Central European Olympiad in Informatics
Tîrgu Mureş, România
July 8 – 14, 2009

Official Solutions

Solution for "boxes" (Day 1)

We group the boxes into "piles." Initially, a pile consists of precisely 100 boxes, followed by precisely 100 empty spaces. In time, the pile may grow up to 199 boxes, and the empty spaces after it may decrease to 1.

Each pile and the empty space after it occupy 200 positions of the tape. Thus, there is a lot of free space after all the piles (initially, half of the tape remains a contiguous interval of empty space). This large empty interval is called *the bubble.*

To insert a new box after box *k*, we find the pile that contains box *k* and check whether *k* is closer to the beginning of the end of the pile. If it is closer to the beginning, we push all boxes before *k* to the left by one position; otherwise we push all boxes after *k* by one position to the right. Since a pile does not exceed 200 boxes, this insertion takes at most 100 moves.

To ensure that piles stay small, we also run a "background process" that rebalances the piles. One step of the background process forms a new pile at the end of the bubble. Specifically, it grabs the 100 boxes before the bubble (these may appear at any position and may be part of different "old piles"). These boxes are pushed to the end of the bubble, where they become a new pile of ideal size (one hundred). Notice that the bubble keeps sliding towards the left (boxes before the bubble are moved after the bubble).

For each insertion, we run one step of the background process. Thus, the number of moves per insertion is at most 200.

There are at most 10,000 boxes on the tape, and 100 boxes are rebalanced for every insertion. Thus, every box takes part in the rebalancing process after at most 100 insertions. We conclude that at most 100 boxes can be inserted to any pile before the pile gets destroyed by rebalancing. Thus, all piles stay small.

The implementation of this idea is quite easy (~ 20 lines of code).

In general, the asymptotic number of moves of the strategy described above is $O(\mathrm{sqrt}(N))$. It is possible to achieve $O(\lg^2 N)$, which we leave as an exercise to the reader. We do not know of any solution better than $O(\lg^2 N)$. If you find one, we would be glad to hear it!

Testing. Several tests had quite simple structure: (1) each insertion happened after a random box; (2) each insertion happened after a fixed box *k*; (3) each insertion happened after the *k*-th box on the current belt; etc.

The next idea was to find the most loaded segment of length *L* on the belt, for some fixed *L*. At each insertion, we find the most loaded belt segment and insert a new box randomly inside that segment.

The downside of this strategy is that it only “attacks” segments of a fixed length *L*. Our most complex test strategy tried to attack segments of any length, uniformly. Specifically, we build a balanced binary tree over the belt. For each insertion, we pick a random level of this tree. Then, we search for the most loaded node at that level (in the sense that the subtree of the node has the most number of boxes). We insert a box randomly in the subtree of this maximally loaded node.

We implemented many hacks that did not have a guaranteed complexity. Most failed for the basic adaptive strategy (finding the most loaded segment), and all failed for the second strategy (based on the binary tree).

Solution for “photo” (Day 1)

The key observation is that any two rectangles in an optimal solution are either disjoint or nested (one of them is taller, and has the base included in the other). If two rectangles were to intersect in any other way, the shorter one can be shrunk horizontally until it is disjoint from the other.

We can decompose an optimal solution by recursively splitting the initial set as follows:

- If the covering rectangles in the optimal solution can be split in two sets using a vertical line that does not intersect any of the rectangles, split the set of points according to this line. We have *N* possibilities to try for such a horizontal partition.
- If the rectangles cannot be split as above, then there is a rectangle whose base covers all the others'. This rectangle must cover the projection of all points on the *x*-axis, so it will stretch between the minimum and maximum *x* coordinate of the points in the set. Without loss of generality, this rectangle can be as tall as possible, given the constraint on the base.

Considering these two cases, we can see that each relevant subset of the initial set is delimited by three lines: it lies between two vertical lines, and above a horizontal line. Let $C(i,j,k)$ be the minimum number of rectangles needed to cover the points $C(x,y)$ with the following properties:

- $x_i \leq x \leq x_j$
- $y_k \leq y$

We can now solve the problem using dynamic programming:

$$C(i,j,k) = \min \left\{ \begin{array}{l} \min_{x=i..j-1} (C(i, x, k) + C(x+1, j, k)), \\ 1 + C(newi, newj, newk) \end{array} \right\}$$

... where *newi*, *newj*, *newk* are the points on the left, right, and down border the subset of points not covered by the base rectangle.

