**USACO 2011 November Contest, Bronze Division**

Problem 1: Contest Timing [Brian Dean] Bessie the cow is getting bored of the milk production industry, and wants to switch to an exciting new career in computing. To improve her coding skills, she decides to compete in the on-line USACO competitions. Since she notes that the contest starts on November 11, 2011 (11/11/11), she decides for fun to download the problems and begin coding at exactly 11:11 AM on 11/11/11. Unfortunately, Bessie's time management ability is quite poor, so she wants to write a quick program to help her make sure she does not take longer than the 3 hour (180 minute) time limit for the contest. Given the date and time she stops working, please help Bessie compute the total number of minutes she will have spent on the contest. PROBLEM NAME: ctiming INPUT FORMAT: \* Line 1: This line contains 3 space-separated integers, D H M, specifying the date and time at which Bessie ends the contest. D will be an integer in the range 11..14 telling the day of the month; H and M are hours and minutes on a 24-hour clock (so they range from H=0,M=0 at midnight up through H=23,M=59 at 11:59 PM). SAMPLE INPUT (file ctiming.in): 12 13 14 INPUT DETAILS: Bessie ends the contest on November 12, at 13:14 (that is, at 1:14 PM). OUTPUT FORMAT: \* Line 1: The total number of minutes spent by Bessie in the contest, or -1 if her ending time is earlier than her starting time. SAMPLE OUTPUT (file ctiming.out): 1563 OUTPUT DETAILS: Bessie ends the contest 1563 minutes after she starts.

**Solution Notes:** A key to making the problem easy to solve is to write a function that converts from (day, hour, minute) to a single integer that reflects an absolute count of number of minutes since some pre-determined starting point. In the sample C solution below, the function total\_mins() computes the total number of minutes elapsed since the beginning of the month. Using this function, it is now easy to compute the number of minutes in the difference of two dates - we first convert the two dates into integers, and then simply subtract!

#include <stdio.h>

int total\_mins(int d, int h, int m)

{

return d \* 24 \* 60 + h \* 60 + m;

}

int main(void)

{

int d, h, m;

freopen ("ctiming.in", "r", stdin);

freopen ("ctiming.out", "w", stdout);

scanf ("%d %d %d", &d, &h, &m);

if (total\_mins(d,h,m) < total\_mins(11,11,11))

printf ("-1\n");

else

printf ("%d\n", total\_mins(d,h,m) - total\_mins(11,11,11));

return 0;

}

Problem 2: Awkward Digits [Brian Dean] Bessie the cow is just learning how to convert numbers between different bases, but she keeps making errors since she cannot easily hold a pen between her two front hooves. Whenever Bessie converts a number to a new base and writes down the result, she always writes one of the digits wrong. For example, if she converts the number 14 into binary (i.e., base 2), the correct result should be "1110", but she might instead write down "0110" or "1111". Bessie never accidentally adds or deletes digits, so she might write down a number with a leading digit of "0" if this is the digit she gets wrong. Given Bessie's output when converting a number N into base 2 and base 3, please determine the correct original value of N (in base 10). You can assume N is at most 1 billion, and that there is a unique solution for N. Please feel welcome to consult any on-line reference you wish regarding base-2 and base-3 numbers, if these concepts are new to you. PROBLEM NAME: digits INPUT FORMAT: \* Line 1: The base-2 representation of N, with one digit written incorrectly. \* Line 2: The base-3 representation of N, with one digit written incorrectly. SAMPLE INPUT (file digits.in): 1010 212 INPUT DETAILS: When Bessie incorrectly converts N into base 2, she writes down "1010". When she incorrectly converts N into base 3, she writes down "212". OUTPUT FORMAT: \* Line 1: The correct value of N. SAMPLE OUTPUT (file digits.out): 14 OUTPUT DETAILS: The correct value of N is 14 ("1110" in base 2, "112" in base 3).

**Solution Notes:** The first step in solving this problem is writing a function to convert a number represented by a string in base N (here, N is 2 or 3) into an integer. This is done by the function convert\_to\_base\_n() below. Once we have this function, the rest of the solution is just trying every possible solution to find the one that works - we look over every possible digit i of the base-2 number and every digit j of the base-3 number, and check whether we can make the two numbers evaluate to an equal amount by changing just these two digits. If so, this amount is our answer. The code below does this in a fairly concise manner by temporarily changing the two digits in question, testing the values of the resulting strings, and then putting the digits back the way they started. Digit i in the binary number is toggled between 0 and 1, and digit j in the base-3 number is cycled between 0, 1, and 2.

#include <stdio.h&rt;

#include <stdlib.h&rt;

#include <string.h&rt;

int convert\_to\_base\_n(char \*s, int n)

{

int i = strlen(s), t=0, p=1;

while (i) {

i--;

t += p \* (s[i] - '0');

p = p \* n;

}

return t;

}

int main(void)

{

char A[100], B[100];

int lenA, lenB, i, j, k;

FILE \*fp;

fp = fopen ("digits.in", "r");

fscanf (fp, "%s %s", A, B);

fclose (fp);

lenA = strlen(A);

lenB = strlen(B);

for (i=0; i<lenA; i++)

for (j=0; j<lenB; j++)

for (k=1; k<=2; k++) {

A[i]=((A[i]-'0')+1)%2+'0';

B[j]=((B[j]-'0')+k)%3+'0';

if (convert\_to\_base\_n(A,2) == convert\_to\_base\_n(B,3)) {

fp = fopen ("digits.out", "w");

fprintf (fp, "%d\n", convert\_to\_base\_n(A,2));

fclose (fp);

return 0;

}

A[i]=((A[i]-'0')+1)%2+'0';

B[j]=((B[j]-'0')+3-k)%3+'0';

}

return 0;

}

Problem 3: Moo Sick [Rob Seay] Everyone knows that cows love to listen to all forms of music. Almost all forms, that is -- the great cow composer Wolfgang Amadeus Moozart once discovered that a specific chord tends to make cows rather ill. This chord, known as the ruminant seventh chord, is therefore typically avoided in all cow musical compositions. Farmer John, not knowing the finer points of cow musical history, decides to play his favorite song over the loudspeakers in the barn. Your task is to identify all the ruminant seventh chords in this song, to estimate how sick it will make the cows. The song played by FJ is a series of N (1 <= N <= 20,000) notes, each an integer in the range 1..88. A ruminant seventh chord is specified by a sequence of C (1 <= C <= 10) distinct notes, also integers in the range 1..88. However, even if these notes are transposed (increased or decreased by a common amount), or re-ordered, the chord remains a ruminant seventh chord! For example, if "4 6 7" is a ruminant seventh chord, then "3 5 6" (transposed by -1), "6 8 9" (transposed by +2), "6 4 7" (re-ordered), and "5 3 6" (transposed and re-ordered) are also ruminant seventh chords. A ruminant seventh chord is a sequence of C consecutive notes satisfying the above criteria. It is therefore uniquely determined by its starting location in the song. Please determine the indices of the starting locations of all of the ruminant seventh chords. PROBLEM NAME: moosick INPUT FORMAT: \* Line 1: A single integer: N. \* Lines 2..1+N: The N notes in FJ's song, one note per line. \* Line 2+N: A single integer: C. \* Lines 3+N..2+N+C: The C notes in an example of a ruminant seventh chord. All transpositions and/or re-orderings of these notes are also ruminant seventh chords. SAMPLE INPUT (file moosick.in): 6 1 8 5 7 9 10 3 4 6 7 INPUT DETAILS: FJ's song is 1,8,5,7,9,10. A ruminant seventh chord is some transposition/re-ordering of 4,6,7. OUTPUT FORMAT: \* Line 1: A count, K, of the number of ruminant seventh chords that appear in FJ's song. Observe that different instances of ruminant seventh chords can overlap each-other. \* Lines 2..1+K: Each line specifies the starting index of a ruminant seventh chord (index 1 is the first note in FJ's song, index N is the last). Indices should be listed in increasing sorted order. SAMPLE OUTPUT (file moosick.out): 2 2 4 OUTPUT DETAILS: Two ruminant seventh chords appear in FJ's song (and these occurrences actually overlap by one note). The first is 8,5,7 (transposed by +1 and reordered) starting at index 2, and the second is 7,9,10 (transposed by +3) starting at index 4.

