SRM 261 problemset

Michal Forišek

<misof@mfnotes.ksp.sk>
Department of Computer Science
Faculty of Mathematics, Physics, and Informatics
Comenius University, Bratislava, Slovakia

August 2005

This is a set of problems I created for TopCoder's Algorithm Competition, Single Round Match (SRM) 261.

The problems are presented in (subjectively) increasing order of difficulty. The actual point values used in the SRM are given in the following table.

	Division 1	**Division 2**
Spreadsheet Column		250
Prime Statistics	250	500
Gomoku Board Checker		1000
Encoding Trees	550	
Stacking Boxes	1000	

Contents

1 Problem statements

1.1 Spreadsheet Column

Problem Statement

Many spreadsheet applications use positive integers to label rows of cells and strings to label columns of cells. Your task is to write a function that gets the number of the column and returns its label.

All 26 uppercase letters are used to label the columns. Column labels are ordered according to their length, and labels with the same length are ordered in alphabetical order. Thus, the first 26 columns have a one-letter label, the following 26×26 columns have a two-letter label. The sequence of the labels looks as follows:

A, B, C, ..., Z, AA, AB, ..., AZ, BA, BB, ..., ZY, ZZ

The columns are numbered from 1, i.e., column number 1 has the label A.

Notes

The constraints will guarantee that the column label is between "A" and "ZZ", inclusive

Constraints

column will be between 1 and 702, inclusive.

Examples

input	output
1	"A"
input	output
2	"B"
input	output
27	"AA"
input	output
111	"DG"

input	output
702	"ZZ"

Note that this is the largest possible input.

1.2 Prime Statistics

Problem Statement

It is a known fact that prime numbers are not evenly distributed. For example, almost all primes give the remainder 1 or 5 when divided by 6 (the only two exceptions are the primes 2 and 3). Your task is to write a program that will help explore this phenomenon.

You will be given three integers: **lowerBound**, **upperBound** and **modulo**. Return the remainder that occurs most often when we take all primes in the set {**lowerBound**, **lowerBound** + 1, . . . , **upperBound**} and divide each of them by **modulo**. If there are multiple remainders that occur most often, return the smallest of them.

Notes

A prime number is a positive integer that has exactly two divisors. The first few primes are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, . . .

Constraints

lowerBound and **upperBound** are between 1 and 200,000 inclusive.

lowerBound is less than or equal to **upperBound**.

modulo is between 2 and 1000 inclusive.

Examples

input	output
3 14 5	3

The primes in this interval are: 3, 5, 7, 11 and 13. Their remainders when divided by 5 are 3, 0, 2, 1 and 3, respectively. Thus the most common remainder is 3.

input

```
3
33
1000
```

output

```
3
```

In this case each of the primes gives a different remainder. According to the tie-breaking rule the smallest of them is returned.

input

```
25
27
17
```

output

```
0
```

There are no primes in this interval. Each remainder occurs zero times, thus each of them is the most common remainder. Zero is the smallest possible remainder. Thus, according to the tie-breaking rule, zero is returned.

input

```
1
200000
2
```

output

```
1
```

Almost all primes are odd; the only even prime is 2.

input

```
1
1000
6
```

output

```
5
```

As mentioned in the introduction, almost all primes give the remainder 1 or 5 modulo 6. In this interval there are more primes of the second kind.

1.3 Gomoku Board Checker

Problem Statement

Gomoku, also called go-moku (Japanese: Gomoku Narabe, "five points") is a tic-tac-toe-like game played on a rectangular board divided into unit squares. There are two players, and each of them has an infinite supply of pieces of his color. They alternately place their pieces on the board in an attempt to place five (or more) of their own pieces in a row (horizontally, vertically, or diagonally). As soon as one of the players has reached this goal, the game ends and he is the winner. If and only if no player has reached the goal and the whole board is full, the game is a draw.

You will be given a **board** containing a position in the game, with the letter 'X' used to denote pieces of one player, the letter 'O' (not zero!) the pieces of the other player and '.' (period)

to denote empty spaces. (Note that you don't know whether the player that started the game used 'O's or 'X's.)

