Olympiads in Informatics, 2021 Vol. 15, 53–74
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.05

Why You Should Know and Not Only Use Sorting Algorithms: Some Beautiful Problems

László NIKHÁZY, Áron NOSZÁLY, Bence DEÁK
Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
e-mail: nikhazy@inf.elte.hu, noszalyaron4@gmail.com, deakbence2002@gmail.com

Abstract. In most programming languages, the built-in (standard library) sort() function is the most convenient and efficient way for ordering data. Many software engineers have forgotten (or never knew) the underlying algorithms. In programming contests, almost all of the tasks involving sorting can be solved with only knowing how to use the sort() function. The question might arise in young students: do we need to know how it works if we only need to use it? Also, why should we know multiple efficient sorting algorithms, is not one enough? In this paper, we help the teachers to give the best answers to these questions: some beautiful tasks where the key to the solution lies in knowing a particular sorting algorithm. In some cases, the sorting algorithms are applied as a surprisingly nice idea, for example, in an interactive task or a geometry question.

Keywords: programming contests, sorting algorithms, competition tasks, geometric algorithms, teaching algorithms.

1. Introduction

Today's students are best motivated to learn by seeing exactly how they can use their knowledge in the future. A computer programmer frequently uses library functions implemented by other programmers, particularly often the built-in or standard library data structures and algorithms of programming languages (e.g., the C++ standard library). We can define increasing levels of knowledge related to them: the interface, the complexity of operations, the theoretical background, and the implementation details. The question naturally arises: how much should students know about them? It depends largely on the level of students and the concrete algorithm or data structure itself.

We focus on talented high school students whose aim is to perform well at competitions. Without question, they need to know the interface and the complexity of standard library elements to succeed in competitive programming. Also, most of them will learn the theoretical backgrounds during university studies in computer science. Is it useful to teach them ahead of university? Well, it makes sense to examine this question sepa-

rately for different algorithms and data structures. In this article, we focus on the sorting algorithms. We would like to make a point that students should know several efficient and less efficient sorting algorithms if they want to excel at national competitions and participate in international ones by showing some problems where this knowledge is a great advantage. With a short survey presented in section 2, we verified that it is a matter worth investigating.

Section 3 presents six tasks from various sources that can be solved by modifying a sorting algorithm, but the standard library sort() function is not applicable. These problems might be given as follow-up tasks after teaching the corresponding sorting algorithm to demonstrate some unusual applications. In multiple cases, the topic of the task is from another domain, e.g., geometry, interactive problems, or graph theory, and the fact that a field of computer science helps tackle a problem in another area makes them beautiful solutions, in our opinion. When discussing implementation details of the solutions, we consider the features of the C++ standard library since it is the most used language in competitive programming (Halim *et al.*, 2013).

2. A Quick Survey

We conducted a little, non-representative survey with 26 students participating from the most talented young Hungarian programmers aged between 13 and 19 years. The questions were about Quicksort (Hoare, 1961a), Merge Sort (Knuth, 1997c), Insertion Sort (Knuth, 1997b), and Minimum/Maximum Selection Sort (Knuth, 1997d, referred to as Selection Sort in the following), whether they know these algorithms, and whether they

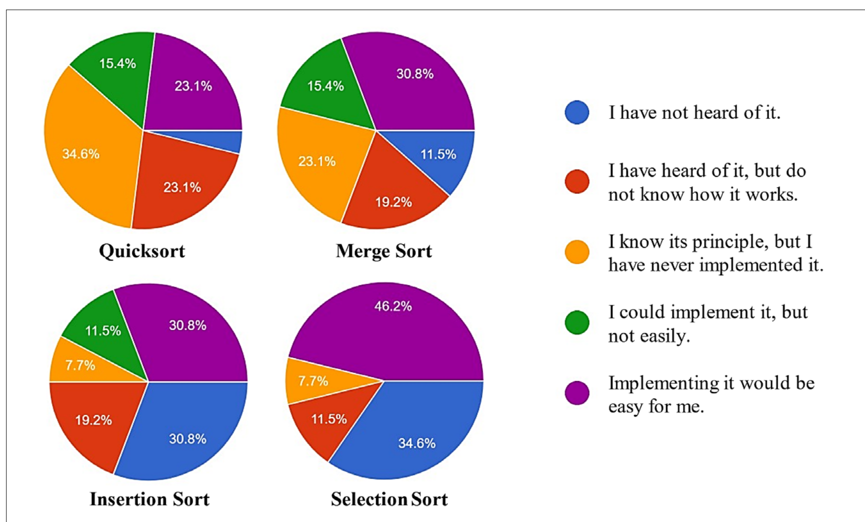


Fig. 1. Results of a non-representative quick survey with gifted pupils about sorting algorithms.

have implemented them already. We also asked if they use the built-in sort() function often, and more than 95% of the students have used it already, 80% very frequently. The results presented in Fig. 1 are intriguing and justify the importance of the topic of this article.

Surprisingly, the two simpler and less efficient sorting algorithms are less known, although students are, in general, more confident in implementing them. Still, it is true for all four algorithms that more than half of the students have never implemented them, and typically about 30% of the pupils think that the implementation would not be a problem for them. We want to mention that we also included a verification question about each algorithm's worst-case complexity, and out of those who stated that they know it, about 30% replied wrongly for Quicksort, 10% for Merge Sort, and 7% for Insertion Sort.

3. Tasks and Solutions

3.1. *Matching Nuts and Bolts*

This problem was first mentioned as an exercise in (Rawlins, 1992, p. 293). Since then, there have been numerous publications related to fast deterministic solutions (Alon *et al.*, 1994; Bradford, 1995; Komlós *et al.*, 1998). We present a version that was featured in the Hungarian IOI Qualification Competition, 2019.

Statement (shortened)

There are N bolts of distinct sizes in one box and N corresponding nuts in another one. Your task is to find the matching nut for every bolt. However, the only operation you can perform is comparing the size of a bolt and a nut by trying to match them, and you have a limited number of such queries. You cannot compare two bolts or two nuts.

This is an interactive task. Inside the solution, you can call the check(i, j) function at most M times in total, to compare the ith bolt to the jth nut, and it will return -1, 0, or 1 if the nut is smaller, the same, or bigger than the bolt, respectively. Important note: the solution is fixed throughout the interaction, i. e. the grader is not adaptive.

Subtasks and constraints

In the Hungarian contest, there was a single subtask: $M = N^2/4$, and $N \geq 40$. However, there is a bit more than a single idea in this task, so there could be other subtasks with different limits, to award suboptimal and simpler solutions as well. We suggest some:

i) $N = 2, M = 2$
ii) $N = 10, M = 100$
iii) $N = 100, M = 5000$
iv) $N = 1000, M = 250\,000$
v) $N = 10^5, M = 10^7$

Solutions

We can select the pair of a bolt by trying out all the N nuts, leading to an $M = N^2$ solution, which is enough to solve subtask ii. If we exclude the nuts for which the pairs have already been found, we can solve the task with $M = N(N-1)/2$ operations that is sufficient for subtasks i–iii. How can we improve this? We might use the information obtained when comparing a bolt with all the nuts – we can form two sets, the bigger and the smaller nuts. And here comes the interesting extra step: since we also have the matching nut, we can compare that with all the bolts to partition them into two sets, too. Those two sets of nuts and bolts will correspond to each other, so we can repeat the process within each of them. Fig. 2 highlights the main steps of the algorithm.

It is a classic divide-and-conquer solution very similar to the Quicksort algorithm (Hoare, 1961a). In fact, it is best to implement the above-shown partitioning by reordering the elements, just like in Quicksort. The number of comparisons is $O(N^2)$ in the worst case, when one of the partitions is always empty. Thus, it is crucial to introduce some randomization, otherwise, there might be a test where the worst case happens. It can be proven similarly to the complexity analysis of Quicksort (Cormen *et al.*, 2009, pp. 170–190), that with randomization, the average number of comparisons is $O(N \log N)$, which easily fits into the limits of subtasks iv–v, so we can expect that the solution passes for all testcases, maybe with tuning the random seed a couple of times.

Analysis

The algorithm is almost the same as Quicksort, but in this problem, two collections need to be sorted in parallel. This fact brings the sole difference: for partitioning one collection, we use an element of the other collection and repeat this step correspondingly in the reverse direction. Naturally, knowing the Quicksort algorithm means a massive advantage for solving this task. The connection is also very important for analyzing the

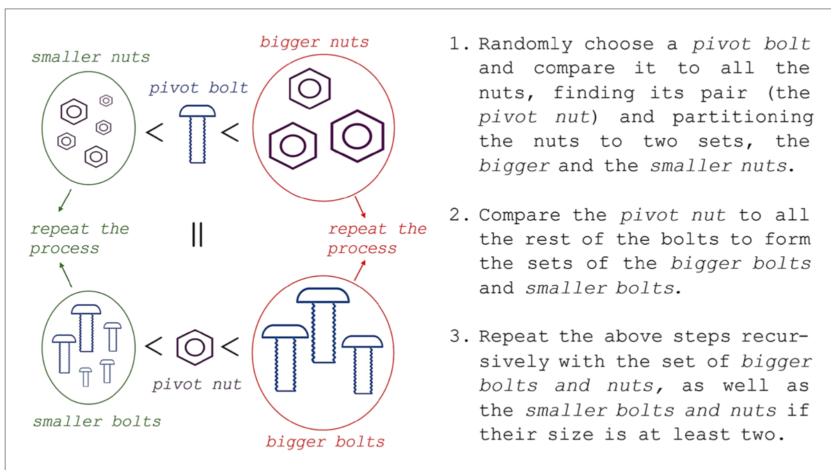


Fig. 2. Steps of the Quicksort-like solution of the Matching Nuts and Bolts problem.

complexity of the solution and knowing that hitting the $O(N^2)$ worst-case complexity can be avoided by randomly choosing the pivot element. It must be noted that the non-adaptive nature of the grader is an essential detail in this regard. With this, we can rely on the knowledge that the expected complexity of Quicksort is $O(N \log N)$.

In general, the task can be seen as a sorting problem instead of the matching problem. It is sufficient to sort both the collections of nuts and bolts. Certain other sorting algorithms might be used as well. For example, we can do a Bubble Sort (Knuth, 1997a) by comparing a nut and a bolt at the same position in the sequence and swapping the bigger one (or any if they are equal) with the next element and then moving to the next position. After two such passes, the biggest items will be at the end of both sequences. Although this is an excellent idea, the complexity of this solution is $O(N^2)$.

Further questions

The solution presented above includes randomization. It is a natural question whether there is a deterministic algorithm with worst-case complexity better than $O(N^2)$ for this problem. There are numerous articles presenting different approaches. First, Alon and colleagues (1994) gave a deterministic $O(N \log^4 N)$ algorithm with the idea finding a good pivot for the partitioning in the Quicksort using expander graphs, later Bradford (1995) showed the existence of an optimal, $O(N \log N)$ algorithm, and finally Komlós and colleagues (1998) found an optimal solution based on the AKS sorting algorithm.

3.2. *Median Strength*

This task was included in the problem set of IOI 2000. Afterwards, Horváth and Verhoeff (2002) published an article about its detailed analysis. A variant of this problem appeared recently in the Qualification Round of Google Code Jam (2021).

Statement (shortened)

There are N objects arranged in a line, each having a unique strength and a label assigned to them. You can compare three (x, y, z) objects by calling a function $\mathrm{Med3}(x, y, z)$ which returns the label of the object with median strength. Your task is to write a program that, among all N objects, determines the one with median strength with no more than M calls to Med3. N is guaranteed to be odd.

Subtasks and constraints

In IOI 2000, there were no subtasks, the points were given for each test case, and they had various sizes. The upper limits were $N \leq 1499, M \leq 7777$.

Solutions

There are several different approaches to this problem, and (almost) all of them require some knowledge of sorting algorithms. The first is based on minimum/maximum selec-

tion, and Horváth and Verhoeff call it onion peeling. In each iteration, we eliminate the two extremes in the following way. First, we pick any two available objects. Then, we iterate through all the other ones, keeping track of the minimum and the maximum we processed so far. See the pseudocode for a better understanding (Fig. 3). After $(N-1)/2$ iterations (and $(N-1)^2/4$ calls to Med3), the only remaining element will be the median of the original set of objects.

The second approach is based on Insertion Sort. We start with obtaining a sorted list of three objects by a single call to Med3. (Note that we do not care if the order is reversed since the final median can be identified anyway.) Each following object z can be inserted in between some (x, y) pair of consecutive objects, increasing the length of the sorted prefix by one. By calling Med3(x, y, z) for each consecutive pair, we can make the following observation: the function returns y for some (possibly 0) initial (x, y) pairs, then z for at most one pair, and finally x for the rest of the pairs. (Med3$(x, y, z) = z$ for exactly one pair, unless object z is the new minimum or maximum of the sorted prefix.)

We may naively search for the place of insertion by trying all (x, y) pairs one by one. Although this linear search is $O(1)$ in the best case, its worst and average-case complexity is still $O(N)$. However, our observation above suggests that the suitable position can be found using ternary search with worst-case time complexity of $O(\log N)$. See the pseudocode below for the details of the ternary search (Fig. 4). (We need an additional call to Med3 beforehand to handle the case where z is a new extremum.)

As a third approach, Quicksort can also be modified according to our needs: we simply have to use a ternary partitioning algorithm. This involves choosing two pivot elements and dividing the array into three subarrays. Moreover, since we are only interested in finding the median, we can base our solution around Quickselect (Hoare, 1961b) instead of Quicksort. The difference is that we do not need to recurse into such subarrays,

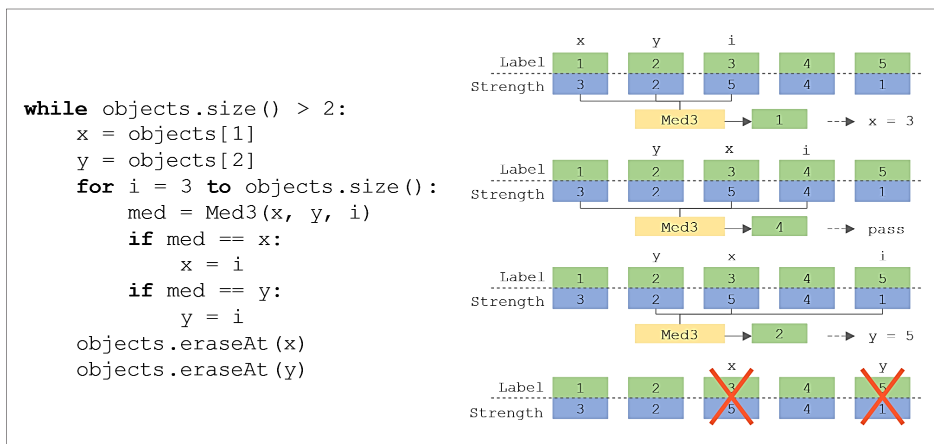


Fig. 3. Pseudocode and illustration of an iteration of the onion peeling method.

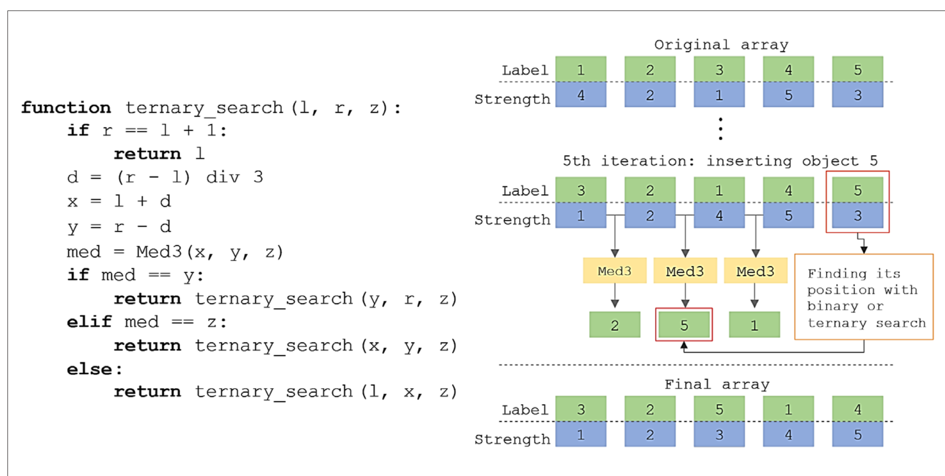


Fig. 4. Pseudocode of ternary search and overview of the insertion based method.

which are guaranteed not to contain the median. Although this improved algorithm still makes $O(N^2)$ calls to Med3 in worst case, the average time complexity is $O(N)$, which is sufficient to solve the problem.

Several interesting heap-based solutions have also been described in (Horváth & Verhoeff, 2002). These algorithms perform pretty well in practice, despite having an average/worst-case time complexity of $O(N \log N)$. We must note that there exist worst-case linear selection algorithms, given that we can compare any pair of elements in $O(1)$. Some of them, e.g. the Median of Medians (Blum *et al.*, 1973), could be modified to fit our case, but they are not advised to use due to their relatively high constant factor.

Further questions

This problem appeared with a different output in Google Code Jam 2021. In that variant, we must find one of the orders (non-descending or non-ascending) explicitly, so the application of some sorting algorithm is a straightforward idea.

We propose the following interesting, related problem. In a query, the median of any subset of objects can be asked, and the task is to determine the object with the Kth smallest strength. (The median of an even-sized set is the smaller one of the two middle elements.) Is it possible to describe an algorithm that uses at most $O(N)$ queries (if so, what is the upper bound)? What if we are not allowed to ask the median of two objects? The solution would be a special application of the Quickselect algorithm, in which we have the power to choose the median of all elements as the pivot. The worst-case linear complexity allows for giving a nice, $c \cdot N$ upper limit for the number of queries.

3.3. *Tournament*

This problem is about the algorithmic aspects of a famous theorem in graph theory, namely that every tournament (complete directed graph) contains a Hamiltonian path (Rédei, 1934). Hell and Rosenfeld (1983) describe an optimal algorithm similar to Binary Insertion Sort; we elaborate on applications of other sorting algorithms below.

The 2019 Nemes Tihamér Programming Competition (a Hungarian national programming contest for 9–10$^{\text{th}}$ grade students) included this task, phrased as constructing such an order of participants of a round-robin tournament, in which every player defeated the next in the sequence. We give an alternative problem statement that is less associated to sorting, which we heard from Lajos Pósa in his mathematics camps.

Statement (short versioN)

A faraway country consists of N islands. There is a strange public transportation system, dragons carry people between any two islands, but they only fly in one of the two directions. As a tourist, you want to take a route consisting of the most islands possible, without visiting any island twice. You can choose any island to start from.

This is an interactive task. The $\text{dir}(i, j)$ function returns true if the dragons fly from the ith to the jth island, and false if they fly in the reverse direction. You can call this function at most M times. You should give the solution as a list of island numbers in the order of visiting. Attention: the grader might be adaptive, meaning that the return value of the $\text{dir}(i,j)$ function may depend on the previous calls made to it.

Subtasks and constraints

Making the problem interactive allows us to differentiate solutions based on the number of edges that they inspect. In the Hungarian competition, the complete graph was given as the input, which prevents distinguishing $O(N \log N)$ and $O(N^2)$ solutions. With M as the upper limit of queried edges, we suggest the following subtasks:

i) $N = 4, M = 5$
ii) $N = 10, M = 50$
iii) $N = 1000, M = 500\,000$
iv) $N = 100\,000, M = 2\,000\,000$

Solutions

In the following, $x \longrightarrow y$ denotes that the edge between x and y is directed towards y (i. e. $\text{dir}(x, y)$ is true). Subtask ii can be solved with brute force; first, query all edges of the graph and then check all permutations of vertices. In subtask i, the number of queries is less than the number of edges ($5 < 6$)), so we need something smarter. If we query the 3 edges between 3 vertices, we can make a path of them: $x \longrightarrow y \longrightarrow z$. Then it makes sense to ask $\text{dir}(y, w)$, where w is the fourth vertex. One can see that depending on the answer, it suffices to check either $\text{dir}(x, w)$ or $\text{dir}(z, w)$ to be able to insert w into the path.

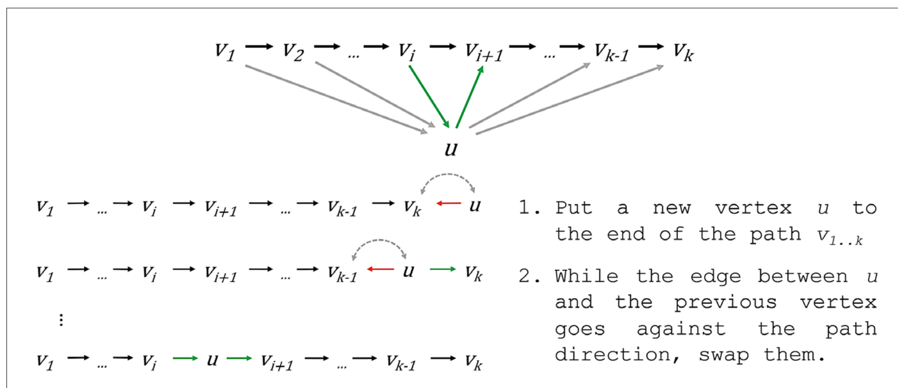


Fig. 5. Solution algorithm of the Tournament problem based on Insertion Sort.

Is there always a path through all vertices, like in the case of $N = 2, 3, 4$? The answer is yes, and we prove it by giving an algorithm that finds this path. Consider a path consisting of k points: $v_1 \longrightarrow v_2 \longrightarrow \ldots \longrightarrow v_k$. If there is a vertex u not included in this path, we can insert it as follows. If $u \longrightarrow v_1$ or $v_k \longrightarrow u$ then we can put u to one end of the sequence. Otherwise, looking at the edges between u and $v_{1..k}$, the first is an in-edge ($v_1 \longrightarrow u$) the last one is an out-edge ($u \longrightarrow v_k$), and we can find $v_i \longrightarrow u \longrightarrow v_{i+1}$ for some index i, since there must be a direction change at least once in the middle (see Fig. 5). Thus, the path can be extended to $v_1 \longrightarrow v_2 \longrightarrow \ldots \longrightarrow v_i \longrightarrow u \longrightarrow v_{i+1} \longrightarrow \ldots \longrightarrow v_k$. One possible implementation of this "repeated insertion" method goes as follows:

This is essentially the Insertion Sort algorithm if we treat $\mathrm{dir}(i, j)$ as the comparator function. It performs $N(N-1)/2$ comparisons in the worst case and solves subtask iii, but not iv. However, the number of comparisons can be reduced by using binary search for finding the position to insert: we maintain a range (v_l, v_r) within the path, where $v_l \longrightarrow u$ and $u \longrightarrow v_r$ (initially $l = 1$ and $r = k$), and it can be halved by checking the middle element. This method, which is analogous to Binary Insertion Sorting, is the algorithm described in (Hell & Rosenfeld, 1983), and it is easy to see that we check $O(N \log N)$ edges of the graph. On the other hand, for an efficient implementation, a data structure is needed that supports fast insertion and retrieval at any position, which is not available in the C++ standard library. Fortunately, we can come up with other solutions.

Viewing the task as a sorting problem brings new ideas. Let us examine Quicksort, for instance. Interestingly, it works without modification, as demonstrated in Fig. 6. After partitioning, there is $v_i \longrightarrow u$ for every v_i vertex on the left side of the pivot point u, and $u \longrightarrow w_i$ for every w_i vertex on the right. If we have a $v_1 \longrightarrow v_2 \longrightarrow \ldots \longrightarrow v_l$ path on the left side and a $w_1 \longrightarrow w_2 \longrightarrow \ldots \longrightarrow w_r$ path on the right, there will be a $v_1 \longrightarrow \ldots \longrightarrow v_l \longrightarrow u \longrightarrow w_1 \longrightarrow \ldots \longrightarrow w_r$ path containing all points. Since Quicksort does $O(N \log N)$ comparisons on average, it could also pass the big tests if the grader was not adaptive, but the $O(N^2)$ worst-case complexity makes it unsuitable with an adaptive grader.

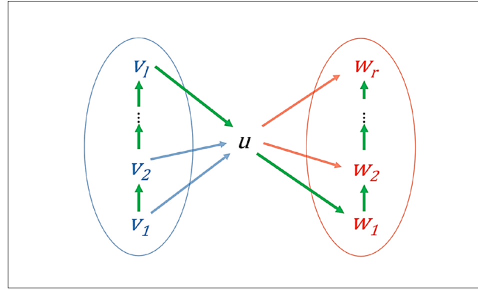


Fig. 6. The mechanism of Quicksort in the Tournament problem.

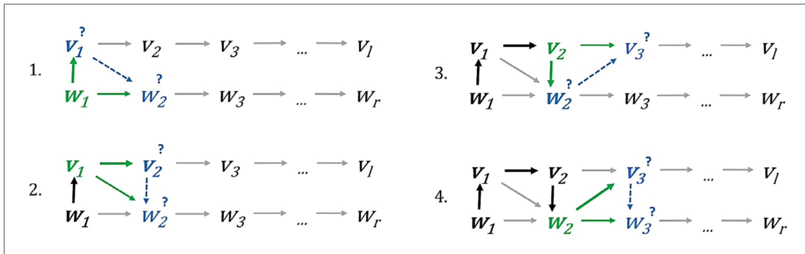


Fig. 7. The first few steps of merging two paths in the Tournament problem.

To decide whether Merge Sort is applicable, we should inspect the merging process of two paths. Fig. 7 shows that at any point when we add a vertex to the merged path, there are out-edges to both possible following points (the next element of the same sequence and the first element of the other). It follows that Merge Sort also solves the problem without any modification. Its worst-case $O(N \log N)$ complexity, and straightforward implementation makes it the best choice for the solution. What more, knowing that the std::stable_sort() in C++ standard library uses Merge Sort, it comes down to calling that on the sequence $1, 2, \ldots, N$, supplying dir(i,j) as the comparator. We must note here that the std::stable_sort() is not applicable in the same manner, because it uses Introsort (the GCC implementatioN), which is a hybrid of Quicksort, Heapsort, and Insertion Sort, and while Quicksort and Insertion Sort both produce a solution, Heapsort does not work for this problem, unfortunately.

Analysis

Finding the Hamiltonian path in a complete directed graph can be viewed as determining a generalized sorted sequence given an almost arbitrary relation. The generalized sorted sequence means that we only check the relation between consecutive elements ($x_1 \longrightarrow x_2 \longrightarrow \ldots \longrightarrow x_N$). The relation is arbitrary in the sense that we can choose $i \longrightarrow j$ or $j \longrightarrow i$ for any (i, j) pair of elements independently, but it still has some important properties:

1. Irreflexive, since there are no loop-edges in the graph.
2. Antisymmetric, because there cannot be both $i \longrightarrow j$ and $j \longrightarrow i$.
3. Semiconnex, since for all $i \neq j$ pairs $i \longrightarrow j$ or $j \longrightarrow i$.

Loop-edges are irrelevant to this problem, so instead of the irreflexivity, we might as well assume reflexivity, and then the semiconnexity becomes connexity (where $i = j$ is also included). Looking at the proofs of the three solutions above (Insertion Sort, Quicksort, Merge Sort), one can notice that they only rely on the connexity property of the relation. This fact is fascinating, namely that the mentioned sorting algorithms produce a generalized sorted sequence if the comparison relation is connex.

Methods that involve extremum selection, such as Selection Sort and Heapsort, normally require the relation to be transitive. Still, Wu (2000) managed to modify the Heapsort algorithm to solve this problem as well.

It is worth mentioning that the reasonings we gave above about the correctness of solutions can be made into precise mathematical proofs by complete induction. All of them would start with the assumption that every tournament graph of less than N vertices contains a Hamiltonian path. When we take a graph with N vertices, we divide it into smaller parts (maybe extracting just one vertex); by our assumption, there exist Hamiltonian paths within the parts, and we can transform them into one path. It is intriguing to see that sorting algorithms produce mathematical proofs in graph theory.

Further questions

A related more difficult problem was introduced in the 2016 Polish Olympiad in Informatics with the title Turysta (Szkopuł, 2016) that goes as follows. Given a complete directed graph, construct the longest path starting from each vertex. One can see that the strongly connected components form a complete directed acyclic graph and thus have a Hamiltonian path. Within a strongly connected component, there is a Hamiltonian cycle (Camion, 1959) that can be constructed similarly to the Hamiltonian path in a tournament. Thereafter, giving the longest path from any vertex is straightforward.

3.4. *Polyline with Acute Angles*

This beautiful geometry problem was featured in Codeforces Round #698 with the title Nezzar and Nice Beatmap (Codeforces, 2021a). It is also contained in the Timus Online Judge, called Mammoth Hunt (Ipatov, 2007), which states that it was introduced in the Ural State University Junior Contest, 2007. A paper by Fekete and Woeginger (1997) discusses the question more generally and includes one of the presented solutions.

Statement (shortened)

There are N points with integer coordinates given in the Cartesian plane. Connect all the points with a polyline that has only acute angles. More precisely, if we take any three consecutive points A, B, and C, the $\measuredangle ABC$ angle is strictly less than 90°. The 0° angle is also accepted (it is not to be confused with the 180°).

Subtasks and constraints

There are no subtasks in any of the task's two sources due to the contest format. The upper limit for the number of points is 5000, which suggests an $O(N^2)$ solution. In an IOI-style contest, it would make sense to introduce two more subtasks for small N, and there could be various other conditions for the set of points.

We propose some:

i) $N \leq 3$
ii) $N \leq 20$
iii) $x_i = 0$ or $y_i = 0$, $i = 1..N$
iv) The points form a regular N-sided polygon
v) $N \leq 5000$

Solutions

The first subtask is just about calculating the angles between three points. We use the $\measuredangle ABC$ notation for the angle between the AB and BC lines. The acuteness can be determined by taking the sign of the dot product of the BA and BC vectors: $\text{dot}(BA, BC) = (A.x - B.x) \cdot (C.x - B.x) + (A.y - B.y) \cdot (C.y - B.y)$. If it is positive (meaning that the cosine of the enclosed angle is positive), then $\measuredangle ABC < 90°$.

The second subtask can be solved by backtracking. In subtask iii, we can visit the points of one axis by going in alternating directions and do an appropriate switch between the two axes. In subtask iv, a good idea is to always go to the opposite or next-to opposite vertex of the polygon. There are two beautiful $O(N^2)$ solutions to the general case.

In the first solution, we build the polyline one by one. The set of points are denoted by $V = \{P_1, P_2, ..., P_N\}$. We can start in an arbitrary $S \in V$ point and let us choose the most distant point T among the rest of the points, i. e. for which $|ST| \geq |SP_i|$ $(\forall P_i \in V)$. If there are multiple such points, we can choose any. It can be proven by contradiction that for any next P_i point $\measuredangle STP_i < 90°$. Indeed, as shown in Fig. 8, if $STP_i \geq 90°$ then $|SP_i| > |ST|$ because opposite the biggest angle is the longest side

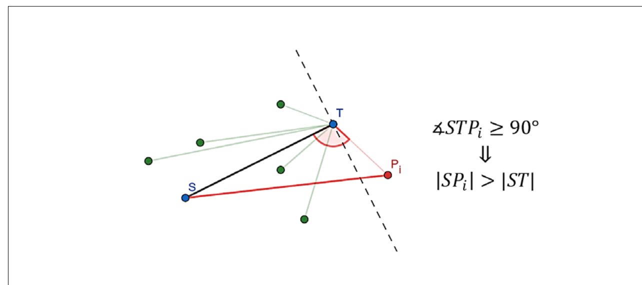


Fig. 8. The correctness of the most distant point method.

of a triangle (also true for degenerate triangles), and it is impossible since we chose T as the furthest point.

We can repeat this step, now take all the remaining points and choose the next point Q so that $|TQ|$ is maximal, i. e. $|TQ| \geq |TP_i|$ $(\forall P_i \in V \setminus \{ST\})$. The acuteness of $\measuredangle STQ$ is ensured by the choice of T as described above, and the condition for the Q point guarantees that the next angle is less than 90°, too, by the same reasoning. We keep doing the same until there are no points left. So, the algorithm goes as follows: see Fig. 9.

The complexity is clearly $O(N^2)$ if the maximum selection is done with a simple loop over all the remaining points. It is an interesting question whether it could be sped up using some data structure; we do not discuss it here.

The second approach is based on the following observation: any three points form a triangle (possible degenerate) where at most one of the three angles is not acute. This way, if there are three points A, B, C in the sequence such that $\measuredangle ABC \geq 90°$, changing their order to A, C, B or C, A, B will surely fix their angle. Let us inspect this algorithm: see Fig. 10.

For correctness, we need to prove that the three new angles introduced by the insertion of a P point are acute. See Fig. 11. (If one of the angles does not exist, the corresponding part can be skipped.) Assume that P is inserted between A_i and A_{i+1} in the acute polyline $A_1, A_2, \ldots A_k$. Then $\measuredangle A_{i-1}A_iP < 90°$, because otherwise A_i and P would have been swapped. The other two angles, $\measuredangle A_iPA_{i+1}$ and $\measuredangle PA_{i+1}A_{i+2}$ must be acute because we observed another obtuse/right angle of the same triangle when P was swapped with one of the points previously. The complexity of this method is $O(N^2)$ as well, because when inserting a new point to a polyline of length k, we perform at most k angle calculations and swaps.

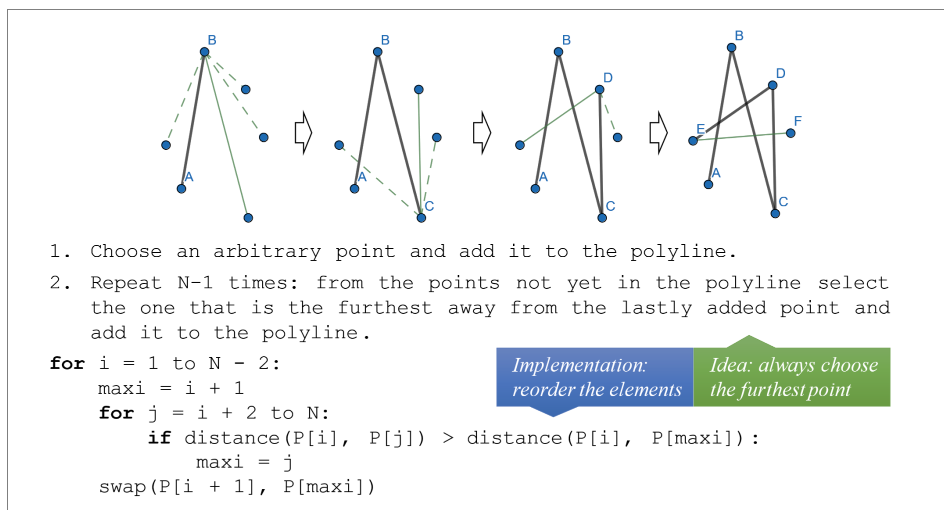


Fig. 9. The algorithm of repeatedly choosing the most distant point.

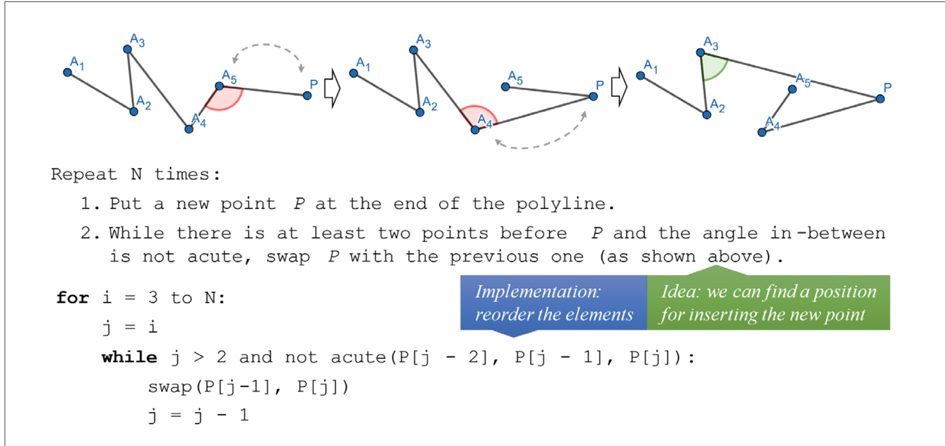


Fig. 10. The algorithm of the insertion method.

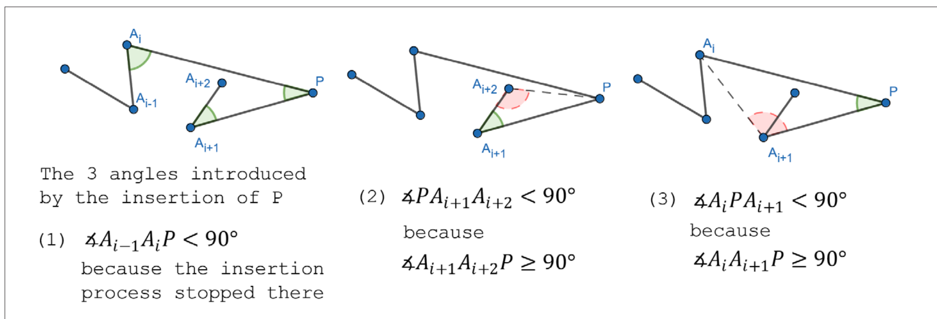


Fig. 11. Correctness of the insertion method.

Analysis

The task can be viewed as an unusual sorting problem where every three consecutive elements must satisfy a relation, let it be denoted by $acute(A, B, C)$. The two general solutions presented above strongly resemble the Selection Sort and the Insertion Sort algorithms. The connection is especially clear in the case of the insertion method, seeing that we get a correct implementation if we simply replace the comparison in the Insertion Sort, as pointed out by the editorial in (Codeforces, 2021b). The method of repeatedly selecting the most distant point can be seen as a Selection Sort. Here the maximum is not something global; it is relative to another element. Namely, if we have a point S, we are looking for a suitable next point T, for which $acute(S, T, P_i)\ \forall P_i \in V$. This is exactly what we proved when T is chosen as the furthest point from S.

Not all sorting algorithms can be used to solve the problem because of the strange nature of the $acute$ relation. The following property of the acute relation enables using Insertion Sort: if not $acute(A, B, C)$ then $acute(A, C, B)$ and $acute(C, A, B)$. This

can be interpreted as the connexity property of this relation, and in the analysis of the previous problem (Tournament), we saw that Insertion Sort works for connex binary relations. Although the same is true for Quicksort and Merge Sort, they are unfortunately not applicable because the behavior of ternary relations differs significantly from binary ones.

Further questions

It remains a challenging open question whether there is an algorithm with better than $O(N^2)$ complexity to solve this problem. (Fekete & Woeginger, 1997) also poses this question, with no answer. We have the impression that there is a faster solution; however, we have not managed to find such. The task can be formulated in space or even more than 3 dimensions. The angle defined by 3 points in a d-dimensional space is calculated from the coordinates similarly as shown above. In fact, the exact solutions can be applied to solve the problem in multiple dimensions.

3.5. *Binary Manipulations*

This problem was introduced in the 4$^{\text{th}}$ stage of the Ukrainian Olympiad in Informatics in 2019.

Statement (shortened)

We have a list of N numbers (N is a power of 2), the ith element of this list is t_i. We want to sort this sequence with a limited amount of $move$ operations. In one $move$ operation, we can move a number from the position 2^x to the beginning of the list ($2^x \leq N$).

Subtasks and constraints

In the original contest, N was at most 128, and at most 16384 $move$ operations were allowed. There were also six subtasks:

i) $N = 2$, all numbers are different
ii) $N = 4$, all numbers are different
iii) $N = 8$, all numbers are different
iv) The list consists of $N/2$ zeros and $N/2$ ones in arbitrary order
v) All numbers are different
vi) No further constraints

Solutions

In the following, $move(i)$ denotes the move operation performed from the position i (where i is always a power of 2). The first subtask can be solved by a simple if statement. Subtasks ii–iii can be solved by breadth-first search as we have at most $8! = 40320$ different lists, and there are 3 transitions from each list. Subtask iv suggests a divide and

conquer approach. Thus, we will assume that we can solve the problem for $K = 2^x$ and try to solve it for $2K = 2^{x+1}$.

First, let us sort the first K numbers. Then, if we perform K times a $move(2K)$ operation, we effectively moved the second part to the beginning, so we can sort that part, too. Now we have two sorted sequences which we want to merge into one. Let us indicate if a number should be in the first/second half of the sorted list by giving it a zero/one label. Obviously, after sorting the two halves, both the first and the second half starts with some number of zeros followed by some number of ones. Then we perform $move(K)$ until the ones in the first half form a prefix of the list. Afterwards, we perform $move(2K)$ operations while the last element is a one, which leads us to all ones in the first half and all zeros in the second. Now we have all the numbers in the wrong half, and they might be in the wrong order too. The first issue is easily fixed, and their order can be fixed, too, by sorting them again. So, in the last steps of the algorithm for sorting the $2K$ numbers, we sort the first K numbers, then perform a $move(2K)$ operation K times and sort the first K numbers again. The overall number of move operations in this algorithm is $T(N) = O(N^2)$ according to the Master theorem (Cormen *et al.*, 2009), since we have the recurrence $T(N) = 4T(N/2) + O(N)$. Overall, the biggest steps of the algorithm are the following: see Fig. 12.

In the middle of the algorithm, we have assigned zeros and ones to numbers based on which half a number should belong, but in the case when not all numbers are different, this is not well defined. To solve this issue, we can replace t_i by $t_i \cdot n + i$, so all numbers will be different while preserving their relative order.

Though we have described a solution that would fit in the limits of the problem, it is not asymptotically optimal as there is a method with $O(N \log N)$ $move$ operations on average. If the input is sufficiently random, then there is an $O(N)$ $move$ operation algorithm to partition along the median element at each recursive step. It is the fol-

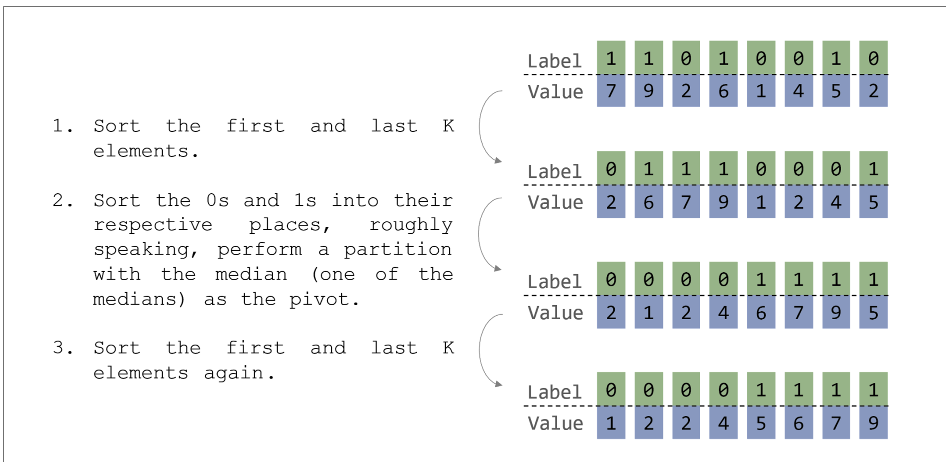


Fig. 12. Overview of the solution of the Binary Manipulations problem.

lowing algorithm: while there is a one among the first K elements, if the Kth element is a one, perform a $move(2K)$, otherwise perform a $move(K)$. This method splits the zeros and ones into their correct half. This partitioning method has a worst-case complexity of $O(N^2)$ though if the input is not random enough. Note that it always terminates since, after each operation, the number of ones does not decrease in the second half.

Analysis

Both solutions presented here are divide and conquer algorithms, resembling Merge Sort and Quicksort. The first one has the properties of Merge Sort, as in each step we sort the halves recursively, but it is slower than usual since the merging step cannot be done without interfering with the already sorted subparts. This means that two more recursive calls are needed. It is quite a natural way to approach this problem as the powers of two suggest a divide and conquer approach, and the most well-known such sorting algorithm is Merge Sort.

The second described solution can be viewed as a Quicksort with a fixed pivot as the current subarray's median. This also arises naturally from knowing Quicksort as we do not have any effective way to select a random pivot due to the limited range of operations. The method used to make all numbers different is also an excellent lesson about the stability of sorting algorithms, as we have indirectly made our sorting algorithms stable with it, as well as making them work for different numbers.

Lastly, we measured the average number of move operations performed by the two algorithms presented. The Table 1 shows the average number of performed move operations (rounded to 5 digits) for different values of N, running the algorithms on 100 uniformly randomly generated tests. The results demonstrate clearly that the Merge Sort-like solution (MS) is expected to do much more move operations than the Quicksort-like approach (QS) on random inputs.

Further questions

Similar questions have been examined throughout the literature, for example, considering the "reverse" of the current $move$ operation, i. e. moving from the beginning to an arbitrary position (Aigner & West, 1987).

An easy follow-up question to this task might be: what if N^{th} is not a power of 2? Let us generalize a bit; we define some set S which contains the positions from which

Table 1

Average number of move operations done by the two solutions

N	16	32	64	128	256	512	1024
MS	169.94	682.59	2640.6	10121	38480	147790	571880
QS	95.28	283.79	781.58	2079.5	5270.2	13108	32031

a valid $move$ operation could be made. This means that in the original problem $S = \{2^i \mid 1 \le 2^i \le N\}$, whereas now we are trying to sort with $S = \{2^i \mid 1 \le 2^i \le N\} \cup \{N\}$. Questions might arise about what conditions are necessary and sufficient to sort with an arbitrary set S. The minimality can also be further explored as in regular swap-based sorting (Humphreys & Liu, 1996). For example, if $S = \{1..N\}$, we have a similar solution as in (Aigner & West, 1987). Designing an algorithm that works well for given S and input sequence might be interesting as well.

3.6. *Inversion Count*

We present an elementary problem, namely counting the inversions in a sequence, and its well-known solution using Merge Sort, also described in (Halim *et al.*, 2013, p. 355). We include it because it is a fundamental example of our topic.

Statement (shortened)

A pair of indices (i, j) is called an inversion of an array A if $i < j$ and $A[i] > A[j]$. Given an array of N integers, you should count the total number of inversions in it. For example, the array $[3, 1, 4, 1, 5, 9, 2]$ has 7 inversions: $(1, 2)$, $(1, 4)$, $(3, 4)$, $(1, 7)$, $(3, 7)$, $(5, 7)$, and $(6, 7)$.

Subtasks and constraints

i) $N \le 10^3$
ii) $N \le 10^5$

Solutions

A naive solution works for the first subtask: we simply iterate over all pairs (i, j) (where $i < j$) and increment the total count if $A[i] > A[j]$. The time complexity is $O(N^2)$ thus infeasible for the second subtask.

To solve the second subtask, we will use a divide and conquer approach based on Merge Sort; namely, with a slight modification of the merging step, one can obtain an answer in $O(N \log N)$ time.

Suppose we have to find the number of inversions in some subarray $[l; r]$ while also sorting it in non-descending order. The idea is to define a function $\text{count}(l, r)$ that solves the problem recursively. If $l = r$, then the answer is trivially 0, and the subarray is already sorted. In the following, we assume that $l < r$ holds. Let us denote the middle index as mid (formally, $mid = \lfloor (l + r)/2 \rfloor$). We can divide the problem into the following three subproblems:

1. The number of (i, j) inversions where $l \le i, j \le mid$.
2. The number of (i, j) inversions where $mid < i, j \le r$.
3. The number of (i, j) inversions where $l \le i \le mid$ and $mid < j \le r$.

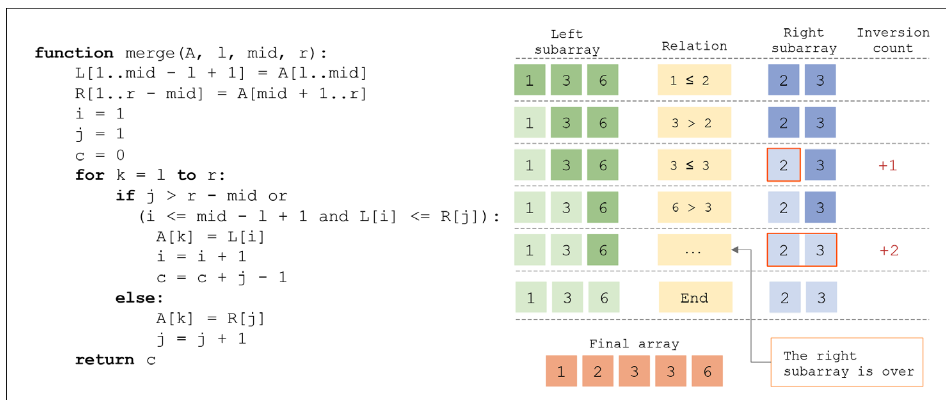


Fig. 13. Pseudocode and example of the modified merging step.

Subproblems 1 and 2 can be solved recursively by calling count(l, mid) and count$(mid + 1, r)$. Given that subarrays $[l; mid]$ and $[mid + 1; r]$ are sorted, the number of inversions of the 3rd kind can also be computed by applying a small change to the merging step of Merge Sort. While merging the two subarrays, we sum up the number of inversions using the contribution technique (Debowski, 2018). In other words, for all $l \leq i \leq mid$, we calculate the "contribution" of i via finding the amount of $mid < j \leq r$ indices such that (i, j) forms an inversion.

It is also worth noting that the total number of inversions is the same as the number of Bubble Sort-like swaps (consecutive swaps) needed to sort the array in non-descending order. This follows from the very fact that Bubble Sort gets rid of the inversions one by one (each consecutive swap accounts for exactly one inversion). So, another $O(N^2)$ solution would be to count the swaps during a Bubble Sort.

As a second note, we need to mention that there are different approaches to solve this problem optimally, many involving the use of some special data structure (e.g., order statistic tree, binary indexed tree, or segment tree).

Further questions

A natural question is whether we can use other asymptotically optimal sorting algorithms (e.g., quicksort, heap sort) to count the number of inversions. The answer is yes. For example, there are some less common, stable versions of Quicksort suitable for this problem.

4. Conclusions

We demonstrated various problems where the solutions are connected to sorting algorithms, in most cases unexpectedly so. In these tasks, the sort() function of the standard library cannot be used, the implementation of a particular algorithm is required. (There

was one exception, the std::stable_sort() from the C++ standard library can be used to solve the Tournament task – but knowing the theoretical background is necessary even in this case.) All the presented tasks were involved in some competition for high school students, and while the frequency of such questions might be very low, we can still conclude that students need to know various comparison-based sorting algorithms if they want to be successful in national and international competitions. Hence, the problems above are beneficial for educational purposes.

