time, and for each booth, compute the greatest number of prizes that can be achieved by being at booth at the time the prize is awarded.

Below is a very concise solution from Poland's JuliuszSompolski.

```
#include <cstdio>
#include <cstdlib>
using namespace std;

FILE *input = fopen("cfair.in","r");
FILE *output= fopen("cfair.out","w");

int T[400][400], P[400], n, best[400];

int main() {
    int i, j;
    fscanf (input, "%d", &n);
    for (i = 0; i < n; i++) fscanf (input, "%d", &P[i]);
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            fscanf (input, "%d", &T[i][j]);
    for (i = 0; i < n;i++)
        if (T[0][i] <= P[i])
            best[i]=1;
    bool chg;
```

```
    do {
        chg=0;
        for (i = 0; i < n; i++)
            for (j = 0; j < n;j++)
                if (i != j && best[j] <= best[i] && P[j]-P[i] >= T[i][j]) {
                    chg = 1;
                    best[j] = best[i] + 1;
                }
    } while(chg);
    int wyn = 0;
    for (i = 0; i < n; i++)
        if (best[i] > wyn) wyn = best[i];
    fprintf (output, "%d\n", wyn);
    return 0;
}
```

2. Apples [Kuipers, 2002] (25, 32) Fall 02 Green

When summer comes to the farm, it is time to harvest the cow's favorite fruit, apples. At FJ's farm, they have an extraordinary way of harvesting apples: Bessie shakes an apple tree, the apples fall down and FJ tries to catch as many of the N (1 <= N <= 5,000) apples as possible.

As an experienced apple catcher, FJ plots his apple catching carefully. He knows exactly where (an X,Y coordinate where each of X and Y is in the range -1,000..1,000) each apple will fall and when (a time in the range 1 <= T <= 1,000,000). He walks to his planned apple-catching spot so that he can catch the apple as it falls.

FJ can walk S (1 <= S <= 1,000) units per unit of time. What is the greatest number of apples he can catch if he starts at point (0,0) at time 0? Note that FJ takes just over 1.41 units of time to walk from (0,0) to (1,1) when walking at speed 1.

PROBLEM NAME: apples

INPUT FORMAT:

* Line 1: Two space-separated integers: N and S

* Lines 2..N+1: Line i+1 contains the cartesian coordinates and time apple i lands with three space-separated integers: X_i, Y_i, and T_i

SAMPLE INPUT (file apples.in):

```
5 3
0 0 1
0 3 2
```

```
-5 12 6
-1 0 3
-1 1 2

OUTPUT FORMAT:

* Line 1: A file with a single line containing an integer that is the
         largest possible number of apples that FJ can catch.

SAMPLE OUTPUT (file apples.out):

3

OUTPUT DETAILS:

FJ can catch apples 1, 5 and 4.
```

Analysis: Apples by Richard Peng

Since Farmer John can't travel back in time (although the cows can, see HighTech cows), it's best to order the 'apple falls' by their falling time.

Now the DP state and transition becomes quite clear: let best[i] be the maximum number of apples FJ can catch while catching the ith one and the transition is:

best[i]=max(best[j]+1, can get from j to i in time)

The criterion of getting from j to i in time is also fairly easy to check since if FJ is not picking any more apples between those two, he can run straight at tree i right after he caught j. Here is a sample code from mahbub:

```
#include<stdio.h>
#include<algorithm>
using namespace std;

struct Apple
{
        long long x,y,t;
}a[5001];

bool operator<(Apple A,Apple B)
{
        return A.t < B.t;
}
```

```
int ans[5001];

int main()
{
        freopen("apples.in","r",stdin);
        freopen("apples.out","w",stdout);

        int n,i,j,max,s;

        scanf("%d%d",&n,&s);

        for(i=1;i<=n;i++)
                scanf("%lld%lld%lld",&a[i].x,&a[i].y,&a[i].t);
        a[0].x=a[0].y=a[0].t=0;

        sort(a+1,a+n+1);

        for(i=1;i<=n;i++) ans[i]=-1000000;

        for(i=1;i<=n;i++)
        {
                for(j=i-1;j>=0;j--)
                {
                        if(ans[j]+1>ans[i])
                        {
                                if( ((a[i].x-a[j].x)*(a[i].x-a[j].x) + (a[i].y-
a[j].y)*(a[i].y-a[j].y))<=s*s*(a[i].t-a[j].t)*(a[i].t-a[j].t))
                                        ans[i]=ans[j]+1;
                        }
                }
        }

        max=0;
        for(i=1;i<=n;i++) if(max<ans[i]) max=ans[i];

        printf("%d\n",max);

        return 0;
}
```

3. Millenium Leapcow [via Nikolay Valtchanov, from Bulgaria '01, 2003] (27, 38) Open 03 Green

The cows have revised their game of leapcow. They now play in the middle of a huge pasture upon which they have marked a grid that bears a remarkable resemblance to a chessboard of N rows and N columns (3 <= N <= 365).

Here's how they set up the board for the new leapcow game:

* First, the cows obtain N x N squares of paper. They write the integers from 1 through N x N, one number on each piece of paper.

* Second, the 'number cow' places the papers on the N x N squares in an order of her choosing.

Each of the remaining cows then tries to maximize her score in the game:

* First, she chooses a starting square and notes its number.

* Then, she makes a 'knight' move (like the knight on a chess board) to a square with a higher number. If she's particularly strong, she leaps to the that square; otherwise she walks.

* She continues to make 'knight' moves to higher numbered squares until no more moves are possible.

Each 'knight' move earns the competitor a single point. The cow with the most points wins the game.

Help the cows figure out the best possible way to play the game.

PROBLEM NAME: leap2

INPUT FORMAT:

* Line 1: A single integer: N

* Lines 2.....: These lines contain space-separated integers that tell the contents of the chessboard. The first set of lines (starting at the second line of the input file) represents the first row on the chessboard; the next set of lines represents the next row, and so on.

To keep the input lines of reasonable length, when N > 15, a row is broken into successive lines of 15 numbers and a potentially shorter line to finish up a row. Each new row begins on its own line.

SAMPLE INPUT (file leap2.in):

```
4
1 3 2 16
4 10 6 7
8 11 5 12
9 13 14 15
```

OUTPUT FORMAT:

* Line 1: A single integer that is the winning cow's score; call it W.

* Lines 2..W+1: Output, one per line, the integers that are the starting square, the next square the winning cow visits, and so on through the last square. If a winning cow can choose more than one path, show the path that would be the 'smallest' if the paths were sorted by comparing their respective 'square numbers'.

SAMPLE OUTPUT (file leap2.out):

```
7
2
4
5
9
10
12
13
```

OUTPUT DETAILS:

The longest tour consists of the moves 2 to 4, 4 to 5, 5 to 9, 9 to 10, 10 to 12, 12 to 13 and has length of 7 squares.

Analysis: Millenium Leapcow by Richard Peng

Since the values of the squares a cow visits can only increase, it's best to reorder the squares by their values as we could not go backwards on the list.

Now the DP state best[i] is longest path that ends on location i. And the transition would be

best[i]=max(0,{best[j]+1 |can reach i from j in 1 step}

As there are at most 8 transitions per square, this runs in O(n^2) time, where n is the side length of the grid.

```
#include <cstdio&rl;
using namespace std;

int loc[200000][2],bes[200000],fro[200000],gri[400][400],n,dir[8][2]={-2,-1,-
2,1,-1,-2,-1,2,1,-2,1,2,2,-1,2,1};
int rec;

int main(){
        int i,j,a,r1,c1,r2,c2;
freopen("leap2.in","r",stdin);
freopen("leap2.out","w",stdout);
        scanf("%d",&n);
        for(i=0;i<n;i++)
                for(j=0;j<n;j++){
                        scanf("%d",&a);
                        gri[i][j]=--a;
                        loc[a][0]=i;
```

```
                        loc[a][1]=j;
                }
        for(i=n*n-1;i&rl;=0;i--){
                bes[i]=0;
                fro[i]=-1;
                r1=loc[i][0];
                c1=loc[i][1];
                for(j=0;j<8;j++){
                        r2=r1+dir[j][0];
                        c2=c1+dir[j][1];

if((r2&rl;=0)&&(c2&rl;=0)&&(r2<n)&&(c2<n)&&((a=gri[r2][c2])&rl;i)){
                                if((bes[a]&rl;bes[i])||
((bes[a]==bes[i])&&(a<fro[i]))){
                                        fro[i]=a;
                                        bes[i]=bes[a];
                                }
                        }
                }
                bes[i]++;
        }
        for(rec=i=0;i<n*n;i++)
                if(bes[i]&rl;bes[rec]) rec=i;
        printf("%d\n",bes[rec]);
        for(;rec!=-1;rec=fro[rec])
                printf("%d\n",rec+1);
        return 0;
}
```

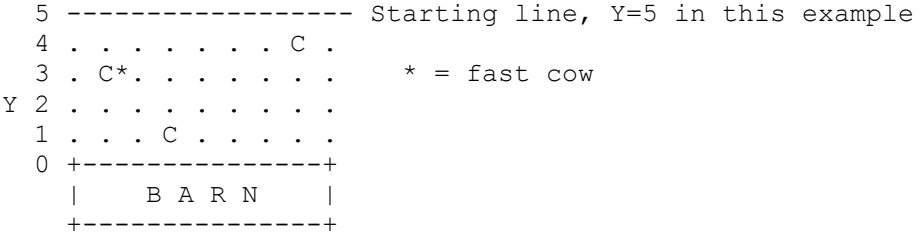
4. Exercising Cows [Percy Liang, 2004] (35, 36) Camp 08 D3

Farmer John's N (1 <= N <= 300) cows have become extremely lazy. They lounge around the pastures doing absolutely nothing, not even mooing. FJ is terribly upset and has an idea to get the cows back in shape: a fit cow gives more milk!

Farmer John drew a starting line parallel to a very wide barn (whose side is at Y=0) and at distance Y (5 <= Y <= 10,000) from that side. Each cow positions herself at a starting place between the barn and the starting line at location X_i,Y_i (1 <= X_i <= 10,000; 1 <= Y_i <= Y); no two cows share the same X coordinate.

When the game starts, each cow races directly towards the side of the barn and, when she reaches the barn, she immediately turns around and races back to the starting line, at which point she turns around, races back to the barn, and so on, hopefully until milking time. All but K (1 <= K <= 8) of the N cows move at the same slow pace; the K (1 <= K <= 8) quick cows move twice as fast. Farmer John moves twice as fast as these fast cows.

FJ thought he had fooled the cows into getting exercise, but there turned out to be a catch: if FJ does not pet cow as she encounters the starting line to cheer her on; she just gives up. Thus, FJ ends up running back and forth along the starting line so he can pet each cow as she approaches y=Y.



```
           0 1 2 3 4 5 6 7 8 <-- X
```

Consider the layout above where the leftmost cow (starting at 1,3) is the fast one. Here is a chart of everyone's position through the first 39 moves:

```
       2     1     1     4    <-- speed
 t   y_C1  y_C2  y_C3  x_FJ
 0     3     1     4     1
 1     1     0     3     1
 2     1     1     2     1
 3     3     2     1     1
 4    >5     3     0     1   <-- FJ pets C1
 5     3     4     1     3
 6     1    >5     2     3   <-- FJ pets C2
 7     1     4     3     5
 8     3     3     4     7
 9    >5     2    >5     7   <-- FJ pets C3; C1 gives up
10           1     4     5
11           0     3     3
12           1     2     2
13           2     1     2
14           3     0     2
15           4     1     2
16          >5     2     2   <-- FJ pets C2
17           4     3     4
18           3     4     6
19           2    >5     7   <-- FJ pets C3
20           1     4     5
31           0     3     3
32           1     2     2
33           2     1     2
34           3     0     2
35           4     1     2
36          >5     2     2   <-- FJ pets C2
37           4     3     4
38           3     4     5
39           2    >5     7
               ...
```

FJ can keep two cows -- and only two cows -- running.

Help FJ find the maximum number of cows that he can rescue from indolence (i.e., by running indefinitely). FJ can position himself at any starting location.

PROBLEM NAME: exercise

INPUT FORMAT:

* Line 1: Three space-separated integers: N, K, and Y

* Lines 2..K+1: Line i+1 describes fast cow i with two space-separated integers: X_i and Y_i

* Lines K+2..N+1: Line i+K+1 describes slow cow i with two space-separated integers: X_i and Y_i

SAMPLE INPUT (file exercise.in):

```
3 1 8
1 3
3 1
11 4
```

OUTPUT FORMAT:

* Line 1: One integer representing the maximum number of cows Farmer John can encourage to play the game indefinitely.

SAMPLE OUTPUT (file exercise.out):

```
2
```

No analysis available yet ☹

5.Treasures [BulgarianOI, 2005] (38, 40) Camp 04

N (1 <= N <= 1,000) treasures have been buried and can be dug out to earn money. While digging is expensive, finding a treasure can create tremendous profit (after expenses are subtracted from the treasure's value). Create a program to compute the maximum profit possible from a set of buried treausres.

The treasures each have value Pi (1 <= Pi <= 1,000,000) and are buried on a two-dimensional grid split into squares, each with coordinates Xi,Yi (-10,000 <= Xi <= 10,000 and -10,000 <= Yi < 0).

Digging the square with coordinates (x, y) is permitted only if y = -1 or if the three squares with coordinates (x-1, y+1), (x, y+1) and (x+1, y+1) have already been dug. The cost for digging a square is 1. When a treasure is dug out, its value is considered earned.

Determine the maximum possible profit by digging the optimal number of treasures (which might be 0).

PROBLEM NAME: treasures

INPUT FORMAT:

* Line 1: The single integer N

* Lines 2..N+1: Line i+1 contains three space-separated integers: Xi, Yi, and Pi.

SAMPLE INPUT (file treasures.in):

```
2
7 -2 9
8 -10 20
```

OUTPUT FORMAT:

* Line 1: A single integer that is the maximum achievable profit

SAMPLE OUTPUT (file treasures.out):

5

Analysis: Treasures by Md. Mahbubul Hasan

Seeing a huge limit of X and Y and N≤=1000 we can easily come to conclusion that we need a N^2 dp solution.

It is quite obvious that when we select a treasure then actually we are selecting a triangle having pick at that point and base on y=-1.

From now on, I will consider base is y=1. (We just make: y[i]=-y[i] for our convenience.)

Now the solution may not seem trivial in the first glance. So we start drawing
some pyramids. We need some kinds of ordering to run the dp. So seeing some figure we can decide to sort by a diagonal of the pyramid. Say we are sorting by: x-y diagonal, and if some two have same x-y diagonal then we sort them by y value.

Now if we consider A and B two pyramids(A comes before B in sorted list) then there can be 3 cases:

1. B is inside A.
2. B is completely right of A (does not intersect)
3. B intersects A.

(If A is inside B then we will consider B intersects A)

this 3 condition can be checked with the help of 4 variables for each pyramid:

left diagonal (x-y)
right diagonal (x+y)
left end point (the left point of pyramid on y=1)
right end point (the right point of pyramid on y=1)

For case 1, just add the treasure to A.
For case 2 and 3, check whether taking A with B is optimal or not.

Be careful when computing the intersection area! There are two types of intersection: (1-3-5-7... (y*y)) and (2-4-6-8... y*(y+1))

Here is my code implementing the above idea:

```
#include<stdio.h>
#include<algorithm>
using namespace std;

int dp[2000];
int d1[2000],d2[2000];
int left[2000],right[2000];

struct TREASURE
{
	int x,y,p;
}T[2000];

bool operator<(TREASURE A, TREASURE B)
{
	if(A.x-A.y < B.x-B.y) return 1;
	if(A.x-A.y == B.x-B.y) return A.y<B.y;
	return 0;
}

int F(int x1,int y1,int d1l,int d1r,int x2,int y2,int d2l,int d2r)
{
	int yt;

	if((d2l-d1r)%2==0)
	{
		yt=(d2l-d1r)/2;
		return y2*y2-yt*yt;
	}
	else
	{
		yt=(d2l-d1r-1)/2;
		return y2*y2-yt*(yt+1);
	}

}

int main()
{
	freopen("treasures.in","r",stdin);
	freopen("treasures.out","w",stdout);

	int N,total,ans;
	int i,j;

	scanf("%d",&N);
	for(i=0;i<N;i++)
	{
		scanf("%d%d%d",&T[i].x,&T[i].y,&T[i].p);
```

```
                T[i].y=-T[i].y;
        }

        sort(T,T+N);

        for(i=0;i<N;i++)
        {
                dp[i]=-T[i].y*T[i].y+T[i].p;
                d1[i]=T[i].x-T[i].y;
                d2[i]=T[i].x+T[i].y;
                left[i]=d1[i]+1;
                right[i]=d2[i]-1;

                for(j=0;j<i;j++)
                {
                        if(d1[j]<=d1[i] && d2[i]<=d2[j])
                        {
                                dp[j]+=T[i].p;
                                continue;
                        }

                        if(right[j]<left[i])
                        {
                                if(dp[j]-T[i].y*T[i].y+T[i].p>dp[i])
                                        dp[i]=dp[j]-T[i].y*T[i].y+T[i].p;
                                continue;
                        }


total=F(T[j].x,T[j].y,d1[j],d2[j],T[i].x,T[i].y,d1[i],d2[i]);

                        if(-total+T[i].p+dp[j] > dp[i])
                                dp[i]=-total+T[i].p+dp[j];
                }
        }

        ans=0;
        for(i=0;i<N;i++) if(dp[i]>ans) ans=dp[i];

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

        return 0;
}
```

6. The Cow Lexicon [Vladimir Novakovski, 2002] (38, 40) USAICO

The cows have a dictionary, you know. It contains W (6 <= W <= 1000) words, each at most 20 characters long. Because their cowmunication system, based on mooing, is not very accurate, sometimes they hear words that do not make any sense. For instance, Bessie once received a message that said "browndcodw." As it turns out, the intended message was "browncow" and the two letter "d"s were noise from other parts of the barnyard.

The cows want you to help them decipher a received message of length L (2 <= L <= 1600) characters that is a bit garbled. In particular, they know that the message has some extra letters, and they want you to determine the smallest number of letters that must be removed to make the message a sequence of words from the dictionary.

PROBLEM NAME: lexicon

INPUT FORMAT:

* Line 1: Two space-separated integers, respectively W and L

* Line 2: L characters (followed by a newline, of course): the received message (the message contains only the characters 'a'..'z')

* Lines 3..W+2: The dictionary, one word per line; words consist only of the characters 'a'..'z'.

SAMPLE INPUT (file lexicon.in):

```
6 10
browndcodw
cow
```

```
milk
white
black
brown
farmer

OUTPUT FORMAT:

A single line with a single integer that is the smallest number of
characters that need to be removed to make the message a sequence of
dictionary words.

SAMPLE OUTPUT (file lexicon.out):

2
```

Analysis: The Cow Lexicon by Richard Peng

The state again comes out fairly simple, with the ith state (best[i]) being the minimum number of character to remove such that w[1..i] can be covered by words in the dictionary given. The transition is also fairly simple:

best[i]=max(best[j]+minimum number of characters to remove so s[j+1..i] is a word in the dictionary)

since we can think of adding one word at a time.

The tricky part is to compute transition(i,j), the minimum number of character to remove so that w[i..j] is a word in the dictionary.

For each word w in the dictionary and j, we can find the shortest substring of s[i..j] that is w. To do this fast, we can loop up in the length of the word k and at for each j store the latest i such that w[1..k] is a substring of s[i..j]. This takes O(length of the text * length of the word) for each word, which is managable since |w|<=20 for all words.

Here is the code implementing the above idea:

ans[i] contains minimum number of character needed to be removed from text[0 to i-1].

temp[i][j] contains the place of the optimal position of j'th character of a particular word considering characters are available from i'th place in the text.

best[i][j] contains the minimum number of character needed to be removed from text[i to j] to make it a dictionary word.

```
#include<stdio.h>
#include<string.h>

#define min(A,B) ((A) < (B) ? (A) : (B))

int best[1700][1700];
int ans[2000];
int temp[2000][30];

int main()
{
        freopen("lexicon.in","r",stdin);
        freopen("lexicon.out","w",stdout);

        int W,L,i,j,now,len;
        char word[100],text[2000];

        scanf("%d%d",&W,&L);
        scanf("%s",text);

        for(i=0;i<L;i++) for(j=i;j<L;j++) best[i][j]=j-i+1;

        while(W--)
        {
                scanf("%s",word);
                len=strlen(word);

                temp[0][0]=((word[0]==text[0]) ? 0 : -1);
                for(i=1;i<L;i++)
                {
                        if(word[0]==text[i]) temp[i][0]=i;
                        else temp[i][0]=temp[i-1][0];
                }

                for(j=1;j<len;j++)
                {
                        for(i=0;i<L;i++)
                        {
                                if(i<j) temp[i][j]=-1;
                                else if(word[j]==text[i])
                                {
                                        if(temp[i-1][j-1]!=-1) temp[i][j]=i;
                                        else temp[i][j]=-1;
                                }
                                else temp[i][j]=temp[i-1][j];
                        }
                }
```

```
                for(i=0;i<L;i++) if(temp[i][len-1]!=-1)
                {
                        now=temp[i][len-1];
                        for(j=len-1;j>=1;j--) now=temp[now-1][j-1];
                        best[now][i]=min(best[now][i],i-now+1-len);
                }
        }

        for(ans[0]=i=0;i<L;i++)
        {
                ans[i+1]=i+1;
                for(j=0;j<i;j++) ans[i+1]=min(ans[i+1],ans[j-
1+1]+best[j][i]);
        }

        printf("%d\n",ans[L-1+1]);

        return 0;
}
```

INTERVAL BASED DP

1.Treats for the Cows [Marco Gallotta, 2005] (18, 30) Feb 06 Gold
FJ has purchased N (1 <= N <= 2000) yummy treats for the cows who get money for giving vast amounts of milk. FJ sells one treat per day and wants to maximize the money he receives over a given period time.

The treats are interesting for many reasons:
  * The treats are numbered 1..N and stored sequentially in single file in a long box that is open at both ends. On any day, FJ can retrieve one treat from either end of his stash of treats.
  * Like fine wines and delicious cheeses, the treats improve with age and command greater prices.
  * The treats are not uniform: some are better and have higher intrinsic value. Treat i has value v(i) (1 <= v(i) <= 1000).
  * Cows pay more for treats that have aged longer: a cow will pay v(i)*a for a treat of age a.

Given the values v(i) of each of the treats lined up in order of the index i in their box, what is the greatest value FJ can receive for them if he orders their sale optimally?

The first treat is sold on day 1 and has age a=1. Each subsequent day increases the age by 1.
PROBLEM NAME: trt
INPUT FORMAT:

* Line 1: A single integer, N
* Lines 2..N+1: Line i+1 contains the value of treat v(i)

SAMPLE INPUT (file trt.in):

5

```
1
3
1
5
2

INPUT DETAILS:

Five treats. On the first day FJ can sell either treat #1 (value 1) or
treat #5 (value 2).

OUTPUT FORMAT:

* Line 1: The maximum revenue FJ can achieve by selling the treats

SAMPLE OUTPUT (file trt.out):

43

OUTPUT DETAILS:

FJ sells the treats (values 1, 3, 1, 5, 2) in the following order
of indices: 1, 5, 2, 3, 4, making 1x1 + 2x2 + 3x3 + 4x1 + 5x5 = 43.
```

USACO February 2006 Problem 'treat/trt' Analysis

by Bruce Merry

This problem can be solved by dynamic programming. Consider starting with a particular pair of treats exposed. This pair determines how much time has already passed and thus how much the remaining treats are worth. At this point there are two choices, corresponding to the two treats that can be removed. The return on each treat is known, and in each case dynamic programming will give the return on the remaining treats. It is thus possible to compute the optimal return for the starting pair in $O(N^2)$ time. With proper sequencing it is also possible to use only O(N) memory, although that optimisation is not necessary here.

USA's Robert Mitchell Burke submitted this Java solution:

```
import java.io.*;
import java.util.*;

public class treat {
    public static void main(String[] NEVAR_USE_THIS_VARIABLE353512) throws
Throw
able {
```

```
        BufferedReader in=new BufferedReader(new FileReader("treat.in"));
        PrintWriter out=new PrintWriter("treat.out");

        int N=Integer.parseInt(in.readLine());
        int[] v=new int[N];

        for(int i=0;i<v.length;i++)
            v[i]=Integer.parseInt(in.readLine());

        int[] prev=new int[N+4];
        int[] dp=new int[N+4];

        for(int i=1;i<=N;i++) {
            for(int k=0;k<i;k++) {
                dp[k]=Math.max(dp[k],prev[k]+(i*v[N-i+k]));
                dp[k+1]=Math.max(dp[k+1],prev[k]+(i*v[k]));
            }
            prev=dp;
            dp=new int[N+4];
        }
        int max=0;
        for(int i:prev)
            max=Math.max(max,i);
        out.println(max);
        out.close();
        System.exit(0);
    }
}
```

2. The Cow Run [Chris Tzamos, 2006] (28, 34)

The N (1 <= N <= 1,000) cows have escaped from the farm and have gone on a rampage. Each minute a cow is outside the fence, she causes one dollar worth of damage. Farmer John must visit each one to install a halter that will calm the cow and stop the damage.

Happily, the cows are positioned in a straight line on a road outside the farm. FJ can see each of them and knows every cow i's unique coordinates (-500,000 <= P_i <= 500,000, P_i != 0) relative to the gate (position 0) where FJ starts.

FJ moves at one unit of distance per minute and can install a halter instantly. Determine the order that FJ should visit the cows so he can minimize the total cost of the damage; report the minimum total damage.

PROBLEM NAME: cowrun

INPUT FORMAT:

* Line 1: A single integer: N
* Lines 2..N+1: Line i+1 contains a single integer: P_i

SAMPLE INPUT (file cowrun.in):

```
4
-2
-12
3
```

```
7

INPUT DETAILS:
Four cows placed in positions: -2, -12, 3, and 7.

OUTPUT FORMAT:
* Line 1: The minimum total cost of the damage.

SAMPLE OUTPUT (file cowrun.out):

50

OUTPUT DETAILS:

The optimal visit order is -2, 3, 7, -12. FJ arrives at position
-2 in 2 minutes for a total of 2 dollars in damage for that cow.

He then travels to position 3 (distance: 5) where the cumulative
damage is  2 + 5 = 7 dollars for that cow.

He spends 4 more minutes to get to 7 at a cost of 7 + 4 = 11 dollars
for that cow.

Finally, he spends 19 minutes to go to -12 with a cost of 11 + 19
= 30 dollars.

The total damage is 2 + 7 + 11 + 30 = 50 dollars.
```

Analysis: The Cow Run by Neal Wu

We first sort the cows' positions in order, and we add a cow at location 0 for simplicity. A key observation is that at any time, the set of cows we have visited forms a consecutive sequence in the sorted list of cows. Thus, we can do DP on the interval of cows we have visited so far. This gives a solution that is $O(N^2)$ overall.

The following is a sample solution:

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

FILE *fin = fopen ("cowrun.in", "r");
FILE *fout = fopen ("cowrun.out", "w");

const int MAXN = 1005;

int N;
int cows [MAXN];
int best [MAXN][MAXN][2];

int main ()
```

```
{
    memset (best, 63, sizeof (best));

    fscanf (fin, "%d", &N);

    for (int i = 1; i <= N; i++)
        fscanf (fin, "%d", cows + i);

    cows [++N] = 0;

    sort (cows + 1, cows + N + 1);

    for (int i = 1; i <= N; i++)
        if (cows [i] == 0)
            best [i][1][0] = 0;

    for (int len = 1; len < N; len++)
    {
        int ccount = N - len;

        for (int i = 1; i + len <= N + 1; i++)
        {
            best [i - 1][len + 1][0] <?= best [i][len][0] + ccount * (cows [i]
-
cows [i - 1]);
            best [i - 1][len + 1][0] <?= best [i][len][1] + ccount * (cows [i
+ len
- 1] - cows [i - 1]);

            best [i][len + 1][1] <?= best [i][len][0] + ccount * (cows [i +
len] -
cows [i]);
            best [i][len + 1][1] <?= best [i][len][1] + ccount * (cows [i +
len] -
cows [i + len - 1]);
        }
    }

    fprintf (fout, "%d\n", best [1][N][0] <? best [1][N][1]);

    return 0;
}
```

In addition, the memory of the solution can be reduced to O(N), as in the following solution:

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

FILE *fin = fopen ("cowrun.in", "r");
FILE *fout = fopen ("cowrun.out", "w");

const int MAXN = 1005;

int N;
int cows [MAXN];
```

```
int best [MAXN][2], best2 [MAXN][2];

inline void mini (int &a, int b)
{
    if (b < a) a = b;
}

int main ()
{
    memset (best, 63, sizeof (best));

    fscanf (fin, "%d", &N);

    for (int i = 1; i <= N; i++)
        fscanf (fin, "%d", cows + i);

    cows [++N] = 0;

    sort (cows + 1, cows + N + 1);

    for (int i = 1; i <= N; i++)
        if (cows [i] == 0)
            best [i][0] = 0;



    for (int len = 1; len < N; len++)
    {
        int ccount = N - len;

        memset (best2, 63, sizeof (best2));

        for (int i = 1; i + len <= N + 1; i++)
        {
            mini (best2 [i - 1][0], best [i][0] + ccount * (cows [i] - cows [i - 1]));
            mini (best2 [i - 1][0], best [i][1] + ccount * (cows [i + len - 1] - cows
[i - 1]));

            mini (best2 [i][1], best [i][0] + ccount * (cows [i + len] - cows [i]));
            mini (best2 [i][1], best [i][1] + ccount * (cows [i + len] - cows [i + len
- 1]));
        }

        memcpy (best, best2, sizeof (best));
    }

    fprintf (fout, "%d\n", best [1][0] <? best [1][1]);

    return 0;
}
```

3. Problem Solving [Hal Burch, 2004] (28, 33) Jan 07 Gold

In easier times, Farmer John's cows had no problems. These days, though, they have problems, lots of problems; they have P (1 <= P <= 300) problems, to be exact. They have quit providing milk and have taken regular jobs like all other good citizens; in a normal month they make M (1 <= M <= 1000) money.

Their problems, however, are so complex they must hire consultants to solve them. Consultants are not free, but they are competent: consultants can solve any problem in a single month. Each consultant demands two payments: one in advance (1 <= payment <= M) to be paid at the start of the month problem-solving is commenced and one more payment at the start of the month after the problem is solved (1 <= payment <= M). Thus, each month the cows can spend the money earned during the previous month to pay for consultants. Cows are spendthrifts: they can never save any money from month-to-month; money not used is wasted on cow candy.

Since the problems to be solved depend on each other, they must be solved mostly in order. For example, problem 3 must be solved before problem 4 or during the same month as problem 4.

Determine the number of months it takes to solve all of the cows' problems and pay for the solutions.

PROBLEM NAME: psolve

INPUT FORMAT:

* Line 1: Two space-separated integers: M and P.
* Lines 2..P+1: Line i+1 describes problem i with two space-separated integers: B_i and A_i. B_i is the payment to the consult BEFORE the problem is solved; A_i is the payment to the consult AFTER the problem is solved.

SAMPLE INPUT (file psolve.in):

```
100 5
40 20
60 20
30 50
30 50
40 40
```

INPUT DETAILS:
The cows make 100 money each month. They have 5 problems to solve, which cost 40, 60, 30, 30, and 40 in advance to solve and then 20, 20, 50, 50, and 40 at the beginning of the month after the problems are solved.

