

Range Medians

Sariel Har-Peled* S. Muthukrishnan†

April 4, 2008

Abstract

We study a generalization of the classical median finding problem to batched query case: given an array of unsorted n items and k intervals in the array, the goal is to determine the median in *each* of the intervals in the array. We give an algorithm that uses $O(n \log k)$ comparisons and show a matching lower bound of $\Omega(n \log k)$ comparisons for this problem.

1 Introduction

The classical median finding problem is to find the *median* item, that is, the item of rank $\lceil n/2 \rceil$ in an unsorted array of size n . We focus on the comparison model, where items in the array can be compared only using comparisons, and we count the number of comparisons performed by any algorithm¹. It is known since the 70's that this problem can be solved using $O(n)$ comparisons in the worst case [BFP⁺73]. Later research [BJ85, SPP76, DZ99, DZ01] showed that the number of comparisons needed for solving the median finding algorithm is between $(2 + \varepsilon)n$ and $2.95n$ in the worst case (in the deterministic case). Closing this gap for a deterministic algorithm is an open problem, but surprisingly, one can find the median using $1.5n + o(n)$ comparisons using a randomized algorithm [MR95].

We study the following generalization of the median problem.

The k -range-medians Problem. The input is an unsorted array \mathcal{S} with n entries. A sequence of k queries Q_1, \dots, Q_k is provided. A query $Q_j = [l_j, r_j]$ is an *interval* of the array, and the output is x_1, \dots, x_k , where

$$x_j = \text{median} \left\{ \mathcal{S}[l_j], \mathcal{S}[l_j + 1], \dots, \mathcal{S}[r_j] \right\}$$

*Department of Computer Science; University of Illinois; 201 N. Goodwin Avenue; Urbana, IL, 61801, USA; sariel@uiuc.edu; <http://www.uiuc.edu/~sariel/>. Work on this paper was partially supported by a NSF CAREER award CCR-0132901.

†Google.com

¹In the algorithms discussed in this paper, the computation performed beyond the comparisons will be linear in the number of comparisons.

for $j = 1, \dots, k$. We refer to this as the *k-range-medians problem*. The problem is to build a data-structure for \mathcal{S} such that it can answer this kind of queries quickly. Notice that the intervals are possibly overlapping.

This is the *interval* version of the classical median finding problem, and it is interesting on its own merit. In addition, there are many motivating scenarios where they arise.

Examples. This motivation arose in our work with analyzing logs of internet advertisements (aka ads). We have the log of clicks on ads on the internet (such as in sponsored search with Yahoo! or Google): each record gives the time of the click as well as the varying price paid by the advertiser for the click, and the log is arranged in time-indexed order. Then, the array $\mathcal{S}[i]$ is the price for the i th click. Any given advertiser runs several ad campaigns simultaneously spread over different intervals of time. As an example, an advertiser may run campaign A from January to March, campaign B for 2 weeks in February, campaign C on the weekends in March, and so on. The advertiser then wishes to compare his cost to the general ad market during the period his campaigns ran, and a typical comparison is to the median price paid for clicks during those time intervals. Hence, what arises is an instance of the *k-range-medians problem*, for possibly intersecting set of intervals.

As another example, consider IP networks where one collects what are known as SNMP logs: For each link that connects two routers, one collects the total bytes sent on that link in each fixed length duration like say 5 minutes [KMZ03]. Then, the array $\mathcal{S}[i]$ is the number of bytes sent on that link in the i th time duration. A traffic analyst is interested in finding the median value of the traffic level within a *specific* time window such as a week, office hours, or weekends, or the median within *each* such time window. Equally, the analyst is sometimes interested in median traffic levels during specific external events such as the time duration when an attack happened or a new network routing strategy was tested.

There are other attributes in addition to time where applications may solve range median problems. For example, $\mathcal{S}[i]$ may be the total value of real estate sold in postal zipcode area i arranged in sorted order, and an analyst may be interested in the median value for a borough or a city represented by a consecutive set of zipcodes. ■

One can ask similar interval versions of other problems too, for example, the median may be replaced by (say) the maximum, minimum, mode or even the sum.

- For sum, a trivial $O(n)$ preprocessing to compute all the prefix sums $P[j] = \sum_{i \leq j} \mathcal{S}[i]$ suffices to answer any interval query $Q_j = [l_j, r_j]$ in optimal $O(1)$ time using $P[r_j] - P[l_j - 1]$.
- If the summation operator (i.e., \sum) is replaced by a semigroup operator (where the subtraction operator is absent), then \mathcal{S} can be preprocessed in $O(nk)$ space and time and each query can be answered in $O(\alpha_k(n))$ where α_k is a slow growing function [Yao82], and this is optimal under general semigroup conditions [Yao85].
- For the special cases of the semigroup operator such as the maximum or minimum, a somewhat nontrivial algorithm is needed to get same optimal bounds as for the \sum case (see for example [BFC04]).

The median operator is not a semigroup operator and presents a more difficult problem. The only prior results we know are obtained by using the various tradeoffs shown in [KMS05]. For the case when $k = 1$, the interesting tradeoffs for preprocessing space and query times are respectively, roughly, $O(n \log^2 n)$ and $O(\log n)$, or $O(n^2)$ and $O(1)$, or $O(n)$ and $O(n^\varepsilon)$ for constant fraction ε [KMS05]. These bounds for individual queries can be directly applied to each of the k interval queries in our problem, resulting in a multiplicative k factor in the query complexity. In particular, the work of Krizanc *et al.* [KMS05] implies a $O(n \log^2 n + k \log n)$ time algorithm for our problem.

Our main result here is an improved, and in fact optimal, bound for the problem.

Theorem 1.1 *There is a deterministic algorithm to solve the k -range-medians problem in $O(n \log k + k \log k \log n)$ time. Furthermore, in the comparison model, any algorithm that solves this problem requires $\Omega(n \log k)$ comparisons.*

The k -range-medians problem seems to be a fairly basic problem and it is worthwhile to have tight bounds for it. In particular, $\Theta(n \log k)$ may not be the bound one suspects at first glance to be tight for this problem.

The lower bound holds even if the intervals are *hierarchical*, that is, for any two intervals, either one of them is contained in the other, or they are disjoint. On the other hand, the upper bound holds even if the queries arrive online, in the amortized sense. Our algorithm uses *relaxed* sorting on pieces of the array, where only a subset of items in a piece are in their correct sorted location. Relaxed sorting like this has been used before for other problems.

In the following, the k th element of a set S (or element of rank k) would refer to the k th smallest element in the set S . For simplicity, we assume the elements of \mathcal{S} are all unique.

2 The Lower Bound

Recall that \mathcal{S} is an unsorted array of n elements. Assume that n is a multiple of k . Let $\Psi(n, k) = \left\{ \frac{in}{k} \mid i = 1, \dots, k \right\}$, for $n > k > 0$. We will say an element of \mathcal{S} is the *i th element* of \mathcal{S} if its rank in \mathcal{S} is i .

Claim 2.1 *Any algorithm **MedianAlg** that computes all the elements of rank in $\Psi(n, k)$ from \mathcal{S} needs to perform $\Omega(n \log k)$ comparisons in the worst case.*

Proof: Let $m_i = in/k$, for $i = 0, \dots, k$. An element would be *labeled* i if it is larger than the m_{i-1} th element of \mathcal{S} and smaller than the m_i th element of \mathcal{S} (note, that the m_k th element of \mathcal{S} is the largest element in \mathcal{S}). An element would be *unlabeled* if its rank in \mathcal{S} is in $\Psi(n, k)$.

Note, that the output of the algorithm is the indices of the k unlabeled elements. We will argue that just computing these k numbers requires $\Omega(n \log k)$ time.

Consider an execution of **MedianAlg** on \mathcal{S} . We consider the comparison tree model, where the input travels down the decision tree from the root, at any vertex a comparison is being made, and the input is directed either to the right or left child depending on the result of the comparison.

A labelling (at a vertex v of the decision tree) is *consistent* with the comparisons seen so far by the algorithm if there is an input with this labelling, such that it agrees with all the comparisons seen so far and it reaches v during the execution. Let Z be the set of labellings of \mathcal{S} consistent with the comparisons seen so far at this vertex v .

We claim that if $|Z| > 1$ then the algorithm can not yet terminate. Indeed, in such a case there are at least two different labellings that are consistent with the comparisons seen so far. If not all the labellings of Z have the same set of k elements marked as unlabeled, then the algorithm has different output (i.e., the output is just the indices of the unlabeled elements), and as such the algorithm can not terminate.

So, let $\mathcal{S}[\alpha]$ be an element that has two different labels in two labellings of Z . There exists two distinct inputs $B = [b_1, \dots, b_n]$ and $C = [c_1, \dots, c_n]$ that realizes these two labellings. Now consider the input $D(t) = [d_1(t), \dots, d_n(t)]$, where $d_i(t) = b_i(1-t) + tc_i$, for $t \in [0, 1]$ and $i = 1, \dots, n$. We can perturb the numbers b_1, \dots, b_n and c_1, \dots, c_n so that there is never a $t \in [0, 1]$ for which three entries of $D(\cdot)$ are equal to each other (this can be guaranteed by adding random infinitesimal noise to each number, and observing that the probability of this bad event has measure zero). Note that $D(0) = B$ and $D(1) = C$.

Furthermore, since for the inputs B and C our algorithm had reached the same node (i.e., v) in the decision tree, it holds that for all the comparisons the algorithm performed so far, it got exactly the same results for both inputs.

Now, assume without loss of generality, that the label for b_α in B is strictly smaller than the label for c_α in C . Clearly, for some value of t in this range, denoted by t^* , $d_\alpha(t)$ must be of rank in the set $\{m_1, \dots, m_k\}$. Indeed, as t increases from 0 to 1, the rank of $d_\alpha(t)$ starts at the rank of b_α in B , and ends up with the rank of c_α in C . But $D(t^*)$ agrees with all the comparisons seen by the algorithm so far (since if $b_i < b_j$ and $c_i < c_j$ then $d_i(t) < d_j(t)$, for $t \in [0, 1]$). We conclude that the assignment that realizes $D(t^*)$ must leave $d_\alpha(t)$ unlabeled. Namely, the set Z has two labellings with different sets of k elements that are unlabeled, and as such the algorithm can not terminate and must perform SOME more comparisons if it reached v (i.e., v is not a leaf of the decision tree).

Thus, the algorithm can terminate only when $|Z| = 1$. Let $\beta = n/k - 1$, and observe that in the beginning of **MedianAlg** execution, it has

$$M = \frac{n!}{k!(\beta!)^k}$$

possible labellings for the output. Indeed, a consistent labeling, is made out of k unlabeled elements, and then β elements are labeled by i , for $i = 1, \dots, k$. Now, by Stirling's approximation, we have

$$M \geq \frac{(\beta k)!}{(\beta!)^k} \approx \frac{\sqrt{2\pi\beta k} \frac{(\beta k)^{\beta k}}{e^{\beta k}} (\beta k)!}{\left(\sqrt{2\pi\beta} \frac{\beta^\beta}{e^\beta}\right)^k (\beta!)^k} = \frac{\sqrt{2\pi\beta k} (\beta k)^{\beta k}}{(\sqrt{2\pi\beta} \beta^\beta)^k} = k^{\beta k} \frac{\sqrt{2\pi\beta k}}{(\sqrt{2\pi\beta})^k}.$$

Each comparison performed can only half this set of possible labellings, in the worst case. It follows, that in the worst case, the algorithms needs

$$\Omega(\log M) = \Omega\left(\beta k \log k - \frac{k}{2} \log(2\pi\beta)\right) = \Omega(\beta k \log k) = \Omega(n \log k)$$

comparisons, as claimed. ■

Lemma 2.2 *Solving the k -range-medians problem requires $\Omega(n \log k)$ comparisons.*

Proof: We will show that given an algorithm for the k -range-medians problem, one can reduce it, in linear time, to the problem of Claim 2.1. That would immediately imply the lower bound.

Given an input array \mathcal{S} of size n , construct a new array \mathcal{T} of size $4n$ where the first n elements of \mathcal{T} are $-\infty$, $\mathcal{T}[n+1, \dots, 2n] = \mathcal{S}$, and $\mathcal{S}[j] = +\infty$, for $j = 2n+1, \dots, 4n$. Clearly, the k th element of \mathcal{S} is the median of the range $[1, 2n+2k-1]$ in \mathcal{T} . Thus, we can solve the problem of Claim 2.1 using k median range queries, implying the lower bound. ■

Observe that the lower bound holds even for the case when the intervals are hierarchical.

3 Our Algorithm

We first consider the case when all the query intervals are provided ahead of time. We will present a slow algorithm first, and later show how to make it faster to get the optimal bounds. Our algorithm uses the following folklore result.

Theorem 3.1 *Given ℓ sorted arrays with total size n , there is a deterministic algorithm to determine median of the set formed by the union of these arrays using $O(\ell \log(n/\ell))$ comparisons.*

Since we were unable to find a reference to this result, we describe this algorithm in Appendix A.

3.1 A Slow Algorithm

Here we show how to solve the k -range-medians problem.

Let I_1, \dots, I_k be the given (not necessarily disjoint) k intervals in the array $\mathcal{S}[1..n]$. We break \mathcal{S} into (at most) $2k-1$ atomic disjoint intervals labeled in the sorted order B_1, \dots, B_m , such that an atomic interval does not have an endpoint of any I_i inside it. Next, we sort each one of the B_i 's, and build a balanced binary tree having B_1, \dots, B_m as the leaves in this order. In a bottom-up fashion we merge the sorted arrays sorted in the leaves, so that each node v stores a sorted array S_v of all the elements stored in its subtree. Let T denote this tree that has height $O(\log k)$.

Now, computing the median of an interval I_j , is done by extracting the $O(\log k)$ canonical nodes in T that covers I_j . Next, we apply Theorem 3.1, and using $O(\log n \log k)$ comparisons, we get the desired median. We now apply this to the k given intervals. Observe that sorting the atomic intervals takes $O(n \log n)$ comparisons and merging them in $O(\log k)$ levels takes $O(n \log k)$ comparisons in all. This gives:

Lemma 3.2 *The algorithm above uses $O(n \log n + k \log n \log k)$ comparisons.*

Note, that this algorithm is still mildly interesting. Indeed, if the intervals I_1, \dots, I_k are all “large”, then the running time of the naive algorithm is $O(nk)$, and the above algorithm is faster for $k > \log n$.

3.2 An Optimal Algorithm

The main bottleneck in the above solution was the presorting of the pieces of the array corresponding to atomic intervals. In the optimal algorithm below, we do not fully sort them.

Definition 3.3 A subarray X is u -sorted if there is a sorted list \mathcal{L}_X of at most (say) $20u$ elements of X such that these elements appear in this sorted order in X (not necessarily as consecutive elements). Furthermore, for an element α of \mathcal{L}_X , all the elements of X smaller than it appear before it in X and all the elements larger than α appear after α in X . Finally, we require that the distance between two consecutive elements of \mathcal{L}_X in X is at most $|X|/u$, where $|X|$ denotes the size of X . We will refer to the elements of X between two consecutive elements of \mathcal{L}_X as a *segment*.

An array X of n elements that is n -sorted is just sorted, and a 0-sorted array is unsorted. Another way to look at it, is that the elements of \mathcal{L}_X are in their final position in the sorted order, and the elements of the intervals are in an arbitrary ordering.

Lemma 3.4 *Given an unsorted array X , it can be u -sorted using $O(|X| \log u)$ comparisons, where $|X|$ denotes the number of elements of X .*

Proof: We just find the median of X , partition X into two equal size subarrays, and continue recursively on the two subarrays. The depth the recursion is $O(\log u)$, and the work at each level of the recursion is linear, which implies the claim. ■

Lemma 3.5 *Given a two u -sorted arrays X and Y , they can be merged into an u -sorted array using $O(|X| + |Y|)$ comparisons.*

Proof: Convert Y into a linked list. Insert the elements of \mathcal{L}_X into Y . This can be done by scanning the list of Y until we arrive at the segment Y_i of Y that should contain an element b of \mathcal{L}_X that we need to insert. We partition this segment using b into two intervals, add b to \mathcal{L}_Y , and continue in this fashion with each such b . This takes $O(|Y_i|) = O(|Y|/u)$ comparisons per b (ignoring the scanning cost which is $O(|Y|)$ overall). Let Z be the resulting u -sorted array, which contains all the elements of Y and all the elements of \mathcal{L}_X , and $\mathcal{L}_Z = \mathcal{L}_X \cup \mathcal{L}_Y$. Computing Z takes

$$O\left(|Y| + |\mathcal{L}_X| \frac{|Y|}{u}\right) = O(|Y|)$$

comparisons.

We now need to insert the elements of $X \setminus \mathcal{L}_X$ into Z . Clearly, if a segment X_i of X has α_i elements of \mathcal{L}_Z in its range, then inserting the elements of X_i would take $O(|X_i| \log \alpha_i)$ comparisons. Thus, the total number of comparisons is

$$O\left(\sum_i |X_i| \log \alpha_i\right) = O\left(\sum_i \frac{|X|}{u} \log \alpha_i\right) = O\left(\frac{|X|}{u} \sum_i \alpha_i\right) = O(X),$$

since $|X_i| \leq |X|/u$, $\log \alpha_i \leq \alpha_i$ and $\sum_i \alpha_i = O(u)$.

The final step is to scan over Z , and merge consecutive intervals that are too small (removing the corresponding elements from \mathcal{L}_Z), such that each interval is of length at most $|Z|/u$. Clearly, this can be done in linear time. The resulting Z is u -sorted since its sorted list contains at most $2u + 1$ elements, and every interval is of length at most $|Z|/u$. ■

Note, that the final filtering stage in the above algorithm is need to guarantee that the resulting list \mathcal{L}_Z size is not to large, if we were to use this merging step several times.

Using the above two lemmas, we get the following result, which is building up to the algorithmic part of Theorem 1.1.

In the following, we need a modified version of Theorem 3.1 that works for u sorted arrays.

Theorem 3.6 *Given ℓ u -sorted arrays A_1, \dots, A_ℓ with total size n and a rank k , there is a deterministic algorithm that returns ℓ contiguous subintervals B_1, \dots, B_ℓ of these arrays and a number k' , such that the following properties hold.*

- (i) *The k' th ranked element of $B_1 \cup \dots \cup B_\ell$ is the k th ranked element of $A_1 \cup \dots \cup A_\ell$.*
- (ii) *The running time is $O(\ell \log(n/\ell))$ time.*
- (iii) $\sum_{i=1}^{\ell} |B_i| = O(\ell \cdot (n/u))$.

Proof: For every element of \mathcal{L}_{A_i} realizing the u -sorting of the array A_i , we assume we have its rank in A_i precomputed. Now, we execute the algorithm of Theorem 3.6 on these (representative) sorted arrays (taking into account their associated rank). The main problem is that now a rank of an element is only estimated approximately up to an 9additive) error of n/u . As such, in the end of process of trimming down the representative arrays, we might still have active intervals of total length $2n/u$ in each one of these arrays, resulting in the bound on the size of the computed intervals.

Note that the required modification of the algorithm of Theorem 3.1 are tedious but straightforward, and we omit the details. ■

Lemma 3.7 *There is a deterministic algorithm to solve the k -range-medians problem in $O(n \log k + k \log k \log n)$ time, when the k query intervals are provided in advance.*

Proof: We repeat the algorithm of Section 3.1 using u -sorting instead of sorting, for u to be specified shortly. Building the data-structure (i.e., the tree over the atomic intervals) takes $O(n \log u)$ comparisons. Indeed, we first u -sort the atomic intervals, and then we merge them as we go up the tree.

A query of finding the median of array elements in an interval is now equivalent to finding the median for $m = O(\log k)$ u -sorted arrays A_1, \dots, A_m . Using the algorithm of Theorem 3.6 results in m intervals B_1, \dots, B_m that belong to A_1, \dots, A_m , respectively, such that we need to find the k' th smallest element in $B_1 \cup \dots \cup B_m$. The total length of the B_i s is $O(mn/u)$. Now we can just use the brute force method. Merge B_1, \dots, B_m into a single array and find the k' th smallest element using the classical algorithm. This take $O(mn/u)$ comparisons. We have to repeat this k times, and the number of comparisons we need is

$$O\left(km \frac{n}{u} + km \log n\right) = O(n + k \log k \log n),$$

for $u = k^2$, since $m = O(\log k)$. Thus, in all, the number of comparisons using by the algorithm is $O(n \log k + k \log k \log n)$. ■

We can extend this bound to the case when the intervals are presented in an online manner, and we get amortized bounds.

Lemma 3.8 (When k is known in advance.) *There is a deterministic algorithm to solve the k -range-medians problem in $O(n \log k + k \log k \log n)$ time, when the k query intervals are provided in an online fashion, but k is known in advance.*

Proof: The idea is to partition the array into $u = k^2$ atomic intervals all of the same length, and build the data-structure of these atomic intervals. The above algorithm would work verbatim, except for every query interval I , there would be two “dangling” atomic intervals that are of size n/u that contain the two endpoints of I .

Specifically, to perform the query for I , we compute $m = O(\log k)$ u -sorted arrays using our data-structure. We also take these two atomic intervals, clip them into the query interval, u -sort them, and add them to the m u -sorted arrays we already have. Now, we need to perform the median query over these $O(\log k)$ u -sorted arrays, which can do, as described above. Clearly, the resulting algorithm has running time

$$O\left(n \log u + k \log u \log n + k \frac{n}{u} \log u\right) = O(n \log k + k \log k \log n),$$

since $u = k^2$. ■

Lemma 3.9 (When k is *not* known in advance.) *There is a deterministic algorithm to solve the k -range-medians problem in $O(n \log k + k \log k \log n)$ time, when the k query intervals are provided in an online fashion.*

Proof: We will use the algorithm of Lemma 3.8.

At each stage, we have a current guess to the number of queries to be performed. In the beginning this guess is a constant, say 10. When this number of queries is exceeded, we *square* our guess, rebuild our data-structure from scratch for this new guess, and continue. Let $k_1 = 10$ and $k_i = (k_{i-1})^2$ be the sequence of guesses, for $i = 1, \dots, \beta$, where $\beta = O(\log \log k)$. We have that the total running time of the algorithm is

$$\sum_{i=1}^{\beta} O(n \log k_i + k_i \log k_i \log n) = O(n \log k + k \log k \log n),$$

since $\log k_{i-1} = (\log k_i)/2$, for all i . ■

Lemma 3.9 implies the algorithmic part of Theorem 1.1.

4 Concluding Remarks

The k -range-medians problem is a natural interval generalization of the classical median finding problem: unlike interval generalizations of other problems such as max, min or sum which can be solved in linear times, our problem (surprisingly) needs $\Omega(n \log k)$ comparisons,

and we present an algorithm that solves this problem with running time (and number of comparisons) $O(n \log k)$. A number of technical problems remain and we list them below.

- Currently, our algorithm uses $O(n \log k)$ space. It would be interesting to reduce this to linear space.
- Say the elements are from an integer range $1, \dots, U$. Can we design $o(n)$ time algorithms in that case using word operations? For the classical median finding problem, both comparison-based and word-based algorithms take $O(n)$ time. But given that the comparison-based algorithm needs $\Omega(n \log k)$ comparisons for our k -range-medians problem, it now becomes interesting if word-based algorithms can do better for integer alphabet.
- Say one wants to only answer median queries approximately for each interval (see [BKMT05] for some relevant results). Can one design $o(n \log k)$ algorithms?

Suppose the elements are integers in the range $1, \dots, U$. We define an approximate version where the goal is to return an element within $(1 \pm \varepsilon)$ of the correct median in value, for some fixed ε , $0 < \varepsilon < 1$. Then we can keep an *exponential histogram* with each atomic interval of the number of elements in the range $[(1+\varepsilon)^i, (1+\varepsilon)^{i+1})$ for each i , and follow the algorithm outline here constructing them for all the canonical intervals on the balanced binary tree atop these atomic intervals. For each interval in the query, one can easily merge the exponential histograms corresponding to and obtain an algorithm that takes time $O(n + k \log k \log U)$, since any two exponential histograms can be merged in $O(\log U)$ time. If the elements are not integers in the range $1, \dots, U$ and one worked in the comparison model, similar results may be obtained using [GK01, GK04], or ε -nets. It is not clear if these bounds are optimal.

- We believe extending the problem to two (or more) dimensions is also of interest. There is prior work for range sum and minimums, but tight bounds for k range medians will be interesting.

Acknowledgments

The authors would like to thank the anonymous referees for their useful comments. In particular, they identified mistakes in an earlier version of this paper.

References

- [BFC04] M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theo. Comp. Sci.*, 321(1):5–12, 2004.
- [BFP⁺73] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Sys. Sci.*, 7(4):448–461, 1973.
- [BJ85] S. W. Bent and J. W. John. Finding the median requires $2n$ comparisons. In *Proc. 17th Annu. ACM Sympos. Theory Comput.*, pages 213–216, 1985.

- [BKMT05] P. Bose, E. Kranakis, P. Morin, and Y. Tang. Approximate range mode and range median queries. In *Proc. 22nd Internat. Sympos. Theoret. Asp. Comp. Sci.*, pages 377–388, 2005.
- [DZ99] D. Dor and U. Zwick. Selecting the median. *SIAM J. Comput.*, 28(5):1722–1758, 1999.
- [DZ01] D. Dor and U. Zwick. Median selection requires $(2 + \epsilon)n$ comparisons. *SIAM J. Discret. Math.*, 14(3):312–325, 2001.
- [GK01] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proc. 2001 ACM SIGOD Conf. Mang. Data.*, pages 58–66, 2001.
- [GK04] M. Greenwald and S. Khanna. Power-conserving computation of order-statistics over sensor networks. In *Proc. 23rd ACM Sympos. Principles Database Syst.*, pages 275–285, 2004.
- [KMS05] D. Krizanc, P. Morin, and M. Smid. Range mode and range median queries on lists and trees. *Nordic J. Comput.*, 12(1):1–17, 2005.
- [KMZ03] F. Korn, S. Muthukrishnan, and Y. Zhu. Checks and balances: Monitoring data quality problems in network traffic databases. In *Proc. 29th Intl. Conf. Very Large Data Bases*, pages 536–547, 2003.
- [MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.
- [SPP76] A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *J. Comput. Sys. Sci.*, 13(2):184–199, 1976.
- [Yao82] A. C. Yao. Space-time tradeoff for answering range queries. In *Proc. 14th Annu. ACM Sympos. Theory Comput.*, pages 128–136, 1982.
- [Yao85] A. C. Yao. On the complexity of maintaining partial sums. *SIAM J. Comput.*, 14(2):277–288, 1985.

A A deterministic algorithm for choosing median from sorted arrays

In this section, we prove Theorem 3.1 by providing a fast deterministic algorithm for choosing the median element of ℓ sorted arrays. As we mentioned before, this result seems to be known, but we are unaware of a direct reference to it, and as such we provide a detailed algorithm.

A.1 The algorithm

Let A_1, \dots, A_ℓ be the given sorted arrays of total size n . As in the randomized algorithm, we maintain ℓ active ranges $[l_i, r_i]$ of the array A_i where the required element (i.e., “median”) lies, for $i = 1, \dots, \ell$. Let k denote the rank of the required median. Let $n_{\text{curr}} = \sum_i (r_i - l_i + 1)$ be the total number of currently active elements.

If $n_{\text{curr}} \leq 32\ell$ then we find the median in linear time, using the standard deterministic algorithm. Otherwise, let $\Delta = \lfloor n_{\text{curr}} / (32\ell) \rfloor$. Pick $u_i - 1$ equally spaced elements from the active range of A_i , where

$$u_i = 4 + \left\lceil \frac{r_i - l_i + 1}{\Delta} \right\rceil.$$

Let L_i be the resulting list of representatives, for $i = 1, \dots, \ell$. Note that L_i breaks the active range of A_i into blocks of size

$$\nu_i \leq \left\lceil \frac{r_i - l_i + 1}{u_i} \right\rceil.$$

For each element of L_i we know exactly how many elements are smaller than it and larger than it in the i th array. Merge the lists L_1, \dots, L_ℓ into one sorted list L . For an element x , let $\text{rank}(x)$ denote the rank of x in the set $A_1 \cup \dots \cup A_\ell$. Note, that now for every element x of L we can estimate its $\text{rank}(x)$ to lie within an interval of length $T = \sum_{i=1}^{\ell} \nu_i$. Indeed, we know for an element of $x \in L$ between what two consecutive representatives it lies for all ℓ arrays. For element $x \in L$, let $R(x)$ denote this range where the rank of x might lie.

Now, given two consecutive representatives x and y in the i th array, if $k \notin R(x)$ and $k \notin R(y)$ then the required median can not lie between x and y , and we can shrink the active range not to include this portion. In particular, the new active range spans all the blocks which might contain the median. The algorithm now updates the value of k and continues recursively on the new active ranges.

A.2 Analysis

We observe that the error estimate for the rank of a representative is bounded by

$$\begin{aligned} U &= \sum_{i=1}^{\ell} \nu_i \leq \ell + \sum_{i=1}^{\ell} \frac{r_i - l_i + 1}{u_i} \leq \ell + \sum_{i=1}^{\ell} \frac{r_i - l_i + 1}{4 + \frac{r_i - l_i + 1}{\Delta}} = \ell + \Delta \sum_{i=1}^{\ell} \frac{r_i - l_i + 1}{4\Delta + r_i - l_i + 1} \\ &\leq \ell + \ell\Delta \leq \frac{n_{\text{curr}}}{32} + \ell \left\lfloor \frac{n_{\text{curr}}}{32\ell} \right\rfloor \leq \frac{n_{\text{curr}}}{16}, \end{aligned}$$

since $\ell \leq n_{\text{curr}}/32$ and by the choice of Δ .

Consider the sorted merged array B of all the active elements. The length of B is n_{curr} , and assume, for the sake of simplicity of exposition, that the desired media is in the second half of B (the other case follows by a symmetric argument). Note, that any representative x that falls in the first quarter of B has a rank that lies in a range shorter than $T < n_{\text{curr}}/4$, and as such it can not include k . In particular, let t_i be the index in A_i of the first representative in the active range (of A_i) that does not falls in the first quarter of B . Observe that

$$\sum_i (t_i - l_i + 1) \geq n_{\text{curr}}/4.$$

As such, the total number of elements that are being eliminated by the algorithm (in the top of the recursion) is at least

$$\sum_i ((t_i - l_i + 1) - 2\nu_i) \geq \sum_i (t_i - l_i + 1) - 2 \sum_i \nu_i = \frac{n_{\text{curr}}}{4} - 2U \geq \frac{n_{\text{curr}}}{8}.$$

Namely, each recursive call continues on an active ranges smaller by a factor of $(7/8)$ from the original array.

The total length of L_1, \dots, L_ℓ is $O(\ell)$, and as such the total work (ignoring the recursive call) is bounded by $O(\ell \log \ell)$. As such, the running time is bounded by

$$T(n) = O(\ell \log \ell) + T((7/8)n),$$

where $T(\ell) = O(\ell \log \ell)$. Thus, the total running time is $O(\ell \log \ell \log(n/\ell))$.

A.3 Doing even better - a faster algorithm

Observe, that the bottleneck in the above algorithm is the merger of the representative lists L_1, \dots, L_ℓ . Instead of merging them, we will instead compute the median x of $L = L_1 \cup \dots \cup L_\ell$. If $R(x)$ does not contain k , then we can throw away at least $n_{\text{curr}}/4$ elements in the current active ranges and we continue recursively. Otherwise, compute the element z of rank $n_{\text{curr}}/4$ in L . Clearly, $k \notin R(z)$ and one can throw, as above, as constant fraction of the active ranges. The resulting running time (ignoring the recursive call) is $O(\ell)$ (instead of $O(\ell \log \ell)$). As such, the running time of the resulting algorithm is $O(\ell \log(n/\ell))$.