Algorithms and Data Structures in the noisy comparison model and applications

Nishanth Dikkala, Anil Shanbhag, Emmanouil Zampetakis

Massachusetts Institute of Technology

Abstract

In contrast to classical computational models where every operation gives the correct answer always, we consider models with noise introduced in the operations they perform. In particular we look at the scenario when comparisions between elements turns noisy, i.e, gives the wrong result with a small probability. In this setting, we present existing algorithms in literature for binary search and sorting. We develop algorithms for finding the minimum among n elements and implementing a noisy binary search tree. We also present various applications of the above algorithms to areas such as learning theory, shortest paths in graphs with edge uncertainties and strategic voting.

Keywords: Algorithms, Noisy Comparisons, Uncertainty

1. Introduction

The classical design of algorithms and data structures assumes that the values of the items are known accurately and any operation involving them, such as comparisons or arithmetic, always returns the correct answer. This assumption doesn't hold true in many real world scenarios where the true value of object may not be known and hence comparisons between objects can return the wrong result with some small probability. Consider for example a sport tournament, in this case every time a match is played we have a winner but, there is always a probability that the weaker team will win. So we can see every match as a noisy comparison with some probability of failure. In settings like this we are interested in finding the best team or the complete ordering of all the teams.

The goal of this project is to study existing literature and design algorithms and data structures which work well in presence of noise. We restrict ourselves to studying the following three problems in a noisy setting :

1. Binary search for an item in a sorted array of items
2. Implementing a binary search tree
3. Sorting a set of items

Existing literature considers three different models for noisy setting. Note that, in all these models, the true value of the items is unknown.

Email addresses: `ndikkala@mit.edu` (Nishanth Dikkala), `anils@mit.edu` (Anil Shanbhag), `mzampet@mit.edu` (Emmanouil Zampetakis)

1. **Noisy Comparator Model:** We are given a noisy comparison operator which takes as input two items and produces a binary output. This comparator gives the wrong answer with probability $\delta < \frac{1}{2}$. The only operation allowed in this model is picking two items and asking the noisy comparator which one is greater. If we repeat a comparison $2k + 1$ times, the probability that at least k of the comparisons give the wrong answer is at most ϵ^k where ϵ is a function of δ such that $0 \leq \epsilon < 1$. This will be shown precisely in Section 2.
2. **Coin Flip Model:** Every item i is a coin whose heads probability, p_i, is unknown. We can no longer compare elements directly. We are only allowed to toss the given coins and answer questions regarding their heads probabilities. If we toss a coin i, $1/\epsilon^2$ times, then the probability that the estimated heads probability differs from p_i by more than ϵ, is bounded by a constant from Chernoff bound.
3. **Noisy Input Model:** In this model every comparison has a probability of failure δ. The difference is that in this case by repeating a comparison we don't get a better comparator because the uncertainty comes from the input and therefore the comparisons are not independent.

1.1. Guarantees of the results of the algorithms

It is obvious that in these models we cannot have algorithms that always work, because there is always a probability, even though small, that all the comparisons of the algorithm went wrong. So the guarantees provided in this work can again be of two forms:

1. Probability of wrong answer upper bounded. For instance it could be bounded by a constant $\leq \frac{1}{2}$ or by a function of the size of the input e.x. $1/n$.
2. Given a number of samples, output the provable best solution, with respect to some objective function, based on these samples.

For the binary search and sorting algorithms we have a guarantee of the first type. In the fourth section we also consider a noisy sorting algorithm with guarantee of the second type. Although these guarantees are incomparable, it is well accepted that the second guarantee is harder to achieve than the first one.

1.2. Performance of the algorithms

The well-known measure that is used to measure the performance in the classical theory of algorithms is *running time* of an algorithm. Apart from the total running time, in these models it is sometimes reasonable to measure the performance of our algorithms based on the number of samples that they use from the noisy comparisons. For instance, in the case of a sport tournament, it is reasonable to not let an arbitrary pair of teams play each other more than one or two times. We refer to this objective as the *query complexity* of the algorithm.

1.3. Roadmap

At first we present a naive reduction that can be used in order to transform an algorithm that solves a problem using the classical comparison model, to an algorithm that solves with high probability the same problem in the noisy comparison model. After that we present our algorithm for finding the minimum among n elements in linear time. Then we present our binary search tree data structure that is based on the work of [1] in the noisy comparison model. Next we present an optimal algorithm for binary search in the coin flip model based on the work of Karp and Kleinberg

[2]. Then we give some interesting applications of these models to the design of algorithms for some problems in theoretical computer science. Finally we present an algorithm for sorting n elements in the noisy input model based on the work of Braverman and Mossel [3] and state a nice application of this to social choice theory.

In the appendix we have some proofs that are omitted from Section 5 of the paper and also some results that we had developed on sorting algorithms before we noticed that there were some better results in the work of [1]. These results include :

1. We develop a noisy Fibonacci heap with $O(\log\log n)$ insert and $O(\log n \log\log n)$ delete amortized runtime. The primary reason for developing such a heap is for its application in the sorting algorithm we develop next.
2. Using the heap developed above, we give a sorting algorithm which takes $O(n \log n \log\log n)$ time to output the sorted order with high confidence, which is the major contribution of this paper.
3. Finally, we present a reduction of sorting to a heap with specific guarantees in order to improve the time bound of the sorting algorithm to $O(n \log n \log^* n)$.

We start by stating the Chernoff bound that we are going to use throughout the paper

Lemma 1.1. ***Chernoff Bound****[4] Suppose $X_1, X_2, ..X_n$ be independent random variables taking values in 0,1. Let X denote the sum of random variables and $\mu = E[X]$. Then we have for any δ,*

$$Pr(X - \mu \geq \delta\mu) \leq e^{\frac{-\delta^2\mu}{3}}$$
$$Pr(X - \mu \leq -\delta\mu) \leq e^{\frac{-\delta^2\mu}{2}}$$

2. Naive Solution

Every problem in the noisy setting has a classical analogue which is noise-free. In this section, we show how we can derive a naive solution for a general comparision-based problem in the noisy setting, using a polynomial-time solution to its noise-free analogue.