OUTPUT FORMAT:
* Line 1: The number of months it takes to solve and pay for all the cows' problems.

SAMPLE OUTPUT (file psolve.out):

```
6
```

OUTPUT DETAILS:

Month	Avail Money	Probs Solved	Before Payment	After Payment	Candy Money
1	0	-none-	0	0	0
2	100	1, 2	40+60	0	0
3	100	3, 4	30+30	20+20	0
4	100	-none-	0	50+50	0
5	100	5	40	0	60
6	100	-none-	0	40	60

Analysis: Problem Solving by Bruce Merry / Richard Peng

There are two ways that this problem can be solved in time, both based on dynamic programming. The first runs in O(P^3) time, independent of M. Let dp[X][Y] be the minimum number of months necessary to solve the first Y problems, with problems X through Y solved in the last month. To compute this, we need consider which problems were solved in the previous months in which we solved problems. Say we previously solved problems W through X-1; if sum[W..X-1] A[i] + sum[X..Y] B[i] > M, then we cannot do this in consecutive months and have to leave a month between to accumulate more money, otherwise we can do them consecutively. Iterating over all values of W for which problems W to X-1 do not themselves consume more than M money at the start or end of the month, we take the one that gives us the earlier possible start for X..Y.

The other solution runs in O(P^2M) time. The states used for dynamic programming are the last solved problem and the amount of money left over after making the end payment for that problem. This is really the same information stored by the previous algorithm, since an (X, Y) pair says which problem was solved last (Y) and how much money is left after paying for it

(M - sum[X..Y] A[i]). The implementation is conceptually a bit simpler; details are left to the reader.

There is a way to reduce the runtime by a factor of P pointed out by a number of contestants. The method to reduce it for the O(P^3) solution is described here. The state transfer function of the dynamic programming function above is essentially dp[x][y]->dp[y+1][z] where sum[x...y]b[i]+sum[y+1..z]b[i] does not exceed m. Then if we fix y, decrease z, the range of possible x candidates to be considered can only increase as sum[y+1...z]b[i] is decreasing. Therefore, it suffices to track the optimum in the range as its being extended in linear time, bringing the total runtime to O(P^2)

Note: this was meant to be an easy problem due to the well known nature of this type of DP state transition functions and the inclusion of schul on the contest. Therefore, the O(P^3) solution was deemed to be sufficient to get full points and the O(P^2) solution was overlooked

Code by Relja Petrovic, who was one of the contestants who pointed this out:

```
#include <cstdio>
#define ffor(_a,_f,_t) for(int _a=(_f),__t=(_t);_a<__t;_a++)

FILE *fin,*fout;
int m , p , vp[303] , vk[303] , dp[303] , cp[303];

int main(){

        fin = fopen("psolve.in","r");
        fscanf(fin,"%d %d",&m,&p);

        ffor(i,1,p+1)
                fscanf(fin,"%d %d",vp+i,vk+i);

        dp[0]=1;
        cp[0] = vp[0] = vk[0] = 0;

        int f,k,j,s;
        ffor(i,1,p+1){
                dp[i]=1000000000;
                f=vp[i],k=vk[i];
                for(j=i-1; j>=0 && f<=m && k<=m;--j){
                        s = dp[j]+2-(cp[j]<=m-f);
                        if (s<dp[i] || (s==dp[i] && cp[i]>k))
                                dp[i]=s , cp[i]=k;
```

```
                        f += vp[j];
                        k += vk[j];
                }
        }
        fout = fopen("psolve.out","w");
        fprintf(fout,"%d\n",dp[p]+1);

        fclose(fout);fclose(fin);

        return 0;
}
```

4. Cheapest Palindrome [Eko Mirhard, 2007](29, 33) Open 07 Gold

Memory Limit: 18MB due to system change.

Keeping track of all the cows can be a tricky task so Farmer John has installed a system to automate it. He has installed on each cow an electronic ID tag that the system will read as the cows pass by a scanner. Each ID tag's contents are currently a single string with length M (1 <= M <= 2,000) characters drawn from an alphabet of N (1 <= N <= 26) different symbols (namely, the lower-case roman alphabet).

Cows, being the mischievous creatures they are, sometimes try to spoof the system by walking backwards. While a cow whose ID is "abcba" would read the same no matter which direction the she walks, a cow with the ID "abcb" can potentially register as two different IDs ("abcb" and "bcba").

FJ would like to change the cows's ID tags so they read the same no matter which direction the cow walks by. For example, "abcb" can be changed by adding "a" at the end to form "abcba" so that the ID is palindromic (reads the same forwards and backwards). Some other ways to change the ID to be palindromic include adding the three letters "bcb" to the begining to yield the ID "bcbabcb" or removing the letter "a" to yield the ID "bcb". One can add or remove characters

at any location in the string yielding a string longer or shorter
than the original string.

Unfortunately as the ID tags are electronic, each character insertion
or deletion has a cost (0 <= cost <= 10,000) which varies depending
on exactly which character value to be added or deleted. Given the
content of a cow's ID tag and the cost of inserting or deleting
each of the alphabet's characters, find the minimum cost to change
the ID tag so it satisfies FJ's requirements. An empty ID tag is
considered to satisfy the requirements of reading the same forward
and backward. Only letters with associated costs can be added to a
string.

PROBLEM NAME: cheappal

INPUT FORMAT:

* Line 1: Two space-separated integers: N and M

* Line 2: This line contains exactly M characters which constitute the
        initial ID string

* Lines 3..N+2: Each line contains three space-separated entities: a
        character of the input alphabet and two integers which are
        respectively the cost of adding and deleting that character.

SAMPLE INPUT (file cheappal.in):

```
3 4
abcb
a 1000 1100
b 350 700
c 200 800
```

INPUT DETAILS:

The nametag is "abcb" with these per-operation costs:

	Insert	Delete
a	1000	1100
b	350	700
c	200	800

OUTPUT FORMAT:

* Line 1: A single line with a single integer that is the minimum cost
        to change the given name tag.

SAMPLE OUTPUT (file cheappal.out):

```
900
```

OUTPUT DETAILS:

If we insert an "a" on the end to get "abcba", the cost would be
1000. If we delete the "a" on the beginning to get "bcb", the cost
would be 1100. If we insert "bcb" at the begining of the string, the
cost would be 350+200+350=900, which is the minimum.

Analysis: Cheapest Palindrome by Richard Peng

This problem can be done using dynamic programming (DP). First observe that the cost of inserting/removing characters can be combined into one cost since inserting another of of the same character at the mirrored poisition would have the same effect as deleting the character. So we can let cost[ch] be the minimum of the two costs for inserting/deleting character ch.

Now consider the dp state of [l,r] which represents a substring. For each state, we try to minimize the cost of changing that substring into a palindrome. Then the state transition function would be:

DP[l,r]=min{DP[l+1,r]+cost[s[l]]. ('pair up' left most character)
DP[l,r+1]+cost[s[r]], ('pair up' right most character)
DP[l+1,r-1] (only if the two end characters are equal, aka. s[l]=s[r]}

The state transition takes O(1) time while there are a total of O(M^2) states. So the algorithm runs in O(M^2) time.

Test Data

All data were generated randomly while specifying N and M.
It's intended that a brute-force search would get cases 1-4, anything O(M^3) or faster would get cases 1-8 while the O(M^2) solution is required to get full points.

Below is the solution of China's Yang Yi:

```
#include <stdio.h>
#define MAXN 2001

FILE *in = fopen("cheappal.in","r");
FILE *out = fopen("cheappal.out","w");

int n, m, is[26], de[26], dp[MAXN][MAXN];
char str[MAXN + 1];

int main () {
    int i, j, a, b;
    char ch;
    fscanf (in, "%d%d%s", &n, &m, str);
    for (i = 0; i < n; i ++) {
        do fscanf(in,"%c",&ch);
            while (ch <= ' ');
        fscanf(in,"%d%d",&a,&b);
        is[ch - 'a'] = a;
        de[ch - 'a'] = b;
     }
    for (a = 0; a < m; a ++)
        for (i = 0; i + a < m; i ++) {
            j = i + a;
            if (i == j || i + 1 == j && str[i] == str[j])
                dp[i][j] = 0;
            else {
                dp[i][j] = (dp[i+1][j] + (is[str[i]-'a'] <? de[str[i]-'a']))
<? (dp[i][j-1] + (is[str[j]-'a'] <? de[str[j]-'a']));
                if (str[i] == str[j])
                    dp[i][j] <?= dp[i+1][j-1];
                }
            }
    fprintf(out,"%d\n",dp[0][m - 1]);
    fclose(in);
    fclose(out);
    return 0;
}
```

5.Haybale Storage [Brian Dean, 2004] (32, 39)

Farmer John would like to store haybales in one of his old two-dimensional barns whose integer width is B (1 <= B <= 250) units and whose integer height H_i (at x=i) is also in the range (1 <= H_i <= 250). The trouble with this particular barn is that it has a very sloped roof. A cross section of the barn looks something like this (in particular, the roof always slopes upward then back downward):

```
           xxxxxxxxxx
       xxxx          xxxxxx
    xxx                    xx
  xx                        x
 x          (inside)        x
x                           x
```

Each of FJ's N (1 <= N <= 250) bales of hay has height H_i (1..250) and width W_i (1..250) . These fragile hay bales may neither be rotated nor stacked on top of each other, so one packing of a set of hay bales might look something like this:

```
           xxxxxxxxxx
       xxxx555       xxxxxx
    xxx444 555             xx
```

```
  xx22 444 555          7777x
 x  223444 555 6666666  7777x
x111223444 555 6666666  7777x
```

It's a pity one can't stack more bales on top of the 6's, but that's
the rule.

Given the width and height of each bale of hay and a description
of the sloped roof, determine the maximum amount of hay (in terms
of total cross sectional area) that can be packed into the barn.

PROBLEM NAME: storage

INPUT FORMAT:

* Line 1: Two space-separated integers: N and B

* Lines 2..N+1: Line i+1 describes bale i with two space-separated
        integers: W_i and H_i

* Lines N+2..N+1+B: Line N+i+1 contains a single integer: H_i

SAMPLE INPUT (file storage.in):

```
4 6
4 1
2 1
2 2
1 3
1
2
2
3
2
1
```

INPUT DETAILS:

The barn looks like this:

```
    x
  xx x
 x    x
x      x
```

OUTPUT FORMAT:

* Line 1: One integer, the maximum obtainable cross-sectional area.

SAMPLE OUTPUT (file storage.out):

```
9
```

OUTPUT DETAILS:

Bales 2, 3 and 4 can be packed like this:

```
   xxx
 xxx4xx
```

```
xx334 xx
x 33422x
```

Analysis: Haybale Storage by Richard Peng

Due to the shape of the ceiling, we can order the haybales by their height in increasing order. By 'pushing' as much bales to the ends as possible, we can also ensure the unused region we're still considering is a continuous interval. So we may sort the haybales in increasing order of the height, maintain the interval of the storage locations we have not used yet. The transition would be either leaving one of the ends empty, or put the bale on one of the ends.

Note we need to consider the intervals in decreasing lengths and for each interval where we could put bales, we need to find the minimum height of the ceiling in that region so we can check whether we can fit a bale in in O(1) time.

Mahbub adds:

We sort the Haybales by their height. Then we run a N^3 DP, we can do it in many ways. My first choice was: 3 parameters, first(i)- starting point,

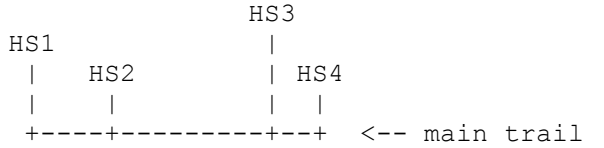
second(j)-ending point, third(k)-serial no of haybale, means 0~(i-1) and (j+1)~(B-1) slots are filled with the haybales 0~k. But this exceeds memory limit as it requires N^3 memory. Then when i looked at my recurrence i understood that i only needed (k-1) at a particular k. So i moved the k loop to the beginning and brought memory limit to 2N^2 using two N^2 matrix. Here is my code:

```
/*
Prob ID: 0263
Name: storage
Author: Md. Mahbubul Hasan
Date: 1st January, 2008
LANG: C++
*/
#include<stdio.h>
#include<algorithm>
using namespace std;

#define max(A,B) ((A) > (B) ? (A) : (B))
struct Box
{
        int w,h;
}box[251];

bool operator<(Box A, Box B) {return A.h < B.h;}

int dp[252][252][2];
int left[252][252],right[252][252];
int h[252];



int main()
{
        freopen("storage.in","r",stdin);
        freopen("storage.out","w",stdout);

        int N,B,i,j,cnt,k,ans,x;

        scanf("%d%d",&N,&B);

        for(i=0;i<N;i++) scanf("%d%d",&box[i].w,&box[i].h);
        for(i=0;i<B;i++) {scanf("%d",&h[i]);}

        sort(box,box+N);

        for(i=0;i<N;i++)
        {
                cnt=0;
                for(j=0;j<B;j++)
                {
                        if(h[j]>=box[i].h) cnt++;
                        else cnt=0;

                        if(j-box[i].w+1>=0) right[i][j-
```

```
box[i].w+1]=(cnt>=box[i].w);
                        left[i][j]=(cnt>=box[i].w);
                }
        }

        x=0;
        for(k=0;k<N;k++)
        {
                for(i=0;i<=B;i++)
                        for(j=B-1;j>=i-1;j--)
                        {
                                if(j==-1) continue;
                                dp[i][j][x]=dp[i][j][1-x];
                                if(i) dp[i][j][x]=max(dp[i][j][x],dp[i-
1][j][x]);
                                if(j<B-1)
dp[i][j][x]=max(dp[i][j][x],dp[i][j+1][x]);
                                if(i>=box[k].w && left[k][i-1])
dp[i][j][x]=max(dp[i][j][x],box[k].h*box[k].w+dp[i-box[k].w][j][1-x]);
                                if(right[k][j+1])
dp[i][j][x]=max(dp[i][j][x],box[k].h*box[k].w+dp[i][j+box[k].w][1-x]);
                        }

                x=1-x;
        }

        ans=0;
        for(i=0;i<B;i++)
                ans=max(ans,dp[i+1][i][1-x]);

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

        return 0;
}
```

6. Needle in a Haystack [Brian Dean, 2006] (33, 37) USAICO

Farmer John is looking for one tiny little needle in the collection of his N (1 <= N <= 200) haystacks. The haystacks are all accessible via paths of different lengths that branch off at right angles from unique locations of a main trail (which corresponds to the line y=0); below is a typical map:



Fortunately, FJ has a remarkable metal detector. If he walks all the way over to a haystack (traversing both an X and Y distance in rectilinear fashion), the detector can tell him if the needle is in that haystack; if not, the detector tells him the needle lies in one of the haystacks to the left (smaller X) or one of the haystacks to the right (larger X) of the current haystack.

FJ begins his search at (0, 0). The N haystacks have coordinates (X_i, Y_1) (-10,000 <= X_i <= 10,000 & -10,000 <= Y_i <= 10,000).

No two haystacks share the same X coordinate. Your goal is to determine how far FJ might walk in the worst case while using an optimal strategy.

The optimal strategy is the one for which the worst needle position is least bad, in terms of how far FJ walks. That is, you should find the minimum over all search strategies of the maximum over all needle locations of the distance that Farmer John walks. Output this value, M.

PROBLEM NAME: haystack

INPUT FORMAT:

* Line 1: A single integer: N

* Lines 2..N+1: Line i+1 contains the two space-separated coordinates of haystack i: X_i and Y_i

SAMPLE INPUT (file haystack.in):

```
4
-9 3
-4 2
6 4
9 2
```

OUTPUT FORMAT:

* Line 1: A single integer, M

SAMPLE OUTPUT (file haystack.out):

```
31
```

OUTPUT DETAILS:

* FJ goes to HS2 (6 units)
* If the needle is on the right, then he goes to HS3 (16 units)
* If the needle is again on the right, then he goes to HS4 (9 units)

These are the worst cases for an optimal strategy.

Analysis: Needle in a Haystack by Neal Wu

This is a fairly straightforward DP problem. Our DP state consists of our current position, the leftmost haystack the needle could be in (call it L), and the rightmost haystack the needle could be in (call it R). To compute the answer for a certain interval, we consider all possible positions we could move to next, and then take the best of those. Unfortunately, this is $O(N^3)$ memory, which is just slightly above the memory limit. However, we can note that except at the beginning, our position is always either the haystack just left of L or the haystack just right of R. Thus there are only two possibilities we have to consider for our position, and we can reduce our memory from $O(N^3)$ to $O(N^2)$. In addition, our solution considers $O(N^2)$ states and for each performs O(N) processing, so our overall runtime is $O(N^3)$.

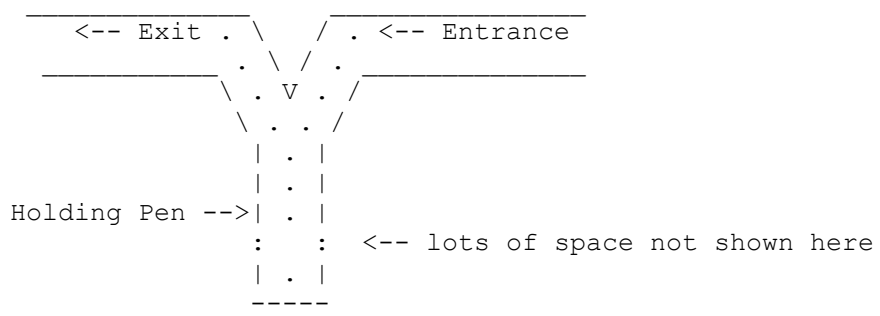
A sample solution follows.

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

FILE *fout = fopen ("haystack.out", "w");
FILE *fin = fopen ("haystack.in", "r");

const int INF = 1000000000;
const int MAXN = 205;

struct cow
{
    int x, y;
};

inline bool operator < (const cow &left, const cow &right)
{
    return left.x < right.x;
}

int N;
cow hay [MAXN];
int best = INF;
int mem [MAXN][MAXN][2];

inline void mini (int &a, int b)
{
    if (b < a) a = b;
}

// the cost of moving from haystack a to haystack b
inline int move (int a, int b)
{
    if (a == b) return 0;

    return abs (hay [a].y) + abs (hay [a].x - hay [b].x) + abs (hay [b].y);
}

int solve (int pos, int left, int right)
{
// pos should be either left - 1 or right + 1, so we only need two states
    int side = (pos < left) ? 0 : 1;

    int &ans = mem [left][right][side];

// already solved
    if (ans != -1) return ans;

// base cases
    if (left >= right)
        return ans = 0;

    if (left + 1 == right)
        return ans = move (pos, left);


    ans = INF;
```

```
// find best place to move next
    for (int i = left; i < right; i++)
       mini (ans, move (pos, i) + max (solve (i, left, i), solve (i, i + 1,
right)));

    return ans;
}

int main ()
{
    memset (mem, -1, sizeof (mem));

    fscanf (fin, "%d", &N);

    for (int i = 0; i < N; i++)
       fscanf (fin, "%d %d", &hay [i].x, &hay [i].y);


    sort (hay, hay + N);

// go through each starting move
    for (int i = 0; i < N; i++)
       mini (best, abs (hay [i].x) + abs (hay [i].y) +
             max (solve (i, 0, i), solve (i, i + 1, N)));


    fprintf (fout, "%d\n", best);


    return 0;
}
```

7.Cow Queuing [Marius Andrei (Romanian National Informatics Olympiad 2005), 2006] (35, 40) USAICO

Farmer John heard about a new way to re-order his N (2 <= N <= 100) cows conveniently numbered 1..N on their way to the barn for dinner. He built a special corral arrangement that looks like this:



All the cows line up to the right of the entrance. Each cow will eventually exit to the left, but must first go into the holding

pen. The holding pen is just wide enough for one cow, and thus acts as a sort-of stack: last-cow-in=>first-cow-out. At any reasonable interval, FJ can direct the cows in one of two ways:

* Direct the cow at the front of the entrance to go into the holding pen.

* Direct the cow that has most recently entered the holding area to the exit.

FJ realizes that by using this corral he can reorder the cows in a large number of ways. FJ chooses a sequence of instructions uniformly from among all possible valid instruction sequences. With three cows in line (sequentially numbered 1, 2, 3) he can effect five final orderings using five instruction sequences:

Instructions	Final Ordering
#1: 1 to exit, 2 to exit, 3 to exit	1, 2, 3
#2: 1 to exit, 2 to queue, 3 to exit, 2 to exit	1, 3, 2
#3: 1 to queue, 2 to exit, 1 to exit, 3 to exit	2, 1, 3
#4: 1 to queue, 2 to exit, 3 to exit, 1 to exit	2, 3, 1
#5: 1 to queue, 2 to queue, 3 to exit, 2 to exit, 1 to exit	3, 2, 1

Bessie is watching FJ implement this scheme. She always wishes to eat ahead of her rival, Joanne. Given Bessie's initial position (B) and Joanne's initial position (J) calculate the fraction of valid instruction sequences which will result in Bessie ending ahead of Joanne.

Express this fraction as a decimal between 0 and 1. Your output should start with a 0 or a 1, have a decimal point, and then have zero or more decimal digits. A special grader will accept all answers within 0.0001 of the official answer.

In the case above, if Bessie is cow #1 and Joanne is cow #3, then Bessie will eat first 60% of the time, with sequences #1, #2, and #3.

PROBLEM NAME: cowq

INPUT FORMAT:

* Line 1: Three space-separated integers: N, B, and J

SAMPLE INPUT (file cowq.in):

```
3 1 3
```

OUTPUT FORMAT:

* Line 1: A decimal number expressed as a floating point number that is the fraction of the time Bessie will eat before Joanne if FJ directs cows randomly.

SAMPLE OUTPUT (file cowq.out):

```
0.600000
```

Analysis: Cow Queuing by Spencer Liang

Another, more convenient, way of looking at this problem is to view the sequence of moves as a list of balanced parentheses --- moving a cow to the holding pen can be interpreted as an opening parenthesis, while moving a cow out of the pen is a closing parenthesis. Then the total number of valid instruction sequences for N cows is the number of ways N pairs of parentheses can be correctly matched, and the sequences for which cow B eats before cow J correspond to the situation where the ')' that matches the Bth '(' comes before the ')' that matches the Jth '('. We can calculate these two values independently, and their quotient is the desired answer.

Imagine an array A[] with N indexes, each containing an '('. Then to calculate the total number of valid instruction sequences, we need to find the number of ways we can place the N ')'s between the indexes of A such

that the parentheses balance. We can do this with dynamic programming: let f(s, e) denote the number of ways we can arrange the ')'s that match the '(' from indexes s to e, and note that the only possible locations for these ')'s are the "gaps" between consecutive indexes. Furthermore, the first pair of matching parentheses (A[s] and its matching ')') always splits the sequence into two smaller (possibly empty) sequences of parentheses, both of which must be balanced. One way of seeing this is to write the sequence as (*)*, where * represents a subsequence, and so both *s must be balanced if (*)* is to be balanced. Combining these two observations yields the recursive formula

```
f(s, e) = f(s+1, i)*f(i+1, e) for all i from s to e.
```

A variant of f can be used to calculate the numerator. Without loss of generality, let B < J and g(s, e) be the number of balanced sequences where the Bth '(' is matched before the Jth ')'. Then the only case where g differs from f is when s == B (it doesn't matter if s < B because the only '(' we match with its ')' in f(s, e) is A[s]) and J <= e (otherwise Bessie eats before Joanne even enters the holding pen). If this is the case (if s == B and J <= e), all we have to do is make sure A[B]'s ')' is placed before A[J] (otherwise Joanne will always eat before Bessie because Joanne's location in the holding pen is ahead of Bessie's). So, we can say

```
g(s, e) = g(s+1, i)*g(i+1, e) for all i from s to e if s != B
        = g(s+1, i)*g(i+1, e) for all i from s to min(e, J-1) if s == B
```

As a side note, the function f is very closely related to the Catalan numbers, and faster methods exist for calculating f. However, $O(N^3)$ is sufficient for the given bounds.

```
#include <stdio.h>
#include <iostream>
#define NUM 105
using namespace std;

FILE *fin, *fout;
int num, realb, realj, bpos, jpos;
double dmem[NUM][NUM], nmem[NUM][NUM];

double calc_denom(int start, int end) {
        int i;
        if (start >= end) return 1;
        if (dmem[start][end] == -1) {
                dmem[start][end] = 0;
                for(i = start; i <= end; i++)
                        dmem[start][end] += calc_denom(start+1,
i)*calc_denom(i+1, end);
        }
        return dmem[start][end];
```

```
}

double calc_numer(int start, int end) {
        int i, last;
        if (start >= end) return 1;
        if (nmem[start][end] == -1) {
                nmem[start][end] = 0;
                if (start == bpos) last = min(end, jpos-1);
                else last = end;
                for(i = start; i <= last; i++)
                        nmem[start][end] += calc_numer(start+1,
i)*calc_numer(i+1, end);
        }
        return nmem[start][end];
}

int main() {
        int i, j, k;
        fin = fopen("cowq.in", "r");
        fout = fopen("cowq.out", "w");
        fscanf(fin, "%d %d %d", &num, &realb, &realj);

        for(i = 0; i <= num; i++) for(j = 0; j <= num; j++)
                dmem[i][j] = nmem[i][j] = -1;

        bpos = min(realb, realj), jpos = max(realb, realj);
        double numer = calc_numer(1, num);
        double denom = calc_denom(1, num);
        double ans = numer/denom;
        if (realj < realb) ans = 1-ans;
        fprintf(fout, "%lf\n", ans);
}
```

8. Grazing on the Run [Brian Dean, 2005] (35, 40) USAICO

A long linear field has N (1 <= N <= 1,000) clumps of grass at unique locations on what will be treated as a number line. Think of the clumps as points on the number line.

Bessie starts at some specified location L on the number line (1 <= L <= 1,000,000) and traverses the number line in the two possible directions (sometimes reversing her direction) in order to reach and eat all the clumps. She moves at a constant speed (one unit of distance in one unit of time), and eats a clump instantly when she encounters it.

Clumps that aren't eaten for a while get stale. We say the ``staleness'' of a clump is the amount of time that elapses from when Bessie starts moving until she eats a clump. Bessie wants to minimize the total staleness of all the clumps she eats.

Find the minimum total staleness that Bessie can achieve while

eating all the clumps.

PROBLEM NAME: ontherun

INPUT FORMAT:

* Line 1 : Two space-separated integers: N and L.

* Lines 2..N+1: Each line contains a single integer giving the
  position P of a clump (1 <= P <= 1,000,000).

SAMPLE INPUT (file ontherun.in):

```
4 10
1
9
11
19
```

INPUT DETAILS:

Four clumps: at 1, 9, 11, and 19. Bessie starts at location 10.

OUTPUT FORMAT:

* Line 1: A single integer: the minimum total staleness Bessie can
  achieve while eating all the clumps.

SAMPLE OUTPUT (file ontherun.out):

```
44
```

OUTPUT DETAILS:

Bessie can follow this route:

* start at position 10 at time 0
* move to position 9, arriving at time 1
* move to position 11, arriving at time 3
* move to position 19, arriving at time 11
* move to position 1, arriving at time 29

giving her a total staleness of 1+3+11+29 = 44. There are other routes with the same total staleness, but no route with a smaller one.

Analysis: Grazing on the Run by Richard Peng

Note that when bessie passes by an uneaten clump, there is no penalty of eating that clump. Therefore, the clump that bessie has eaten will always form an interval, which can always be defined by [i,j], the left and right most clumps respectively.

Bessie's location in the clump also matters, but only when she leaves the clump, therefore 2 additional states can be added, indicating which end of the interval Bessie is on.

We can change around the way we sum the total staleness: everytime bessie moves, we charge a cost equaling to the number of uneaten clumps

times the distance she moves as we can break down the total staleness of each clump into the staleness accumulated in each of her moves.

Then we can DP on the clumps, at each step, we try to extend the interval either one to the left or one to the right, for a constant number of transitions. As there are O(n^2) states, the algorithm runs in O(n^2). Be sure to process in intervals so that if an interval is nested in another, it's processed before the one it's nested in. Processing the intervals in incrementing length will allow the usage of a rotating array, brining down the memory usage to O(n).

Further analysis from Md. Mahbubul Hasan:

We can do it in backword process. Say bessie will start at an endpoint either at pos[0] or pos[n-1] and then she will gradually eat from either end of the uneaten interval. In this solution the state is as stated above, the two ends of uneaten range and at which end bessie is. The code is given below implementing this idea:

```
#include<stdio.h>
#include<algorithm>
using namespace std;

#define abs(A) ((A) > 0 ? (A) : (-(A)))
#define min(A,B) ((A) > (B) ? (B) : (A))

int ans,dp[1002][1002][2];
int pos[1002];

int main()
{
        freopen("ontherun.in","r",stdin);
        freopen("ontherun.out","w",stdout);

        int N,L,i,x,y,len,ok,j;

        scanf("%d%d",&N,&L);

        for(ok=i=0;i<N;i++)
        {
                scanf("%d",&pos[i]);
                ok|=(pos[i]==L);
        }

        sort(pos,pos+N);

        for(i=0;i<N;i++)
             for(j=0;j<N;j++)
                  dp[i][j][0]=dp[i][j][1]=1000000000;

        dp[0][N-1][0]=dp[0][N-1][1]=0;
```

```
        for(len=N-1;len>=0;len--)
                for(x=0;x+len<N;x++)
                {
                        y=x+len;

        dp[x+1][y][0]=min(dp[x+1][y][0],dp[x][y][0]+(x+N-y)*(pos[x+1]-
pos[x]));
        dp[x+1][y][1]=min(dp[x+1][y][1],dp[x][y][0]+(x+N-y)*(pos[y]-pos[x]));
        dp[x][y-1][1]=min(dp[x][y-1][1],dp[x][y][1]+(x+N-y)*(pos[y]-pos[y-
1]));
        dp[x][y-1][0]=min(dp[x][y-1][0],dp[x][y][1]+(x+N-y)*(pos[y]-pos[x]));
                }

        ans=1000000000;
        for(i=0;i<N;i++)
             if(ans>dp[i][i][0]+N*abs(pos[i]-L))
                  ans=dp[i][i][0]+N*abs(pos[i]-L);

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

        return 0;
}
```

9. Milking Orders [Hal Burch and Russ Cox, 2004] (36, 36) Camp 05

Now that dairy products have been mooved to the bottom of the USDA food pyramid, milk is in higher demand than ever. Farmer John has put his recent profits back into the farm, increasing the size of his herd to C (1 <= C <= 1,000) cows. Unfortunately, he didn't leave any money to hire anyone to milk the new cows, so he must milk them all himself.

Each cow wakes up at a particular time W (1 <= W <= 10,000) minutes after dawn each day (in the hot sun on the farm, the days can seem very long). Farmer John must wait for a cow to wake up on her own before the cow can be milked. As you must have noticed, FJ is a laid-back, play-it-by-ear kind of guy. In the old days, he would have just gone out each day, walked around in no particular pattern, and milked cows as they woke up. But now he has so many cows that

he needs to plan the order in which he milks them exactly, so that he can finish in time to visit the farmhouse and watch his favorite evening game show, Cheese Wheel of Fortune.

The cow fields are located all over the farm, but are all reachable via a single winding path of length P (1 <= P <= 1,000) yards that starts at the barn. FJ ambles down the path at a speed of one yard per minute. The farmhouse is at location H (0 <= H <= P) along the path.

Given the waking time of each cow along with the distance of its field from the barn, determine how quickly FJ can milk all the cows and get to the farmhouse. FJ starts at the barn and can milk a cow in no time at all.

PROBLEM NAME: milking

INPUT FORMAT:

* Line 1: Three integers: C, P, and H.

* Lines 2..C+1: Each line contains two space-separated integers describing a cow. The first integer (0..P) is the location of the cow's field along the path. The second integer (0..10,000) is the number of minutes after dawn that the cow wakes up.

SAMPLE INPUT (file milking.in):

```
4 10 3
8 9
4 21
3 16
8 12
```

INPUT DETAILS:

FJ has four cows: Anna, Bessie, Charlotte, and Denise.

OUTPUT FORMAT:

* Line 1: A single integer: the earliest time (in minutes since dawn) that FJ can get to the farmhouse having milked all the cows.

SAMPLE OUTPUT (file milking.out):

```
22
```

OUTPUT DETAILS:

```
Time  Action
  0   FJ walks to the field 8 yards down the path.
  8   FJ rests for one minute, waiting for Anna to wake up, and then milks
Anna.
  9   FJ rests for three more minutes, waiting for Denise to wake up, and
then milks Denise.
 12   FJ walks to the field 4 yards from the barn.
 16   FJ waits five minutes for Bessie to wake up, and then milks Bessie.
```

```
 21   FJ walks to the field 3 yards from the barn (one yard from his
current location).
 22   FJ milks Charlotte, who has been awake for six minutes already.
 22   FJ enters the farmhouse, which is right next to this field.