**Solution Notes:** At a high level, all we need to do to solve this problem is to test every window of length C within our larger piece of music to see if it "matches" our chord pattern. The is done in the match() function below, where P[] is a length-C array containing a window from the larger piece of music, and Q[] is a length-C array containing the chord pattern. How do we compare these in a manner that is insensitive to re-ordering and transposition? There are several approaches that would work here; perhaps the simplest is to convert P and Q into a "canonical" form that removes re-ordering and transposition from the picture. For example, if we sort P and Q, then this makes re-ordering no longer matter. We can also shift the contents of P and Q so the minimum in each array is zero. Afterwards, we simply compare P and Q element by element to see if they are equal.

#include <stdio.h>

#define MAX\_N 20000

#define MAX\_C 10

int A[MAX\_N], B[MAX\_C], M[MAX\_N];

int N, C;

void translate\_so\_min\_is\_zero(int \*X)

{

int i, min;

for (i=0; i<C; i++)

if (i==0 || X[i]<min)

min = X[i];

for (i=0; i<C; i++)

X[i] -= min;

}

void sort(int \*X)

{

int i, j, tmp;

for (i=0; i<C; i++)

for (j=0; j<C-1; j++)

if (X[i] > X[j]) {

tmp = X[i];

X[i] = X[j];

X[j] = tmp;

}

}

/\* Return 1 if match at index idx \*/

int match(int idx)

{

int P[MAX\_C], Q[MAX\_C];

int i, j, min, tmp;

for (i=0; i<C; i++) {

P[i] = A[i+idx];

Q[i] = B[i];

}

translate\_so\_min\_is\_zero(P);

translate\_so\_min\_is\_zero(Q);

sort(P);

sort(Q);

for (i=0; i<C; i++)

if (P[i] != Q[i])

return 0;

return 1;

}

int main(void)

{

int i, total = 0;

freopen ("moosick.in", "r", stdin);

freopen ("moosick.out", "w", stdout);

scanf ("%d", &N);

for (i=0; i<N; i++)

scanf ("%d", &A[i]);

scanf ("%d", &C);

for (i=0; i<C; i++)

scanf ("%d", &B[i]);

for (i=0; i+C<=N; i++) {

M[i] = match(i);

total += M[i];

}

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

for (i=0; i<N; i++)

if (M[i])

printf ("%d\n", i+1);

return 0;

}

Problem 4: Cow Beauty Pageant (Bronze Level) [Brian Dean] Hearing that the latest fashion trend was cows with two spots on their hides, Farmer John has purchased an entire herd of two-spot cows. Unfortunately, fashion trends tend to change quickly, and the most popular current fashion is cows with only one spot! FJ wants to make his herd more fashionable by painting each of his cows in such a way that merges their two spots into one. The hide of a cow is represented by an N by M (1 <= N,M <= 50) grid of characters like this: ................ ..XXXX....XXX... ...XXXX....XX... .XXXX......XXX.. ........XXXXX... .........XXX.... Here, each 'X' denotes part of a spot. Two 'X's belong to the same spot if they are vertically or horizontally adjacent (diagonally adjacent does not count), so the figure above has exactly two spots. All of the cows in FJ's herd have exactly two spots. FJ wants to use as little paint as possible to merge the two spots into one. In the example above, he can do this by painting only three additional characters with 'X's (the new characters are marked with '\*'s below to make them easier to see). ................ ..XXXX....XXX... ...XXXX\*...XX... .XXXX..\*\*..XXX.. ........XXXXX... .........XXX.... Please help FJ determine the minimum number of new 'X's he must paint in order to merge two spots into one large spot. PROBLEM NAME: pageant INPUT FORMAT: \* Line 1: Two space-separated integers, N and M. \* Lines 2..1+N: Each line contains a length-M string of 'X's and '.'s specifying one row of the cow hide pattern. SAMPLE INPUT (file pageant.in): 6 16 ................ ..XXXX....XXX... ...XXXX....XX... .XXXX......XXX.. ........XXXXX... .........XXX.... INPUT DETAILS: The pattern in the input shows a cow hide with two distinct spots, labeled 1 and 2 below: ................ ..1111....222... ...1111....22... .1111......222.. ........22222... .........222.... OUTPUT FORMAT: \* Line 1: The minimum number of new 'X's that must be added to the input pattern in order to obtain one single spot. SAMPLE OUTPUT (file pageant.out): 3 OUTPUT DETAILS: Three 'X's suffice to join the two spots into one: ................ ..1111....222... ...1111X...22... .1111..XX..222.. ........22222... .........222....

**Solution Notes:** This was by far the most challenging problem on the bronze contest. To solve it, we first label each of the two spots by using the recursive "flood fill" function label() that spreads out and sets every character in the spot to 1 (for the first spot) or 2 (for the second spot). This recursive function is first called when we see an 'X', after which it labels the spot containing that 'X' with 1s; it then continues scanning until it finds another 'X', after which it is called to label the spot containing that 'X' with 2s. Each time label() is called, it marks a single character and then recursively tries to visit the neighbors of that character, stopping any time we land on a character that isn't 'X'. One concern with this approach is sometimes that if the input grid is large enough, then we may run out of stack space if the label() function recurses too deeply. Fortunately, the grid here is small enough that this is not a concern (if you want to be particularly careful about this issue, you can explicitly allocate and manage the stack of recursive locations to visit, although this is a bit more code).

Once our spots are labeled, we want to find the '1' character and the '2' character that are closest together (i.e., the two characters that we need to join with a path to merge the two spots). Distance here is measured by taking the sum of absolute difference in coordinates - this is sometimes called "Manhattan" or "L1" distance. Since the grid is small enough, we can simply loop over all possible pairs of '1' characters and '2' characters and test the distance between each. If the grid was much larger, we would need to use slightly more sophisticated techniques, such as for example a breadth-first search (which is more of a silver-level technique) to quickly compute the shortest path distance from every character in the grid to a spot.

#include <stdio.h>

#define MAX\_N 50

#define MAX\_M 50

char G[MAX\_N][MAX\_M+1];

int N, M;

int label(int r, int c, char ch)

{

if (G[r][c]!='X') return;

G[r][c] = ch;

if (r>0) label(r-1,c,ch);

if (c>0) label(r,c-1,ch);

if (r<N-1) label(r+1,c,ch);

if (c<M-1) label(r,c+1,ch);

}

int abs(x)

{

if (x>=0) return x;

return -x;

}

int mindist(void)

{

int r1, r2, c1, c2, min=MAX\_N+MAX\_M;

for (r1=0; r1<N; r1++)

for (c1=0; c1<M; c1++)

if (G[r1][c1]=='1')

for (r2=0; r2<N; r2++)

for (c2=0; c2<M; c2++)

if (G[r2][c2]=='2')

if (abs(r1-r2) + abs(c1-c2) < min)

min = abs(r1-r2) + abs(c1-c2);

return min - 1;

}

int main(void)

{

int r, c;

char ch='0';

freopen ("pageant.in", "r", stdin);

freopen ("pageant.out", "w", stdout);

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

for (r=0; r<N; r++)

scanf ("%s", &G[r]);

for (r=0; r<N; r++)

for (c=0; c<M; c++)

if (G[r][c] == 'X')

label(r,c,++ch);

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

return 0;

}

## USACO 2011 December Contest, Bronze Division

Problem 1: Hay Bales [Brian Dean, 2011] The cows are at it again! Farmer John has carefully arranged N (1 <= N <= 10,000) piles of hay bales, each of the same height. When he isn't looking, however, the cows move some of the hay bales between piles, so their heights are no longer necessarily the same. Given the new heights of all the piles, please help Farmer John determine the minimum number of hay bales he needs to move in order to restore all the piles to their original, equal heights. PROBLEM NAME: haybales INPUT FORMAT: \* Line 1: The number of piles, N (1 <= N <= 10,000). \* Lines 2..1+N: Each line contains the number of hay bales in a single pile (an integer in the range 1...10,000). SAMPLE INPUT (file haybales.in): 4 2 10 7 1 INPUT DETAILS: There are 4 piles, of heights 2, 10, 7, and 1. OUTPUT FORMAT: \* Line 1: An integer giving the minimum number of hay bales that need to be moved to restore the piles to having equal heights. SAMPLE OUTPUT (file haybales.out): 7 OUTPUT DETAILS: By moving 7 hay bales (3 from pile 2 to pile 1, 2 from pile 2 to pile 4, 2 from pile 3 to pile 4), we can make all piles have height 5.

