ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute of Theoretical Computer Science
Mohsen Ghaffari, Angelika Steger, Emo Welzl, Peter Widmayer

Solution of In-Class Exercise 1: Lines Intersecting Segment

Let s be a line segment in the plane $\mathbf{R}^2$, and let $u, v \in \mathbf{R}$ be the two endpoints that delimit it. Let furthermore $\mathcal{L}$ be the set of all non-vertical lines l that intersect s. A simple characterization of $\mathcal{L}$ would be that any non-veritcal line l is contained in $\mathcal{L}$ if and only if one of the following occurs:

(i) l contains u or

(ii) l contains v or

(iii) u lies above l and v lies below l or

(iv) v lies above l and u lies below l.

In the dual space, u^* and v^* are two fixed lines and we are looking for the set of points $\mathcal{L}^*$ corresponding to the duals of all lines in $\mathcal{L}$. According to Observation 2.3 and the preceding discussion, the above criteria for l hitting the segment s are satisfied if and only if in the dual,

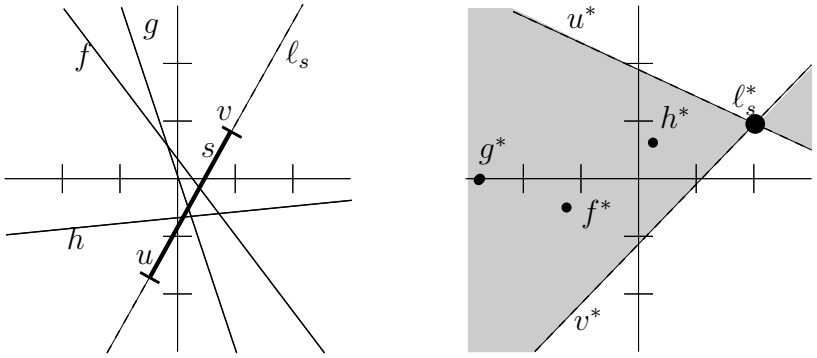
(i') l^* lies on u^* (incidence) or

(ii') l^* lies on v^* (incidence) or

(iii') l^* lies above u^* and l^* lies below v^* (2.3) or

(iv') l^* lies above v^* and l^* lies below u^* (2.3)

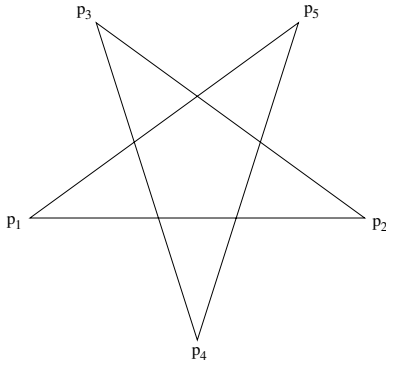
So the space $\mathcal{L}^*$ we are looking for consists of four regions defined by these criteria. The following picture shows what those regions typically look like. On the left, the primal space is shown, containing a line segment s and several lines f, g, h intersecting s in exactly one point, where on the right their duals are shown. Here ℓ_s is the line going through the endpoints u and v. The set of points $\mathcal{L}^*$ consists of the two lines u^* and v^* as well as all points in the left wedge that have u^* above and v^* below (region (iv')) and all points in the right wedge that have v^* above and u^* below (region (iii')).



Note that $\mathcal{L}^*$ always consists of two wedges, unless u^* and v^* are parallel (which is the case iff s is vertical), then it becomes a strip between two parallel lines.

Solution of In-Class Exercise 2: Locally vs. Globally Convex

No, the following picture shows a counterexample for $n = 5$. As you can easily check, any three consecutive vertices describe a triangle in counterclockwise order, but the entirety of the sequence does not describe a polygon in the order required.



Solution 1: Building a Treap

(a) We consider the event that the root of the treap changes after the insertion of key i is completed. Let X_i be the indicator variable for that event. According to how a treap works, we now have

$$X_i = 1 \;\Leftrightarrow\; \text{the node } i \text{ is rotated to the top when it is inserted,}$$

thus equivalently,

$$X_i = 1 \;\Leftrightarrow\; i \text{ receives minimum priority among the keys } \{1, \ldots, i\}$$

or equivalently

$$X_i = 1 \;\Leftrightarrow\; i \text{ appears first in the random sorting order of } \{1, \ldots, i\}.$$

From the last characterization it is evident that $\Pr[X_i = 1] = \mathbf{E}[X_i] = \frac{1}{i}$. Therefore the expected number of changes of the root of the treap is $\mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i] = \sum_{i=1}^{n} \frac{1}{i} = H_n$.