Thus, FJ can finish his rounds and make it to the farmhouse 22 minutes
after dawn.  No better schedule exists.
```

Analysis: Milking Orders by Richard Peng

Note that the segment of the cows that farmer john has already milked will always form a contiguous segment.

Suppose not, then if all cows in the range [a,b] except c are milked and we're at a. Then there is no point of walking from a to c, milk c and walk from c back to a since on the way back, we can milk all cows in[a,c] anyways.

So we can DP on the segment of cows which has already been milked and which end of the segment we're on. So we can do O(1) transition (on which

side do we extend the range and whether we 'go across' the range) per state for a total runtime of O(n^2).

```
#include <stdio.h>

#define MAX(i,j) ((i) > (j) ? (i) : (j))
#define MIN(i,j) ((i) < (j) ? (i) : (j))

int H, C, D;
int T[1003][1003];
/* T[i][j]: earliest time one can arrive at i having successfully covered
   everything except the interval [min(i,j), max(i,j)].
   All indices are 1-based.
*/
int mintime[1003];

int main(void) {

  int i, j, t, rightmost = 0, answer;
  FILE *fp;
  fp = fopen( "milking.in", "r" );
  fscanf( fp, "%d %d %d", &C, &H, &D );
  for (i=0; i<C; i++) {
    fscanf( fp, "%d %d", &j, &t );
    j++;
    if (j > rightmost) rightmost = j;
    mintime[j] = MAX(mintime[j], t);
  }
  D++;
  fclose(fp);

  /* Base cases --- Bessie starts at position 1. */
  T[1][rightmost] = 0;
  T[rightmost][1] = rightmost - 1;
  for (i=0; i<=rightmost+1; i++)
    T[0][i] = T[i][0] = T[i][rightmost+1] = T[rightmost+1][i] = 999999999;

  if (D >= rightmost) {
    answer = D;
    for (i=1; i<D; i++)
      if (mintime[i] + (D-i) > answer)
        answer = mintime[i] + (D-i);
  } else {

  /* DP */
  for (t = rightmost-2; t >= 0; t--)
    for (i = 1; i <= rightmost; i++ ) {
      j = i+t;
      if (j <= rightmost)
        T[i][j] = MIN( MAX(T[i-1][j], mintime[i-1])+1,
                       MAX(T[j+1][i], mintime[j+1])+t+1 );
      j = i-t;
      if (j >= 1)
        T[i][j] = MIN( MAX(T[i+1][j], mintime[i+1])+1,
                       MAX(T[j-1][i], mintime[j-1])+t+1 );
    }
```

```
  answer = T[D][D];
  }

  fp = fopen( "milking.out", "w" );
  fprintf(fp,"%d\n", MAX(answer, mintime[D]));
  fclose(fp);
}
```

10. Turning in Homework [Hal Burch, 2004] (41, 41) Open 04 Green

Bessie must turn in her homework for her C classes (1 <= C <= 1,000) at Moo U so that she still has time to chew the cud with her fellow classmates as they wait for the bus to go home.

Teachers accept homework submissions only after they have finished their classes and also cleaned the chalkboard, put away lab supplies, and so on. The input data tells the earliest time a teacher will accept homework.

Bessie starts at one end (distance 0) of a hallway H (1 <= H <= 1,000) meters long and walks at the rate of one meter per second to various classrooms (in any order she chooses) to turn in her homework. Each classroom is located along this hallway, as well as the door to the waiting area for the buses.

The distance from the bus waiting area to the hall entrance is B (0 <= B <= H).

Given the location of both the exit and the classrooms and also the teachers' schedules, determine the earliest time that Bessie can exit the door to the waiting area for the buses. Bessie must turn in all her homework before exiting. The act of turning in the homework takes no time, by the way.

PROBLEM NAME: turnin

INPUT FORMAT:

* Line 1: Three integers: C, H, and B.

* Lines 2..C+1: Each line contains two integers that describe a classroom where homework is to be submitted. The first integer (0..H) is the number of meters to the classroom from the hallway entrance. The second integer (0..10,000) is the first time (in seconds) that the teacher for that course will accept homework.

SAMPLE INPUT (file turnin.in):

```
4 10 3
8 9
4 21
3 16
8 12
```

OUTPUT FORMAT:

* Line 1: A single integer: the earliest second that Bessie can exit the door to the waiting area for the buses.

SAMPLE OUTPUT (file turnin.out):

```
22
```

OUTPUT DETAILS:

```
Time   Action
  0   Bessie walks to the classrooms 8 meters down the hall (at 8m)
  8   Bessie waits 1 second
  9   Bessie turns in the first set of homework
  9   Bessie waits 3 seconds, thinking about cool hay in the summertime
 12   Bessie turns in the other homework for this location
 12   Bessie walks back to the classroom 4 meters down the hall (at 4m)
 16   Bessie waits 5 seconds, thinking of a handsome bull she once met
 21   Bessie turns in her homework
 21   Bessie walks back to the classroom 1 meters down the hall (at 3m)
 22   Bessie turns in her homework
 22   Bessie exits, since this also the location of the bus exit
```

Thus, Bessie can leave at time 22. No better schedule exists

Analysis: Turning in Homework by Md. Mahbubul Hasan and Spencer Liang

---

Spencer:

Call the classes for which Bessie has turned in her homework to be covered. Then there exists an optimal schedule such that at any moment in time, the uncovered classes form a contiguous interval. Assume the contrary: there is a covered class B between two uncovered classes A and C, and without loss of generality, let Bessie be at A. Now at some point in the future, Bessie will have to travel from A to C in order to cover all the classes, but then she can cover B at that point, without taking any additional time. Thus it is possible

to modify every optimal schedule such that the set of uncovered classes always forms a contiguous interval.

```
#include <stdio.h>
#include <iostream>
#define MAX 1005
#define INF 0x3f3f3f3f
using namespace std;
#define REP(i,n) for(int i=0;i<n;i++)
#define FORD(i,a,b) for(int i=a;i>=b;i--)
typedef pair<int, int> PII;

int C, H, B; PII A[MAX];
int l[MAX][MAX], r[MAX][MAX];

int main() {
    FILE *fin = fopen("turnin.in", "r"), *fout = fopen("turnin.out", "w");
    fscanf(fin, "%d %d %d", &C, &H, &B);
    REP(i, C) fscanf(fin, "%d %d", &A[i].first, &A[i].second);
    sort(A, A+C);

    memset(l, INF, sizeof(l)); memset(r, INF, sizeof(r));
    l[C-1][0] = A[0].first, r[C-1][0] = A[C-1].first;
    FORD(x, C-2, 0) REP(a, C-x) {
        int b = a+x;
        if (a) l[x][a] <?= max(l[x+1][a-1], A[a-1].second)+A[a].first-A[a-
1].first;
        if (b+1 < C) l[x][a] <?= max(r[x+1][a], A[b+1].second)+A[b+1].first-
A[a].first;
        if (a) r[x][a] <?= max(l[x+1][a-1], A[a-1].second)+A[b].first-A[a-
1].first;
        if (b+1 < C) r[x][a] <?= max(r[x+1][a], A[b+1].second)+A[b+1].first-
A[b].first;
    }

    int res = INF; REP(i, C) res <?= max(l[0][i], A[i].second)+abs(B-
A[i].first);
    fprintf(fout, "%d\n", res);
    return 0;
}
```

Md. Mahbubul Hasan:

It is quite obvious that if there is more than one classroom at a same place then we just need to keep track of the last class only. Now if we play with some sample then it would be understood that we have to move zigzagly between the two ends of the unvisited class position. That means say: we have class rooms at:

a_1, a_2, ..., a_n

then at a time we will cover some portion: (a_1 to a_i) from the beginning and some portion from end (a_j to a_n).

If we can convince our self with the truth of above fact then the rest will be upon implementation.

Here is a sample code:

```
#include<stdio.h>

#define min(A,B) ((A) < (B) ? (A) : (B))
#define max(A,B) ((A) > (B) ? (A) : (B))

struct TASK
{
        int time,dist;

        TASK(int a,int b) {time=a; dist=b;}
        TASK() {}
}task[2000];

int times[2000],dp[1002][1002][2];
int C,H,B;

int main()
{
        freopen("turnin.in","r",stdin);
        freopen("turnin.out","w",stdout);

        int i,len,x,y,a,b,j;
        int cnt,target;

        scanf("%d%d%d",&C,&H,&B);
        for(i=0;i<C;i++)
        {
                scanf("%d%d",&a,&b);
                if(times[a]<b) times[a]=b;
        }

        cnt=0;
        task[cnt++]=TASK(0,0);
        for(i=0;i<=H;i++)
        {
                if(i==B || times[i])
                {
                        if(i==B) target=cnt;
                        task[cnt++]=TASK(times[i],i);
                }
        }

        for(i=0;i<cnt;i++)
                 for(j=0;j<cnt;j++)
                         dp[i][j][0]=dp[i][j][1]=1000000000;

        dp[0][cnt-1][0]=0;
```

```
        for(len=cnt-1;len>0;len--)
                for(x=0;x+len<cnt;x++)
                {
                        y=x+len;


dp[x+1][y][0]=min(dp[x+1][y][0],max(task[x+1].time,dp[x][y][0]+task[x+1].dist
-
task[x].dist));

dp[x+1][y][1]=min(dp[x+1][y][1],max(task[y].time,dp[x][y][0]+task[y].dist-
task[x].dist));

dp[x][y-1][1]=min(dp[x][y-1][1],max(task[y-1].time,dp[x][y][1]+task[y].dist-
task[y-1].dist));

dp[x][y-1][0]=min(dp[x][y-1][0],max(task[x].time,dp[x][y][1]+task[y].dist-
task[x].dist));
                }

        printf("%d\n",dp[target][target][0]);

        return 0;
}
```

LARGE STATE DP

1. Last Minute Writing [Hal Burch, 2004] (19, 29) Camp 04 day 3

Bessie forgot that her essay on the advantages of Bermuda grass over Crab grass was due today. She now must do as well as she can as she writes the essay on her tractor ride to Moo U. Unfortunately, Bessie gets tractor sick when she tries to write. Bessie writes one word per second; help her determine the longest essay she can write.

```
The tractor ride to Moo U is S (1 <= S <= 1,000) seconds long. Each
second, Bessie can choose whether or not to write. If she writes,
her queasiness increases. If she does not write, her queasiness
decreases. At the beginning of the tractor ride, Bessie's queasiness
is 0 (it never drops below 0, of course). If her queasiness rises
above Q (1 <= Q <= 5,000), then Bessie becomes sick and tosses her
hay, ruining the paper on which she is writing her essay.

Her queasiness increases or decreases based on many variables: speed
of the tractor, acceleration of the tractor, and the smoothness of
the road. For second #i of her ride, you are given how her queasiness
will change if she writes (0 <= W_i <= Q) or if she does not write
(0 <= N_i <= Q).

PROBLEM NAME: essay

INPUT FORMAT:

* Line 1: Two space-separated integers: S and Q

* Lines 2..S+1: Line i+1 describes second #i with two space-separated
        integers: W_i and N_i

SAMPLE INPUT (file essay.in):

5 50
10 40
45 7
25 25
1 13
39 20

OUTPUT FORMAT:

* Line 1: A single integer, the maximum number of words Bessie can
        write.

SAMPLE OUTPUT (file essay.out):

3

OUTPUT DETAILS:

Bessie should write at time 1, rest at time 2, write at times 3 and 4, and
rest at time 5.
```

Analysis: Last Minute Writing by Richard Peng

The state is defined by the time and the amount of 'queasiness' Bessie has already accumulated, and we try to maximize the amount of the essay Bessie has written. At each time, Bessie has the choice of either to write or not to write, which is 2 transitions.

So there are a total of O(SQ) states, which is the runtime. We can reduce the memory usage by using a rotating array on the time to O(Q).

The following is a sample solution from Neal Wu:

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

FILE *fin = fopen ("essay.in", "r");
FILE *fout = fopen ("essay.out", "w");
const int MAXQ = 10005;
int S, Q, W, N;
int best [MAXQ], best2 [MAXQ];

inline void maxi (int &a, int b)
{
    if (b > a) a = b;
}

int main ()
{
    fscanf (fin, "%d %d", &S, &Q);

    while (S--)
    {
        fscanf (fin, "%d %d", &W, &N);

        memset (best2, 0, sizeof (best2));
        for (int i = 0; i <= Q; i++)
        {
            maxi (best2 [i + W], best [i] + 1);
            maxi (best2 [(i - N) >? 0], best [i]);
        }
        memcpy (best, best2, sizeof (best));
    }

    int ans = 0;
    for (int i = 0; i <= Q; i++)
        maxi (ans, best [i]);

    fprintf (fout, "%d\n", ans);

    return 0;
}
```

2. Bessie's Path [Mircea Pasoi, 2003]

Farmer John has announced he will present a lecture on Cowmputer Science in exactly K (1 <= K <= 600) minutes. Knowing there are N (1 <= N <= 5,000) farms conveniently numbered from 1..N in his county, FJ decided to present the lecture in farm N.

Bessie is at farm 1 and wants to see FJ's lecture. She is so meticulous that she wants her path to take exactly K minutes (so that she will reach FJ's lecture exactly on time).

The cows know all about the roads. The farms are connected by M (1 <= M <= 15,000) two-way roads, and between any two different farms with id's F1_i and F2_i, there is at most one road which Bessie might traverse. Each road requires precisely one minute to traverse, and each road i has its own traversal tax (0 <= T_i <= 32,000).

Help Bessie find the cheapest path (minimizing the sum of all the taxes; see below) from farm 1 to farm N in exactly K minutes. It is guaranteed that such a path always exists, and that Bessie has enough money to pay the crossing taxes.

PROBLEM NAME: path

INPUT FORMAT:

* Line 1: Three space-separated integers: N, M, and K

* Lines 2..M+1: Line i+1 describes road i with three space-separated integers: F1_i, F2_i, and T_i

SAMPLE INPUT (file path.in):

```
5 5 3
1 2 2
1 3 1
2 5 2
3 4 2
4 5 3
```

OUTPUT FORMAT:

* Line 1: The minimum tax that Bessie has to pay.

SAMPLE OUTPUT (file path.out):

```
6
```

OUTPUT DETAILS:

The path 1 -> 3 -> 4 -> 5 takes exactly 3 minutes to cross and its cost is the sum of taxes: 1 + 2 + 3 = 6. There is no other cheaper path.

Analysis: Bessie's Path by Neal Wu

This is a simple (and fairly well-known) DP problem, where the state consists of the best path to each node that has a certain length. We can update the lengths in our array by 1 in O(M) time, since we only need to consider traversing each edge, giving an overall O(MK) runtime.

The following is a sample solution. Note that it uses a sliding window to keep the memory down to O(M).

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

FILE *fin = fopen ("path.in", "r");
FILE *fout = fopen ("path.out", "w");

const int MAXN = 5005;
const int MAXM = 15005;

int N, M, K;
int edge1 [MAXM], edge2 [MAXM], cost [MAXM];
int best [MAXN], best2 [MAXN];

inline void mini (int &a, int b)
{
    if (b < a) a = b;
}

int main ()
{
    fscanf (fin, "%d %d %d", &N, &M, &K);

    for (int i = 0; i < M; i++)
    {
        fscanf (fin, "%d %d %d", edge1 + i, edge2 + i, cost + i);
        edge1 [i]--, edge2 [i]--;
    }


    memset (best, 63, sizeof (best));
    best [0] = 0;

    for (int t = 0; t < K; t++)
    {
        memset (best2, 63, sizeof (best2));

        for (int i = 0; i < M; i++)
        {
            mini (best2 [edge1 [i]], best [edge2 [i]] + cost [i]);
            mini (best2 [edge2 [i]], best [edge1 [i]] + cost [i]);
        }

        memcpy (best, best2, sizeof (best));
    }

    fprintf (fout, "%d\n", best [N - 1]);

    return 0;
}
```

3 . Inspecting the Farm [Vlad Novakovski, 2003] (28, 33) Camp 04 day2

The cows are in trouble. After a rowdy Hay Festival, they have wrought havoc on the paths that connect the N (2 <= N <= 2,500) pastures conveniently numbered 1..N on the farm, and Farmer John has set out to investigate. To make a somewhat thorough inspection of his farm, he must visit at least K (1 <= K <= 500) paths.

In order to make amends for their mischievous behavior, the cows wish to perform a favor for Farmer John. They want to plan his trip

subject to his constraints. They know that he will start his journey
from pasture 1 and end it at pasture N. They also have a list of
the the P (2 <= P <= 10,000) unidirectional inter-pasture paths,
each with starting pasture S_i and ending pasture E_i. The keenly
observant cows have even observed that there is not a single set
of directed paths on the farm that forms a cycle.

By way of example, suppose there are 4 pastures connected by paths
1->2, 2->3, 3->4, 1->3, and 1->4, and that Farmer John must travel
along at least K=2 paths when walking from pasture 1 to pasture 4.
In this case, the only two feasible routes are 1->3->4 and 1->2->3->4.

Help the cows determine the number of possible routes that Farmer
John can take.

PROBLEM NAME: inspect

INPUT FORMAT:

* Line 1: Three space-separated integers: N, K, and P

* Lines 2..P+1: Line i+1 contains two space-separated integers: S_i
  and E_i

SAMPLE INPUT (file inspect.in):

```
4 2 5
1 2
1 3
1 4
3 2
2 4
```

OUTPUT FORMAT:

* Line 1: A single integer, the number of possible journeys for Farmer
  John. The answer is guaranteed to fit into a 64-bit signed
  integer.

SAMPLE OUTPUT (file inspect.out):

```
2
```

Analysis: Inspecting the Farm by Richard Peng

As there are no cycles in the graph, we can perform a topological sort on the
the graph and use that as our DP ordering.

Then our state is location and number of paths used by FJ (with a flag to
indicate the case FJ used at least k paths). This is a total of O(NK) paths
and
our r state transition would be:

total[i,j]=sum(total[i1,j-1], there is a path from i1 to i)

So this runs in O(KP) time, which is sufficient.

Here is a sample solution from *Mahbub* :

```
/*
Prob ID: 0161
Name: inspect
Author: Md. Mahbubul Hasan
Date: 9th January, 2008.
LANG: C++
*/
#include<stdio.h>
#include<vector>
using namespace std;

vector<int> V[2502],TS;
int U[2502];
long long int way[2502][502];

void DFS(int at)
{
        U[at]=1;
        for(int i=V[at].size()-1;i>=0;i--)
                if(!U[V[at][i]])
                        DFS(V[at][i]);

        TS.push_back(at);
}

int main()
{
        freopen("inspect.in","r",stdin);
        freopen("inspect.out","w",stdout);

        int N,K,P;
        int i,j,sz,sz1,u,v,a,b,s;

        scanf("%d%d%d",&N,&K,&P);

        for(i=0;i<P;i++)
        {
                scanf("%d%d",&a,&b);
                V[a].push_back(b);
        }

        DFS(1);

        sz=TS.size();
        way[1][0]=1;

        for(i=sz-1;i>=0;i--)
        {
                if(TS[i]==N) break;

                u=TS[i];
```

```
                sz1=V[u].size();

                for(j=0;j<sz1;j++)
                {
                        v=V[u][j];
                        for(s=0;s<=K;s++)
                                way[v][s+1]+=way[u][s];
                        way[v][K]+=way[v][K+1];
                        way[v][K+1]=0;
                }
        }

        printf("%lld\n",way[N][K]);

        return 0;
}
```

4.Naptime [Tiankai Liu, 2004](29, 39) JAN05 Gold

Goneril is a very sleep-deprived cow. Her day is partitioned into N (3 <= N <= 3,830) equal time periods but she can spend only B (2 <= B < N) not necessarily contiguous periods in bed. Due to her bovine hormone levels, each period has its own utility U_i (0 <= U_i <= 200,000), which is the amount of rest derived from sleeping during that period. These utility values are fixed and are independent of what Goneril chooses to do, including when she decides to be in bed.

With the help of her alarm clock, she can choose exactly which periods to spend in bed and which periods to spend doing more critical items such as writing papers or watching baseball. However, she can only get in or out of bed on the boundaries of a period.

She wants to choose her sleeping periods to maximize the sum of the utilities over the periods during which she is in bed. Unfortunately, every time she climbs in bed, she has to spend the first period falling asleep and gets no sleep utility from that period.

The periods wrap around in a circle; if Goneril spends both periods N and 1 in bed, then she does get sleep utility out of period 1.

What is the maximum total sleep utility Goneril can achieve?

PROBLEM NAME: naptime
INPUT FORMAT:

* Line 1: Two space-separated integers: N and B
* Lines 2..N+1: Line i+1 contains a single integer: U_i

SAMPLE INPUT (file naptime.in):
```
5 3
2
0
3
1
4
```

INPUT DETAILS:
The day is divided into 5 periods, with utilities 2, 0, 3, 1, 4 in that order. Goneril must pick 3 periods.

OUTPUT FORMAT:

* Line 1: A single integer, the maximum total sleep utility Goneril can achieve.

SAMPLE OUTPUT (file naptime.out):

```
6
```

OUTPUT DETAILS:

Goneril can get total utility 6 by being in bed during periods 4, 5, and 1, with utilities 0 [getting to sleep], 4, and 2 respectively.

Analysis: Naptime by Richard Peng

First we break the problem into 2 cases to get rid of the cyclicness: sleeping during the first period and not sleeping.

Our DP state is [period,number of sleep periods,whether the cow is sleeping the previous period] and the transition between the cases can be broken down to casework:

We go from non-sleeping or sleeping to non-sleeping with added utility.

Non-sleeping to sleeping will cause the number of sleep periods to go up by 1, but has no added utility.

sleeping to sleeping will cause added utility total, but no changes otherwise.

So the transition is 2-by-2, which is best implemented by hand. Some care must be taken into making sure the last sleeping period and the 1st one match up to preserve the cyclicness.

Here is a solution from *Mahbub*:

```
/*
Prob ID: 0356
Name: naptime
Author: Md. Mahbubul Hasan
Date: 8th January, 2008.
LANG: C++
*/
#include<stdio.h>

inline int MAX(int &A,int B)
{
        if(B>A) A=B;
        return A;
}

int num[4000],n,dp1[2][3832][2],dp2[2][3832][2];

int main()
{
        freopen("naptime.in","r",stdin);
        freopen("naptime.out","w",stdout);

        int ans1,ans2,x,y,z,b,i,u,v;

        scanf("%d%d",&n,&b);
        for(i=0;i<n;i++) scanf("%d",&num[i]);

        u=0, v=1;
        for(y=0;y<=b;y++)
                for(z=0;z<=1;z++)
                {
                        if(!y) dp1[u][y][z]=0;
                        else dp1[u][y][z]=-1000000000;

                        if(z==1 && y==0) dp2[u][y][z]=0;
```

```
                        else dp2[u][y][z]=-1000000000;
                }


        for(x=n;x>=1;x--)
        {
                for(y=0;y<=b;y++)
                {
                        dp1[v][y][0]=dp1[v][y][1]=-1000000000;
                        dp2[v][y][0]=dp2[v][y][1]=-1000000000;

                        for(z=0;z<=1;z++)
                        {
                                MAX(dp1[v][y][z],dp1[u][y][0]);
                                if(y) MAX(dp1[v][y][z],dp1[u][y-1][1]+num[x-
1]*z);

                                MAX(dp2[v][y][z],dp2[u][y][0]);
                                if(y) MAX(dp2[v][y][z],dp2[u][y-1][1]+num[x-
1]*z);
                        }

                }

                u=1-u;
                v=1-v;
        }

        printf("%d\n",MAX(dp1[u][b][0],dp2[u][b][1]));

        return 0;
}
```

5.Palindromic Tax Paths [Alex Schwendner, 2002] (30, 50) Fall 2002 Green
The cows are bored. To amuse themselves, they have started searching for palindromes in Farmer John's tax forms. Palindromes are numbers that read the same forwards and backwards, e.g., 1234321 (but not 1231). A palindrome can be as short as 1 digit in length.

Each tax form is an N x N (2 <= N <= 20) table of random single digits (0..9). The cows wish to find paths on the grid such that

the sequence of digits the path traces is a palindrome. For example, in the table below, the path shown traces out the palindrome 122221:

```
1-2 0 0
   \
3 8 2=2
   /
3 1 8 9

3 4 5 4
```

Some of the other palindromes that exist in the above example are 000, 121, 131, 1331, 13331, 2002 (many different ways), 318813, 454, and 8338. The paths may loop back upon themselves (as is the case with, e.g., 121, 122221, and 000). However, the paths may not use the same table entry more than once sequentially (i.e., 9889, 95559, and 9999999 are not valid palindromes). The length of a palindromic path is the number of digits it uses (12221 uses 5 digits).

The cows would like to know how many palindromic paths of a given length L (1 <= L <= 18) exist. Write a program to determine the number of palindromic paths of length L for a given table of digits. If a path and its inverse are distinct, count them as two paths. That is, if traversing a path backwards is not the same as traversing it forwards, count it as two paths. In the example table above, count 122221 as two paths because it can be traversed either starting on the 1 in the first row and ending on the 1 in the third row, or starting on the 1 in the third row and ending on the 1 in the first row. On the other hand, count `949' only once, as it can only be traversed starting at 9, going to 4, and returning to the single 9.

PROBLEM NAME: palpath

INPUT FORMAT:

* Line 1: Two space-separated integers: N and L
* Lines 2..N+1: The tax form: N lines of N space-separated digits

SAMPLE INPUT (file palpath.in):

```
3 3
1 2 3
1 2 3
1 2 3
```

OUTPUT FORMAT:

* Line 1: A single integer: the total number of palindromic paths. All answers for this problem's test data fit into signed 32 bit entities.

SAMPLE OUTPUT (file palpath.out):

```
86
```

Analysis: Palindromic Tax Paths by Richard Peng

We can think of constructing the palindromes from its inner most pairs outward:
at each step, we pick two grids with the same letter that's adjacent to the current endpoints of the path and add them to the path to form a path of length

2 greater.

Since the path can overlap back onto itself, the only 'state' that matters is the locations of the endpoints of the path, which gives O(n^4) possibilities. Along with L, this gives an algorithm that uses O(LN^4) states and runs in O(LN^4) time as there are a total of 8*8=64 possible state transitions.

Care must be taken into processing the odd and even cases separately as they have different inner most pairs.
Here is a sample solution from *Mahbub*:
(Note that we could improve the solution by just considering 1-3-5-.. and 2-4-
6-... for odd and even case seperately. Instead of that here I have just evaluated for all possible length.)

