ICPC 2015 Regional Contest
Jakarta Site

Problem Analysis

A. Arithmetical CAPTCHA
B. Udin and Ucok
C. Counting Partition
D. An ICPC Problem without Statement
E. Awesome Cipher Machine
F. Problem on Group Trip
G. Dungeon Trap
H. Harvest Season
I. National Disaster
J. Alien Abduction 3
K. Amplified Energy
L. Summation and Divisor

A. Arithmetical CAPTCHA

This problem supposed to be the easiest problem in this contest. The number of operands are fixed (i.e. 4) and there are only 3 available operators to be chosen (i.e. +, −, =). You can solve this problem by using backtracking (testing each operator one by one), or you can simply list all possible combinations in IF clauses. For example,

```
if ( n1 == n2 && n2 == n3 && n3 == n4 ) { /* n1 = n2 = n3 = n4 */ }
if ( n1 == n2 && n2 == n3 + n4 )        { /* n1 = n2 = n3 + n4 */ }
if ( n1 == n2 && n2 = n3 - n4 )         { /* n1 = n2 = n3 - n4 */ }
if ( n1 == n2 + n3 + n4 )               { /* n1 = n2 + n3 + n4 */ }
...
```

In total there are $3^3 = 27$ combinations, but since there should be at least one equality operator, the total number of combinations is only 2 * 3 * 3 = 18.

B. Udin and Ucok

This is a typical problem on combinatorial game theory which can be solved by Sprague-Grundy theorem, i.e. any impartial game (like this problem) is equivalent to a nim heap of a certain size (also known as nim value). The nim value of a position is defined recursively as the smallest ordinal which does not exist among its neighbours' nim value. In this problem, we only need to find the nim value for each position (the number of stones).

For example, let nim(N) be the nim value when the number of stones is equal to N.

nim(0) = 0
nim(1) = 1, as when N = 1, we can make a move to N = 0.
nim(2) = 1, as when N = 2, we can make a move to N = 0.
nim(3) = 2, as when N = 3, we can make a move to N = 0 and N = 1.
nim(4) = 0, as when N = 4, we can make a move to N = 1, and N = 2.

Notice the neighbours of N = 4 are N = 1 and N = 2 which nim values are nim(1) = 1 and nim(2) = 1. The smallest ordinal which does not exist among its neighbours is 0. Any position with nim value of 0 is a losing position. When nim(K) = 0, you can only make moves to any other position with nim(K') ≠ 0, whil when nim(K) ≠ 0, you can always make a move to a position where nim(K') = 0, i.e. a forcing move.

However, computing the nim value for a position N requires an O(N) time, assuming the nim value for all other position 1..N-1 have already been computed. It's not enough to solve this problem.

One way to solve this problem is by generating the output for all small N (e.g., N = 0..1000) and observe patterns in it. In this problem, the output will converge to a certain pattern after a while. Once you got the pattern, it's an O(1) solution.

C. Counting Partition

A naive approach for each query is to create a new partition whenever we can (greedily) from the left to the right. Following pseudocode will give you a correct answer.

```
Query(X, Y) :
  countX   = 0
  countAll = 0
  ANS      = 1

  for i in range(N) :
    countAll++
    if ( A[i] == X ):
      countX++
      if ( countAll >= B[Y] and countX >= Y ):
        countX   = 0
        countAll = 0
        ANS++

  if ( countAll == 0 ):
    ANS—

  return ANS
```

However, this code is too slow because it runs in Θ(N) for each query. We need to optimize this. Suppose the current partition starts at index p. This partition will end at q, where q is the smallest index which satisfies (1) $q \geq p + B[Y]$, (2) X appears at least Y times between p and q, and (3) A[q] = X. This can be computed quickly using binary search if we have the list of all X's positions. The algorithm now runs in in O(M / Y log M) for each query where M is the number of X occurrences in A.

Next, we need the second trick: memoization. If the query has been computed before (for the same X and Y), just return the result immediately. Assuming the given query only has a constant value of X, then the total time complexity for all the queries is: O(M / 1 log M) + O(M / 2 log M) + O(M / 3 log M) + ... + O(M / M log M) = $O(M \log^2 M)$. Notice that M/1 + M/2 + M/3 + ... + M/M is the M-th harmonic number, which equal to O(M log M).

Now, let's consider the situation where X is not constant in all given query (which is the problem). Let $X_1, X_2, X_3, ..., X_K$ are all possible values of X in the queries, and $M_1, M_2, M_3, ..., M_K$ are their number of occurrences in A, respectively. The total time complexity to process all possible queries then is equal to $O(M_1 \log^2 M_1) + O(M_2 \log^2 M_2) + O(M_3 \log^2 M_3) + ... + O(M_K \log^2 M_K)$. Since we know that $M_1 + M_2 + M_3 + ... + M_K \leq N$, then the total time complexity to process all queries would be $O(N \log^2 N)$.

D. An ICPC Problem without Statement

It's a very lovely day in the year of 2016. Having read and tried the problem "An ICPC Problem without Statement", you began searching for the solution on the internet. You found the blog of the chief of judge of the contest. Then, you read a blog post containing the solution …

High-level description
First, split the input into two arrays: array of **absolute** values of negative integers (let's call it X) and array of nonnegative integers (let's call it Y). There are two cases to consider. In each case, we greedily brute force how many negative integers to pick and how many nonnegative integers to pick.

1. The solution consists of an **odd** number of negative integers.

   In this case, the product will be non-positive. Therefore, we need to minimize the absolute value of the product. First, we brute force the number of negative integers we want to pick. We always pick the smallest negative integers first. Then, we pick some smallest nonnegative integers only if necessary (if the number of picked negative integers is less than A).

   pseudocode:

```
let nX be the number of integers in X
let nY be the number of integers in Y

res = -INF
for ( cntX = 1; cntX <= nX && cntX <= A; cntX += 2 ):
    cntY = min(nY, A - cntX)
    if (cntX + cntY < A) {
        continue;
    }

    // pick the smallest cntX negative integers
    // and the smallestcntY nonnegative integers
    // compare and update res with this current solution
}
```

2. The solution consists of an **even** number of negative integers.
   In this case, the product will be nonnegative. Therefore, we need to maximize the value of the product. First, we brute force the number of negative integers we want to pick. We always pick the largest negative integers first. Then, we pick as many as largest nonnegative integers (up to B). We must be careful in handling zeroes. We should only pick zeroes only if we must do it (i.e., to make the number of picked integers not less than A).

pseudocode:

```
let nX be the number of integers in X
let nY be the number of integers in Y
let nYPositiveOnly be number of positive integers in Y

res = -INF
for ( cntX = 0; cntX <= nX && cntX <= B; cntX += 2):

      // do not pick zeroes
      cntY = min(nYPositiveOnly, B - cntX)

      // only pick zeroes if must
      if (cntX + cntY < A) {
      cntY = A - cntX
      }

      if (cntX + cntY < A) {
      continue;
      }

      // pick the largest cntX negative integers
      // and the largest cntY nonnegative integers
      // compare and update res with this current solution
}
```

Pick the better of two solutions.

Implementation details

Picking the smallest/largest integers in X/Y can be done by pre-sorting the arrays. The next challenge is how to represent the product of candidate solutions. Note that the product can only in the form of 0, or 2^k, for some integer k. Then, we can actually represent the product as follows.

- 0 denotes 0.
- P (P > 0) denotes $2^{P - 1}$
- P (P < 0) denotes $-2^{(-P) - 1}$.

(Note that there is an offset of 1 since we have to represent the number 0.) Using this representation, the computation of product of candidate solutions then can be done using simple prefix/suffix sums. Extra care must be taken for handling 0 since we have offsets.

E. Awesome Cipher Machine

First of all, this problem has nothing to do with any cryptographic stuff mentioned in the problem statement. This problem is basically an inverted coin change problem. Given the output of a coin change problem, predict its input. There are many solutions to this problem. Here we show you one we used as reference solution.

If there are N ones, there will be ${}_NC_K$ ways to construct K (for $K \le N$); basic counting. Let's introduce a new shifting number S, where $S > N$. If there are a single S and N ones, then there are ${}_NC_K$ ways to construct S + K. Now let's look at this from another perspective; given a number M where $M - S \le N$, there will be ${}_NC_{M-S}$ ways to construct M. What happened if we have multiple S? Let's say we have S_1, S_2, …, S_x, and N ones, then there will be ${}_NC_{K_1} + {}_NC_{K_2} + \ldots + {}_NC_{K_x}$ ways to construct M, where $K_i = M - S_i$. Of course, all S must be strictly larger than M / 2 (to prevent two or more S making M).

Back to the problem! We want to make a set of coins which can construct N with K different ways. Using aforementioned trick, we can achieve K by using the sum of combination numbers, i.e. we want $K = w_i * {}_LC_i$ for all i = 0..L, where w is a weight vector and L is an integer. Our task is to find a proper w and L which satisfy the equation. The final solution will be L ones and w_i times of N - i, thus there are $L + w_0 + w_1 + \ldots + w_L$ coins used. For example, let N = 10 and K = 5. If we choose L = 3, then w will be [0, 2, 0, 1]. The coins are 1, 1, 1, 7, 7, and 9.

How to find such L and w? The highest K is 20000, while the largest value in ${}_NC_K$ is always ${}_NC_{N/2}$. Thus, we have to find the largest L such that ${}_LC_{L/2} <= 20000$. We used L = 15 in our solution. Finally, we can find w using dynamic programming in coin change problem. Alternatively, we're being generous so that greedy approach on multiple L also works for this problem.

F. Problem on Group Trip

The problem statement is long, but it's only a simulation problem. You have to implement the solution carefully, especially when handling the 0 (skipping the station). One easy way is by processing each station one by one in order, ignoring all other "future" stations. When processing a station, we need to find out what the ending time (the time when he completed the station) for each person is. This ending time is then used to process the next station. Of course, there are other approaches as well, just find one which suits you well.

G. Dungeon Trap

Player can transform empty cells into obstacles as long as there is at least one path from A to B; otherwise the game ends. Suppose the player pick a path P from A to B. The player can transform all empty cells which are not in the P. This ensures that there is only one remaining path from A to B, i.e. the path P. Then the player transforms one of the cells in P to cut off the path and end the game.

Since we're asked to find the maximum number of tokens that can be spent by the player, selecting a random path may not coincide with what we're looking for. We have to search all possible paths and find one which minimize the following score: score(P) = T - (tokens(P) - max_token(P)), where T is the sum of all tokens in the entire map, tokens(P) is the sum of tokens in P, max_token(P) is the largest token in P.

Equivalently, to find P which maximize score(P), we want to find a path which minimize the value of tokens(P) - max_token(P). Such path can be found efficiently using Dijkstra's algorithm with a slight modification.

H. Harvest Season

First, notice that the Y component is independent problem. No matter how you move the machines, it will not affect the cost for all Y components. Thus, we can count it separately and focusing ourselves to the X components only.

Given a set of apples' position (only x-coordinate) and one machine, where should we put this machine to minimize the total cost (assuming cost of moving a machine is 0)? The answer is the median of all apples' position. Let's introduce cost to move the machine which is equal the cost to pick the apple (A = B). In this situation, we can simply treat the machine as another apple, and find the median from all the data. Now, what if A ≠ B? Simply make a new set which contains A number of machines in the same position, duplicate each apple into B times, and find the median of these data.

Now, in the problem, we're given N machines instead of 1. To solve this, we can use dynamic programming approach. First we should sort all the apples and machines. Let D[a, b] be the minimum cost to pick all apples from index 1 to b with machine from index 1 to a, thus the next machine we should determine the location is machine with index a. To compute D[a, b], we need to evaluate for all range of apples from index [1, b] to index b, i.e. how many apples should be picked by machine a. For each range, the best position would be the median as has been mentioned before. A naive implementation of the median finding would give a total time complexity of $O(NM^3)$, which will cause the solution to get TLE. The median can be found in O(1) amortized by maintaining a partial sum array (needless to say, the apple range should be processed in order). However, you should be careful when implementing the solution (it's easy to introduce bugs here), thus, making the problem more challenging. The total time complexity is $O(NM^2)$.

I. National Disaster

With N ≤ 200, we only need to test each distance and find which one has the least number of lies. There are $O(N^2)$ distance which we should test. When testing the distance, if we do it naively by iterating the matrix in $O(N^2)$, then we'll get an $O(N^4)$ time complexity, which will give you a TLE result. We can do the testing in O(1).

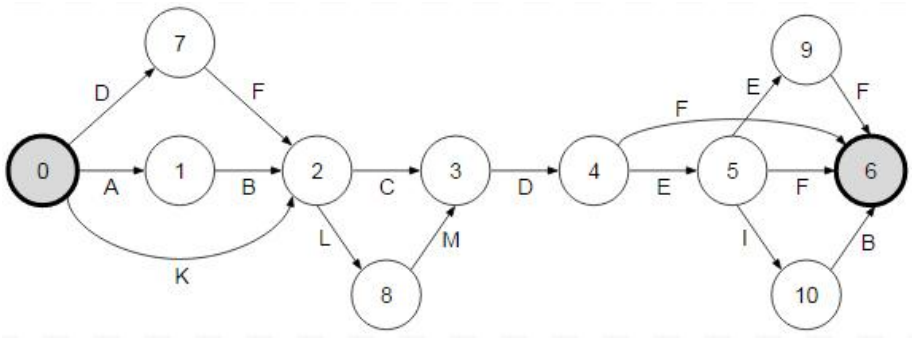
First, we have to linearize the matrix -- make it into 1D array-- and sort them by the distance in increasing order. Now, when you want to compute the number of lies for a certain distance (certain index in the 1D array), you need to find how many 0 to the left (inclusive the index) and 1 to the right (exclusive to the index) of the index. Partial sum do the trick. Compute it once, and for each index query, you can answer it in O(1). However, you need to beware with non-unique distances. In this case, you should consider largest index with the same distance.

J. Alien Abduction 3

Looking at the maximum string length of 20, it is tempting to do brute-force. Experienced contestants would know that this problem has an exponential search space and would need efficient implementation. Now you are faced with a dilemma: should you go ahead write a brute-force solution (perhaps with heavy pruning tricks)? or should you spend more time to think of a better solution (or just skip this problem)?

If you chose to do brute-force with heavy pruning, you're going to have a bad time. While it is not obvious, there exists a simple test case that can make many brute-force or greedy strategies (that do not explore all search space) to get either wrong answer or time limit exceeded (see the ultimate test case in the appendix). The branching factor for each substring of length at most 4 is 4; therefore, with an input string of length 20, there are about 4^{20} different target strings.

The problem can be solved using standard dynamic programming. The hard part is minimizing number of states needed. Consider the input query for the first sample case ("`ABCDEF`") and all available substitutions applicable to that input. You can form a directed graph as follow:



The graph represents all possible paths from the start node #0 to the final node #6. If there is no substitution, then traversing the graph from #0, #1, #2, #3, #4, #5, #6 produces the original input string "`ABCDEF`". If we were to replace "`AB`" to "`DF`" then we would've taken the path #0, **#7**, #2, #3, #4, #5, #6. Notice that this process satisfies all the rules to mutate a molecule sequence.

Since the maximum length of any string is 20, there are at most 210 substrings. Since each substring can be replaced by another string (with length at most 20), the number of nodes in the graph is at most 210 * 19 = 3990 nodes.
We create a separate graph for the second string in the input query in the same way, thus we now have two DAGs (directed acyclic graphs). From here, we can use a standard dynamic programming:

Let dp[i][j] be the shortest string that can reach this state (tie break by picking the lexicographically smaller string), where i is the node position in the first graph and j is the node position in the second graph.

The starting state is dp[0][0] and our answer is at dp[$|A|$][$|B|$]. To transition from one state to another, both i and j must each find an edge with the same character and move together to the new state.

The two DAGs can be built in linear time in about D * 5 * 26 * 26 operations. The transition from one state costs only 26 operations. Thus, the total number of operations to answer a query is at most 3,990 * 3,990 * 26 = 413,922,600. Note that in most cases, we don't hit the max number of states.

Below is the ultimate test case that should kill most greedy / heuristics / backtracking with pruning strategy. Some of the judges have squeezed any idea they can think of to "cheat" the problem (not using the proper solution) and failed. So, if any of you managed to get correct output with the prepared test case in a few second, we will gladly accept your solution.

```
1
17
5 A AB ABC ABCD NW
5 B BC BCD BCDE LR
5 C CD CDE CDEF BB
5 D DE DEF DEFG MQ
5 E EF EFG EFGH BH
5 F FG FGH FGHI AR
5 G GH GHI GHIJ ZO
5 H HI HIJ HIJK WK
5 I IJ IJK IJKL KY
5 J JK JKL JKLM DD
5 K KL KLM KLMN QS
5 L LM LMN LMNO XR
5 M MN MNO MNOP JM
5 N NO NOP NOPQ OW
5 O OP OPQ OPQR FR
5 P PQ PQR PQRS XS
5 Q QR QRS QRST JY
100
ABCDEFGHIJKLMNOPQRST ABCDEFGHIJKLMNOPQRST
ABCDEFGHIJKLMNOPQRST BCDEFGHIJKLMNOPQRST
BCDEFGHIJKLMNOPQRST ABCDEFGHIJKLMNOPQRST
...
```

Now, the brilliant scientist is trying to find a cure for the "alien" babies, but that's a story for another time. Always be prepared for the next call from the brilliant scientist!

K. Amplified Energy

To avoid overflow and working with large number in this problem, we should work with sum of log instead of the actual product of number. Notice that $\text{argmax}_{i,j}(N_i * ... * N_j)$ is equal to $\text{argmax}_{i,j}(\log N_i + \cdots + \log N_j)$. Therefore, we can work our solution in the log space (by calculating the log value for all elements) and find the continuous subsequence which has the highest sum value.

We can use dynamic programming approach to find such subsequence. If there is no replacement cube, then Kadane's algorithm (finding maximum sum in sub-array) solves this problem perfectly. As

there might be replacement cubes in our problem, we need to modify the algorithm a little bit. Let S be the array of replacement cubes sorted based on its energy power in descending order. We need to sort this as we want to pick the replacement cube greedily, i.e. if we decide to use a replacement cube, we will use the one with highest power first. Let D[a, k] be the maximum sum of consecutive cubes from 1 to a after using k replacement cubes. Then, D[a, k] = max{ 0, D[a-1, k] + log(N_i), D[a-1, k-1] + log(S_k) }, i.e. maximizing the case where we don't use replacement cube and the case where we use one. While computing the DP table for D, we keep track the global maximum result.

L. Summation and Divisor

This problem looks hard as the number of combinations which you should check is enormous, O(M^N). However, if one do randomization (randomly pick the elements), one can get lucky. Nevertheless, there is an exact solution for this problem which not involving any randomization. Here's one.

Assume N = 2, A_0[..] = P[..], and A_1[..] = Q[..]. This analysis applies for any larger N, but it's easier to explain with smaller N. B in the problem statement will be {$P_0 + Q_0$, $P_0 + Q_1$, ..., $P_1 + Q_0$, $P_1 + Q_1$, ...}. Let GCD of all elements in B equals to x. The fact that x divides $P_0 + Q_0$ and $P_0 + Q_1$ implies that x also divides $Q_0 - Q_1$. Deriving for all differences, later we will notice that x divides all $P_0 - P_i$ and $Q_0 - Q_i$. Therefore, instead of working on all elements in B, we can work with C which consists of $P_0 + Q_0$, $P_0 - P_1$, $P_0 - P_2$, ... $P_0 - P_i$, $Q_0 - Q_1$, $Q_0 - Q_2$, ..., $Q_0 - Q_i$. The size of C is only 1 + |P| + |Q|.

In general, we only need to compute GCD of all elements in C which contains the sum of all the first elements in each array and the difference between each element with the first element in the array. Actually, it doesn't need to be the first element, but it's good to have a deterministic approach. Size of C is equal to the total size of all arrays + 1.