Universal Sorting: Finding a DAG using Priced Comparisons

Mayank Goswami∗ Riko Jacob†

Abstract

We resolve two open problems in sorting with priced information, introduced by [Charikar, Fagin, Guruswami, Kleinberg, Raghavan, Sahai (CFGKRS), STOC 2000]. In this setting, different comparisons have different (potentially infinite) costs. The goal is to find a sorting algorithm with small competitive ratio, defined as the ratio of its cost to the cost of the cheapest proof.

1) When all costs are in \{0, 1, n, ∞\}, we give an algorithm that has \(\tilde{O}(n^{3/4})\) competitive ratio. Our result refutes the hypothesis that a widely cited \(\Omega(n)\) lower bound for finding the maximum extends to sorting. This lower bound by [Gupta, Kumar, FOCS 2000] uses costs in \{0, 1, n, ∞\} and was claimed as the reason why sorting with arbitrary costs seemed bleak and hopeless. Our algorithm generalizes the algorithms for generalized sorting (all costs are either 1 or ∞), a version initiated by [Huang, Kannan, Khanna, FOCS 2011] and addressed recently by [Kuszmaul, Narayanan, FOCS 2021].

2) We answer the problem of bichromatic sorting posed by [CFGKRS]: We are given two sets \(A\) and \(B\) of total size \(n\), and the cost of an \(A\)−\(A\) comparison or a \(B\)−\(B\) comparison is higher than an \(A\)−\(B\) comparison. The goal is to sort \(A \cup B\). We give a randomized algorithm with a \(O(\text{polylog} n)\) competitive ratio. Unlike most algorithms on generalized sorting that focus only on query complexity, the running time of our algorithm is at most \(O(n^2)\).

These results are obtained by introducing the universal sorting problem, which generalizes the existing framework in two important ways. First, we remove the promise of a directed Hamiltonian path in the DAG of all comparisons, which is assumed in all previous work on generalized sorting. Instead, we require that an algorithm outputs the transitive reduction of the DAG. For bichromatic sorting, when \(A\)−\(A\) and \(B\)−\(B\) comparisons cost ∞, this generalizes the well-known nuts and bolts problem. Second, we initiate an instance-based study of the universal sorting problem. Our definition of instance-optimality is inherently more algorithmic than that of the competitive ratio in that we compare the cost of a candidate algorithm to the cost of the optimal instance-aware algorithm. This unifies existing lower bounds, and opens up the possibility of an \(O(1)\)-instance optimal algorithm for the bichromatic version.

∗Queens College CUNY, Flushing, New York, USA. Supported by NSF grant CCF-1910873. mayank.goswami@qc.cuny.edu
†IT University of Copenhagen, København S, Denmark. Part of this work done during the second Hawaiian workshop on parallel algorithms and data structures, University of Hawaii at Manoa, Hawaii, USA, NSF Grant CCF-1930579. rikj@itu.dk
## Contents

1 Introduction and Overview
   1.1 Is the general version "bleak" and "hopeless"? .............................. 1
   1.2 The bichromatic version ......................................................... 2
   1.3 Our approach, in a nutshell ..................................................... 3
   1.4 Further related work ............................................................. 5
   1.5 Future Directions ...................................................................... 5

2 Universal Sorting and Instance Optimality: The Framework .......................... 6
   2.1 Defining Instance Optimality ...................................................... 7

3 Generalized Sorting with costs in ........................................................... 8

4 InversionSort: An Algorithm for Bipartite Sorting ..................................... 10
   4.1 Some lower bounds on OPT .......................................................... 10
   4.2 Framework for InversionSort ....................................................... 11
   4.3 Inversions and how to find them .................................................. 12
   4.4 InversionSort: Description .......................................................... 12

5 Bichromatic Sorting using InversionSort .................................................... 13

6 Sketch: Inversionsort is polylog-Instance Optimal ...................................... 14

References ................................................................................................. 16

A Proofs of Theorems 13 and Lemma 14 .................................................... 18
   A.1 Proof of Lemma 14 ................................................................. 18
   A.2 Proof of Theorem 13 ............................................................... 19

B Bipartite Sorting ...................................................................................... 21
   B.1 Instance optimality spelled out for bipartite sorting ..................... 21
   B.2 Yao’s principle ........................................................................... 21
   B.3 Verification Lower Bound .......................................................... 22
   B.4 Inversion finding bound ............................................................. 23
   B.5 Decomposition bound ............................................................... 23

C InversionSort is polylog-instance optimal for Bipartite Sorting: Proof of Theorem 21 ............................................................. 25
   C.1 Setting of the analysis ............................................................... 25
   C.2 Bounding the depth of the refinement tree .................................. 26
   C.3 Pivoting Cost Analysis ............................................................... 28
   C.4 Inversion Finding Cost Analysis ................................................ 28

D InversionSort is polylog-instance optimal for Bichromatic Sorting: Proof of Theorem 22 ............................................................. 31
   D.1 Optimality of inversion finding including monochromatic queries .... 32
   D.2 Overall proof of Theorem 22 ......................................................... 32
   D.3 Variations of bichromatic costs ................................................... 33
   D.4 More than two colors ................................................................. 33

E Bipartite Sorting with Hamiltonian .......................................................... 35
   E.1 BackboneSort: Quicksort Adaptation for Perfectly Interleaved Instance ... 35
   E.2 InversionSort for bipartite sorting with Hamiltonian .................... 35
   E.3 Allowing Matches (Equality) as the outcome of a query ................ 37

F Lower bound for finding the maximum with priced information .................. 38
1 Introduction and Overview

In their seminal paper “Query strategies for priced information”, Charikar, Fagin, Guruswami, Kleinberg, Raghavan and Sahai [8] [CFGKRS STOC 2000] study the problem of computing a function \( f \) of \( n \) inputs, where querying an input has a certain cost associated to it, and one wants to find the cheapest query strategy that computes \( f \). The competitive ratio is defined as the cost of the query strategy to the cost of the cheapest proof of \( f \). This work initiated a multitude of papers on priced information, studying problems like learning with attribute costs [18], stochastic boolean function evaluation [10], searching on trees [24, 23], priced information in external memory [6], and others.

The problem of sorting with priced information, or even a simple bichromatic version of it (stated by [CFGKRS [8]]), remains tantalizingly open. After describing their main results, [CFGKRS [8]] mention in further directions:

"Sorting items when each comparison has a distinct cost appears to be highly non-trivial. Suppose, for example, we construct an instance of this problem by partitioning the items into sets A and B, giving each A-to-B comparison a very low cost, and giving each A-to-A and B-to-B comparison a very high cost. We then obtain a very simple non-uniform cost structure in the spirit of the well-known hard problem of ‘sorting nuts and bolts’.”

One can assume that the authors in [CFGKRS [8]] were envisioning the general version of the sorting with priced information problem as: given the cost matrix for each possible comparison, or equivalently, a weighted undirected complete graph, find the cheapest sorting algorithm. To define the competitive ratio one needs to define the cheapest proof, which here is simply the sum of the costs to compare pairs of keys that are adjacent in the final sorted order. Restated, it is the cost of the directed Hamiltonian path that must be present (if the keys can be sorted) in the DAG of all possible comparisons. One immediately observes that even in the completely understood unit-cost setting, a \( \log n \) lower bound on the competitive ratio is unavoidable under this definition. We will return to this arguably unnecessary gap later.

1.1 Is the general version "bleak" and "hopeless"?

In [CFGKRS [8]], a result reported as a personal communication [[16], 2000] claimed that when different comparisons can have different (arbitrary) costs, there exists a lower bound of \( \Omega(n) \) on the competitive ratio of any algorithm that finds the maximum of a set of \( n \) elements. This was spelled out one year later by Gupta and Kumar [15], and it was stated that due to this fact the situation seems very bleak and hopeless (consequently, [15] focuses on the setting where the cost to compare two keys is well-behaved, or structured). This instance is simple to describe, and for our purposes it suffices that all comparisons in it have costs in \( \{0, 1, n, \infty\} \), as spelled out in Appendix F. Note that this example was never formally stated for sorting, yet its implication seems to have been generalized to sorting.

Apart from the structured setting studied by [15], the next version relevant to us was termed generalized sorting by Huang, Kannan, and Khanna [17], and is the version where all costs are in \( \{1, \infty\} \). Here the input is an undirected graph \( G \) on the vertices representing the keys to be sorted, and the edges of \( G \) indicate the allowed comparisons (with the \( \infty \) cost edges missing in \( G \)). It is assumed that the input can be sorted with finite cost, and so there is a
complete order on the vertices, implying the existence of a directed Hamiltonian path in the DAG $\vec{G}$ where all comparisons have been revealed. The goal is to sort the input by revealing this Hamiltonian Path while querying as few edges as possible. In the rest of this paper, by the “Hamiltonian assumption” we will mean the assumption that $\vec{G}$ contains a Hamiltonian path. [17] gave the first subquadratic algorithm for this $\{1, \infty\}$ cost setting that has a total query cost of $\tilde{O}(n^{1.5})$, or equivalently, has a competitive ratio of $\tilde{O}(\sqrt{n})$. We remark that the actual runtime of this algorithm is a high degree polynomial, as it repeatedly approximates the average rank of all vertices over all possible linear extensions of the currently revealed DAG.

The last two paragraphs naturally pose the question:

**Question 1:** Similar to the $\{1, \infty\}$ setting of generalized sorting, does there exist an algorithm with $o(n)$ competitive ratio for the $\{0, 1, n, \infty\}$ case? If so, the $\Omega(n)$ lower bound for maximum cannot generalize to sorting, and either more costs are needed, or the situation is not as bleak.

It is not hard to see that such an algorithm must be adaptive to the number of cost $n$ edges on the Hamiltonian. Furthermore, one should try to use the cost 0 and 1 edges as much as possible. This hints at the subproblem of finding the transitive reduction of the DAG defined by the 0-1 edges, even though this DAG may not have a Hamiltonian.

### 1.2 The bichromatic version

Let us now also consider the simpler bichromatic problem stated by [CFGKRS [8]] above.\(^1\) Surprisingly, no results are known for it. When $A - A$ and $B - B$ costs are $\infty$, we are in the bipartite case of the $\{1, \infty\}$ setting, and it has been claimed [15, 20] that this is equivalent to the famous nuts-and-bolts problem. In this problem one is given $n$ nuts (A) and $n$ bolts (B), is only allowed to compare a nut to a bolt (A-B), and is promised a matching between the nuts and the bolts. The goal is to find this matching. Note that the result of a comparison can be $<, >$ or $=$. The problem is originally mentioned as an exercise in [26], page 293, and a simple Quicksort type algorithm can be shown to solve this problem in $O(n \log n)$ comparisons with high probability: Pick a random nut, compare to all bolts, find the matching bolt, and compare that bolt to all nuts. The problem is now partitioned into two subproblems with the match at the boundary; recurse. In a later work by Alon, Blum, Fiat, Kannan, Naor, Ostrovsky, [3] the authors developed a sophisticated deterministic algorithm in $O(n \text{polylog } n)$ time, which was then improved to an optimal $O(n \log n)$ by Komlós, Ma, Szemerédi,[19].

We claim that this equivalence overlooks an important subtlety: Assume you have an algorithm for the “no-matches” version of nuts and bolts. If $A \cup B$ is sortable and there are no matches, then elements of $A$ and $B$ must alternate in the sorted order. It is easy to use such an algorithm to solve the nuts and bolts problem, by replacing $=$ with $<$ consistently. In contrast, it is unclear how one could use a nuts and bolts algorithm to solve the no-match

\(^1\)Observe that if the cost of the bichromatic comparisons is between the two costs for the two types of monochromatic comparisons, then this can be solved as a monotonic structured cost [15]. The case of bichromatic being most expensive is discussed in Appendix D.3. Thus the version posed in [8] where bichromatic comparisons are the cheapest is the most interesting variant of bichromatic sorting, and our focus here.
version of the problem. Indeed, the quicksort-type algorithm for the nuts and bolts problem that picks a random (nut) \( x \in A \) and compares to all (bolts) elements in \( B \) does not produce a match in the no matches version, and so the problem cannot be partitioned perfectly. One could pick a random nut and a random bolt and pivot using both, but now we get three subproblems which overlap, giving an \( n^{1+\Omega(1)} \) running time. At any rate, the simple randomized quicksort-type solution does not extend. One may resort to heavier machinery from the \( \tilde{O}(n^{1.5}) \) algorithm in [17] like probing edges that reduce the number of possible linear extensions by a constant factor (it can be shown that since the graph is bipartite, such edges always exist), but this requires computing or approximating the average rank of a vertex over all linear extensions. This gives an algorithm with \( O(n \log n) \) probes but a much higher polynomial run-time, which is decidedly dissatisfactory. It turns out that there is a simple randomized algorithm that solves this version in \( O(n \log n) \) probes with high probability. We call this algorithm BackboneSort, and it is described in Appendix E.1.