In the Matching Nuts and Bolts task, the solution is a slightly modified Quicksort, while Bubble Sort could also work as a slower solution. The Median Strength problem has connections to many sorting algorithms: Selection Sort, Insertion Sort, Quicksort, and Heapsort. The Tournament problem is an example where Insertion Sort, Quicksort, and Merge Sort can be applied without modification. The Polyline with Acute Angles has a very surprising solution using the Insertion Sort with a ternary relation, while a connection to Selection Sort can also be made. The Binary Manipulations task is a neat restricted sorting problem where the knowledge of Quicksort and Merge Sort comes handy. Inversion Count has a well-known enlightening solution using Merge Sort along with a solid connection to Bubble Sort.

The Tournament problem and the Polyline with Acute Angles are examples where sorting algorithms are helpful to prove mathematical theorems. This reminds us of elegant proofs in which theories of another field of mathematics are applied, like Euler's proof of the existence of infinite primes using infinite series, or the fundamental theorem of algebra proved using complex analysis or even geometry, the probabilistic method used in combinatorics, and statements in Euclidean geometry proved by linear algebra or projective geometry, just to mention a couple of them.

By taking a close look at these few problems, we can observe a pattern of how Quicksort, Merge Sort, and Insertion Sort (also with binary search) can be helpful in many different scenarios, and Heapsort, Selection Sort and Bubble Sort could also be involved in some cases. Moreover, although our aim was to point out some problems where they appear directly, we believe that the biggest gain of learning these sorting algorithms is the capability of applying their principles in other situations.

It would be worthwhile to examine the question similarly for other elements of the standard library. For instance, the priority queue (heap), lower and upper bound (binary search), set and map (dynamically balanced binary search trees), unordered set and map (hash tables), among others. We think that the answer is a clear yes for binary search since it is also a basic optimization strategy, much less clear for dynamically balanced binary trees, considering that we have not seen a need to implement them, although there is a curious application of the C++ set with a custom comparator to find a pair of intersecting segments (CP-Algorithms, 2018), where the internals of the data structure need to be known to a certain extent. The question is definitely interesting for the heap and hash tables; if there were similar problems, they would be profitable for education.

References

Aigner, M., West, D.B. (1987). Sorting by insertion of leading elements. *Journal of Combinatorial Theory, Series A*, 45(2), 306–309.

Alon, N., Blum, M., Fiat, A., Kannan, S., Naor, M., Ostrovsky, R. (1994). Matching nuts and bolts. SODA,

Blum, M., Floyd, R.W., Pratt, V.R., Rivest, R.L., Tarjan, R.E. (1973). Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4), 448–461.

Bradford, P.G. (1995). Matching nuts and bolts optimally.

Camion, P. (1959). Chemins et circuits hamiltoniens des graphes complets. *Comptes Rendus de l'Académie des Sciences de Paris*, 249, 2151–2152.

Codeforces. (2021a). *1477C. Nezzar and Nice Beatmap*. Retrieved March 31 from `https://codeforces.com/contest/1477/problem/C`

Codeforces. (2021b). *Editorial of Codeforces Round #698*. Retrieved March 31 from `https://codeforces.com/blog/entry/87294`

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2009). *Introduction to algorithms*. MIT press.

CP-Algorithms. (2018). *Search for a pair of intersecting segments*. Retrieved March 31 from `https://cp-algorithms.com/geometry/intersecting_segments.html`

Debowski, K. (2018). *Sums and Expected Value*. Codeforces Blog. Retrieved March 31 from `https://codeforces.com/blog/entry/62690`

Fekete, S.P., Woeginger, G.J. (1997). Angle-restricted tours in the plane. *Computational Geometry*, 8(4), 195–218.

Google. (2021). *Median Sort – Analysis*. Google Code Jam. Retrieved March 31 from `https://codingcompetitions.withgoogle.com/codejam/round/000000000043580a/00000000006d1284#analysis`

Halim, S., Halim, F., Skiena, S.S., Revilla, M.A. (2013). *Competitive programming 3*. Citeseer.

Hell, P., Rosenfeld, M. (1983). The complexity of finding generalized paths in tournaments. *Journal of Algorithms*, 4(4), 303–309.

Hoare, C.A.R. (1961a). Algorithm 64: quicksort. *Communications of the ACM*, *4*(7), 321.

Hoare, C.A.R. (1961b). Algorithm 65: find. *Communications of the ACM*, *4*(7), 321–322.

Horváth, G., & Verhoeff, T. (2002). Finding the median under IOI conditions. *Informatics in Education*, 1(1), 73–92.

Humphreys, J.F., Liu, Q. (1996). *A course in group theory* (Vol. 6). Oxford University Press on Demand.

Ipatov, A. (2007). *1578. Mammoth Hunt*. Timus Online Judge. Retrieved March 31 from `https://acm.timus.ru/problem.aspx?space=1&num=1578`

Knuth, D. (1997a). Sorting by Exchanging. In *The Art of Computer Programming* (3rd ed., Vol. 3: Sorting and Searching, pp. 106–110). Addison–Wesley.

Knuth, D. (1997b). Sorting by Insertion. In *The Art of Computer Programming* (3rd ed., Vol. 3: Sorting and Searching, pp. 80–105). Addison–Wesley.

Knuth, D. (1997c). Sorting by Merging. In *The Art of Computer Programming* (3rd ed., Vol. 3: Sorting and Searching, pp. 158–168). Addison–Wesley.

Knuth, D. (1997d). Sorting by Selection. In *The Art of Computer Programming* (3rd ed., Vol. 3: Sorting and Searching, pp. 138–141). Addison–Wesley.

Komlós, J., Ma, Y., Szemerédi, E. (1998). Matching nuts and bolts in O (n log N) time. *SIAM Journal on Discrete Mathematics*, 11(3), 347–372.

Rawlins, G.J. (1992). *Compared to what?: an introduction to the analysis of algorithms*. Computer Science Press New York.

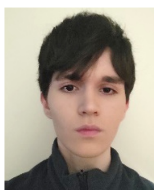
Rédei, L. (1934). Ein kombinatorischer Satz. *Acta Litteraria Szeged*, 7, 39–43.

Szkopuł. (2016). *Turysta*. Polish Olympiad in Informatics. Retrieved March 31 from `https://szkopul.edu.pl/problemset/problem/kqBM3UKWL-qlFiXIOxPXL35m/site/?key=statement`

Wu, J. (2000). On finding a hamiltonian path in a tournament using semi-heap. *Parallel Processing Letters*, 10(04), 279–294.

L. Nikházy is a Ph.D. student at Department of Media & Educational Informatics, Faculty of Informatics, Eötvös Loránd University in Hungary. His current research interest is computer programming talent education. He started his career as a software engineer at Google, but he gradually shifted to computer science education, and now devotes all his time to teaching talented children. He has been actively involved in organizing programming competitions and team coaching in Hungary in the past 4 years.

Á. Noszály is a Computer Science BSc student at the Faculty of Informatics, Eötvös Loránd University in Hungary. He has participated in several international programming competitions including the IOI. His current interests are in mathematics, problem setting and grading system development.



B. Deák is a first-year BSc student of Computer Science at the Faculty of Informatics, Eötvös Loránd University. He has been participating regularly in mathematics and informatics competitions since his early high school years, and now takes part in preparing several national programming contests in Hungary. He is interested in most subfields of theoretical computer science and discrete mathematics.