(b) For this part (and also then Part (c)) of the exercise, it is crucial to make use of the special (increasing) order of the keys in which the nodes are inserted into the treap. The fact that the key of every current node i is strictly larger than all keys already in the treap implies that temporarily —at the start of its insertion— it must always end up as the right-most leaf of the current treap. This means that after the necessary rotations for its insertion are completed, our node *must* end up somewhere on the *right spine* of the treap. Furthermore, all rotations that will ever be performed in the whole process are therefore "right-to-left"-rotations of some edge on the current right spine, in other words rotating the current node i upwards on the right spine.

Knowing what all the rotations in the process look like, we can now think about the right child of the root, and conclude the following:
Observation 1. *Node i will only ever occur as the right child of the root if it does so directly after its own insertion.*

Proof. All rotations that occur in our process are rotations of an edge of the right spine: The currently inserted node i moves one step up on the right spine, and replaces some node there which will be moved to the left, away from the spine. This also implies that the current root can never become the right child of the root anymore.

After the insertion of our node i is completed, and the later nodes are inserted, i will only potentially get rotated away from the right spine, and never upwards in the tree, which proves our claim. □

Now we want to compute the probability of the event

$$Y_i := i \text{ is the right child of root (after } i \text{ was inserted into the treap).}$$

We can compute $\Pr[Y_i]$ by the standard trick of conditioning on the root of the treap, for $1 \le i \le n$:

$$\Pr[Y_i] = \sum_{r=1}^{i} \underbrace{\Pr[Y_i \mid \text{root is } r]}_{*} \underbrace{\Pr[\text{root is } r]}_{**}$$

where

$$* = \begin{cases} \Pr[i \text{ gets minimum priority among } \{r+1, \dots, i\}] = \frac{1}{i-r} & \text{if } r < i, \\ 0 & \text{if } r = i. \end{cases}$$

$$** = \Pr[r \text{ gets minimum priority among } \{1, \dots, i\}] = \frac{1}{i}.$$

Therefore we obtain

$$\Pr[Y_i] = \frac{1}{i} \sum_{r=1}^{i-1} \frac{1}{i-r} = \frac{1}{i} \sum_{s=1}^{i-1} \frac{1}{s} = \frac{1}{i} H_{i-1}.$$

Alternative proof. If we consider the event

$$Y_i^k := i \text{ is the right child of root after } k \text{ nodes have been inserted into the treap,}$$

for $1 \leq i \leq k \leq n$, and use the standard approach of conditioning on the root, we obtain

$$\Pr\left[Y_i^k\right] = \frac{1}{k}\sum_{r=1}^{i-1}\frac{1}{k-r} = \frac{1}{k}\sum_{s=k-i+1}^{k-1}\frac{1}{s} = \frac{1}{k}(H_{k-1} - H_{k-i}),$$

where we defined $H_0 := 0$. Now we again have to use the crucial Observation 1 from above, that "i will only occur as the right child of the root if it does so directly after its own insertion". Formally, this means that the $\bigvee_{k=i}^{n} Y_i^k = Y_i^i$, therefore the desired probability of the union of the events is $\Pr\left[Y_i^i\right] = \frac{1}{i}H_{i-1}$.

(c) Using the same argument as in the beginning of the previous part (ii), we know: The only way that the left child of the root of the treap can change in the given process is by rotating the top-most edge of the right spine, meaning that the rotation replaces the root with its former right child, and therefore the 'old' root now becomes the new left child of the new root. Therefore the indicator variable

$$X_i = [\text{the root changes after node } i \text{ is inserted}]$$

is equivalent to

$$Z_i = [\text{the left child of the root changes after node } i \text{ is inserted}]$$

except at the very first insertion, for $i = 1$, where the treap was still empty, so no rotation will be performed.

Therefore by using the result of (i) we have that $\Pr[Z_i = 1] = \Pr[X_i = 1] = \frac{1}{i}$ for $i \geq 2$, and $\Pr[Z_1 = 1] = 0$. So $\mathbf{E}\left[\sum_{i=1}^{n} Z_i\right] = \sum_{i=1}^{n} \mathbf{E}[Z_i] = H_n - 1$.

Solution 2: Comparisons in Quicksort

To begin with, note that when we talk about the 'number of comparisons', we mean the number of times one element is compared to another element of the sequence to sort. We do not count internal comparisons of the algorithm (like when the algorithm compares the index of an element to another index).

Let a, b be two distinct elements of rank i and j, respectively. In the script (cf. page 22), a mapping is described from the possible runs of quicksort(S) to binary search trees (with the same distribution as our random search trees). According to that mapping, a, b are compared in quicksort(S) if and only if in the corresponding search tree, either b is the ancestor of a or a is the ancestor of b. We have denoted these events using indicator variables by $A_i^j = 1$ and $A_j^i = 1$, respectively.

$$\begin{aligned}\Pr[a, b \text{ are compared in quicksort}(S)] &= \Pr\left[A_i^j = 1 \vee A_j^i = 1\right] \\ &= \Pr\left[A_i^j = 1\right] + \Pr\left[A_j^i = 1\right] = \frac{2}{|i-j|+1}.\end{aligned}$$

The first equality follows from the fact that the two events A_i^j and A_j^i are disjoint (for $i \neq j$).