![Figure 1: An example output to an instance of the bipartite sorting problem. Continuous runs of incomparable red and blue elements are called stripes.](image)

Finally, here is an idea to solve the bichromatic version: first, use the cheap bichromatic \( A - B \) comparisons as much as possible. If elements from \( A \) (red) and \( B \) (blue) alternate perfectly in the final sorted order, this sorts the input and no monochromatic comparisons are needed. If not, one gets an ordered collection of red and blue stripes \((S_1, \cdots, S_k)\), where a stripe \( S_i \) is a set of elements of the same color\(^2\), and now use monochromatic comparisons to sort the contents of these stripes. We will call the problem of finding stripes as bipartite sorting, where the term sorting should be understood as “sort as much as you can”, i.e., the bipartite DAG may not have a Hamiltonian path. See Figure 1.

**Question 2:** Is there an efficient randomized algorithm for bipartite sorting? If so, does it extend to an algorithm for bichromatic sorting?

### 1.3 Our approach, in a nutshell

We adopt the “rising sea” metaphor attributed to the mathematician Grothendieck. Paraphrasing\(^3\), Grothendieck stated that a problem can be seen as a hard nut to crack, and when one cannot crack it by sheer force, immersing it in water for a while cracks it open effortlessly.

\(^2\) \( S_1 \) is the set of all sources in the DAG \( \vec{G} \); since red and blue elements are comparable, \( S_1 \) must necessarily be of one color. \( S_i \) can be iteratively defined as the set of sources in \( \vec{G} \) after all stripes 1 to \( i - 1 \) have been deleted.

\(^3\) Grothendieck [14], pp. 552-3-1, translated in [22] as: "I can illustrate the second approach with the same image of a nut to be opened.

The first analogy that came to my mind is of immersing the nut in some softening liquid, and why not simply water? From time to time you rub so the liquid penetrates better, and otherwise you let time pass. The shell becomes more flexible through weeks and months – when the time is ripe, hand pressure is enough, the shell opens like a perfectly ripened avocado!

A different image came to me a few weeks ago.
In other words, generalizing a problem sometimes makes the solution to the original special case apparent.

Our first insight is that generalizing generalized sorting further by removing the Hamiltonian assumption helps. While removing this assumption may seem weird (a sorted order necessitates a Hamiltonian path), this is because one wants to use the cheap comparisons as much as possible, and the progress obtained solely by exhausting cheap comparisons produces a DAG that is not complete. To this end, we redefine the required output of the algorithm from a Hamiltonian path of $\vec{G}$ to a transitive reduction of $\vec{G}$, and term the new problem universal sorting$^4$. This generalization allows us to develop the tools needed to answer Question 1 in the affirmative:

**Theorem 1.** Consider the problem of sorting when every comparison has a cost in $\{0, 1, F, \infty\}$, for any $F \geq n^{3/4}$. There exists a polynomial time randomized algorithm whose competitive ratio is $\tilde{O}(n^{3/4})$, with high probability.

In particular, the existing $\Omega(n)$ lower bound (as we spell out in Appendix F) for maximum with costs in $\{0, 1, n, \infty\}$, cannot extend to sorting.

To motivate our second insight, consider the problem of bipartite sorting: the input is a bipartite graph, but there may be no directed Hamiltonian path in it, as red and blue elements may not alternate in the final sorted order. This can also be viewed as a generalization of the nuts and bolts problem. Recall that in generalized sorting with $\{1, \infty\}$ costs, the cheapest proof of sorting is always the cost of the guaranteed directed Hamiltonian path, which is always $n - 1$. The cost of the transitive reduction, however, is highly instance-dependent. If all red elements are less than all blue elements in the sorted order, then it costs $|A||B|$, and these many bichromatic comparisons are necessary and sufficient. This necessitates a finer, instance-dependent study of the problem.

This brings us back to the unnecessary $\log n$ lower bound for the competitive ratio if we compare to the cheapest proof, even in the case of unit-cost sorting. Rather than add lower bounds (e.g., for classical sorting, one could add the entropy bound $\log(n!)$ as a lower bound) to this definition in hopes of making the problem more meaningful, we define a notion of instance optimality which is more algorithmic, where we compare with the optimal instance-aware algorithm. This has the benefit that all lower bounds derived by such a definition are honest lower bounds on an (perhaps unreasonably powerful) optimal algorithm. Note that instance optimality for static problems is different from the competitive analysis performed in the field of online algorithms, where the setting is dynamic, and one compares an online algorithm to the offline optimum that is aware of the entire sequence of future queries or updates. We then develop an an algorithm called InversionSort, that leads to the following result.

---

The unknown thing to be known appeared to me as some stretch of earth or hard marl, resist[ing] penetration... the sea advances insensibly in silence, nothing seems to happen, nothing moves, the water is so far off you hardly hear it... yet it finally surrounds the resistant substance.”

$^4$We adopt the term “universal” with some diffidence, the only justification being that generalized sorting was already taken, and generalized generalized sorting would either be too cumbersome or mistaken for a typo.
Theorem 2. For the problem of bipartite sorting, and the bichromatic sorting problem posed by [8], there exists an algorithm InversionSort that runs in $O(n^2)$ time, and is $O(\text{polylog } n)$-instance optimal, with high probability. Here $n$ is the total size of the two sets.

1.4 Further related work

As mentioned, [15] consider the setting where the cost to compare two keys is well-behaved; specifically, the cost function is monotone in the weight of the two keys being compared, examples being the sum and the product. [17] also studied the stochastic version of generalized sorting, when each edge in $G$ exists with probability $p$. Subsequent work considered settings where $G$ is dense [5], the setting where predictions are available for the edges not in $G$ [21], and the setting where the costs induce a metric space over the elements [25]. Recently, Kuszmaul and Narayanan [20] show that the complexity of stochastic generalized sorting is $O(n \log(pn))$, and the complexity of worst-case generalized sorting is $O(\sqrt{nm})$, where $m$ is the number of edges (allowed comparisons).

Incomparable is not the same as forbidden: We would also like to mention a line of work initiated by Faigle and Turán [13], (STOC 1985, SIAM Journal of Computing 1988) called sorting a partial ordered set, or identifying a poset. This was followed up by several works such as [9] and [12]. This problem asks to sort a DAG of width\(^5\) $w$ (whether $w$ is known or not does not make much of a difference), when some elements are incomparable. This may sound identical to our setting of universal sorting without the promise of a Hamiltonian (with incomparable edges costing $\infty$), but there is an important difference: the oracle for these works returns “incomparable” for certain pairs of elements, thus indicating that they must be in different chains in the DAG. In universal sorting, it is not that these elements are necessarily incomparable - they may have a directed path between each other on the DAG, but they cannot be compared directly because the cost of this comparison is $\infty$. For a survey on this line of work that also includes generalized sorting, we refer the reader to [7].

1.5 Future Directions

The obvious open question after this work is whether there still is an $\Omega(n)$ lower bound on sorting with arbitrary costs. This seems to be closely related to proving lower bounds on universal sorting. We know that such an instance would need to go beyond the $\{0, 1, n, \infty\}$ costs, and it would be surprising if universal sorting, or sorting with arbitrary costs, had $o(n)$ competitive ratio.

The polylog factors in the instance optimality for the bichromatic and bipartite problems could potentially be reduced further. We conjecture that with our notion of instance optimality, there exists an algorithm that is $O(1)$-instance optimal for bichromatic sorting. We remark that we have examples where InversionSort is not within $O(1)$ of OPT. We also remark that to get to within a constant factor of OPT one would need the entropy bound as a lower bound, which can be seen to be implied by our definitions. Finally, whether there is another hidden lower bound on OPT apart from the ones we describe is an interesting open question.

\(^5\)The width of a DAG is the size of the longest antichain, defined as a set of incomparable elements.
Roadmap: In Section 2 we define the problem statements and generalize the existing framework by introducing the problem of universal sorting, and our notion of instance optimality. Section 3 is dedicated to the proof of Theorem 1. In Section 4 we solve the bipartite version of sorting using our InversionSort algorithm, and in Section 5 show how to extend it to the bichromatic version, obtaining Theorem 2.

2 Universal Sorting and Instance Optimality: The Framework

A DAG is a directed graph $\vec{G} = (V, \vec{E})$ (without parallel edges), where $\vec{E} \subseteq V \times V$ and there is no directed cycle in $\vec{G}$. Note that, equivalently, we can demand that there is a topological sorting of $\vec{G}$.

Definition 3 (Implied and essential edge, transitive reduction [2]). Given a DAG $\vec{G} = (V, \vec{E})$, an edge $(u, v) \in \vec{E}$ is implied, if there is a directed path in $\vec{G}$ from $u$ to $v$. Otherwise, $(u, v)$ is essential. The set of essential edges is called the transitive reduction of $\vec{G}$.

Note that for every implied edge $(u, v) \in \vec{E}$, there is a directed $u$ to $v$ path of essential edges in $\vec{G}$.

Definition 4 (A DAG with costs, and its undirected version). A DAG with costs is a directed acyclic graph $\vec{G} = (\{1, \ldots, n\}, \vec{E})$ without parallel edges, and a cost function $\vec{c} : \vec{E} \to \mathbb{Q}_{\geq 0}$. Given a DAG with costs $\vec{G} = (\{1, \ldots, n\}, \vec{E})$, the undirected version of it is $G = (\{1, \ldots, n\}, E)$ with $\{(u, v) | (u, v) \in \vec{E}\}$ and cost function $c : E \to \mathbb{Q}_{\geq 0}$ where $c\{(u, v)\} := \vec{c}(\{u, v\})$.

Infinity cost queries are modeled by the edge not being in $\vec{G}$. Probing the undirected edge $\{u, v\}$ incurs a cost $c\{(u, v)\}$ and reveals its direction in $\vec{G}$.

Definition 5 (Universal Sorting). An instance $\mathcal{I}$ of universal sorting is defined by a DAG with costs $\vec{G}_\mathcal{I} = (\{1, \ldots, N\}, \vec{E})$. The Input is the undirected version $G_\mathcal{I} = (V, E)$ of $\vec{G}_\mathcal{I}$ with costs as cost graph. An Oracle answers queries: given $\{u, v\} \in E$, answer if $(u, v) \in \vec{E}$ or $(v, u) \in \vec{E}$. An algorithm $A$ solves the problem for an instance $\mathcal{I}$, if $A$, when interacting with the oracle based on $\mathcal{I}$ using queries $Q$, produces the transitive reduction of $\vec{G}_\mathcal{I}$ as output. The cost of $A$ on $\mathcal{I}$ is the total cost of the queries $c(A, \mathcal{I}) = \sum_{\{u,v\} \in Q} c(\{u,v\})$.

Definition 6 (Universal Sorting with Hamiltonian). An instance $\mathcal{I}$ of universal sorting with Hamiltonian is an instance $\mathcal{I}$ of universal sorting such that $\vec{G}_\mathcal{I}$ has a directed Hamiltonian path.

Note that universal sorting with Hamiltonian and costs in $\{1, \infty\}$ is generalized sorting as defined by [17]. Of particular interest to us will be universal sorting with Hamiltonian and costs in $\{0, 1, F, \infty\}$, where $F \geq n^{3/4}$.

Definition 7 (Bichromatic Sorting [8]). An instance $\mathcal{I}$ of universal sorting with Hamiltonian is said to be bichromatic sorting, if $G_\mathcal{I}$ is the complete graph and the costs are based on two colors: The vertices are split into two parts $R = \{1, \ldots, m\}$ (red) and $B = \{m+1, \ldots, m+n = N\}$ (blue). Red-blue edges have cost 1, red-red edges have cost $\alpha$ and blue-blue edges have cost $\beta$. 

6
We will assume that $\alpha > 1$ and $\beta > 1$. The other cases are discussed in Appendix D.3.

**Definition 8 (Bipartite Sorting).** An instance $\mathcal{I}$ of universal sorting is said to be bipartite sorting, if all costs are 1 and $G_\mathcal{I} = (V,E)$ is a complete bipartite graph, w.l.o.g., $V = R \cup B$ with $R = \{1, \ldots, m\}$ (red) and $B = \{m+1, \ldots, m+n = N\}$ (blue), and $E = R \times B$.

Similarly, we can define multipartite sorting, where $G$ is required to be a complete multipartite graph. We discuss an extension of the algorithm for bipartite sorting for this case in Appendix D.4.

