IDI Open Programming Contest April 25th, 2009

Solution sketches

A Letter Cookies
B Bicycle Puzzle
C Geometry Darts
D Box Betting
E Communication Channels (Easy)
F Train Tickets
G Counting Sheep (Easy)
H Rubik's Cube
I Marble Madness
J Robberies
K Robot Encryption

Problem A

Letter Cookies

Problem author: Børge Nordli

This problem can be solved by fairly straightforward counting: For each test case, allocate an array of ints, `Box`$[1 \ldots 26]$, which keep track of the number of each letter in the box. Then, for each candidate word, create another array in exactly the same manner, `Word`$[1 \ldots 26]$, and compare the two arrays value by value:

$$\forall i \in \{1, \ldots, 26\}, \qquad \texttt{Word}[i] \leq \texttt{Box}[i].$$

Tip: The easiest way to convert from `char c` to `int a` is by using the idiom `int a = c - 'A'`.

Problem B

Bicycle puzzle

Problem author: Børge Nordli

First you need to find the optimal strategy of the puzzle: It is obvious that one swap of rectangles should at least put one of them into its correct position. But sometimes the player is lucky and both rectangles will be placed in the correct position. It is quite easy to show that if the player plays this "one-rectangle-into-its-correct-position"-strategy, it does not matter which order the swaps are conducted in:

Treat the puzzle as a permutation of $N = WH$ numbers, and write the permutation in cycle notation, including cycles of length one, for example: $[341526] = (13)(254)(6)$. Then each swap will reduce the length of one cycle by one and create one new cycle with length 1. Example: $[143526] = (1)(254)(3)(6)$, and $[321546] = (13)(2)(54)(6)$. So the optimum number of swaps from the current permutation is always $N - (\#$ of cycles in the permutation). Therefore, the players are totally dependent on the original scrambling of the puzzle, and the problem is reduced to finding the distribution of the number of cycles of a random permutation of N numbers.

This can be found in many ways, but the simplest is by Dynamic Programming:

- A permutation of 1 number always has 1 cycle.
- A permutation of n numbers can be generated uniformly from taking a random permutation of $n-1$ numbers and then inserting the number n in a random position.
- A permutation of n numbers with c cycles is either generated from a permutation of $n-1$ numbers with $c-1$ cycles (if n is inserted in the last and correct position: 1 possibility), or from a permutation of $n-1$ numbers, also with c cycles, where n is inserted in any of the existing cycles ($n-1$ possibilities).

Let $D[n, c]$ be the number of permutation of n numbers with c cycles, then $D[1, 1] = 1$ and $D[n, c] = 1 \cdot D[n-1, c-1] + (n-1)D[n-1, c]$. The probability that the puzzle can be solved in less than S moves is equal to the probability that a random permutation of N numbers has more than $N - S$ cycles:

$$\frac{\#\text{ valid permutations}}{\#\text{ total permutations}} = \frac{1}{N!} \sum_{i>N-S} D[N, i].$$

The limits are chosen such that the numbers fit nicely in a 64 bit integer ($20! < 2^{63}$).

(You also need to implement Greatest Common Divisor (gcd) to output the fraction correctly: $\gcd(a, b) = (b == 0)\ ?\ a : \gcd(b, a \bmod b)$.)

Problem C

Geometry Darts

Problem author: Børge Nordli

This problem is about deciding whether a given point is inside our outside various shapes, and then it is just a matter of counting and adding to find the two player's scores.

There are three cases that have to be solved separately:

1. *Circle* with center (x, y) and radius r: A point (x', y') is inside this circle if and only if its distance to the center is less than r:

$$(x' - x)^2 + (y' - y)^2 < r^2.$$

2. *Rectangle* with corners (x_1, y_1) and (x_2, y_2): A point (x', y') is inside this rectangle if and only if its coordinates lie inside the respective intervals:

$$x_1 < x' < x_2 \text{ AND } y_1 < y' < y_2.$$

3. *Triangle* with corners P_1, P_2 and P_3: This is the trickiest case, but the standard way to solve it is to check whether the point P lies at the *same side* (left or right) of all the directed line segments $\overrightarrow{P_1P_2}$, $\overrightarrow{P_2P_3}$ and $\overrightarrow{P_3P_1}$. This can be done either by looking at the signs of the cross products $\overrightarrow{P_iP_{i+1}} \times \overrightarrow{P_iP}$, or simply by using `java.awt.geom.Line2D(`$P_i, P_{i+1}$`).relativeCCW(`$P$`)`.

It is also possible to check whether the triangle areas match:

$$P_1PP_2 + P_2PP_3 + P_3PP_1 = P_1P_2P_3,$$

but this method is more suspectible to over- and underflows.

Problem D

Box Betting

Problem author: Eirik Reksten

In this problem we need to find the probability that the sum of a substring chosen with the described method lies below, above or inside an interval of two numbers.

In short, sum the probabilities for each possible starting point, and divide by the length of the sequence. Then the only problem that remains is to find the probability for a given starting point in at most $O(\log(N))$.