Solution 3: Quickselect vs. Random Search Trees

Let the random variable $X_{k,n}$ denote the number of comparisons carried out by a call to quickselect(k,S) with $|S| = n$. Clearly, $X_{k,n}$ depends on the randomness in the algorithm. We can associate in a natural way a (partial) search tree to every call of quickselect. We want to express the random variable $X_{k,n}$ with reference to the random search tree only.

The algorithm quickselect follows the path from the root down to the node containing the key of rank k. To do that it has to compare the pivot v to all the members of the subtree rooted at v (excluding v itself) for all v on the path from the root to the node with rank k. (Indeed it has also to compare the element of rank k with the members of its subtree.) Therefore the number of comparisons is

$$X_{k,n} = \sum_{i \in A_k} (W_n^{(i)} - 1).$$

where A_k is the set of all ancestors of the element of rank k (including k itself) and $W_n^{(i)}$ is (just as in Exercise 1.26) the size of the subtree rooted at the vertex of rank i in a random search tree with n elements.

We can also express $X_{k,n}$ exclusively in terms of the indicator variables A_i^j, namely

$$\sum_{i \in A_k} (W_n^{(i)} - 1) = \sum_{i=1}^{n} A_k^i \cdot (W_n^{(i)} - 1) = \sum_{i=1}^{n} A_k^i \cdot \left(\left(\sum_{j=1}^{n} A_j^i \right) - 1 \right).$$

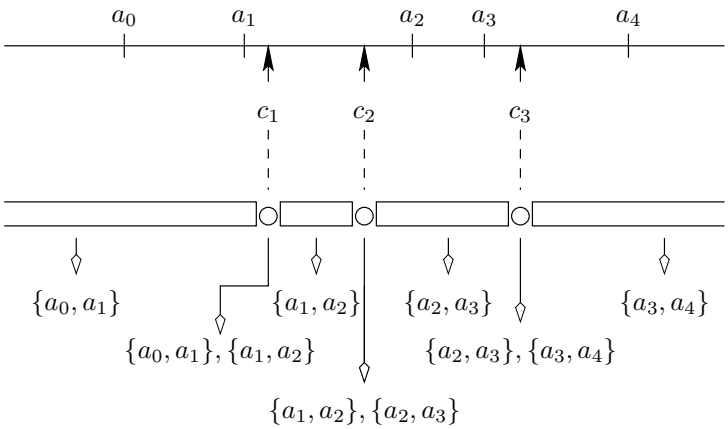
Solution 4: Two Closest Numbers

This problem lends itself to a straightforward solution using the "locus approach" as described in the lecture. First we have to find the regions of of equal answer. While it is trivial to see where the regions with the same closest point lie, it is not so obvious for the closest pair. Note that when we move a query point from $-\infty$ to $+\infty$, we will see each pair $\{a_i, a_{i+1}\}$ as the correct answer for some interval. The transition from $\{a_{i-1}, a_i\}$ to $\{a_i, a_{i+1}\}$ occurs as soon as a_{i+1} is closer to q than a_{i-1}. The threshold position for this is obviously at $(a_{i-1} + a_{i+1})/2$. Therefore we may proceed as follows.

First sort the n elements $a_i \in S$ such that $a_0 < a_1 < \cdots < a_{n-1}$. Then partition the real line into the following intervals and points:

$$(-\infty, c_1), c_1, (c_1, c_2), c_2, (c_2, c_3), c_3, \ldots, c_{n-2}, (c_{n-2}, \infty),$$

where $c_i := (a_{i-1} + a_{i+1})/2$ for every $i \in \{1..n-2\}$. The next picture schematically shows the construction.



Each interval is associated with a nearest neighbor pair. At the points c_i, there are two possible pairs and the way the task is described, the answer for $c_i = q$ is ambiguous as both $\{a_{i-1}, a_i\}$ and $\{a_i, a_{i+1}\}$ fulfill the condition.

Given this data structure and a query point q, a pair of nearest neighbors can be found in $\mathcal{O}(\log n)$ time. Preprocessing can be done in $\mathcal{O}(n \log n)$ time by any optimal sorting algorithm and $\mathcal{O}(n)$ space.

Solution 5: Finding a Key vs. Line Hitting Convex Polygon

Let us first give the proof the exercise asked for. Then we would like to elaborate on the purpose of this exercise.