Lemma 2.1. *Given a noisy comparator operator X which has error probability δ, it is possible to create another noisy comparator operator Y which invokes X $O(c \log n)$ times and has an error probability $O(1/n^c)$, where c is a constant.*

Proof Let Y make $2k$ comparisons where $k = c \log n$. The answer returned is the majority answer among the k comparisons. This comparison has a wrong answer if greater than half the comparisons returned the wrong result. By Lemma 1.1, this probability can be upper bounded by:

$$\begin{aligned} Pr[\Sigma X_i > k] &= Pr[\Sigma X_i - 2k\delta > k - 2k\delta] \\ &= Pr[\Sigma X_i - 2k\delta > (1/2\delta - 1)2k\delta] \\ &< e^{-(1/2\delta-1)^2 k\delta} \\ &< n^{-(1/2\delta-1)^2 c\delta} \end{aligned}$$

□

Consider an algorithm which makes $O(n^c)$ comparisons in the worse-case to get the right solution. Now in the noisy setting, each of these comparisons can return wrong result with probability δ. Let us repeat each of these comparisons $O(c \log n)$ times. By Lemma 2.1, each can go wrong with probability $O(1/n^c)$. Since we are making $O(n^c)$ comparisons, by union bound over all the comparisons we have that the order thus returned is wrong with a constant probability.

The naive solution leads to a multiplicative $O((\log n)^c)$ factor in the running time of the algorithm. In the rest of the paper, we will study or design algorithms which have better than $\log n$ multiplicative factor.

3. Noisy minimum

The simplest thing to find given a comparison operator is the minimum/maximum. For the noise-free case, this can be done trivially using $O(n)$ comparisons. We present an algorithm to do the same in noisy setting with $O(n)$ time complexity which is strictly better than the naive solution based on Section 2 which gives a $O(n \log n)$ solution.

The goal is to find the minimum of a set of elements given a noisy comparison operator (model 1). Given the set of n elements, we construct a tournament graph as follows. Create a leaf node corresponding to every element. Construct a tournament tree over these leaves. Figure 1 shows the graph for 8 elements. Each internal node of the tree contains the minimum of its children. Hence the root of the tree represents the minimum. If we perform a single comparision to find out the item at each internal node, the error probability scales with the depth giving an $O(\log n)$ probability of failure. On the other hand, performing $2 \times O(\log \log n) - 1$ comparisions and taking the majority outcome at every internal node ensures a constant failure probability but the total number of comparisions needed is now $O(n \log \log n)$ (number of internal nodes is $O(n)$). The trick to achieve both the goals is to perform more repeated comparisions higher up the tree and less repeated comparisions lower in the tree. In particular, at each level of the tree, we do $2h - 1$ comparisions where h is height of the level. This uses $O(n)$ comparisions to give a constant failure probability.

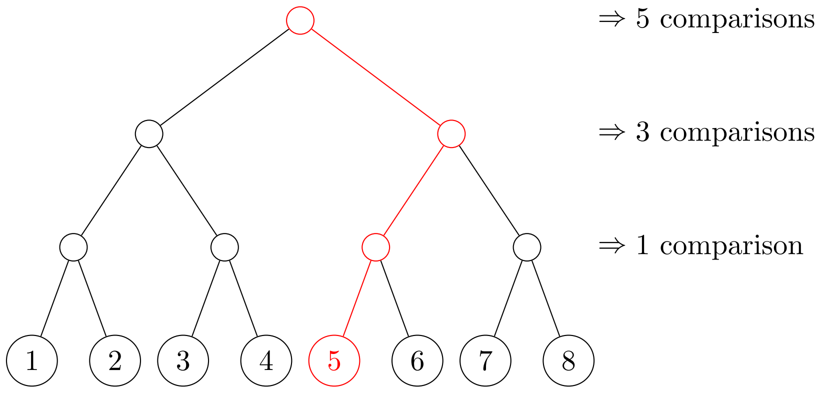

Figure 1: Finding minimum using tournament graph for 8 items

The total number of comparisons forms an arithmetico-geometric sequence and its value is

$$\begin{aligned}
S &= n/2 * 1 + n/4 * 3 + ... + 1 * (2 * log_2(n) - 1) \\
&< n/2 * (1/(1-0.5) + 2 * 0.5/(1-0.5)^2) \\
&< 3n
\end{aligned}$$

An individual comparison can be wrong with probability δ. We want to upper bound the probability of error in minimum. Consider figure 1 again, let element 5 be the minimum. The answer would be correct if all the internal nodes from 5 to the root are correct. The result of rest of comparisons is immaterial. Hence the probability of failure is the probability that the minimum element didn't make it up to the root which is upper bounded by the sum of error probabilities along the path from the minimum leaf to the root. At every level, the node has the wrong result if atleast h of the $2h-1$ the comparisons returned the wrong result. Hence, the probability the minimum is wrong is

$$\begin{aligned}
p &\leq \delta + \delta^2 + ..\delta^{\log_2 n} \\
&\leq \frac{\delta}{1-\delta}
\end{aligned}$$

Since $\delta < 1/2$, $p < 1$. Hence, with probability $> 1 - p$ the algorithm finds the right minimum.

4. Dynamic noisy binary search tree

In this section we present how we can use the ideas of [1] in order to build a dynamic binary search tree with the guarantee that at every point all the elements are in their correct positions with probability $> 1 - 1/n$ where n is the number of elements in the data structure. One basic issue that [1] didn't deal with is what happens if we don't know n a priori or if we have a long sequence of insertions and deletions.

So, we first present an algorithm that finds the correct place to insert an element in a binary search tree conditioned on the ordering of the binary search tree being correct. After that we use this basic operation to describe a dynamic binary search tree when n is unknown at the beginning of the algorithm's execution.

4.1. Inserting into a correct binary search tree

The basic idea is to use binary search with backtracking. In order to be able to understand whether we have to backtrack during the execution of the binary search on the tree, we have to design a test such that each time we reach a node u the test will indicate whether u is on the correct path that we had to follow from the root. If it found to not be on the correct path, we backtrack to the parent of u. Of course this test will also have a probability of failure because our comparisons are noisy.

We now describe a single step of the insert algorithm. The number of steps that the algorithm runs for is a fixed number m that will be specified later.

Suppose we wish to find the position of the element x, and after k steps of execution we reach the node u. We wish to check whether x belongs to the subtree T_u with root u.

If u is an internal node, for this to hold x must be greater than the smallest and smaller than the largest element of T_u. So let $a_u = \min_{v \in T_u} v$ and $b_u = \max_{v \in T_u} v$ then we test whether $a_u \leq x \leq b_u$. So at every node u we also maintain pointers to a_u and b_u. Now once x has reached u we compare it with a_u and b_u and if the test fails, we backtrack and x goes to its parent. If the test succeeds then we compare x with u in order to decide to which child of u to move to in the next step.