Your task is to write a function that decides:

- whether this is a valid position (i.e., one that could arise if both players played according to the rules), and
- whether a player already won.

If the position is not valid, return the string "`INVALID`". If the position is valid and represents a draw game, return the string "`DRAW`". If the position is valid and one of the players won, return "`X WON`" or "`O WON`" respectively. Otherwise, return the string "`IN PROGRESS`".

Notes

Either of the players could have started the game.

Each square of the board may contain at most one piece.

Constraints

board will contain between 1 and 15 elements, inclusive.

Each element of **board** will contain between 1 and 15 characters, inclusive.

Each element of **board** will contain the same number of characters.

Each element of **board** will contain only the characters 'X', 'O' and '.' .

Examples

input

```
{"O.X..",
 ".OX..",
 "X.O..",
 "X..O.",
 "....O"}
```

output

```
"O WON"
```

Clearly, player O started the game and won.

input

```
{"O.X..",
 ".OX..",
 "X.O..",
 "X..O.",
 "...XO"}
```

output

```
"O WON"
```

This time, player X started the game, but player O won.

input

```
{"OOOOO",
 "OOOOO",
 "OOOOO",
 "OOOOO",
 "OOOOX"}
```

output

```
"INVALID"
```

This is an invalid position because players have to alternate placing the pieces.

input

```
{"O...",
 "...X",
 "....",
 "...."}
```

output

```
"IN PROGRESS"
```

According to the problem statement, this is a game in progress (even though nobody can win on a 4x4 board).

input

```
{"O.X.O.",
 ".OX.O.",
 "X.O.O.",
 "X..OO.",
 "..XXOX",
 "XXXXOO"}
```

output

```
"O WON"
```

input

```
{"O.XX.O",
 ".OX..O",
 "X.O..O",
 "X..O.O",
 "..XXOO",
 "XXXX.O"}
```

output

```
"INVALID"
```

input

```
{"XXXXO",
 "OOOOX",
 "XXXXO",
 "OOOOX",
 "XOXOX"}
```

output

```
"DRAW"
```

1.4 Encoding Trees

Problem Statement

A binary tree is either empty or it consists of a root node and two binary trees, called the left subtree and the right subtree of the root node. Each node of our binary trees will contain one lowercase letter. We say that a binary tree is a binary search tree (BST) if and only if for each node the following conditions hold:

- All letters in the left subtree of the node occur earlier in the alphabet than the letter in the node.
- All letters in the right subtree of the node occur later in the alphabet than the letter in the node.

Note that if a tree is a BST, then each subtree of this tree is also a BST. As a consequence, if the tree is non-empty, then both subtrees of the root node are BSTs again.

Examples of BSTs with 4 nodes:

```
    c        b        a
   / \      / \        \
  b   d    a   d        c
 /           /        / \
a           c        b   d
```

A pre-order code of a BST is a String obtained in the following way:

- The pre-order code of an empty BST is an empty string.
- The pre-order code of a non-empty BST is obtained in the following way: Let L and R be the pre-order codes of the left and right subtree, respectively. Then the pre-order code of the whole BST is the concatenation of the letter in its root node, L and R (in this order).

The pre-order codes for the trees above are "cbad", "badc" and "acbd", respectively.

Consider all BSTs with exactly **N** nodes containing the first **N** lowercase letters. Order these trees alphabetically by their pre-order codes. Our sequence of BSTs is one‘-based, i.e., the index of the first tree in this sequence is 1. Return the pre-order code of the BST at the specified **index** in this sequence. If **index** is larger than the number of BSTs with exactly **N** nodes, return an empty string.

Notes

You may assume that the number of our BSTs with 19 nodes doesn't exceed 2,000,000,000.

Constraints

N is between 1 and 19, inclusive.

index is between 1 and 2,000,000,000, inclusive.