There are $O(N^3)$ states and for each, we perform $O(N)$ operations. Thus, the time complexity is $O(N^4)$. Solutions based on backtracking earn between 20 and 40 points, depending on the optimizations used in the backtracking.

Solution for "harbingers" (Day 1)

A quadratic solution. The first approach is to use dynamic programming. If we consider opt_i to be the minimum time required to send a message from town *i* to the capital, then we have the following recursion:

$$opt_i = \min (opt_j + dist_{j->i} \cdot V_i + S_i) \qquad (1)$$

where *j* is a node on the path from town *i* to town 1. This relation is obtained by considering every harbinger *j* that may receive the message from harbinger *i*. By $dist_{j->i}$ we denote the distance between nodes *j* and *i*. This distance can be computed in constant time if we initially compute a vector *D*, where D_i is the distance from town *i* to the capital. A direct implementation of (1) takes $O(N^2)$ time and earns 20 points.

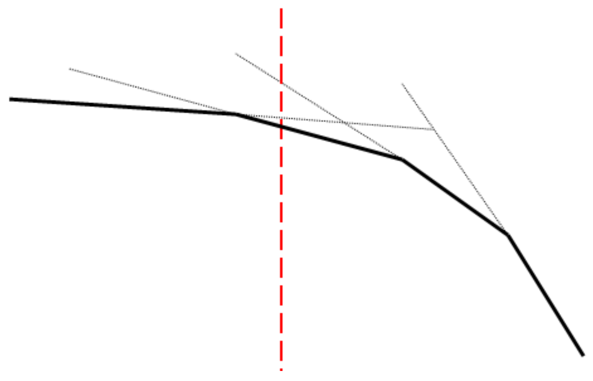
The special case of lines. Recurrence (1) can be rewritten as:

$$opt_i = \min (opt_j - D_j \cdot V_i + D_i \cdot V_i + S_i) \qquad (2)$$

When computing the value for opt_i, $D_i \cdot V_i + S_i$ is constant for all *j*, so:

$$opt_i = \min (opt_j - D_j \cdot V_i) + D_i \cdot V_i + S_i \qquad (3)$$

We need to find the minimum of expressions of the form $opt_j - D_j \cdot V_i$. This has a useful geometric interpretation: for each node *i*, we consider a line in the plane given by the equation $Y = opt_i - D_i \cdot X$.



Finding the minimum is now equivalent to vertical ray shooting: find the lowest line in the plane intersected by the $X=V_i$ line. To solve this query efficiently, we maintain the lower envelope of the lines. Because the road lengths are positive, we have that $D_j < D_i$ for every *i* and *j*, where *j* is an ancestor of *i*. This means that the slopes of the lines in the envelope are descending. Thus, we can store the lines in a stack.

At each step *i*, we must:

- query which line in the envelope produces the minimum for $X=V_i$
- insert $Y = opt_i – D_i \cdot X$ in the envelope (possibly deleting others).

The first step can be solved efficiently using binary search, since the intersections of $X=V_i$ with lines in the envelope are a unimodal function.

The second step can be solved in O(Δ), if Δ lines are deleted from the envelope. We simply pop the head of the stack as long as the newly inserted line completely shadows the segment at the head of the stack. Since every line in the stack is popped from the stack at most once, the overall complexity of this step is $O(N)$. Thus, the entire algorithm runs in $O(N \lg N)$ time.

The general case. To handle arbitrary trees, we implement the dynamic program by depth-first search. To maintain the lower envelope during DFS, we have to support two data structure operations:

1. Insert a line efficiently (when moving down to a child).
2. Delete a line and restore the envelope to the previous state (when moving up to a parent).

Point 1. cannot proceed as for line graphs, since a line can be deleted multiple times. Quadratic behavior is obtained if a node has many children and the Δ is large between the node and each child. To insert a line in $O(\lg N)$ time, we can once more use binary search on a unimodal function. (Solutions that do linear insertion here earn 70 points.)

For the undo operations, we maintain the lower envelope in a global array. To delete Δ lines, we simply store the new line Δ positions back from the top of the stack, and leave the other Δ-1 positions intact. We also save the line that was overwritten by the newly inserted line. To restore the stack, we simply reset its top pointer, and restore the saved line to its original location in the stack. The overall complexity is $O(N \lg N)$.

An alternative visualization of the algorithm does not use lower envelopes, but linear programming; we maintain the convex hull, and use unimodal search to find the optimum.

Though the reasoning behind the algorithm is quite complicated, the code is fairly short (~ 40 lines).

Solution for “logs” (Day 2)

The simplest solution runs in O($N^2 \cdot M$) time, and earns up to 30 points. Choose two row indices *i* and *j*, and assume the largest rectangle is between these rows. It remains to find all columns that filled with ones between rows *i* and *j*.

A much better solution runs in time O($N \cdot M \cdot \lg M$), and earnes up to 60 points. The idea is to sweep through each line *i*, and maintain the “height” of each column, i.e., the number of ones in that column extending upwards from line *i*.

Given the heights of each column, one has to find a subset *S* of columns maximizing $|S| \cdot \min(S)$. You can find this subset by trying values for min(*S*) and counting how many heights are greater or equal. By maintaining the heights in sorted order, this computation becomes linear time. A standard sorting algorithm takes O($M \cdot \lg M$) per line.

One can replace the sorting algorithm with counting sort, taking advantage of the fact that, for line *i*, the height values are integers in range [0..*i*]. This leads to a running time of $O(N \cdot (N+M))$, which earns up to 80 points.

The optimal solution, earning full score, runs in O($N \cdot M$) time. The key observation is that, while the sweep line advances to another row, a height is either incremented or reset to zero. This means that one can maintain the sorted order in linear time: place the values that become 0 up front, and maintain the order for the rest (which is unchanged by incrementing them).

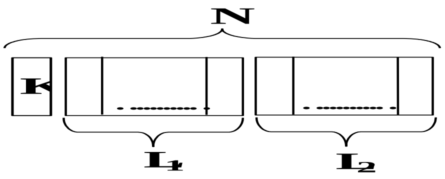
Solution for “sorting” (Day 2)

The solution is based on dynamic programming, and runs in O(N^{10}) time. A slower dynamic program, running in time O(N^{12}), earns 70 points. Solutions based on backtracking earn 20 points.

We begin by the describing the O(N^{12}) solution. The number of comparisons for Insertion Sort is the number of inversions in the given permutation, plus $N - 1$. We can focus on the number of inversions, which is at most N^2.

Let *N* be the size of the permutation. We fill the dynamic program by increasing values of *N*. Conceptually, during each step we are inserting a new number in front of an existing permutation. Let this number be the *k*-th in sorted order. Since it is added in the front, the new value will also be the pivot for the Quick Sort call. The permutation is partitioned into a list of length $K - 1$ and another of length $N - K$.

Official Solutions



We iterate over the number of comparisons needed by Quick Sort in the first and the second lists; let these be Q_1 and Q_2. We also iterate over the number of inversions in each part; let these be I_1 and I_2.

The overall number of comparisons made by Quick Sort is Q_1+Q_2+N-1. To compute the number of inversions, we need to understand the number of ways in which the two parts can be interleaved, i.e., find the distribution of the number of inversions between elements of the two parts.

This is a separate dynamic program. Let $D[L_1][L_2][I]$ be the number of permutations containing I element pairs (x_1, x_2) such that:

- x_1 is from the first part (which has length L_1);
- x_2 is from the second part (which has length L_2);
- $x_1 > x_2$.

The recurrence for this dynamic program is straightforward:

$$D[L_1+1][L_2][i+L_2] = D[L_1+1][L_2][i+L_2] + D[L_1][L_2][i]$$
$$D[L_1][L_2+1][i] = D[L_1][L_2+1][i] + D[L_1][L_2][i]$$

At any point you can append elements belonging to the first partition (first line) or the second partition (second line).

Now that we have D, the number of inversions is the number of inversions for each partition, plus the number of inversions between the two partitions, plus $k-1$ for the newly inserted element.

