Chapter 1

Lower Bounds

By Sariel Har-Peled, February 25, 2021[1] Version: 1.0

1.1. Sorting

We all know that sorting can be done in $O(n \log n)$ time. Interestingly enough, one can show that one needs $\Omega(n \log n)$ time to solve this.

Rules of engagement. We need to define exactly what the sorting algorithm can do, or can not do. In the comparison model, we allow the sorting algorithm to do only one operation: it compare two elements. To this end, we provide the sorting algorithm a black box compare(i, j) that compares the ith element in the input to the jth element.

Problem statement. Our purpose is to solve the following problem.

Problem 1.1.1. Consider an input of n distinct elements, with an ordering defining over them. In the worst case, how many calls to the comparison subroutine (i.e., compare) a deterministic sorting algorithm have to perform?

1.1.1. Decision trees

Well, we can think about a sorting algorithm as a decision procedure, at each stage, it has the current collection of comparisons it already resolved, and it need to decide which comparison to perform next. We can describe this as a decision tree, see Figure 1.1. The algorithm starts at the root.

But what is a sorting algorithm? The output of a sorting algorithm is the input elements – in a certain order. That is, a sorting algorithm for n elements outputs a permutation π of $[\![n]\!] = \{1, \ldots, n\}$. Formally, if the input is $x_1, \ldots, x_n$ the output is a permutation π of $[\![n]\!]$, such that $x_{\pi(1)} < x_{\pi(2)} < \ldots < x_{\pi(n)}$.

Initially all $n!$ permutations are possible, but as the algorithm performs comparisons, and as the algorithm descend in the tree it rules out some of these orderings as not being feasible. For example, the root r of the decision tree of Figure 1.1 have all possible 6 permutations as a possible output; that is,

$$\Phi(r) = \Big\{(1, 2, 3),\ (1, 3, 2),\ (2, 1, 3),\ (2, 3, 1),\ (3, 1, 2),\ (3, 2, 1)\Big\}.$$

But after the comparison in the root is performed, the algorithm is informed that $x_1 < x_2$. Then the algorithm descends into the node u, and the possible ordering of the output that are still valid (in light

[1] This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/3.0/` or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

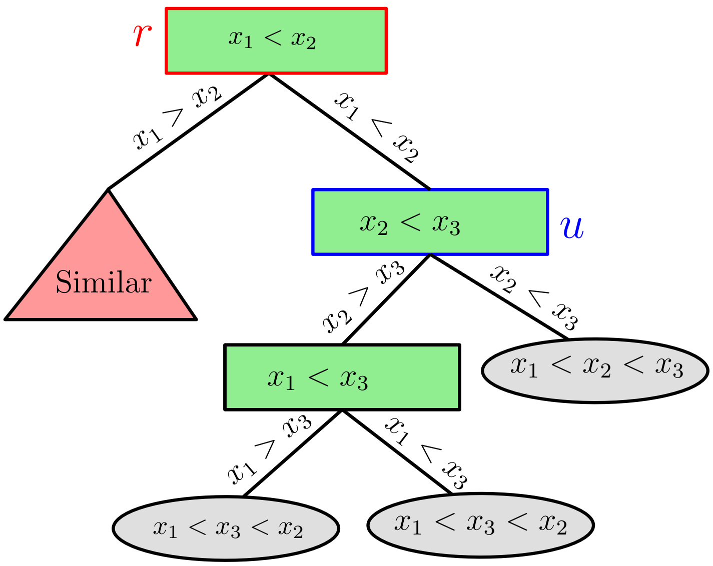


Figure 1.1: A decision tree for sorting three elements.

of the comparison the algorithm performed), is

$$\Phi(u) = \left\{(1,2,3),\ (1,3,2),\ (3,1,2)\right\}.$$

In particular, for a node v of the decision tree, let $\Phi(v)$ be the set of ***feasible permutations***; that is, it is the set of all permutations that are compatible with the set of comparisons that were performed from the root to v.