### 2.1 Defining Instance Optimality

An instance optimal algorithm is traditionally defined to be one whose cost is (perhaps up to constant factors) comparable to the cost of an algorithm that is “tailor-made” for the instance. In other words, OPT is the cost of the algorithm that is told the instance that it is working on. Obviously, for our problem, if OPT knows the DAG $\tilde{G}$, then no comparisons are needed, and it can just output the transitive reduction, incurring 0 comparison costs (as input reading and output writing are not part of the comparison cost).\(^6\) For convex hull algorithms, instance optimality has been introduced by Afshani, Barbay, and Chan [1] by comparing to the cost of the algorithm that knows the geometry of the convex hull, but not which point appears at which position in the input. The following definition is in this spirit.

**Definition 9.** The automorphism neighborhood of an instance $\mathcal{I}$ is defined as

$$\mathcal{N}_A(\mathcal{I}) = \{\mathcal{I}' \mid G_{\mathcal{I}'} = G_\mathcal{I} \land \tilde{G}_{\mathcal{I}'} \text{ is cost-isomorphic to } \tilde{G}_\mathcal{I}\}$$

Consider the instance $\mathcal{I}$ of bipartite sorting where all reds are smaller than all blues. Intuitively, we would not accept an algorithm as correct, if it did not probe all edges in this instance. But $\mathcal{N}_A(\mathcal{I}) = \{\mathcal{I}\}$, which allows an optimal algorithm with cost zero. To circumvent this, and similar situations, the following neighborhood is useful.

**Definition 10 (Essential-Flip neighborhood of an instance $\mathcal{I}$).** Let $\tilde{G} = (V, \tilde{E})$ be a DAG and $e = (u,v) \in \tilde{E}$ an edge. Define the flip of $\tilde{G}$ with respect to $e$ as $\tilde{G}^e = (V, \{(v,u)\} \cup \tilde{E}' \setminus \{(u,v)\})$. Define the essential-flip neighborhood of an instance $\mathcal{I}$ as

$$\mathcal{N}_E(\mathcal{I}) = \{\mathcal{I}' \mid \tilde{G}_{\mathcal{I}'} = \tilde{G}^F_\mathcal{I} \text{ for some } F \subseteq \tilde{E} \text{ such that all edges in } F \text{ are essential, and } \tilde{G}^F_\mathcal{I} \text{ is a DAG}\}$$

**Definition 11.** The Automorphism-Essential (AE) neighborhood of an instance $\mathcal{I}$ is defined as $\mathcal{N}_{AE}(\mathcal{I}) = \bigcup_{\mathcal{I}' \in \mathcal{N}_E(\mathcal{I})} \mathcal{N}_A(\mathcal{I}')$

We then take OPT to be the worst-case cost of a neighborhood-aware optimal algorithm on any instance in this AE neighborhood, i.e.,

---

\(^6\)This issue exists for any static problem. The competitive ratio is a well-established notion for dynamic or online problems, where the input is revealed piecemeal to the algorithm. It is defined as the ratio of the cost of the online algorithm to the cost of an optimal algorithm that knows the complete input from the start. This has also been called instance optimality, for example in the dynamic optimality conjecture. The term competitive ratio has also been used for sorting, and for other problems regarding querying with priced information. We consider this choice of words somewhat misleading, due to the static nature of these problems.
Definition 12 (Instance Optimal Solution).

\[
\text{OPT}^{N_{AE}}(I) = \inf_{A_{\varepsilon} \in A_{\varepsilon}} \max_{I' \in N_{AE}(I)} \mathbb{E}_{A_{\varepsilon}}[\text{cost}(A_{\varepsilon}, I')]
\]

Here, \(A_{\varepsilon}\) denotes the set of randomized algorithms that are correct with probability \(1 - \varepsilon\) on every instance in \(N_{AE}(I)\), and the expectation is over the random choices of the algorithm. If \(\varepsilon\) is omitted, we assume \(\varepsilon = 1/4\), and use \(\text{OPT}(I) := \text{OPT}^{N_{1/4}}_{AE}(I)\).

An algorithm \(A\) is said to be \(c\) instance optimal if \(\text{cost}(A, I) \leq c \cdot \text{OPT}(I)\).

We remark that algorithms in \(A_{\varepsilon}\) are allowed to have their workings depend on \(I\); we can therefore think of them as being given \(I\) as a parameter, and they only have to succeed on instances near \(I\). Obviously, it is meaningful to instantiate this notion of instance optimality for different notions of neighborhoods. Then, the smaller the neighborhood, the more difficult it is for an algorithm to achieve a good instance-optimality factor. As we have seen, if the neighborhood consists only of a single instance, the factor is infinite because the cost of \(\text{OPT}\) is 0. On the other hand, if the neighborhood consists of all instances, this definition turns into classical worst-case analysis.

This step of phrasing the quality of an algorithm relative to an optimal algorithm on a relatively small set of input instances avoids the ad-hoc nature of comparing to some combinatorial lower bound of the instance.

3 Generalized Sorting with costs in \(\{0, 1, F, \infty\}\): Theorem 1

In this section we consider universal sorting with the promise of a Hamiltonian, and extend the allowed costs in the generalized sorting problem by [17], from \(\{1, \infty\}\) to \(\{0, 1, F, \infty\}\), where \(F\) is any value greater than \(n^{3/4}\). Our goal is to obtain an algorithm with a competitive ratio of \(\tilde{O}(n^{3/4})\).

We set up some notation first. Let \(\tilde{G}\) denote the true DAG that contains a directed Hamiltonian path. Define \(\tilde{G}_0\) as the DAG obtained by revealing all cost 0 edges, and let \(w_0\) be the width of \(\tilde{G}_0\). Similarly, denote by \(\tilde{G}_{01}\) the DAG obtained by revealing all cost 0 and 1 edges. Clearly, \(\tilde{G}_{01}\) may not have a Hamiltonian path. Let \(k_1\) and \(k_F\) be the number of cost 1 and cost \(F\) edges on the Hamiltonian path in \(\tilde{G}\). Denote by \(w_{01}\) the width of \(\tilde{G}_{01}\).

Note that we always have the option of running the algorithm in [17] that works on costs \(\{1, \infty\}\). If \(k_F = 0\), running this algorithm while pretending \(F = \infty\) gives cost at most \(O(n^{1.5} \log n)\). On the other hand, if \(k_F \geq 1\), then we can run this algorithm pretending \(F = 1\) and its true cost is bounded by \(O(F n^{1.5} \log n)\). We don’t know \(k_F\), but we can run both versions in parallel. The next theorem gives another algorithm for computing \(\tilde{G}_{01}\).

Theorem 13 (Universal Sorting, computing 0-1 DAG). There exists a randomized polynomial time algorithm that computes the transitive reduction of \(\tilde{G}_{01}\) and has query cost in \(O(\min(n^{1.5} \log n, w_0 k_1 \log n))\) if \(k_F = 0\), else if \(k_F \geq 1\), it has query cost in \(O(w_{01} n^{1.5} \log n)\).

We defer the proof of Theorem 13 for now, and first show how the algorithm in Theorem 13 can be extended to handle the \(\{0, 1, F, \infty\}\) case. We will use the following algorithm that
finds a Hamiltonian path in a partially revealed DAG, which is also used in the proof of Theorem 13. For two DAGs $D'$ and $D$ on the same set of vertices, we will write $D' \subset D$ if all the directed edges in $D'$ are also contained in $D$.

**Lemma 14** (Hamiltonian by predecessor search). Let $D' \subset D$ be two DAGs on the vertex set $V$ and assume that $D$ contains a Hamiltonian path. Assume that the Hamiltonian path in $D$ contains a set $\vec{S}$ of $k$ edges that are not in $D'$, and let $S$ be the undirected version of $\vec{S}$. Let $E$ be a set of edges that can be queried and assume $S \subset E$. Let $w$ be the width of $D'$. Then, $k + 1 \geq w$ and the Hamiltonian in $D$ can be found with cost at most $O(wk \log n)$ queries on edges in $E$.

This search explains the $w_0k_1 \log n$ term in the case when $k_F = 0$ in Theorem 13 as follows. We first query all cost 0 edges, obtaining $\vec{G}_0$. Since $k_F = 0$, the width $w_0$ of $\vec{G}_0$ is at most $k_1 + 1$. We next probe only cost 1 edges in the algorithm of Lemma 14 with $D' = \vec{G}_0$, which implies that the Hamiltonian can be found with cost at most $O(w_0k_1 \log n)$. However, we abort this algorithm once it has incurred a total query cost of $O(n^{1.5} \log n)$. This is because our algorithm for universal sorting runs in time $O(w_0n^{1.5} \log n)$ (we develop this in the proof of Theorem 13), which for $w_0 = 1$ may incur fewer queries than this predecessor search.

After obtaining $\vec{G}_0$ from Theorem 13, we first check if it contains a Hamiltonian path. If not, we run the algorithm in Lemma 14 with $D' = \vec{G}_0$, which probes at most $O(w_0k_F \log n)$ cost $F$ edges. Recall that we are also running in parallel the algorithm in [17]. Thus we get an algorithm for the $\{0, 1, F, \infty\}$ case that has query cost at most

\[
\begin{cases} 
O(\min(n^{1.5} \log n, w_0k_1 \log n)) & \text{if } k_F = 0 \\
O(\min(Fn^{1.5} \log n, w_0n^{1.5} \log n + Fw_0k_F \log n)) & \text{if } k_F > 0
\end{cases}
\]

We claim that the competitive ratio is always bounded by $O(n^{3/4} \log n)$. Observe that the cost of the Hamiltonian is $k_1 + Fk_F$.

Consider the case $k_F = 0$ first. Note that this implies that the width $w_0$ of $\vec{G}_0$ is 1. First consider the subcase when $w_0 \leq n^{3/4}$. In this case, the competitive ratio is bounded above by $O(w_0k_1 \log n) / k_1 = O(w_0 \log n) \leq O(n^{3/4} \log n)$. In the subcase when $w_0 > n^{3/4}$, observe that this implies that $k_1 \geq n^{3/4}$ which implies that the competitive ratio is bounded above by $O(n^{1.5} \log n) / k_1 \leq O(n^{3/4} \log n)$.

Now consider the case $k_F \geq 1$, and the cost of the Hamiltonian is at least $Fk_F$. Since $w_0 \leq k_F + 1$, the cost of the algorithm is at most $O(w_0n^{1.5} \log n + Fw_0Fk_F \log n) < O(k_Fn^{1.5} \log n + Fw_0k_F \log n)$, and dividing by $FK_F$ (the lower bound on the cost of the Hamiltonian), we get a competitive ratio of at most $O((n^{1.5} / F + w_0) \log n)$. Since $F \geq n^{3/4}$, this ratio is $O(n^{3/4} \log n)$ as long as $w_0 \leq n^{3/4}$. Else if $w_0 > n^{3/4}$, we observe that $k_F \geq n^{3/4}$, and then the $Fn^{1.5} \log n$ query cost gives us a competitive ratio of at most $Fn^{1.5} \log n / Fk_F \leq n^{3/4} \log n$. This completes the proof of Theorem 1.

It remains to prove Theorem 13 and Lemma 14. Both proofs can be found in Appendix A, and we briefly sketch them here.

Lemma 14 generalizes binary search to searching for predecessors of a vertex in a DAG of width $w$. Binary searching for a vertex $v$ into one of the $w$ chains takes $O(\log n)$ probes, and in $O(w \log n)$ probes one is sure to have at least discovered one edge from the Hamiltonian, namely the incoming edge to $v$. This can then be repeated $k$ times, revealing the Hamiltonian.
Theorem 13 generalizes the algorithm of [17], which we briefly summarize. At a high level, the algorithm alternates between three ways of making progress:

1. Finding and probing balanced edges, defined as those that reduce the number of possible linear extensions of the current DAG by a constant factor. Finding such edges requires approximating the average rank of vertices under all possible linear extensions at all stages of the algorithm.

2. After estimating the indegree of vertices up to an additive error of $\tilde{O}(\sqrt{n})$ by a $\tilde{O}(n^{1.5})$ sampling procedure, probing free edges, defined as the set of edges $\langle u, v \rangle$ where the average rank of $u$ is smaller than the average rank of $v$, and $v$ has most $\tilde{O}(\sqrt{n})$ unprobed incoming edges. Free edges that are balanced again reduce the number of linear extensions by a constant factor. Otherwise, they can contribute at most $\tilde{O}(n^{1.5})$ to the total cost.

3. Binary Search - When there are no free edges, there must exist a set of $\sqrt{n}$ vertices with known total order (Lemma 3.5 in [17]). The other vertices can perform binary search into these $\sqrt{n}$ vertices at a cost of $O(n \log n)$, and doing so removes these $\sqrt{n}$ vertices from the picture. The total cost of binary search is therefore $\tilde{O}(n^{1.5})$.

