Task Information for Cluedo

Task Author: Gordon Cormack (CAN)

This was intended to be a very easy task. The number of features to be determined (murderer, location, weapon), and the number of options for each feature were intentionally fixed, and not parameterized.

Given that there are 6 candidate murderers, 10 candidate locations, and 6 candidate weapons, there is a total of 6*10*6=360 theories.

Subtask 1 could be solved by trying each possible theory (three nested `for` loops).

Because the response to a refuted theory will identify one feature for which a wrong option was guessed, the search can be expedited. All theories having that wrong option for that particular feature are now ruled out.

Subtask 2 can be solved by a single loop, incrementing whichever feature was wrong (a monotonic search). The total number of options equals 6+10+6=22, and the last option not ruled out must be correct (it was given that there is exactly one correct theory). Therefore, at most 22-3=19 refuted calls to **Theory** are needed. One confirming call to **Theory** was required, so a total of 20 calls suffices.

Here is a Pascal solution that can readily be generalized (the constant, type, and auxiliary function definitions could be eliminated, but they document the relevant concepts nicely):

```
const
  NFeatures = 3; { number of features }
  Confirmed = 0; { result when theory is confirmed }

type
  TFeature = 1 .. NFeatures;
  TOption = 1 .. MaxInt; { value for a feature }
  TTheory = array [ TFeature ] of TOption;
  TResult = Confirmed .. NFeatures;

function TestTheory(T: TTheory): TResult;
  begin TestTheory := Theory(T[1], T[2], T[3]) end;

procedure Solve;
  var
    T: TTheory = (6, 10, 6); { candidate theory }
    i: TResult; { result of TestTheory(T) }
  begin
    repeat
      i := TestTheory(T)
    ; if i <> Confirmed then { T refuted } T[i] := T[i] - 1
    until i = Confirmed
    { T confirmed }
  end;
```

In C, without constant and type definitions, it could look like this:

```
void Solve() {
   int T[] = {0, 6, 10, 6}; // candidate theory, ignore T[0]
   int i; // result of Theory
   do {
     i = Theory(T[1], T[2], T[3]);
     if (i != 0) --T[i];
   } while (i != 0);
}
```

Task Information for Hotter Colder

Task Author: Gordon Cormack (CAN)

This problem is an interesting variant of the well-known guessing game *Higher-Lower*, also featured in the demonstration task *Guess*.

Higher-Lower is efficiently solved by the, also well-known, *Binary Search* algorithm. Binary Search maintains an interval of still possible numbers (candidates). Initially, this interval includes all numbers in the range. By comparing to the middle candidate, the interval can be halved by a single guess. Thus, the secret number can be determined in a logarithmic (to base 2) number of guesses. Or, to put it differently, if the range of allowed numbers is doubled, than the secret number can be determined with one additional guess.

Subtask 1

Doing a *Linear Search*, that is, successively calling **Guess(i)** for i from 1 to N, yields a solution requiring N calls to **Guess**, in the worst case. This solves Subtask 1. See below for a Pascal program.

Analysis

To get a better understanding of the Hotter-Colder problem, it helps to formalize the rules of this game.

Let J be Jill's number, and let P be the most recent guess, that is, **Guess(P)** was called last. In that situation, **Guess(G)** will return

```
HOTTER if abs(G - J) < abs(P - J)
COLDER if abs(G - J) > abs(P - J)
SAME   if abs(G - J) = abs(P - J)
```

Or in a single formula: `sign(abs(P - J) - abs(G - J))`. Letting `M = (P + G)/2`, this can be rephrased as

```
if P <= G then
  HOTTER if J > M
  COLDER if J < M
  SAME   if J = M
else
  HOTTER if J < M
  COLDER if J > M
  SAME   if J = M
```

Or in a single formula: `sign(G - P) * sign(J - M)`.

Thus, we see that each **Guess(G)** effectively provides a high-low comparison to the midpoint M. In fact, `sign(G - P) * Guess(G) = sign(J - M)` offers a genuine high-low comparison.

Unfortunately, due to the range restriction on G, we cannot make the midpoint M go wherever we want. So, a straightforward Binary Search is not going to work.

Subtask 2

Ignoring the results of all odd calls to **Guess**, we can extract one bit of information out of every successive pair of odd-numbered and next even-numbered call to **Guess**. This yields a solution that calls **Guess** at most W times, where W is the largest integer such that $2^{W/2} \leq N$. That is, it makes at most $\log_2 N^2$ (rounded up) calls to **Guess**. For N=500 (almost 2^9), this boils down to making at most 18 calls.

Subtask 3

By exploiting the fact that we actually do a high/low/equal comparison instead of a pure high/low (binary) comparison, we can gain almost one extra bit of information (taken over all guesses). Explanation: a complete binary tree with 2^k leaves has 2^k-1 internal nodes. So, the same number of high/low/equal guesses can reach twice the number of nodes minus one (compared to using just binary high/low guesses).

A Pascal program is given below.

Subtask 4

The preceding approaches obviously throw away (ignore) valuable information. However, using this information requires careful tuning of the guesses. It helps to do some small cases by hand.

- N=3 can obviously be done in 2 guesses, by straddling the middle, for example, **Guess(1)** followed by **Guess(3)** does a high/low/equal comparison to 2.
- N=5 can be done in 3, but this already needs some care, because it does not work to set this up so that the first two guesses compare to the middle number 3. When, after **Guess(1) Guess(5)**, or **Guess(2) Guess(4)**, the result of the second guess is *colder*, you won't be able to solve the remaining problem in a single guess.

  You need to start with **Guess(1) Guess(3)** (or symmetrically **Guess(5) Guess(3)**). If the result of the second guess is *same*, Jill's number is 2; if the result is *colder*, only candidate 1 remains and this must be Jill's number. If the result is *hotter*, candidates 3, 4, and 5 remain. Since 3 was the most recent guess, doing **Guess(5)** will compare to 4, and we are done.

In general, it turns out to be possible to determine Jill's number in no more than $\log_2 3*N = \log_2 3 + \log_2 N$ calls of **Guess**.

We explain one such algorithm. Because of the nature of the guess (being a comparison), at any moment you have an *interval* of remaining candidate numbers. You can distinghuish two cases for the location of this interval with respect to the initial interval:

1. either this interval of candidates contains 1 or N (is "against a wall");
2. or it contains neither 1 nor N (is "in the middle").

Furthermore, you know what the previous guess was, say P.

If the interval of candidates is "in the middle", then you are home free (provided you are a bit careful), because now each guess can be made to reduce the interval sufficiently. In K more guesses, you can find Jill's number among 2^{K+1}-1 candidates. [*Details suppressed (for the time being)*]

If the interval of candidates is "against a wall", then you can always arrange it so that the interval is 1 through P (or symmetrically on the other side). With two extra steps you can grow a solution that solves for P in K more guesses to one that solves for P+2^{K+2} in K+2 more guesses.

The base cases are P=3, K=1 and P=7, K=2.

The construction works like this. Consider the interval

```
aaaabbbbbbddddddddddd
```
where

- `aaaa` is the interval 1 through P (and we assume that if the most recent guess was at P, then an additional K guesses can determine Jill's number in this interval);
- `bbbbbb` is of length 2^{K+1}-2;
- `ddddddddddd` is of length 2^{K+1}+2;
- the most recent guess was R = P+2^{K+2}.

Your next guess is G = P-2:

```
aaaabbbbbbddddddddddd
1G P      M        R
```
This guess compares to M = (G+R)/2 = (P-2 + P+2^{K+2})/2 = P + 2^{K+1}-2 + 1, that is, the first element of the `d`-labeled subinterval. Do a case distinction on the result of this guess:

- *Same*: Jill's number is M; done.
- *Colder*: the interval is reduced to M+1 through R; continue with a "middle game" on `ddddddddd` of length 2^{K+1}+1;
- *Hotter*: the interval is reduced to 1 through M-1:
- `aaaabbbbbb`
- `1G P      M`

  Next, guess P, which boils down to comparing to (G+P)/2 = P-1. Do a case distinction on the result:

  - *Colder*: "wall game" on interval 1 through P (`aaaa`), which we assumed can be solved in K more guesses;
  - *Hotter*: "middle game" on abbbbbb of length 2^{K+1}-1.

A C program solving Subtask 4 can be found.

Pascal program for Linear Search solving Subtask 1

```
const
  Colder = -1;
  Same = 0;
  Hotter = +1;

type
  TResult = Colder .. Hotter;

function HC(N: Longint): Longint;
  { returns secret number of Jill }

  var
    r: TResult;  { result of Guess }
    G: Longint; { argument for Guess }

  begin
    if N = 1 then begin
      HC := N
    ; Exit
    end { if }
    { N >= 2 }
  ; G := 1
  ; r := Guess(G) { ignored }

  ; repeat
      { numbers >= G are remaining candidates; G < N }
      G := G + 1
    ; r := Guess(G) { compares to G - 0.5; r <> Same }
    until (r = Colder) or (G = N)

  ; case r of
      Colder: HC := G - 1;
      Hotter: HC := G;
    end { case r }
  end;
```

Pascal program for wasteful Binary Search solving Subtask 3

```
const
  Colder = -1;
  Same = 0;
  Hotter = +1;

type
  TResult = Colder .. Hotter;

function HC(N: Longint): Longint;
  { returns secret number of Jill }

  var
    r: TResult;  { result of Guess }
    a, b: Longint; { [a .. b] is interval of remaining candidates }

  begin
    if N = 1 then begin
      HC := N
    ; Exit
    end { if }
    { N >= 2 }

  ; a := 1
  ; b := N
```

```
  { invariant: 1 <= a <= b <= N }
; while a <> b do begin
    r := Guess(a) { ignored }
  ; r := Guess(b) { compares to (a+b)/2 }
  ; case r of
      Colder: b := (a + b - 1) div 2; { largest integer < (a+b)/2 }
      Same: begin a := (a + b) div 2 ; b := a end;
      Hotter: a := (a + b + 1) div 2; { smallest integer > (a+b)/2 }
    end { case r }
  end { while }
  { a = b }

; HC := a
end;
```

Task Information for Quality of Living

Task Author: Christopher Chen (AUS)

This problem looks like many other grid tasks. Such problems have also appeared on some previous IOIs. Heavy *range-search* algorithms might seem to be useful, but actually a much simpler 100% solution exists.

Let N = R*C measure the *size of a problem instance*.

Subtask 1

Subtask 1 can be solved by the most obvious *brute force* algorithm that considers each rectangle (there are (R-H+1)*(C-W+1) of these), quadratically sorts its contents ($(H*W)^2$ steps), and directly picks out the median rank, and optimizes this. The worst case situation is obtained by H=R/2 and W=C/2. Therefore, this algorithm's time complexity is $O(N^3)$.

Subtask 2

Using any O(N log N) sort algorithm (these are well-known), the brute force algorithm can be improved to $O(N^2 \log N)$. This solves subtask 2.

Also, an O(N) sort (*bucket sort*) is possible, to obtain a simple $O(N^2)$ algorithm, but this does not suffice to solve Subtask 3. See below for a Pascal implementation.

There are some obvious opportunities for improvement, such as exploiting the large overlap between certain rectangles when filling/emptying the array to be sorted. But these improvements do not affect the time complexity.

Subtask 3

$O(N^{1.5} \log N)$ algorithms are also possible and they solve Subtask 3. Here is one: say the vertical offset of the final rectangle is known [$O(N^{0.5})$ possibilities]. Then scan across the row, using some efficient data structure to keep track of the median (think of incremental/sliding window, such as a range tree plus binary search, or a pair of heaps for values less/greater than the median). [Each value is added/subtracted from the data structure exactly once, for an O(N log N) scan length.]

This is rather involved to code; see Subtask 5 for a simpler and better solution.

Subtask 4

This subtask accommodates possible O(N log N log N) algorithms, although we have not encountered them.

Subtask 5

Here is a O(N log N) solution. Observe that the program's output can be *verified* by some algorithm which answers the question "Does any rectangle have median ≤ X?" This query can be answered in $O(n^2)$ time. A rectangle has median ≤ X if and only if it contains more values ≤ X than otherwise. Assign all cells in the grid a 'value' according to a 'threshold' function: -1 if greater than X, 0 if equal to X, 1 if less than X. Using the well-known cumulative data structure for queries on rectangular sums, try all possible rectangle locations and return "yes" if the 'values' inside any sum to ≥ 0. We simply *binary search* values of X to find the minimum value for which the answer is "yes".

N.B. An O(N log N) algorithm suffices for 100% score.

Linear Solution

Mihai Patrascu (ROM) found an O(N) solution with the following reasoning.

Claim 1:

> Given a value X, one can identify in O(HW) time which rectangles have a median better than X.

Proof:

> In linear time, build a *partial-sums array*: A[i][j] = #{elements in Q[0..i][0..j] better than X }. The number of elements better than X in any rectangle can now be found by combining 4 values of A. The median of an H by W rectangle is better than X if and only if the number of elements better than X is less than HW/2. QED

Claim 2:

> One can find the best median of K designated rectangles, in running time $O(K^2 \log(HW) + HW)$.

Proof:

> Compress everything to a KxK grid and binary search for the best median. In each step of the binary search, I need to go over the entire KxK grid, and a number of elements that is originally HW, but decreasing geometrically each time. QED

Claim 3:

> One can find the best median of all rectangles in O(HW) time.

Proof:

By the above, one can set K = sqrt(HW) / log(HW) and still get linear time. Sample K rectangles; apply Claim 2. Apply Claim 1 to filter everybody with worse medians. One is left with HW/K rectangles. Do the same again, one is left with $HW/K^2 \le \log(HW)$ rectangles. Now just apply Claim 2. QED

$O(N^2)$ Solution for Subtask 2 in Pascal using bucket sort

```
const
  MaxDimension = 3000;
  MaxRank = sqr(MaxDimension);

type
  TCoordinate = 0 .. MaxDimension - 1;
  TRank = 1 .. MaxRank;
  Qtype = array [ TCoordinate, TCoordinate ] of TRank;

  TRankSet = array [ TRank ] of Boolean;

var
  sorttemp: TRankSet; { for bucket sorting and median finding,
    kept global for speed }

function rectangle(R, C, H, W: Longint; var Q: Qtype): TRank;
  { returns best median of all HxW rectangles in RxC city Q }

  function median(a, b: TCoordinate): TRank;
    { returns median rank in rectangle with top-left corner at [a, b] }
    var
      i, j: TCoordinate; { traverses Q }
      r: Longint; { traverses [ 0 ] + sorttemp }
      c: Longint; { counts elements in sorttemp <= r }
    begin
      { insert elements from rectangle into the buckets }
      for i := a to a + H - 1 do begin
        for j := b to b + W - 1 do begin
          sorttemp[ Q[i, j] ] := True
        end { for j }
      end { for i }

      { determine the median by counting off half the elements }
    ; r := 0 { r in [ 0 ] + sorttemp }
    ; for c := 1 to (H * W) div 2 + 1 do begin
        repeat r := r + 1 until sorttemp[r]
      end { for c }
    ; median := r

      { restore invariant for sorttemp, by removing the elements }
    ; for i := a to a + H - 1 do begin
        for j := b to b + W - 1 do begin
          sorttemp[ Q[i, j] ] := False
        end { for j }
      end { for i }
    end; { median }

  var
    i: TRank; { traverses sortemp for initialization }
    result: TRank; { best rank seen so far }
    a, b: TCoordinate; { traverses all rectangles }
    m: TRank; { median of rectangle with top-left corner at [a, b] }
```

```
begin
  for i := 1 to R * C do sorttemp[i] := False
; result := MaxRank;

; for a := 0 to R - H do begin
    for b := 0 to C - W do begin
      m := median(a, b)
    ; if m < result then result := m
    end { for b }
  end { for a }

; rectangle := result;
end; { rectangle }
```

Task Information for Language

Task Author: Gordon Cormack (CAN)

The nature of this problem is innovative within the IOI. Its purpose is to bring the field of *information retrieval* under the attention. This problem is discussed in detail in the book *Information Retrieval: Implementing and Evaluating Search Engines* by S. Büttcher, C.L.A. Clarke, and G.V. Cormack (MIT Press, to appear soon). Especially see Chapter 10 on Categorization and Filtering.

One important observation is that excerpts from the same language version of Wikipedia will share some characteristics in a statistical sense. Because many random excerpts are offered, the variability between excerpts from the same language play a negligible role. It has been confirmed that the statistical resemblance between the provided test input and the official grader input is highly predictable.

Note that because of the random re-coding of the language codes and symbol codes, there is no opportunity to hard code any specific (personal) language knowledge into a solution.

There are many approaches possible. *Rocchio's method*, which was informally described in the task description, suffices to solve Subtask 1.

For Subtask 2, one needs to do more than simply look at symbol frequencies. Collecting statistics on *bigrams* (pairs of neighboring symbols), *trigrams* (three consecutive symbols) will yield higher accuracies.

Task Information for Memory

Task Author: Gordon Cormack (CAN)

This was intended to be another very easy task, though slightly more difficult than the one on Day 1 when aiming for a full score.

Turning each possible pair of cards face up in some sequence is guaranteed to obtain all 25 candies. There are 50-choose-2 = 50 * 49 / 2 = 1225 such pairs. Hence, doing 2450 card turns (that is, calls to **faceup**) suffices. This can be programmed with two nested `for`-loops, and it solves Subtask 1

But it does not solve Subtask 2, where no more than 100 card turns are allowed. Note that the try-all-pairs solution does not look at what is on the cards that are turned face up. That is, it does not make use of the values returned by **faceup**. By using these returned values, you can gather information that can be used later to reduce the number of cards turned up.

In particular, taking this to an extreme, you can first turn all cards, in pairs, to discover and record where all the letters are, without caring about turning up equal pairs. In this first round, you might already obtain some candies by accident, but that is irrelevant. In the next round, you know where equal pairs are and you can flip them, in sequence, to obtain all remaining candies.

The first round requires 50 turns (calls to **faceup**), and the second round another 50 turns. Thus, altogether 100 times a card is turned, and thereby Subtask 2 is solved.

In the second round, you could skip equal pairs that were already identified in the first round. However, that will not improve the worst-case performance and it will complicate the coding.

Here is a Pascal solution that can readily be generalized (the constant and type definitions could be eliminated, but they document the relevant concepts nicely):

```
type
  TCard = 'A' .. 'Y'; { which letters appear on the cards }

const
  NLetters = Ord(High(TCard)) - Ord(Low(TCard)) + 1; { number of letters }
  NCardsPerLetter = 2; { number of cards per letter }
  NCards = NCardsPerLetter * NLetters; { number of cards }
  Unknown = 0; { when card index is unknown }

type
  TIndex = 1 .. NCards; { index of card }

procedure play;

  var
    index: array [ TCard, 1 .. NCardsPerLetter ] of Unknown .. NCards;
      { index[lt, k] = index of k-th card with letter lt }
    lt: TCard; { traverses index }
```

```
    k: 1 .. NCardsPerLetter; { traverses index }
    i: TIndex; { traverses cards }
    r: TCard; { result of faceup }

  begin

    { initialize index to Unknown }
    for lt := Low(TCard) to High(TCard) do begin
      for k := 1 to NCardsPerLetter do begin
        index[lt, k] := Unknown
      end { for k }
    end { for lt }
  ;
    { first round: don't care about candy; discover where all letter are }
    for i := 1 to NCards do begin
      r := faceup(i)
    ; k := 1
    ; while index[r, k] <> Unknown do k := k + 1
    ; index[r, k] := i
    end { for i }
  ;
    { second round: now collect all (remaining) candies }
    for lt := Low(TCard) to High(TCard) do begin
      for k := 1 to NCardsPerLetter do begin
        r := faceup( index[lt, k] ) { ignore result }
      end { for k }
    end { for lt }

  end;
```

In C, without constant and type defintions, it could be coded as follows:

```
void play() {
  // letters 'A' to 'Y' are converted to integers 0 to 24
  int index[50][2]; // locations of cards; 0 = unknown
  int lt, k; // traverses index
  int i; // traverses cards
  char r; // result of faceup

  // initialize index
  for (lt = 0; lt < 25; ++lt) {
    for (k = 0; k < 2; ++k) {
      index[lt][k] = 0;
    }
  }
  // first round
  for (i = 1; i <= 50; ++i) {
    r = faceup(i);
    lt = (int)(r) - (int)('A'); // int corresponding to char r
    k = (index[lt][0]) ? 1 : 0;
    index[lt][k] = i;
  }
  // second round
  for (lt = 0; lt < 25; ++lt) {
    faceup( index[lt][0] ); // result ignored
    faceup( index[lt][1] ); // result ignored
  }
}
```

Task Information for Traffic Congestion

Task Author: Jorge Bernadas (VEN)

This was (by intention) a fairly standard task. Though, it should be mentioned that graph problems always are a bit trickier than one might at first think because of the need to handle specific graph encodings.

The information provided below will be expanded in the future, but for now should help in understanding what each subtask was expecting in the form of algorithms.

- Subtask 1: Quadratic works. Because of the highly regular (linear) structure of the network graph, it is easy to try each city as location for the arena, calculate the worst congestions and pick out the location where this worst congestion is minimal.
- Subtask 2: Requires linear algorithm, but because there are only two leaves and the graph representation is highly regular, it is easy to see that one sweep over the cities along the roads suffices to determine the optimum location.
- Subtask 3: Quadratic works, but now the general graph must be handled. Again, as in Subtask 1, every city can be tried as arena location, the worst congestion can then be calculated, and best location can be found.
- Subtask 4: This is the full problem. A linear traversal of the graph, accumulating congestion information appropriately, enables one to determine the optimal location of the arena in linear time.

Task Information for Maze

Task Authors: Monika Steinová (CHE/SWK), Michal Forišek (SWK)

This is known to be a hard problem. No general polynomial algorithm is known for this problem, that is, no algorithm has been discovered that is guaranteed to solve each problem instance in a time polynomial in the size of the problem (area of the corn field).

This is also the reason why this task was offered as an output-only task, where the contestants were given 10 specific instances (corn fields) to tackle. They could do this by hand, or write programs to analyze these mazes, and write yet other programs (potentially, a different program for each field) to produce a long(est) path. Note that the path need not be optimal; points could also be scored for (good) approximations.

Here are some characteristics and results by Tor Myklebust (HSC member) for the 10 instances that the contestants had to tackle:

	Characteristics		Tor's result	
Instance	**Dimensions**	**Obstructions**	**Path length**	**Score**
1	10x10	8	20	10.00
2	100x100	1766	4026	10.15
3	100x100	3216	3740	8.61
4	100x100	2283	3733	8.58
5	100x100	1357	4738	8.86
6	11x11	0	54	10.00
7	20x20	210	33	10.00
8	20x20	122	95	10.00
9	11x21	10	106	10.45
10	200x200	15224	7506	9.17
Total				**95.82**

Task Information for Saveit

Task Author: Mihai Patrascu (ROM)

This task also is innovative for the IOI. In general, for most IOI tasks efficiency matters. However, in this case it is not execution time or memory usage but rather *communication efficiency*: how to represent some complex data in as few bits as possible, without losing information.

This difference in focus makes the tasks possibly somewhat harder to understand. Furthermore, it is technically more complicated, because the contestant has to program two independent procedures that are inverses to each other. The communication format is not prescribed; all that matters is that the decoder programmed by the contestant can decode the data from the encoder that is also programmed by the contestant. The grading server then connects these two procedures to verify that the decoder can indeed "understand" what the encouder produced.

Two things are important. First, find a way to encode adjacency information about Xedef's package transportation network. Second, to transmit that information with communication efficiency.

Briefly stated, the subtasks could be tackled as follows:

- Subtask 1: You can send the entire adjacency matrix "as is"; this information is naturally expressed in terms of bits, other encodings are imaginable as well. All that the encoder and decoder need to agree upon is the order of the bits. Since there are 1000 cities, this requires no more than 1000*1000=1 000 000 bits. Other approaches using more bits also work in this subtask.
- Subtask 2: You can send the entire table with all hop counts directly. Since there are no more than 1000 cities, the maximum hop count is less than 1000, and thus can be encoded in 10 bits. The size of the table is at most 1000*36=36 . Hence, not more than 360 000 bits are needed.
- Subtask 3: One needs a new idea to improve the communication efficiency further. The crux is to come up with the idea of considering a spanning tree; any spanning tree will do. The distance from v1 to v2 is one of
    1. distance from parent(v1) to v2
    2. 1 + distance of parent(v1) to v2
    3. -1 + distance of parent(v1) to v2

    Then all one has to do is encode these possibilities with 2 bits each.

- Subtask 4: Using two bits to record a one-of-three choice is excessive. It is possible to map 3 ternary decisions (27 choices) to 5 bits (32 possibilities). This improves the communication further.