Example 1.1.2. Assume the input is x_1, x_2, x_3, x_4. If the permutation $\{(3,4,1,2)\}$ is in $\Phi(v)$ then as far as the comparisons the algorithm performed in traveling from the root to v, it might be that this specific ordering of the input is a valid ordering. That is, it might be that $x_3 < x_4 < x_1 < x_2$.

Lemma 1.1.3. *Given a permutation π of $[\![n]\!]$, an input that is sorted in the ordering specified by π is the following: $x_i = \pi^{-1}(i)$, for $i = 1, \ldots, n$.*

Proof: We construct an input which is sorted according to the permutation π. To this end, the input is the numbers of $[\![n]\!]$. Now, clearly, $x_{\pi(1)}$ must be the smallest number, that is 1, and as such, set $x_{\pi(1)} = 1$. Applying this argument repeatedly, we set $x_{\pi(i)} = i$, for all i. In particular, take $j = \pi^{-1}(i)$, and observe that

$$x_i = x_{\pi(\pi^{-1}(i))} = x_{\pi(j)} = j = \pi^{-1}(i),$$

as claimed. ■

Example 1.1.4. A convenient way to do the above transformation is the following. Write the permutation as a function $[\![n]\!]$ by writing it as matrix with two rows, the top row having $1, \ldots, n$, and the second row having the real permutation. Computing the inverse permutation is then no more than exchanging the two lines, and sorting the columns. For example, for $\pi = (3,4,2,1) = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 2 & 1 \end{pmatrix}$. Then the input realizing this permutation, is the input $\pi^{-1} = (3,4,1,2) = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \end{pmatrix}$. Specifically, the input $x_1 = 4$, $x_2 = 3$, $x_3 = 1$, and $x_4 = 2$.

Observation 1.1.5. *Assume the algorithm had arrived to a node v in the decision tree, where $|\Phi(v)| > 1$. Namely, there are more than one permutation of the input that comply with the comparisons performed so far by the algorithm. Then, the algorithm must continue performing comparisons (otherwise, it would not know what to output – there are still at least two possible outputs).*

Lemma 1.1.6. *Any deterministic sorting algorithm in the comparisons model, must perform $\Omega(n \log n)$ comparisons.*

Proof: An algorithm in the comparison model is a decision tree. Indeed, an execution of the sorting algorithm on a specific input is a path in this tree. Imagine running the algorithm on all possible inputs, and generating this decision tree.

Now, the idea is to use an adversary argument, which would pick the worse possible input for the given algorithm. Importantly, the adversary need to show the input it used only in the end of the execution of the algorithm – that is, it can change the input of the algorithm on the fly, as long as it does not change the answer to the comparisons already seen so far.

So, let $\mathcal{T}$ be the decision tree associated with the algorithm, and observe that $|\Phi(r)| = n!$, where $r = \mathrm{root}(\mathcal{T})$.

The adversary, at the beginning, has no commitment on which of the permutations of $\Phi(r)$ it is using for the input. Specifically, the adversary computes the sets $\Phi(u)$, for all the nodes $u \in \mathsf{V}(\mathcal{T})$.

Imagine, that the algorithm performed k comparisons, and it is currently at a node v_t of the decision tree. The algorithm call compare to perform the comparison of x_i to x_j associated with v_k. The adversary can now decide what of the two possible results this comparison returns. Let u_L, u_R be the two children of v_t, where u_L (resp. U_R) is the child if the result of the comparison is $x_i > x_j$ (resp. $x_i < x_j$).

The adversary computes $\Phi(U_L)$ and $\Phi(U_R)$. There are two cases:

(I) If $|\Phi(U_L)| < |\Phi(U_R)|$, the adversary prefers the algorithm to continue into U_R, and as such it returns the result of comparison of x_i and x_j as $x_i < x_j$.
(II) If $|\Phi(U_L)| \geq |\Phi(U_R)|$, the adversary returns the comparison results $x_i > x_j$.

