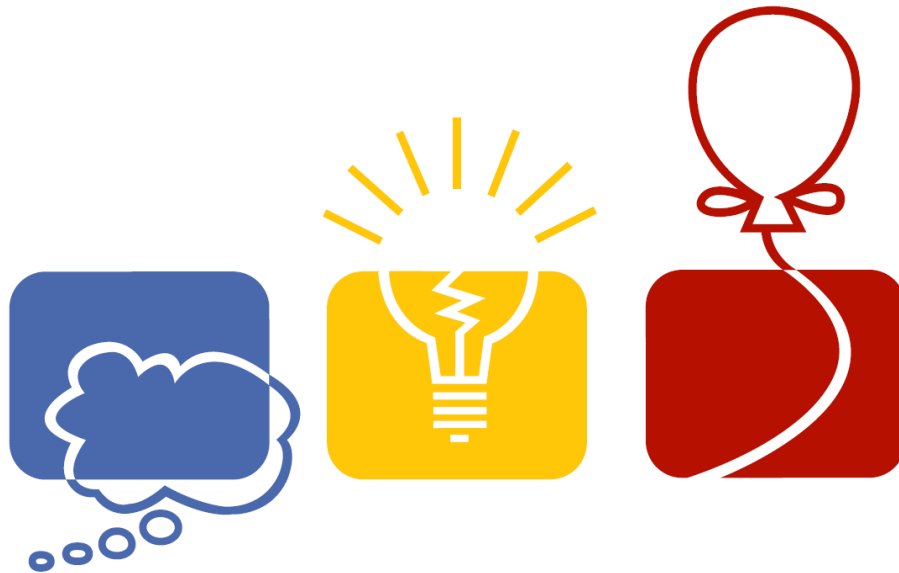




Egyptian National Contest 2009

Problem Set Analysis

Written by: Khaled Hafez (khaled912@gmail.com)
Reviewed by: Hamza Darwish (cpphamza@yahoo.com)



This document contains detailed explanation for the solutions of the problem set of the Egyptian National Contest 2009. You are advised to have fun trying to solve the problems before reading this document.

A – SPEED

What would a contestant love more than a problem with no description? The only thing that should be avoided is “a[x%2]++;” and “if (x%2 == 1)” as the input values could be negative.

B – BAGS

The required expected value can be calculated as $\sum_{i=n}^m i * prob(i)$. Nothing new so far, this is a general way of calculating an expected value. So what is prob(i)? It is the probability of putting n bags in m slots such that all the locations that the bags occupy are $\leq i$ with one condition: one of the placed bags must be exactly at location i. If you did not fully understand the previous statement read it again. If it is still confusing read it once more. If you're still confused, bring a paper and a pen and draw slowly what it is describing until you fully understand it; this is the most important part you need to understand in order to solve the problem. prob(i) can be evaluated using DP. The first thing we need to stress is that the state needs a Boolean variable to indicate whether or not a bag has been placed at location i. For the first while, the other DP parameters needed seem to be: i, n and m. Those 3 parameters are too much given the current limits. We then notice that if we keep i, n and m global (not as DP parameters) all we need now is the number of bags placed so far (call it b). The current i, remaining n and remaining m will then be the initial (global) i, n and m each minus b, respectively. i.e. $remaining_m = initial_m - b$, $remaining_n = initial_n - b$ and $current_i = initial_i - b$. In each state, we put a single bag. We consider placing the bag in the first remaining_n out of the remaining_m positions and update the probability, and then we consider it being placed exactly at the current_i position if no bag was placed there already and, again, update the probability. The overall complexity is $O(n*m)$.

C – PERFUME

Once analyzed properly, this problem is not at all difficult. Let's start by taking two mixtures and see what kind of other mixtures we can produce out of them. Let's say mixture A has ratios x_1 and y_1 , while mixture B has ratios x_2 and y_2 . Since there are no quantities involved and we're concerned only about the ratios, we can totally ignore the volume of the new mixture. If the new mixture is fully made from mixture A, the ratios of the new mixture will be x_1 and y_1 . If it is fully made from mixture B, the ratios of the new mixture will be x_2 and y_2 . If half of the new mixture is from A and the other half is from B, the new mixture ratios will be $(x_1 + x_2)/2$ and $(y_1 + y_2)/2$. In general, if T ($0 \leq T \leq 1$) of the new mixture is from mixture A, then $(1-T)$ of the new mixture is from mixture B. In this case, the ratios of the new mixture will be $(x_1 + x_2) \cdot T$ and $(y_1 + y_2) \cdot (1 - T)$. Wait a minute, doesn't this equation look familiar? Yes! It is the equation of a 2D line segment! So now we can look at the problem differently: the given mixtures are actually Cartesian points. So let's see what we can obtain using three different mixtures. Thinking about points, it is not difficult to see that we can produce any point that lies inside the triangle formed of those three points. So for 4 points, can we produce any point that lies inside the quadrilateral formed of those 4 points? Not always: think about a case where the given 4 points do not form a valid quadrilateral (draw a triangle of 3 points and put a 4th point inside it). For such a case we need to consider only the "outer" points. In general, for N points, we actually need to consider their convex hull! So that's what the problem is asking for: given a set of points S in 2D space, determine whether or not each point in another set of query points lies inside the convex hull of S . Computing the convex hull and testing if a point lies within a convex polygon can be easily found in many online computational geometry tutorials. Don't forget to handle special cases such as having a case with collinear points or with 2 or fewer points in the input. The overall complexity is $O(N \log N + Q \cdot N)$.