```
/*
Prob ID: 0819
Name: palpath
Author: Md. Mahbubul Hasan
Date: 10th January, 2008.
LANG: C++
*/
#include<stdio.h>

int deg[410],adj[410][10],num[410];
long long dp[401][401][3],ans;

int main()
{
        freopen("palpath.in","r",stdin);
        freopen("palpath.out","w",stdout);

        int N,L,i,j,x,y,s,t,u,len;
        int a,b,c;

        scanf("%d%d",&N,&L);
        for(i=0;i<N;i++) for(j=0;j<N;j++)
        {
                scanf("%d",&num[i*N+j]);
                u=i*N+j;
                deg[u]=0;
                for(x=-1;x<=1;x++)
                        for(y=-1;y<=1;y++)
                                if(!(x==0 && y==0))
                                {
                                        s=i+x;
                                        t=j+y;
                                        if(s>=0 && s<N && t>=0 &&
t<N)
                                                adj[u][deg[u]++]=s*N+t;
                                }

                dp[u][u][1]=1;
        }
```

```
        N*=N;

        for(i=0;i<N;i++)
                for(j=0;j<deg[i];j++)
                        dp[i][adj[i][j]][2]=(num[i]==num[adj[i][j]]);

        a=1; b=2; c=0;
        for(len=3;len<=L;len++)
        {
                for(i=0;i<N;i++) for(j=0;j<N;j++) dp[i][j][c]=0;

                for(i=0;i<N;i++)
                        for(j=0;j<N;j++) if(dp[i][j][a])
                                for(x=0;x<deg[i];x++)
                                        for(y=0;y<deg[j];y++)

if(num[adj[i][x]]==num[adj[j][y]])

dp[adj[i][x]][adj[j][y]][c]+=dp[i][j][a];
                a=(a+1)%3;
                b=(b+1)%3;
                c=(c+1)%3;
        }

        if(L==1) b=1;
        for(ans=i=0;i<N;i++)
                for(j=0;j<N;j++)
                        ans+=dp[i][j][b];

        printf("%lld\n",ans);

        return 0;
}
```

6. Traffic Lights [J. Kuipers, 2002] (30, 39) FEB 03 Green

The cows are going downtown! Just like everyone else, they want to optimize their driving time.

They have noted that when driving on a straight road with traffic

lights, the best strategy to get to their destination as quickly
as possible is not necessarily to drive as fast as possible to the
next traffic light, brake if it's red, wait for a green light,
accelerate, and then drive on. It is often better to approach a
traffic light more slowly in order to have some speed when the light
turns green.

The cows have observed the traffic lights for a very long time.
They know that each traffic light behaves in the following way:
  * it is green for a certain amount of time Tg_i,
  * then it is red for an amount of time Tr_i,
  * then green again,
  * and so on.

Given
  * the integer length of the road L (1 <= L <= 100)
  * the number of traffic lights N (0 <= N <= L+1) along with
    information about each light:
     * P_i: the unique position (0 <= P_i <= L)
     * Tg_i (1 <= Tg_i <= 10)
     * Tr_i (1 <= Tr_i <= 10)
     * c_i: color at t=0 (R or G)
     * Tc_i (the integer amount of time since the light last changed)

write a program to determine the minimal amount of time needed to
get to the end of the road. Note that at each discrete time (starting
at t=0), a car may either change its speed (expressed in positional
units per time unit) by one or keep it constant. The speed is always
0 or positive, of course. No driving backwards!

The car starts at position zero has has speed zero. The car must
complete its trip at position L, also with speed zero. The car must
stop at all red lights -- be sure its speed is 0 at the red light's
position if it encounters a red light. The car may move when the
light changes from red to green, but not when it changes from green
to red.

PROBLEM NAME: traffic
INPUT FORMAT:
* Line 1: Two space-separated integers: L and N
* Lines 2..N+1: Line i+1 describes traffic light i with five
        space-separated entities (all are integers except #4 which is
        a single character 'R' or 'G'): P_i, Tg_i, Tr_i, c_i, and Tc_i

SAMPLE INPUT (file traffic.in):
4 1
1 10 10 R 0
OUTPUT FORMAT:
A single line with a single integer that is the minimal time needed.

SAMPLE OUTPUT (file traffic.out):
12

Analysis: Traffic Lights by Richard Peng

Since there is very few things we can fall back onto, we use the location,

speed and time of the car as the state in our DP (note the car can take at most
1000 turns to get there as only 10 seconds needs to be spent waiting at each intersection and the car's speed can never go above 9).

Our state transition consists of checking all speed changes and whether the car
runs into any red lights. The red light conditions are fairly tricky, so care must be taken into implementing it. The algorithm runs in O(L*maxV*time), which
is good enough.
Here is a solution from *Tomek*

```
#include <cstdio>
#include <stdio.h>
using namespace std;

#define FOR(i,a,b) for(int i=(a);i<=(b);++i)

const int MAXL = 100; // max length
const int MAXS = 20; // max speed

struct Light {
  int tg,tr;
  int start;
  Light() { tg=1; tr=0; start=0; }
  Light(int g,int r,char c,int tc) {
    tg=g; tr=r;
    start = c=='G'? tc : tc+tg;
  }
  // is it green starting at time t?
  bool green(int t) {
    return (t+start)%(tg+tr) < tg;
  }
};

int ll;
bool ok[MAXL+1][MAXS+1];
Light lights[MAXL+1];

void read() {
  int n;
  FILE *IN = fopen("traffic.in", "r");
  fscanf(IN, "%d%d",&ll,&n);
  for(int i=0;i<=ll;++i) lights[i]=Light();
  FOR(i,1,n) {
    int p,tg,tr,tc;
    char c;
    fscanf(IN, "%d%d%d %c%d",&p,&tg,&tr,&c,&tc);
    lights[p] = Light(tg,tr,c,tc);
  }
}

void init() {
  FOR(x,0,ll) FOR(s,0,MAXS) ok[x][s]=false;
  ok[0][0]=true;
}
```

```
void step(int tm) {
  static int canGo[MAXL+1]; // how much forward can we go
  static bool ok2[MAXL+1][MAXS+1];

  canGo[ll]=0;
  for(int x=ll-1;x>=0;--x)
    if(lights[x].green(tm))
      canGo[x] = canGo[x+1]+1;
    else
      canGo[x] = 0;

  FOR(x,0,ll) FOR(s,0,MAXS) ok2[x][s]=false;
  FOR(x,0,ll) FOR(s,0,MAXS) if(ok[x][s])
    FOR(s2,s-1,s+1) {
      if(s2>=0 && s2<=MAXS && x+s2<=ll && canGo[x]>=s2)
        ok2[x+s2][s2] = true;
    }
  FOR(x,0,ll) FOR(s,0,MAXS) ok[x][s] = ok2[x][s];
}

int main() {
  FILE *OUT = fopen("traffic.out", "w");
  read();
  init();
  int tm=0;
  while(!ok[ll][0]) {
    step(tm++);
  }
  fprintf(OUT, "%d\n",tm);
  fclose(OUT);
  exit (0);
}
```

The cows have decided to publish a scientific journal complete with P (1 <= P <= 70) paragraphs and F (1 <= F <= P) numbered figures with figure numbers FN_i conveniently numbered 1..F. A paragraph references no more than one figure; figures are referenced precisely once.

In order to be printed, the figures require a certain length in deci-lines (1 <= FL_j <= 100); so do the paragraphs (1 <= PL_i <= 100). No paragraph or figure is so long that it cannot fit on a single page. Some (perhaps all, perhaps not) paragraphs reference some figure by its number (no figure is references more than once; figures might be referenced in any order).

The best journals require that each figure and each paragraph must fit entirely on one page whose length is L deci-lines (1 <= L <= 100). Thus, they do not span from the bottom of one page to the top of the next.

Both the paragraphs and figures must run through the paper in the order presented in the input. The figures can be interspersed among the paragraphs anywhere as long as they are no further than one page away from their referencing paragraph (i.e., on the page before, the page of, or the page after the referencing paragraph).

Find an optimal ordering that minimize deci-lines and hence paper. There will always be some ordering that will enable proper printing of the journal.

PROBLEM NAME: journal

INPUT FORMAT:

* Line 1: Three space-separated integers: F, P, and L

* Lines 2..P+1: Line i+1 describes paragraph i with two space-separated integers: PL_i and FN_i. FN_i is 0 if no figure is referenced by paragraph i.

* Lines P+2..P+F+1: Line P+1+j contains a single integer: FL_j

SAMPLE INPUT (file journal.in):

```
2 4 20
10 1
7 0
9 2
3 0
12
11
```

INPUT DETAILS:

The first paragraph has length 10 and references figure 1 whose length is 12. The second paragraph has length 7 and references no figures. The third paragraph has length 9 and references figure 2 whose length is 11. The fourth paragraph has length 3 and references no figures.

OUTPUT FORMAT:

* Line 1: A single line with two space-separated integers: the total
  number of complete and partial pages used and the number of
  decilines used on the last page for the optimal layout.

SAMPLE OUTPUT (file journal.out):

```
4 3
```

OUTPUT DETAILS:

```
One optimal arrangement:
   Page 1: Para 1, Para 2
   Page 2: Fig 1
   Page 3: Para 3, Fig 2
   Page 4: Para 4
```

Analysis: The Bovine Journal by Md. Mahbubul Hasan

We can solve it in O(N^2) memory limit with time limit still O(N^4). For this we will change the problem a bit, we have N paragraph and N figures, each paragraph having a figure. If a paragraph originally does not have any figure then it is 0 in cost. Now we denote each state by dp[a][b] which means we have
completed a paragraphs and b figures. And from it we can go to any other N^2 possible state. So total runtime O(N^4) with memory O(N^2). Below is the sample solution implementing this idea:

```
/*
Prob ID: 0829
Name: journal
Author: Md. Mahbubul Hasan
Date: 10th January, 2008.
LANG: C++
*/
#include<stdio.h>
#include<utility>
using namespace std;

#define INF 120
#define MAX(A,B) ((A) > (B) ? (A) : (B))
typedef pair<int,int> PII;


PII dp[72][72];
int sum0[72],sum1[72],temp0[72],temp1[72],temp2[72];
int i,j,x,y,need,F,P,L,t;

int main()
{
        freopen("journal.in","r",stdin);
        freopen("journal.out","w",stdout);

        scanf("%d%d%d",&F,&P,&L);
        for(i=0;i<P;i++) scanf("%d%d",&temp0[i],&temp1[i]);
        for(i=1;i<=F;i++) scanf("%d",&temp2[i]);

        for(i=1;i<=P;i++) sum0[i]=sum0[i-1]+temp0[i-1], sum1[i]=sum1[i-
1]+temp2[temp1[i-1]];
        for(i=0;i<=P;i++) for(j=0;j<=P;j++) dp[i][j]=PII(INF,INF);
        dp[0][0]=PII(0,0);

        for(i=0;i<=P;i++) for(j=0;j<=P;j++)
        {
                for(t=MAX(i,j),x=t;x<=P;x++) for(y=t;y<=P;y++)
                        if( (need=sum0[x]-sum0[i] + sum1[y]-sum1[j]) <= L
&& dp[x][y] > PII(dp[i][j].first+1,need))
                                dp[x][y] = PII(dp[i][j].first+1,need);
        }

        printf("%d %d\n",dp[P][P].first,dp[P][P].second);

        return 0;
}
```

8. TwoFour [Romanian Olympiad, via Stroe, 2002] (36, 39) MAR03 Green

Bessie has a new two-person game named TwoFour. To play it, she has N (3 <= N <= 30) piles, each with a number of balls (0 <= nballs <= 4). The total number of balls is 2*N.

The players alternate taking turns in the game; player 1 makes the first move. Each turn consists of a single valid move. A valid move consists of the following two actions:

* First, the player chooses two different piles.

* Second, she takes a single ball from one pile and moves it to the other pile. She can do this only if the number of balls in the second pile (including the new ball) is not greater than the number of balls remaining in the first pile after the ball is removed.

The game ends when no more moves can be made. In fact, at the end of the game, every pile will contain exactly two balls.

The winner of the game is the player who 'owns' most piles at the end of the game. Ties are possible. A player 'owns' a pile if the pile has two balls and this size resulted from the particular player's most recent move to or from that pile. Consider these examples:

* If a player moves a ball from a pile with four balls to a pile with one ball, then she owns the second pile (with two balls).

* If a player moves a ball from a pile with three balls to a pile with no balls, then she owns the first pile, now that it has two balls.

* If a player moves a ball from a pile with three balls to a pile with one ball, then she owns both piles (both with two balls).

Ownership can change. Consider a pile of two balls owned by one player. If the other player chooses a pile with four balls and moves a ball to the pile with two, then neither pile is owned by anyone.

If, at the beginning of the game, some piles have two balls, then the piles are equally distributed among the two players with any extra pile being owned by player two.

Your program must decide, for an initial game state, who will be the winner or if the game ends in a tie. Of course, the decision is based on optimal play by both players. Your program will be presented with G (1 <= G <= 1000) game states to 'solve'.

Use no more than 1.000 Megabytes of memory for this problem.

PROBLEM NAME: twofour

INPUT FORMAT:

* Line 1: Two space-separated integers: N and G.

* Lines 2..G+1: Each line contains N space-separated integers
  describing a game. The first integer is the number of balls
  in pile 1, the second integer is the number of balls in pile
  2, and so on. Line 2 describes game 1, line 3 describes game
  2, and so on. Your program should compute the winner for each
  game.

SAMPLE INPUT (file twofour.in):

```
5 4
0 3 4 1 2
2 2 2 2 2
1 1 2 2 4
4 3 2 1 0
```

OUTPUT FORMAT:

* Lines 1..G: The output contains the outcome of each game. Line 1
  gives the outcome of game 1, and so on. The outcome is a
  single integer: 1 if the first player wins, 2 if the second
  player wins, and 0 if the game is a tie.

SAMPLE OUTPUT (file twofour.out):

```
1
2
1
1
```

Analysis: TwoFour by Bruce Merry/Richard Peng/Md. Mahbubul Hasan

---

The problem is a straightforward minimax, the trick being to squeeze everything into memory. Each heap is clearly defined by 6 numbers: number of 0, number of 1, number of 3, number of 4, number of 2 owned by A and number of 2 owned by B;

Right off the bat this looks like 30^6. However one can eliminate two degrees of freedom since the total number of heaps and the total number of pieces are known, giving 30^4 = 810000 states.

Since the sum of squares of the numbers are decreasing as the games proceed, there cannot be a cycle in the states, therefore min-max DP on the states tracking a player's best score would work. At each step, a player picks a move which minimizes the opponents max score on the board left to him.

Here is the solution from Bruce Merry:

```
/* TwoFour: a USACO problem (1000K memory limit)
 * Solution by Bruce Merry
 */

/* The problem is really a standard minimax problem. The key is to reduce the
 * number of states to consider. Firstly the order of the heaps is
irrelevant,
 * so one ownly needs to consider the number of heaps containing 0, 1, 3, 4,
 * the number containing 2 owned by the current player (labelled 2A), and the
 * number containing 2 owned by the other player (labelled 2B). Secondly one
 * can note that there are in fact only 4 degrees of freedom since the total
 * number of pieces is 2N and the number of heaps is N, so only 4 of these
 * 6 need to be stored (as long as at least of of 2A and 2B are stored).
Finally
 * one can observe that there can be at most N/2 0's and N/4 4's, in order
for
 * the sum to be 2N. Hence indexing by 0's, 2A's, 2B's and 4's gives an array
 * of size (N+1)^2*(N/2+1)^2 = 246016, easily inside the memory limit.
 *
 * The game tree is acyclic and fairly shallow (less than 60 deep), so a
 * depth first approach with caching (memoizing) is sufficient. Since the
 * cache can be reused across lines of input, the number of lines of
 * input is not a factor in terms of speed.
 */

#include
#include
#include
#include

int N;
```

```
FILE *in, *out;
int lines;

/* indices are 0's, 2A's, 2B's, 4's. There can be at most 15 0's and 15 4's
to
get the
 * sum to be 2N. The 1's and 3's can be computed from the sum and the count.
Always
 * works from the point of view of the player who's turn it is to play. The
values
 * are -1=lose, 0=draw, 1=win, 2=not yet computed.
 */
signed char answers[16][31][31][16];

/* indices are          0, 1, 2A, 2B, 3, 4 */
const int downmap[6] = {-1, 0, 1, 1, 2, 4};   /* for removing one from a pile
*/
const int upmap[6] =    {1, 2, 4, 4, 5, -1};  /* for inserting one into a
pile
*/
const int valuemap[6] = {0, 1, 2, 2, 3, 4};   /* the number of pieces in a
pile */

/* internally -1=lose, 0=tie, 1=win; this maps back to the external system */
const int answermap[3] = {2, 0, 1};

void init() {
  int i;

  in = fopen("twofour.in", "r");
  assert(in);
  out = fopen("twofour.out", "w");
  assert(out);

  fscanf(in, "%d %d", &N, &lines);
  memset(answers, 2, sizeof(answers));  /* indicates dont-know */

  for (i = 0; i <= N; i++)
    answers[0][i][N - i][0] = (i == N - i) ? 0 : (i > N - i) ? 1 : -1;
}

int get(int a, int b, int c1, int c2, int d, int e) {
  int counts[6], next[6];
  int i, j, best, cur;

  if (answers[a][c1][c2][e] != 2) return answers[a][c1][c2][e];

  counts[0] = a;
  counts[1] = b;
  counts[2] = c1;
  counts[3] = c2;
  counts[4] = d;
  counts[5] = e;
  best = -2;
  for (i = 2; i < 6; i++)
    if (counts[i])
      for (j = 0; valuemap[j] <= valuemap[i] - 2; j++)
```

```
        if (counts[j]) {
          memcpy(next, counts, sizeof(next));
          next[i]--;
          next[downmap[i]]++;
          next[j]--;
          next[upmap[j]]++;
          cur = -get(next[0], next[1], next[3], next[2], next[4], next[5]);
          if (cur > best) best = cur;
        }
  assert(best != -2);
  answers[a][c1][c2][e] = best;
  return best;
}

int nextline() {
  int counts[5];
  int cur;
  int i;

  memset(counts, 0, sizeof(counts));
  for (i = 0; i < N; i++) {
    if (fscanf(in, "%d", &cur) < 1) return 0;
    counts[cur]++;
  }
  fprintf(out, "%d\n", answermap[get(counts[0], counts[1], counts[2] / 2,
counts[2] - counts[2] / 2, counts[3], counts[4]) + 1]);
  return 1;
}

int main() {
  init();
  while (nextline());
  fclose(in);
  fclose(out);
  return 0;
}
```

9. Telephone Wire [Jeffrey Wang, 2007] (36, 39) NOV 07 Gold

Farmer John's cows are getting restless about their poor telephone service; they want FJ to replace the old telephone wire with new, more efficient wire. The new wiring will utilize N (2 <= N <= 100,000) already-installed telephone poles, each with some height_i meters (1 <= height_i <= 100). The new wire will connect the tops of each pair of adjacent poles and will incur a penalty cost C * the two poles' height difference for each section of wire where the poles are of different heights (1 <= C <= 100). The poles, of course, are in a certain sequence and can not be moved.

Farmer John figures that if he makes some poles taller he can reduce his penalties, though with some other additional cost. He can add an integer X number of meters to a pole at a cost of X^2.

Help Farmer John determine the cheapest combination of growing pole heights and connecting wire so that the cows can get their new and improved service.

PROBLEM NAME: telewire

INPUT FORMAT:

* Line 1: Two space-separated integers: N and C

* Lines 2..N+1: Line i+1 contains a single integer: height_i

SAMPLE INPUT (file telewire.in):

```
5 2
2
3
5
1
4
```

INPUT DETAILS:

There are 5 telephone poles, and the vertical distance penalty is $2/meter. The poles initially have heights of 2, 3, 5, 1, and 4, respectively.

OUTPUT FORMAT:

* Line 1: The minimum total amount of money that it will cost Farmer John to attach the new telephone wire.

SAMPLE OUTPUT (file telewire.out):

```
15
```

OUTPUT DETAILS:

```
The best way is for Farmer John to raise the first pole by 1 unit and the
fourth pole by 2 units, making the heights (in order) 3, 3, 5, 3, and 4.
This costs $5. The remaining wiring will cost $2*(0+2+2+1) = $10, for a
total of $15.
```

Analysis: Telephone Wire by Spencer Liang & Brian Dean

Let H[i] be the original height of the i-th pole, and let f(n, h) be the minimum cost for n poles with the n-th pole having height h. Then:

```
f(n, h) = min { (H[i]-h)^2 + f(n-1, h') + C|h'-h| } for all h'
```

With a straightforward dynamic programming implementation, this runs in $O(N*H^2)$, where H is the maximum height. However, by splitting up the recurrence relation into two cases, one where h' >= h and one where h' < h, we can rewrite it as:

```
f(n, h) = (H[i]-h)^2 + min { -C*h + min { f(n-1, h')+C*h' } (for h' >= h),
                              C*h + min { f(n-1, h')-C*h' } (for h' < h) }

Define low(n, h)  := min over h' >= h { f(n, h')+C*h' }
   and high(n, h) := min over h' < h  { f(n, h')-C*h' }

Then f(n, h) = (H[i]-h)^2 + min { -C*h+low(n-1, h), C*h+high(n-1, h) }
```

low(n, h) and high(n, h) for all n, h can be computed in O(N*H) time; thus f(n, h) can be computed in O(N*H) time as well. A final implementation detail: an array of size O(N*H) exceeds the the memory limit, but only two "rows" of the DP table are needed at a time, so an array of size 2*H is sufficient.

Below is Richard Peng's solution:

```
#include <cstdio>

#define MAXN 110000
#define MAXH 101

int h[MAXN], bes[2][MAXH], ans, huge, bes1, c, n;

inline int sqr (int x){return x*x; }

int main (){
    int i, j, pre, cur;
    freopen ("telewire.in", "r", stdin);
    freopen ("telewire.out", "w", stdout);
    huge = 2100000000;
    scanf ("%d%d", &n, &c);
    for (i = 0; i<n; i++)    scanf ("%d\n", &h[i]);
    for (i = 0; i<MAXH; i++)
        bes[0][i] = (i> = h[0]) ? sqr(h[0]-i) : huge;
```

```
    for (i = 1; i<n; i++){
        pre = (i+1)%2;
        cur = i%2;
        for (bes1 = huge, j = 0; j<MAXH; j++){
            bes1 <?= bes[pre][j]-j*c;
            bes[cur][j] = bes1+j*c;
        }
        for (bes1 = huge, j = MAXH-1; j> = 0; j--){
            bes1 <?= bes[pre][j]+j*c;
            bes[cur][j] <?= bes1-j*c;
        }
        for (j = 0; j<MAXH; j++)
            bes[cur][j] = (j> = h[i])? (bes[cur][j] + sqr(j-h[i])) : huge;
    }
    ans = huge;
    for (i = 0; i<MAXH; i++) ans<? = bes[cur][i];
    printf ("%d\n", ans);
    return 0;
}
```

10. Zoo [APIO, 2007] (37, 47)

The pride of the Asia-Pacific region is the newly constructed Great Circular Zoo. Situated on a small Pacific island, it comprises a large circle of different enclosures, each containing its own exotic animal as illustrated below.

You are in charge of public relations for the zoo, which means it is your job to keep people as happy as possible. A bus load of schoolchildren has just arrived, and you are eager to please them. There are animals that some children love, and there are animals that some children fear. For example, little Alex loves monkeys and koalas because they are cute, but fears lions because of their sharp teeth. On the other hand, Polly loves lions because of their beautiful manes, but fears koalas because they are extremely smelly.

You have the option of removing some animals from their enclosures, so that children are not afraid. However, you are worried that if you remove too many animals then this will leave the children with nothing to look at. Your task is to decide which animals to remove so that as many children can be made happy as possible.

Each child is standing outside the circle, where they can see five consecutive enclosures. You have obtained a list of which animals each child fears, and which animals each child loves. A child will be made happy if

* At least one animal they fear is removed from their field of vision

or

* At least one animal they love is not removed from their field of vision.

For example, consider the list of children and animals illustrated below:

Child	Visible Enclosures	Fears	Loves
Alex	2, 3, 4, 5, 6	Enclosure 4	Enclosures 2, 6
Polly	3, 4, 5, 6, 7	Enclosure 6	Enclosure 4
Chaitanya	6, 7, 8, 9, 10	Enclosure 9	Enclosures 6, 8
Hwan	8, 9, 10, 11, 12	Enclosure 9	Enclosure 12
Ka-Shu	12, 13, 14, 1, 2	Enclosures 12, 13, 2	

Suppose you remove the animals from enclosures 4 and 12. This will make Alex and Ka-Shu happy, because at least one animal that they fear has gone. This will also keep Chaitanya happy, since both enclosures 6 and 8 still contain animals that he loves. However, both Polly and Hwan will be unhappy, since they cannot see any animals that they love but they can still see all the animals that they fear. This arrangement therefore gives a total of three happy children.

Now suppose you put these animals back into their enclosures, and instead remove the animals from enclosures 4 and 6. Alex and Polly will be happy because the animals that they fear in enclosures 4 and 6 have gone. Chaitanya will be happy because, even though animal 6 has gone, he can still see the animal in enclosure 8 which he loves. Likewise, Hwan will be happy because she can now see the animal in enclosure 12, which she loves. The only person unhappy will be Ka-Shu.

Finally, suppose you put the animals back once more and then remove only the animal from enclosure 13. Ka-Shu will now be happy since one animal that he fears has been removed, and Alex, Polly, Chaitanya and Hwan will all be happy since they can all see at least one animal that they love. Thus this arrangement gives five happy children, the largest number possible.

The first line of input will be of the form "N C", where N is the number of animal enclosures (10 <= N <= 10,000) and C is the number of children (1 <= C <= 50,000). The enclosures are numbered 1, 2, ..., N clockwise around the circle.

Following this will be C additional lines of input, where each line describes a single child. Each of these lines will be of the form "E F L X1 X2 ... XF Y1 Y2 ... YL" where:

* E is the first enclosure that the child can see (1 <= E <= N). In other words, the child can see enclosures E, E + 1, E + 2, E + 3 and E + 4. Note that numbers larger than N wrap back around the circle, so if N = 14 and E = 13 then the child can see enclosures 13, 14, 1, 2 and 3.

* F is the number of animals that the child fears, and L is the number of animals that the child loves.

* Enclosures X1, ..., XF contain the animals that the child fears (1 <= X1, ..., XF <= N).

* Enclosures Y1, ..., YL contain the animals that the child loves (1 <= Y1, ..., YL <= N).

* No two of the integers X1, ..., XF ; Y1, ..., YL are equal, and all of these integers describe enclosures that the child can see. Children will be listed in sorted order according to the first enclosure E (so the child with lowest E will appear first and the child with largest E will appear last). Note that more than one child may have the same first enclosure E.

PROBLEM NAME: zoo

INPUT FORMAT:

* Line 1: Two space-separated integers: N and C

* Lines 2..C+1: Line i+1 contains the following information for child i: Three space-separated integers: The first enclosure the child can see, and the number of enclosures the child fears

and loves, respectively. After this are the enclosures the child fears (space-separated) and then the enclosures the child loves (space-separated).

SAMPLE INPUT (file zoo.in):

```
14 5
2 1 2 4 2 6
3 1 1 6 4
6 1 2 9 6 8
8 1 1 9 12
12 3 0 12 13 2
```

OUTPUT FORMAT:

* Line 1: A single integer giving the largest number of children that can be made happy.

SAMPLE OUTPUT (file zoo.out):

```
5
```

OUTPUT DETAILS:

The sample case is the example discussed earlier, in which all C = 5 children can be made happy.

Analysis: Zoo by Weidong Hu, Md. Mahbubul Hasan

This is a special case of Max-SAT, where each clause can only contain variables which are close together (in a circular order). Specifically:

• Each animal is a boolean variable in Max-SAT. We have N variables X1 , ... , XN in a circular order.

• Each child is a clause in Max-SAT. If a child can see k consecutive enclosures, this means that each clause may only contain variables Xc , Xc+1 , ... , Xc+k-1 for some c (where XN +1 = X1, XN +2 = X2 and so on). For this problem, we are given k = 5.

A model solution uses dynamic programming.

Linear case:

Consider a simpler problem when the enclosures X1 , ... , XN are placed in a linear order (i.e., XN does not wrap back around to X1). For each i, let Ci be the set of all children whose visible enclosures lie in the range X1 , ... , Xi . We iterate through i = 1, 2, ... , N , and at each stage we solve problems of the following form:

Consider the set of children Ci , and consider all 2k ways in which the final k enclosures Xi-k+1 , Xi-k+2 , ... , Xi may be filled or empty. For each filled/empty combination for enclosures Xi-k+1 , Xi-k+2 , ... , Xi , what is the maximum number of children in Ci who can be made happy?

We solve the problems at stage i as follows. We assume we have already computed the maximum number of happy children in Ci-1 for all filled/empty combinations for enclosures Xi-k , ... , Xi-1. To calculate the new answers for stage i and a particular filled/empty combination for enclosures Xi-k+1 , ;k+2 , ... , Xi :

(i) We count how many “new” children in Ci, Ci+1 are made happy. We can calculate this directly from our choice of filled/empty values for Xi-k+1 , Xi-k+2 , ... , Xi , because every “new” child is looking at precisely these enclosures.

(ii) Assuming the “old” enclosure Xi-k is filled, we can look up our solutions from stage i-1 to find out how many children in Ci-1 can be made happy with our selected filled/empty values for Xi-k+1 , ... , Xi-1 . Likewise,we can look up how many children in Ci-1 can be made happy when the old enclosure Xi-k is empty. The larger of these two numbers tells us how many 'old' children in Ci-1 we can keep happy overall.

(iii) The number of happy “new” children from step (i) can be added to the number of happy “old” children from step (ii), giving the solution to our current problem.

The total number of problems to solve is 2^kN , and the overall running time for the algorithm is O(2k (N + C)). The space required is only O(2^k), since each stage i only requires the solutions from the previous stage i − 1.

Circular case:

In the circular case, we need to permanently remember the states of the first k-1 enclosures X1 , . . . , Xk-1 (in additional to the “most recent” k enclosures Xi-k+1 , . . . , Xi). This allows us to correctly deal with the final children who can see back around past XN to X1 , ... , Xk-1 again. The remainder of the algorithm remains more or less the same.

We therefore have 2^(2k) problems to solve at each stage, giving running time O(2^(2k) (N + C)) and storage space O(2^(2k)).

The algorithm can be further optimized using bitmasks to reduce the 2^k work into a single integer operation.

Here is the solution by *Md, Mahbubul Hasan*

```
/*
Prob ID: 0777
Name: zoo
Author: Md. Mahbubul Hasan
Date: 2nd January, 2008
LANG: C++
*/
#include<stdio.h>
#include<vector>
#include<utility>
using namespace std;

typedef pair<int,int> PII;
vector<PII> V[10001];
int dp[2][100];

void MAX(int &A,int B) {if(B > A) A=B;}
```

```
int main()
{
        freopen("zoo.in","r",stdin);
        freopen("zoo.out","w",stdout);

        int i,u,v,j,x,y,t,ans=0,now,temp,z,a,at,cnt,sz,s;
        int N,C,e,l,f;

        scanf("%d%d",&N,&C);

        for(i=0;i<C;i++)
        {
                scanf("%d",&e);
                scanf("%d%d",&f,&l);
                x=0;
                for(j=0;j<f;j++) {scanf("%d",&a); x|=1<<((a-e+N)%N);}
                y=0;
                for(j=0;j<l;j++) {scanf("%d",&a); y|=1<<((a-e+N)%N);}
                V[e-1].push_back(PII(y,x));
        }

        u=0;
        v=1;
        for(t=0;t<16;t++)
        {
                for(at=0;at<N;at++)
                {
                        for(z=0;z<32;z++) dp[v][z]=0;

                        if(at==0)
                        {
                                for(x=0;x<=1;x++)
                                {
                                        now=t|(x<<4);
                                        cnt=0;
                                        sz=V[0].size();
                                        for(i=0;i<sz;i++)
                                                cnt+=( (now & V[at][i].first) |
| ((now & V[at][i].second)!=V[at][i].second));
                                        dp[v][now]=cnt;
                                        MAX(ans,cnt);
                                }

                                u=1-u;
                                v=1-v;

                                continue;
                        }

                        for(s=0;s<32;s++)
                        {
                                temp=s>>1;

                                if(N-at>=5)
                                {
                                        for(x=0;x<=1;x++)
```

```
                                        {
                                                now=temp | (x<<4);
                                                cnt=0;
                                                sz=V[at].size();
                                                for(i=0;i<sz;i++)
                                                        cnt+=( (now &
V[at][i].first) || ((now & V[at][i].second)!=V[at][i].second));
                                                MAX(dp[v][now],dp[u][s]+cnt);
                                        }
                                }
                                else
                                {
                                        x=(((1<<(4-N+at)) & t) > 0);

                                        now=temp | (x<<4);
                                        cnt=0;
                                        sz=V[at].size();
                                        for(i=0;i<sz;i++)
                                                cnt+=( (now & V[at][i].first) |
| ((now & V[at][i].second)!=V[at][i].second));
                                        MAX(dp[v][now],dp[u][s]+cnt);
                                }
                        }

                        for(s=0;s<32;s++) MAX(ans,dp[v][s]);

                        u=1^u;
                        v=1^v;
                }
        }

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

        return 0;
}
```

11. Crisis on the Farm [Thomas Williamson, 2004] (37, 35) Open 08 Gold

Farmer John and his herd of exotic dancing bovines have been practicing for his new moosical, "The Street Cow Named Desire". At one point in the middle of rehearsal, his cows are stacked on top of each other in N (1 <= N <= 1,000) sets of 30, one cow standing on the back of the other (they are quite amazing cows). Thus, the pasture is dotted with both these stacks of 30 cows and also, in separate locations, M (1 <= M <= 1,000) haystacks. Below is a sample of one way they might be laid out:

```
8 .........
7 ....CH.H.         C = stack of 30 cows
6 .........
5 .........         H = haystack
4 ..C.HH...
3 .........
2 .....C.HH
1 .........
  123456789
```

As the musical's conductor, Farmer John has four whistles with various tones. One whistle commands the cow at the bottom of each stack to move (along with all the stacked cows) one unit north, another indicates a move to the south, one indicates a move to the east, and a fourth to order a move to the west.

Any time the stack of cows enters a grid location with a haystack, the cow on the top of the stack (even if the stack has height one) will jump onto the haystack while the remaining cows move into the same location as the haystack. Thus, if the bottom cow encounters 30 haystacks (perhaps different haystacks, perhaps not), the stack of 30 cows is exhausted with all the cows standing on top of haystacks (or standing on cows on haystacks). The sturdy haystacks can each support an unlimited number of cows.

Farmer John glances across his pasture to Farmer Don's milking facility to see, to his horror, a huge milk tank exploding and unleashing a giant tidal wave of milk making its way toward the performing cows! Since any cows on a haystack are safe, FJ must now do what he can to save the lives of as many cows as possible using what has turned from a simple dance routine into a lifesaving technique.

Given the number of times K (1 <= K <= 30) farmer John can blow a whistle until the wave of milk crashes over the pasture and also the X_i, Y_i positions (1 <= X_i <= 1,000; 1 <= Y_i <= 1,000) of the N stacks of cows and M haystacks (none of which currently has any cows on it), report the greatest number of cows that can be saved and find a sequence of whistle blows that does the job. The sequence should be reported in terms of the four directions, 'E'

for east, 'N' for north, 'W' for west, 'S' for south. Among all such sequences, farmer John wants the lexicographically least. Initial locations of cows and haystacks will not share the same coordinates in the input file.

Cows can be moved to any location, including ones outside the pasture.

PROBLEM NAME: crisis

INPUT FORMAT:

* Line 1: Three space-separated integers: N, M, and K

* Lines 2..N+1: Line i+1 describes the X,Y location of a stack of 30 cows using two space-separated integers: X_i and Y_i

* Lines N+2..N+M+1: Line i+N+1 describes the X,Y location of a haystack using two space-separated integers: X_i and Y_i

SAMPLE INPUT (file crisis.in):

```
3 6 3
3 4
6 2
5 7
8 2
9 2
6 4
5 4
6 7
8 7
```

OUTPUT FORMAT:

* Line 1: A single integer that is the most number of cows that can be saved.

* Line 2: K characters, the lexicographically least sequence of commands FJ should issue to maximize the number of cows saved.

SAMPLE OUTPUT (file crisis.out):

```
6
EEE
```

OUTPUT DETAILS:

Use the 'east' whistle three times, at which point the milk floods the area. Each haystack ends up saving 1 cow.

Analysis: Crisis on the Farm by Richard Peng

Note that the relative position of the cows do not change (we're shifting all the cows together per move). Also, our score is exactly the number of cows that overlap with hays after each move, which is determined from the relative position of cows and hay.

Therefore, we could perform DP with [# of moves,# relative position] as our state. Since we can move by distance of 1 per move, our DP state has size O(K^3). We also need to precompute the number of haystacks that overlap with cows for each of the relative positions, which we could do either in O(NM) or O(NK^2), both of which suffices.

After we do our DP, we need to print the lexicographically least trace. This is not hard, but a bit tricky to get right nontheless.

Below is Brian Dean's solution:

```
#include <stdio.h>

#define MAX_N 1000
#define MAX_M 1000
int K, N, M;
int Cx[MAX_N], Cy[MAX_N];
int Hx[MAX_M], Hy[MAX_M];
int count[63][63];
int best[31][63][63];
char soln[31][63][63];

#define ABS(x) ((x) > 0 ? (x) : -(x))
#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
  int i, j, k, b;
  freopen("crisis.in", "r", stdin);
  freopen("crisis.out", "w", stdout);

  scanf ("%d %d %d", &N, &M, &K);
  for (i=0; i<N; i++) scanf ("%d %d", &Cx[i], &Cy[i]);
  for (i=0; i<M; i++) scanf ("%d %d", &Hx[i], &Hy[i]);

  for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      if (ABS(Cx[i]-Hx[j]) <= 30 && ABS(Cy[i]-Hy[j]) <= 30)
```

```
      count[31+Cx[i]-Hx[j]][31+Cy[i]-Hy[j]]++;

  for (i=0; i<=62; i++)
    for (j=0; j<=62; j++)
      for (k=0; k<=K; k++) {
        best[k][i][j] = -999999999;
        soln[k][i][j] = 'Z';
      }

  best[0][31][31] = 0;

  for (k=1; k<=K; k++)
    for (i=1; i<=61; i++)
      for (j=1; j<=61; j++) {
        b = best[k-1][i-1][j];
        b = MAX(b, best[k-1][i+1][j]);
        b = MAX(b, best[k-1][i][j-1]);
        b = MAX(b, best[k-1][i][j+1]);
        best[k][i][j] = b + count[i][j];
      }

  b = 0;
  for (i=1; i<=61; i++)
    for (j=1; j<=61; j++)
      b = MAX(b, best[K][i][j]);

  for (i=1; i<=61; i++)
    for (j=1; j<=61; j++)
      if (best[K][i][j] == b) soln[K][i][j] = 'Y';

  for (k=K-1; k>=0; k--)
    for (i=1; i<=61; i++)
      for (j=1; j<=61; j++) {
        if (best[k][i][j] + count[i+1][j] == best[k+1][i+1][j] &&
            soln[k+1][i+1][j] < 'Z') soln[k][i][j] = 'W';
        if (best[k][i][j] + count[i][j+1] == best[k+1][i][j+1] &&
            soln[k+1][i][j+1] < 'Z') soln[k][i][j] = 'S';
        if (best[k][i][j] + count[i][j-1] == best[k+1][i][j-1] &&
            soln[k+1][i][j-1] < 'Z') soln[k][i][j] = 'N';
        if (best[k][i][j] + count[i-1][j] == best[k+1][i-1][j] &&
            soln[k+1][i-1][j] < 'Z') soln[k][i][j] = 'E';
      }

  printf ("%d\n", b);
  i = 31; j = 31;
  for (k=0; k<K; k++) {
    printf ("%c", soln[k][i][j]);
    if (soln[k][i][j] == 'E') i--;
    if (soln[k][i][j] == 'W') i++;
    if (soln[k][i][j] == 'S') j++;
    if (soln[k][i][j] == 'N') j--;
  }
  printf ("\n");
  return 0;
}
```

Michael Cohen writes: There is a simpler variant of the above solution that leaves the most cows that can be saved overall in a specific index of the array and gets the specific path far more simply. The idea is to use the same precomputation and essentially the same dynamic programming algorithm as the above solution with a simple change. That change is to do the dynamic programming "backwards" instead of "forwards," storing the most cows that can be saved starting from each time in each relative position, rather than ending at each time in each relative position. This means, of course, that later times are computed first, but other than that the "forwards" solution can be used essentially unchanged. It also means that the most cows that can be saved overall is simply that array value for the initial position and the initial time. But most importantly, it means that an array storing the move made from each position at each time (soln in Brian Dean's solution) can be computed at the same time as the most cows that can be saved; the analysis solution in fact generates this through backtracking, but that is not needed if you are going backwards in the first place!

```
#include <algorithm>
#include <fstream>
using namespace std;

struct coord {
        int x;
        int y;
        int sum;
        int sum2;
};

bool operator<(coord a, coord b) {
        return a.sum < b.sum;
}

int N, C;
coord coords[100000];
int dsds[100000];
int size[100000];
int numNeighborhoods;

int find(int i) {
    if (dsds[i] == i) return i;
    else return (dsds[i] = find(dsds[i]));
}

int main() {
    ifstream inp("nabor.in");
    ofstream outp("nabor.out");

    inp >> N >> C;
    for (int i = 0; i < N; i++) {
        inp >> coords[i].x >> coords[i].y;
        coords[i].sum = coords[i].x+coords[i].y;
```

```
        coords[i].sum2 = coords[i].x-coords[i].y;
        dsds[i] = i;
        size[i] = 1;
    }
    sort(coords, coords+N);

    int numNeighborhoods = N;
    int largest = 1;
    for (int i = 0; i < N; i++) {
        for (int j = i-1; j >= 0 && coords[i].sum-coords[j].sum <= C; j--) {
            if (coords[j].sum2-coords[i].sum2 <= C && coords[j].sum2-
coords[i].sum2 >= -C) {
                int end1 = find(i);
                int end2 = find(j);
                if (end1 != end2) {
                    dsds[end1] = end2;
                    size[end2] += size[end1];
                    if (size[end2] > largest) largest = size[end2];
                    numNeighborhoods--;
                }
            }
        }
    }

    outp << numNeighborhoods << ' ' << largest << endl;
    return 0;
}
```

12. **Award [Chinese TST 2002, Day 2 Problem 1, 2007] (38, 38) Camp 07 day 6**

Memory Limit: 65 MB
Time Limit: 1.5 second

The award ceremony of IOI 2002 will be held in Yong-In Hall. To make the ceremony more interesting, the organizers want to build the stage in the shape of an I to represent Informatics. However, many sponsors have erected displays in the hall and, of course, they do not want to move these displays. You have been asked to perform the task of selecting the location of this stage.

Yong-In Hall has the shape of a rectangular grid with R (1 <= R <= 150) rows and C (1 <= C <= 150) columns. The I-shaped stage will be placed rectilinearly. The stage will be in the shape of 3 rectangles stacked on top of each other. The width of the middle rectangle must be less than the widths of both the top and bottom rectangles. The upper and lower rectangles must have at least one segment farther right and one segment farther left than the middle segment's span (otherwise, the stage might be mistaken for T, L or J). The widths of the top and bottom may differ as may the heights.

For example, the stage layouts below are valid (I represents part of the stage, . represents empty space:

```
IIIII.      .IIIII
..II.. and  ..II..
IIIIII      IIIII.
IIIIII
```

The following stages are not valid:

```
IIII
.II.
III. (lower rectangle does not extend beyond the middle segment)

IIIII
.....
..I..
III.. (since it's not connected)

I..I
IIII
I..I (the top and bottom rectangles are not rectangles)
```

Your program should find the area of the largest 'I' that does not overlap with any of the displays. Output 0 if no 'I' can be made.

PROBLEM NAME: award

INPUT FORMAT:

* Line 1: Two space-separated integers: R and C

* Lines 2..R+1: Line i+1 contains row i expressed as C space-separated integers. The jth integer is 1 if there is a display board in column j and 0 otherwise.

SAMPLE INPUT (file award.in):

```
6 8
1 1 1 1 1 0 0 1
1 0 0 0 0 1 1 1
1 0 0 0 0 0 1 1
1 0 1 0 1 0 1 0
1 0 0 0 0 0 0 1
1 1 0 0 0 1 0 1
```

INPUT DETAILS:

The grid has 6 rows and 8 columns with a total of 23 displays

OUTPUT FORMAT:

SAMPLE OUTPUT (file award.out):

```
15
```

OUTPUT DETAILS:

The maximum sized I is denoted by the Is in the following diagram:

```
XXXXX..X
XIIIIXXX
XIIII.XX
X.XIX.X.
X.III..X
XXIIIX.X
```

Analysis: Award by Richard Peng

The 'I' described in this problem is essentially 3 rectangles stacked on top of
each other so that the middle one is 'thinner' than the one above and below it.
So naturally, the solution is a lot of DPs stacked on top of each other.

We first compute the 2D sum array so we can calculate the sum of any rectangle
of the grid in O(1) time. It's a useful tool for us to have in determining whether a rectangle is indeed all 1s.

We now consider the uppermost rectangle.

We want to compute the following info:

For each [r,c1,c2], what's the biggest rectangle with bottom in row r such that
[c1,c2] is strictly contained in its range of columns.

To do this, we first find the largest rectangle we can get extend upwards from
each [r,c1',c2'] ([c1',c2'] is the base). Then we can perform subset DP to calculate the values of [r,c1,c2].

We can do this for the bottom similarly.

So now the problem becomes using this information for the middle rectangle.

If we fix the two columns of the middle rectangle, we have, for each r, whether
we can use that row or now, and the 'bonus' we receive in the form of the upper/lower rectangle.

So we can consider each contiguous sequence of available rs and reduce the problem to a linear list version, namely maximizing:
bestupper[i]+(j-i+1)*(c2-c1+1)+bestlower[j].
or alternatively maximizing:
bestlower[j]+(j+1)*(c2-c1+1) + (bestupper[i]-i*(c2-c1+1))
The 1st term only depends on j and the 2nd only depends on i, so we can do this
in linear time by looping through the list and tracking the best value of bestupper[i]-i*(c2-c1+1) encountered so far.

Here is a sample solution from *Mahbub*

```
/*
```

```
Prob ID: 0726
Name: award
Author: Md. Mahbubul Hasan
Date: 8th January, 2008.
LANG: C++
*/
#include<stdio.h>

#define INF 1000000000
inline int MAX(int &A,int B) {if(B>A) A=B; return A;}
inline int MIN(int &A,int B) {if(B<A) A=B; return A;}

int R,C,c1,c2,r,c,now,ans,len,cc,lim,j;
int
dpup[160][160][160],dpdown[160][160][160],left[160][160],up[160][160],down[16
0][160];
int a[160][160];

int main()
{
        freopen("award.in","r",stdin);
        freopen("award.out","w",stdout);

        scanf("%d%d",&R,&C);

        for(r=1;r<=R;r++)
                for(c=1;c<=C;c++)
                {
                        scanf("%d",&a[r][c]);
                        left[r][c]=left[r][c-1]+(a[r][c]==1);
                }

        for(r=1;r<=R;r++)
                for(c=1;c<=C;c++)
                        if(a[r][c]) up[r][c]=0;
                        else up[r][c]=up[r-1][c]+1;

        for(r=1;r<=R;r++)
        {
                for(c1=1;c1<=C;c1++)
                        for(now=INF,c2=c1;c2<=C;c2++)
                                MAX(dpup[r][c1][c2],MIN(now,up[r][c2])*(c2-
c1+1));

                for(len=C;len>=2;len--)
                        for(lim=C-len+1,c1=1;c1<=lim;c1++)
                        {
                                c2=c1+len-1;
                                MAX(dpup[r][c1+1][c2],dpup[r][c1][c2]);
                                MAX(dpup[r][c1][c2-1],dpup[r][c1][c2]);
                        }
        }

        for(r=R;r>=1;r--)
                for(c=1;c<=C;c++)
                        if(a[r][c]) down[r][c]=0;
                        else down[r][c]=down[r+1][c]+1;
```

```
        for(r=R;r>=1;r--)
        {
                for(c1=1;c1<=C;c1++)
                        for(now=INF,c2=c1;c2<=C;c2++)
                                MAX(dpdown[r][c1][c2],MIN(now,down[r][c2])*(c2-
c1+1));

                for(len=C;len>=2;len--)
                        for(lim=C-len+1,c1=1;c1<=lim;c1++)
                        {
                                c2=c1+len-1;
                                MAX(dpdown[r][c1+1][c2],dpdown[r][c1][c2]);
                                MAX(dpdown[r][c1][c2-1],dpdown[r][c1][c2]);
                        }
        }

        ans=0;
        for(c1=2;c1<C;c1++)
                for(c2=c1;c2<C;c2++)
                {

                        now=-INF;
                        cc=c2-c1+1;

                        for(j=1;j<R;j++)
                        {
                                if(left[j][c2]-left[j][c1-1]) {now=-INF;
continue;}
                                if(j>1)
                                {
                                        MAX(now,dpup[j-1][c1-1][c2+1]-j*cc);
                                        if(dpdown[j+1][c1-1][c2+1])
MAX(ans,now+dpdown[j+1][c1-1][c2+1]+(j+1)*cc);
                                }

                        }
                }

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

        return 0;
}
```

13. **Ranklist Sorting [Baltic BOI, 2007] (45, 40)**

You are given the unique competition scores S_i (0 <= S_i <= 1,000,000) of N (2 <= N <= 1,000) tournament players conveniently numbered 1..N. Your task is to create a rank list of the players, sorted in descending order by score.

Unfortunately, the data structure used for the list of players supports only one operation: it moves a player from position i to position j without changing the relative order of other players. If i > j, the positions of players at positions between j and i-1 increase by 1; otherwise if i < j the positions of players at positions between i+1 and j decrease by 1. The operation has cost i+j.

Determine a sequence of these operations to create the rank list such that the sum of the moves' costs is minimized.

PROBLEM NAME: sorting

INPUT FORMAT:

* Line 1: A single integer: N.

* Lines 2..N+1: Line i+1 contains the single integer: S_i.

SAMPLE INPUT (file sorting.in):

```
5
20
30
5
15
10
```

OUTPUT FORMAT:

* Line 1: The minimum total cost of sorting the list.

SAMPLE OUTPUT (file sorting.out):

```
11
```

```
OUTPUT DETAILS:

                                        Cost   Total cost
Initial sequence:     20, 30, 5, 15, 10              0
Operation (2,1):      30, 20, 5, 15, 10  1+2         3
Operation (3,5):      30, 20, 15, 10, 5  3+5        11
```

Analysis: Ranklist Sorting by Md. Mahbubul Hasan

This is a very tough DP problem. Firstly we should understand that,

1. We should fix the smallest number first, then next smallest in this way.

2. While moving a number we should place it at bottom.

As we understand these two facts the rest relies on implementation.

My solution uses N^2 DP,

dp[x][y] = the minimum cost of fixing 1~x th numbers while xth one is at y.

pos[i] = the desired position of the number at i th position in the initial list.

cnt[i] = the number of numbers in the initial list from 1 to i th position which are greater than the number at this position in the sorted list.(it also includes the number itself)

For more detailed analysis you can download the official analysis: file:// localhost/tasks/book.pdf

Here is my solution:

```
/*
Prob ID: 0997
Name: sorting
Author: Md. Mahbubul Hasan
Date: 7th January, 2008.
LANG: C++
*/
#include<stdio.h>
#include<utility>
```

```
#include<vector>
#include<functional>
#include<algorithm>
using namespace std;

#define MAX 1002
#define MIN(A,B) ((A) < (B) ? (A) : (B))

typedef pair<int,int> PII;
vector<PII> V;
int cnt[MAX],pos[MAX],dp[MAX][MAX];

int main()
{
        freopen("sorting.in","r",stdin);
        freopen("sorting.out","w",stdout);

        int n,i,j,val,x,y;

        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                scanf("%d",&val);
                V.push_back(PII(val,i));
        }

        sort(V.begin(),V.end(),greater<PII>());

        for(i=0;i<n;i++) pos[V[i].second]=i+1;

        for(i=1;i<=n;i++)
                for(cnt[pos[i]]=0,j=1;j<=i;j++)
                        cnt[pos[i]]+=(pos[j]<=pos[i]);

        for(i=0;i<=n;i++) for(j=0;j<=n;j++) dp[i][j]=1000000000;

        dp[0][0]=0;
        for(x=1;x<=n;x++)
                for(y=0;y<=n;y++)
                        if(cnt[x]==y)
                                dp[x][y]=MIN(dp[x][y],dp[x-1][y-1]);
                        else if(cnt[x]<y)
                        {
                                dp[x][y]=MIN(dp[x][y],dp[x-1][y-1]+cnt[x]+y);
                                dp[x][y]=MIN(dp[x][y],dp[x-1][cnt[x]-
1]+((y+cnt[x]+1)*(y-cnt[x]))/2);
                        }
                        else
                                dp[x][y]=MIN(dp[x][y],dp[x-1][y]+cnt[x]);

        printf("%d\n",dp[n][n]);

        return 0;
}
```

14. Weight Reconstruction [Coaches, 2004] (47, 40) Camp 05 Day 3

Farmer John has forgotten how old his N (1 <= N <= 1,000) cows are. Obviously, some cows might be the same age as others. Also, some ages might not be represented in FJ's herd.

FJ keeps the cows on a precise feed schedule, so he knows that every cow that is the same age weighs the same. FJ decided to weigh the cows to determine their ages.

FJ took the cows to have them weighed, but the weigh station had a new method for weighing cows. To weigh the cows, the cows are put into a long line. The station then weighs the first cow, the first two cows together, the first three cows together, and so on for the whole line. Subsequenty, the station weighs the last cow, the last two cows together, the last three cows together, and so on for the whole line.

Given either of these weighing sequences, FJ could subtract adjacent weighings and determine the weight of each cow. Unfortunately, the weigh station forgot to tell him which weighings are which and, to make matters worse, mixed them up.

For example, suppose FJ took 5 cows of weight 1, 1, 5, 2, and 5, in that order. The weigh station would have recorded the following weights:

```
 1 = 1                      5 = 5
 2 = 1+1                    7 = 2+5
 7 = 1+1+5                 12 = 5+2+5
 9 = 1+1+5+2               13 = 1+5+2+5
14 = 1+1+5+2+5             14 = 1+1+5+2+5
```

FJ has the list of possible weights (range: 1..10,000) and the set of weights from the weigh station (range: 1...10,000,000), in no particular order. Help FJ determine the possible weights of his cows that would result in the measured weights. If there are multiple possible weights, find the one that puts lighter cows first.

PROBLEM NAME: weight

INPUT FORMAT:

* Line 1: Two space-separated integers, N and K, the number of
        possible cow weights (1 <= K <= 1,000)

* Lines 2..K+1: The possible cow weights, one per line

* Lines K+2..K+2N+1: An integer which is either the sum of the weights
        of the first k cows or the last k cows for some k <= N.

SAMPLE INPUT (file weight.in):

```
5 4
1
2
5
4
1
2
5
7
14
7
9
12
13
14
```

INPUT DETAILS:

There are 5 cows, whose weights can be 1, 2, 4, or 5. The weights of all the non-empty prefixes and suffixes of the original line are 1, 2, 5, 7, 7, 9, 12, 13, 14, and 14, as in the example described above.

OUTPUT FORMAT:

* Line N lines: An ordering of the weights of N cows that would give
        rise to the prefix and suffix weights in the input. If there
        is more than one ordering, print the lexicographically least
        one: favor orderings that put lighter cows first. If there is
        no ordering, output "0".

SAMPLE OUTPUT (file weight.out):

```
1
1
5
2
5
```

OUTPUT DETAILS:

There are other possible weights, but 1, 1, 5, 2, 5 is the one that puts lighter cows first. (e.g., 1, 4, 2, 5, 2 is another possible weights, but 1, 1, 5, 2, 5 is better because it puts a lighter cow first (second in line, 1 instead of 4).

Analysis: Weight Reconstruction by Richard Peng

The statement about choosing a selection that puts lighter cows first suggests
strongly of memorization search since any sort of iterative DP would result in
a messy reconstruction of the trace.

If we sort the sums, the problem becomes diving a list into two lists of size n
such that the consecutive difference of the two lists are in reversed order, and the condition of putting smaller cows first is equivalent to break tied by
putting a cow into the forward group whenever possible when considering the sums in increasing order.

So this yields a O(2^n) bruce algorithm, which we attempt to optimize by checking whether an answer is possible on the spot rather than at the end of the search.

A key observation is that the largest sum is the total sum of all the cows, and
if x is in the list, so must be sum-x it's the other part of the partial sum. So it suffices to check just whether the difference of the two elements selected is a possible cow weight.

So our DP state can just be location and the location of the previous cows chosen for either group. This looks like O(n^3), but one of the cows must be the previous one, so there are acutally O(n^2) states. A straight implementation using memorization suffices.

A sample solution from *Mahbub* follows:

```
/*
Prob ID: 0369
Name: weight
Author: Md. Mahbubul Hasan
Date: 9th January, 2008.
LANG: C++
```

```
*/
#include<stdio.h>
#include<vector>
#include<algorithm>
#include<utility>
using namespace std;

typedef pair<int,int> PII;
vector<PII> Z;
vector<int> V,Y;
int dp[1010][1010],num[1010],u[10002],N;

int DP(int left,int right,int n)
{
        if(n==N+1) return ((Z[right].second-Z[left].first)<=10000 &&
u[Z[right].second-Z[left].first]);

        if(dp[left][right]) return 0;
        dp[left][right]=1;

        if((Z[n].first - Z[left].first)<=10000 && u[Z[n].first -
Z[left].first])
        {
                num[n]=Z[n].first;
                if(DP(n,right,n+1))
                        return 1;
        }

        if((Z[right].second - Z[n].second)<=10000 && u[Z[right].second -
Z[n].second])
        {
                num[n]=Z[n].second;
                if(DP(left,n,n+1))
                        return 1;
        }

        return 0;
}

int main()
{
        freopen("weight.in","r",stdin);
        freopen("weight.out","w",stdout);

        int K,a,i,j,sz;

        scanf("%d%d",&N,&K);
        while(K--)
        {
                scanf("%d",&a);
                u[a]=1;
        }

        for(i=1;i<=2*N;i++)
        {
                scanf("%d",&a);
                V.push_back(a);
```

```
	}

	V.push_back(0);
	V.push_back(0);

	sort(V.begin(),V.end());

	sz=V.size();

	Z.push_back(PII(0,V[sz-1]));
	Z.push_back(PII(0,V[sz-1]));

	for(i=2,j=sz-3;i<j;i++,j--)
		Z.push_back(PII(V[i],V[j]));

	sz=Z.size();

	num[0]=0;
	num[1]=Z[0].second;

	if(DP(0,1,2)==0)
	{
		printf("0\n");
		return 0;
	}

	sort(num,num+N+1);

	for(i=N;i>=1;i--) num[i]-=num[i-1];

	a=0;
	for(i=0;i<=N;i++)
	{
		a+=num[i];
		Y.push_back(a);
	}

	Y.push_back(a=0);
	for(i=N;i>=1;i--)
	{
		a+=num[i];
		Y.push_back(a);
	}

	sort(Y.begin(),Y.end());

	if(V!=Y) {printf("0\n"); return 0;}
	for(i=1;i<=N;i++)
		printf("%d\n",num[i]);

	return 0;
}
```

BFS IN DP STATE-SPACE

1.Around the world [Dutch Championships, via Jan Kuipers, 2004] (33, 40) Open 05 Gold

Memory Limit: 18MB due to system change.

Over the years, FJ has made a huge number of farmer friends all around the world. Since he hasn't visited 'Farmer Ted' from England and 'Boer Harms' from Holland for a while, he'd like to visit them.

He knows the longitude of the farm where each of his worldwide friends resides. This longitude is an angle (an integer in the range 0..359) describing the farm's location on the Earth, which we will consider to be a circle instead of the more complex and traditional spherical representation. Except for the obvious discontinuity, longitudes increase when traveling clockwise on this circle.

FJ plans to travel by airplane to visit his N (1 <= N <= 5,000) friends (whose farms are uniquely numbered 1..N). He knows the schedules for M (1 <= M <= 25,000) bidirectional flights connecting the different farms. Airplanes always travel shortest paths on the Earth's surface (i.e., on the shortest arc of a circle).

There will always be a unique shortest path between two farms that are directly connected. No pair of antipodal farms (exactly opposite each other on the circle) is ever directly connected.

Each airplane flight can be described as traveling in clockwise or counterclockwise direction around the Earth's surface. For example, a flight from longitude 30 to longitude 35 would be clockwise, as would be a flight from longitude 350 to longitude 10. However, a flight from longitude 350 to longitude 200 follows a shortest path counterclockwise around the circle.

FJ would find it very cool if he could make a trip around the world, visiting some of his friends along the way. He'd like to know if

this is possible and if so, what is the minimum number of flights
he can take to do so.

He wants to start and finish his journey at the location of his
best friend (the one listed first in the input below). In order
to make sure he actually circles the Earth, he wants to ensure that
the clockwise distance he travels is different from the counterclockwise
distance he travels.

PROBLEM NAME: around

INPUT FORMAT:

* Line 1: Two space-separated integers: N and M

* Lines 2..N+1: Line i+1 contains one integer: the longitude of the
  i-th farm. Line 2 contains the location of the farm of his
  best friend.

* Lines N+2..N+M+1: Line i+N+1 contains two integers giving the
  indices of two farms that are connected by a flight.

SAMPLE INPUT (file around.in):

```
3 3
0
120
240
1 2
2 3
1 3
```

INPUT DETAILS:

Farmer John has three friends at longitudes 0, 120, and 240. There are
three flights: 0<->120, 120<->240, and 0<->240. The journey must start and
finish at longitude 0.

OUTPUT FORMAT:

* Line 1: A single integer specifying the minimum number of flights FJ
  needs to visit to make a trip around the world. Every time FJ
  moves from one farm to another counts as one flight. If it is
  impossible to make such a trip, output the integer -1.

SAMPLE OUTPUT (file around.out):

```
3
```

OUTPUT DETAILS:

FJ must visit all 3 friends to make a full trip around the world.

Analysis: Around the world by Richard Peng

The DP state can be the location of FJ and how many times he has gone around the world. On first look, this gives 5000*(5000*2-1), which is way too many. However, notice that by symmetry, it suffices to consider cases where FJ went around a positive number of times. Also notice at least 3 flights are required
to go around the world once, so FJ can go around the world at most 5000/3=1700
times. Also, once FJ has gone around the world more than 900 times, he cannot possibly go in the reverse direction 900 times as that would mean he has gone around a total of more than 1800 times, so we can have a special state to keep
track of that. So our states are not reduced to 5000*900=4,500,000, on which we
could perform BFS . So our algorithm runs in O(900*M) time, which is sufficient.

Here is the solution implementing the above idea:

```
/*
ID: peng_ri1
LANG: C++
TASK: around
*/

#include <cstdio>
#include <vector>
using namespace std;

#define MAXN 6000

int mi(int a,int b){return (a<b)?a:b;}
int ma(int a,int b){return (a>b)?a:b;}

int loc[MAXN],tai,n,m;
short dis[5000][1000],shift,q[1000000][2];
```

```
vector <int> e[MAXN],typ[MAXN];

void add(int v1,int v2,int d){
        if(dis[v1][v2+shift]==-1){
                dis[v1][v2+shift]=d;
                q[tai][0]=v1;
                q[tai][1]=v2;
                tai++;
        }
}

int main(){
        int a,b,c,i,j;
freopen("around.in","r",stdin);
freopen("around.out","w",stdout);
        scanf("%d%d",&n,&m);
        for(i=0;i<n;i++)        scanf("%d",&loc[i]);
        for(i=0;i<m;i++){
                scanf("%d%d",&a,&b);
                a--;    b--;
                if(mi(loc[a],loc[b])+180<ma(loc[a],loc[b])){
                        if(loc[a]<loc[b])       c=-1;
                        else c=1;
                }
                else    c=0;
                e[a].push_back(b);
                typ[a].push_back(c);
                e[b].push_back(a);
                typ[b].push_back(-c);
        }
        for(i=0;i<n;i++)        for(j=0;j<1000;j++)    dis[i][j]=-1;
        tai=0;
        add(0,0,0);
        for(i=0;i<tai;i++){
                a=q[i][0];
                b=q[i][1];
                if((a==0)&&(b!=0)){
                        printf("%d\n",dis[a][b+shift]);
                        return 0;
                }
                for(j=0;j<e[a].size();j++)
                        add(e[a][j],b+typ[a][j],dis[a][b+shift]+1);
        }
        printf("%d\n",-1);
        return 0;
}
```

Further analysis by *Mahbub* :

We can also consider it as graph theoritical problem. Say we are starting from
'0' and return to it, when can we say that we have gone around the world? if and only if our directed distance is not 0. If we are not been through the world then we have went same amound cw and same amount ccw. so net 0. But if we have been through the entire world then we would add 360 degree each time. so non zero at least.

One more thing to note that, we dont need to store more than one "reaching

incident" for each node. Say we have once reached at x, by d[x] angular distance and step[x], then reaching again at x, by same d[x] is not worthy once step[x] is least. But by other d[x] it is important. that means, by 0->x by prev d[x] and 0->x by someother d[x]. Hence we got the cycle. so when we reach x by some other distance other than d[x] we will try to update our answer.

Here is the code implementing the above idea:

```
/*
Prob ID: 0423
Name: around
Author: Md. Mahbubul Hasan
Date: 13th October, 2008.
LANG: C++
*/
#include<stdio.h>
#include<queue>
#include<vector>
#include<utility>
using namespace std;

#define MIN(A,B) ((A) < (B) ? (A) : (B))
typedef pair<int,int> PII;
vector<PII> V[5002];
int loc[5002];
int N,M;
int d[5002],step[5002];
queue<int> Q;

int main()
{
        freopen("around.in","r",stdin);
        freopen("around.out","w",stdout);

        int i,a,b,c,x,sz,ans;

        scanf("%d%d",&N,&M);

        for(i=1;i<=N;i++)
                scanf("%d",&loc[i]);

        while(M--)
        {
                scanf("%d%d",&a,&b);
                c=loc[b]-loc[a];
                if(c<0) c+=360;
                if(c>180) c=-(360-c);
                V[a].push_back(PII(b,c));
                V[b].push_back(PII(a,-c));
        }

        for(i=2;i<=N;i++) d[i]=-1;

        Q.push(1);
        ans=1000000000;
        while(!Q.empty())
        {
```

```
                x=Q.front();
                Q.pop();

                sz=V[x].size();
                for(i=0;i<sz;i++)
                {
                        if(d[V[x][i].first]==-1)
                        {
                                d[V[x][i].first]=d[x]+V[x][i].second;
                                step[V[x][i].first]=step[x]+1;
                                Q.push(V[x][i].first);
                        }
                        else if(d[x]+V[x][i].second != d[V[x][i].first])
                                ans=MIN(ans,step[x]+1+step[V[x][i].first]);
                }
        }

        if(ans==1000000000) printf("%d\n",-1);
        else printf("%d\n",ans);

        return 0;
}
```

2. Amazing Robots [IOI, 2003] (33, 33)

Memory Limit: 18MB due to system change.

You are the proud owner of two robots that are located in separate rectangular mazes each of which has no more than 20 rows or columns. Square (1, 1) in a maze is the square in the upper-left (northwest) corner. Maze i (i = 1, 2) has a set of G_i (0 <= G_i <= 10) guards trying to capture the robots. They patrol back and forth on a straight patrol path.

Your goal is to determine a sequence of robot commands such that both robots exit the mazes without either robot being captured by a guard.

Each robot command is a direction (north, south, east, or west). At the beginning of each minute, you broadcast the same command to both robots. A robot moves one square in the direction of the command, unless the robot would collide with a wall, in which case the robot does not move for that minute. A robot exits the maze by walking out of it. A robot ignores commands after it has exited its maze.

Guards move one square at the beginning of each minute, at the same time as the robots. Each guard begins at a given square facing a given direction and moves forward one square per minute until the guard has moved one fewer square than the number of squares in his patrol path. The guard then turns around instantaneously and walks in the opposite direction back to his starting square, where he again turns around and repeats his patrol path until each robot has exited its maze. A guard's patrol will not require the guard to walk through walls or exit the maze.

Although guard patrols may overlap, no two guards will ever collide: they will never occupy the same square at the end of a minute, and they will not exchange squares with each other during a minute. A guard in a maze will not start in the same square as the robot in that maze. A guard captures a robot if the guard occupies the same square at the end of a minute as the robot, or if the guard and the robot exchange squares with each other during a minute.

Given the layout of the two mazes along with the initial square of each robot and the patrol paths of the guards in each maze, determine a sequence of commands for which the robots exit without being caught by the guards. Minimize the time it takes for the later robot to exit its maze. If the robots exit at different times, the time at which the earlier robot exited does not matter.

TIME LIMIT: 5 seconds

PROBLEM NAME: robots

INPUT FORMAT:

* Line 1: Two space-separated integers: R1 and C1, the the number of rows and columns of the first maze

* Lines 2..R1+1: C1 characters each specifying the maze layout. The robot's starting square is specified by an `X'. A `.' represents an open space and `#' represents a wall. Each maze will contain exactly one robot.

* Line R1+2: A single integer: G_1

* Lines R1+3..R1+G1+2: Line i+R1+G1+1 contains three space-separated integers and a character describing guard i's: row, column, guard path length (2 <= length <= 4), and the initial direction the guard faces (`N', `S', `E', or `W'; north, south, east, and west, respectively).

* Lines R1+G1+3..end of input: Another set of lines formatted the same way, the description of the second maze.

SAMPLE INPUT (file robots.in):

```
5 4
####
#X.#
#..#
...#
##.#
1
4 3 2 W
4 4
####
#...
#X.#
####
0
```

OUTPUT FORMAT:

* Line 1: A single integer K (K <= 10000), the number of commands for
        both robots to exit the maze without being captured. If no
        such sequence exists, output a single line containing ``-1''.

SAMPLE OUTPUT (file robots.out):

8

OUTPUT DETAILS:

E N E S S S E S will do the trick; nothing shorter will.

Analysis: Amazing Robots by original analysis from IOI

Note that the guards simultaneously return to their starting location every
12
minutes.

For analysis purposes, assume that the mazes are of equal size. More over,
let
n be the number of squares in the maze (R×C).

So we could perform a breadth-first search where the state of the search is
the
location of the two robots and the current minute within the guard cycle of
12
minutes. Use memorization to ensure that no state is visited more than once.

This algorithm takes O(12 n^2) time, where n is the size of the maze.

Note:

What's amazing about this problem is how few people actually got it during
the
contest. It seems almost everybody managed to make a mistake somewhere in the
150+ lines of code. The best way to implement this is probably first
precompute
for each of the 12 states whether a position and transitoin on the board is
doable, which also requires precomputation of the locations of the robots.
Overall, it's quite a pain.

Here is a compact implementation from *Spencer*
/*

```
ID: iceeight1
LANG: C++
TASK: robots
SOURCE: IOI 2003
*/

#include <stdio.h>
#include <iostream>
#include <queue>
#define MAXD 25
#define MAXG 12
#define MAXP 12
#define OUT 22
using namespace std;
#define REP(i,n) for(int i=0;i<n;i++)
#define FOR(i,a,b) for(int i=a;i<=b;i++)

const int dy[4] = { -1, 0, 1, 0 },
                  dx[4] = { 0, 1, 0, -1 };

struct loc {
        int y, x;
        loc() {}
        loc(int _y, int _x): y(_y), x(_x) {}
        bool operator==(loc a) { return y == a.y && x == a.x; }
};

struct state {
        loc a, b; int d;
        state(loc _a, loc _b, int _d) { a = _a; b = _b; d = _d; }
};

int R[2], C[2], map[2][MAXD][MAXD]; loc s[2];
int G[2]; loc g[2][MAXG][MAXP];
queue<state> q; bool seen[MAXD][MAXD][MAXD][MAXD][MAXP];

bool mark(loc a, loc b, int p) {
        if (seen[a.y][a.x][b.y][b.x][p]) return false;
        return seen[a.y][a.x][b.y][b.x][p] = true;
}

int main() {
        FILE *fin = fopen("robots.in", "r"),
                 *fout = fopen("robots.out", "w");
        REP(n, 2) {
                fscanf(fin, "%d %d\n", &R[n], &C[n]);
                REP(i, R[n]) {
                        REP(j, C[n]) {
                                char c; fscanf(fin, "%c", &c);
                                if (c == '#') map[n][i][j] = 1;
                                if (c == 'X') s[n] = loc(i, j);
                        }
                        fscanf(fin, "\n");
                }
                fscanf(fin, "%d", &G[n]);
                REP(i, G[n]) {
                        int y, x, p; char c; fscanf(fin, "%d %d %d %c", &y,
```

```
&x, &p, &c); y--; x--;
                        int dy = 0, dx = 0; if (c == 'N') dy = -1; if (c ==
'S') dy = 1; if (c == 'E') dx = 1; if (c == 'W') dx = -1;
                        g[n][i][0] = loc(y, x);
                        FOR(j, 1, MAXP-1) {
                                g[n][i][j] = loc(g[n][i][j-1].y+dy, g[n][i][j-
1].x+dx);
                                if (j%(p-1) == 0) dy *= -1, dx *= -1;
                        }
                }
        }

        q.push(state(s[0], s[1], 0)); mark(s[0], s[1], 0);
        while(!q.empty()) {
                state top = q.front(); q.pop();
                loc c[2]; c[0] = top.a, c[1] = top.b; int d = top.d;
                if (c[0].y == OUT && c[1].y == OUT) {
                        fprintf(fout, "%d\n", d);
                        return 0;
                }

                REP(i, 4) {
                        loc n[2]; REP(j, 2) {
                                if (map[j][c[j].y+dy[i]][c[j].x+dx[i]] == 1)
n[j] = c[j];
                                else n[j].y = c[j].y+dy[i], n[j].x =
c[j].x+dx[i];
                                if (c[j].y == OUT || n[j].y < 0 || n[j].y
>= R[j] || n[j].x < 0 || n[j].x >= C[j]) n[j] = loc(OUT, OUT);
                        }
                        int nd = d+1;
                        bool bad = false; REP(j, 2) if (n[j].y != OUT) REP(k,
G[j]) {
                                if (n[j] == g[j][k][nd%MAXP]) bad = true;
                                if (n[j] == g[j][k][d%MAXP] && c[j] ==
g[j][k][nd%MAXP]) bad = true;
                        }
                        if (!bad && mark(n[0], n[1], nd%MAXP))
q.push(state(n[0], n[1], nd));
                }
        }
        fprintf(fout, "-1\n");
        return 0;
}
```

3.Lilypad Pond [Richard Ho, 2006] (37, 36) FEB 07 Gold

FJ has installed a beautiful pond for his cows' aesthetic enjoyment and exercise.

The rectangular pond has been partitioned into square cells of M rows and N columns (1 <= M <= 30; 1 <= N <= 30). Some of the cells have astonishingly sturdy lilypads; others have rocks; the remainder contain just beautiful, cool, blue water.

Bessie is practicing her ballet moves by jumping from one lilypad to another and is currently located at one of the lilypads. She wants to travel to another lilypad in the pond by jumping from one lilypad to another.

Surprising only to the uninitiated, Bessie's jumps between lilypads always appear as a chess-knight's move: one move in one direction and then two more in the orthogonal direction (or perhaps two in one direction and then one in the orthogonal direction).

Farmer John is observing Bessie's ballet drill and realizes that sometimes she might not be able to jump to her destination lilypad because intermediary lilypads are missing.

Ever thrifty, he wants to place additional lilypads so she can complete her quest (perhaps quickly, perhaps by using a large number

of intermediate lilypads). Of course, lilypads cannot be placed where rocks already intrude on a cell.

Help Farmer John determine the minimum number of additional lilypads he has to place, and in how many ways he can place that minimum number.

PROBLEM NAME: lilypad

INPUT FORMAT:

* Line 1: Two space-separated integers: M and N

* Lines 2..M+1: Line i+1 describes row i of the pond using N space-separated integers with these values: 0 indicates empty water; 1 indicates a lilypad in place; 2 indicates rock in place; 3 indicates the lilypad Bessie starts on; 4 indicates the lilypad Bessie wants to travel to.

SAMPLE INPUT (file lilypad.in):

```
4 5
1 0 0 0 0
3 0 0 0 0
0 0 2 0 0
0 0 0 4 0
```

INPUT DETAILS:

The ponds has 4 rows and 5 columns. Bessie is at row 2 column 1, and she wants to reach row 4 column 4. A rock and three lilypads are present.

OUTPUT FORMAT:

* Line 1: One integer: the minimum number of additional lilypads required. If it is not possible to help Bessie jump to her destination, print -1.

* Line 2: One integer: the total number of possible ways the additional lilypads can be positioned. This number is guaranteed to fit into a single 64-bit signed integer. Do not output this line if line 1 contains -1.

SAMPLE OUTPUT (file lilypad.out):

```
2
3
```

OUTPUT DETAILS:

Two lilypads are required. There are three ways to place them: row 4 column 2 and row 2 column 3; row 1 column 3 and row 3 column 2; or row 1 column 3 and row 2 column 5:

```
R1C2,R2C3     R1C3,R3C2     R1C3,R2C5
1 0 0 0 0     1 0 X 0 0     1 0 X 0 0
3 0 X 0 0     3 0 0 0 0     3 0 0 0 X
```

```
0 0 2 0 0     0 X 2 0 0     0 0 2 0 0
0 X 0 4 0     0 0 0 4 0     0 0 0 4 0
```

Analysis: Lilypad Pond by Richard Peng

It suffices to consider the grid as a graph and the possible jumps as edges. Then the problem becomes: Given a graph where some vertices are marked, find the least number of additional vertices to mark so there exist a path between s and t.

First notice that each connected component can be 'shrinked' into one vertex which combines all the edges of the vertices it contains as we could walk around the component free of cost.

The standard way to go here is to construct some kind of weight on the edges to change the problem into finding the shortest path and counting the number of shortest paths. It follows that the distance should be the least number of vertices that needs to be marked

Clearly, any edge from an unmarked vertex to an unmarked vertex has cost one as another vertex needs to be marked. Things gets tricky in situations where a marked vertex is traversed, namely the following situations:
v1, v2 are marked, a and b are not. The following edges exist:

a->v1, a->v2, v1->b, v2->b
Then any kind of shortest path which involves v1 and v2 would end up calculating the same set of unmarked vertices (a and b) twice since there are two paths. Therefore, it's necessary to 'skip' marked vertices by having a path of length 1 directly from a to b.

It's quite clear that the least number of vertices to be marked equals the shortest distance from s to t minus one (the last edge to t should be free).

To get the answers, do a BFS from s, then go through each vertex in order and count the number of paths by summing the paths that lead to of all its neighbors which are distance d-1 away (assuming this vetex has distance d from s).

Here is 1000 point scorer Yang Yi's solution, which might or might not match the description above:

```
#include <stdio.h>
#include <queue>
#include <fstream>
#define MAXN 30

using namespace std;

FILE *in;
int fx[8] = {-2,-1,+1,+2,+2,+1,-1,-2};
int fy[8] = {-1,-2,-2,-1,+1,+2,+2,+1};
int m,n,sx,sy,ex,ey;
int data[MAXN][MAXN];
int rea[MAXN][MAXN][MAXN][MAXN];
int dist[MAXN][MAXN];
long long counter[MAXN][MAXN];
queue<pair<int, int> > Q;

void init () {
    int i, j;

    in = fopen("lilypad.in","r");
    fscanf(in,"%d%d",&m,&n);
    for (i = 0; i < m; i ++)
        for (j = 0; j < n; j ++) {
            fscanf(in,"%d",data[i] + j);
            if (data[i][j] == 3) {
                data[i][j] = 0;
                sx = i;
                sy = j;
                }
            if (data[i][j] == 4) {
                data[i][j] = 0;
                ex = i;
                ey = j;
                }
```

```
            }
    fclose(in);
    }

void solve () {
    int i, j, x, y;

    memset(rea, 0, sizeof(rea));
    for (i = 0; i < m; i ++)
        for (j = 0; j < n; j ++)
            if (data[i][j] == 0) {
//                printf("i j %d %d\n",i,j);
                Q.push(make_pair(i,j));
                rea[i][j][i][j] = 1;
                while (!Q.empty()) {
                    x=Q.front().first;
                    y=Q.front().second;
                    Q.pop();
                    for (int k = 0; k < 8; k ++)
                        if (x + fx[k] >= 0 && x + fx[k] < m && y + fy[k] >= 0
&& y + fy[k] < n && data[x+fx[k]][y+fy[k]] == 1 &&
rea[i][j][x+fx[k]][y+fy[k]] == 0) {
                            Q.push(make_pair(x + fx[k], y + fy[k]));
                            rea[i][j][x+fx[k]][y+fy[k]] = 1;
                            }
                    }
                for (x = 0; x < m; x ++)
                    for (y = 0; y < n; y ++)
                        if (rea[i][j][x][y] == 1) {
//                          printf("Lily Reachable %d %d\n",x,y);
                            for (int k = 0; k < 8; k ++)
                                if (x + fx[k] >= 0 && x + fx[k] < m && y +
fy[k] >= 0 && y + fy[k] < n && data[x+fx[k]][y+fy[k]] == 0 &&
rea[i][j][x+fx[k]][y+fy[k]] == 0)
                                    rea[i][j][x+fx[k]][y+fy[k]] = 2;
                            }
                }

    memset(dist, -1, sizeof(dist));
    dist[sx][sy] = 0;
    counter[sx][sy] = 1;
    Q.push(make_pair(sx,sy));
    while (!Q.empty()) {
        x=Q.front().first;
        y=Q.front().second;
        Q.pop();

//        printf("%d %d : %d %I64d\n",x,y,dist[x][y],counter[x][y]);

        for (i = 0; i < m; i ++)
            for (j = 0; j < n; j ++)
                if (rea[x][y][i][j] == 2) {
//                  printf("(%d %d) -> (%d %d)\n",x,y,i,j);
                    if (dist[i][j] == -1) {
                        dist[i][j] = dist[x][y] + 1;
                        counter[i][j] = counter[x][y];
                        Q.push(make_pair(i,j));
```

```
                        }
                    else
                        if (dist[i][j] == dist[x][y] + 1)
                            counter[i][j] += counter[x][y];
                    }
        }
//    while (1);
    }

void output () {
    ofstream fout("lilypad.out");

    if (dist[ex][ey] == -1)
        fout<<-1<<endl;
    else
        fout<<dist[ex][ey] - 1<<endl<<counter[ex][ey]<<endl;
    }

int main () {

    init();
    solve();
    output();

    return 0;
 }
```

4. Cow Imposters [Hal Burch, 2002] (38, 38) FEB 03 Green

Farmer John no longer uses the barbaric custom of branding to mark the cows that he owns. Instead, he creates a binary code of B (1 <= B <= 16) bits for each cow and embosses it onto a metal strip that is fastened to each cow's ear.

The cows have taken in a stray and wish to create an ID strip for it. Unknown to FJ, they have created a machine that can make a new ID strip by combining two existing ID strips using the XOR operation (ID strips are not consumed by this machine, and the same ID strip can be used for both inputs).

The cows wish to create a specific ID strip for the stray or at least get as close to a desired ID as possible -- with the smallest possible number of bits differing between the goal ID strip and the best possible new ID strip.

Given a set of E (1 <= E <= 100) existing ID strips, the goal ID strip, and a large number of blank ID strips to hold intermediate results, calculate the closest possible ID strip that can be created from the existing ID strips.

If more than one ID is closest, choose the one that can be created in the fewest steps. If there is still a tie, choose the `smallest' ID (i.e., if you sorted all the IDs, the one that is first).

Of course, to get a new strip you'll have to use the machine at

least once.

PROBLEM NAME: impster

INPUT FORMAT:

* Line 1: Two space-separated integers: B and E.
* Line 2: The goal ID string, represented as a string of B 0's and 1's
        (with no spaces).
* Lines 3..E+2: Each line contains an existing ID string, represented
        as a string of B 0's and 1's (with no spaces).

SAMPLE INPUT (file impster.in):
5 3
11100
10000
01000
00100

OUTPUT FORMAT:

* Line 1: A single integer that is the minimum number of steps
        required to create the best possible ID strip.
* Line 2: A single line with the best possible ID strip that can be
        created from the E existing ID strips

SAMPLE OUTPUT (file impster.out):

2
11100

Analysis: Cow Imposters by Ricahrd Peng

This problem is essentially path finding. Note that XOR is associative and there is never a point of combining two strings that are identical (unless we want to get 0). So the finished product can be constructed by adding the original bit vectors in one-by-one.

Now we treat all the possible strings as states and perform a BFS. Then we just
go through all the 'reachable' strings and break ties using distance (which is
also obtained from the BFS).

```
#include <iostream>
#include <string>
using namespace std;

int n,m,lis[100],q[100000],tai,dis[100000],target,siz,ans,rec,disa;

int getval(){
        int i,v;
        string s;
        cin>>s;
        for(v=i=0;i<n;i++)
                v=v*2+s[i]-'0';
        return v;
```

```
}

int dif(int x){
        int res,i;
        for(i=res=0;i<n;i++)    res+=(((x>>i)%2)==((target>>i)%2));
        return res;
}

void show(int x){
        int i;
        for(i=n-1;i>=0;i--)     cout<<((x>>i)%2);
        cout<<endl;
}

int main(){
        int i,j,a,tem;
freopen("impster.in","r",stdin);
freopen("impster.out","w",stdout);
        cin>>n>>m;
        target=getval();
        for(i=0;i<m;i++)        lis[i]=getval();
        siz=1<<n;
        for(i=0;i<siz;i++)      dis[i]=-1;
        for(i=tai=0;i<m;i++)
                for(j=0;j<m;j++)
                        if(dis[lis[i]^lis[j]]==-1){
                                q[tai++]=lis[i]^lis[j];
                                dis[lis[i]^lis[j]]=1;
                        }
        for(i=0;i<tai;i++)
                for(j=0;j<m;j++)
                        if(dis[a=q[i]^lis[j]]==-1){
                                dis[a]=dis[q[i]]+1;
                                q[tai++]=a;
                        }
//for(i=0;i<siz;i++)    if(dis[i]!=-1)
cout<<i<<" "<<dif(i)<<endl;
        for(ans=-1,i=0;i<siz;i++)
                if((dis[i]!=-1)&&(((tem=dif(i))>ans)||
((tem==ans)&&(dis[i]<dis[rec])))){
                        ans=tem;
                        rec=i;
                }
        disa=dis[rec];
        cout<<disa<<endl;
        show(rec);
        return 0;
}
```

ROW BY ROW DP

1. Corn Fields [Richard Ho, 2006] (32, 34) NOV06 Gold

Farmer John has purchased a lush new rectangular pasture composed of M by N (1 <= M <= 12; 1 <= N <= 12) square parcels. He wants to grow some yummy corn for the cows on a number of squares. Regrettably, some of the squares are infertile and can't be planted. Canny FJ knows that the cows dislike eating close to each other, so when choosing which squares to plant, he avoids choosing squares that are adjacent; no two chosen squares share an edge. He has not yet made the final choice as to which squares to plant.

Being a very open-minded man, Farmer John wants to consider all possible options for how to choose the squares for planting. He is so open-minded that he considers choosing no squares as a valid option! Please help Farmer John determine the number of ways he can choose the squares to plant.

PROBLEM NAME: cowfood

INPUT FORMAT:

```
* Line 1: Two space-separated integers: M and N

* Lines 2..M+1: Line i+1 describes row i of the pasture with N
        space-separated integers indicating whether a square is
        fertile (1 for fertile, 0  for infertile)

SAMPLE INPUT (file cowfood.in):

2 3
1 1 1
0 1 0

OUTPUT FORMAT:

* Line 1: One integer: the number of ways that FJ can choose the
        squares modulo 100,000,000.

SAMPLE OUTPUT (file cowfood.out):

9

OUTPUT DETAILS:

Number the squares as follows:

1 2 3
  4

There are four ways to plant only on one squares (1, 2, 3, or 4),
three ways to plant on two squares (13, 14, or 34), 1 way to plant
on three squares (134), and one way to plant on no squares. 4+3+1+1=9.
```

Analysis: Corn Fields by Richard Peng

This problem is a classical state compression DP problem. Since the forbidden component has length 2, only the configuration of the previous row determines whether the current row is valid. There are 2^n, which is worst case 4096 possible configurations per each row (the total is much lower as no two neighboring grids on the same row can both be planted) and its possible to transfer from one row to another if and only if there does not exist a column where both rows have occupied squares. Clever usage of bit operators can help reduce the amount of code.

The runtime of the code is O(2^(2n)*n) although in practice it's much less. In the original version of the problem, the exact result rather than the answer mod 10^8 was required, which would have made the problem a bit

more difficult to code.
Code (by Bruce Merry):

```
#include <fstream>
#include <algorithm>
#include <vector>
#include <iterator>

using namespace std;

#define MAXR 12
#define MAXC 12
#define MAX2C (1 << MAXC)
#define MOD 100000000

int main() {
    ifstream in("cowfood.in");
    ofstream out("cowfood.out");
    int infertile[MAXR + 1] = {0};
    int dp[MAXR + 1][MAX2C];
    int R, C, C2;
    vector<int> valid;

    in >> R >> C;
    C2 = 1 << C;
    fill(infertile, infertile + R, 0);
    for (int i = 0; i < R; i++)
        for (int j = 0; j < C; j++) {
            int x;
            in >> x;
            if (!x) infertile[i + 1] |= 1 << j;
        }

    for (int i = 0; i < C2; i++)
        if (!(i & (i << 1))) valid.push_back(i);

    fill(dp[0], dp[0] + C2, 1);
    for (int i = 1; i <= R; i++) {
        fill(dp[i], dp[i] + C2, 0);
        for (int j = 0; j < C2; j++) {
            if (~j & infertile[i]) continue;
            for (size_t k = 0; k < valid.size(); k++) {
                int u = valid[k];
                if (u & j) continue;
                dp[i][j] += dp[i - 1][u | infertile[i - 1]];
                dp[i][j] %= MOD;
            }
        }
    }

    out << dp[R][infertile[R]] << "\n";
    return 0;
}
```

Mahbub adds: You can solve it in O(R*C*2^n) as well.

This is left as home work :)

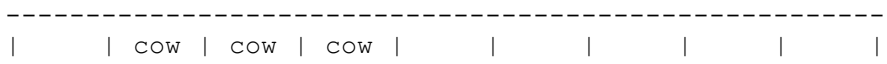
2. Lazy Cows [Brian Dean, 2004] (36, 31) OPEN05 Gold

Farmer John regrets having applied high-grade fertilizer to his pastures since the grass now grows so quickly that his cows no longer need to move around when they graze. As a result, the cows have grown quite large and lazy... and winter is approaching.

Farmer John wants to build a set of barns to provide shelter for his immobile cows and believes that he needs to build his barns around the cows based on their current locations since they won't walk to a barn, no matter how close or comfortable.

The cows' grazing pasture is represented by a 2 x B (1 <= B <= 15,000,000) array of cells, some of which contain a cow and some of which are empty. N (1 <= N <= 1000) cows occupy the cells in this pasture:

```
-------------------------------------------------------
|     | cow |     |     |     | cow | cow | cow | cow |
```



Ever the frugal agrarian, Farmer John would like to build a set of just K (1 <= K <= N) rectangular barns (oriented with walls parallel to the pasture's edges) whose total area covers the minimum possible number of cells. Each barn covers a rectangular group of cells in their entirety, and no two barns may overlap. Of course, the barns must cover all of the cells containing cows.

By way of example, in the picture above if K=2 then the optimal solution contains a 2x3 barn and a 1x4 barn and covers a total of 10 units of area.

PROBLEM NAME: lazy

INPUT FORMAT:

* Line 1: Three space-separated integers, N, K, and B.

* Lines 2..N+1: Two space-separated integers in the range (1,1) to (2,B) giving the coordinates of the cell containing each cow. No cell contains more than one cow.

SAMPLE INPUT (file lazy.in):

```
8 2 9
1 2
1 6
1 7
1 8
1 9
2 2
2 3
2 4
```

INPUT DETAILS:

As pictured above.

OUTPUT FORMAT:

* Line 1: The minimum area required by the K barns in order to cover all of the cows.

SAMPLE OUTPUT (file lazy.out):

```
10
```

OUTPUT DETAILS:

As discussed above.

Analysis: Lazy Cows by Richard Peng

Since the barn has 2 rows, the 'cross section' of any rectangle covering at a column which has a barn has to be one of 4 cases:

one row of height 1 (2 symmetric situations)

two rows of height 1

a single rectangle of height 2

So we can sort the columns with barns by their x values and DP on those, with our state being the column being considered and the number of rectangle used.

The transition is fairly tricky and is best worked out by hand and implemented
by if statements.

Here is a solution from *Mahbub* implementing above idea. Here

0 means rectangle available in lower row
1 means rectangle available in upper row
2 means seperate rectangles availble in both rows
3 combined rectangle available in both rows.

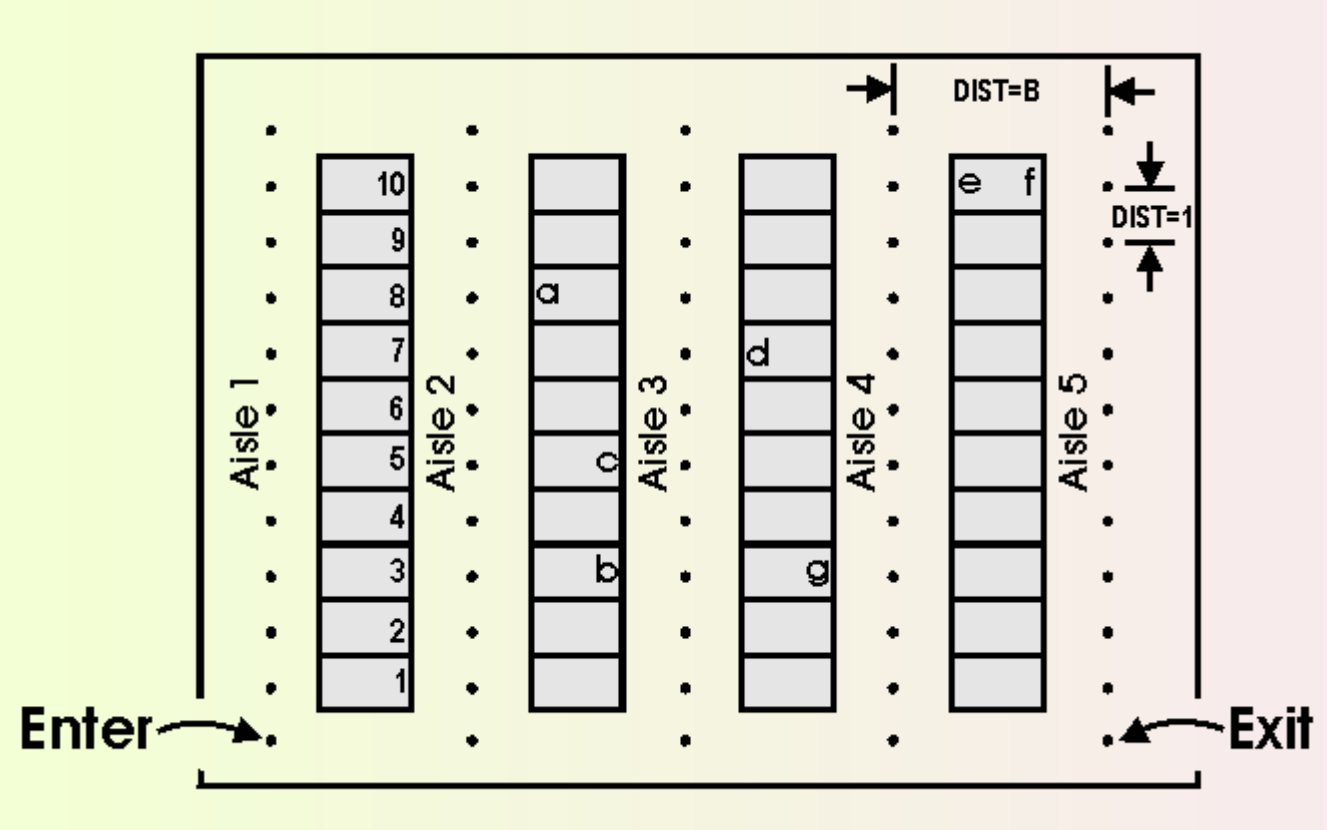
in[x][0] mean whether there is cow in position x and upper row, and in[x][1] means whether there is cow in position x and lower row.