**Solution Notes:** We can calculate K by taking the total number of hay bales and dividing by N. Now that we know the target height K of each pile, let X be the total number of hay bales sitting at height above K. Each one of these hay bales must be moved at some point, so we know the optimal solution has to be at least as large as X. Moreover, we can always get by with moving at most X haybales by repeatedly moving a bale from any pile taller than K to any pile shorter than K until every pile has height K. Therefore, the answer is exactly X. It is important to note with this problem that we don't need to "explicitly" compute how the hay bales are supposed to be re-distributed in order to solve the problem.

#include <stdio.h>

#define MAX\_N 10000

int N, K, A[MAX\_N];

int main(void)

{

int i, sum=0, answer=0;

freopen ("haybales.in", "r", stdin);

freopen ("haybales.out", "w", stdout);

scanf ("%d", &N);

for (i=0; i<N; i++) {

scanf ("%d", &A[i]);

sum += A[i];

}

K = sum / N;

for (i=0; i<N; i++)

if (A[i] > K)

answer += A[i] - K;

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

return 0;

}

Problem 2: Cow Photography (Bronze) [Brian Dean, 2011] The cows are in a particularly mischievous mood today! All Farmer John wants to do is take a photograph of the cows standing in a line, but they keep moving right before he has a chance to snap the picture. Specifically, FJ's N (1 <= N <= 20,000) cows are tagged with ID numbers 1...N. FJ wants to take a picture of the cows standing in a line in a very specific ordering, represented by the contents of an array A[1...N], where A[j] gives the ID number of the jth cow in the ordering. He arranges the cows in this order, but just before he can press the button on his camera to snap the picture, up to one cow moves to a new position in the lineup. More precisely, either no cows move, or one cow vacates her current position in the lineup and then re-inserts herself at a new position in the lineup. Frustrated but not deterred, FJ again arranges his cows according to the ordering in A, but again, right before he can snap a picture, up to one cow (different from the first) moves to a new position in the lineup. The process above repeats for a total of five photographs before FJ gives up. Given the contents of each photograph, see if you can reconstruct the original intended ordering A. Each photograph shows an ordering of the cows in which up to one cow has moved to a new location, starting from the initial ordering in A. Moreover, if a cow opts to move herself to a new location in one of the photographs, then she does not actively move in any of the other photographs (although she can end up at a different position due to other cows moving, of course). PROBLEM NAME: photo INPUT FORMAT: \* Line 1: The number of cows, N (1 <= N <= 20,000). \* Lines 2..5N+1: The next 5N lines describe five orderings, each one a block of N contiguous lines. Each line contains the ID of a cow, an integer in the range 1..N. SAMPLE INPUT (file photo.in): 5 1 2 3 4 5 2 1 3 4 5 3 1 2 4 5 4 1 2 3 5 5 1 2 3 4 INPUT DETAILS: There are 5 cows, with IDs 1, 2, 3, 4, and 5. In each of the 5 photos, a different cow moves to the front of the line (although the cows could have moved anywhere else, if they wanted). OUTPUT FORMAT: \* Lines 1..N: The intended ordering A, one ID per line. SAMPLE OUTPUT (file photo.out): 1 2 3 4 5 OUTPUT DETAILS: The correct original ordering A[1..5] is 1,2,3,4,5.

**Solution Notes:** There are several ways to approach this problem. Perhaps the simplest (which also solves the harder silver/gold variant of this problem where multiple cows can move in each photo) is just to sort, as shown in the code below. Sorting requires that we can compare two cows A and B to tell which should go first. Fortunately, if we look at any two cows A and B, they will be in the correct relative order in at least 3 of the 5 photographs, since movements of other cows will not change the relative ordering of A and B -- only movement of A or B can change their relative order, and A and B can themselves move in at most 2 of photos. We can therefore compare any pair (A,B) by taking a "majority vote" based on their relative order in all 5 photographs (i.e., A < B if A precedes B in at least 3 of the 5 photos). The code below uses a "map" to record the position in each ordering of each cow based on its ID number; this approach is nice because it does not need to assume the ID numbers are small consecutive integers; however, since we know our ID numbers are 1...N, a simple array would have worked also for this task.

Since we know that at most one cow moves per photo in this problem, other solutions exist, some of them even more efficient than the O(n log n) sorting approach above. For example, the "correct" first cow must appear in one of the first two positions in at least 4 of the 5 photos. If there is only one cow satisfying this property, we place it first in the final output and continue. Otherwise, if there are two cows satisfying this property, we compare them as above and place the correct cow first, continuing in this fashion as we output all the cows in the correct order in only O(n) total time.

#include <iostream>

#include <algorithm>

#include <map>

#include <cstdio>

using namespace std;

#define MAXN 20000

int A[MAXN];

map<int, int> pos[5];

bool cmp(int a, int b) {

int f = 0;

for(int i = 0; i < 5; i++) {

f += pos[i][a] < pos[i][b];

}

return f > 2;

}

int main() {

freopen("photo.in", "r", stdin);

freopen("photo.out", "w", stdout);

int N; cin >> N;

for(int i = 0; i < 5; i++) {

for(int j = 0; j < N; j++) {

int x; cin >> x;

pos[i][x] = j;

A[j] = x;

}

}

sort(A, A + N, cmp);

for(int i = 0; i < N; i++) {

cout << A[i] << '\n';

}

}

Problem 3: Escaping the Farm [Brian Dean and Kalki Seksaria, 2011] The cows have decided on a daring plan to escape from the clutches of Farmer John. They have managed to procure a small inflatable raft, and during the cover of night, a group of cows will board the raft and row across the river bordering the farm. The plan seems perfect, until the cows realize that their small inflatable raft may not be able to hold much weight! The N cows (1 <= N <= 20) have weights w\_1 ... w\_N. To figure out if a group of cows is light enough to avoid sinking the raft, the cows add up all of the weights in the group. Unfortunately, cows are notoriously bad at arithmetic, and if the addition of the weights of the cows in a group causes any carries to occur (using standard base 10 addition), then the cows give up and conclude that group must weigh too much to use the raft. Any group whose weights can be added without any carries is assumed to be light enough to fit on the raft. Please help the cows determine the size of the largest group that they believe can fit on the raft (that is, the largest group whose weights can be added together with no carries). PROBLEM NAME: escape INPUT FORMAT: \* Line 1: The number of cows, N (1 <= N <= 20). \* Lines 2..N+1: Each line contains the weight of one cow, an integer in the range 1...100,000,000. SAMPLE INPUT (file escape.in): 5 522 6 84 7311 19 INPUT DETAILS: There are 5 cows, with weights 522, 6, 84, 7311, and 19. OUTPUT FORMAT: \* Line 1: The number of cows in the largest group whose weights can be added together with no carries. SAMPLE OUTPUT (file escape.out): 3 OUTPUT DETAILS: The three weights 522, 6, and 7311, can be added together with no carries: 522 6 + 7311 ------ 7839

**Solution Notes:** The problem is solved by "brute force", by enumerating through all possible subsets of cows and keeping track of the largest one that fits on the raft. One way to do this enumeration is recursively, shown in the code below, where we consider cow #1 and then recursively enumerate all solutions that do contain cow #1, and then all solutions that do not contain cow #1. Along the way, we prune the search and backtrack if we ever notice that our current solution is infeasible (i.e., if it generates a carry), or if the number of cows remaining plus the number of cows we have used so far cannot possibly do better than the best solution we have generated thus far.

Another nice "trick" for enumerating all 2^20 subsets of cows for this problem is to simply count from 0 up to 2^20-1. The 0s and 1s in the binary representation of our integer counter tell us a particular subset, and we will generate every possible subset as we count. For example, if the counter is (in binary) 10011000000000000000, then this means we are considering a subset that includes cows 1, 4, and 5 (the positions of the 1 bits).

#include <fstream>

using namespace std;

int n, w[20], best=0;

/\* Can x and y be added with no carries? \*/

int check(int x, int y)

{

for ( ; x>0 && y>0; x/=10,y/=10)

if (x%10+y%10>=10) return 0;

return 1;

}

/\*

x = index into w array we're currently considering (i.e., we have already

added a subset of w[1...x-1] and are considering whether to add w[x]).

sum = cumulative sum of subset added so far.

count = number of elements in subset added so far.