Our algorithm for Theorem 13 differs mainly in the third step, because the existence of a set of $\sqrt{n}$ vertices with known total order relies upon the existence of the Hamiltonian. We generalize this lemma in the case of a DAG without the Hamiltonian, and then invoke our algorithm for Lemma 14 to replace the binary search procedure.

4 InversionSort: An Algorithm for Bipartite Sorting

In this section we will present our algorithm InversionSort for bipartite sorting (Definition 8), and prove that it is instance optimal up to polylogarithmic factors. Before we describe our algorithm, we state some lower bounds that follow from our definition of instance optimality (Definition 12), which will be useful in proving the guarantee on InversionSort.

4.1 Some lower bounds on OPT

All lower bounds are restated in Appendix B together with their proofs. First, we use Yao’s principle to switch from randomized algorithms to deterministic algorithms and distributions of input.

Lemma 15 (Yao’s principle for OPT on a neighborhood $\mathcal{N}(\mathcal{I})$ of instance $\mathcal{I}$).

$$OPT(\mathcal{N}(\mathcal{I})) \geq \Omega \left( \sup_{D \text{ distribution over } \mathcal{N}(\mathcal{I})} \inf_{A \in A_{\text{det}}, \text{ failure prob. } \leq 1/4} \mathbb{E}_{\mathcal{I}' \in D} [\text{cost}(A, \mathcal{I}')] \right)$$

Again, note that $A_{\text{det}}$ is allowed to depend on $\mathcal{I}$. The following lower bounds utilize Yao’s principle. Observe that while these lower bounds would be easy to prove for algorithms unaware of $\mathcal{I}$ using adversary arguments, we prove this for OPT, which requires some extra
care. All instances mentioned from now on in this section are instances of bipartite sorting, unless mentioned otherwise.

**Lemma 16** (Verification lower bound). Let $\mathcal{I}$ be an instance of bipartite sorting, let $K \subset \overline{E}_\mathcal{I}$ be its transitive reduction, and define $C_V = |K|$. There exists a distribution $\mathcal{D}$ over $\mathcal{N}_{E}(\mathcal{I}) \subset \mathcal{N}_{AE}(\mathcal{I})$ such that any deterministic algorithm $\mathcal{A}$ that does at most $C_V/2$ comparisons must fail with probability (at least) $1/4$.

**Lemma 17** (Inversion finding lower bound). Let $\mathcal{I}$ be an instance of bipartite sorting with $n \geq 2$ red and $m \geq 2$ blue elements, where not all comparisons come out the same, and define

$$C_I = \frac{nm}{\min(|\{(r,b) \in R \times B \mid r < b\}|, |\{(r,b) \in R \times B \mid r > b\}|)}.$$  

There exists a distribution $\mathcal{D}$ over $\mathcal{N}_{A}(\mathcal{I})$ such that any deterministic algorithm $\mathcal{A}$ that does at most $C_I/2$ comparisons must fail with probability (at least) $1/4$.

Let $\mathcal{I}$ be an instance and $(S_1, \cdots, S_k)$ its stripes (see Figure 1, and Definition 31). Consider pairs of indices $(a_1, b_1), \cdots, (a_\ell, b_\ell)$, where for all $1 \leq j \leq \ell$, $a_j$ and $b_j$ both belong to $\{1, \cdots, k\}$, and $a_j < b_j < a_{j+1} < b_{j+1}$ for all $j < \ell$. For $1 \leq j \leq \ell$ define the subinstance $\mathcal{I}_j$ by the subgraph of $\overline{G}_\mathcal{I}$ on the vertices $V_j = \bigcup_{i=a_j}^{b_j} S_i$.

**Lemma 18** (Decomposition into Lower Bounds for Subproblems). Consider any $0 \leq \epsilon \leq 1/2$ and for $1 \leq j \leq \ell$, let $\mathcal{I}_j$ be a subinstance of $\mathcal{I}$ as above. Then $\text{OPT}(\mathcal{I}) \geq \sum_{j=1}^\ell \text{OPT}(\mathcal{I}_j)$, where both $\text{OPT}(\mathcal{I})$ and $\text{OPT}(\mathcal{I}_j)$ are allowed to err on any input in $\mathcal{N}_{AE}(\mathcal{I})$ and $\mathcal{N}_{AE}(\mathcal{I}_j)$, respectively, with probability at most $\epsilon$.

### 4.2 Framework for InversionSort

We first develop some notation to better explain InversionSort. For simplicity, we artificially add a smallest red element and a largest blue element to the input. A generic state of InversionSort will be defined using a *backbone*, which is a sequence of elements of alternating colors, called *representatives*. Each representative will be assigned a *bucket*, which is a set of elements of the same color, that lie between the two neighboring representatives of the other color on the backbone.

InversionSort gets from one state to the next by performing three steps: a) finding an *inversion* (which we define soon; for now we think of an inversion as a bichromatic pair of elements) between neighboring representatives on the backbone, b) inserting this pair on the backbone, and c) pivoting with these two elements, thereby refining the buckets.

**Definition 19** (Backbone, Representatives, and Buckets). The backbone consists of a totally ordered, alternating list of representatives $(u_0, u_1, u_2, \ldots, u_{2k+1}) = (r_0, b_1, r_2, \ldots, r_{2k}, b_{2k+1})$, where $r_{2i} \in R$ and $b_{2j+1} \in B$ with $r_i < b_{i+1}$ and $b_i < r_{i+1}$. Here, $r_0$ is an artificial red element that is smaller than all blue elements, and the last element $b_{2k+1}$ is an artificial blue element that is larger than all red elements. The representatives define the buckets $(X_0, X_1, X_2, \ldots, X_{2k}, X_{2k+1}) = (R_0, B_1, R_2, \ldots, R_{2k}, B_{2k+1})$ by $R_i = \{x \in R \mid b_{i-1} < x < b_{i+1}\} \setminus \{r_i\}$ and $B_i = \{x \in B \mid r_{i-1} < x < r_{i+1}\} \setminus \{b_i\}$. Here, as a convention, the representative is not included in the bucket. Again, $R_0 = \{x \in R \mid x < b_1\}$ and $B_{2k+1} = \{x \in B \mid r_{2k} < x\}$ are special cases.
From now on, we will use $u_i$ and $X_i$ when we don’t care about the color of the $i$th representative and bucket, otherwise we will switch to $r_i/b_i$ and $R_i/B_i$. We observe that the buckets of two non-adjacent representatives on the backbone are ordered in the same way as the representatives:

**Lemma 20.** Let $X_i$ and $X_j$ be buckets (of arbitrary color) and assume $i + 1 \leq j - 1$. Then for all $x \in X_i$ and $y \in X_j$ we have $x < y$.

**Proof.** By definition, $x < u_{i+1}$, and $u_{j-1} < y$. If $i + 1 < j - 1$, then $u_{i+1} < u_{j-1}$ is implied by transitivity on the backbone, otherwise $u_{i+1}$ is the same element as $u_{j-1}$. In either case, we get that $x < u_{i+1} \leq u_{j-1} < y$. \hfill \qed

### 4.3 Inversions and how to find them

We now define an inversion, which gives the algorithm its name. Consider adjacent representatives $u_i$ and $u_{i+1}$, their corresponding adjacent buckets $X_i$ and $X_{i+1}$, and a bichromatic pair $(x, y)$ of elements $x \in X_i$ and $y \in X_{i+1}$. Unlike in Lemma 20, $x$ and $y$ are not ordered by transitivity of the backbone. Because $x$ and $y$ are of different color, they can be compared directly. If $y < x$, we call the pair an inversion. This is because it allows us to extend the backbone: we get $u_i < y < x < u_{i+1}$, which is a chain of actual comparisons between elements of alternating color.

The simplest way to find an inversion is to uniformly at random, from all pairs in $X_i$ and $X_{i+1}$, pick $x$ and $y$. If the fraction of inversions is $p$, then the probability of finding an inversion is $p$ and the expected number of trials to find one is $1/p$.

This simple algorithm only becomes problematic if there are no inversions, which is the situation where this inversion-finding procedure stops. For each pair of neighboring pivots $u_i$ and $u_{i+1}$ where we do inversion finding, we keep a counter of the number of random pairs that already have been compared between associated buckets $X_i$ and $X_{i+1}$. We don’t reset the counter in case $X_i$ or $X_{i+1}$ change (they might get smaller because of pivoting in the neighborhood). If the counter increases above $|X_i||X_{i+1}|$ attempts (without finding an inversion), we just do all comparisons with a doubly nested loop. This actually completely identifies the subinstance between $u_i$ and $u_{i+1}$: if there are any inversions, the algorithm finds all, and arbitrarily creates a consistent backbone with buckets, and remembers that there are no further inversions. Note that this strategy can at worst do 2 times the comparisons than what is the expected number of comparisons when uniformly drawing a pair until we find an inversion. It also never performs more than $2|X_i||X_{i+1}|$ comparisons, where $|X_i|$ and $|X_{i+1}|$ are the final sizes of the buckets if there is no further inversion.

### 4.4 InversionSort: Description

InversionSort starts by (trivially) having the backbone consist only of the artificial smallest red element $r_0$ and largest blue element $b_1$, and $R_0 = R$ and $B_1 = B$.

For a given backbone $(u_0, u_1, u_2, \ldots, u_{2k}, u_{2k+1}) = (r_0, b_1, r_2, \ldots, r_{2k}, b_{2k+1})$, InversionSort first, for each pair $X_i, X_{i+1}$ of adjacent buckets that have not yet found an inversion, in a round-robin manner, draws one additional pair (or finishes by doing all pairs of comparisons
as explained above), saves this pair if it is an inversion, and moves to the next pair of adjacent buckets.

This can lead to new inversions that are used to extend the backbone. We wait for the round to finish before extending the backbone. Assume that we found the pair \((x, y)\) that forms an inversion between \(u_i\) and \(u_{i+1}\). For the moment, assume that no inversions were found between \(u_{i-1}\) and \(u_i\), or between \(u_{i+1}\) and \(u_{i+2}\). Then, after inserting this pair into the backbone, we have \(u_i < y < x < u_{i+1}\). We now pivot all elements in \(X_i\) with \(y\) and pivot all elements in \(X_{i+1}\) with \(x\). After pivoting, \(y\) partitions \(X_i\) into \(X_i^{\text{left}}\) and \(X_i^{\text{right}}\) and similarly \(x\) partitions \(X_{i+1}\) into \(X_{i+1}^{\text{left}}\) and \(X_{i+1}^{\text{right}}\). The new (representative,bucket) pairs on this part of the backbone are now as follows: \((u_i, X_i^{\text{left}}), (y, X_i^{\text{left}}), (x, X_i^{\text{right}}), (u_{i+1}, X_{i+1}^{\text{right}})\).

If in this round inversions were also found between \(u_{i-1}\) and \(u_i\) (say a pair \((x', y')\) such that \(u_{i-1} < y' < x' < u_i\)), or between \(u_{i+1}\) and \(u_{i+2}\) (say a pair \((x'', y'')\) such that \(u_{i+1} < y'' < x'' < u_{i+2}\)), or both, we also pivot \(X_i\) (or rather \(X_i^{\text{left}}\)) with \(x'\) (in addition to \(y\)) and \(X_{i+1}\) (or \(X_{i+1}^{\text{right}}\), depending on the order in which the pivoting is done) with \(y''\) (in addition to \(x\)). Thus, each bucket is pivoted against at most two new pivots.

This reestablishes the backbone and creates some new pairs of neighboring buckets, for which we start the next round of inversion finding procedures.

The algorithm stops once all neighboring pairs of buckets are shown to not have an inversion, i.e., the inversion finding procedure finishes all comparisons unsuccessfully. In this case, the current buckets form the stripes of the instance. The algorithm outputs these buckets in order, which is equivalent to producing the transitive reduction of the directed graph of the instance. Since, by convention, we chose representatives not to be part of their buckets, the algorithm includes the representatives in the bucket in the output (omitting the two artificial extreme elements).

**Theorem 21** (Instance Optimality of InversionSort). There exists a constant \(c > 0\), such that for every instance \(I\), the cost of InversionSort on \(I\) is, with probability at least \(1 - 1/n\), at most \(c(\log n)^4 \text{OPT}(I)\), where \(\text{OPT}(I)\) is as in Definition 12.

The proof of Theorem 21 is sketched in Section 6 and spelled out in Appendix C. We next show how to use InversionSort to get an algorithm for bichromatic sorting.

