Triangle – Analisys

Author: Boris Mihov, Solution and Analysis: Emil Indzhev

1 Initial thoughts

In problems of "finding a hidden permutation" the most common approach is to reduce the problem to sorting, i.e. to figure out how to perform a comparison between two elements. Unfortunately that doesn't seem quite possible.

A next step (or perhaps something we've already considered) is to look at "simple" queries (or sets of queries) that give some sort of easily interpretable information: (a, a, b). The only case in which it returns `false` is when $2a \leq b$. So this gives us something like a comparison. The solutions we'll look at will use primarily this type of query or variations thereof.

Another useful thing to consider, before attempting a solution is to compare the target to the theoretical lower bound for finding a hidden permutation using binary queries. The theoretical minimum is $\log_2(N!) \approx N\log_2(N) - \log_2(e)N \approx 8523$, while the target is 8770. This tells us we are looking for something very optimized and close to the theoretic minimum. This is important to keep in mind, so we know that when deciding "should I just do this slightly easier but with double the queries", the answer is no (if we are trying to get close to 100 points). In fact, not only the scaling of the $N\log_2(N)$ term matters, but even on the (negative) linear term. This, of course, also depends on the scoring, but plugging in some sample values into the scoring function confirms this.

2 $O(N^2)$ solution

Multiple $O(N^2)$ solutions are possible based on the above simple query. An example one is to ask all queries of the form (a, a, b) (where $a \neq b$) and then sort the b-s based on how many of those failed. This gives us a full sorting of the numbers up to $N/2$, since for each a we know how many b-s are $\geq 2a$. Then, for the remaining numbers, we have a nearly complete sorting: for each b we know how many $2a$-s are $\leq b$; the issue is that numbers of the form $2k$ and $2k + 1$ are not distinguishable. However, this can be corrected by a linear pass. Suppose we have fixed all numbers before $2k$; then check if $(k, k + 1, b)$ returns `true`: if yes, then this is $2k$, if no, then it is $2k + 1$.

3 Divide and conquer ideas

The main query we are currently working with does not let us directly compare, but it does let us divide numbers into a prefix and suffix. We can fix a and iterate over all b-s and partition them up this way, or fix b and iterate over all a-s and partition those. Both can lead to solutions, but the latter seems better. In the first idea, if $a \geq N/2$ (which has $1/2$ probability), the partition does nothing. Similarly, if a is a bit under $N/2$, the partition is very imbalanced (though we do get a perfect split when a is around $N/4$ which is not that rare). On the other hand, the latter idea is more consistent: when b is around $N/2$, we still get an okay partition ($1/4$ to $3/4$) and this improves as b increases (ending with a perfect split when $b = N$).

Unfortunately, as soon as we do one partitioning, the upper half can usually no longer be partitioned using these ideas (e.g. if we get the perfect split the first time, no queries of the form (a, a, b) in the second half give us any information). Luckily, at least the lower half is equivalent to the original problem (but with a smaller N) and we can keep doing this there.

This process (which recurs only on the lower half) would give us a sequence of (on average) geometrically decreasing parts (ideally $N/2, N/4, ...$, if we have lucky splits). Notice that the last partition will always have size 1 (and thus contain just the number 1). It would also take $O(N)$ queries.

However, this idea doesn't lead to a solution so we leave it aside for now and figure out how to use it later.

4 Binary search ideas: a $2N\log_2(N)$ solution

Another thing we should consider doing in tasks of "sorting" is doing a binary search. Suppose, we've managed to figure out all numbers up to $N/2$, then for all remaining numbers we can just do a binary search. To check if a number is $\leq x$ for some x, we just take two numbers that sum up to x (easiest is $\lfloor x/2 \rfloor$ and $\lceil x/2 \rceil$) and use those in a query with our number. More generally, if we know all numbers up to k, we can binary search all numbers up to $2k$. Therefore, one simple idea is to start with some "small" sorted set (just 1 and 2 is enough), then for all remaining numbers find which ones are $< 2k$ (since we cannot distinguish between the ones $\geq 2k$) and for each of those do a binary search. This does about $2N\log_2(N)$ queries ($N\log_2(N)$ for binary searches and for checks).

Now, we have to figure out how to find 1 and 2. What we can do is use the divide and conquer idea. It directly gives us 1. Then look at the penultimate partition. There are 3 cases depending on its size):

1. If it has size 1, that number is 2.
2. If it has size 2, those numbers are just 2 and 3. The query $(2, 2, b)$ fails for all $b > 3$, while $(3, 3, b)$ succeeds for $b = 4$ and $b = 5$. Therefore, we can take one of the two numbers a and iterate through all remaining b asking (a, a, b) – if all fail, $a = 2$, otherwise $a = 3$ (of course, we can break as soon as we get a success). As an optimization, we can iterate only through all b in the previous partition (which will always contain at least 4 and hopefully only a few other numbers).
3. If it has size ≥ 3, it will contain both 2 and 4. Notice that $(2, 2, 4)$ is the only (a, a, b) query that fails for $b = 4$ or $b = 5$. Additionally, $b = 4$ and $b = 5$ are the only b-s for which only a single a fails. Therefore, we iterate over all b-s and a-s and stop when we find a b with a single failing a; that a is 2. With some breaks (such as when we get two failures and when we find a b with only one failure), this is quite efficient (especially because on average this is a small partition).