\*/

void rec(int x, int sum, int count)

{

if (count>best) best=count;

if (x>=n || count+n-x<=best) return;

if (check(sum,w[x]))

rec(x+1,sum+w[x],count+1);

rec(x+1,sum,count);

}

int main()

{

ifstream fin("escape.in");

ofstream fout("escape.out");

fin >> n;

for (int i=0; i<n; i++)

fin >> w[i];

fin.close();

rec(0,0,0);

fout << best << endl;

fout.close();

return 0;

}

## USACO 2012 January Contest, Bronze Division

Problem 1: Gifts [Kalki Seksaria and Brian Dean, 2012] Farmer John wants to give gifts to his N (1 <= N <= 1000) cows, using his total budget of B (1 <= B <= 1,000,000,000) units of money. Cow i requests a gift with a price of P(i) units, and a shipping cost of S(i) units (so the total cost would be P(i)+S(i) for FJ to order this gift). FJ has a special coupon that he can use to order one gift of his choosing at only half its normal price. If FJ uses the coupon for cow i, he therefore would only need to pay P(i)/2+S(i) for that cow's gift. Conveniently, the P(i)'s are all even numbers. Please help FJ determine the maximum number of cows to whom he can afford to give gifts. PROBLEM NAME: gifts INPUT FORMAT: \* Line 1: Two space-separated integers, N and B. \* Lines 2..1+N: Line i+1 contains two space-separated integers, P(i) and S(i). (0 <= P(i),S(i) <= 1,000,000,000, with P(i) even) SAMPLE INPUT (file gifts.in): 5 24 4 2 2 0 8 1 6 3 12 5 INPUT DETAILS: There are 5 cows, and FJ has a budget of 24. Cow 1 desires a gift with price 4 and shipping cost 2, etc. OUTPUT FORMAT: \* Line 1: The maximum number of cows for whom FJ can purchase gifts. SAMPLE OUTPUT (file gifts.out): 4 OUTPUT DETAILS: FJ can purchase gifts for cows 1 through 4, if he uses the coupon for cow 3. His total cost is (4+2)+(2+0)+(4+1)+(6+3) = 22. Note that FJ could have used the coupon instead on cow 1 or 4 and still met his budget.

**Solution Notes:** If there were no coupon, this problem would be solved by "greedily" using gifts in increasing order of P+S values (since we want to pack in as many gifts as possible before we hit the budget). In fact, if we know the gift to which we want to apply the coupon, the rest of our budget should be filled this same way -- by purchasing the remaining gifts "greedily" in increasing order of P+S. A simple solution now is to just try every possible gift as the "coupon" gift (greedily purchasing any remaining gifts that fit), and taking the best solution. The code below does this in O(N^2) time, which is plenty fast for the limits in this problem. As an exercise, can you figure out how to solve it in only O(N log N) time?

[Note: This solution has been modified since it was originally posted to correct a bug.]

#include <stdio.h>

#define MAX\_N 1000

int N, B, P[MAX\_N], S[MAX\_N];

void sort\_by\_p\_plus\_s(void)

{

int i, tmp, done=0;

while (!done) {

done = 1;

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

if (P[i]+S[i] > P[i+1]+S[i+1]) {

tmp = P[i]; P[i] = P[i+1]; P[i+1] = tmp;

tmp = S[i]; S[i] = S[i+1]; S[i+1] = tmp;

done = 0;

}

}

}

int try\_coupon(int c)

{

int i, budget = B - (P[c]/2+S[c]), total=1;

if (budget < 0) return 0;

for (i=0; i<N; i++)

if (P[i]+S[i] <= budget && i!=c) {

budget -= P[i]+S[i];

total++;

}

return total;

}

int main(void)

{

int i, best=0;

freopen ("gifts.in", "r", stdin);

freopen ("gifts.out", "w", stdout);

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

for (i=0; i<N; i++)

scanf ("%d %d", &P[i], &S[i]);

sort\_by\_p\_plus\_s();

for (i=0; i<N; i++)

if (try\_coupon(i) > best)

best = try\_coupon(i);

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

return 0;

}

Problem 3: Grazing Patterns [Brian Dean, 2012] Due to recent budget cuts, FJ has downsized his farm so that the grazing area for his cows is only a 5 meter by 5 meter square field! The field is laid out like a 5x5 grid of 1 meter by 1 meter squares, with (1,1) being the location of the upper-left square, and (5,5) being the location of the lower-right square: (1,1) (1,2) (1,3) (1,4) (1,5) (2,1) (2,2) (2,3) (2,4) (2,5) (3,1) (3,2) (3,3) (3,4) (3,5) (4,1) (4,2) (4,3) (4,4) (4,5) (5,1) (5,2) (5,3) (5,4) (5,5) Every square in this grid is filled with delicious grass, except for K barren squares (0 <= K <= 22, K even), which have no grass. Bessie the cow starts grazing in square (1,1), which is always filled with grass, and Mildred the cow starts grazing in square (5,5), which also is always filled with grass. Each half-hour, Bessie and Mildred finish eating all the grass in their respective squares and each both move to adjacent grassy squares (north, south, east, or west). They want to consume all the grassy squares and end up in exactly the same final location. Please compute the number of different ways this can happen. Bessie and Mildred always move onto grassy squares, and they never both move onto the same square unless that is the very last grassy square remaining. PROBLEM NAME: grazing INPUT FORMAT: \* Line 1: The integer K. \* Lines 2..1+K: Each line contains the location (i,j) of a non-grassy square by listing the two space-separated integers i and j. SAMPLE INPUT (file grazing.in): 4 3 2 3 3 3 4 3 1 INPUT DETAILS: The initial grid looks like this (where . denotes a grassy square, x denotes a non-grassy square, b indicates the starting location of Bessie, and m indicates the starting location of Mildred): b . . . . . . . . . x x x x . . . . . . . . . . m OUTPUT FORMAT: \* Line 1: The number of different possible ways Bessie and Mildred can walk across the field to eat all the grass and end up in the same final location. SAMPLE OUTPUT (file grazing.out): 1 OUTPUT DETAILS: There is only one possible solution, with Bessie and Mildred meeting at square (3,5): b b--b b--b | | | | | b--b b--b b | x x x x b/m | m--m--m--m--m | m--m--m--m—m

**Solution Notes:** A much simpler way to look at this problem is to simply count all possible self-avoiding walks from (1,1) to (5,5) that cover all the valid squares. That is, we pretend that there is just one cow who wants to move from (1,1) to (5,5), instead of there being two cows moving at the same time. It is easy to see that there is a one-to-one correspondence between paths involving two cows meeting halfway (as in the original problem statement) and paths involving just one cow who starts at (1,1) and ends at (5,5); the midpoint of the one-cow path would correspond to the meeting point in the analogous two-cow solution.

Now that we know we need to count paths from (1,1) to (5,5), we proceed to enumerate and count all such paths with a recursive function. The "count" function below does this by temporarily noting that the current square is blocked (so we don't return there), temporarily incrementing the blocked square count K, and then recursively visiting all neighbors, accumulating the counts we get from each of them.

#include <stdio.h>

int A[5][5];

int K;

int count(int i, int j)

{

int c;

if (i<0 || i>4 || j<0 || j>4 || A[i][j]!=0) return 0;

A[i][j] = 1; K++;

if (K==25 && i==4 && j==4)

c = 1;

else

c = count(i-1,j) + count(i+1,j) + count(i,j-1) + count(i,j+1);

A[i][j] = 0; K--;

return c;

}

int main(void)

{

int i,j,k;

freopen ("grazing.in", "r", stdin);

freopen ("grazing.out", "w", stdout);

scanf ("%d", &K);

for (k=0; k<K; k++) {

scanf ("%d %d", &i, &j);

A[i-1][j-1] = 1;

}

printf ("%d\n", count(0,0));

return 0;

}

## USACO 2012 February Contest, Bronze Division

## Problem 1: Rope Folding [Brian Dean, 2012] Farmer John has a long rope of length L (1 <= L <= 10,000) that he uses for various tasks around his farm. The rope has N knots tied into it at various distinct locations (1 <= N <= 100), including one knot at each of its two endpoints. FJ notices that there are certain locations at which he can fold the rope back on itself such that the knots on opposite strands all line up exactly with each-other: http://usaco.org/current/data/fig_knots.pngPlease help FJ count the number of folding points having this property. Folding exactly at a knot is allowed, except folding at one of the endpoints is not allowed, and extra knots on the longer side of a fold are not a problem (that is, knots only need to line up in the areas where there are two strands opposite each-other). FJ only considers making a single fold at a time; he fortunately never makes multiple folds. PROBLEM NAME: folding INPUT FORMAT: \* Line 1: Two space-separated integers, N and L. \* Lines 2..1+N: Each line contains an integer in the range 0...L specifying the location of a single knot. Two of these lines will always be 0 and L. SAMPLE INPUT (file folding.in): 5 10 0 10 6 2 4 INPUT DETAILS: The rope has length L=10, and there are 5 knots at positions 0, 2, 4, 6, and 10. OUTPUT FORMAT: \* Line 1: The number of valid folding positions. SAMPLE OUTPUT (file folding.out): 4 OUTPUT DETAILS: The valid folding positions are 1, 2, 3, and 8.

**Solution Notes:** Perhaps the easiest way to solve this problem is to first sort the knot locations and then build an array of differences between successive locations. For example, the locations of 0, 2, 4, 6, and 10 in the sample input would be translated into the array of differences 2, 2, 2, 4. We then observe that any prefix or suffix of the difference array that is a palindrome (reads the same forward as backward) corresponds to a valid fold. For example, the prefix "2, 2" corresponds to a fold at the knot at location 2, the prefix "2, 2, 2" corresponds to a fold in between the knots at locations 2 and 4, and the suffix "4" corresponds to a fold in between the knots at locations 6 and 10. Even-length palindromes correspond to folds at knots, and odd-length palindromes correspond to folds in between two knots.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_N 100

int knots[MAX\_N];

int intcmp(const void \*p1, const void \*p2)

{

int \*i1 = (int \*)p1;

int \*i2 = (int \*)p2;

return \*i1 - \*i2;

}

int check\_palindrome(int start, int end)

{

int i;

for (i=0; start+i<=end-i; i++)

if (knots[start+i] != knots[end-i]) return 0;

return 1;

}

int main(void)

{

int N, L, i, count=0;

freopen ("folding.in", "r", stdin);

freopen ("folding.out", "w", stdout);

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

for (i=0; i<N; i++)

scanf ("%d", &knots[i]);

/\* Sort knots \*/

qsort (knots, N, sizeof(int), intcmp);

/\* Convert knots array into successive differences \*/

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

knots[i] = knots[i+1] - knots[i];

/\* Check left palindromes \*/

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

if (check\_palindrome(0,i)) count++;

/\* Check right palindromes \*/

for (i=1; i<N-1; i++)

if (check\_palindrome(i,N-2)) count++;

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

return 0;

}

## Problem 2: Overplanting (Bronze) [Brian Dean, 2012] Farmer John has purchased a new machine that is capable of planting grass within any rectangular region of his farm that is "axially aligned" (i.e., with vertical and horizontal sides). Unfortunately, the machine malfunctions one day and plants grass in not one, but N (1 <= N <= 10) different rectangular regions, some of which may even overlap. Given the rectangular regions planted with grass, please help FJ compute the total area in his farm that is now covered with grass. PROBLEM NAME: planting INPUT FORMAT: \* Line 1: The integer N. \* Lines 2..1+N: Each line contains four space-separated integers x1 y1 x2 y2 specifying a rectangular region with upper-left corner (x1,y1) and lower-right corner (x2,y2). All coordinates are in the range -10,000...10,000. SAMPLE INPUT (file planting.in): 2 0 5 4 1 2 4 6 2 OUTPUT FORMAT: \* Line 1: The total area covered by grass. SAMPLE OUTPUT (file planting.out): 20

## Solution Notes: There are several ways to approach this problem. One common type of solution is a "sweep line" approach, described in the analysis of the silver version of this problem (along with code for the solution). Another approach is to extend every line segment in our scene, creating a grid of horizontal and vertical lines (the grid is easy to loop over if we first sort all the x and y coordinates in the scene). For each rectanglular cell in the grid, we add its area into the total if some rectangle overlaps that cell (and this is easy to check by looping over all the rectangles in the input). For the mathematically inclined, the problem can also be solved via the principle of inclusion-exclusion: loop over all 2^N subsets of rectangles, compute the area A of the intersection of the rectangles in each subset, and either add this area to the total if the subset contains an odd number of rectangles, or subtract it from the total if the subset contains an even number of rectangles (i.e., add the areas of all single rectangles, then subtract the areas of all pairwise intersections, then add back in the areas of all 3-wise intersections, etc.)

## Problem 3: Moo [Brian Dean, 2012] The cows have gotten themselves hooked on a new word game, called "Moo". It is played by a group of cows standing in a long line, where each cow in sequence is responsible for calling out a specific letter as quickly as possible. The first cow who makes a mistake loses. The sequence of letters in Moo can technically continue forever. It starts like this: m o o m o o o m o o m o o o o m o o m o o o m o o m o o o o o The sequence is best described recursively: let S(0) be the 3-character sequence "m o o". Then a longer sequence S(k) is obtained by taking a copy of the sequence S(k-1), then "m o ... o" with k+2 o's, and then another copy of the sequence S(k-1). For example: S(0) = "m o o" S(1) = "m o o m o o o m o o" S(2) = "m o o m o o o m o o m o o o o m o o m o o o m o o" As you can see, this process ultimately builds an infinitely long string, and this is the string of characters used for the game of Moo. Bessie the cow, feeling clever, wishes to predict whether the Nth character of this string will be an "m" or an "o". Please help her out! PROBLEM NAME: moo INPUT FORMAT: \* Line 1: A single integer N (1 <= N <= 10^9). SAMPLE INPUT (file moo.in): 11 INPUT DETAILS: Bessie wants to predict the 11th character. OUTPUT FORMAT: \* Line 1: The only line of output should contain a single character, which is either m or o. SAMPLE OUTPUT (file moo.out): m

**Solution Notes:** This problem has a nice recursive solution. We first write a recursive function to compute the length of S(k) (given by twice the length of S(k-1) plus the length of the middle section, k+3). Afterwards, we can figure out the nth chracter in S(k) by checking if n lies in the left copy of S(k-1) (in which case we can proceed by recursion), in the middle section, or in the right copy of S(k-1) (in which case we can again proceed by recursion).

#include <stdio.h>

/\* Compute the length of S(k) \*/

int len(int k)

{

int x;

if (k==-1) return 0;

x = len(k-1);

return x + k+3 + x;

}

/\* Return nth character in S(k) \*/

char solve(int n, int k)

{

if (n > len(k)) return solve(n,k+1);

if (n <= len(k-1)) return solve(n,k-1);

n = n - len(k-1); /\* Discount S(k-1) from beginning of string \*/

if (n <= k+3) /\* n in middle section? \*/

return (n==1) ? 'm' : 'o';

n = n - (k+3);

return solve(n,k-1);

}

int main(void)

{

int n;

freopen ("moo.in", "r", stdin);

freopen ("moo.out", "w", stdout);

scanf ("%d", &n);

printf ("%c\n", solve(n,0));

return 0;

}

## USACO 2012 March Contest, Bronze Division

## Problem 1: Times17 [Brian Dean, 2012] After realizing that there is much money to be made in software development, Farmer John has launched a small side business writing short programs for clients in the local farming industry. Farmer John's first programming task seems quite simple to him -- almost too simple: his client wants him to write a program that takes a number N as input, and prints 17 times N as output. Farmer John has just finished writing this simple program when the client calls him up in a panic and informs him that the input and output both must be expressed as binary numbers, and that these might be quite large. Please help Farmer John complete his programming task. Given an input number N, written in binary with at most 1000 digits, please write out the binary representation of 17 times N. PROBLEM NAME: times17 INPUT FORMAT: \* Line 1: The binary representation of N (at most 1000 digits). SAMPLE INPUT (file times17.in): 10110111 OUTPUT FORMAT: \* Line 1: The binary representation of N times 17. SAMPLE OUTPUT (file times17.out): 110000100111 OUTPUT DETAILS: The binary number 10110111 is equal to 183 in decimal form. 183 x 17 = 3111 is 110000100111 in binary format.

**Solution Notes:** This problem is not too hard if we make the observation that 17N = 16N + N, and in binary 16N is just the binary representation of N followed by four digits of 0 (that is, N shifted right by four digits). We therefore add these two binary numbers to obtain our answer.

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

char num1[1010], num2[1010], s[1010], result[1010] = {0};

int i, L;

freopen ("times17.in", "r", stdin);

freopen ("times17.out", "w", stdout);

scanf ("%s", s);

sprintf (num1, "00000%s", s);

sprintf (num2, "0%s0000", s);

L = strlen(num1);

for (i=L-1; i>0; i--) {

result[i] += num1[i]-'0'+num2[i]-'0';

if (result[i] >= 2) { result[i] -= 2; result[i-1] += 1; }

}

i = 0;

if (result[0]==0) i = 1;

while (i < L)

printf ("%d", result[i++]);

printf ("\n");

return 0;

}

## Problem 2: Connect the Cows [Brian Dean, 2012] Every day, Farmer John walks around his farm to check on the health and well-being of his N (1 <= N <= 10) cows. The location of each cow is described by a point in the 2D plane, and Farmer John starts out at the origin (0,0). To make his route more interesting, Farmer John decides that he will only walk in directions parallel to the coordinate axes -- that is, only north, south, east, or west. Furthermore, he only changes his direction of travel when he reaches the location of a cow (he may also opt to pass through the location of a cow without changing direction, if desired). When he changes his direction of travel, he may make either a 90-degree or 180-degree turn. FJ's route must take him back to the origin after visiting all his cows. Please compute the number of different routes FJ can take to visit his N cows, if he changes direction exactly once at the location of each cow. He is allowed to pass through the location of a cow without changing direction an arbitrary number of times. The same geometric route taken forward versus backward counts as two different routes. PROBLEM NAME: connect INPUT FORMAT: \* Line 1: The integer N. \* Lines 2..1+N: Line i+1 contains the x and y coordinates (space-separated) of the ith point (each values is in the range -1000...1000). SAMPLE INPUT (file connect.in): 4 0 1 2 1 2 0 2 -5 INPUT DETAILS: There are 4 cows, at positions (0,1), (2,1), (2,0), and (2,-5). OUTPUT FORMAT: \* Line 1: The number of different routes FJ can take (this could be zero if there are no valid routes). SAMPLE OUTPUT (file connect.out): 2 OUTPUT DETAILS: There are two different routes: Farmer John can visit cows in the orders 1-2-4-3 or 3-4-2-1 before returning to the origin.

**Solution Notes:** We solve this problem by "brute force". Since we need to change direction exactly once at each cow (many students seem to have overlooked this condition!) it suffices to enumerate all possible N! permutations of cows. For each permutation, we see if it gives us a valid ordering of the direction change events at our N cows (i.e., is each successive cow horizontally or vertically located relative to the previous cow on the permutation, and does the direction to this cow represent a change in direction from our previous direction?). In C++, one can use the next\_permutation() function to generate all N! permutations very easily; since not all competitors might know about this function, the straight C code below shows how to enumeration through permutations using recursion.

#include <stdio.h>

#define SWAP(a,b) tmp=a; a=b; b=tmp

int N, X[12], Y[12];

int get\_dir(int x1, int y1, int x2, int y2)

{

if (x1!=x2 && y1!=y2) return -1; /\* No direction \*/

if (x1==x2 && y1<y2) return 0; /\* p2 north of p1 \*/

if (x1==x2 && y1>y2) return 1; /\* p2 south of p1 \*/

if (y1==y2 && x1<x2) return 2; /\* p2 east of p1 \*/

if (y1==y2 && x1>x2) return 3; /\* p2 west of p1 \*/

}

/\* Check a permutation... \*/

int check(int \*p)

{

int i;

for (i=1; i<=N+1; i++)

if (get\_dir(X[p[i]], Y[p[i]], X[p[i-1]], Y[p[i-1]])==-1) return 0;

for (i=1; i<=N; i++)

if (get\_dir(X[p[i]], Y[p[i]], X[p[i-1]], Y[p[i-1]]) ==

get\_dir(X[p[i+1]], Y[p[i+1]], X[p[i]], Y[p[i]])) return 0;

return 1;

}

/\* Generate and check all permutations... \*/

int perm(int n, int \*so\_far, int \*remaining)

{

int i, tmp, total=0;

if (n==N) return check(so\_far);

for (i=0; i<N-n; i++) {

so\_far[n+1] = remaining[i];

SWAP(remaining[i], remaining[N-n-1]);

total += perm(n+1, so\_far, remaining);

SWAP(remaining[i], remaining[N-n-1]);

}

return total;

}

int main(void)

{

int P[12] = {0}, D[10] = {1,2,3,4,5,6,7,8,9,10}, i;

freopen ("connect.in", "r", stdin);

freopen ("connect.out", "w", stdout);

scanf ("%d", &N);

for (i=1; i<=N; i++)

scanf ("%d %d", &X[i], &Y[i]);

printf ("%d\n", perm (0,P,D));

return 0;

}

## Problem 3: Wrong Directions [Brian Dean, 2012] Farmer John has just purchased a fancy new programmable tractor. To make the tractor move, he types in a string of length N (1 <= N <= 100,000) consisting of only the characters F, L, and R. Each 'F' instructs the tractor to move forward one unit, and the characters 'L' and 'R' result in left and right turns of 90 degrees, respectively. The tractor starts out at the origin (0,0) facing north. After programming his tractor by typing in his intended command string, FJ remembers that he typed exactly one character in the command string incorrectly, but he can't remember which one! For example, he might have typed 'F' or 'L' when his intended string contained the character 'R'. Please compute the number of different locations in the plane at which the tractor might end up as a result (the direction the tractor faces in its final location does not matter). PROBLEM NAME: wrongdir INPUT FORMAT: \* Line 1: Farmer John's intended command string. SAMPLE INPUT (file wrongdir.in): FF INPUT DETAILS: Farmer John wants the tractor to advance forward twice, ideally ending at position (0,2). OUTPUT FORMAT: \* Line 1: The number of positions at which the tractor might end up, given that FJ mistypes one of the characters in his command string. SAMPLE OUTPUT (file wrongdir.out): 3 OUTPUT DETAILS: There are 4 possible mistyped sequences: FL, FR, LF, an RF. These will land the tractor at (0,1), (0,1), (-1,0), and (1,0) respectively, a total of 3 distinct locations.

**Solution Notes:** For each character in our input string, there are two possible "typo" characters to try, giving us potentially 2N different sets of directions to check if our input has length N. To check these quickly, we first scan over our string backwards and compute, for each index i, the relative offset we will reach if we carry out just the "suffix" of instructions i..N. This takes O(N) time. Scanning forward and keeping a running offset of our position and direction relative to the origin, it is now easy to check each index i: the effect of a typo at i is given by our running position/direction up to index i-1, plus the typo command at index i, plus the relative offset of the suffix starting at i+1. We can therefore check all the typo strings in only O(N) time. Each one generates a potential ending point, after which we sort all of these and scan through to count unique entries in the list.

#include <stdio.h>

#include <string.h>

#define MAX\_N 100000

typedef struct {

int x, y;

} Point;

char S[MAX\_N+1];

Point P[MAX\_N\*2], offset[MAX\_N+1];

/\* N E S W \*/

int Dx[4] = { 0, +1, 0, -1};

int Dy[4] = { +1, 0, -1, 0};

int right\_turn(int dir) { return (dir+1)%4; }

int left\_turn(int dir) { return (dir+3)%4; }

int rotate\_x(int dir, Point p) {

if (dir==0) return p.x;

if (dir==1) return p.y;

if (dir==2) return -p.x;

if (dir==3) return -p.y;

}

int rotate\_y(int dir, Point p) {

if (dir==0) return p.y;

if (dir==1) return -p.x;

if (dir==2) return -p.y;

if (dir==3) return p.x;

}

/\* Sort by x, breaking ties by y \*/

static int pcomp(const void \*p1, const void \*p2)

{

Point \*q1 = (Point \*)p1;

Point \*q2 = (Point \*)p2;

if (q1->x == q2->x)

return q1->y - q2->y;

return q1->x - q2->x;

}

int main(void)

{

int i, L, total=0, x=0, y=0, dir=0, n=0;

freopen ("wrongdir.in", "r", stdin);

freopen ("wrongdir.out", "w", stdout);

scanf ("%s", S);

L = strlen(S);

/\* Compute action of every "suffix" of S \*/

for (i=L-1; i>=0; i--) {

if (S[i]=='F') { offset[i].x = offset[i+1].x; offset[i].y = 1 + offset[i+1].y; }

if (S[i]=='L') { offset[i].x = -offset[i+1].y; offset[i].y = offset[i+1].x; }

if (S[i]=='R') { offset[i].x = offset[i+1].y; offset[i].y = -offset[i+1].x; }

}

/\* Build a list of all possible destination points \*/

for (i=0; i<L; i++) {

if (S[i]!='F') {

P[n].x = x + Dx[dir] + rotate\_x(dir, offset[i+1]);

P[n].y = y + Dy[dir] + rotate\_y(dir, offset[i+1]);

n++;

}

if (S[i]!='L') {

P[n].x = x + rotate\_x(left\_turn(dir), offset[i+1]);

P[n].y = y + rotate\_y(left\_turn(dir), offset[i+1]);

n++;

}

if (S[i]!='R') {

P[n].x = x + rotate\_x(right\_turn(dir), offset[i+1]);

P[n].y = y + rotate\_y(right\_turn(dir), offset[i+1]);

n++;