The adversary continues the traversal down the tree in this fashion, always picking the child that has more permutations associated with it. Let $v_1, \ldots, v_k$ be the path taken by the algorithm. The input the adversary pick, is the input realizing the single permutation of $\Phi(v_k)$.

Note, that $1 = \left|\Phi(v_k)\right| \geq \dfrac{|\Phi(v_{k-1})|}{2} \geq \ldots \geq \dfrac{|\Phi(v_1)|}{2^{k-1}}$. Thus, $2^{k-1} \geq |\Phi(v_1)| = n!$. Implying that $k \geq \lg(n!) + 1 = \Omega(n \log n)$. We conclude that the depth of $\mathcal{T}$ is $\Omega(n \log n)$. Namely, there is an input which forces the given sorting algorithm to perform $\Omega(n \log n)$ comparisons. ■

1.1.2. An easier direct argument

Proof: (Proof of Lemma 1.1.6.) Consider the set Π of all permutations of $[\![n]\!]$ (each can be interpreted as a sequence of the n numbers $1, \ldots, n$). We treat an element $(x_1, \ldots, x_n) \in \Pi$ as an input to the algorithm. Next, stream the inputs one by one through the decision tree. Each such input ends up in a leaf of the decision tree. Note, that no leaf can have two different inputs that arrive to it – indeed, if this happened, then the sorting algorithm would have failed to sort correctly one of the two inputs.

Now, the decision tree is a binary tree, it has at least $n!$ leafs, and as such, if h is the maximum depth of a node in the decision tree, we must have that $2^h \geq n!$. That is, $h \geq \lg n! = \Omega(n \log n)$, as desired. ■

The reader might wonder why we bothered to show the original proof of Lemma 1.1.6. First, the second proof is simpler because the reader is already familiar with the language of decision trees.

Secondly, the original proof brings to the forefront the idea of computation as a game against an adversary – this is a rather powerful and useful idea.

1.2. Uniqueness

Problem 1.2.1 (Uniqueness). Given an input of n real numbers $x_1, \ldots, x_n$. Decide if all the numbers are unique (i.e., different).

Intuitively, this seems significantly easier than sorting. In particular, one can solve this in expected linear time. Nevertheless, this problem is as hard as sorting.

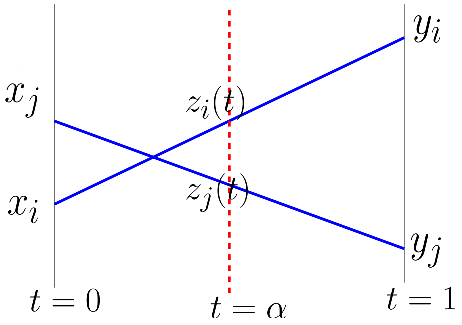
Theorem 1.2.2. *Any deterministic algorithm in the comparison model that solves Uniqueness, has $\Omega(n \log n)$ running time in the worst case.*

Note, that the linear time algorithm mentioned above is in a different computation model (allowing floor function, randomization, etc). The proof of the above theorem is similar to the sorting case, but it is trickier. As before, let $\mathcal{T}$ be the decision tree (note that every node has three children).

Lemma 1.2.3. *For a node v in the decision tree $\mathcal{T}$ for the given deterministic algorithm solving Uniqueness, if the set $\Phi(v)$ contains more than one permutation, then there exists two inputs which arrive to v, where one is unique and other is not.*

Proof: Let σ and σ' be any two different permutations in $\Phi(v)$, and let $X = x_1, \ldots, x_n$ be an input realizing σ, and let $Y = y_1, \ldots, y_n$ be an input realizing σ'. Let $Z(t) = z_1(t), \ldots, z_n(t)$ be an input where $z_i(t) = tx_i + (1-t)y_i$. Clearly, $Z(0) = x_1, \ldots, x_n$ and $Z(1) = y_1, \ldots, y_n$.

We claim that for any $t \in [0,1]$ the input $Z(t)$ will arrive to the node v in $\mathcal{T}$.