If u is a leaf then a_u is defined as the greatest element in the tree which is less than u and b_u is defined as the smallest element in the tree that is greater than u. Therefore the test on the leaf is again $a_u \leq x \leq b_u$. On reaching a leaf, our algorithm doesn't stop. It continues executing until it finishes m steps. If the test at a leaf succeeds we stay at the leaf.

We can model the above search algorithm as a Markov process. Consider a leaf w of the tree and suppose that x belongs to the interval labeling this leaf. Orient all the edges of the tree towards w. Note that for every node u, exactly one adjacent edge is directed away from u and the other adjacent edges are directed towards u. It is easy to see that at every point, the probability that the algorithm will follow the direction of the edges is at least $1 - \delta$. This is because if the comparisons are correct then the algorithm will follow the direction of the edges.

Now we define X_t as the random variable that is 1 if at step t, the execution of the algorithm follows the correct direction and equal to -1 if it follows the opposite direction. Because of the previous arguments, we have that $\mathbb{E}[\sum_t X_t] = (1-\delta)m$ and so by applying Chernoff bound we can get that $\Pr[\sum_t X_t \leq (1-\delta)m/2] \leq \epsilon^{-m}$ for some $0 \leq \epsilon \leq 1$. So if we let $m = (4/\log \epsilon)(1/\delta)\log n$ then with high probability $1/n^2$, $\sum_t X_t > \log n$. But $\sum_t X_t = m_f - m_b$ where m_f is the number of forward movements and m_b is the number of backward movements. Therefore $m_f - m_b > \log n$ which implies that the random walk ends to the correct leaf. Since $m = O(\log n)$ this procedure takes logarithmic time and gives the result with high probability.

4.2. Building a dynamic tree

Using the above procedure we can repeatedly add elements to the tree. Each time when we find the correct position that an element should be placed we know with high probability the ordering of the elements in the tree. So we can use a self balancing binary tree like AVL-trees and once we have the correct position of the elements we can do the balancing without making any additional noisy comparisons by just assume that the correct ordering is the one that we have in the tree at this point. Therefore if we know the number of operations n, that will be done, a priory, then using a self balancing tree and the above procedure we can create a dynamic binary search tree using $m = O(\log n)$ which takes time $O(\log n)$ for insertion, deletion and search. At every point the probability of error of the data structure is less than $n(1/n^2) = 1/n$.

Now if n is unknown then we keep two counters n and n_{old}. At every point $m = O(\log(2n_{old}))$ and $n_{old} \leq n \leq 2n_{old}$. Each time we do an insert we increase n by 1. When n becomes equal to $2n_{old}$, we destroy the data structure and we build it from scratch with $m = O(\log(2n))$ and $n_{old} = n$. This will take $O(n \log n)$ time but we can charge the re-building operations to the last $n/2$ operations performed without reconstructing the data structure. Hence each element gets charged $O(\log n)$ for the initial operation and $2O(\log n)$ for the renewing process. So the amortized cost for each operation is still $O(\log n)$ and the probability that there is a mistake in our data structure is less that $1/n$ at any point because $m = O(\log n)$ at any point.

Therefore we have a dynamic binary search tree that works in the model with noisy comparisons, has the same amortized complexity as the regular binary search trees and at any point during it's operation, has probability at most $1/n$ of failure. All these guarantees are achieved without knowing n in advance.

5. Noisy Binary Search

We now present the noisy binary search algorithm of Karp and Kleinberg[2] which works with the noisy model 2. We are given n biased coins in order of their increasing heads probabilities. Let them be $p_1, p_2, \ldots p_n$ where $p_1 \leq p_2 \leq \cdots \leq p_n$. The algorithm doesn't know $\{p_i\}$ but is allowed to toss coins and observe the outcomes. For simplicity of analysis, we assume $p_0 = 0$ and $p_{n+1} = 1$ and consider the set $\{p_0, \ldots, p_{n+1}\}$ as our search space. A target value $\tau \in [0, 1]$ and a small number ϵ are specified as part of the input and the goal is to find two consecutive indices i and $i + 1$ such that $[\tau - \epsilon, \tau + \epsilon]$ intersects $[p_i, p_{i+1}]$. Such a solution is called an ϵ-good solution. It is easy to see that this goal cannot be achieved always as our estimates of p_i might go wrong so the algorithm finds the right interval with $\geq \frac{3}{4}$ probability.

5.1. A naive algorithm

A naive algorithm would use binary search to locate a good coin, maintaining a pair of indices a and b (initialized to 1 and n) and always testing the coin midway in the current interval $[a, b]$. The heads probability p of the midway coin is to be compared with τ. If p lies outside the interval $[\tau - \epsilon, \tau + \epsilon]$, using $O(1/\epsilon^2)$ tosses of the coin, the algorithm tests whether $p < \tau$ or $p > \tau$ with at most a constant error probability (from the Chernoff bound). Since, we do $\log n$ such comparisons over the course of a binary search, and wish for a constant overall error bound, we ensure that each comparison has $O(1/\log n)$ error probability. To do this, we toss the coin $O(\frac{\log\log n}{\epsilon^2})$ times instead of just $O(\frac{1}{\epsilon^2})$ times, giving an algorithm of running time $O(\frac{\log n \log\log n}{\epsilon^2})$.

5.2. A Faster Algorithm: Binary Search with Backtracking

The naive algorithm introduces an additional $O(\log\log n)$ factor in the running time of the classical binary search algorithm which is removed by a more clever binary search. This algorithm uses backtracking and finds a 1/3-good solution. Later, we see how to get an ϵ-good solution, for an arbitrary ϵ, given the 1/3-good solution. We will present the algorithm when the target $\tau = 1/2$ as this simplifies the analysis of the algorithm. Later, we will show how to reduce the problem of searching for a general τ to one with $\tau = \frac{1}{2}$.

Algorithm for $\tau = 1/2$: Let $\mathbb{T}$ be an infinite rooted binary tree. Each node of the tree is labeled with a pair of indices from the set $\{0, 1, \ldots, n + 1\}$. The labeling is defined recursively as follows. First, the root is labeled $(0, n+1)$. For every vertex v with label (a, b), let $m = \lfloor (a+b)/2 \rfloor$ and label the left child $L(v)$ with (a, m) and the right child $R(v)$ with (b, m). Note that if the label (a, b) of a node satisfies $b = a + 1$, both its children will be labeled (a, b) as well and hence an infinite tree exists (We need an infinite tree due to the randomized nature of the algorithm). Let $P(v)$ denote the parent of v. If v is the root, we let $P(v) = v$.

1. Start round 0 at the root of $\mathbb{T}$.
2. In round t, suppose we are at node $v(t)$ with label (a, b) and whose children are labeled (a, m) and (m, b). We flip coin a twice and coin b twice. If both a-flips are heads or both b-flips are tails, we take this as evidence of $p_a > 1/2$ or $p_b < 1/2$ and we backtrack to the parent node. That is $v(t + 1) = P(v(t))$.

3. Otherwise, we flip coin m and move to $L(v(t))$ if the flip yields heads and move to $R(v(t))$ if the flip yields tails.
4. At the end of every round t, with probability $1/\log n$, we perform a termination test to decide if we should halt before proceeding to round $t+1$. The termination test essentially checks if either coin a or coin b is close to the target.
5. **Termination Test:** Let $k = \lceil 300 \log n \rceil$. We perform k flips of coin a and k flips of coin b. Let the number of heads be h_a and h_b respectively.
   (a) If $\frac{1}{4} \leq h_a/k \leq \frac{3}{4}$, halt and output a.
   (b) If $\frac{1}{4} \leq h_b/k \leq \frac{3}{4}$, halt and output b.
   (c) If $b = a+1$ and $h_a/k < \frac{1}{2} < h_b/k$, halt and output a.
   (d) If none of the above happen, proceed to round $t+1$.

Analysis of the algorithm: The above mentioned algorithm outputs a coin which is 1/3-good and has expected running time $O(\log n)$. The constant 300 in the termination test isn't unique. Any constant $> 2(12)^2 = 288$ would have worked. We sketch the steps of the proof below in 4 lemmas. We defer the full proofs of 3 of the sub lemmas we use to Appendix A. We define a node of $\mathbb{T}$ to be promising if one of the following conditions holds for its label (a, b):

1. $p_a \in [1/4, 3/4]$
2. $p_b \in [1/4, 3/4]$
3. $b = a+1$ and $\frac{1}{2} \in [p_a, p_b]$

Denote by W the set of all promising nodes in $\mathbb{T}$.

Lemma 5.1. *For any t, if $v(t) \in W$, and a termination test is performed at time t, then the algorithm halts with at least a constant probability.*

Lemma 5.2. *Let $d(v(t), W)$ denote the distance in $\mathbb{T}$, from $v(t)$ to the closest node of W. If $v(t) \notin W$, the probability that $d(v(t+1), W) = d(v(t), W) - 1$ is at least $9/16$.*

Using the above two lemmas, we now show that the expected number of rounds for which the algorithm runs is $O(\log n)$. Since each round takes $O(1)$ time in expectation (termination test is performed only with $O(1/\log n)$ probability), this will imply the $O(\log n)$ time complexity of the algorithm. To prove this we will use Azuma's inequality for submartingales which we will state here.

Definition A sequence of random variables $X_0, X_1, \ldots X_n$ is called a submartingale if

$$\mathbb{E}\left[X_{i+1} | X_0, \ldots X_i\right] \geq X_i \ \forall i$$

Lemma 5.3. ***Azuma's inequality:*** *Suppose $X_0, X_1, \ldots, X_n$ is a submartingale and $|X_{i+1} - X_i| \leq 1$ for $0 \leq i < n$. Then $Pr\left(X_n \leq X_0 - t\right) \leq \exp\left(\frac{-t^2}{2n}\right)$.*

Lemma 5.4. *The expected number of rounds for which the algorithm runs is $O(\log n)$.*

We start the proof by defining a potential function Φ. Let $Y_t = d(v(t), W)$ and $Z_t = \#\{s < t | v(s) \in W\}$

$$\Phi_t = \lceil \log_2 n \rceil + Z_t - Y_t - \frac{1}{8}(t - Z_t)$$

Now, $\Phi_0 = \lceil \log_2 n \rceil - d(v(0), W) \geq 0$ because the closest promising node to the root is at a depth $\leq \lceil \log_2 n \rceil$. Next, we show that $\{\Phi_t\}$ is a submartingale. If $v(t) \in W$, then $Z_{t+1} = Z_t + 1$ and $Y_{t+1} \leq Y_t + 1$. Hence, $\Phi_{t+1} \geq \Phi_t$ which implies $\mathbb{E}[\Phi_{t+1} | \Phi_t, v(t)] \geq \Phi_t$. If $v(t) \notin W$, then $Z_{t+1} = Z_t$. From Lemma 5.2, we have that $d(v(t+1), W) = d(v(t), W) + 1$ with probability at most $7/16$ and hence, $\mathbb{E}[Y_{t+1} | Y_t, v(t)] \leq Y_t - 1/8$ which implies $\mathbb{E}[\Phi_{t+1} | \Phi_t, v(t)] \geq \Phi_t$ proving that $\{\Phi_t\}$ is a submartingale.

Now, if the algorithm did not terminate before round t and $t > 160(1 + \log_2(n))$, then $\Phi_t \leq \log_2(n) + 1 + \frac{9}{8}Z_t - \frac{t}{8}$. If $Z_t \leq t/10$, then $\log_2(n) + 1 + \frac{9}{8}Z_t - \frac{t}{8} < t/160$. From Azuma's inequality, we have $\Pr(\Phi_t < t/160) < (1 - \delta)^t$ for some constant δ. If $Z_t > t/10$, then from Lemma 5.1, we have $\Pr(Z_t > t/10) < (1 - \delta')^{(t/\log n)}$ for some other constant δ'. Hence when $t > 160(1 + \log_2(n))$, the probability of the algorithm running for t rounds decays exponentially in $t/\log(n)$ giving an expected number of rounds as $O(\log n)$.

Lemma 5.5. *If the algorithm halts, the probability that the result is not* $1/3$*-good is at most* $4/n$.

Getting an ϵ-good solution from a $1/3$-good solution: Now all that's left is to get an ϵ-good solution given the above algorithm which outputs a $1/3$-good solution. We will sketch the idea in brief and leave out the details. Intuitively, the construction runs by amplifying the deviation of the heads probability of each coin from that of a fair coin. The amplification is such that a $1/3$-good solution among the amplified coins corresponds to an ϵ-good solution in the original coin set. Such an amplification is achieved by repeated tosses of the coin and an application of the Chernoff bound. This introduces an additional $1/\epsilon^2$ factor in the running time of the algorithm. This completes the analysis of the algorithm for $\tau = 1/2$.

Reduction from a general τ to $\tau = 1/2$: The principle behind the reduction is to construct a simulation procedure which, given a coin i with heads probability p_i, simulates a coin with heads probability $f(p_i)$ where f is a strictly increasing linear function such that $f(\tau) = 0.5$. To construct such a procedure, we assume the availability of a fair coin. It is a well-known result in literature that using two tosses of a fair coin in expectation, a coin with an arbitrary heads probability can be simulated. We will use this result to construct the above procedure.

If $\tau > 1/2$, the procedure flips coin i and a simulated coin with heads probability $1/2\tau$, and declares heads if both outcomes are heads. Probability of heads being declared by the procedure is $\frac{p_i}{2\tau}$. If $\tau \leq 1/2$, the procedure flips coin i and a simulated coin with heads probability $\frac{1/2-\tau}{1-\tau}$ and declares heads if at least one of the outcomes is heads. Probability of heads being declared in this case is 1 minus the probability that both the flips resulted in tails, which is, $\frac{(1-p_i)}{2(1-\tau)}$. Hence probability of heads being declared is $1 - \frac{(1-p_i)}{2(1-\tau)} = \frac{1+p_i-2\tau}{2(1-\tau)}$. Finally note that, if $f(\tau(1-\epsilon)) = 1/2(1-\epsilon')$ for some constant ϵ', then $f(\tau(1+\epsilon)) = 1/2(1+\epsilon')$ due to linearity of f. Hence we are still searching for overlap with a symmetric interval. The only difference being that it is now centered at $1/2$ instead of τ.

Given this simulation procedure, for an arbitrary τ, we do the following. Whenever we wish to compare τ with p_i, using 3 tosses in expectation for every toss of coin i, we compare instead $1/2$

with $f(p_i)$.

6. Applications on the noisy comparison model

6.1. Application to Machine Learning

One very interesting application of the model that we are presenting in this paper is an application to machine learning and computational learning theory. A very common problem in this field is the following. Given access to a sample generator of an unknown distribution D that belongs to a class of distributions C, compute a distribution D' that is $\epsilon-$close to D with respect to some distance measure such as the total variation distance.

A common technique for this problem is to find an $\epsilon-$cover C_ϵ of C. This means that for every $D \in C$ there exists $D' \in C_\epsilon$ such that the distance between D and D' is less than ϵ. After finding such a cover of polynomial size, we would like to choose a $D' \in C_\epsilon$ such that the distance between D and D' is less than ϵ. Since C_ϵ is an $\epsilon-$cover it suffices to find the $D' \in C_\epsilon$ which is closest to D. So we set up a noisy comparison that given $D', D'' \in C_\epsilon$ finds the one that is closest to D. To do so we use the samples from D and we also generate some samples from D' and D'' and after $O(\log(1/\delta)/\epsilon^2)$ samples we can conclude the winner if the distance is greater than ϵ with probability of failure δ. For more details on this construction we suggest looking here [5]. There appears to be an analogy between this model and the coin comparison that we described in detail before.

To solve this find minimum problem we set up a tournament as we did in section 3 and we get a winner with the same guarantees doing just $O(n)$ comparisons and using $O(\log n/\epsilon^2)$ samples, i.e. constant number of samples for every comparison.

6.2. Finding shortest paths in graphs with uncertainty

In many real life applications the input that we are given has a lot of uncertainty. One very common example is the case of finding shortest paths in computer networks. A classical problem in this direction is the so called stochastic shortest path problem. In this problem the cost of each edge is a random variable and the goal is to find a path which is optimal in expectation[6].

Here, we assume that we don't know exactly the distribution of the costs of the edges in the network. We assume that we know the topology of the network. First we have to set up a noisy comparison model. So assume that we want to compare the cost of an edge $e_1 = (u_1, v_1)$ with the cost of the edge $e_2 = (u_2, v_2)$. To do this, we send a constant number of packets from u_1 to v_1 and the same number of packets from u_2 to v_2. After that we compare the average times for sending the packets along each of the edges and we output the edge with smaller average time as the winner of this comparison. Here we make the assumption that after this procedure the above comparison has a constant probability of failure δ which is bounded away from $1/2$.

We also assume that we have a lower bound on δ, lets say that we know that we know that the means of the weights are separated by at least a constant. Now we run the usual Dijkstra's algorithm but using the noisy binary search tree that we described in 4, as a heap instead of using a classical binary heap. Then we can ensure that we will find all the optimal paths with high probability because the binary search tree is guaranteed to work will high probability at any point. So our solution is optimal, with a small probability of error compared to the algorithms that find the optimal in expectation solution. The interesting part here is that we do so with running time which is the same as the usual Dijkstra's algorithm which runs is $O(|E|\log|V|)$ time.

7. Noisy sorting without resampling

In this section we describe the work of Braverman and Mossel [3] to deal with sorting in the noisy input model. A challenge that arises in noisy comparison model is that sometimes it is impossible to do a specific comparison more than one or two times. Such a restriction is natural when we want to report a ranking of n sport teams and we are not allowed to ask the teams to play more than once since every tournament has a specific duration. In this case we have to be able to sort the teams, as well as possible, based on the results of the games that they have already played. So since the number of teams is maybe limited we are not interested so much in the total running time of our algorithm but we are interested in the total number of samples that we use conditioned on the fact that every pair should be compared at most once.

In another interpretation of this problem we have as input a directed graph on the n items where the direction of each edge in the graph describes the result of the noisy comparison between them. Now our goal is to find a topological ordering of the items in which the least number of edges are backward and do not follow the topological ordering. This problem is a well known NP-hard problem [7] called *feedback arc set problem for tournaments*. For this problem a PTAS is known, so we can, in polynomial time, approximate as well as we want the best ordering. In this work we are interested in finding the best ordering with high probability and looking at just $O(n\log n)$ comparisons.

Let $q(a_i, a_j) = 1$ if the result of the comparison between the results $a_i \geq a_j$ and let $q(a_i, a_j) = -1$ otherwise. For any σ a permutation of the n elements, we define the score of this permutation to be $s_q(\sigma) = \sum_{i,j:\sigma(i)>\sigma(j)} q(a_{\sigma(i)}, a_{\sigma(j)})$. Our goal in this section is to find a permutation τ that minimizes the quantity $s_q(\tau)$ with high probability. After finding this permutation $s_q(\tau)$ we have to prove that if the comparisons that we made were noisy, but a real ordering π exists, then τ is very close to π.
The algorithm that produces τ consists of two parts

- Sorting an almost sorted list
- Producing an almost sorted list via insertion sort

After having the above procedures the algorithm produces an almost sorted list and then finds the optimal ordering using the first procedure for sorting an almost sorted list.

7.1. Sorting an almost sorted list

Let a list be $k-$almost sorted if, for all $1 \leq i \leq n$, the ith element of the list satisfies $|i-\tau(i)| \leq k$. Then using dynamic programming we can prove the following

Theorem 7.1. *There is an algorithm with running time $O(n^2 \cdot 2^{6k})$, which produces an optimal ordering τ from an $k-$almost sorted list.*

Suppose we want to find the set of elements S_I that will be indexed in the interval $I = [i, j]$ in the optimal ordering, then based on the assumption of the $k-$almost sorted list we know that all the elements with current index $[i+k, j-k]$ will belong to S_I. For the rest we know that they will be at most k positions away, so their current index will be in either $I_1 = [i-k, i+1]$ or $I_2 = [j-k, j+k]$. So in order to find the whole S_I we need to choose the correct $2k$ elements from the set of $4k$ elements in the intervals I_1 and I_2. We have 2^{4k} different choices for doing this. We can enumerate each one of them and choose the best one.

Based on this idea the algorithm proceeds by finding S_I for big interval and recursively finds S'_I for $I' \subseteq I$ until we get an ordering with high probability. Also memoization is used in order to avoid repeated work.

7.2. Producing an almost sorted list via insertion sort

We call a list of elements *almost sorted* if for all $1 \leq i \leq n$ the ith element of the list satisfies $|i - \tau(i)| \leq c \cdot \log n$. Using a careful variation of insertion sort we can prove the following

Theorem 7.2. *There is an algorithm that runs in time n^r, where $r = O(\gamma^{-4}(\beta+1))$ that outputs an almost sorted list with probability* $1 - n^{-\beta}/2$.

Here we use a randomized version of insertion sort. Let $l = c \log n$ and assume that we have an $l-$almost sorted sublist S_{k-1} with $|S_{k-1}| = k-1$. We choose a random element a_k from the list that is not in S_{k-1}. We want to insert a_k in the correct position in S_{k-1}. To do so we split S_{k-1} into parts $B_1, \ldots, B_s$ with length $l' = (c/4) \log n$ each. So our first goal is to find the B_i in which a_k should be placed. We do a binary search through $B'_j s$ in order to find B_i and each time we compare a_k with B_j we compare a_k with every element in B_j and take the majority. Because S_{k-1} is $l-$almost sorted we can prove that with high probability the comparisons between a_k and B_j with $|i-j| > 1$ are correct. Therefore after the binary search we will find $j = i \pm 2$. If we place a_k arbitrarily in B_j then we will get an $l-$almost sorted list S_k. We now repeat by randomly choosing the next element to insert.

7.3. Performance of the algorithm

It is easy to show that has the above algorithm has running time n^r, where $r = O(\gamma^{-4}(\beta+1))$ by carefully calculating the number of steps that we described above. The analysis of the number of comparisons that the algorithm needs is a more complicated procedure and we refer to the paper for the proof that with high probability the number of comparisons made is $O(n \log n)$.

The most interesting and non obvious guarantee of this algorithm is that it approximates the initial ordering π. This is not obvious because the result of our algorithm is the ordering τ that minimizes $s_q(\tau)$ and therefore has no direct relation with π. Despite that we can show that τ approximates π. In order to be able to express this approximation we define $d(a_i) = |\pi(i) - \tau(i)|$. We have two kinds of approximation, approximation in expectation and approximation with high probability. If we let I be the random variable that uniformly takes a value from 1 to n, in expectation the result is

$$\mathrm{E}[d(a_I)] = \text{constant}$$

For the high probability result we have

$$\Pr[d(a_i) \geq c \log n] \leq \tilde{O}(n^{-c}) \quad \forall i$$

7.4. Application to Voting [8]

Social choice theory studies the aggregation of individual preferences towards a collective choice. In one of the most common models, both the individual preferences and the collective decision are represented as rankings of the alternatives. A *voting rule* takes the individual rankings as input and outputs a social ranking.

One can imagine many different voting rules; which are better than others? The popular axiomatic approach suggests that the best voting rules are the ones that satisfy intuitive social choice axioms. For example, if we replicate the votes, the outcome should not change; or, if each and every voter prefers one alternative to another, the social ranking should follow suit. It is well-known though that natural combinations of axioms are impossible to achieve [9], hence the axiomatic approach cannot give a crisp answer to the above question.

A different in a sense competing approach views voting rules as estimators. From this viewpoint, some alternatives are objectively better than others, i.e., the votes are sim- ply noisy estimates of an underlying ground truth. One voting rule is therefore better than another if it is more likely to output the true underlying ranking; the best voting rule is a maximum likelihood estimator (MLE) of the true ranking.

From this point we can view the work of Braverman and Mossel [3]. Given samples from the rankings of the agent, they aim to compute the Kemeny rule; which is the maximum likelihood estimator of the ordering in this model with noisy comparisons. And so we have an efficient algorithm that computes the Kemeny ranking with arbitrarily high probability.

8. Conclusions

In this paper we presented basic algorithms and data structures in different, commonly used, noisy comparison models. We also present some applications of these in a variety of settings in theoretical computer science proving the importance of the research in this field.

One future direction that is maybe interesting especially for real life applications is the investigate what happens when the probability of error in a comparison depends on the actual distance between the elements. This case is captured by the coin flip model and this is the main importance of this model. So it would be interesting if we could extend the binary search tree construction to this model in order to be able to run more complicated algorithms which will have applications in solving real life problem.

References

[1] U. Feige, P. Raghavan, D. Peleg, E. Upfal, Computing with noisy information, SIAM J. Comput. 23 (1994) 1001–1018.

[2] R. M. Karp, R. Kleinberg, Noisy binary search and its applications, in: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, pp. 881–890.

[3] M. Braverman, E. Mossel, Noisy sorting without resampling, in: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 268–276.

[4] M. Tulsiani, S. K. Kundu, M. Mitzenmacher, E. Upfal, J. H. Spencer, Probability and computing: Randomized algorithms and probabilistic analysis, 2013.

[5] C. Daskalakis, G. Kamath, Faster and sample near-optimal algorithms for proper learning mixtures of gaussians, in: Proceedings of The 27th Conference on Learning Theory, COLT 2014, Barcelona, Spain, June 13-15, 2014, pp. 1183–1213.

[6] C. H. Papadimitriou, M. Yannakakis, Shortest paths without a map, Theor. Comput. Sci. 84 (1991) 127–150.

[7] N. Alon, Ranking tournaments, SIAM J. Discrete Math. 20 (2006) 137–142.

[8] I. Caragiannis, A. D. Procaccia, N. Shah, When do noisy votes reveal the truth?, in: ACM Conference on Electronic Commerce, EC '13, Philadelphia, PA, USA, June 16-20, 2013, pp. 143–160.

[9] K. Arrow, Social choice and individual values, John Wiley and Sons (1951).

[10] M. L. Fredman, R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM 34 (1987) 596–615.

Appendix A. Lemmas from Section 5

Lemma Appendix A.1. *For any t, if $v(t) \in W$, and a termination test is performed at time t, then the algorithm halts with at least a constant probability.*

Proof: Since $v(t) \in W$, one of the following three cases holds.

1. $p_a \in [1/4, 3/4]$. In this case, the algorithm will halt if $h_a/k \in [1/4, 3/4]$. The probability that $h_a/k \in [0, 1/4)$ is bounded by the probability that $h_a/k \in [p_a - 1/4, p_a]$ since $p_a \geq 1/4$. This is in turn bounded by a constant (from the Chernoff bound and the fact that h_a/k is symmetrically distributed around p_a). Similarly, the probability that $h_a/k \in (3/4, 1]$ is bounded by a constant, implying that the algorithm halts with at least a constant probability.
2. $p_b \in [1/4, 3/4]$. Use a similar analysis as above.
3. $b = a + 1$ and $1/2 \in [p_a, p_b]$. The algorithm will halt if $h_a/k < 1/2$ and $1/2 < h_b/k$, each of which happens with at least a constant probability using a similar argument as above. Hence the algorithm will halt with a probability at least as much as the product of two constants, which is a constant.

Lemma Appendix A.2. *Let $d(v(t), W)$ denote the distance from $v(t)$ to the closest node in W. If $v(t) \notin W$, the probability that $d(v(t+1), W) = d(v(t), W) - 1$ is at least $9/16$.*

Proof: Note that the path from $v(t)$ to the closest promising node might go up from $v(t)$. Let (a, b) be the label of $v(t)$. If it isn't a promising node, either

1. $p_a < 1/4$ and $p_b < 1/4$. Note that any descendant of $v(t)$ in this case will have $p_a < 1/4$ and $p_b < 1/4$ and hence won't be in the promising set. So the path from $v(t)$ to the closest promising node goes up from $v(t)$. Hence when we backtrack, $d(v(t+1), W)$ decreases by 1. The probability that both b tosses yield tails is at least $9/16$ and hence we backtrack with at least $9/16$ probability.
2. $p_a > 3/4$ and $p_b > 3/4$. We use a similar analysis as above.
3. $p_a < 1/4$ and $p_b > 3/4$ but $b - a > 1$. Now note that, no ancestor of $v(t)$ is in W. Let NB denote the event that we do not backtrack at round t. This happens if at least one of the a tosses is tails and one of the b tosses is heads. Hence $\Pr(NB) \geq (15/16)^2$. Consider the midway coin $m = \lfloor (a+b)/2 \rfloor$. There are two cases
   - $p_m \in [1/4, 3/4]$. In this case, both children of $v(t)$ are in W and hence we move to a mode in W with probablity at least $(15/16)^2 > 9/16$.
   - $p_m < 1/4$ or $p_m > 3/4$. Assume, wlog, the former. In this case, all of W is contained in the right subtree of $v(t)$ and we move to the right when the m toss results in tails which happens with probability $1 - p_m \geq 3/4$. Hence we move closer to a node in W with probability at least $3/4(15/16)^2 > 9/16$.

   Hence, in all cases we move closer to a node in W with a probability at least $9/16$.

Lemma Appendix A.3. *If the algorithm halts, the probability that the result is not $1/3$-good is at most $4/n$.*

Proof: Informally, we show that h_a/k and h_b/k are close to p_a and p_b. And since $\frac{h_a}{k}, \frac{h_b}{k}$ satisfy the conditions for halting, they must in turn be close to $1/2$ ensuring the closeness of p_a, p_b, or far enough such that the interval $[p_a, p_b]$ includes $1/2$ (in this case $b = a + 1$ also).
Formally, the probability that $|h_a/k - p_a| > 1/12$, using the Chernoff bound 1.1 is upper bounded by $2\exp(-k/288) \leq 2/n$. Similarly $|h_a/k - p_a| > 1/12$ with at most $2/n$ probability. If neither of the above 2 events happen, given that the algorithm halted one of the following must happen:

1. $h_a/k \in [1/4, 3/4]$ and the algorithm outputs a. In this case $p_a \in [1/6, 5/6]$.
2. $h_b/k \in [1/4, 3/4]$ and the algorithm outputs b. In this case $p_b \in [1/6, 5/6]$.
3. $b = a + 1$ and $1/2 \in [h_a/k, h_b/k]$. Also, $h_a/k < 1/4$ and $h_b/k > 3/4$. In this case, $p_a < 1/4 + 1/12 = 1/3$ and $p_b > 3/4 - 1/12 = 2/3$ and hence $1/2 \in [p_a, p_b]$.

In all 3 cases we output a coin which is 1/3-good, and since the probability of one of the above not holding is at most $4/n$ this proves our statement.

Appendix B. Our Results

Appendix B.1. Noisy Fibonacci Heap

We design a noisy heap which can perform insert and delete-minimum operations efficiently. The intention is to use this heap to develop an efficient sorting algorithm which we present in **??**. We assume that we know the maximum number of elements in the heap at any time. We denote this number by n. We do not need the decrease-key operation for our sorting algorithm and hence our heap does not support decrease-key. We modify the classical Fibonacci heap to get our noisy heap which performs insert in $O(\log\log n)$ amortized time and delete-min in $O(\log n \log\log n)$ amortized time. This noisy heap has the guarantee that every call to delete-min can independently be wrong with at most $O(1/\log^2(n))$ probability. We present the sketch of the design and the analysis below. We assume the reader has working knowledge of Fibonacci heaps [10]. Choose as the potential function $\Phi = \text{Number of roots} \times k \log\log n$.