5 Taking better advantage of the partitions: an $N\log_2(N)$ solution

In our previous solution, the $N\log_2(N)$ for binary searching seems fixed. However, just as many queries are spent on silly checks of the form "is this number within the binary search range" (where the answer is no for almost all numbers, for most of the iterations of this). On the other hand, we have a partitioning procedure (which we currently just use to find 2 and 3), which leaves us in a much better spot to answer this question.

We will do our binary searching and overall sorting inside the recursive partitioning procedure; i.e. for a given iteration of the solution: we first partition the current chunk, then recur on the lower half, then sort the upper half, using the already sorted lower half. In the ideal world, where all splits are perfect, this directly works (since we've sorted the numbers up to $N/2$, we can binary search for the rest). Unfortunately, we are not in this ideal world.

Suppose, we've sorted the numbers up to $k \geq 2$ (when $k = 1$, we use the same three cases as described above to find the 2) and the current chunk is from $l = k + 1$ to r; also suppose $r > 2k$ (otherwise we can directly binary search). We can, just like in our previous solution, ask all queries of the form (k, k, b), but only for $l \leq b \leq r$. After this split, the lower half is sortable, and we can recur with this sorting procedure on the upper half ($2k$ to r). Most of the time, this upper half will also be directly binary searchable, unless $2(2k - 1) < r$, i.e. $4k - 2 < r$, which would happen a bit less than $1/2$ of the time when partitioning originally. Otherwise, we may need a 2nd or even 3rd (etc.) splitting, but each of those is with halving probability. Therefore, the average number of runs of "split all numbers relative to $2k$" is less than 2 and this is done for each partition. Therefore, this procedure leads to only a linear term for the checks of the form "is this number within the binary search range".

6 Even better splitting: optimizing the linear term

Currently, we just check whether all numbers in the range of the current partition are binary searchable: does $2k \geq r$ hold. If not we split them up based on this by doing the (k, k, b) queries to compare all numbers in the partition with $2k$. Instead, we can just check "is the first step of binary searching them doable". The first query to make would be a comparison with $(l + r)/2$ (using $\lfloor (l + r)/4 \rfloor$ and $\lceil (l + r)/4 \rceil$). If those are $\leq k$, we can do the first step of the binary search of the numbers in the partition, so we should do that. Instead of structuring it like a binary search, we do it as a split for divide and conquer. Then we simply proceed with the lower half and then the upper half. The majority of the time, when we would have needed to do the bad/inefficient splitting, we can just do the first query correctly, and then when we reach the upper half, it is now fully binary searchable.

Code-wise, this just looks like taking our previous code and instead of doing queries (k, k, b), we do queries (q_1, q_2, b) where $q_1 = \min(k, \lfloor (l+r)/2 \rfloor)$ and $q_2 = \min(k, \lceil (l+r)/2 \rceil)$. In fact, once we do this, we don't need the separate binary searching code, as this handles both the bad split cases and the good binary search cases with a general divide and conquer approach.

7 Optimizing the binary searches

Supposing now, that the numbers in a partition are all fully binary searchable (we need to bring back a separate function for that case). Let the current chunk's size is $m = r - l + 1$. With our solution, we do $\log_2(m)$ queries for each (rounded up or down, depending on chance), for a total of about $m \log_2(m)$. However, the theoretic minimum for figuring out this sequence with binary queries (of some form) is $\log_2(m!) \approx m \log_2(m) - \log_2(e)m$. Therefore, there is probably some room for improvement.

An obvious case is the last number to binary search: there is only one option for it (by exclusion), so we should do no binary searching. The number before that only has two options, so we should do just one query. This generalizes. We keep a sorted vector of possible values (initially all values in the range l to r). Then, we repeatedly binary search on the index within that vector (using the value at the index of the current mid to figure out what query to do); and after finding the correct index (and thus value), we delete the value from the vector.

This optimization makes the binary searching more efficient – quite close to the $m \log_2(m) - \log_2(e)m$ (an easy upper bound of the new procedure is $m \log_2(m) - m$, since the second half of the searches take one fewer queries, the last quarter of searches take another query fewer, etc.)

8 Smart pivot for the initial partitioning

The initial partitioning we do has a non-small impact on how efficient our solution is: in the worst case, our first partitioning uses 1 as the "pivot" and we are left with a slightly optimized version of the $2Nlog_2N$ solution.

In traditional quick sorting (which this should remind us of), we may take a random sample of a few numbers, sort those and use their median as the pivot for the split. Here, we actually want as large a number as possible. So, one thing we can do is take two random numbers p_1 and p_2 and check if $2p_1 \leq p_2$, in which case use p_2 (we don't care if $2p_2 \leq p_1$, if we're not using it anyway). This is already quite impactful, as our worst cases are when the pivot is a very small number, in which case almost any p_2 will replace it (and it will usually be a random fairly big number). We can, however, repeat this process a few times: generating a new p_3 and checking if the current pivot p (which is either p_1 or p_2) fulfills $p_3 \geq 2p$, and so on. This was empirical best with 7 iterations and essentially guarantees we end up with a fairly large number. Of course, more involved procedures are possible too: take a set of random numbers and do all queries of the form (a, a, b) and take the b with the most failures. However, there is a tradeoff between cost and effectiveness and this is a lot more expensive (for the same number of potential pivots) without increasing the split quality that much.

9 Summary and code

You can find the code of the full 100 points solution in `triangle_emil.cpp`. Versions without these optimizations are not materialized, but it is easy enough to play around with the code to experiment. Relative to the solution after section 5, the order in importance/improvement of the described optimizations is (from highest to lowest): section 7, section 8 and then section 6.