Proof. Observe that $\ell : y = 2kx - k^2$ is the tangent of the parabola $g : y = x^2$ at point (k, k^2): The slope of line ℓ is $2k$ which is the derivative of g at position k and (k, k^2) lies on both, ℓ and g. Thus $k = a_i$ for any $i \in \{0, \ldots, n-1\}$ implies that (a_i, a_i^2) lies on ℓ and the polygon C. For the other direction let us assume that C intersects ℓ at some point p. Because the parabola is a strictly convex function[1], g touches C only at the points (a_i, a_i^2). Therefore by also using that ℓ is a tangent of g, we get $p = (a_i, a_i^2)$ for some $i \in \{0, \ldots, n-1\}$. □

This exercise showcases a way to prove a lower complexity bound for a geometric problem. Such so-called *negative results* ("there is *no* algorithm better than...") are often very hard to prove because the argument has to go over all possible algorithms, a family of objects so complex that we are today hardly able to understand it. In the present case, the above geometric considerations show that deciding whether a (query) line hits a (preprocessed) convex polygon consisting of n points cannot be any easier than deciding whether a (query) number $q \in \mathbf{R}$ is contained in a (preprocessed) set of n keys $S \subseteq \mathbf{R}$.

[1] Recall: a function $f : \mathbf{R} \to \mathbf{R}$ is strictly convex iff $\forall \lambda \in (0,1), \forall x_1, x_2 \in \mathbf{R}, x_1 \neq x_2$,

$$\lambda f(x_1) \; + \; (1-\lambda) f(x_2) \; < \; f\big(\, \lambda x_1 \; + \; (1-\lambda) x_2 \big)\,.$$

Because if that were easier, then we could preprocess the set S of keys by generating the points $P := \{(x, x^2) | x \in S\}$, preprocess them as a polygon and then each time a query $q \in \mathbf{R}$ is asked, instantiate the line $l_q : y = 2qx - q^2$ and use the (faster) algorithm to check whether l_q hits it. Thence if we have a lower bound for the effort needed to answer the query $q \in S$, the same bound applies to the polygon and query line problem. So do we have such a lower bound?

You have learnt in your first year already that searching for a key in a set of n keys needs a least $\Omega(\log n)$ steps, which is also achievable, e.g. through binary search. *Searching* here means to say for the query number q, which is the next-smallest (or next-largest, or both) key in S. The argument there was the following: Since there are at least $n + 1$ possible answers the algorithm needs to be able to give, at least $\lceil \log(n + 1) \rceil$ bits of information have to be acquired to distinguish between them. So if the only thing we can do with the numbers is comparing two of them, we need at least $\lceil \log(n + 1) \rceil$ comparisons.

Does this now imply together with the exercise that deciding whether a query line hits a polygon needs at least $\Omega(\log n)$ steps and the algorithm we saw in the lecture is therefore optimal? Not yet really, we have to be extremely careful. Here are two flaws: firstly, we have proved that deciding whether a query line hits a polygon is as difficult as deciding whether a real q is in a set of reals S. That's not the same as *searching*, it is a decision problem. Nobody asks you to say which is the next-smallest or next-largest key in S, you just have to say *yes* or *no*. That might be easier than the lower bound we know. Secondly, what was proved in earlier classes was that searching in a set S of comparable keys which you cannot do anything else with but *comparing* them takes $\Omega(\log n)$ of these comparisons. But here we do not have such a set of keys, we have *real numbers*. There is more we can do with real numbers than comparing. The computational model we are considering when designing geometric algorithms is usually a model where basic operations like addition, subtraction, multiplication, division and taking roots can be evaluated in constant time. So if we really want to prove that our algorithm is optimal in this powerful model of computation, we have to prove that every algorithm that just *decides* whether a number $q \in \mathbf{R}$ is in a preprocessable set S of keys carries out at least $\Omega(\log n)$ basic arithmetic operations.

Proving this is very cumbersome and would exceed the scope of this course by far. But Michael Ben-Or has done it in [1].

Theorem 2 (Ben-Or). *Consider a computational model operating on reals by executing the basic operations addition, subtraction, multiplication, division, taking of the square root and comparisons ($<$, $>$ and $=$). Then for any preprocessed set $S \subset \mathbf{R}$ of size $|S| = n$, any algorithm answering for a query $q \in \mathbf{R}$ whether $q \in S$ needs to carry out in the worst case at least $\Omega(\log n)$ basic operations.*

Using this, we can infer that our $O(\log n)$ time algorithm for the line-hitting-polygon problem is in fact best possible. Even though the result turned out as expected, it has to be distinctly understood that this was by no means a priori obvious.

References

[1] Michael Ben-Or. *Lower bounds for algebraic computation trees*. In Proceedings of the Fifteenth Annual ACM Symposium on the Theory of Computing STOC, pages 80-86, 1983.