}

if (S[i]=='F') { x += Dx[dir]; y += Dy[dir]; }

if (S[i]=='L') { dir = left\_turn(dir); }

if (S[i]=='R') { dir = right\_turn(dir); }

}

/\* Sort and count unique points \*/

qsort (P, 2\*L, sizeof(Point), pcomp);

for (i=0; i<2\*L; i++)

if (i==0 || P[i].x!=P[i-1].x || P[i].y!=P[i-1].y) total++;

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

return 0;

}

## USACO 2012 US Open, Bronze Division

Problem 1: Cows in a Row [Brian Dean, 2012] Farmer John's N cows (1 <= N <= 1000) are lined up in a row. Each cow is identified by an integer "breed ID"; the breed ID of the ith cow in the lineup is B(i). FJ thinks that his line of cows will look much more impressive if there is a large contiguous block of cows that all have the same breed ID. In order to create such a block, FJ decides remove from his lineup all the cows having a particular breed ID of his choosing. Please help FJ figure out the length of the largest consecutive block of cows with the same breed ID that he can create by removing all the cows having some breed ID of his choosing. PROBLEM NAME: cowrow INPUT FORMAT: \* Line 1: The integer N. \* Lines 2..1+N: Line i+1 contains B(i), an integer in the range 0...1,000,000. SAMPLE INPUT (file cowrow.in): 9 2 7 3 7 7 3 7 5 7 INPUT DETAILS: There are 9 cows in the lineup, with breed IDs 2, 7, 3, 7, 7, 3, 7, 5, 7. OUTPUT FORMAT: \* Line 1: The largest size of a contiguous block of cows with identical breed IDs that FJ can create. SAMPLE OUTPUT (file cowrow.out): 4 OUTPUT DETAILS: By removing all cows with breed ID 3, the lineup reduces to 2, 7, 7, 7, 7, 5, 7. In this new lineup, there is a contiguous block of 4 cows with the same breed ID (7).

**Solution Notes:** This problem can be solved by "brute force", by simply trying to remove each possible cow ID from the line, checking after each one whether it gives the best answer (the longest consecutive block of equal cow IDs). Below is Travis Hance's solution using this idea. Although the running time of this method is O(N^2) (which is plenty fast for the limits in this problem), note that it is possible to solve the problem even faster, in only O(N) time. The idea behind the faster solution is to scan through the array in just one pass, remembering the two most recent distinct IDs you have seen, as well as a count of each one. For example, if the array is 31221254 and we are located at the third "2", then our current state will tell us that we have just scanned across three 2s and two 1s (giving a consecutive block size of 3, if we delete the 1s).

#include <cstdio>

int id[1005];

int get\_largest\_block(int n, int idignore) {

int maxBlockSize = 0;

int curBreed = -1;

int curSize = 0;

for(int i = 0; i < n; i++) {

if(id[i] != idignore) {

if(curBreed == id[i]) {

curSize++;

} else {

curBreed = id[i];

curSize = 1;

}

if(curSize > maxBlockSize)

maxBlockSize = curSize;

}

}

return maxBlockSize;

}

int main() {

freopen("cowrow.in","r",stdin);

freopen("cowrow.out","w",stdout);

int n;

scanf("%d", &n);

for(int i = 0; i < n; i++) {

scanf("%d", &id[i]);

}

int ans = 0;

for(int i = 0; i < n; i++) {

int size = get\_largest\_block(n, id[i]);

if(size > ans)

ans = size;

}

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

}

## Problem 2: Three Lines [Brian Dean, 2012] Farmer John wants to monitor his N cows (1 <= N <= 50,000) using a new surveillance system he has purchased. The ith cow is located at position (x\_i, y\_i) with integer coordinates (in the range 0...1,000,000,000); no two cows occupy the same position. FJ's surveillance system contains three special cameras, each of which is capable of observing all the cows along either a vertical or horizontal line. Please determine if it is possible for FJ to set up these three cameras so that he can monitor all N cows. That is, please determine if the N locations of the cows can all be simultaneously "covered" by some set of three lines, each of which is oriented either horizontally or vertically. [Note: programs that do nothing more than make random guesses about the output may be disqualified, receiving a score of zero points] PROBLEM NAME: 3lines INPUT FORMAT: \* Line 1: The integer N. \* Lines 2..1+N: Line i+1 contains the space-separated integer x\_i and y\_i giving the location of cow i. SAMPLE INPUT (file 3lines.in): 6 1 7 0 0 1 2 2 0 1 4 3 4 INPUT DETAILS: There are 6 cows, at positions (1,7), (0,0), (1,2), (2,0), (1,4), and (3,4). OUTPUT FORMAT: \* Line 1: Please output 1 if it is possible to monitor all N cows with three cameras, or 0 if not. SAMPLE OUTPUT (file 3lines.out): 1 OUTPUT DETAILS: The lines y=0, x=1, and y=4 are each either horizontal or vertical, and collectively they contain all N of the cow locations.

**Solution Notes:** There are several ways to approach this problem that lead to O(N) or O(N log N) solutions (O(N^2) isn't quite fast enough to solve all of the cases in time). To start with, there are really two interesting cases to check: either the points can all be covered by 3 horizontal lines, or by 2 horizontal lines and 1 vertical line. There are two other cases symmetric to these, but we can easily transform them into the two previous cases by swapping x and y for each point. In Richard's code below, he uses an STL map to store a "histogram" of how many times each distinct y coordinate appears, as well as the total number of distinct y coordinates. When a point is added to this data structure, its count is incremented (and if the count was previously zero, then we also increment the number of distinct y coordinates in total). When a point is removed from the data structure, its count is decremented (and if the count is now zero, we also decrement the total number of distinct y coordinates). Now to test if all our points can be covered with 3 horizontal lines, we add them all to our structure and check if the count of the total number of distinct y coordinates is at most 3. To test if 2 horizontal lines and 1 vertical line are sufficient, we sort the points on x, and for each distinct x coordinate, we temporarily remove all the points having that x coordinate and test if the number of distinct y coordinates drops to at most 2.

#include <cstdio>

#include <cstring>

#include <map>

#include <algorithm>

using namespace std;

const int MAXN = 1001000;

pair<int,int> lis[MAXN];

map<int, int> cou;

int distinct;

int n;

void inc(int x) {

if(cou[x] == 0) {

distinct++;

}

cou[x] = cou[x] + 1;

}

void dec(int x) {

cou[x] = cou[x] - 1;

if(cou[x] == 0) {

distinct--;

}

}

int moo() {

sort(lis, lis + n);

distinct = 0;

cou.clear();

for(int i = 0; i < n; ++i) {

inc(lis[i].second);

}

if(distinct <= 3) return 1;

int i = 0, i1 = 0;

while(i < n) {

while(i1 < n && lis[i].first == lis[i1].first) {

i1++;

}

for(int i2 = i; i2 < i1; ++i2) {

dec(lis[i2].second);

}

if(distinct <= 2) return 1;

for(int i2 = i; i2 < i1; ++i2) {

inc(lis[i2].second);

}

i = i1;

}

return 0;

}

int main() {

freopen("3lines.in", "r", stdin);

freopen("3lines.out", "w", stdout);

scanf("%d", &n);

for(int i = 0; i < n; ++i) {

scanf("%d%d", &lis[i].first, &lis[i].second);

}

if(moo()) {

puts("1");

}

else {

for(int i = 0; i < n; ++i) {

swap(lis[i].first, lis[i].second);

}

if(moo()) {

puts("1");

}

else {

puts("0");

}

}

return 0;

}

## Problem 3: Islands [Brian Dean, 2012] Whenever it rains, Farmer John's field always ends up flooding. However, since the field isn't perfectly level, it fills up with water in a non-uniform fashion, leaving a number of "islands" separated by expanses of water. FJ's field is described as a one-dimensional landscape specified by N (1 <= N <= 100,000) consecutive height values H(1)...H(n). Assuming that the landscape is surrounded by tall fences of effectively infinite height, consider what happens during a rainstorm: the lowest regions are covered by water first, giving a number of disjoint "islands", which eventually will all be covered up as the water continues to rise. The instant the water level become equal to the height of a piece of land, that piece of land is considered to be underwater. http://www.usaco.org/current/data/fig_islands.pngAn example is shown above: on the left, we have added just over 1 unit of water, which leaves 4 islands (the maximum we will ever see). Later on, after adding a total of 7 units of water, we reach the figure on the right with only two islands exposed. Please compute the maximum number of islands we will ever see at a single point in time during the storm, as the water rises all the way to the point where the entire field is underwater. PROBLEM NAME: islands INPUT FORMAT: \* Line 1: The integer N. \* Lines 2..1+N: Line i+1 contains the height H(i). (1 <= H(i) <= 1,000,000,000) SAMPLE INPUT (file islands.in): 8 3 5 2 3 1 4 2 3 INPUT DETAILS: The sample input matches the figure above. OUTPUT FORMAT: \* Line 1: A single integer giving the maximum number of islands that appear at any one point in time over the course of the rainstorm. SAMPLE OUTPUT (file islands.out): 4