## 5 Bichromatic Sorting using InversionSort

We show how to extend InversionSort when monochromatic comparisons are allowed. The highlevel algorithm is simple: first, we run a slightly adapted version of the InversionSort algorithm to identify the DAG of bichromatic comparisons, as detailed below. Then we sort the resulting stripes using merge sort with monochromatic comparisons.

In InversionSort, when doing inversion-finding, for every \(\alpha\) bichromatic comparisons, we do one red-red-comparison, and for every \(\beta\), we do one blue-blue. These comparisons are used to maintain the max / min of a sample of elements in the bucket as follows. Choose a random non-sampled element in the bucket and compare it to the max / min. If this leads to a new max/min, compare to the min or max of the opposing bucket. If this shows an inversion, arbitrarily choose between the max and the min, pivot with that, and choose a random element from the new bucket formed as its representative on the backbone (note that
two buckets will be formed, but one already has a representative). Also, for every red-blue comparison where both elements are randomly chosen, we do a comparison of one random element with the max and a comparison of one random element with the min (unless the sample is still empty). We call this a triple of red-blue comparisons.

Unlike in the bipartite setting, where the only proof that there is no further inversion, is to perform all pairwise comparisons between two stripes, here, in the bichromatic setting, other proofs might be cheaper, as spelled out in the four cases below. For two buckets with $A$ red and $B$ blue elements, w.l.o.g. red smaller than blue, there are four proofs:

1. all $AB$ red-blue comparisons at total cost $AB$.

2. $A - 1$ red-red comparisons to find the maximal red, and $B$ red-blue comparisons with this maximum, total cost $\alpha(A - 1) + B$.

3. $B - 1$ blue-blue comparisons to find the minimal blue, and $A$ red-blue comparisons with this minimum, total cost $\beta(B - 1) + A$.

4. $A - 1$ red-red comparisons to find the maximal red, $B - 1$ blue-blue comparisons to find the minimal blue, and 1 red-blue comparison between the maximum and the minimum, total cost $\alpha(A - 1) + \beta(B - 1) + 1$. (This case is never the cheapest if $\alpha, \beta > 1$.)

Observe that the cost of these proofs only depends on the sizes of the stripes/buckets. Hence, InversionSort computes these values and chooses the cheapest. If the accumulated cost of randomized inversion finding exceeds this cost of the cheapest proof, we deterministically perform all comparisons required by the proof. If this indeed shows that there are no further inversions, this subproblem is finished. Else, this attempt to establish a proof must find an inversion, and, like in the randomized inversion finding procedure, we randomize such that at least one pivot / backbone element is chosen uniformly among the eligible inversions (leading to Lemma 36).

These changes to InversionSort require some adjustments to its analysis. These are presented in Appendix D, leading to the following result.

**Theorem 22** (Instance Optimality of Bichromatic InversionSort). There exists a constant $c > 0$, such that for every instance $I$, the cost of bichromatic InversionSort on $I$, with probability at least $1 - 1/n$, is at most $c(\log n)^4 \text{OPT}(I)$, where $\text{OPT}(I)$ is the cost of the Hamiltonian.

6 **Sketch of the Proof that Inversionsort is $O((\log N)^4)$-Instance Optimal**

This analysis constitutes the most technically challenging part of the paper. Let us visualize a run of Inversionsort as a ternary tree, with nodes corresponding to subproblems. For an internal node $v$, there is a corresponding subinterval on the backbone defined by two consecutive pivots, say a blue pivot followed by a red pivot, $b_v < r_v$. If Inversionsort finds an inversion $y < x$ ($x$ is blue and $y$ is red) between $b_v$ and $r_v$, then $v$ has three children with the respective pivots $(b_v, y)$, $(y, x)$, $(x, r_v)$. Note that at a snapshot somewhere during
a run of Inversionsort, the tree explored so far may be far from being a complete tree - Inversionsort could be stuck on a large-sized problem in one region, while refining and working way down a descendant of an interval in another region. Hence Inversionsort does not proceed layer-by-layer on this tree.

We first prove an important property of the inversion \((y, x)\): conditioned on \(x\) being in the inversion, \(y\) is equally likely to be any red element in the range \((b_v, x)\). Similarly, conditioned on \(y\) being in the inversion, \(x\) is equally likely to be any blue element in the range \((y, r_v)\). This observation can be used to show that the Inversionsort tree has depth \(O(\log N)\), with high probability. Here \(N = n + m\).

As a second challenge in the analysis, because of the overlapping nature of the problem, Inversionsort has to do more than what is indicated just by the subinstance between neighboring pivots. For example, for the child indicated by pivots \((y, x)\), instead of only getting the reds and blues that actually lie in this range as input, Inversionsort instead has to also work with the red elements contained in \((b_v, y)\) and the blue elements inside \((x, r_v)\). This “spill-in” from the neighboring subintervals on the backbone needs to be analyzed. See Figure 2 for an example. Thus we distinguish between subinstances and subproblems.

![Figure 2: The difference between a subproblem and a subinstance.](image)

**Figure 2:** The difference between a subproblem and a subinstance. \(b_v\) and \(r_v\) are on the backbone, and \(y < x\) is the inversion obtained between them. The stripes in the subinstance \((y, x)\) are solid, whereas the stripes in the subproblem \((y, x)\) are the union of the solid and shaded stripes.

Having derived lower bounds (Section 4.1) on the instance \(I\) and its subintervals, we would ideally like to charge the work done by Inversionsort on a subproblem to the lower bound given by the corresponding subinterval. However, if the spill-in for this subproblem is too large, these two quantities of work are not comparable. We then identify subproblems that do not have too much spill-in from their neighbors - we call these subproblems unaffected. Unaffected subproblems can easily be charged to their subintervals.

To take care of the affected subproblems, as spelled out in detail in Appendix C.4, we show that at any time, with high probability, roughly a \(1/(\log N)^2\) fraction of all problems
are unaffected. This requires a careful accounting of how the unaffected nodes are distributed in the tree.

**Putting everything together** As mentioned above, a \((\log N)^2\) factor appears while accounting (with high probability) for the affected nodes on one snapshot of the backbone. Accounting over the whole tree introduces another \(\log N\) factor corresponding to the depth of the tree. Finally, when Inversionsort tries to find an inversion, it randomly picks pairs of elements to compare, without keeping track of the comparisons it has already done. This makes it run into a coupon-collector type of situation, which introduces and extra \(\log N\). If we are allowed quadratic space and some extra time for bookkeeping, this issue can be avoided to reduce the final guarantee from a \((\log N)^4\) to a \((\log N)^3\).

**References**

[1] Peyman Afshani, Jérémy Barbay, and Timothy M Chan. Instance-optimal geometric algorithms. *Journal of the ACM (JACM)*, 64(1):1–38, 2017.

[2] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[3] Noga Alon, Manuel Blum, Amos Fiat, Sampath Kannan, Moni Naor, and Rafail Ostrovsky. Matching nuts and bolts. In *Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete Algorithms (SODA’94)*, pages 690–696, 1994.

[4] Hans-Jürgen Bandelt and Victor Chepoi. Metric graph theory and geometry: a survey. In *Surveys on Discrete and Computational Geometry: Twenty Years Later*, editors, Jacob E. Goodman and János Pach and Richard Pollack, volume 453 of *Contemporary Mathematics*, pages 49–86. AMS, 2008.

[5] Indranil Banerjee and Dana Richards. Sorting Under Forbidden Comparisons. In *15th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2016)*, editors, Rasmus Pagh, volume 53 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[6] Michael A Bender, Mayank Goswami, Dzejla Medjedovic, Pablo Montes, and Kostas Tsichlas. Batched predecessor and sorting with size-priced information in external memory. In *Latin American Symposium on Theoretical Informatics*, pages 155–167. Springer, 2021.

[7] Jean Cardinal and Samuel Fiorini. On generalized comparison-based sorting problems. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 164–175. Springer, 2013.

[8] Moses Charikar, Ronald Fagin, Venkatesan Guruswami, Jon Kleinberg, Prabhakar Raghavan, and Amit Sahai. Query strategies for priced information. *Journal of Computer and System Sciences*, 64(4):785–819, 2002.
[9] Constantinos Daskalakis, Richard M Karp, Elchanan Mossel, Samantha J Riesenfeld, and Elad Verbin. Sorting and selection in posets. *SIAM Journal on Computing*, 40(3):597–622, 2011. 5

[10] Amol Deshpande, Lisa Hellerstein, and Devorah Kletenik. Approximation algorithms for stochastic boolean function evaluation and stochastic submodular set cover. In *Proceedings of the twenty-fifth annual ACM-SIAM Symposium on Discrete Algorithms (SODA’14)*, pages 1453–1466. SIAM, 2014. 1

[11] Robert P Dilworth. A decomposition theorem for partially ordered sets. In *Classic Papers in Combinatorics*, pages 139–144. Springer, 2009. 18

[12] Devdatt P Dubhashi, Kurt Mehlhorn, Desh Ranjan, and Christian Thiel. Searching, sorting and randomised algorithms for central elements and ideal counting in posets. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 436–443. Springer, 1993. 5

[13] Ulrich Faigle and Gy Turán. Sorting and recognition problems for ordered sets. *SIAM Journal on Computing*, 17(1):100–113, 1988. 5

[14] Alexandre Grothendieck. *Récoltes et semailles*. éditeur non identifié, 1992. 3

[15] Anupam Gupta and Amit Kumar. Sorting and selection with structured costs. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science (FOCS’01)*, pages 416–425. IEEE, 2001. 1, 2, 5, 33, 38

[16] J Hartline, E Hong, A Mohr, E Rocke, and K Yasuhara. Personal communication. In , 2000. 1

[17] Zhiyi Huang, Sampath Kannan, and Sanjeev Khanna. Algorithms for the generalized sorting problem. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS’11)*, pages 738–747, 2011. 1, 2, 3, 5, 6, 8, 9, 10, 19, 20

[18] Haim Kaplan, Eyal Kushilevitz, and Yishay Mansour. Learning with attribute costs. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing (STOC’05)*, pages 356–365, 2005. 1

[19] János Komlós, Yuan Ma, and Endre Szemerédi. Matching nuts and bolts in O(n log n) time. *SIAM Journal on Discrete Mathematics*, 11(3):347–372, 1998. 2

[20] William Kuszmaul and Shyam Narayanan. Stochastic and worst-case generalized sorting revisited. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS’22)*, pages 1056–1067. IEEE, 2022. 2, 5

[21] Pinyan Lu, Xuandi Ren, Enze Sun, and Yubo Zhang. *Generalized Sorting with Predictions*, pages 111–117. SIAM, 2021. 5

[22] Colin McLarty. *The rising sea: Grothendieck on simplicity and generality*. na, 2003. 3
Since Lemma 14 is used both in the proof of Theorem 13, and after running the algorithm in Theorem 13, we prove it first. We will use a classical result by Dilworth; recall that an antichain is a set of incomparable elements.

**Theorem 23** (Dilworth’s Theorem [11]). In any finite partially ordered set, the size of the largest antichain is equal to the number of the chains in the smallest chain decomposition.

For a DAG, these numbers will be called the width of the DAG.

### A.1 Proof of Lemma 14

**Lemma 14** (Hamiltonian by predecessor search). Let $D' \subset D$ be two DAGs on the vertex set $V$ and assume that $D$ contains a Hamiltonian path. Assume that the Hamiltonian path in $D$ contains a set $\bar{S}$ of $k$ edges that are not in $D'$, and let $S$ be the undirected version of $\bar{S}$. Let $E$ be a set of edges that can be queried and assume $S \subset E$. Let $w$ be the width of $D'$. Then, $k + 1 \geq w$ and the Hamiltonian in $D$ can be found with $O(wk \log n)$ queries on edges in $E$.

In order to prove the lemma, we first prove another lemma.

**Lemma 24** (Predecessor search in DAG). Given a DAG $D' = (V, E')$ of width $w$, and a vertex $v \in V$, $|V| = n$, let $D$ be the DAG obtained by extending $D'$ by probing all edges involving $v$. Define $P_v = \{u \mid (u,v) \text{ is in the transitive reduction of } D\}$. There exists an algorithm that computes $P_v$ with $O(w \log n)$ queries, and runs in $O(n^2)$ time.

**Proof.** Observe that any chain in $D'$ can contain at most one element of $P_v$, and hence $|P_v| \leq w$. Indeed, we can run one binary search on each of the $w$ chains in $D'$, leading to at most $w$ candidate predecessors. The number of queries is $O(w \log n)$ (after computing an optimal partitioning into chains in polynomial time, or by choosing a random query and excluding all implied queries by BFS in linear time each).
Proof of Lemma 14 By Dilworth’s Theorem, \( D' \) can be partitioned into \( w \) chains. To show \( k + 1 \geq w \), assume otherwise, \( w > k + 1 \), and let \( A \) be \( k + 2 \) non-comparable vertices in \( D' \). By the pigeonhole principle, there must be two vertices of \( A \) in the same of the \( k + 1 \) stretches of cost 0 edges on the Hamiltonian, a contradiction to them being incomparable.