Given a starting point, the probability for the sum of a subsequence lying in an interval is the amount of substrings in that interval divided by the total amount of substrings with that starting point. This can be found in $O(\log(N))$ using binary search: Start by creating a cumulative list of sums for the lists starting at index 1 and ending at index i. Then the sum in a single substring $[i, j]$ is $\texttt{cumu}[i] - \texttt{cumu}[j-1]$. The binary search then gives us the indices of the end points of the shortest substring with a sum higher than or equal to L, and the first with sum higher than U. Simple math then gives us the probability of each of the cases for this starting point.

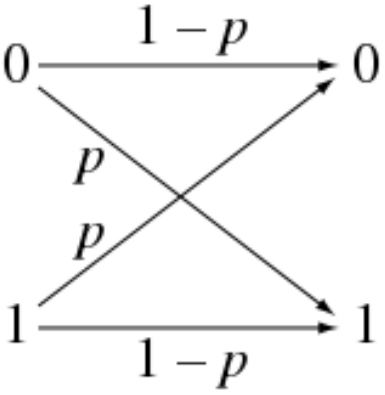
But we can do even better than that. We can use the fact that the boundary (first substrings above the limits) indices will never decrease as the starting point is moved through the list. Therefore, it suffices with a linear search through the list from the previous boundary indices. This gives an amortized running time of $O(1)$ for each starting index, as none of the end pointers will be moved more than N times.

Problem E

Communication Channels

Problem author: Børge Nordli

The problem statement can more or less be ignored. By reading the input and output specification, you can see that you only need to compare the input and output strings of the channel for equality.



A complete solution in Java:

```
public class channel_bn {
  public static void main(String[] args) {
    java.util.Scanner in = new java.util.Scanner(System.in);
    for (int i = in.nextInt(); i --> 0;)
      System.out.println(in.next().equals(in.next()) ? "OK" : "ERROR");
  }
}
```

Problem F

Train Tickets

Problem author: Eirik Reksten

At first sight, one would be tempted to believe that there has to be some dynamic programming solution that works for this problem. In such an approach you'll quickly run into problems when it comes to dividing the problem into independent subproblems.

Thus, we need to develop a different solution, and I'll derive a max-flow solution here. To begin with, we add a node for each group of people (those travelling from station i to station j), as well as one node for each station. Adding a source edge to each group of people will represent the demand for that exact ticket. We now need to enforce that each group actually travel to the station they want to go. Adding an infinity capacity edge from the group to its arrival station, as well as an edge from each station to the sink ensures this. The latter edge has a capacity equal to the total amount of people wanting to go to that station.

At this point, we've make sure that all people are able to go to the station they'd like, but we are still not making any money (they are travelling by other means, so to speak). In order to allow travelling along the train, we add an edge between each station, with the capacity of the train as its capacity (government officials subtracted, of course). In addition, we add another edge from a demand group to its departure station (infinite capacity here as well). Now, travelling by train is possible, but this doesn't give us the configuration that leads to a maximization of profit. This is found by adding the ticket price as cost on the edge between every demand group to its departure station.

Running a max-cost max-flow algorithm (i.e. max-flow with a shortest path algorithm for finding augmenting paths) will find the optimal configuration, while the cost found is the answer to the task.

Problem G

Counting Sheep

Problem author: Eirik Reksten

This problem was marked easy even though it probably is a little harder than Letter Cookies. The reason is that for everyone that knows how a breadth-first search works, this is a very standard problem. For everyone that doesn't, now would be the best time to start learning. BFS is one of the most fundamental algorithms (not to mention its usefulness), and can be found as subroutines to a lot of other algorithms as well (i.e. max-flow).

There are several approaches to this problem. The most obvious ones are probably to iterate through the matrix, initiating a breadth-first searc (BFS) or non-recursive depth-first search (DFS) every time you find an unmarked sheep. Mark all sheep found by the search and increase the flock count by one.

You could also do this with a disjoint set datastructure. In the beginning, initiate the structure with one set for each different sheep. Then iterate through the matrix. Whenever you encounter two "neighbouring" sheep, merge the sets containing these. The solution is the amount of sets in the structure when this is done.

Problem H

Rubiks Cube

Problem author: Børge Nordli

Counting rotated versions of the same cube multiple times, the $2 \times 2 \times 2$ Rubik's cube has $8!3^7 = 88179840$ states[1]. This is too much for an exhaustive search, but by fixing the position and rotation of one corner, thus counting rotated versions of the same cube only once, there are only $7!3^6 = 3674160$ states. This will still take some time for a normal Breadth First Search (BFS), but by applying the standard trick of double-sided (or meet-in-the-middle) BFS, a back-of-the-envelope calculation[2] suggests that an order of about 50000 states would be examined.

After deciding the algorithm, all what's left is the implementation. But this is definitively not trivial. The trickiest part is implementing the moves. Clearly a state can be represented as 24 integers (even half-bytes will suffice), and a move is just a permutation of these numbers. It is advised to check which faces goes to which ones with paper and pencil before inserting numbers in the code. Just remember that one corner is keeped fixed, so there are three separate moves (turning the "other" half of the cube) that can be performed in both directions. One move becomes a permutation consisting of 3 cycles of 4 numbers each, and the 12 other numbers are kept untouched.

Then of course you need to implement `equals` and `hashCode` (or `compareTo`) to be able to recognize states we have already seen, using a `HashSet` (or a `TreeSet`), but we can use standard implementations of by using standard methods on the underlying array, or even on the `String` representation given in the problem.

The last hurdle is deciding how the solved cube should look like, from where you should start the BFS in the other direction. (The cubes in the input are not all the same!) The $2 \times 2 \times 2$ cube is different from the $3 \times 3 \times 3$ cube in such a way that its faces have no centers, so it is a little harder to determine how the solved cube would look like, given a scrambled cube. But by looking at all corners, it is possible to find out which colors are adjacent. Then you can either just create a different solved version for each test case, or canonicalize the colors of the scrambled cube[3].

[1] There are 8 corners that can be freely placed, but after placing and rotating the first 7 corners, the last corner must be placed in a correct position in order to be able to solve the cube.

[2] Assuming one move on average reaches about 4 new states, the maximum number of moves would be around $\log_4 7!3^6 \simeq 11$, adding a couple of moves for outliers, the reasonable bound of the numbers of states that must be examined is about $2 \cdot 4^{14/2} \simeq 32000$.

[3] And with this approach you can even reuse the solved half of the BFS between test cases.

Problem I

Marble Madness

Problem author: Truls A. Bjørklund

Note that for each move, the total number of marbles in the bag is reduced by one, and the number of white marbles either stays constant or is reduced by two. So, if the bag originally contains an odd number of white marbles, the last marble will always be white, and if the bag originally contains an even number of white marbles, the last marble will always be black.

A complete solution in C/C++:

```
#include <stdio.h>
int main() {
  int n, w; scanf("%d", &n); while (n--) {
    scanf("%d %d", &w, &w);
    printf("%d %d\n", (w+1)%2, w%2);
  }
}
```

Problem J

Robberies

Problem author: Nils Grimsmo

After you realize that this can be solved by Dynamic Programming on the banks and outcome (in millions), this problem is quite easy to code:

Let $D[b, m]$ be the (best) probability Roy the Robber can achieve of escaping with exactly m millions after considering banks 1 through b. Then:

- Roy will never get caught if he doesn't rob any banks: $D[b, 0] = 1.0$, for all $1 \leq b \leq N$.
- Let bank i contain M_i millions: Roy could chose either to rob this bank or not: If he doesn't rob it, the probability of getting away with exactly j millions is equal to $D[i-1, j]$. If he chooses to rob it, the probability is $(1 - P_i) \cdot D[i-1, j - M_i]$. Roy would of course want to maximize the probability of not getting caught, so

  $$D[i, j] = \max(D[i-1, j], (1 - P_i) \cdot D[i-1, j - M_i]).$$

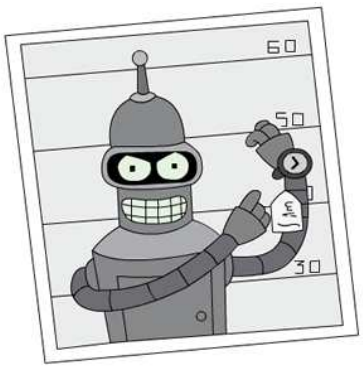
  Of course $j - M_i$ can be less than 0, so make sure to check for this.
- In the end, Roy wants to get away with as much as possible, so the answer is the largest m such that this inequality is true: $D[N, m] \geq 1 - P$.

(You can also do DP to *minimize* the probability that he will get caught, or letting $D[b, m]$ be the possibility that he will escape with *at least* m millions.)

Problem K

Robotic Encryption

Problem author: Rune Fevang



The first thing to realise is that trying to simulate the entire behavior of the robot will consume too much time. In the worst case (by nesting "FF" 16 levels deep), the robot requires $2 \cdot 9^{16} = 3,706,040,377,703,682$ steps. This is obviously too much.

The key to solving this problem lies in realising that there is a limited number of possible states the robot can be in. There are a $W \cdot H$ positions, and 4 directions, giving us $S = 4 \cdot W \cdot H$ possible states.

Now, each instruction can be seen as just a permutation of those states. A permutation representation of an instruction would be an array $\texttt{P}[1 \ldots S]$, where $\texttt{P}[i]$ would contain the state index of where state i would end up after executing the instruction.

Using this representation we can build more complex instructions from simpler ones. Two consecutive instructions `A` and `B` can be combined into `C` by executing $\forall i \in [1 \ldots S], \texttt{C}[i] = \texttt{B}[\texttt{A}[i]]$. Loops can be implemented simply by combining the instruction inside it with itself the number of times the loop indicates.

Homework: How would you solve this problem if there was no bound on the number of times a loop could be repeated?