Assuming that Q_1 and Q_2 are given (we iterate over them), this is the main iteration for the $O(N^{12})$ solution, where $cmax(n) = n^2$ is the maximum number of comparisons of each algorithm.

```
for $I_1$ = 0 to $cmax(L_1)$ do
    for $I_2$ = 0 to $cmax(L_2)$ do {
        $Q = Q_1 + Q_2 + N - 1$
        for $I_c$ = 0 to $L_1 \cdot L_2$ do {
            $I = I_1 + I_2 + I_c + K - 1$
            $A[N][Q][I]$ += $A[L_1][Q_1][I_1] \cdot A[L_2][Q_2][I_2] \cdot D[L_1][L_2][I_c]$
        }
    }
```

To reduce the running time, we notice that the iterations over I_1 and I_2 are "independent" of I_c. If you look at the inner most iteration you see that $A[L_1][Q_1][I_1] \cdot A[L_2][Q_2][I_2]$ doesn't change as I_c changes. Also the recurrence line doesn't care about I_1 and I_2. It only cares about $I_1 + I_2$.

This means we can precompute sum($A[L_1][Q_1][I_1] \cdot A[L_2][Q_2][I_2]$) for the same $I_1 + I_2$.

```
for $I_1 = 0$ to $cmax(L_1)$ do
        for $I_2 = 0$ to $cmax(L_2)$ do
                $S[I_1 + I_2] = S[I_1 + I_2] + A[L_1][Q_1][I_1] \cdot A[L_2][Q_2][I_2]$
$Q = Q_1 + Q_2 + N - 1$
for $SI = 0$ to $cmax(L_1) + cmax(L_2)$ do
        for $I_c = 0$ to $L_1 \cdot L_2$ do {
                $I = SI + I_c + K - 1$
                $A[N][Q][I]$ += $S[SI] \cdot D[L_1][L_2][I_c]$
        }
```

Solution for "tri" (Day 2)

The brute-force solution, earning 20 points, runs in $O(K \cdot M)$ time. For each pair (query point, triangle), one can check in constant time whether the point is inside the triangle, using elementary geometric calculations.

Special case: right triangles. As stated in the problem text, one can obtain 50 points by only dealing with triangles that have one edge on the x-axis, and another on the y-axis. The solution runs in $O((K + M) \cdot \lg K)$ time.

First, one can find the set S of points on the convex hull which are visible from the origin. This can be done in $O(K \cdot \lg K)$ using any convex hull algorithm. Subsequently, each triangle query can be answered in $O(\lg K)$ time – we can use binary search to test whether a line intersects a convex polygon. The details are simple: for each line, the normal to the line is described by a linear function $f: \mathrm{R}^2 \to \mathrm{R}$, $f(x,y) = a \cdot x + b \cdot y + c$. It is easy to see that the function is unimodal on a convex set of points. Thus, we can use a variation of binary search to find the minimum of the function. If any point is inside the triangle, the minimum point must be inside.

The general case. Generalizing this solution to arbitrary triangles uses divide and conquer. First, we sort the K points by polar angle. Now consider a balanced binary tree over the points. In each node, we only care about the convex hull of the points appearing in the node's subtree (as before, only those visible from the origin).

A triangle is decomposed into $O(\lg K)$ triangles; each one spans a node of the binary tree. Thus, inside each node we only need to compare the points with the third triangle side (the one not containing the origin). For that, we can use the binary search solution from the special case.

Overall, the time complexity is $O((K + M) \cdot \lg^2 K)$, which earns 100 points. It is possible to further improve the solution to $O(\text{sort}(K+M) \cdot \lg K)$, where sort($N$) denotes the time to sort N numbers. Indeed, the complexity of convex hull is equal to sorting, and the binary search can be replaced by sorting (sort all lines by slope, and then "merge" them with the points of the convex hull in linear time). By using radix sort for sorting, the running time is improved significantly. However, this improvement is not needed for a maximal score.

Test cases. In the test cases for the special case, most points are on the convex hull (with a few points that don't matter inside the hull). The queries are tangents to the convex hull, either moved "in" (to make the answer yes), or "out" (to make the answer no).

For the general case, we have two similar strategies:

1. We generate sqrt(N) "top level" points in convex position. Between any consecutive pair (looking at the angle from the origin), we insert sqrt(N) "bottom level" points. The bottom points are also in convex position, and further out than the top-level points. For the queries, we draw tangents, either to the top convex hull, or to a random bottom convex hull.

2. We consider an array $A[1..K]$, where every $A[i]$ has value x with probability $O(2^{-x})$. We consider a random line, and transform each $A[i]$ into a point shifted inside from the line by ~ $A[i]$. The data looks like $O(\lg K)$ nested hulls, each having few points. For the triangles, we pick a random level, and take a tangent to a hull of that level.