Examples

input

```
2
1
```

output

```
"ab"
```

There are 2 BSTs with 2 nodes. The first of them is:

```
a
 \
  b
```

input

```
2
2
```

output

```
"ba"
```

The second one is

```
  b
 /
a
```

input	output
2 3	""

There are only 2 BSTs with 2 nodes, so the empty string is returned for an index of 3.

input	output
4 9	"cbad"

The 14 valid pre-order codes of BSTs with 4 nodes: abcd, abdc, acbd, adbc, adcb, bacd, badc, cabd, cbad, dabc, dacb, dbac, dcab, dcba. The 9th tree:

```
    c
   / \
  b   d
 /
a
```

input	output
15 14023	"abcdeohgfniljkm"

1.5 Stacking Boxes

Problem Statement

Yesterday I was cleaning my house and I made a startling discovery. In the corner of the living room stood a nice decorated Christmas tree. The next Christmas is still too far away, thus I decided to remove all the decorations, put them into cardboard boxes and store them in the garage. However, the garage is almost full of other stuff. Therefore I'd like to arrange the boxes to form a tall stack, one atop another.

This may not be possible. Christmas tree decorations are fragile and the boxes that contain them aren't exactly made of steel. I weighed each of the boxes and for each of them I estimated the maximum weight that can be placed on the top of the box without it collapsing. In the following text we will use the term carrying capacity of a box when referring to this maximum weight.

You will be given two sequences of integers: **weight** and **canCarry**. Each element of **weight** will be the weight of one of the boxes. Similarly, each element of **canCarry** will be the carrying capacity of one of the boxes. The carrying capacities of the boxes will be given in the same order as their weights.

Your task is to find and return the largest N such that N of the boxes can be selected and placed one atop another in some order such that none of the boxes collapse.

Constraints

weight will contain between 1 and 1500 elements, inclusive.

canCarry will contain between 1 and 1500 elements, inclusive.

weight and **canCarry** will contain the same number of elements.

Each element of **weight** will be between 1 and 100,000, inclusive.

Each element of **canCarry** will be between 1 and 1,000,000,000, inclusive.

Examples

input

```
{10,20,30}
{11,100,10}
```

output

```
3
```

Here we are given 3 boxes. The first one has weight 10 and can carry 11, the second one has weight 20 and can carry 100, the third one has weight 30 and can carry 10. It is possible to create a stack using all three of them: the first box goes on the top, the third one below and the second one on the bottom.

input

```
{11,20,30}
{11,100,10}
```

output

```
2
```

The first box is now too heavy, so the previous arrangement doesn't work anymore.

input

```
{10,20,91}
{11,100,10}
```

output

```
2
```

Again, the original arrangement doesn't work anymore. This time boxes 1 and 3 together are too heavy – box 2 won't be able to carry both of them.

input

```
{100000}
{1000000000}
```

output

```
1
```

You can always use at least one box, as it doesn't have to carry anything.

input

```
{100,100,1000,100}
{90,91,92,93}
```

output

```
1
```

Each of the boxes is too heavy to be placed on any other box.

input

```
{200,200,600,700,400}
{1000,20,150,700,10}
```

output

```
3
```

2 Solutions

2.1 Spreadsheet Column

This problem was pretty simple. To avoid any confusion, probably the best way is to handle the two cases separately. As long as you tested your solution on all examples, you should've been on the safe side.

```
string result;
if (column <= 26)
  result += char('A'+column-1);
else {
  column -= 27;
  result += char('A' + (column/26));
  result += char('A' + (column%26));
}
```

2.2 Prime Statistics

The first step to solve this problem is (of course) being able to say whether a given number is prime. Either you could precompute this using the sieve of Erathostenes, or you could write a simple function that checks whether a number is prime.

How to do this? If a number N is not prime, it has a non-trivial divisor. Let D be the smallest of them. Clearly $D \geq 2$. The number N/D is also a non-trivial divisor of N. But D is the smallest one, therefore $D \leq N/D$. In other words, $D^2 \leq N$. A simple way of primality testing follows:

```
bool isPrime(int N){
  if (N<2) return false;
  for (int i = 2; i*i <= N; i++){
    if (N%i == 0) return false;
  }
  return true;
}
```

The time complexity of this function is clearly $O(\sqrt{N})$. Note that the upper bound for the cycle is often written as "`i <= sqrt(N)`", but our way is a bit faster and we avoid using real numbers.

Now all you had to do was to loop through the given interval, for each number check whether it is prime and for each prime compute its remainder.

There were lots of successful challenges for this problem. The problem with most of the flawed solutions is that they considered 1 to be prime. Note that 1 is never considered to be a prime number. This was also clearly stated in the problem statement... but avoided in the examples. Many coders spotted this and were rewarded in the challenge phase.

2.3 Gomoku Board Checker

Algorithmically there's nothing that difficult in this problem. But there were many cases to consider... and not all of them were covered by the examples. This made the problem pretty tricky, with only two solutions handling all the cases correctly.

Clearly, we need a systematic way of approaching the problem. First of all, let's count the number of 'O's and the number of 'X's on the board. If these counts differ by more than 1, the position is illegal. If they differ by 1, we know, who made the last move. If they are equal, either player could make the last move.

Now we check whether each player has got a 5-in-a-row. The easiest way: Loop through all squares, loop through all 8 directions and check, whether there's a 5-in-a-row starting on the chosen square and going in the chosen direction.

```
// constants for the 8 directions
int dr[] = { -1, -1, -1, 0, 0, 1, 1, 1 };
int dc[] = { -1, 0, 1, -1, 1, -1, 0, 1 };

// the number of rows and columns
int R, C;

bool inside(int x, int y) { return (x>=0 && x<R && y>=0 && y<C); }

bool hasFive (char who, vector<string> board) {
  for (int r=0; r<R; r++)
    for (int c=0; c<C; c++)
      for (int d=0; d<8; d++)
        if (inside( r+4*dr[d] , c+4*dc[d] )) {
          bool ok=true;
          for (int k=0; k<5; k++)
            if (board [ r+k*dr[d] ][ c+k*dc[d] ] != who) { ok=false; break; }
          if (ok) return true;
        }
  return false;
}
```

If both players have got a 5-in-a-row, the position is invalid. If nobody has 5-in-a-row, the position is either a game in progress, or a draw. To distinguish between these two cases, we simply check whether the board is full.

We are left with the case that exactly one player has got some 5-in-a-rows. If she couldn't make the last move, the position is invalid. If she could, she had to make all of them in her last move. (The game has to end immediately after one of the players won.) The easiest way to check whether this is possible is brute force: For each square with her symbol try whether this could be the last move (i.e., check whether the board with this symbol removed contains any 5-in-a-rows). If no possible last move is found, the position is invalid, otherwise the player won.

2.4 Encoding Trees

In the whole text, the abbreviation BST means "a binary search tree with N nodes labeled from 0 to $(N-1)$". (In the problem statement the nodes were labeled by lowercase letters, but dealing with numbers will simplify this text a little bit. Converting numbers to letters is trivial.) Also, we will assume that the trees are indexed from 0.

Counting all BSTs with N nodes is a pretty standard combinatorial tasks. Let $C(N)$ be the count of BSTs with N nodes, let $C(N,K)$ be the count of BSTs with N nodes with the label K in the root node. How to compute $C(N,K)$? The left subtree contains K nodes (labeled from 0 to $K-1$) and the right one contains $N-1-K$ nodes (labeled from $K+1$ to $N-1$). The left subtree is again a BST, thus there are $C(K)$ possible left subtrees. If we subtract $(K+1)$ from each label in the right subtree, we get another BST – and vice versa, from each BST on $N-1-K$ nodes we can create a possible right subtree by adding $(K+1)$ to each label. Thus there are $C(N-1-K)$ possible right subtrees.

Clearly, each pair (left subtree, right subtree) corresponds to one BST. We get the equation:

$$C(N,K) = C(K) \cdot C(N-1-K)$$

and we may compute the total count of BSTs with N nodes by summing through all possible labels of the root node:

$$C(N) = \sum_{K=0}^{N} C(K) \cdot C(N-1-K)$$

(As a side note, the numbers $C(N)$ are known under the name Catalan numbers and they can be found in many different places of combinatorics.)

After computing all the numbers $C(N,K)$ the reconstruction of a tree from its index is straightforward. In fact, the approach we use is the most common way of solving such combinatorial tasks.

First, we need to determine the label of the root node (which is also the first label in the preorder code of the tree). This can be done by counting the trees with the root node having the label 0, 1, etc.: The trees with indices from 0 to $C(N,0)-1$ have the label 0, trees from $C(N,0)$ to $C(N,0)+C(N,1)-1$ have the label 1, etc.

Now we have the label K of the root node. Moreover, we can easily determine a new index I of our tree between these $C(N,K)$ trees.

Now all we have to realize is that these trees are ordered by the preorder codes of their left subtrees and only in case of equality by the preorder codes of the right subtrees. In other words: We know that there are $C(K)$ possible left subtrees and $C(N-1-K)$ possible right subtrees. Given our index I, the left subtree we seek has the index $I \operatorname{div} C(N-1-K)$ and the right subtree has the index $I \bmod C(N-1-K)$. Thus we can compute their codes recursively using the same approach. Note that the right subtree doesn't use labels 0 to $N-K-2$, thus we need to shift the computed preorder code by $K+1$.

The value we return is: (label in the root node) + (preorder code of the left subtree) + (appropriately shifted preorder code of the right subtree).

```
string getCode(int N, int index) {
  if (index >= C[N]) return "";

  int seen = 0;
  for (int rootLabel=0; rootLabel<N; rootLabel++) {
    seen += C[rootLabel] * C[N-1-rootLabel];
    if (seen > index) {
      // enough trees found, rootLabel is correct

      seen -= C[rootLabel] * C[N-1-rootLabel];
      int newIndex = index - seen;

      int leftIndex = newIndex / C[N-1-rootLabel];
      int rightIndex = newIndex % C[N-1-rootLabel];

      string result1 = getCode(rootLabel, leftIndex);
      string almostResult2 = getCode(N-1-rootLabel, rightIndex);
      string result2 = "";
      for (int i=0; i<almostResult2.length(); i++)
        result2 = result2 + ( (char)(rootLabel+1+(int)res2[i]) );

      return (char)(rootLabel + 97) + result1 + result2;
    }
  }
}
```

2.5 Stacking Boxes

When trying to solve this problem greedily, there are many simple and wrong approaches. Putting the box with the largest carrying capacity on the bottom doesn't work. And neither does putting the heaviest box on the bottom. Many of the coders went for a greedy solution... and failed.

But still, the intuition behind these approaches is not that faulty. Suppose that we finished building a valid stack of boxes. Let B be any box in the stack and let A be the box immediately above B. From now on, consider only the stack formed by the box B and all boxes above it. The total weight of this stack is at most $w_B + c_B$. Now suppose that $w_A + c_A > w_B + c_B$. This means that $c_A > (w_B + c_B) - w_A \geq$ (weight of all the boxes) $- w_A$. In other words, the box A is able to carry all the other boxes in our stack. Thus by swapping A and B we obtain a new valid stack with the same number of boxes. (We should note that B is carrying less than before and nothing has changed for the other boxes, thus the stack is valid indeed.)

As a consequence, we may rearrange each valid stack of boxes in such a way that the boxes are ordered according to the sum of their weight and carrying capacity. Thus we only have to search for an optimal stack, where the boxes appear in this order.

This can be done using a simple dynamic programming approach: Process the boxes in increasing order of their sum. For each stack size, keep the smallest weight of a valid sorted stack created from the currently processed boxes.

```
// assumptions:
// W[i] is the weight of the i-th box
// C[i] is its carrying capacity
// for all i, W[i] + C[i] <= W[i+1] + C[i+1]

// in the beginning, the only valid stack is the empty one
for (int i=0; i<=N; i++) best[i]=1000000047;
best[0]=0;

// the largest stack size so far is zero
int maxCount=0;

for (int i=0; i<N; i++) {
  // we now process the i-th box
  for (int j=maxCount+1; j>0; j--) {
    // let's try to make a stack with j boxes
    // with the current one on the bottom
    // for this to work, the currently smallest stack
    // of j-1 boxes mustn't be too heavy
    if (C[i] >= best[j-1]  &&  best[j-1]+W[i] < best[j]) {
        best[j]=best[j-1]+W[i];
        if (j>maxCount) maxCount=j;
    }
  }
}
```