D – IN LOVE WITH TV

The first step to solve this problem is to reduce the lengthy description into the following: “Given a sequence S , find the length of the shortest subsequence of S that contains all the elements in S ”. From the limit, we’re looking for a linear or a sub-linear solution. However, let us start by building an easy $O(n^2)$ solution and see how to optimize it. The solution assumes that we know that the optimal subsequence ends at location Y . From there, it is easy to keep running backwards till we hit all the elements in the sequence. How do we know Y ? Well, we don’t; we just try all possible values of Y , excluding those that won’t lead to a solution (if there is not enough elements to cover S in all locations $\leq Y$). A straight forward implementation would be $O(n^2)$. However, there’s a trick to reduce it to $O(n \log n)$. Here’s the trick: let’s inspect all values of Y in increasing order. When finding the starting location X of the optimal subsequence ending at Y , we use the starting location Z of optimal subsequence ending at $Y - 1$. It is easy to calculate Z by initializing it with X and advancing it until all unnecessary elements are removed (the count of $S[Z]$ is exactly 1 in the subsequence starting at Z and ending at Y). Pseudo-code follows:

```
x = y = total = 0
result = infinity
// Calculate the minimum value of y.
while total < number of distinct elements in S
    if count[S[y]] == 0 then total++
    count[S[y]]++
end while
// Iterate through all solutions ending at y for all valid y < n.
while y < n
    while count[S[x]] > 1
        count[S[x]]--
        x++
    end while
    result = min(y - x + 1, result)
    y++
    if y < n then count[S[y]]++
end for
```

This looks like $O(n)$, but since keeping count of arbitrary values is assumed to be done using a multiset, the overall complexity will be $O(n \log n)$.

E – BALL

There are two conditions that would make a person lose. The first condition is that the person has no friends and the shortest path from person S (where the ball starts) to this person is $< N$. The second condition that would make a person X lose is that if there is a path from person S to person X with length exactly N. Let's now focus on the second condition, and then the first one will immediately follow. For small values of N, this is a straightforward DP problem. For such a big value, DP is out of the question. Let's recall the first lecture in "discrete mathematics 101": if we have an adjacency matrix M of a directed graph where $M[i][j]$ is true if there is a directed edge between i and j, then $(M^K)[i][j]$ is true only if there is a path of length exactly K between i and j. M^K is the matrix M raised to the K. Keep in mind that this is Boolean arithmetic though: $1 + 1 = 1$. What remains is to calculate M^N and then we will have the answer. Still, N is too big, how can we do that? The answer is: using the same small, neat divide and conquer power function that we discussed in problem I (see the solution for Problem I for details). So how about the first condition? Here's a neat trick (thanks to Hamza): if a person X has no friends, set $M[X][X]$ to 1. This means that the presence of a path with length exactly B between S and X will imply the presence of a path of length B + 1 between S and X. In other words, once the ball reaches person X (after C throws, where C is the shortest path between S and X), it will stay there forever. The final check would be: A person X can lose only if $(M^N)[S][X] == 1$. Overall complexity is $O(M^3 \log(N))$.

F – HYPER

This is a simple scheduling problem. We need to consider two types of events: a customer starts to checkout and a customer leaves the shop (ends checkout). For each customer arriving at time X and staying for time T at the checkout we generate two events: "Customer starts at time X" and "Customer leaves at time X + T". After generating all the events, we sort them by time to get a sequence of events such as this: ("start", "start", "leave", "start", "leave", "leave")

Notice that now the actual timestamps of the events are irrelevant; only their order is what matters. All what we need to do afterwards is to compute maximum number of customers that has started but has not yet finished at any point in time. This is easily done in linear time using this pseudo code:

```
result = 0
count = 0
for each event e in sorted_events
    if e == "start" count++
    else if e == "leave" count--
    result = max(result, count)
end for
return result
```

The only thing that you need to consider is that if two events happen at the same time, the one with "leave" should come first – this is demonstrated in the first case of the sample input.