```
/*
Prob ID: 0257
Name: lazy
Author: Md. Mahbubul Hasan
Date: 10th Janaury, 2008.
LANG: C++
*/
#include<stdio.h>
#include<utility>
#include<algorithm>
using namespace std;

#define MIN(A,B) ((A) < (B) ? (A) : (B))
#define INF 10000000
pair<int,int> cow[1002];
int ans[2][1002][4],in[1002][2],pos[1002];
int N,K,B,i,j,dif,sz,u,v,x1,x2,res,temp;

int main()
{
        freopen("lazy.in","r",stdin);
        freopen("lazy.out","w",stdout);

        scanf("%d%d%d",&N,&K,&B);
        for(i=0;i<N;i++)
                scanf("%d%d",&cow[i].second,&cow[i].first);
        sort(cow,cow+N);

        sz=0;
        for(i=0;i<N;i++)
        {
                if(!sz || pos[sz-1]!=cow[i].first) {pos[sz]=cow[i].first;
in[sz++][cow[i].second-1]=1;}
                else {in[sz-1][cow[i].second-1]=1;}
        }

        for(i=0;i<=K;i++) for(j=0;j<4;j++) ans[0][i][j]=INF;
        if(in[0][0]) ans[0][1][0]=INF; else ans[0][1][0]=1;
        if(in[0][1]) ans[0][1][1]=INF; else ans[0][1][1]=1;
        ans[0][2][2]=ans[0][1][3]=2;

        u=0;
        v=1;
        for(i=1;i<sz;i++)
        {
                dif=pos[i]-pos[i-1];
```

```
                for(j=1;j<=K;j++)
                {
                        if(j==1) x1=INF; else x1=MIN(MIN(ans[u][j-
1][0],ans[u][j-1][1]),MIN(ans[u][j-1][2],ans[u][j-1][3]));
                        if(j<=2) x2=INF;
                        else x2=MIN(MIN(ans[u][j-2][0],ans[u][j-
2][1]),MIN(ans[u][j-2][2],ans[u][j-2][3]));
                        if(!in[i][0])
ans[v][j][0]=MIN(x1+1,MIN(ans[u][j][0],ans[u][j][2])+dif); else
ans[v][j][0]=INF;
                        if(!in[i][1])
ans[v][j][1]=MIN(x1+1,MIN(ans[u][j][1],ans[u][j][2])+dif); else
ans[v][j][1]=INF;
                        if(j==1) temp=INF; else temp=MIN(ans[u][j-
1][0],ans[u][j-1][1]);
                        ans[v][j][2]=MIN(

MIN(x2+2,ans[u][j][2]+dif*2),
                                                        temp+dif+1
                                                        );
                        ans[v][j][3]=MIN(x1+2,ans[u][j][3]+2*dif);
                }


                u=1-u;
                v=1-v;
        }

        res=INF;
        for(i=1;i<=K;i++)
                for(j=0;j<4;j++)
                        res=MIN(res,ans[u][i][j]);
        printf("%d\n",res);

        return 0;
}
```

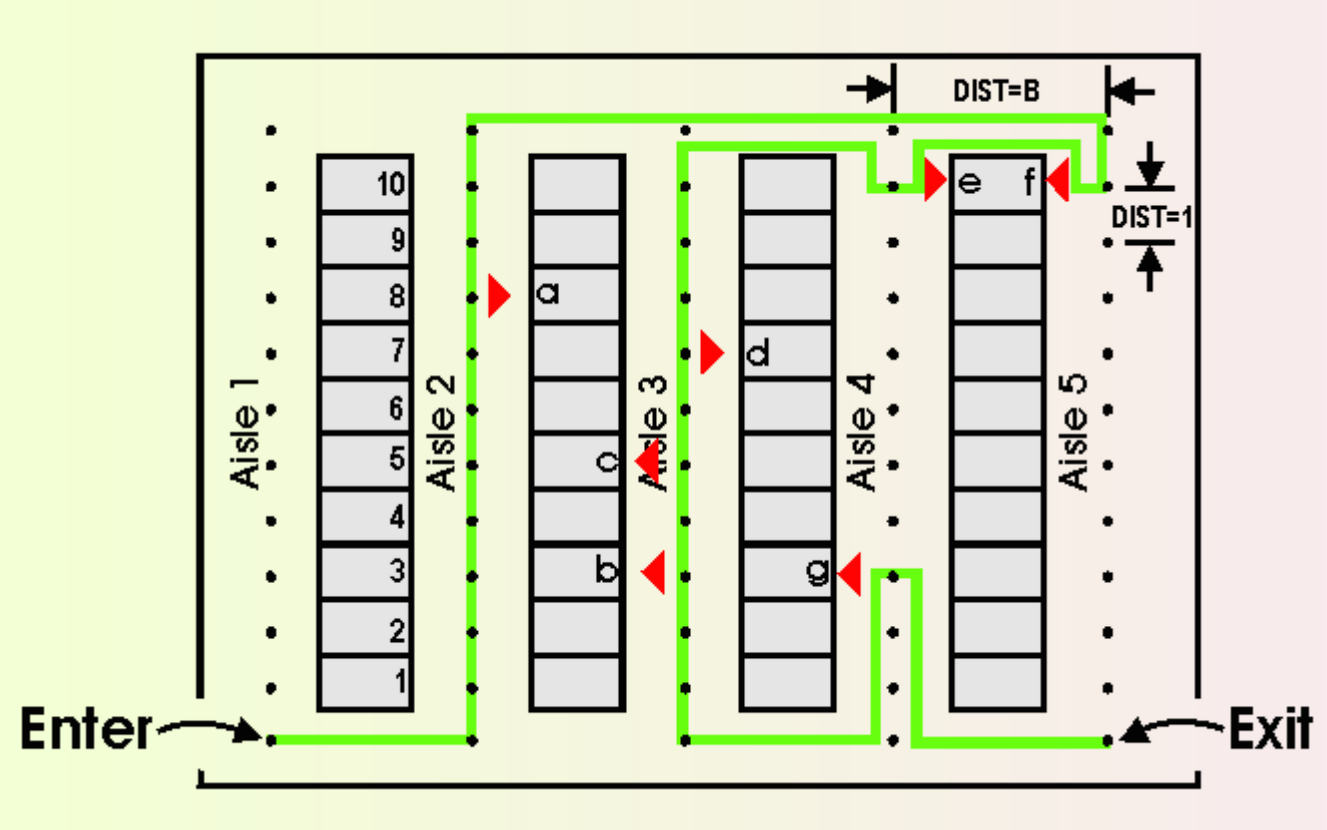
3. Problem XX: Grocery Store [Cox/Kolstad, 2006] (46, 39) Camp 06 Day 3

Bessie has gone to the grocery store with her list of P (1 <= P <= 300) items to purchase. This grocery store is laid out like so many grocery stores with N aisles (1 <= N <= 350). Below is a diagram showing not only the layout of a store with N=5 aisles but also the location of seven items to purchase, the entrance, the length A of each aisle (1 <= A <= 25), and the distance B from the center of one isle to the center of the next aisle (1 <= B <= 5). The dots represent the places Bessie might stop; many adjacent dots are distance 1 apart.



Bessie wants to minimize her walking distance to retrieve all the items she wishes to purchase and then check out at the lower-right corner.

Given A, B, and a list of P items to purchase (each item described by its aisle number and distance from the 'beginning' of the aisle [nearest the entrance]), calculate the minimum distance Bessie must travel. Note that one step is required to enter a given vertical aisle or leave a vertical aisle (to go to/from the right-left traversal aisles). See the example below. Here is a sample route (which might or might not be optimal) Bessie might consider:



For the diagram above, A=10 (which means distance 11 from lower aisle to upper aisle) and B=3. From the entrance to item 'a' is length B + 8 = 11. 3 more steps to the upper cross-aisle (not surprisingly a total of P+1=11 to traverse the entire aisle). 3*B=9 to aisle 5; 1 more to item f. Turning around, 1 more back to upper cross-aisle, B=3 more to aisle 4, 1 down to get item 'e'. 1 back up then B=3 left to aisle 3. Traverse 11 down aisle 3, picking up 'd', 'c', and 'b' along the way. B=3 right to Aisle 4. 3 up, get item 'g', 3 down, B=3 right to the exit. Total = 56 steps.

PROBLEM NAME: groc

INPUT FORMAT:

* Line 1: Four space-separated integers: P, N, A, and B

* Lines 2..P+1: Each line contains two space-separated integers that describe the location of an item to buy, respectively the aisle number and distance from the 'beginning' of that aisle

SAMPLE INPUT (file groc.in):

```
7 5 10 3
2 8
3 3
3 5
3 7
4 10
```

```
5 10
4 3
```

OUTPUT FORMAT:

* Line 1: A single integer that is the smallest distance Bessie must travel to obtain all her items.

SAMPLE OUTPUT (file groc.out):

```
54
```

TREE BASED DP

1. Cell Phone Network [Jeffrey Wang, 2007] (28, 35) JAN08 Gold

Farmer John has decided to give each of his cows a cell phone in hopes to encourage their social interaction. This, however, requires him to set up cell phone towers on his N (1 <= N <= 10,000) pastures (conveniently numbered 1..N) so they can all communicate.

Exactly N-1 pairs of pastures are adjacent, and for any two pastures A and B (1 <= A <= N; 1 <= B <= N; A != B) there is a sequence of adjacent pastures such that A is the first pasture in the sequence and B is the last. Farmer John can only place cell phone towers in the pastures, and each tower has enough range to provide service to the pasture it is on and all pastures adjacent to the pasture with the cell tower.

Help him determine the minimum number of towers he must install to provide cell phone service to each pasture.

PROBLEM NAME: tower

INPUT FORMAT:

* Line 1: A single integer: N

* Lines 2..N: Each line specifies a pair of adjacent pastures with two space-separated integers: A and B

SAMPLE INPUT (file tower.in):

```
5
1 3
5 2
4 3
3 5
```

INPUT DETAILS:

Farmer John has 5 pastures: pastures 1 and 3 are adjacent, as are pastures 5 and 2, pastures 4 and 3, and pastures 3 and 5. Geometrically, the farm looks like this (or some similar configuration)

```
   4  2
   |  |
1--3--5
```

OUTPUT FORMAT:

* Line 1: A single integer indicating the minimum number of towers to install

SAMPLE OUTPUT (file tower.out):

```
2
```

OUTPUT DETAILS:

The towers can be placed at pastures 2 and 3 or pastures 3 and 5.

Analysis: Cell Phone Network by Jeffrey Wang

This problem, known as the "minimum dominating set problem" in a tree, has a nice greedy solution. Note that the graph in the problem is indeed a tree because it has V-1 edges. We say a vertex is dominated by a set of vertices if it is either in the set or adjacent to a vertex in the set. The algorithm is as follows:

(1) Keep track of a set of vertices S.
(2) Root the tree at an arbitrary node.
(3) Visit the vertices of the tree with a postorder traversal.
(4) At each vertex, if it is not dominated by S, add it and all of its neighbors to S.
(5) Once the traversal is done, visit the vertices in S in the reverse order that they were added.
(6) At each vertex, if S is still a dominating set when the vertex is removed, then do so.

The vertices left in S form a minimum dominating set. For a proof of why this works, see http://www.cs.clemson.edu/~bcdean/greedy_tree_dom.swf.

Note that the running time of this algorithm is very good, O(N), since each vertex is visited once in the postorder traversal and each vertex is added to S at most once. Checking whether a vertex can be removed can also be done efficiently by keeping track of how many vertices in S it is adjacent to. An implementation of this algorithm is shown below.

```
#include <stdio.h>
#include <iostream>
#include <vector>

using namespace std;

int N, c[10100], p[10100];
vector <vector <int> > E;
vector <int> B;

void traverse(int v)
{
     for(int i=0; i<E[v].size(); i++)
         if(p[v] != E[v][i]) { p[E[v][i]] = v; traverse(E[v][i]); }

     if(c[v] == 0)
     {
         B.push_back(v); c[v]++;
         for(int i=0; i<E[v].size(); i++)
         {
             c[E[v][i]] += 2; B.push_back(E[v][i]);
             for(int j=0; j<E[E[v][i]].size(); j++) c[E[E[v][i]][j]]++;
```

```
        }
    }
}

int main()
{
    FILE* in = fopen("tower.in", "r");
    FILE* out = fopen("tower.out", "w");

    fscanf(in, "%d", &N);
    E.resize(N);
    for(int i=0; i<N; i++) { p[i] = -1; c[i] = 0; }

    int u, v;

    for(int i=0; i<N-1; i++)
    {
        fscanf(in, "%d %d", &u, &v);
        E[u-1].push_back(v-1);
        E[v-1].push_back(u-1);
    }

    traverse(0);

    int numRemove = 0;
    for(int i=B.size()-1; i>=0; i--)
    {
        bool canRemove = (c[B[i]] > 1);
        for(int j=0; j<E[B[i]].size(); j++)
            canRemove = canRemove && (c[E[B[i]][j]] > 1);
        if(canRemove)
        {
            c[B[i]]--;
            for(int j=0; j<E[B[i]].size(); j++) c[E[B[i]][j]]--;
        }
        numRemove += canRemove;
    }

    fprintf(out, "%d\n", B.size()-numRemove);
}
```

USA's Michael Cohen writes:

There is actually another simple O(N) solution to this problem. The problem states that every vertex has a tower or is adjacent to a vertex with a tower; since the graph is a tree (it is connected and has N-1 edges) this is equivalent to the statement that for any vertex there is a tower on that vertex, its parent, or one of its children.

If the tree is divided into subtrees, the only connections that exists in the tree as a whole but not the subtrees are those between the root of the tree and the roots of the subtrees. This means that only the roots of the subtrees

need to be considered when merging subtrees into the tree as a whole. By this logic, there are 3 possible types of valid subtrees:

- Subtrees without a tower on their root or any of its immediate children. These subtrees are only valid if their parent has a tower on its root.
- Subtrees without a tower on their root but with a tower on at least one immediate child. These subtrees are always valid, but do not give tower coverage to their parent.
- Subtrees with a tower on their root. These subtrees are always valid and are sufficient to cover their parent as well.

If we know for each subtree the overall smallest number of towers to make that subtree valid, the smallest number of towers to make it valid and of type 2 or type 3, and the smallest number of towers to make it a valid type 3 tower, we can then generate these numbers for the whole tree. The problem can then be solved by recursively subdividing the tree (a valid entire tree is a valid subtree of type 2 or 3), and since each vertex and each edge will be processed only one, it is linear time.

2.Milk Team Select [Percy Liang, 2002] (38, 36) MAR 06 Gold

Farmer John's N (1 <= N <= 500) cows are trying to select the milking team for the world-famous Multistate Milking Match-up (MMM) competition. As you probably know, any team that produces at least X (1 <= X <= 1,000,000) gallons of milk is a winner.

Each cow has the potential of contributing between -10,000 and 10,000 gallons of milk. (Sadly, some cows have a tendency to knock over jugs containing milk produced by other cows.)

The MMM prides itself on promoting family values. FJ's cows have no doubt that they can produce X gallons of milk and win the contest, but to support the contest's spirit, they want to send a team with as many parent-child relationships as possible (while still producing at least X gallons of milk). Not surprisingly, all the cows on FJ's farm are female.

Given the family tree of FJ's cows and the amount of milk that each would contribute, compute the maximum number of parent-child relationships that can exist in a winning team. Note that a set of cows with a grandmother-mother-daughter combination has two parent-child relationships (grandmother-mother, mother-daughter).

PROBLEM NAME: tselect

INPUT FORMAT:

* Line 1: Two space-separated integers, N and X.

* Lines 2..N+1: Line i+1 contains two space-separated integers describing cow i. The first integer is the number of gallons of milk cow i would contribute. The second integer (range 1..N) is the index of the cow's mother. If the cow's mother is unknown, the second number is 0. The family information has no cycles: no cow is her own mother, grandmother, etc.

SAMPLE INPUT (file tselect.in):

```
5 8
-1 0
3 1
5 1
-3 3
2 0
```

INPUT DETAILS:

There are 5 cows. Cow 1 can produce -1 gallons and has two daughters, cow 2 and 3, who can produce 3 and 5 gallons, respectively. Cow 3 has a daughter (cow 4) who can produce -3 gallons. Then there's cow 5, who can produce 2 gallons.

OUTPUT FORMAT:

* Line 1: The maximum number of parent-child relationships possible on

a winning team. Print -1 if no team can win.

SAMPLE OUTPUT (file tselect.out):

2

OUTPUT DETAILS:

The best team consists of cows 1, 2, 3, and 5. Together they produce (-1)+3+5+2 = 9 >= 8 gallons and have 2 parent-child relationships (1--2 and 1--3). Note that a team with cows 2, 3, and 5 would be able to produce more milk (10 gallons), but would have fewer parent-child relationships (0).

USACO March 2006 Problem 'tselect' Analysis

by Bruce Merry

This is quite a complex dynamic programming task. We use dynamic programming to determine, for a given number of relationships, the maximum milk production. The output answer is then the smaller number of relationships for which this maximum production is at least X.

DP would naturally lead us to consider the following subproblems: given a particular subtree and number of relationships, determine the maximum milk production. This does not give us enough information to immediately and efficiently solve the problem for a larger tree, because there are exponentially many ways to distribution the relationships amongst the subtrees. However, we can use dynamic programming on this problem, too, since it is essentially a knapsack problem. We iterate over the child trees, updating an array indicating the maximum milk production for that child tree and all previous child trees.

A slight complication is that if we use the root cow, we also have to count the relationships between her and her children. To facilitate this, we solve the DP subproblems separate for the case that the root is used and that the root is not used.

The running time is $O(N^3)$ in the worse case, although some intelligent coding ensures that the constant factor is quite small and thus it is possible to solve the largest test cases in time.

Below is a solution from Romania's Adrian Diaconu that might or might not match the description above.

```
#include <stdio.h>
#include <string.h>
#define NMAX 511
#define LMAX 2100
#define INF 5110111

int n, x, par[NMAX], val[NMAX], nr[NMAX], urm[NMAX], z, rez,
    sol[NMAX][2][NMAX], v[NMAX], aux[2][NMAX];

void Vadd(int i,int j) {
    urm[++z] = v[i];
    v[i] = z;
    nr[z] = j;
}

void parc(int i) {
    int j, g, k, t, r=(i!=0);
```

```
    for(j = v[i]; j; j = urm[j])
        parc(nr[j]);
    for(sol[i][0][0] = 0, sol[i][1][0] = val[i], j = 1;  j <= n; j++)
        sol[i][0][j] = sol[i][1][j] = -INF;
    for(j=v[i]; j; j=urm[j]) {
        memcpy(aux, sol[i], sizeof(sol[i]));
        for(g=nr[j], k=0; k<=n; k++)
            if(sol[g][1][k]>-INF||sol[g][0][k]>-INF)
                for(t=0;t+k<=n;++t) {
                    if (aux[0][t] > -INF && aux[0][t]+sol[g][0][k] >
sol[i][0][t+k])
                        sol[i][0][t+k] = aux[0][t] + sol[g][0][k];
                    if (aux[0][t] > -INF && aux[0][t]+sol[g][1][k] >
sol[i][0][t+k])
                        sol[i][0][t+k] = aux[0][t]+sol[g][1][k];
                    if(aux[1][t] > -INF && aux[1][t]+sol[g][0][k] >
sol[i][1][t+k])
                        sol[i][1][t+k] = aux[1][t]+sol[g][0][k];
                    if(t+k+r<=n)
                        if(aux[1][t] > -INF && aux[1][t]+sol[g][1][k] >
sol[i][1][t+k+r])
                            sol[i][1][t+k+r] = aux[1][t]+sol[g][1][k];
                }
    }
}

int main() {
    FILE *fi=fopen("tselect.in","r");
    FILE *fo=fopen("tselect.out","w");
    fscanf (fi, "%d %d", &n, &x);
    int i;
    for(i=1; i<=n; i++) {
        fscanf(fi, "%d %d", &val[i], &par[i]);
       Vadd(par[i],i);
    }
    fclose(fi);
    parc(0);
    for(rez=-1,i=0;i<=n;++i)
        if(sol[0][0][i] >= x || sol[0][1][i] >= x)
            rez=i;
    fprintf(fo,"%d\n",rez);
    fclose(fo);
    return(0);
}
```

3.Rivers [IOI, 2005] (40, 50)

Nearly all of the Kingdom of Byteland is covered by forests and rivers. Small rivers meet to form bigger rivers, which also meet and, in the end, all the rivers flow together into one big river. The big river meets the sea near Bytetown.

There are N (2 <= N <= 100) lumberjack's villages (conveniently number 1..N) in Byteland (which is village number 0), each placed near a river. Currently, there is a big sawmill in Bytetown that processes all trees cut in the Kingdom. The trees float from the villages down the rivers to the sawmill in Bytetown. The king of Byteland decided to build K (1 <= K <= 50; K <= N) additional sawmills in villages to reduce the cost of transporting the trees downriver. After building the sawmills, the trees need not float to Bytetown, but can be processed in the first sawmill they encounter downriver. Obviously, the trees cut near a village with a sawmill need not be transported by river. It should be noted that the rivers in Byteland do not fork. Therefore, for each village, there is a unique way downriver from the village to Bytetown.

The king's accountants calculated how many trees are cut by each village per year. You must decide where to build the sawmills to minimize the total cost of transporting the trees per year. River transportation costs one cent per kilometre, per tree.

Write a program that:

* Reads from the input file the number of villages, the number of additional sawmills to be built, the number of trees cut near each village, and descriptions of the rivers,

* Calculates the minimal cost of river transportation after building additional sawmills,

* Writes the result to the output file.

PROBLEM NAME: rivers

INPUT FORMAT:

* Line 1: Two integers: N and K

* Lines 2..N+1: Line i+1 contains three space-separated integers: W_i, V_i, and D_i.

* W_i: The number of trees cut near village i per year (0 <= W_i <= 10,000),

* V_i: The first village (or Bytetown) downriver from village i (0 <= V_i <= N),

* D_i: The distance in kilometres by river from village i to V_i (1 <= D_i <= 10,000).

SAMPLE INPUT (file rivers.in):

```
4 2
```

```
1 0 1
1 1 10
10 2 5
1 2 3
```

OUTPUT FORMAT:

* Line 1: A single integer that is the minimal cost of river transportation in cents.

SAMPLE OUTPUT (file rivers.out):

```
4
```

OUTPUT DETAILS:

The sawmills should be built in villages 2 and 3.

Analysis: Rivers by original analysis from IOI

It is not hard to observe that since for each village there is a unique way down the river from the village to Bytetown, we can treat the rivers and villages as a tree with the root in Bytetown. Nodes of the tree correspond to the villages (for convenience we will refer to Bytetown as a village too), and
node v is the parent of node u when v is the first village downriver from u.

Let r denote the root of the tree, i.e. r corresponds to Bytetown. By depth(u)
we will denote the number of edges on a unique path from u to r. Clearly, the values of depth(u) can be computed for all villages u in linear time. The number of children of node u will be denoted by deg(u), and the number of trees
cut near village u will be denoted by trees(u).

We can apply dynamic programming to solve the task. Let Av,t,l denote the minimal cost of transportation of the trees cut in the subtree rooted in v, assuming that t additional sawmills can be built in the subtree, and the trees
not processed in v can be processed in the village of depth l (on the way from
v to Bytetown). We compute values of Av,t,l for each village v, and such numbers t, l, that 0 <= t <= k and 0 <= l < depth(v). Clearly, when the tree rooted in v has at most t nodes, then A(v,t,l) = 0, as we can simply place a sawmill in every village. We can use the following formula:

A(v,t,l)=0 when the tree rooted in v has at most t nodes, min(A'v,t,l ,A''v,t)
otherwise.

where A'v,t,l is the cost of transportation when there is no sawmill in v, and
A''v0,t is the cost of transportation when there is one. These costs depend on
the distribution of sawmills between subtrees rooted in children of v. Let d =
deg(v) and let v1,v2, . . . ,vd be the children of v. Then:

$$A0v,t,l = trees(v) \cdot (depth(v)-l) + \min_{t1+...+td=t} sum(i=1,d,Av\_i,t\_i,l)$$

$$A0v0,t = \min_{t1+...+td=t-1} sum(i=1,d,Av\_i,t\_i,depth(v))$$

The analysis then talked about DP one more time, however, this can be thought as considering one more tree at a time and merging those results.

Here is a sample solution from *Mahbub* :

```
/*
Prob ID: 0853
```

```
Name: rivers
Author: Md. Mahbubul Hasan
Date: 21st January, 2008.
LANG: C++
*/
#include<stdio.h>
#include<vector>
using namespace std;

#define INF 1000000000
#define MIN(A,B) ((A) < (B) ? (A) : (B))

int done[102][102];
int dp[102][51][102];
vector<int> V[102];
int N,K;
int temp[2][52];
int tree[102],p[102],weight[102];
int dist[102][102];

void DP(int at,int depth)
{
        int sz,u,v,i,j,x;

        if(done[at][depth]) return;

        done[at][depth]=1;

        sz=V[at].size();
        for(i=0;i<sz;i++)
        {
                DP(V[at][i],depth+1);
                DP(V[at][i],1);
        }

        for(i=0;i<=K;i++) temp[0][i]=INF;
        temp[0][0]=0;

        u=0;
        v=1;
        for(x=0;x<sz;x++)
        {
                for(i=0;i<=K;i++) temp[v][i]=INF;

                for(i=0;i<=K;i++)
                        for(j=0;j<=K-i;j++)

temp[v][i+j]=MIN(temp[v][i+j],temp[u][i]+dp[V[at][x]][j][depth+1]);

                u=1-u;
                v=1-v;
        }

        for(i=0;i<=K;i++)
        {
                dp[at][i][depth]=temp[u][i]+tree[at]*dist[at][depth];
                temp[0][i]=INF;
```

```
        }

        temp[0][1]=0;

        u=0;
        v=1;
        for(x=0;x<sz;x++)
        {
                for(i=0;i<=K;i++) temp[v][i]=INF;

                for(i=0;i<=K;i++)
                        for(j=0;i+j<=K;j++)

temp[v][i+j]=MIN(temp[v][i+j],temp[u][i]+dp[V[at][x]][j][1]);

                u=1-u;
                v=1-v;
        }

        for(i=0;i<=K;i++)
                dp[at][i][depth]=MIN(dp[at][i][depth],temp[u][i]);
}

int main()
{
        freopen("rivers.in","r",stdin);
        freopen("rivers.out","w",stdout);

        int i,j,cnt,d,x;

        scanf("%d%d",&N,&K);

        for(i=1;i<=N;i++)
        {
                scanf("%d%d%d",&tree[i],&p[i],&weight[i]);
                V[p[i]].push_back(i);
        }

        for(i=1;i<=N;i++)
        {
                for(j=1;j<=N;j++) dist[i][j]=INF;
                cnt=d=0;
                x=i;
                while(x)
                {
                        d+=weight[x];
                        dist[i][++cnt]=d;
                        x=p[x];
                }
        }

        DP(0,0);

        printf("%d\n",dp[0][K][0]);

        return 0;
}
```

4. Landscaping [Brian Dean, 2005] (41, 39) OPEN05 GOLD

Farmer John is making the difficult transition from raising mountain goats to raising cows. His farm, while ideal for mountain goats, is far too mountainous for cattle and thus needs to be flattened out a bit. Since flattening is an expensive operation, he wants to remove the smallest amount of earth possible.

The farm is long and narrow and is described in a sort of two-dimensional profile by a single array of N (1 <= N <= 1000) integer elevations (range 1..1,000,000) like this:

1 2 3 3 3 2 1 3 2 2 1 2,

which represents the farm's elevations in profile, depicted below with asterisks indicating the heights:

```
    * * *     *
  * * * * *   * * *   *
* * * * * * * * * * * *
1 2 3 3 3 2 1 3 2 2 1 2
```

A contiguous range of one or more equal elevations in this array is a "peak" if both the left and right hand sides of the range are either the boundary of the array or an element that is lower in elevation than the peak. The example above has three peaks.

Determine the minimum volume of earth (each unit elevation reduction counts as one unit of volume) that must be removed so that the resulting landscape has no more than K (1 <= K <= 25) peaks. Note well that elevations can be reduced but can never be increased.

If the example above is to be reduced to 1 peak, the optimal solution is to remove 2 + 1 + 1 + 1 = 5 units of earth to obtain this set of elevations:

```
    * * *     -
  * * * * *   - - -   -
* * * * * * * * * * * *
1 2 3 3 3 2 1 1 1 1 1 1
```

where '-'s indicate removed earth.

PROBLEM NAME: peaks

INPUT FORMAT:

* Line 1: Two space-separated integers: N and K

* Lines 2..N+1: Each line contains a single integer elevation. Line
        i+1 contains the elevation for index i.

SAMPLE INPUT (file peaks.in):

```
12 1
1
2
3
3
3
2
1
3
2
2
1
2
```

INPUT DETAILS:

This is the example used above.

OUTPUT FORMAT:

* Line 1: The minimum volume of earth that must be removed to reduce
        the number of peaks to K.

SAMPLE OUTPUT (file peaks.out):

```
5
```

Analysis: Landscaping by Richard Peng

If we turn the figure upside down, we get something looking like this:

```
1 2 3 3 3 2 1 3 2 2 1 2
* * * * * * * * * * * *
  * * * * *   * * *   *
    * * *     *
```

And if we create a node for each line contiguous horizontal segment of *s, we get :

```
    +
  / | \
 +  +  +
 |  |
 +  +
```

Which is a tree. It's easy to verify this claim for the way the peaks are defined in the problem. So now the problem becomes: given a tree, remove the a
set of vertex such that k leafs remain and minimize the sum of the weights of the leafs removed.

This can be done using DP, with the state being the vertex and the number of leafs left in its subtree. Combining two subtrees is equivalent to distributing
the leafs in them so the sum of the two states is minimum while special care must be taken in cases where the current node becomes a leaf.

Another issue is that the tree can have a huge number of vertices since the elevations can be up to 1,000,000. However, notice the original tree can only have up to N leafs, so most of the nodes will be on paths, which allows us to combine them. This can be done in the tree creation process as we only add up to the height of the least plateau in that range when we create a new vertex, with its weight equaling to the area of the rectangle.

5. Training [IOI, 2007]

Mirko and Slavko are training hard for the annual tandem bicycling marathon taking place in Croatia. They need to choose a training route from the N (2 <= N <= 1,000) cities conveniently numbered 1..N and M (N-1 <= M <= 5,000) roads in their country.

Each road connects two different cities and can be traversed in both directions. Exactly N-1 of those roads are paved, while the rest are unpaved trails. Fortunately, the network of roads was designed so that each pair of cities is connected by some sequence of paved roads. In other words, the N cities and the N-1 paved roads form a tree structure.

Additionally, each city is an endpoint for as many as 10 roads.

A training route starts in some city, follows some roads and ends in the same city it started in. Mirko and Slavko like to see new places, so they made a rule never to go through the same city nor travel the same road twice. The training route may start in any city and does not need to visit every city.

Riding in the back seat is easier, since the rider is shielded from the wind by the rider in the front. Because of this, Mirko and Slavko change seats in every city. To ensure that they get the same amount of training, they must choose a route with an even number of roads.

Mirko and Slavko's competitors have decided to block some of the unpaved roads to make it impossible to find a training route satisfying the above requirements. For each unpaved road i that connects cities C1_i and C2_i (1 <= C1_i <= N, 1 <= C2_i <= N) there is a B_i (0 <= B_i <= 10,000) associated with blocking the road (B_i==0 means the road is paved). It is not possible to block paved roads.

Write a program that, given the description of the network of cities and roads, finds the smallest total cost needed to block the roads so that no training route can satisfy the above requirements.

PROBLEM NAME: training

INPUT FORMAT:

* Line 1: Two space-separated integers: N and M

* Lines 2..M+1: Line i+1 describes road i with three space-separated
        integers C1_i, C2_i, and B_i.

SAMPLE INPUT (file training.in):