- INSERT: We add the item to be inserted into the list of root nodes. Real cost is $O(1)$ but increase in potential is $O(\log\log n)$. Hence amortized cost $= O(\log\log n)$.

- MERGE: To merge two trees, classically, Fibonacci heaps would compare the roots of the two trees and would make one the child of the other. This took constant time. Now, we perform $O(3\log\log n)$ repeated comparisons. The decrease in potential is also $O(\log\log n)$ and hence the amortized cost of a merge is $O(1)$.

- DELETE-MIN: We maintain a pointer to the minimum root and when delete-min is called, we delete the root and add all it's children to the list of root nodes. Then we consolidate all the heap-ordered trees until there is at most one tree of each rank, by using the merge operation. Since our heap doesn't support decrease-key, each heap-ordered tree we have is a perfect binomial tree and hence every root has rank $\leq O(\log n)$. Therefore, the number of roots after the consolidate operation is at most $O(\log n)$. Now, we scan all the roots to find the new minimum using the offline minimum finding algorithm of Section 3. This finishes in $O(\log n)$ time. The total real cost of delete-min is $O(\text{No. roots})\log\log n + O(\log n)$. The total change in potential is $O(\log\log n)(\log n - \text{No. roots})$. Hence the amortized cost of delete-min is $O(\log n \log\log n)$.

Now, all that's left to show is the constant error guarantee for delete-min. The probability that the minimum element isn't one of the roots of the HOTs, is upper bounded by the probability of the failure of at least one of the merge operations applied on the minimum element during the formation of the tree. Since the depth of each HOT is at most $O(\log n)$, the number of merge operations on the minimum element is at most $O(\log n)$. And the probability of failure of any single merge operation independently is $O(1/\log^3(n))$. Hence by the union bound, we achieve the a failure probability $O(1/\log^2(n))$ for the minimum element not being among the root nodes. The offline minimum finding algorithm fails to find the minimum among the root nodes with constant probability δ_2. By increasing the number of repeated comparisions in that algorithm by a multiplicative factor of 3, we achieve an error bound of $O(1/\log^2(n)$. Hence by union bound, the probability that delete-min goes wrong is $O(1/\log^2(n)$.

Appendix B.2. Sorting in $O(n \log n \log \log n)$ time

As we have said before the naive solution to the noisy sorting problem takes time $O(n \log^2 n)$. In this section we reduce the multiplicative factor of $\log n$ and to $\log \log n$. Inspired by [3] our sorting algorithm consists of two steps

- Producing a list in which $O(n/\log n)$ items are misplaced
- Sorting the above list using insertion sort

Producing a list in which $O(n/\log n)$ items are misplaced
We run a heapsort algorithm using the heap presented in the previous section. As we have seen, with $O(\log n \log \log n)$ complexity we can ensure that every time delete-min gives the minimum with probability at least $1 - 1/\log^2 n$ and so with high probability at most $O(n/\log n)$ items will be misplaced after this heap sort step. So in total this step costs $O(n \log n \log \log n)$ time. Next we formalize this argument.

Sorting the list with $O(n/\log n)$ misplaced items
To do this step, we take the list obtained in the previous part and start comparing sequentially every item i with its successor $i+1$, repeating each comparison $O(\log n)$ times, in order to get the answer of this comparison with very high probability. If $a_i \leq a_{i+1}$ then we continue with the next item. Otherwise we remove both the ith and the $i+1$th elements and we add them to a set S. After this procedures finishes (which takes $O(n \log n)$ time) we have a sorted list with $O(n(1-1/\log n))$ items and an unsorted set S with $O(n/\log n)$ items. We sort S using any standard sorting algorithm for the noisy setting, which takes $O((n/\log n)\log^2 n) = O(n \log n)$ time. So finally we have two sorted lists and we have to merge them. We do so using the merging procedure of the mergesort, but now repeating each comparison $O(\log n)$ times. The noise-free merge takes linear time and when we repeat each comparison $O(\log n)$ times it takes $O(n \log n)$ time. So in total we have $O(n \log n)$ time for noisy merge. Therefore the total running time of our algorithms is dominated by the previous part and it is $O(n \log n \log \log n)$.

Appendix B.3. Sorting in $O(n \log n \log^* n)$ time

Suppose we have a heap with $O(\log n)$ insertion, $O(\log n)$ delete-min and each time we ask for the minimum it gives the answer with constant probability of error. In this case by multiplying the number of repetitions of every comparison by $O(k)$ we can have a heap with $O(k \log n)$ insertion, $O(k \log n)$ delete-min and each time we ask for the minimum gives the answer with probability of error $O(1/(2^k))$ by applying the Chernoff bounds as in the previous sections.

Now we assume that we have a sorting algorithm with running time $O(n \log n 2^k)$ then we do the same procedure as in the previous section.

Producing a list in which $O(n/2^k)$ items are misplaced

To do this step we run a heapsort using the heap that is presented in the previous section. As we have seen there with $O(k \log n)$ complexity for delete-min we can ensure that every time we get the minimum with probability at least $1 - 1/2^k$ and so with high probability at most $O(n/2^k)$ items will be misplaced after this heap sort step. So in total this step costs $O(n \log nk)$ time. Next we formalize this argument.

Sorting the list with $O(n/2^k)$ misplaced items

As we before to do this step we take the list of the previous part and we start comparing sequentially every item i with the next item $i+1$ $O(\log n)$ times in order to get the answer of this comparison with very high probability. If $a_i \leq a_{i+1}$ then we continue with the next item. Otherwise we remove both the ith and the $i+1$th elements and we add them to a set S. After this procedures finishes, which takes $O(n \log n)$ time, we have a sorted list with $O(n(1 - 1/2^k)$ items and an unsorted set S with $O(n/2^k)$ items. We sort S using the hypothetical algorithm which takes $O((n/2^k) \log n 2^k) = O(n \log n)$ time. So finally we have to sorted lists and we have to merge them. We do so using the merging procedure of the mergesort repeating each comparison $O(\log n)$ times. The merging procedure takes linear time and when we repeat each comparison $O(\log n)$ times it takes $O(n \log n)$ time. So in total we have $O(n \log n)$ time for this procedure. Therefore the total running time of our algorithms comes from the previous part and it is $O(kn \log n)$.

The above result give that we can have an algorithm with running time $O(kn \log n)$ from an algorithm with running time $O(2^k n \log n)$ by spending an additional $O(n \log n)$ time. So by repeating this procedure $O(\log^* n)$ times we end up with an algorithm with running time $O(n \log n \log^* n)$.