Here is the procedure to turn \( D' \) into the Hamiltonian with a total of \( O(kw \log n) \) queries: Observe that adding edges to \( D' \) does not increase its width. As long as \( D' \) is not the Hamiltonian path, let \( S \) be the first layer of a BFS traversal of the transitive reduction of \( D' \) with \( |S| \geq 2 \), and observe that \( S \) is an antichain and \( |S| \leq w \). Invoke the predecessor finding procedure of Lemma 24 for each \( v \in S \) and add to \( D' \) the edges \( \{(u,v) \mid u \in P_v\} \), resulting in the DAG \( D \). By the above arguments, \( D \) contains the Hamiltonian with only \( k - |S| + 1 \) unrevealed edges missing. We used \( O(|S|k \log n) \) queries to reduce the number of unrevealed edges by \(|S| - 1\) for \(|S| \geq 2\), hence each search creating a missing edge of the Hamiltonian, and this search must justify at most one additional such search. Hence, the total number of queries to arrive at the Hamiltonian is \( O(wk \log n) \).

A.2 Proof of Theorem 13

We recall the notation set forth in the main part of the paper. Let \( \vec{G} \) denote the true DAG that contains a directed Hamiltonian path. Define \( \vec{G}_0 \) as the DAG obtained by revealing all cost 0 edges, and let \( w_0 \) be the width of \( \vec{G}_0 \). Similarly, denote by \( \vec{G}_{01} \) the DAG obtained by revealing all cost 0 and 1 edges. Clearly, \( \vec{G}_{01} \) may not have a Hamiltonian path. Let \( k_1 \) and \( k_F \) be the number of cost 1 and cost \( F \) edges on the Hamiltonian path in \( \vec{G} \). Denote by \( w_{01} \) the width of \( \vec{G}_{01} \).

Theorem 13 (Universal Sorting, computing 0-1 DAG). There exists a randomized polynomial time algorithm that computes the transitive reduction of \( \vec{G}_{01} \) and has query cost in \( O(\min(n^{1.5} \log n, w_0 k_1 \log n)) \) if \( k_F = 0 \), else if \( k_F \geq 1 \), it has query cost in \( O(w_{01} n^{1.5} \log n) \).

Since the second term in the case when \( k_F = 0 \) comes from Lemma 14, we will show that there is an algorithm that computes \( \vec{G}_{01} \) with cost at most \( O(w_{01} n^{1.5} \log n) \). This algorithm will only probe cost 0 and 1 edges, and will be a generalization of the algorithm in [17]. Note that while the algorithm in [17] works on a cost \( \{1, \infty\} \) setting under the promise of a Hamiltonian path in the true graph, our algorithm finds the transitive reduction of the DAG \( \vec{G}_{01} \) of width \( w_{01} \).

Theorem 25. There is a poly-time algorithm that with high probability solves universal sorting (Definition 5) for an instance with edge costs in \( \{0,1,\infty\} \) with total cost \( O(w n^{3/2} \log n) \), where \( w \) is the width of \( \vec{G}_I \).

The remainder of this subsection is dedicated to the proof of this theorem. All of the definitions, algorithms, and accounting to estimate the in-degree of a vertex in [17] remain valid and unchanged. Observe that any topological sorting of the underlying directed graph, together with the undirected graph, reveal the directed graph. Define the Average rank of a vertex as the average rank over all linear extensions of the true underlying directed graph. The following result implies that the the average rank \( r \) on a path in (the transitive reduction of) a DAG is increasing by at least one per edge.
Lemma 26. Let \( D = (V, E) \) be a DAG and \( r: V \to \mathbb{Q}^{\geq 0} \) be the average rank. Then for \((u, v) \in E\) we have \( r(u) + 1 \leq r(v) \).

Proof. Let \( \Pi \) be the set of all linear extensions that are compatible with \( D \), such that \( r(x)|\Pi| = \sum_{\pi \in \Pi} \pi(x) \). Then \( |\Pi|(r(v) - r(u)) = \sum_{\pi \in \Pi} \pi(v) - \pi(u) \geq |\Pi| \cdot 1 \).

Definition 27 (Convex vertex subset). In a DAG \( G = (V, E) \), a subset of vertices \( S \subseteq V \) is convex if for every pair of vertices \( u, v \in S \), every vertex \( w \) on any directed path from \( u \) to \( v \) in \( G \) is also in \( S \).\(^7\)

Hence, considering a subset of the vertices by an upper and a lower bound on the average rank, leads to a convex subset.

A vertex is **live** if there is an unprobed edge incident to it, otherwise it is **exhausted**. The **assumed graph** is the same directed graph as in [17]. An **active vertex** is one that has at most \( 4\sqrt{n} \log n \) unprobed in-edges in the assumed graph. A **free edge** is an unprobed edge \((u, v)\) where the endpoint \( v \) is active.

Lemma 28 (Generalization of Lemma 3.5 in [17]). The \( \sqrt{n} \) live vertices with smallest average rank are all active.

Proof. Identical to that in [17].

If the width of \( \vec{G} \) is at least \( \sqrt{n}/4 \), the statement of Theorem 25 is easy to achieve by probing all edges. Hence, let us assume the width is at most \( \sqrt{n}/4 \). Then, by Lemma 26, at most \( \sqrt{n}/4 \) vertices can have the same average rank.

Lemma 29 (Generalization of Lemma 3.6 in [17]). If there are no free edges, then there exists a set of at least \((3/4)\sqrt{n}\) live vertices with known partial order.

Proof. adapted from [17] Consider the set \( S \) of \( \sqrt{n} \) live vertices with smallest average rank, i.e., filtered by some upper bound on the average rank. Tie breaking might reduce the size of \( S \) by \( 1/4 \); we set \( S \) to be this perhaps smaller, filtered, set. By Lemma 28, all vertices of \( S \) are active. Assume there are no free edges. Let \( u, v \in S \) be a pair of vertices that have a directed path \( P \) from \( u \) to \( v \) in \( \vec{G} \). Then, all of this path \( P \) is in \( S \), and all live vertices of \( P \) are in \( S \). Hence, because there are no free edges, and all vertices of \( P \) not in \( S \) are exhausted, all edges of \( P \) must be probed.

Lemma 30. The width of the current DAG restricted to \( S \) is at most \( w \).

While \( S \) is defined As we already know the average ranks of all vertices, we can compute \( S \) as in the proof of Lemma 29. Then we run the algorithm of Lemma 24, and determine for every remaining vertex its predecessors in \( S \), with \( O(w \log n) \) queries each, compared to the \( O(\log n) \) queries if there is a Hamiltonian. Hence, the total number of queries increases from \( O(n^{3/2} \log n) \) to \( O(wn^{3/2} \log n) \), as claimed in the theorem.

\(^7\)Our definition of convexity differs from the definition in the metric graph theory (defined on undirected graphs), where convex subgraph contains the vertices of only the shortest paths between every pair of vertices [4].
B Bipartite Sorting

B.1 Instance optimality spelled out for bipartite sorting

Let us make the definitions of instance optimality concrete for the case of bipartite sorting as given in Definition 8. Such a red-blue DAG $G$ leads to the notion of stripes: Because all bicolored comparisons are allowed, all sources of $G$ are of the same color. We call them stripe 1 (one could also think of it as a layer), and iteratively define stripe $i$ as the sources after removing stripes 1 to $i-1$. Note that the transitive reduction of $G$ is the complete bipartite subgraph between neighboring stripes and nothing else.

**Definition 31** (Canonical Value Function, Stripe, Size Vector). Given an instance $I$ with $G_I = (R \cup B, \vec{E})$. Let $S^A_1 = \{x \in A \mid \forall y \in B: x < y\}$ or $S^B_1 = \{x \in A \mid \forall y \in B: x < y\}$ be the set of smallest elements. Note that one of them is empty, and define $S_1 = S^A_1 \cup S^B_1$. A function $s: R \cup B \to \mathbb{N}$ is called a canonical value function, if it has the following properties:

1. For every $x \in S_1$, $s(x) = 1$.
2. Let $M = \max_x s(x)$, for $i < M$, the preimage $s^{-1}(i) \neq \emptyset$, define $S_i = s^{-1}(i)$.
3. For all directed edges $\vec{e} = (x, y) \in \vec{E}$ if $x < y$ then $s(x) < s(y)$.
4. For two elements $x$ and $z$ of the same color with $s(x) < s(z)$, there exists an element $y$ of the other color with $s(x) < s(y) < s(z)$.
5. If the first canonical stripe $S_1 \subset R$ of $I$ is red (blue), then $s(x)$ is odd (even) for all $x \in R$ and $s(y)$ is even (odd) for all $y \in B$.

We say that $M$ is the number of stripes, the preimage of a value $i$ taken by a value function $s$, $s^{-1}(i)$, is called the $i$th stripe of $s$. Given the canonical value function $s$, define the canonical size vector of $s$ as:

$$v(s) = (|\{x \mid s(x) = i\}|)_{i=1...M}.$$ 

B.2 Yao’s principle

**Lemma 15** (Yao’s principle for $\text{OPT}$ on a neighborhood $\mathcal{N}(I)$ of instance $I$).

$$\text{OPT}(\mathcal{N}(I)) \geq \Omega \left( \sup_{\mathcal{D} \text{ distribution over } \mathcal{N}(I)} \inf_{A \in \mathcal{A}_{\text{det}, \text{failure prob. } \leq 1/4}} \mathbb{E}_{I' \in \mathcal{D}}[\text{cost}(A, I')] \right)$$

**Lemma 32** (Yao’s principle, Theorem 3 in [27]). Let $\mathcal{X}$ be a set of inputs, let $\mathcal{D}$ be a distribution over $\mathcal{X}$ and let $\mathcal{R}_\varepsilon$ be a randomized algorithm that outputs the correct answer with probability at least $1 - \varepsilon$ for $0 \leq \varepsilon \leq 1/2$. Then

$$\max_{x \in \mathcal{X}} \mathbb{E}_{\mathcal{R}_\varepsilon}[c(\mathcal{R}_\varepsilon, x)] \geq \frac{1}{2} \inf_{A \in \mathcal{A}_{\mathcal{D}_\varepsilon}} \mathbb{E}_{x \in \mathcal{D}}[c(A, x)]$$

where $\mathcal{A}_{\mathcal{D}_\varepsilon}$ denotes the set of all deterministic algorithms that are correct with probability at least $1 - 2\varepsilon$ when run on input drawn from distribution $\mathcal{D}$. 

21
All our lower bounds will be derived via an application of the following lemma:

**Lemma 33.** Let $\mathcal{I}$ be an instance, $\mathcal{N}(\mathcal{I})$ be some neighborhood of instances containing $\mathcal{I}$, $\mathcal{D}$ a distribution over $\mathcal{N}(\mathcal{I})$, and assume that any deterministic algorithm with expected cost of at most $C$ must fail with probability (at least) $1/4$ for input drawn from $\mathcal{D}$. Then $\text{OPT}(\mathcal{I}) > C/16$, where $\text{OPT}$ is allowed to have error probability at most $1/16$.

**Proof.** Assume to the contrary that $\text{OPT}(\mathcal{I}) \leq C/16$, where the corresponding algorithm is allowed to err, on every instance $\mathcal{I}' \in \mathcal{N}(\mathcal{I})$, with probability at most $1/16$. By Yao’s principle, Lemma 33, this means that

$$
\frac{1}{2} \inf_{A_{1/8} \in A_{1/8}} \mathbb{E}_{I' \sim \mathcal{D}}(c(A_{1/8}, I')) \leq C/16,
$$

where $A_{1/8}$ is the class of deterministic algorithms with error probability at most $1/8$ on the distribution $\mathcal{D}$. Multiplying throughout by 2, in particular, there exists a deterministic algorithm $A_{1/8} \in A_{1/8}$ whose expected cost on a random input drawn from $\mathcal{D}$ is at most $C/8$. Now consider the deterministic algorithm $A$ that does the following: it runs as $A_{1/8}$ does on an input $\mathcal{I}'$ until one of the following two events happen: either $A_{1/8}$ terminates, or $C$ comparisons are performed. If $A_{1/8}$ terminates using at most $C$ comparisons, $A$ produces the same output as $A_{1/8}$, and if $A_{1/8}$ is taking longer than $C$ comparisons, $A$ outputs an arbitrary answer.