G – SECURITY CLEARANCE

There are countless wrong approaches to solve this problem which would seem correct for the first while. The main challenge was to think about the various cases that would make such solutions fail. In general, if your solution needs this line:

```
if (board[row][col] == 'A') return false;
```

Then it is most probably wrong. The hint in the problem statement was “Doors with security level ‘A’ are, **by definition**, useless”. A general solution that implements the ‘definition’ of a useless door should be able to refuse any door with level ‘A’ without having to do to an explicit check. Here are few cases that made many solutions fail:

5 5	6 3	7 6
xxxxxx	xxxxxxx	xxxxxxx
*..Bx	*.BC.x	x...B.x
x..Bx	xxxxxx	x.xxx.x
x..Bx		*...Bx
xxxxx	Expected output:	x.xxx.x
	“Faulty!”	xxxxxxx
Expected output:		Expected output:
“Faulty!”		“Ok!”

All what you need to know in order to deduce the correct output of those cases can be directly or indirectly inferred from the problem statement.

The most elegant way to solve the problem is to look from a different angle: instead of thinking about how to implement a “bool IsUseless()” function, think about how to implement a “bool IsUsefull()” function. This will make a lot of difference. The idea is as follows:

- For each cell, calculate the minimum security clearance level (MSCL) a person needs to reach that cell. This is easily done through a single BFS. We start at the entrance with security level ‘A’. We move to adjacent cells until we reach a door, in which case we increase the exploring MSLC to that door’s level. We always pop from the queue the cell with the lowest MSCL. Since we only have a total of 26 security levels, we can optionally use the concept of “buckets” to avoid multiplying the complexity by $O(\log n)$.
- Unreachable cells are assigned any sentinel MSCL value (INF or so). Implementation wise, the MSCL array is initialized with this value before starting the BFS.
- The elegant check now easily follows: A door with security level L is useful only if it has two neighboring cells with MSCL M1 and M2 such that $M1 < L$ and $M2 == L$.

H – DARTS

The limits do not allow inspecting all pairs of darts and adding the distance between them to the total sum. The key point is to notice that the distance required is not the Euclidian distance but rather the Manhattan distance. What we need to calculate is:

$$\sum_{i=0}^n \sum_{j=i+1}^n (|x_i - x_j| + |y_i - y_j|)$$

Which can be easily rewritten as:

$$\sum_{i=0}^n \sum_{j=i+1}^n (|x_i - x_j|) + \sum_{i=0}^n \sum_{j=i+1}^n (|y_i - y_j|)$$

So now, we can solve for X and for Y independently using the same method. If we concentrate on the X for now, one thing to notice is that each co-ordinate x_i will be added to the total sum A_i times and will be subtracted from the total sum B_i times. It is not difficult to see that A_i is the number of x_j such that $x_i > x_j$ while B_i is the number of x_j such that $x_i < x_j$. Those two numbers can be easily computed through sorting the array of X co-ordinates and inspecting the position of each co-ordinate. Pseudo code is as follows:

```
// Calculate the contributions of X
sum = 0
sort x_locations[]
for each x in x_locations
    sum += (number of elements before x) * x
    sum -= (number of elements after x) * x
end for
// Do the exact same for Y
sort y_locations[]
for each y in y_locations
    sum += (number of elements before y) * y
    sum -= (number of elements after y) * y
end for
return sum
```

Remember to use 64-bit data types to avoid overflows.

I – PASCAL

The triangle consists of three sides; two of them consist only of 1s. The bottom side was the only non-straight forward part in the problem. It is a well known fact (and it is also mentioned in the problem statement) that row n and column k of the Pascal's triangle corresponds to C_k^n . From this, we can deduce that the value of the last row is actually $\sum_{i=0}^n C_i^n$. So let's see the details of this calculation taking 3 as an example. This corresponds to “the number of ways to choose 0 elements out of 3” + “the number of ways to choose 1 element out of 3” + “the number of ways to choose 2 elements out of 3” + “the number of ways to choose 3 elements out of 3”. More than that, let's even enumerate the subsets that we are counting.

The number of ways to choose 0 elements out of 3 (1 way):

000

The number of ways to choose 1 element out of 3 (3 ways):

100

010

001

The number of ways to choose 2 element out of 3 (3 ways):

110

101

011

The number of ways to choose 3 element out of 3 (1 way):

111

Total = 8.

Did you see what has just happened? We have just enumerated all the possible subsets of a set of size 3. Yes, this summation $\sum_{i=0}^n C_i^n$ corresponds to 2^n . Now a small, neat divide and conquer power function will be able to calculate 2^n in $O(\log n)$:

```
int power(int base, int pwr, int m) {
    if (pwr == 0) return 1;
    int result = power(base, pwr/2, m);
    result = (result * result) % m;
    if (pwr % 2 != 0) result = (result * base) % m;
    return result;
}
```

Please note: In practice you will need to use 64-bit data types to avoid overflowing when doing multiplications. Also, you will most probably need to handle the case of $n = 1$ separately (the answer should be 1, not 0).