```
5 8
2 1 0
3 2 0
4 3 0
5 4 0
1 3 2
3 5 2
2 4 5
2 5 1
```

OUTPUT FORMAT:

* Line 1: A single integer, the smallest total cost as described in
        the problem statement.

SAMPLE OUTPUT (file training.out):

```
5
```

OUTPUT DETAILS:

Mirko and Slavko have five possible routes. If the roads 1-3, 3-5
and 2-5 are blocked, then Mirko and Slavko cannot use any of the
five routes. The cost of blocking these three roads is 5. It is
also possible to block just two roads, 2-4 and 2-5, but this would
result in a higher cost of 6.

Analysis: Training by original analysis from IOI

Detecting an odd cycle in a graph is a well-known problem. A graph does not contain an odd cycle if and only if it is bipartite. On the other hand, the problem of detecting an even cycle in a graph is not widely known.

We are given a graph consisting of N vertices and M edges. Exactly N−1 edges are marked as tree edges and they form a tree. An edge that is not a tree edge will be called a non-tree edge. Every non-tree edge e has a weight w(e) associated with it.

The task asks us to find a minimum-weighted set of non-tree edges whose removal results in a graph that does not contain a cycle of even length. We will call such a cycle an even cycle. Reasoning backwards, starting from a graph containing tree edges only, we have to find a maximum-weighted set of non-tree edges that can be added to the graph without forming any even cycles.

In order to describe the model solution, we first need to make a few observations about the structure of the graph we are working with.

Even and odd edges

Consider a non-tree edge e={A, B}. We define the tree path of the edge e to be the unique path from A to B consisting of tree edges only. If the length of the tree path is even, we say that e is an even edge; otherwise we say that e is an odd edge. We will use TP(e) to denote the tree path of an edge e.

Obviously, any odd edge present in the graph together with its tree path forms an even cycle. Therefore, we can never include an odd edge in our graph and we can completely ignore them.

Relation between two even edges

Individual even edges may exist in the graph. However, if we include several even edges, an even cycle might be formed. More precisely, if e1 and e2 are even edges such that TP(e1) and TP(e2) share a common tree edge, then adding both e1 and e2 to the graph necessarily creates an even cycle.

In order to sketch the proof of this claim, consider the two odd cycles created by e1 and e2 together with their respective tree paths. If we remove all common tree edges from those cycles we get two paths P1 and P2. The parity of P1 is equal to the parity of the P2 since we removed the same number of edges from the two initial odd cycles. As P1 and P2 also have the same endpoints, we can merge them into one big even cycle.

Tree edges contained in odd cycles

As a direct consequence of the previous claim, we can conclude that every tree edge may be contained in at most one odd cycle.

Conversely, if we add only even edges to the tree in such a way that every tree edge is contained in at most one odd cycle, then we couldn't have formed any even cycles. We briefly sketch the proof of this claim here. If an even cycle existed, it would have to contain one or more non-tree edges. Informally, if it contains exactly one non-tree edge we have a contradiction with the assumption that only even edges are added; if it contains two or more non-tree edges then we will arrive at a contradiction with the second assumption.

Model solution

Now, we can use our observations to develop a dynamic programming solution for the problem. A state is a subtree of the given tree. For each state we calculate the weight of the maximum-weighted set of even edges that can be added to the subtree while maintaining the property that each tree edge is contained in at most one odd cycle. The solution for the task is the weight associated with the state representing the initial tree.

To obtain a recursive relation, we consider all even edges with tree paths passing through the root of the tree.

We can choose to do one of the following:

1. We do not add any even edge whose tree path passes through the root of the tree. In this case, we can delete the root and proceed to calculate the optimal solution for each of the subtrees obtained after deleting the root node.

2. We choose an even edge e whose tree path passes through the root of the tree and add it to the tree. Next, we delete all tree edges along TP(e) (since, now, they are contained in one odd cycle), and, finally, we proceed to calculate the optimal solution for each of the subtrees obtained after deleting the tree path. Add w(e) to the total sum.

Because of the way the trees are decomposed, all subtrees that appear as subproblems can be represented with an integer and a bit mask. The integer represents the index of the subtree's root node, while the bit mask represents which of the root node's children are removed from the subtree.

The total number of possible states is, therefore, bounded by $N*2^K$ where K is the maximum degree of a node. Depending on the implementation details, the time complexity of the algorithm can vary. The official implementation has time complexity $O(M \log M + MN + M * 2^K)$.

GEOMETRIC DP PROBLEMS

1. Ski Lift [Adam Rosenfield, 2004] (31, 36) MAR06 Gold

Farmer Ron in Colorado is building a ski resort for his cows (though budget constraints dictate construction of just one ski lift). The lift will be constructed as a monorail and will connect a concrete support at the starting location to the support at the ending location via some number of intermediate supports, each of height 0 above its land. A straight-line segment of steel connects every pair of adjacent supports. For obvious reasons, each section of straight steel must lie above the ground at all points.

Always frugal, FR wants to minimize the number of supports that he must build. He has surveyed the N (2 <= N <= 5,000) equal-sized plots of land the lift will traverse and recorded the integral height H (0 <= H <= 1,000,000,000) of each plot. Safety regulations require FR to build adjacent supports no more than K (1 <= K <= N - 1) units apart. The steel between each pair of supports is rigid and forms a straight line from one support to the next.

Help FR compute the smallest number of supports required such that: each segment of steel lies entirely above (or just tangent to) each piece of ground, no two consecutive supports are more than K units apart horizontally, and a support resides both on the first plot of land and on the last plot of land.

PROBLEM NAME: skilift

INPUT FORMAT:

* Line 1: Two space-separate integers, N and K

* Lines 2..N+1: Line i+1 contains a single integer that is the height of plot i.

SAMPLE INPUT (file skilift.in):

```
13 4
0
1
0
2
4
```

6
8
6
8
8
9
11
12

OUTPUT FORMAT:

* Line 1: A single integer equal to the fewest number of lift towers
        FR needs to build subject to the above constraints

SAMPLE OUTPUT (file skilift.out):

5

OUTPUT DETAILS:

FR builds five supports (at locations 1, 5, 7, 9, and 13). The steel
is tangent to the ground at four locations: 1-5, 5-7, 7-9, and
12-13.

If FR only builds supports at the four locations 1, 5, 9, and 13,
then the steel would be below ground from 5-9. If FR built supports
at 1, 7, and 13, then -- although the steel is always above ground
-- supports 7 and 13 are more than 4 units apart horizontally. There
is no solution using fewer than 5 supports such that no two consecutive
supports are more than 4 units apart horizontally.

USACO March 2006 Problem 'skilift' Analysis

by Bruce Merry

This is a reasonably straightforward dynamic programming problem, with some computational geometry thrown in. Let F(x) be the smallest number of supports required to build the rail from point 0 to point x, including a support at point x. Then F(x) = min F(y) + 1 where x - K <= y < x and a segment can be built from y to x. Once we can determine which segments are viable, the rest is dynamic programming.

For a given x, we run y downwards from x - 1 to x - K. For each y value, we keep track of the maximum gradient possible. If the new y value requires a steeper (or equal) gradient, then the segment is viable and we update the maximum; otherwise it would pass underground.

```
#include <fstream>
#include <algorithm>

using namespace std;

#define MAXN 5000

int main() {
    int N, K;
    int h[MAXN];
    int dp[MAXN];
    ifstream in("skilift.in");
    ofstream out("skilift.out");

    in >> N >> K;
    for (int i = 0; i < N; i++) in >> h[i];

    dp[0] = 1;
    for (int i = 1; i < N; i++) {
        dp[i] = INT_MAX;
```

```
        int high = 0;
        int top = min(i, K);
        for (int j = 1; j <= top; j++) {
            if ((long long) (h[i - j] - h[i]) * high >= (long long) (h[i -
high] - h[i]) * j) {
                high = j;
                dp[i] <?= dp[i - j] + 1;
            }
        }
    }
    out << dp[N - 1] << "\n";

    return 0;
}
```

2. Roping the Field [Hal Burch, 2004] (35, 38) JAN06 Gold

Farmer John is quite the nature artist: he often constructs large works of art on his farm. Today, FJ wants to construct a giant "field web". FJ's field is large convex polygon with fences along the boundary and fence posts at each of the N corners (1 <= N <= 150). To construct his field web, FJ wants to run as many ropes as possible in straight lines between pairs of non-adjacent fence posts such that no two ropes cross.

There is one complication: FJ's field is not completely usable. Some evil aliens have created a total of G (0 <= G <= 100) grain circles in the field, all of radius R (1 <= R <= 100,000). FJ is afraid to upset the aliens, and therefore doesn't want the ropes to pass through, or even touch the very edge of a grain circle. Note that although the centers of all the circles are contained within the field, a wide radius may make it extend outside of the field, and both fences and fence posts may be within a grain circle.

Given the locations of the fence posts and the centers of the circles, determine the maximum number of ropes that FJ can use to create his field web.

FJ's fence posts and the circle centers all have integer coordinates X and Y each of which is in the range 0..1,000,000.

PROBLEM NAME: roping

INPUT FORMAT:

* Line 1: Three space-separated integers: N, G, and R

* Lines 2..N+1: Each line contains two space-separated integers that are the X,Y position of a fence post on the boundary of FJ's field.

* Lines N+2..N+G+1: Each line contains two space-separated integers that are the X,Y position of a circle's center inside FJ's field.

SAMPLE INPUT (file roping.in):

```
5 3 1
6 10
10 7
9 1
2 0
0 3
2 2
5 6
8 3
```

INPUT DETAILS:

A pentagonal field, in which all possible ropes are blocked by three grain circles, except for the rope between fenceposts 2 and 4.

OUTPUT FORMAT:

* Line 1: A single integer that is the largest number of ropes FJ can use for his artistic creation.

SAMPLE OUTPUT (file roping.out):

```
1
```

January 2006 Problem 'roping' Analysis

by Bruce Merry

The problem consists of two parts: determining which ropes do not intersect the circles, and choosing the maximum number of these ropes.

We first consider how to test whether a line (as opposed to a line segment) intersects a circle. Consider a triangle with vertices at the centre of the circle and at the posts defining the line. Its area can be computed in two ways: as the length of the line segment times the distance from the centre of the tree to the line, over 2, or using the cross product. The latter can be computed directly, so equating it to the former gives the perpendicular distance. If this is less than or equal to R, then the line intersects the circle. To avoid dealing with floating point errors, it is a good idea to square out and cross-multiply to work entirely in integers; one catch is that this could push the values over 64 bits, so it is necessary to create a big integer type.

It is also possible that the line meets the circle but the line segment does not. Clearly if either end-point is inside the circle that the line cuts the circle. Otherwise, orient the diagram so that the centre of the circle is at the origin and point A is on the positive X axis. If point B lies to the right of point A, then the line clearly misses the circle. If point B lies to the left of point A, and A also lies to the left of point B when point B is placed on the X axis, then the line segment will hit the circle if the line does.

This procedure allows us to identify valid ropes in $O(N^2T)$ time, but we still need to pick a subset that is free of intersections. We use dynamic programming, solving for the largest set between circles whose indices range from A to B (for every pair $A < B$). Suppose that there is at least one rope emanating from A, and consider the one that ends closest to B (i.e. the

endpoint C has the highest index). The interior of triangle ABC must be completely empty, so the set of ropes is just the union of those ropes in the smaller cases (A, C) and (C, B), as well as the rope AB if it is legal. If A has no ropes emanating from it then we can set C = A + 1 and the above statements remain true. By iterating over all possible values of C, we can compute the optimal number of ropes for (A, B) in terms of the optimal numbers for smaller cases. The entire DP takes $O(N^3)$ time, making the overall algorithm $O(N^2T + N^3)$.

```
#define __STDC_LIMIT_MACROS

#include <iostream>
#include <fstream>
#include <complex>
#include <cassert>
#include <climits>

using namespace std;

#define MAXN 150
#define MAXT 100
#define MAXR 100000
#define MAXXY 1000000

typedef complex<long long> point;

static int N, T, R;
static point posts[MAXN];
static point trees[MAXT];
static bool valid[MAXN][MAXN];
static int dp[MAXN][MAXN];

static void readin() {
    long long x, y;
    ifstream in("roping.in");

    in >> N >> T >> R;
    assert(2 <= N && N <= MAXN);
    assert(0 <= T && T <= MAXT);
    assert(1 <= R && R <= MAXR);
    for (int i = 0; i < N; i++) {
        in >> x >> y;
        posts[i] = point(x, y);
    }
    for (int i = 0; i < T; i++) {
        in >> x >> y;
        trees[i] = point(x, y);
    }
}

static inline long long dot(const point &a, const point &b) {
    return real(conj(a) * b);
}
```

```
static inline long long cross(const point &a, const point &b) {
    return imag(conj(a) * b);
}
static inline long long area(const point &a, const point &b, const point
 &c) {
    return cross(b - a, c - a);
}

static void validate() {
    for (int i = 0; i < N; i++) {
        assert(0 <= real(posts[i]) && real(posts[i]) <= MAXXY);
        assert(0 <= imag(posts[i]) && imag(posts[i]) <= MAXXY);
    }

    long long l = LONG_LONG_MAX, h = LONG_LONG_MIN;
    if (N >= 3) {
        for (int i = 1; i < N; i++)
        {
            int j = (i + 1) % N;
            int k = (i + 2) % N;
            assert(posts[i] != posts[j]);
            long long s = area(posts[i], posts[j], posts[k]);
            if (s == 0)
                assert(norm(posts[j] - posts[i]) < norm(posts[k] - posts[i])
                       && norm(posts[k] - posts[j]) < norm(posts[k] -
 posts[i]));
            l <?= s;
            h >?= s;
        }
        assert(l >= 0 || h <= 0);
    }

    for (int i = 0; i < T; i++) {
        long long l2 = LONG_LONG_MIN;
        long long h2 = 0;

        assert(0 <= real(trees[i]) && real(trees[i]) <= MAXXY);
        assert(0 <= imag(trees[i]) && imag(trees[i]) <= MAXXY);
        for (int j = 0; j < N; j++) {
            int k = (j + 1) % N;
            long long s = area(posts[j], posts[k], trees[i]);
            if (s == 0)
                assert(norm(trees[i] - posts[j]) < norm(posts[k] - posts[j])
                       && norm(trees[i] - posts[k]) < norm(posts[k] -
 posts[j]));
            l2 <?= s;
            h2 >?= s;

            // assert(norm(trees[i] - posts[j]) >= (long long) R * R);
        }
        assert(l < 0 || l2 >= 0);
        assert(h > 0 || h2 <= 0);
    }
}

struct huge {
    /* Base 2^64 */
```

```
    unsigned long long high;
    unsigned long long low;
};

static inline void huge_accum(huge &h, unsigned long long m) {
    unsigned long long m1 = m >> 32;
    unsigned long long m0 = (m & UINT_MAX) << 32;
    h.high += m1;
    if (h.low >= UINT64_MAX - m0) h.high++;
    h.low += m0;
}

static huge huge_mult(unsigned long long x, unsigned long long y) {
    unsigned long long x1 = x >> 32;
    unsigned long long x0 = x & UINT_MAX;
    unsigned long long y1 = y >> 32;
    unsigned long long y0 = y & UINT_MAX;

    huge ans = {x1 * y1, x0 * y0};
    huge_accum(ans, x1 * y0);
    huge_accum(ans, x0 * y1);
    return ans;
}

static bool intersects(point A, point B, point O, long long R) {
    A -= O;
    B -= O;
    if (norm(A) <= R * R) return true;
    if (norm(B) <= R * R) return true;
    /* We want to check if cross(A, B)^2 > norm(A - B) * R^2, which would
     * tell us if the line itself misses the circle. However, both sides
     * may overflow.
     */
    unsigned long long c = ::llabs(cross(A, B));
    unsigned long long n = norm(A - B);
    unsigned long long r = R * R;
    huge left = huge_mult(c, c);
    huge right = huge_mult(n, r);
    if (left.high > right.high
        || (left.high == right.high && left.low > right.low)) return
false;

    if (cross(A, B) > sqrtl(n) * R) return false;
    if (dot(A, B) > norm(A)) return false;
    if (dot(A, B) > norm(B)) return false;
    return true;
}

static void make_valid() {
    memset(valid, 0, sizeof(valid));
    for (int i = 0; i < N; i++)
        for (int j = i + 2; j < N; j++)
        {
            valid[i][j] = true;
            for (int k = 0; k < T; k++)
                if (intersects(posts[i], posts[j], trees[k], R))
                {
```

```
                    valid[i][j] = false;
                    break;
                }
            valid[j][i] = valid[i][j];
        }
    valid[0][N - 1] = valid[N - 1][0] = false;
}

static void solve() {
    make_valid();
    memset(dp, 0, sizeof(dp));
    for (int d = 2; d < N; d++)
        for (int i = 0; i < N - d; i++)
        {
            int j = i + d;
            for (int k = i + 1; k < j; k++)
                dp[i][j] >?= dp[i][k] + dp[k][j];
            dp[i][j] += valid[i][j] ? 1 : 0;
        }
}

static void writeout() {
    ofstream out("roping.out");
    out << dp[0][N - 1] << "\n";
}

int main()
{
    readin();
    validate();
    solve();
    writeout();
    return 0;
}
```

CONVEX HULL DP

1. Land Acquisition [Paul Christiano, 2007] MAR08 GOLD

Farmer John is considering buying more land for the farm and has his eye on N (1 <= N <= 50,000) additional rectangular plots, each with integer dimensions (1 <= width_i <= 1,000,000; 1 <= length_i <= 1,000,000).

If FJ wants to buy a single piece of land, the cost is $1/square unit, but savings are available for large purchases. He can buy any number of plots of land for a price in dollars that is the width of the widest plot times the length of the longest plot. Of course, land plots cannot be rotated, i.e., if Farmer John buys a 3x5 plot and a 5x3 plot in a group, he will pay 5x5=25.

FJ wants to grow his farm as much as possible and desires all the plots of land. Being both clever and frugal, it dawns on him that he can purchase the land in successive groups, cleverly minimizing the total cost by grouping various plots that have advantageous width or length values.

Given the number of plots for sale and the dimensions of each, determine the minimum amount for which Farmer John can purchase all

PROBLEM NAME: acquire

INPUT FORMAT:

* Line 1: A single integer: N

* Lines 2..N+1: Line i+1 describes plot i with two space-separated integers: width_i and length_i

SAMPLE INPUT (file acquire.in):

```
4
100 1
15 15
20 5
1 100
```

```
INPUT DETAILS:

There are four plots for sale with dimensions as shown.

OUTPUT FORMAT:

* Line 1: The minimum amount necessary to buy all the plots.

SAMPLE OUTPUT (file acquire.out):

500

OUTPUT DETAILS:

The first group contains a 100x1 plot and costs 100. The next group
contains a 1x100 plot and costs 100. The last group contains both the 20x5
plot and the 15x15 plot and costs 300. The total cost is 500, which is
minimal.
```

Analysis: Land Acquisition by Richard Peng

We make a few observations to get to a O(nlogn) solution:

- If suffices to consider purchases whose lenght/width are the length/width of some plot.
- Consider purchases in increasing order of their lengths. If we purchase a plot with length l, then its width must be the maximum of all the widths of uncovered plots with lesser lengths. Suppose not, then we could either decrease its width to cover the same set, or some point needs to be covered with a plot with larger length/width, which means we don't need this plot in the first place.
  Note this gives a $O(N^2)$ solution by considering all transitions.
- If a plot is exceeded in both length and width by another plot, then it does not need to be considered.

So it suffices to consider the subset of plots whose widths exceed all plots after this. This set of points can be obtained in 1 right to left sweep.

Let the length and width of the plots be:
l[1]...l[n]
w[1]...w[n]
Then we have w[i]>w[i+1] and our DP formula becomes:
best[i]= min{l[i]*max{w[i]...w[j+1]} + best[j] | 1<=j<I}
= min(l[i]*w[j+1] + best[j] | 1<=j

At this point, observe the DP transition formula is linear in two values dependent on j. So this becomes minimizing the y-intercept of a line with given slope among a set of points. So the standard convex-hull method can

be used to get a O(N) solution for this stage. Along with sorting, this gives O(nlogn)

Here is the solution of MIT student Brian Jacokes:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 50005

struct rect { long long w, h; };

struct line {
  long long m, b;
  line(long long x=0, long long y=0) { m=x; b=y; }
};

int N, lstart, lend;
rect plots[MAX];
long long best[MAX];
line lines[MAX];

int cmp(const void *a, const void *b) {
  rect p = *(rect *)a, q = *(rect *)b;
  if (p.w < q.w) return -1;
  if (p.w > q.w) return 1;
  if (p.h < q.h) return -1;
  return 1;
}

bool bad(line x, line y, line z) {
  // lines x and y intersect at x-coordinate p1
  // lines y and z intersect at x-coordinate p2
  // bad <--> p1 >= p2
  return ((y.m-z.m) * (y.b-x.b) >= (x.m-y.m) * (z.b-y.b));
}

int main() {
  FILE *in = fopen("acquire.in", "r");
  fscanf(in, "%d", &N);
  for (int i = 0; i < N; i++) {
    fscanf(in, "%lld%lld", &plots[i].w, &plots[i].h);
  }
  fclose(in);

  qsort(plots, N, sizeof(plots[0]), cmp);
  int tN = 0;
  for (int i = 0; i < N; i++) {
    plots[tN] = plots[i];
    while (tN > 0 &&
           plots[tN].w >= plots[tN-1].w &&
           plots[tN].h >= plots[tN-1].h) {
      plots[tN-1] = plots[tN];
      tN--;
    }
    tN++;
```

```
  }
  N = tN;

  best[0] = 0;
  lines[0] = line(plots[0].h, best[0]);
  lstart = 0;
  lend = 1;
  for (int i = 0; i < N; i++) {
    // compute best[i+1]
    for (int j = lstart; j < lend; j++) {
      long long tmp = lines[j].m * plots[i].w + lines[j].b;
      if (j == lstart) {
        best[i+1] = tmp;
      }
      else if (tmp < best[i+1]) {
        best[i+1] = tmp;
        lstart = j;
      }
      else {
        break;
      }
    }

    if (i < N-1) {
      // compute new line
      lines[lend] = line(plots[i+1].h, best[i+1]);
      lend++;

      // erase irrelevant lines
      while (lend-lstart >= 3 &&
             bad(lines[lend-3], lines[lend-2], lines[lend-1])) {
        lines[lend-2] = lines[lend-1];
        lend--;
      }
    }
  }

  FILE *out = fopen("acquire.out", "w");
  fprintf(out, "%lld\n", best[N]);
  fclose(out);

  return 0;
}
```

2. The Bovine Accordion and Banjo Orchestra [Lei Huang, 2007] (42, 38) Chinese 07

The 2*N (3 <= N <= 1,000) cows have assembled the Bovine Accordion and Banjo Orchestra! They possess various levels of skill on their respective instruments: accordionist i has an associated talent level A_i (0 <= A_i <= 1,000); banjoist j has an associated talent level B_j (0 <= B_j <= 1,000).

The combined `awesomeness' of a pairing between cows with talents A_i and B_j is directly proportional to the talents of each cow in the pair so a concert with those two cows will earn FJ precisely A_i * B_j dollars in "charitable donations". FJ wishes to maximize the sum of all revenue obtained by his cows by pairing them up in the most profitable way.

Unfortunately, FJ's accordionists are a bit stuck up and stubborn. If accordionist i is paired with banjoist j, then accordionists i+1..N refuse to be paired with banjoists 1..j-1. This creates restrictions on which pairs FJ can form. FJ thus realizes that in order to maximize his profits, he may have to leave some cows unpaired.

To make matters worse, when one or more of the musicians is skipped, they will be greatly upset at their wasted talent and will engage in massive binge drinking to wash away their sorrows.

After all pairings are made, a list is constructed of the groups of each of the consecutive skipped musicians (of either instrument). Every group of one or more consecutive skipped cows will gather together to consume kegs of ice cold orange soda in an amount proportional to the square of the sum of their wasted talent.

Specifically, FJ has calculated that if the x-th to y-th accordionists are skipped, they will consume precisely (A_x + A_x+1 + A_x+2 + ... + A_y)^2 dollars worth of orange soda in the process of drinking themselves into oblivion. An identical relationship holds for the banjoists. FJ realizes that he'll end up getting stuck with the

bill for his cows' drinking, and thus takes this into account when choosing which pairings to make.

Find the maximum amount of total profit that FJ can earn after the contributions are collected and the orange soda is paid for.

Memory Limit: 64MB

PROBLEM NAME: baabo

INPUT FORMAT:

* Line 1: A single integer: N

* Lines 2..N+1: Line i+1 contains the single integer: A_i

* Lines N+2..2*N+1: Line i+N+1 contains the single integer: B_i

SAMPLE INPUT (file baabo.in):

```
3
1
1
5
5
1
1
```

INPUT DETAILS:

There are 6 cows: 3 accordionists and 3 banjoists. The accordionists have talent levels (1, 1, 5), and the banjoists have talent levels (5, 1, 1).

OUTPUT FORMAT:

* Line 1: A single integer that represents the maximum amount of cash that FJ can earn.

SAMPLE OUTPUT (file baabo.out):

```
17
```

OUTPUT DETAILS:

FJ pairs accordionist 3 with banjoist 1 to get earn A_3 * B_1 = 5 * 5 = 25 in profit. He loses a total of (1 + 1)^2 + (1 + 1)^2 = 8 dollars due to the cost of soda for his remaining cows. Thus his final (net) profit is 25 - 8 = 17.

Analysis: The Bovine Accordion and Banjo Orchestra by Richard Peng

---

Since the pairings cannot intersect, this problem can be solved using DP with the state being the last 'pair' of accordionist and banjoist (denoted A and B from here on). This gives O(n^2) states and O(n^2) state transitions for each state as there are O(n^2) possible previous pairs, the state transition function is:

```
Best(i,j)=max{i1<i,j1<j|Best(i1,j1)-SQR(sum(A_i1...A_(i-1)))-
SQR(sum(B_j1...B_(j-1)))}+A_i*B_j
```

where SQR is the square of a number.

This gives an O(n^4) solution, which we try to optimize all the way down to O(n^2). The first observation we make is we cannot skip cows in both A and B when we go from the current pair to the previous one. This because pairing any two from those skipped ones will increase the total profit. So this gives an O(n^3) solution, sufficient to get 60% of the points.

Now we try to get down to O(n^2) by optimizing the state transition using convexity. We first consider the case where no cows in A are skipped, or when i1=i-1. So we need to find:

max{j1<j|Best(i-1,j)-SQR(sum(B_j1...B_(j-1))}+A_i*B_j

If we let SB denote the partial sum of the B array, then our transition formula becomes:

max{j1<j|Best(i-1,j)-SQR(SB_(j-1)-SB(j1-1))}+A_i*B_j =max{j1<j|Best(i-1,j)-SQR(SB_(j1-1))+2*SB_(j1-1)*SB_(j-1)}+A_i*B_j+SQR(SB_(j-1))

We can ignore the last two terms since they're constant depending only on i and j. If we look at the first term, we're looking for the max of a linear combination of two values depending on j1 where the ratio is dependent on j.

Geometrically, this is equivalent to taking the y-intercept of a line with a given slope within a set of points. With some proof, it can be shown that only points on the convex hull matters. So we can insert values of (2*SB_j,Best(i-1,j)-SQR(SB_(j-1)))) into the convex hull (note SB_j is always increasing, so we can do this using a stack). And when we query, we can show the point where this value is minimized/maximized is always monotone in respect to x. This gives an algorithm that does this in O(1) amortized cost.

The case where nothing in B is skipped can be dealt with similarly. The only catch is that should we process the states in incremental i and then incremental j, we'll need to keep track of N+1 convex hulls, one for the previous 'row' and one for each of the columns.

This gives the desired O(N^2) algorithm. Code (by Richard Peng):

```
#include <cstdio>
#include <cstring>

#define MAXN 1200

int n;
double a[MAXN],b[MAXN],bes[MAXN][MAXN],ans;
double s1[MAXN],s2[MAXN];

double sqr(double x){return x*x;}

double hull[MAXN][MAXN][2];
int hullt[MAXN],p[MAXN];

void initialize(int id){
	hullt[id]=p[id]=0;
}

double crossp(double a[2],double b[2],double c[2]){
	return (b[0]-a[0])*(c[1]-a[1])-(c[0]-a[0])*(b[1]-a[1]);
}

void hulladd(int id,double x,double y){
	double point[2];
	point[0]=x;
	point[1]=y;
```

```
        if((hullt[id]>0)&&(x==hull[id][hullt[id]-1][0])){
                if(y>hull[id][hullt[id]-1][1]) hullt[id]--;
                else return;
        }
        while((hullt[id]>1)&&(crossp(point,hull[id][hullt[id]-
1],hull[id][hullt[id]-2])<=0))
                hullt[id]--;
        hull[id][hullt[id]][0]=x;
        hull[id][hullt[id]][1]=y;
        p[id]<?=hullt[id];
        hullt[id]++;
}

double query(int id,double a){
        double tem,tem1;
        tem=hull[id][p[id]][0]*a+hull[id][p[id]][1];
        while((p[id]+1<hullt[id])&&((tem1=(hull[id][p[id]+1][0]*a+hull[id][p[i
d]+1][1]))>tem)){
                tem=tem1;
                p[id]++;
        }
        return tem;
}

int main(){
        int i,j,i1,j1;
freopen("mkpairs.in","r",stdin);
freopen("mkpairs.out","w",stdout);
        scanf("%d",&n);
        for(i=0;i<n;i++)        scanf("%lf",&a[i]);
        for(s1[0]=a[0],i=1;i<n;i++) s1[i]=s1[i-1]+a[i];
        for(i=0;i<n;i++)        scanf("%lf",&b[i]);
        for(s2[0]=b[0],i=1;i<n;i++) s2[i]=s2[i-1]+b[i];

        memset(bes,0,sizeof(bes));

        for(i=1;i<n;i++)
                initialize(i);
        for(ans=i=0;i<n;i++){
                initialize(0);
                for(j=0;j<n;j++){
                        bes[i][j]=-((i==0)?0:sqr(s1[i-1]))-((j==0)?0:sqr(s2[j-
1]));

                        if(i>0){
                                if(j>0){
                                        bes[i][j]>?=query(0,2*s2[j-1])-
sqr(s2[j-1]);
                                        bes[i][j]>?=query(j,2*s1[i-1])-
sqr(s1[i-1]);
                                }
                                hulladd(0,s2[j],bes[i-1][j]-sqr(s2[j]));
                        }


                        bes[i][j]+=a[i]*b[j];
                        ans>?=bes[i][j]-sqr(s1[n-1]-s1[i])-sqr(s2[n-1]-s2[j]);
```

```
                }
                for(j=0;j+1<n;j++){
                        hulladd(j+1,s1[i],bes[i][j]-sqr(s1[i]));
                }
        }
        printf("%0.0lf\n",ans);
        return 0;
}
```