Clearly, $A$ always finishes after at most $C$ comparisons. Given that $\mathbb{E}_{I' \sim \mathcal{D}}(c(A_{1/8}, I')) \leq C/8$, by application of Markov’s inequality we get that

$$
\Pr_{I' \sim \mathcal{D}}(c(A_{1/8}, I') > C) < (C/8)/C = 1/8.
$$

In addition, by definition, $A_{1/8}$ could fail on inputs from $\mathcal{D}$ with probability at most $1/8$. By a union bound, the probability that $A$ finishes correctly on an input drawn from $\mathcal{D}$, which happens when both $A_{1/8}$ is correct and terminates within $C$ comparisons, on that input, is at least $1 - (1/8 + 1/8) = 3/4$. This means there is a deterministic algorithm $A$, whose cost is always at most $C$, and which fails on inputs drawn from $\mathcal{D}$ with probability less than $1/4$. However, this contradicts the hypothesis of the theorem, implying that $\text{OPT}(\mathcal{I}) > C/16$. \hfill \square

### B.3 Verification Lower Bound

**Lemma 16 (Verification lower bound).** Let $\mathcal{I}$ be an instance of bipartite sorting, let $K \subset \bar{E}_I$ be its transitive reduction, and define $C_V = |K|$. There exists a distribution $\mathcal{D}$ over $\mathcal{N}_E(\mathcal{I}) \subset \mathcal{N}_{AE}(\mathcal{I})$ such that any deterministic algorithm $A$ that does at most $C_V/2$ comparisons must fail with probability (at least) $1/4$.

**Proof.** For $e = (x, y) \in K$, let $\mathcal{I}_{xy}$ be the instance based on $G_{\mathcal{I}e}$ (recall Definition 10), and observe that it is a DAG. Importantly, $\mathcal{I}_{xy}$ can only be distinguished from $\mathcal{I}$ by comparing $x$ and $y$.

Define $\mathcal{D}$ to be the distribution where we draw instances from $\mathcal{N}_E(\mathcal{I})$ in the following way: With probability $1/2$, choose $\mathcal{I}$. Otherwise, choose uniformly $(x, y)$ from $K$ and produce the instance $\mathcal{I}_{xy}$. 

22
Let $\mathcal{A}$ be any deterministic algorithm that does at most $C_V/2$ comparisons on $\mathcal{I}$, and let $C \subset K$ be the set of compared pairs. Because $\mathcal{I}$ is chosen in $\mathcal{D}$ with probability 1/2, $\mathcal{A}$ has to return $G_\mathcal{I}$ when running on $\mathcal{I}$, otherwise the error probability would be too high. Also, the set $C = K \setminus C$ has size at least $C_V/2$, and for any $(x, y) \in C$, the algorithm $\mathcal{A}$ on $\mathcal{I}_{xy}$ will run exactly as on $\mathcal{I}$ and hence incorrectly return $G_\mathcal{I}$. This happens with probability at least 1/4, since with probability 1/2 the chosen input is different from $\mathcal{I}$ (say the input is $I_{xy}$), and with probability 1/2, the comparison $(x, y) \in C$ is missed, so the algorithm returns the incorrect output $G_\mathcal{I}$.

\[\square\]

### B.4 Inversion finding bound

**Lemma 17** (Inversion finding lower bound). Let $\mathcal{I}$ be an instance of bipartite sorting with $n \geq 2$ red and $m \geq 2$ blue elements, where not all comparisons come out the same, and define

$$C_\mathcal{I} = \frac{nm}{\min(|\{(r, b) \in R \times B \mid r < b\}|, |\{(r, b) \in R \times B \mid r > b\}|)}.$$ 

There exists a distribution $\mathcal{D}$ over $\mathcal{N}_A(\mathcal{I})$ such that any deterministic algorithm $\mathcal{A}$ that does at most $C_\mathcal{I}/2$ comparisons must fail with probability (at least) 1/4.

**Proof.** Let $\mathcal{D}$ be the uniform distribution over $\mathcal{N}_A(\mathcal{I})$. Note that $\mathcal{N}_A(\mathcal{I})$ contains all instances where the stripes are internally arbitrarily permuted. Observe $|\mathcal{N}_A(\mathcal{I})| \geq 4$ by the bounds on $n$ and $m$. W.l.o.g., assume $|\{(r, b) \in R \times B \mid r < b\}| < |\{(r, b) \in R \times B \mid b < r\}|$, i.e., the usual outcome of a comparison is $b < r$ and the inversion is $r < b$. Let $p = 1/C_\mathcal{I} = |\{(r, b) \in R \times B \mid r < b\}|/nm \leq 1/2$ be the probability that a randomly chosen pair of elements is an inversion. Let $\mathcal{A}$ be a deterministic algorithm and think of it as a decision tree $\mathcal{T}$ where nodes are red-blue comparisons and non-inversion go to the left, inversions go to the right. Each leaf of the tree is marked with an output (that declares which instance was represented by the input), or a failure output.

**Claim:** Let $v_k$ be the node $v$ that is reached by $k$ comparisons returning “non-inversion” (i.e. the leftmost node of $\mathcal{T}$ at depth $k$). When input is drawn from $\mathcal{D}$, the node $v_k$ is reached with probability at least $1 - kp$.

**Proof of Claim:** An input reaches $v_k$ if $k$ (potentially dependent) random experiments all came out as “non-inversion”, each having a probability $1 - p$. The claim follows from a union bound over the fail events.

From the claim it follows that if $v_k$ is a leaf for $k \leq C_\mathcal{I}/2$, the algorithm must fail with probability at least 1/4: Then $kp \leq C_\mathcal{I}/2 \cdot 1/C_\mathcal{I} = 1/2$ and $1 - kp \geq 1/2$, so half of the inputs reach $v_k$. Because there are at least four inputs, at least two reach $v_k$, but it can only be labeled with one, the other(s), which stand for at least 1/4 of the inputs in $\mathcal{N}_A(\mathcal{I})$, make the algorithm fail.

\[\square\]

### B.5 Decomposition bound

Let $\mathcal{I}$ be an instance and $(S_1, \ldots, S_k)$ its stripes (see Definition 31). Consider pairs of indices $(a_1, b_1), \ldots, (a_\ell, b_\ell)$, where for all $1 \leq j \leq \ell$, $a_j$ and $b_j$ both belong to $\{1, \ldots, k\}$, and $a_j < b_j < a_{j+1} < b_{j+1}$ for all $j < \ell$. For $1 \leq j \leq \ell$ define subinstances $\mathcal{I}_j$ by the subgraph of $G_\mathcal{I}$ on the vertices $V_j = \bigcup_{i=a_j}^{b_j} S_i$.  

\[\end{raw_text} \]
Lemma 18 (Decomposition into Lower Bounds for Subproblems). Consider any $0 \leq \epsilon \leq 1/2$ and for $1 \leq j \leq \ell$, let $I_j$ be a subinstance of $I$ as above. Then $\text{OPT}(I) \geq \sum_{j=1}^{\ell} \text{OPT}(I_j)$, where both $\text{OPT}(I)$ and $\text{OPT}(I_j)$ are allowed to err on any input in $N_{AE}(I)$ and $N_{AE}(I_j)$, respectively, with probability at most $\epsilon$.

Proof. Given subinstances $I'_j \in N_{AE}(I_j)$, form the instance $I'$ which is the same as $I$, but with subinstances $I_j$ replaced by $I'_j$. More precisely, this means that for $\tilde{G} = \tilde{G}_I$, each graph $\tilde{G}_j$ could be modified by flipping a subset $F_j$ of essential edges, respecting the implied edges of $\tilde{G}_j$, i.e. $\tilde{G}_{j,F_j}$ is a DAG. Now any path in $\tilde{G}_{F_j}$ using an edge in $F_j$ is consistent with previously implied edges: If the edge is inside $\tilde{G}_j$ this holds by definition; if not, then the edge must span from a layer before $\tilde{G}_j$ to a layer after $\tilde{G}_j$, and the unchanged complete directed bipartite graphs on levels $a_j - 1$ vs $a_j$ and $b_j$ vs $b_{j+1}$ ensure that the direction is upward. Hence, $F = \cup_j F_j$ is consistent with all implied edges of $\tilde{G}$, because $F_i$ and $F_j$ are separated by at least one such layer.

Additionally, there is an automorphism $\pi_j: V_j \to V_j$ of $\tilde{G}_j$ that keeps $G_j$ unchanged, i.e., it permutes $R_j$ and $B_j$, but nothing else. Observe that the extension of $\pi_j$ to $\pi'_j: V \to V$ with $\pi'_j(x) = x$ for $x \notin V_i$, has the same property, and hence the composition of these operations $\pi = \pi'_1 \circ \cdots \circ \pi'_\ell$ respects red and blue.

Hence, the instance $I' \in N_{AE}(I)$.

Consider any algorithm $A$ that runs on $N_{AE}(I)$ (with error probability at most $\epsilon$ on any instance in $N_{AE}(I)$). We will show that there exist algorithms $A_j$, each with error probability at most $\epsilon$ on $N_{AE}(I_j)$, such that for any collection of instances $I'_j \in N_{AE}(I_j)$, for $1 \leq j \leq \ell$, and $I'$ defined as above,

$$\text{cost}(A, I') \geq \sum_{j=1}^{\ell} \text{cost}(A_j, I'_j)$$

In particular, we get that

$$\text{cost}(A, I') \geq \max_{I'_1, \ldots, I'_\ell} \sum_{j=1}^{\ell} \text{cost}(A_j, I'_j) = \sum_{j=1}^{\ell} \max_{I'_j} \text{cost}(A_j, I'_j) \geq \sum_{j=1}^{\ell} \text{OPT}(I_j),$$

where the second equality follows from the fact that $I'_j$ are disjoint instances, and hence there is no interaction between the algorithms $A_j$; and the third inequality follows by definition of $\text{OPT}(I_j) = \inf_{A'_j} \max_{I'_j \in N_{AE}(I_j)} \text{cost}(A'_j, I'_j)$. With a slight abuse of notation, let $I'_j$ be the instance that achieves the maximum in $\max_{I'_j} \text{cost}(A_j, I'_j)$, and $I'$ be defined using these $I'_j$. We have so far that $\text{cost}(A, I') \geq \sum_{j=1}^{\ell} \text{cost}(A_j, I'_j) \geq \sum_{j=1}^{\ell} \text{OPT}(I_j)$.

Taking the infimum over all algorithms $A$ would then give

$$\text{OPT}(I) \geq \inf_A \text{cost}(A, I') \geq \sum_{j=1}^{\ell} \text{OPT}(I_j),$$

proving the lemma.
We now show how to prove
\[ \text{cost}(A, I') \geq \sum_{j=1}^{\ell} \text{cost}(A_j, I'_j). \]

Consider running A on I'. Define our algorithms A_j for I'_j, for each 1 \leq j \leq \ell as follows:

- If A compares two keys, both of which are in I'_j, A_j compares the same keys, and and obtains the same result of the comparison as A.
- If A compares two keys, at least one of which is not in I'_j, A_j skips this comparison.
- At the end, A_j outputs the stripes output by A that have indices between a_j and b_j, i.e., it outputs the graph given by \{S_{a_j}, S_{a_j+1}, \ldots, S_{b_j}\} in this order.

Clearly the failure probability of A_j is no larger than the failure probability of A (A may fail on parts of the input of I' outside \bigcup I'_j, without affecting correctness of A_j). Also, the work done by A (which is at most OPT(I)) is at least \sum_{j=1}^{\ell} \text{cost}(A_j, I'_j), and we have almost proven what we wanted, except we also need to show that A_j are valid algorithms for I'_j. This is easily observed: a) clearly, the output of A_j must be consistent with its comparisons, otherwise the output of A is inconsistent with A’s comparisons too, and b) because the subinstances are disjoint, if A compares two keys x and y, at least one of which, say x, is not in I'_j, this comparison has no implication on solving I'_j, since all elements in I'_j are either smaller than or larger than x, and therefore x cannot distinguish between order of elements in I'_j. This completes the proof. \( \square \)

C  InversionSort is polylog-instance optimal for Bipartite Sorting: Proof of Theorem 21

C.1 Setting of the analysis

Refinement tree as a trace and mean to analyze  Looking only at the backbone, the algorithm does an interval refinement. If we look back in time, the final backbone is given by the instance, only the choice of the representative of a stripe is arbitrary (everything else constitutes the output). In contrast, the evolution of the backbone is very much driven by the random choices of the algorithm and leads to a hierarchy of intervals on the backbone. A new inversion and the pivoting steps split one interval of the backbone (not to be confused with a bucket that somehow spans two intervals of the backbone) into three, as detailed in the following definition.

