The Collection Game (swaps)

BY ANTTI RÖYSKÖ (FINLAND)

Let $A_1, \dots, A_N$ be such that room i has the A_i-th most aesthetic art collection. Note that computing $A_1, \dots, A_N$ suffices to solve the task because *answer* should just output all rooms ordered according to increasing A_i. In this reformulation, a call to *schedule*(i, j) will then return whether $A_i < A_j$, potentially swapping A_i and A_j beforehand. So, we seek to determine $A_1, \dots, A_N$ after our last call to *visit*.

Subtask 1. $V = 5\,000$ and the museum never swaps art collections.

Because the A_i will never get swapped in the first two subtasks, a call to *schedule*(i, j) will determine whether $A_i < A_j$ after the last call to *visit*, and so whether room i should appear before room j in the output. Therefore, we only have to sort all rooms using our calls to *schedule* and *visit*.

For Subtask 1, a solution calling *schedule* and directly afterwards *visit* every time can in total call these functions $N\lceil \log N \rceil$ times without exceeding V visits. So, we can use any sorting algorithm which needs only that many comparisons to sort all the rooms.

Note that `sort` in C++ will not suffice for this purpose. However, we can insert the rooms one-by-one into a `vector` using binary search, for example with `lower_bound`. Also, a custom implementation of, for example, merge sort will work.

Subtask 2. $V \geq 1\,000$ and the museum never swaps art collections.

One option here is to use that fact that merge sort splits the list to be sorted into two equal halves which can then be sorted independently before merging them to obtain a sorting of the whole list. Since these sublists can be sorted independently and because we can compare disjoint pairs of indices at the same time, we can sort the two halves simultaneously. On the highest level, we will then need $N - 1$ comparisons to merge the lists, on the second level we will need $\lceil N/2 \rceil - 1$ comparisons for the merging, then we will need $\lceil N/4 \rceil - 1$ comparisons and so on until 1 comparison on the lowest level. In total, this gives $(N - 1) + (\lceil N/2 \rceil - 1) + (\lceil N/4 \rceil - 1) + \cdots + 1 \leq N + N/2 + N/4 + \cdots + 1 \leq 2N$ comparisons, solving Subtask 2.

There is also another solution which just compares all pairs of rooms. This can be done in parallel using a total of $2N$ calls to *visit*, and it is easy to reconstruct the order all rooms from these comparisons, for example by a topological sort. This also solves Subtask 2.

Subtask 3. $N \leq 100$, $V = 5\,000$

In this subtask, we can iteratively single out the smallest element A_j and recurse on the remaining elements.* To do so, we iterate $i = 1, \dots, N$ and keep the index j of the currently smallest element A_j among $A_1, \dots, A_i$. For $i = 1$, this is simply $j = 1$. Then, when moving from i to $i + 1$, we compare the elements at indices j and $i + 1$ (calling *schedule* and *visit* each once), and set j to the smaller of the two elements. Note that even if A_j and A_{i+1} are swapped beforehand, this will yield the smallest element among $A_1, \dots, A_{i+1}$.

* Also called "selection sort."

This approach needs $N - 1$ calls to *visit* to determine the smallest element of the list and afterwards reduces N by one. So, the total number of call to *visit* will be $(N-1)+(N-2)+\dots+1 = N(N-1)/2 \le N^2/2$ which suffices for Subtask 3.

Subtask 4. $V = 5\,000$

In fact, it is possible to determine the smallest element of the list using only $\lceil \log N \rceil$ many calls to *visit*. One way to do so is by creating a single-elimination tournament amongst the elements with $\lceil \log N \rceil$ many rounds.[†] That is, with one call to *visit* (and $N/2$ calls to *schedule*) we determine the smaller element of A_1 and A_2, of A_3 and A_4, of A_5 and A_6, and so on. Afterwards, there will only remain $\lceil N/2 \rceil$ many elements on which we can repeat this procedure until we are with left with only the smallest element of the whole list.

By proceeding as above on the remaining $N - 1$ elements, we can solve the problem using at most $N\lceil \log N \rceil$ calls to *visit*, solving Subtask 4.

Subtask 5. $V \ge 500$

Essentially, the two previous subtasks used selection sort and heap sort. What other sorting algorithm is there?

Well, there is bubble sort, but that needs N^2 many bubble steps. However, note that we can parallelize those bubble steps. Namely, in a single phase we could bubble the elements A_1 and A_2, the elements A_3 and A_4, and so on simultaneously. The same applies to A_2 and A_3, A_4 and A_5, and so on. By alternating between these two phases, this would eventually sort the list.[‡]

Two problems remain. First, how many phases does this process need? It turns out that it needs only N phases[§], as can be proven by a submission during the contest. Therefore, this would suffice for Subtask 5.

However, we still have to implement these phase using the functions *schedule* and *visit*. If *schedule*(i, j) always puts the smaller element of A_i and A_j into A_i, this works out of the box, earning 60% of the points for this subtask.

If not, we can use the following observation: Any solution relying on the fact that the smaller element gets moved into A_i for every call to *schedule*(i, j) can actually be adapted to a full solution as follows. If *visit* returns that the smaller element is in A_j, we simply exchange the indices i and j in every query from now on, and so we may assume that the smaller element will be A_i after a call to *schedule*. Note that many implementations of the above idea will do this implicitly anyway.

Subtask 6. $V \ge 100$

From this subtask on, it gets considerable harder to solve the problem. It turns out that the special case where the smaller element is always moved into A_i for each call to *schedule*(i, j) is the problem of designing sorting networks, and as we know from the previous subtask, this special case suffices to solve the problem completely.

[†] Also called "heap."
[‡] Also called "odd-even sort."
[§] See Wikipedia.

In Subtask 6, any sorting network of depth $\leq \lceil \log N \rceil (\lceil \log N \rceil + 1)$ will suffice to solve the problem. Here, it is possible to come up with your own solutions of a sorting network of such a depth, but which does not quite solve the next subtask.

For example, we can go for a merge sort style approach which first sorts the two halves of the list recursively and then merges them. To do this merging, we can iterate over a gap size $g = 2^k, 2^{k-1}, \dots, 1$ where k is the maximal value satisfying $2^k \leq N$ and bubble each element A_i with A_{i+g} (using two calls to *visit*). It can be proven that this sorting network correctly sorts any list.[¶]

Subtask 7. $V \geq 50$

For the last subtask, we need a sorting network of depth $\leq \lceil \log N \rceil (\lceil \log N \rceil + 1)/2$ to obtain all the points. Some standard sorting networks will work for that, for example bitonic mergesort, Batcher's odd-even mergesort or the pairwise sorting network. But it is also possible to come up with your own solutions.

[¶] Using the zero-one principle from Wikipedia.