Indeed, assume for the sake contradiction that this is false, and assume that for $t = \alpha$, that algorithm did not arrive to v in $\mathcal{T}$. Assume that the algorithm compared the ith element of the input to the jth element in the input, when it decided to take a different path in $\mathcal{T}$ than the one taken for X and Y. The claim is that then $x_i < x_j$ and $y_i > y_j$ or $x_i > x_j$ and $y_i < y_j$. Namely, in such a case either X or Y will not arrive to v in $\mathcal{T}$.

To this end, consider the functions $z_i(t)$ and $z_j(t)$, depicted on the right. The ordering between the $z_i(t)$ and $z_j(t)$ is either the ordering between x_i and x_j or the ordering between y_i and y_j. As such, if is $Z(t)$ followed a different path than X in $\mathcal{T}$, then Y would never arrive to v. A contradiction.

Thus, all the inputs $Z(t)$, for all $t \in [0,1]$ arrive to the same node v.

Now, X and Y are both made of unique numbers and have a different ordering when sorted. In particular, there must be two indices, say f and g, such that, either:

(i) $x_f < x_g$ and $y_f > y_g$, or
(ii) $x_f > x_g$ and $y_f < y_g$.

Indeed, if there where no such indices f and g, then X and Y would have the same sorted ordering, which is a contradiction.

Now, arguing as in the above figure, there must be $\beta \in (0,1)$ such $z_f(\beta) = z_g(\beta)$.

This is a contradiction. Indeed, there are two inputs $Z(0)$ and $Z(\beta)$ where one is unique and the other is not, such that they both arrive to the node v in the decision tree. The algorithm must continue performing comparisons to figure out what is the right output, and v can not be a leaf. ■

Proof: (of Theorem 1.2.2) We apply the same argument as in Lemma 1.1.6. If in the decision tree $\mathcal{T}$ for Uniqueness, the adversary arrived to a node containing more than one permutation, it continues into the child that have more permutations associated with it. As in the sorting argument it follows that there exists a path in $\mathcal{T}$ of length $\Omega(n \log n)$. ■

1.3. Other lower bounds

1.3.1. Algebraic tree model

In this model, at each node, we are allowed to compute a polynomial, and ask for its sign at a certain point (i.e., comparing x_i to x_j is equivalent to asking if the polynomial $x_i - x_j$ is positive/negative/zero).

One can prove things in this model, but it requires considerably stronger techniques.

Problem 1.3.1 (Degenerate points). Given a set P of n points in $\mathbb{R}^d$, deciding if there are $d+1$ points in P which are co-linear (all lying on a common plane).

Theorem 1.3.2 (Jeff Erickson and Raimund Seidel [ES95]). *Solving the degenerate points problem requires $\Omega(n^d)$ time in a "reasonable" model of computation.*

1.3.2. 3Sum-Hard

Problem 1.3.3 (3SUM). Given three sets of numbers A, B, C are there three numbers $a \in A$, $b \in B$ and $c \in C$, such that $a + b = c$.

We leave the following as an exercise to the interested reader.

Lemma 1.3.4. *One can solve the 3SUM problem in $O(n^2)$ time.*

Somewhat surprisingly, no better solution is known. An interesting open problem is to find a subquadratic algorithm for 3SUM. It is widely believed that no such algorithm exists. There is a large collection problems that are 3SUM-Hard: if you solve them in subquadratic time, then you can solve 3SUM in subquadratic time Those problems include:

(I) For n points in the plane, is there three points that lie on the same line.
(II) Given a set of n triangles in the plane, do they cover the unit square
(III) Given two polygons P and Q can one translate P such that it is contained inside Q?

So, how does one prove that a problem is 3SUM-Hard? One uses reductions that have subquadratic running time. The details are interesting, but are omitted. The interested reader should check out the research on the topic [GO95].

Bibliography

[ES95] J. Erickson and R. Seidel. Better lower bounds on detecting affine and spherical degeneracies. *Discrete Comput. Geom.*, 13:41–57, 1995.

[GO95] A. Gajentaan and M. H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Comput. Geom. Theory Appl.*, 5:165–185, 1995.