Definition 34  (Refinement tree). The root node of the tree has as bounding pivots the artificial smallest red and largest blue element. Every node v of the tree has two pivots b_v and r_v of the (final) backbone associated to it with the guarantee that b_v and r_v were neighbors on the backbone at some stage of the algorithm. If r_v < b_v the node is said to have polarity “red smaller than blue”, otherwise “blue smaller than red”. If the algorithm finds an inversion x, y
between $b_v$ and $r_v$, then $v$ has three children with the respective pivots $(b_v, y)$, $(y, x)$, $(x, r_v)$
or $(r_v, y)$, $(y, x)$, $(x, b_v)$. The polarity of the middle child is the opposite of the polarity of $v$,
wheras the two outer children have the same polarity as $v$. See Figure 3. If the algorithm
finishes the subproblem by completing verification and concludes that $b_v$ and $r_v$ are representing
neighboring stripes of the output, then $v$ is a leaf of the tree.

Figure 3: A parent node $v$ and its three children $o, p, q$ in the refinement tree. The pivots for
the parent are $(b_v, r_v)$, and since the parent is of type blue < red, the inversion found is of
type red < blue. The left and right children have the same polarity as the parent, whereas
the middle child gets opposite polarity.

Observe that any snapshot of the algorithm, any existing backbone corresponds to the
in-order leaf traversal of some subtree of the refinement tree that includes the root. See
Figure 4.

C.2 Bounding the depth of the refinement tree

To analyse the refinement tree, it is convenient to fix one topological sorting of $\vec{G}_I$, i.e., a
totally ordered list $L_I = (x_0, x_1, \ldots, x_{n+m-1})$ of (mixed) red and blue elements. This fixes an
arbitrary order inside each stripe of the instance.

**Definition 35** (List subinstance). Let $r \in R$ and $b \in B$ be two elements and let $i, j$ be their
indices in $L_I$, i.e., $x_i = r$ and $x_j = b$. Then, the instance $I_{rb}$ given by the list $(x_i, \ldots, x_j)$ or
$(x_j, \ldots, x_i)$ is called the list subinstance of $L_I$ defined by $r$ and $b$.

The following lemma is stated slightly weaker than what InversionSort actually achieves.
Working with this version of the lemma makes it easier to argue about the version of
InversionSort that is modified for the bichromatic setting, as we will see later.

**Lemma 36** (Randomness in Inversion Finding). At any stage of the algorithm, consider
a successful inversion finding procedure, which finds an inversion $y < x$ on the part of
the backbone between representatives $u_i < u_{i+1}$, with $y \in X_{i+1}$ and $x \in X_i$. Say, w.l.o.g., that $u_i$
is red and $u_{i+1}$ is blue, and hence $x$ is red and $y$ is blue. Consider the list subinstances $I_{yu_{i+1}}$
and $I_{u_ix}$, and denote by $I_{yu_{i+1}}^b$ (resp. $I_{yu_{i+1}}^r$) the subset of blue (resp. red) elements in the
subinstance $I_{yu_{i+1}}$.

\[\text{This subtree is obtained by removing from the full decomposition tree subtrees of those internal nodes}
\text{that have not yet found inversions.}\]
Figure 4: A sketch of the complete ternary refinement tree (in black) and the partial refinement tree corresponding to a snapshot of the backbone during a run of InversionSort.

• $\forall z \in I^b_{u_ix}, \Pr(A \text{ finds the inversion } (z, x) \mid x \text{ is one of the keys in the inversion}) = \frac{1}{I^b_{u_ix}}$. That is, conditioned on $x$ being in the inversion, $y$ is uniformly distributed among all the blue elements in $I^b_{u_ix}$.

• In contrast, conditioned on $y$ being in the inversion, $x$ is not uniformly distributed among all the red elements in $I^r_{u_{i+1}x}$, but there is a bias towards the outside, i.e., conditioned on $y$ being fixed, $P(x = a) \leq P(x = b)$ for $a < b$.

Proof. Any blue element in $I^b_{u_ix}$ creates precisely one inversion with $x$, and no other elements in $I^b_{u_{i-1}u_i}$ create an inversion with $x$. Conditioned on the fact that $x$ is one of the two elements in the inversion, all inversions including $x$ are equally likely, and therefore all elements in $I^b_{u_ix}$ are equally likely to be the second element $y$ in the inversion. The symmetric consideration show that also $x$ is uniformly distributed for fixed $y$, and this implies the second statement of the lemma.

Definition 37. Let $v$ be a node in the refinement tree with the pivots $r_v$ and $b_v$, and let $I^r_{r_vb_v}$ the list-subinstance of $v$. Then define $R_v = I^r_{r_vb_v} \cap R$ and $B_v = I^r_{r_vb_v} \cap B$, and $P_v$ as the number of pairs of $v$ as $P_v = |R_v| \cdot |B_v|$.

We stress that this $P_v$ is not the same as the one in Lemma 24, which was used for predecessors in the setting of universal sorting.
Lemma 38. Let \( v \) be a non-leaf node of the refinement tree of bichromatic InversionSort. Then, with probability at least 1/2,
\[
\max_{w \in \{o,p,q\}} P_w \leq \frac{7}{8} P_v
\]

Proof. Let \( b \in B, r \in R \) be the inversion and assume w.l.o.g \( r_v < b < r < b_v \), and call the three children of \( v \) as \( o,p,q \) with \( r_o = r_v, b_o = b_p = b, r_p = r_q = r \) and \( b_q = b_v \). See Figure 3. W.l.o.g., assume \( b \) is uniformly chosen given \( r \), but \( r \) has a bias towards making the outer child \( q \) smaller, i.e., since \( r \) is a max, \( |P_q| \) is smaller than if \( r \) were uniform. By Lemma 36, we can conclude that with probability at least 3/4,
\[
\max(|B_o|, |B_p|) \leq \frac{7}{8} (|B_o| + |B_p|) \leq \frac{7}{8} |B_v|,
\]
and \( |R_q| \leq \frac{3}{4} |R_v| \) with probability at least 3/4. By a union bound, we have both statements with probability at least 1/2, and in the following we assume they both hold.

From the first statement follows (even if \( R_o = R_v \) or \( R_p = R_v \)) that \( P_o \leq \frac{7}{8} P_v \) and \( P_p \leq \frac{7}{8} P_v \). From the second statement follows \( P_q \leq \frac{3}{4} P_v \) (even if \( B_q = B_v \)).

Theorem 39 (Height of the refinement tree). Let \( T \) be the refinement tree of running InversionSort on an instance \( I \) with \( N = n + m \) elements. With high probability in \( N \), the height of \( T \) is \( O(\log N) \).

Proof. Let \( x \) be an element of \( I \) and consider the root-to-leaf path \( P \) of \( T \) that consists of the nodes \( v \) that contain \( x \) in \( I_v \). Let \( Q \subset P \) be the set of nodes where the number of pairs is reduced by a factor 7/8, as in Lemma 38. Then \( |Q| = \Theta(\log N) \) and by a Chernoff bound, with probability at least \( 1 - N^{-2}, |P| = O(|Q|) \). A union bound over all \( x \) gives the statement of the theorem.

C.3 Pivoting Cost Analysis

Recall that InversionSort always pivots with the newly found inversion, and in doing so, it compares all elements in a bucket \( X_i \) to at most two new pivots. Even though InversionSort does not proceed layer-by-layer (in fact, different vertices of the same layer could have wildly varying times of InversionSort finding inversions in them and decomposing them into children; see Figure 4), we can still upper bound the total pivoting cost performed on a layer by \( O(n + m) \). Using Theorem 39, we get that

Theorem 40 (Pivoting Cost of InversionSort). Let \( N = n + m \). With probability at least \( 1 - 1/N \), the pivoting cost of InversionSort on an instance of size \( N \) is at most \( O(N \log N) \).

C.4 Inversion Finding Cost Analysis

What remains to be analysed is the cost, in number of comparisons, that is incurred to find the pairs of new pivots. For every stage of InversionSort, the current backbone corresponds to a left-to-right traversal of the current subtree (including the the root) of the decomposition tree. In each stage of InversionSort, for every neighboring pair of pivots, say \( r_k \) and \( b_{k+1} \), or rather the corresponding pair of buckets \( R_k \) and \( B_{k+1} \), a comparison is performed, searching for a new pair of pivots. To explain this subproblem in terms of the underlying instance, we need to refer to the stripes of \( b_{k-1} \) and \( r_{k+2} \). Using Definition 31, these would be the
indices \( i = s_f(r_k), j = s_f(b_{k+1}) \) \( j' = s_f(b_{k-1}) \) and \( i' = s_f(r_{k+2}) \) with \( j' < i < j < i' \). With this \( R_k = R \cap \bigcup_{l=j'+1}^{j-1} S_l \setminus \{r_k\} \), and analogously \( B_{k+1} = B \cap \bigcup_{l=i'+1}^{i-1} S_l \setminus \{b_{k+1}\} \). In contrast, the subinstance between \( r_k \) and \( b_{k+1} \) would only consist of the stripes \( S_i \) to \( S_j \) with \( R' = R \cap \bigcup_{l=i}^{j-1} S_l \) and \( B' = B \cap \bigcup_{l=i}^{j} S_l \). The additional red stripes \( S_{j'+1}, S_{j'+3}, \ldots, S_{i-1} \) constitute a spill-in of additional red elements that make it even harder to find a new pair of pivots. Similarly, there can be a spill-in of blue elements from the subinstance between \( b_{k-1} \) and \( r_k \), that are part of \( B_k \) besides \( B' \). This spill-in is the shaded part in Figure 2. We call a subproblem unaffected if \( |R'| \geq |R_k|/4 \) and \( |B'| \geq |B_{k+1}|/4 \). Note that in the course of running InversionSort, subproblems can turn from affected to unaffected, but not vice versa.

Observe that neighboring subinstances share the stripe of the pivot; this stripe does not constitute spill-in. Still, over a single pivot, there typically is spill-in of the same color in both directions, but only one of them can affect the receiving subproblem: \( R_k \) consists of the red elements in the two neighboring subproblems, \( S_i \subset R_k \cup \{r_k\} \) is part of both, so one of the two sets of spilling-in elements must be less than half of \( R_k \), so clearly not both of them can be more than 3/4 of \( R_k \).

**Overall idea**

What remains to complete the proof of Theorem 21, is the following bound on the overall cost of inversion finding. This outlines the remaining task, the lemmas used in the proof are presented in the remaining part of this section.

**Theorem 41 (Searching Cost of InversionSort).** Let \( N = n + m \). There is a constant \( c \), such that with probability at least \( 1 - 1/N \), the total number of comparisons done by InversionSort while searching new pivot pairs, on an instance of size \( N \), is at most \( \text{OPT}(I)c(\log N)^4 \).

**Proof.** If we can charge \( t \) inversion finding comparison of InversionSort as one comparison in a specific subinstance of the refinement tree, then Lemma 42 states that the total cost is bounded by \( 2t \text{OPT}(I) \).

Consider an unaffected subproblem between neighboring pivots \( u_i \) and \( u_{i+1} \) on the backbone with the buckets \( B_i \) and \( B_{i+1} \). Here, searching for an inversion is charged to \( O = \text{OPT}(I_v) \) where \( v \) is the node with the pivots \( u_i \) and \( u_{i+1} \), and \( I_v \) is the subinstance defined by the stripes of \( r_v \) and \( b_v \), and all the stripes of \( I \) between them.

If \( u_i \) and \( u_{i+1} \) are in neighboring stripes \( X,Y \) of the instance, then \( O = |X||Y|/2 \) by Lemma 16. Now, observe that the procedure of Section 4.3 does at most a total of \( a = 2|B_i||B_{i+1}| \) comparisons for the two pivots. By assumption the subproblem is unaffected and hence the spill-in from left and right is limited, meaning that \( |B_i| \leq 4|X_i| \) and \( |B_{i+1}| \leq 4|X_{i+1}| \), and hence \( a \leq 32O \).

Otherwise, let \( p \) be the probability of a random pair forming an inversion, and let \( a \) be the number of comparisons done while the subinstance is unaffected (until an inversion is found). From the inversion finding bound of Lemma 17 follows \( 1/p \leq 2O \) and hence w.h.p. \( a/O = O(\log N) \): The expected number of found inversions among \( (d\log N)/p \) trials (comparisons) is \( d\log N \) and by a Chernoff bound follows that the probability of not finding an inversion is \( N^{-d} \).

Hence, the comparisons done while inversion searching in unaffected subproblems are upper bound by \( \text{OPT}(I)c(\log N)^2 \) for some sufficiently large \( c \). It remains to show that at
any stage of InversionSort, a $\Theta((\log N)^2)$ fraction of the active subproblems is unaffected, which is the statement of Lemma 45.

\begin{proof}

Lemma 42. