CS166 Midterm Exam

Luis A. Perez

Problem One: Range Minimum Queries

The ***range distinct query problem*** (or ***RDQ***) is the following problem: preprocess an array A to efficiently support queries of the form "given a subarray of the array A, how many distinct elements are there in that subarray?" For example, consider this array shown below:

Value	2	7	1	8	2	8	1	8	2
Index	0	1	2	3	4	5	6	7	8

Here, performing an RDQ over the range $[2, 5]$ would return 3, as there are three distinct values in the range (namely, 1, 2, and 8). Performing an RDQ over the range $[5, 7]$ would return 2 because there are two distinct values in the range (1 and 8), performing an RDQ over the range $[0, 0]$ would return 1, and performing an RDQ over the range $[0, 8]$ would return 4 (with the distinct values being 1, 2, 7, and 8).

Design a data structure for RDQ that has a (possibly expected) preprocessing time of $O(n)$ and a (worst-case) query time of $O(z)$, where z is the number of distinct elements in the queried subarray.

Some hints:

- Tag each element with the index of the previous occurrence of that element in the overall array (or with -1 if it's the first copy of that element).
- To count the number of distinct elements in a range, search for the first copy of each distinct element within that range. (You don't care about the second, third, etc. copy of each element.)
- Use a divide-and-conquer strategy.

And, of course, given the title of this question, use RMQ to combine the three above ideas together.

Solution:

Overview We mostly follow the hints. The general idea for our data structure is to first tag our elements with the index of the previous occurrence. Let us define $prev[i]$ as this tag value. The key insight is that RDQ(i, j) consists of all corresponding tags for $i \leq k \leq j$

such that $prev[k] < i$ (e.g, the previous occurrence of this value occurred before this range). We can then use the traditionall RMQ (modified to return the index) to perform a type of search which gives us all such k (of which there are z). We stop the search as soon as the minimum in a range is $\geq i$, since this tells us all the values in that sub-range are duplicates and we don't need to process them further. This will give us the $O(z)$ runtime for query.

Pre-Processing

We describe the pre-processing step. First, we tag each element with the index of the previous occurrence of that element in the overall array. We can do this by creating a hash table H, which starts empty, and a *prev* array which is also empty. We then iterate over our input array from left to right, using the hash table to store mappings from elements to the previously seen index. We set $prev[i]$ to the previously seen index for $A[i]$ (if it is in H) or -1 if it does not. We then replace the mapping for $A[i]$ in H to have the current index.

As an example, our *prev* array would look like $prev = \{-1, -1, -1, -1, 0, 3, 2, 5, 4\}$ for the figure given in our problem statement.

We then construct the Fischer-Heun RMQ data-structure from lecture on the now fully constructed *prev* array. Note that this data structure will be modified such that $\text{RMQ}_{prev}(i, j)$ returns the *index* of the left-most occurrences of the minimum, rather than the value itself. This modification does not affect the analysis we carried out in lecture of RMQ. This completes our pre-processing.

Query

Let us now describe how to answer the query $\text{RDQ}(i, j)$. We first store $l = i$. and then issue the query $k = \text{RMQ}_{prev}(i, j)$. If $k \geq l$, we end the process for this subrange. Otherwise, we increment our distinct count by 1, and repeat this process on the subranges $[i, k-1]$ and $[k+1, j]$ (if non-empty), again comparing the returned index to l and either stopping or again splitting the range further.

At the end of this process, we return the count variable we've maintained, which essentially consists of the number of RMQ queries we issued which returned a value $k < l$.

Proof of Correctness

The correctness of this algorithm relies on two key facts. We proof these seperately.

Fact 1 The number of distinct elements in $A[i \cdots j]$ is equal to the number of elements in $prev[i \cdots j]$ such that $prev[i] < j$.

The proof of this fact is rather straight-forward. We can see that by construction, $prev[k]$ corresponds to the index of the previous occurrence in our array of $A[k]$, or -1 if it is the first occurrence. As such, the cases where $prev[k] < i$ correspond to the *first* occurrence of the element contained in $A[k]$ in our subarray (since it implies the previous occurrence was before the start of our range). Since this counts only the first-occurrence of each distinct element, the count is equal to the number of distinct elements.

Fact 2 Our RMQ loop decribed above finds (and counts) all k such that $prev[k] < i$.

The proof is also rather straight forward. We show both direction. Suppose we have a k such that $prev[k] < i$. Then $prev[k]$ is the minimum of some subrange of $[i, j]$. Since the process only stops when when a subrange's minimum is $\geq l$, then we will eventually perform an RMQ such that the returned index is k. At this point, we will count k, and the ignore it in all future RMQs. As such, it is counted exactly once. For the other direction, suppose that we count some k returned from our RMQ loop. Then this k (by definition of RMQ), must be the minimum value in some subrange of $[i, j]$ and it must be the case that $prev[k] < i$.

Combining the two facts above, we have that our data structure returns the number of distinct elements in $A[i \cdots j]$.

Pre-Processing Runtime Complexity The tagging of elements is done in expected $O(n)$ time, since inserting into the hash table is expected $O(1)$ and setting each index of $prev$ is $O(1)$ ($O(n)$ to construct the array once). We do this for the n elements in our input. Constructing the Fischer-Heun RMQ structure takes $O(n)$ pre-processing time, since it is constructed over the $prev$ array which has n elements.

As such, the total pre-processing time is expected $O(n)$.

Query Runtime Complexity This is a bit more interesting. From Fact 2, we know that the RMQ loop is executed at most $3z$ times, where z is the number of uniques. This is because each successful execution of RMQ captures a single distinct element (by Fact 2), but can spawn 2 RMQs each of which might be unsuccessful. As such, the number of RMQs executed is $O(z)$. Keeping the counter and returning it is $O(1)$. As such, the total run-time of a single query is $O(z)$, where z is the number of distinct elements in the queried subarray.

The above completes our data structure, with the desired runtimes.

Problem Two: String Data Structures

The ***label*** of a node in a suffix tree is the string formed by tracing the path from the root of the tree to that node. Prove that if the suffix tree for some nonempty string T contains a node with label w, then the suffix tree for T also contains a node with label x for each suffix x of w. (Some algorithms on suffix trees rely on this fact and associate each node in the suffix tree with a pointer to its longest suffix.)

Solution: We wish to prove that if the suffix tree for some non-empty string T contains a node with label w, then the suffix tree for T also contains a node with label x for each suffix x of w.

First, we recall the definition of a *branching word*. A *branching word* in T is a string S where there are characters $a \neq b$ such that Sa and Sb are substrings of T. Next, we recall the following theorem from class:

Theorem 1

The suffix tree for a string T has an internal node for a string S if and only if S is a branching word in T.

The above is sufficient for our proof.

We know that w is the ***label*** of a node of the suffix tree for T. If $|w| = 1$ (the label is one character), there are no suffixes, and our claim is vacuously true. If we have $w = a\$$ (e.g, w is a suffix of T), then our claim also holds almost immediately, since any suffix x of w will be a suffix of T, and as such a leaf-node of the suffix tree will be ***label***ed with x.

As such, we consider the more interesting case where $|w| > 1$ and w does not end with \$. This immediately implies that w must be the label ending at an internal node of the suffix tree for T. As such, by Theorem 1, w must be a branching word of T. By the definition of branching word, this implies that there exists $a \neq b$ such that wa and wb are both substrings of $T\$$. Now, consider any suffix x of w. Note that xa and xb are substrings of T, making x a *branching word*. By another application of Theorem 1, we have there exists an internal node for the string x, since x is a branching word.

This concludes our proof, showing that the tree for T also contains a node with label x for each suffix x of w.

Problem Three: Balanced Trees

Suppose we will be performing a series of lookups in a binary search tree where the choice of which key to look up next is sampled independently from a fixed probability distribution. More specifically, let the keys in the tree be $x_1 < x_2 < \cdots < x_n$ and let the probability of looking up key x_i be p_i. You can assume that the access probabilities are all positive and that every search is for a key that is present in the tree.

As a reminder, a binary search tree has the ***entropy property*** if the expected cost of a lookup is $O(1+H)$, where H is the Shannon entropy of the probability distribution from which lookups are sampled. (See the lecture on splay trees for a refresher on Shannon entropy.)

In lecture, we claimed that weight-balanced trees have the entropy property, assuming that each key's associated weight is its access probability. Prove this.

Solution: The intuition here is that, essentially, the weighed balance tree will tend to place items with higher access probability closer to the root of the tree, since this will be the only way to maintain a difference in weights which is small. This will lead precisely to the ***entropy property*** of the tree.

We first recall that the expected cost of a lookup can be defined as

$$1 + \sum_{i=1}^{n} p_i c(x_i)$$

where $c(x_i)$ is simply the cost of accessing element x_i in the tree from the root. We note that the cost of accessing an element is proportional to the number of edges followed when performing the access, with the cost to access the root being 0, its children 1, and so on until the leaves cost $\lg n$ operations to access (for a BST).

First, we recall results we achieved in Problem Set 2 related to weighed-balanced trees. We re-state it here. Note that since $p_i > 0$, all of our results still apply.

Claim 1 Let $w(T)$ be the total weight in a weight-balanced tree T. Then for $\varepsilon = \frac{2}{3}$, the left and right substrees each must have weight at most εW. More formally, $w(T_r), w(T_l) \leq \varepsilon w(T)$. We note that intuitively, this means weights of the subtrees along any path decrease geometrically.

Proof of Claim 1 This was done as part of Problem Set 2, so we do not re-state. Please refer to the solution there.

Claim 2 Let S be a sub-tree of T whose weight is such that $w(S) > \varepsilon^k w(T)$. Then S must be rooted at layer k or above (with $k = 0$ corresponding to the layer containing the root node of T).

Proof of Claim 2 The proof can be done by induction on k. For $k = 0$, the claim is vacuously true (no such sub-tree exists). Consider $k = 1$. Then S is a sub-tree such that $w(S) > \varepsilon w(T)$. By Claim 1, this sub-tree must in fact be T, since this is the only tree whose weight is strictly greater than $\varepsilon W(T)$, and as such, our claim holds true.

Now, assume the claim holds for sub-trees whose weight is such that $w(S) > \varepsilon^k w(T)$. We will show that it holds for trees of smaller weight. In particular, for all trees whose weight is such that $w(S) > \varepsilon^{k+1} w(T)$. Let $p(S)$ be the tree whose root's left or right child is S. We first show that $w(p(S)) > \varepsilon^k w(T)$.

$$\begin{aligned} & w(S) \leq \varepsilon w(p(S)) && \text{(Claim 1 results)} \\ \implies\; & w(p(S)) \geq \frac{1}{\varepsilon} w(S) && \text{(Divide by } \varepsilon\text{)} \\ & > \frac{1}{\varepsilon}\varepsilon^{k+1} w(T) = \varepsilon^k w(T) && \text{(Definition of } S\text{)} \end{aligned}$$

As such, with the above, we have show that $w(p(S)) > \varepsilon^k w(T)$. By the inductive hypothesis, this means that $p(S)$ must be rooted at layer k or above. As such, the direct child of $p(S)$ (e.g, S), must be rooted at layer $k+1$ or above, concluding the proof for Claim 2.

Claim 3

We now make the following claim: In a weighed-balance tree, T, where the element weights are their access probabilities, an element with access probability p_i is located in layer $\lg \frac{1}{p_i}$ or higher (closer to the root).

Proof of Claim 3 First, we note that $w(T) = 1$ (probabilities must sum to 1). Next, let us consider the subtree, S, rooted at the node whose weight/access probability is p_i. It is trivially true that $w(S) \geq p_i$. Taking $k = -\lg p_i \geq 0$ (since $p_i \in (0,1)$, we note that:

$$\begin{aligned} \varepsilon^k w(T) &= \varepsilon^k && (w(T) = 1) \\ &= \varepsilon^{-\lg p_i} && (k = -\lg p_i) \\ &= 2^{-\lg \varepsilon \lg p_i} && (\varepsilon = 2^{\lg \varepsilon}) \\ &< 2^{\lg p_i} = p_i && \text{(Note that } -\lg \varepsilon \leq -\lg \tfrac{2}{3} < 1 \text{ since } \varepsilon \in (\tfrac{2}{3}, 1)) \end{aligned}$$

We can therefore conclude that $w(S) \geq p_i > \varepsilon^k w(T)$, for our T and chosen value of $k = -\lg p_i$. It follows then from Claim 2 that the subtree S must be rooted at layer $k = \lg \frac{1}{p_i}$ or above. Since the root of S is the node with access probability p_i, we have that the element with access probability p_i is located in layer $\lg \frac{1}{p_i}$ or higher.

Claim 4

The expected cost of a lookup in a weighed-balance tree T whose element weights are their access probabilities p_i with $p_i > 0$ is $O(1 + H)$ where $H = \sum_{i=1}^n -p_i \lg p_i$ (the shannon entropy).

Proof of Claim 4

As previously discussed, the cost of the lookups will be given by $1 + \sum_{i=1}^n p_i c(x_i)$ where $c(x_i)$ is the number of edges traversed when accessing x_i. From the results of Claim 3, we can upper-bound $c(x_i) \leq \lg \frac{1}{p_i}$, since we've shown that element x_i must be at level $\lg \frac{1}{p_i}$ or above. As such, an upper bound on the expected cost is:

$$1 + \sum_{i=1}^{n} p_i c(x_i) \leq 1 + \sum_{i=1}^{n} -p_i \lg p_i = 1 + H$$

As such, the expected cost of a lookup in this weighed balance tree is $O(1+H)$, which means they have the **entropy property**, as desired.

Problem Four: Amortization

Suppose you have a sorted sequence of keys $x_1 < x_2 < \cdots < x_n$ from which you'd like to construct a B-tree of order b. To do so, you could insert each key into an empty tree one at a time. If you were to do this, though, you'd find that you were doing a lot of unnecessary work traversing the tree top-down, since each search is guaranteed to terminate in the rightmost leaf. (Do you see why?) This top-down searching would mean that the runtime of inserting the keys would be $\Omega(n \log_b n)$.

There's a much faster way to build a B-tree from scratch. Maintain a pointer to the rightmost leaf node in the B-tree, and annotate each node in the B-tree with the number of keys it contains. Then, insert each key, one at a time and in sorted order, using the following procedure:

1. Place the key in the next free slot in the rightmost leaf.
2. If the rightmost leaf overflows, split that leaf and propagate any further splits higher up in the tree using the usual node-splitting procedure.

Prove that the amortized cost of inserting each element this way is $O(1)$ using either the banker's method or the potential method. This shows that the cost of building a B-tree from a sorted sequence using this algorithm is $O(n)$, independent of the order of the B-tree.

Although B-trees are typically stored on-disk and analyzed in terms of disk reads and writes, for the purposes of this problem please analyze this algorithm in terms of the total number of operations performed, not the number of block transfers.

Solution: We show that the run-time of the above algorith is amortized $O(1)$. We do this using the potential method, by defining the potential Φ as the number of nodes in our tree containing $2b$ keys (e.g, the nodes have been maxed out). We note that we-re using the definition of b-trees where each node has $[b, 2b]$ keys, and where the minimum number of keys is $1 \implies b \geq 1$.

We note that an empty tree has $\Phi = 0$, and that the potential is never negative, so our total change in potential can only be positive, giving us an upper-bound on the run-time.

There are two cases to consider.

- If the rightmost leaf node has a free slot, we simply place the key directly there. This requires that we (1) follow a pointer to the right-most leaf node, (2) check the count, (3) increment the count, and (4) place the key into the free slot.

  As such the actual run-time in this case is $O(1)$, with $\Delta\Phi = O(1)$ (since our right-most leaf node might now contain $2b$ keys).

- Now for the more interesting case. In this case, our rightmost leaf node has overflown. As such, we must split the leaf and propogate any further splits up the tree. Note that this once again requires (1) following a pointer to the right-most leaf-node, (2) checking the count, (3) splitting the leaf node into two, (4) modifiying the counts, (5) placing the key into the free slot, and (6) propagating the split upwards.

Steps (1)-(5) are all constant time operations, for an actual run-time of $O(1)$. Now, propogating the split upwards will lead to a similar sequence of operations possibly being performed since we're again inserting a new key into a node. Let k be the number of nodes which require a split. This means that the actual run-time is $\Theta(k)$ (each split is constant-time, and we must perform k of them).

Now, looking at the potential, we note that $\Delta\Phi = -k$. We've stated above that $b \geq 1$, and that the nodes we split are node with $2b$ keys (+1 for the key we're trying to insert). As such, after the split, we're left with two nodes each of size b, and since $b \geq 1$, we have that $b \leq 2b$. This means that our potential decreases by 1 per-split we perform. As such, we have an amortized run-time of $O(1)$.

As such, the total number of operations performed while building the a B-tree from a sorted sequence is amortized $O(n)$, since each insert takes $O(1)$ amortized time.

Problem Five: Randomization

You have a randomized data structure D that estimates some (unknown) quantity A. Let's denote the estimate returned by the data structure as $\hat{D}$. Let's further suppose that $\mathrm{E}\left[\hat{D}\right] = A$ (the estimator is unbiased) and also that $\mathrm{Var}\left[\hat{D}\right] = A^2$ (the variance of the estimate is fairly high).

Using D as a black box, design an improved data structure that estimates A with tunable accuracy and confidence parameters. Specifically, your data structure should be parameterized over two user-provided values $\varepsilon \in (0, 1)$ and $\delta \in (0, 1)$ and should satisfy the following guarantees:

- $\Pr\left[|\hat{B} - A| > \varepsilon A\right] < \delta$, where $\hat{B}$ is the estimate returned by your data structure;
- the space usage is at most a factor of $O(\varepsilon^{-2} \log \delta^{-1})$ larger than the space usage for D; and
- the runtime of every operation on your data structure is at most a factor of $O(\varepsilon^{-2} \log \delta^{-1})$ larger than the runtime cost of the corresponding operation on D.

As a hint, you can reduce the variance of an unbiased estimator by running lots of copies of that estimator in parallel and taking the average.

Solution:

Overview: The hint essentially gives it away. We can use the average estimate to reduce our variance (by a constant factor), and then the median of the average to achieve the final bounds.

The way we do this is by creating $d = 24 \ln \delta^{-1}$ groups each containing $w = 3\varepsilon^{-2}$ copies of our D data structure. Label each data structure as D_{ij}, meaning it belongs to group i and is copy j within that group.

We then run all copies in parallel. Let $\hat{D}_{ij}$ be the estimate for our unknown quantity A returned by D_{ij}. Then our data structure will simply return $\hat{B} = \mathrm{median}_{i=1}^{d}\{D_i = \frac{1}{w}\sum_{j=1}^{w} \hat{D}_{ij}\}$ as its estimate (the median of averages). It is important that the randomness for each of our dw copies is independent. More formally, if we treat $\hat{D}_{ij}$ as a random variable, we have that $\hat{D}_{ij} \perp \hat{D}_{lm}$ for all $i \neq l, j \neq m$.

We also assume that $\mathrm{E}\left[\hat{D_{ij}}\right] = A$ is a finite value. As for $\mathrm{Var}\left[\hat{D_{ij}}\right] = A^2$, we assume that it is non-zero. If it is 0, then that means that a single data structure D is a perfect-estimator for A, so it trivially satisfies all of the specified requirements and as such is not interesting (just run this single, perfect estimator).

Details of the Data Structure For any operation which is not returning the estimate, simply copy and forward any arguments to each of D_{ij}'s corresponding operation.

For the estimate operation, obtain an estimate $\hat{D}_{ij}$ from each D_{ij}. Take the average over each group i, and then take the median of the results and return this as the estimate $\hat{B}$.

Proof of Approximate Correcness Most of the Time

We wish to show that the data structure designed above will have the following guarantee:

$$\Pr\left[|\hat{B} - A| > \varepsilon A\right] < \delta$$

Intuitively, this means that we will fail (provide an answer outside ε percent the true value) less than δ of the time. In other words, we'll be close to our true answer (as specifed by ε) $(1-\delta)$ percent of the time.

First, we note that we can treat $\hat{D}_{ij}$ as a random variable from some distribution, and that $\hat{D}_{ij} \perp \hat{D}_{lm}$ for $i \neq l, j \neq m$ since each data structure has its own independent source of randomness. From the problem statement, we have that $\mathrm{E}\left[\hat{D}_{ij}\right] = A$ (unbiased estimator with finite expectation) and that $\mathrm{Var}\left[\hat{D}_{ij}\right] = A^2 > 0$. Let is now consider each of our groups. We have $\hat{D}_i = \frac{1}{w}\sum_{j=1}^{w} \hat{D}_{ij}$, which is just the average of w identically distributed independent variables (by our assumptions stated previously). As such, we have:

$$\begin{aligned}
\mathrm{E}\left[\hat{D}_i\right] &= \mathrm{E}\left[\frac{1}{w}\sum_{j=1}^{w}\hat{D}_{ij}\right] && \text{(Definition)} \\
&= \frac{1}{w}\sum_{j=1}^{w}\mathrm{E}\left[\hat{D}_{ij}\right] && \text{(Linearity of Expectation)} \\
&= \frac{1}{d}\sum_{j=1}^{w} A && \text{(Previous results)} \\
&= A
\end{aligned}$$

and we also have:

$$\begin{aligned}
\text{Var}\left[\hat{D}_i\right] &= \text{Var}\left[\frac{1}{w}\sum_{j=1}^{w}\hat{D}_{ij}\right] && \text{(Definition)}\\
&= \frac{1}{w^2}\text{Var}\left[\sum_{j=1}^{w}\hat{D}_{ij}\right] && (\text{Var}\left[aX\right] = a^2\text{Var}\left[X\right])\\
&= \frac{1}{w^2}\sum_{j=1}^{w}\text{Var}\left[\hat{D}_{ij}\right] && (D_ij \perp D_ik \text{ for } k \neq j)\\
&= \frac{1}{w^2}\sum_{j=1}^{w}A^2 && \text{(Previous results)}\\
&= \frac{A^2}{w}
\end{aligned}$$

As we can see, $\hat{D}_i$ is still an unbiased estimator, and the more copies we run, the more we can decrease our variance. In fact, we note that by our assumptions, for $w \geq 0$, we have that $\text{E}\left[\hat{D}_i\right]$ is finite and that $\text{Var}\left[\hat{D}_i\right] > 0$. We can therefore apply Chebyshev's inequality, with $k = \sqrt{3}$, to have:

$$\begin{aligned}
\text{Pr}\left[|\hat{D}_i - A| > k\frac{A}{\sqrt{w}}\right] &< \frac{1}{k^2} && (\text{For any } k > 0)\\
\text{Pr}\left[|\hat{D}_i - A| > \sqrt{3}\frac{A}{\sqrt{w}}\right] &< \frac{1}{3} && (\text{Using } k = \sqrt{3})\\
\text{Pr}\left[|\hat{D}_i - A| > \varepsilon A\right] &< \frac{1}{3} && (\text{Using } w = 3\varepsilon^{-2})
\end{aligned}$$

As such, the estimator $\hat{D}_i$ from each group will be within a factor of ε with some non-zero probability (for our specific case, we've chosen at least $\frac{2}{3}$).

With the above, we can now consider our final estimate $\hat{B}$, which is simply the median of all of the $d = 24\ln\delta^{-1}$ group estimates. Let us analyze this final estimate.

We let X be the random variable which counts the number of $\hat{D}_i$ which are "bad" (eg, $|\hat{D}_i - A| > \varepsilon A$). From the previous analysis, we know that each data structure has a failure probability of at most $\frac{1}{3}$. We also know that each data structure is independent, by construction. As such, we can upper-bound X with a binomial variable Binom$(d, \frac{1}{3})$. Note that this satisfies the requirements for applying the Chernoff bound, so we have:

$$\begin{aligned}
\Pr\left[X > \frac{d}{2}\right] &< e^{\frac{-d(\frac{1}{2}-\frac{1}{3})^2}{\frac{2}{3}}} \\
&= e^{-\frac{d}{24}} && \text{(Simplify)} \\
&= e^{-\ln \delta^{-1}} && \text{(Substitute } d = 24 \ln \delta^{-1}\text{)} \\
&= \delta
\end{aligned}$$

As such, we can now safely conclude that:

$$\Pr\left[|\hat{B} - A| > \varepsilon A\right] \leq \Pr\left[X > \frac{d}{2}\right] < \delta$$

which is what we desired.

Space Usage and Run-time Analysis

Let the space usage of a single D be $O(s)$, and the runtime for operation i (including estimate) be $O(r_i)$. The the total space usage of our data structure above is $O(wd) \cdot O(s) = O(\varepsilon^{-2} \log \delta^{-1}) \cdot O(s)$, as required.

Similarly, let us consider the run-time for any operation i on our data structure which is not `estimate`. We simply call the corresponding operation on each of the wd copies, so the total run-time is $O(wd) \cdot O(r_i) = O(\varepsilon^{-2} \log \delta^{-1}) \cdot O(r_i)$.

As for our estimate call, the analysis is similar to all other operations. Additionally, we must average across wd values, and them compute the median of d values. All of this can be done in linear time in the size of the input (average is obvious, computing the median can be done using quickselect or median of medians algorithm if we don't want randomization to take place), so the total run-time is still $O(wd) \cdot O(r_e) = O(\varepsilon^{-2} \log \delta^{-1}) \cdot O(r_e)$ where $O(r_e)$ is the run-time to return the estimate in D.