**Solution Notes:** We can solve this problem in O(N log N) time by simulating the rising water line by visiting the cells of the landscape in increasing order by height. Let us think of the landscape initially as a large string of length N, where each character is either L (for land) or W (for water). Initially, this string is set to "LLLL..LLLL", since no land is covered by water. After sorting the cells in the landscape by height, let us now switch the Ls to Ws as we visit cells in order of height, keeping a running count of the number of islands. Whenever we change LLL to LWL (i.e., when we change an L to a W and the neighboring cells are both L), we increment the island count, since we have split one island into two. Whenever we change WLW to WWW (i.e., when we change an L to a W and the neighboring cells are both W), we decrement the island count, since we have destroyed an island.

#include <cstdio>

#include <algorithm>

#include <cstring>

using namespace std;

#define nmax (5 + 100000)

struct land {

int x, h;

};

inline bool operator<(land a, land b) {

return a.h < b.h;

}

land lands[nmax];

bool underwater[nmax];

int main() {

freopen("islands.in","r",stdin);

freopen("islands.out","w",stdout);

int n;

scanf("%d", &n);

for(int i = 0; i < n; i++) {

scanf("%d", &lands[i].h);

lands[i].x = i;

}

memset(underwater, 0, n);

sort(lands, lands + n);

int numIslands = 1;

int maxIslands = 1;

for(int i = 0; i < n; i++) {

int x = lands[i].x;

underwater[x] = true;

bool landToLeft = (x > 0 && !underwater[x-1]);

bool landToRight = (x < n-1 && !underwater[x+1]);

if(landToLeft && landToRight) {

numIslands++;

}

else if(!landToLeft && !landToRight) {

numIslands--;

}

if((i == n-1 || lands[i+1].h > lands[i].h) && numIslands > maxIslands) {

maxIslands = numIslands;

}

}

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

}

## Problem 4: Unlocking Blocks [Brian Dean, 2012] A little-known fact about cows is that they love puzzles! For Bessie's birthday, Farmer John gives her an interesting mechanical puzzle for her to solve. The puzzle consists of three solid objects, each of which is built from 1x1 unit squares glued together. Each of these objects is a "connected" shape, in the sense that you can get from one square on the object to any other square on the object by stepping north, south, east, or west, through squares on the object. An object can be moved by repeatedly sliding it either north, south, east, or west one unit. The goal of the puzzle is to move the objects so that they are separated -- where their bounding boxes are disjoint from each-other. Given the shapes and locations of the three objects, your task is to help Bessie decide if they can be separated or not. A configuration that is non-separable is said to be locked. http://www.usaco.org/current/data/fig_unlock.png[Note: programs that do nothing more than make random guesses about the output may be disqualified, receiving a score of zero points] PROBLEM NAME: unlock INPUT FORMAT: \* Line 1: Three space-separated integers: N1, N2, and N3, describing respectively the number of unit squares making up objects 1, 2, and 3. \* Lines 2..1+N1: Each of these lines describes the (x,y) location of the south-west corner of single square that is part of object 1. All coordinates lie in the range 0..9. \* Lines 2+N1..1+N1+N2: Each of these lines describes the (x,y) location of the south-west corner of single square that is part of object 2. All coordinates lie in the range 0..9. \* Lines 2+N1+N2..1+N1+N2+N3: Each of these lines describes the (x,y) location of the south-west corner of single square that is part of object 3. All coordinates lie in the range 0..9. SAMPLE INPUT (file unlock.in): 12 3 5 0 0 1 0 2 0 3 0 3 1 0 1 0 2 0 3 0 4 1 4 2 4 3 4 2 1 2 2 1 2 2 3 3 3 4 3 4 4 4 2 INPUT DETAILS: Object 1 is made from 12 squares, object 2 is made from 3 squares, and object 3 is made from 5 squares. The shapes of the objects are those in the figure above. OUTPUT FORMAT: \* Line 1: Output 1 if the objects can be separated from each-other, or 0 if they are locked. SAMPLE OUTPUT (file unlock.out): 1

**Solution Notes:** This problem is solved via recursive depth-first search: Our current state is described by 4 numbers, giving the (x,y) offset of objects 2 and 3 relative to their initial positions. Each move we make, we can change one of these numbers by +1 or -1 to slide either object 2 or 3 (as long as this doesn't create any overlap between the objects), or we can change both x offsets or both y offsets by +1 or -1 to simulate moving object 1 (since moving object 1 is equivalent to keeping object 1 fixed and moving both objects 2 and 3 in the opposite direction).

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <set>

#include <algorithm>

using namespace std;

int n[3];

int lis[3][100][2];

int ok[3][3][51][51];

int dir[4][2] = {-1, 0, 1, 0, 0, 1, 0, -1};

char seen[42][42][42][42];

set<pair<int, int> > occupied;

int check(int i1, int i2, int dx, int dy) {

occupied.clear();

for(int i = 0; i < n[i1]; ++i) {

occupied.insert(make\_pair(lis[i1][i][0], lis[i1][i][1]));

}

for(int i = 0; i < n[i2]; ++i) {

if(occupied.find(make\_pair(lis[i2][i][0] + dx, lis[i2][i][1] + dy)) != occupied.end()) {

return 0;

}

}

return 1;

}

int isOkPair(int i1, int i2, int dx, int dy) {

if(dx < -20 || dx > 20 || dy < -20 || dy > 20) return 1;

return ok[i1][i2][dx + 20][dy + 20];

}

int isOk(int dx1, int dy1, int dx2, int dy2) {

return isOkPair(0, 1, dx1, dy1) &&

isOkPair(0, 2, dx2, dy2) &&

isOkPair(1, 2, dx2 - dx1, dy2 - dy1);

}

const int LIM = 20;

char q[3000000][4];

int tail;

void add (int dx1, int dy1, int dx2, int dy2) {

if(dx1 < -LIM || dy1 < -LIM || dx1 > LIM || dy1 > LIM ||

dx2 < -LIM || dy2 < -LIM || dx2 > LIM || dy2 > LIM) return;

if(!isOk(dx1, dy1, dx2, dy2)) return;

if(seen[dx1 + LIM][dy1 + LIM][dx2 + LIM][dy2 + LIM]) return;

seen[dx1 + LIM][dy1 + LIM][dx2 + LIM][dy2 + LIM] = 1;

q[tail][0] = dx1;

q[tail][1] = dy1;

q[tail][2] = dx2;

q[tail][3] = dy2;

tail++;

}

int main() {

freopen("unlock.in", "r", stdin);

freopen("unlock.out", "w", stdout);

scanf("%d%d%d", &n[0], &n[1], &n[2]);

for(int i = 0; i < 3; ++i) {

for(int j = 0; j < n[i]; ++j) {

scanf("%d%d", &lis[i][j][0], &lis[i][j][1]);

}

}

for(int i1 = 0; i1 < 3; ++i1)

for(int i2 = 0; i2 < i1; ++i2)

for(int dx = -20; dx <= 20; ++dx)

for(int dy = -20; dy <= 20; ++dy) {

ok[i2][i1][dx + 20][dy + 20] = check(i1, i2, dx, dy);

}

memset(seen, 0, sizeof(seen));

tail = 0;

add(0, 0, 0, 0);

for(int i = 0; i < tail; ++i) {

int dx1 = q[i][0];

int dy1 = q[i][1];

int dx2 = q[i][2];

int dy2 = q[i][3];

if(dx1 == LIM && dy1 == LIM && dx2 == -LIM && dy2 == -LIM) {

puts("1");

return 0;

}

for(int j = 0; j < 4; ++j) {

add(dx1 + dir[j][0], dy1 + dir[j][1], dx2, dy2);

add(dx1, dy1, dx2 + dir[j][0], dy2 + dir[j][1]);

add(dx1 + dir[j][0], dy1 + dir[j][1],

dx2 + dir[j][0], dy2 + dir[j][1]);

}

}

puts("0");

return 0;

}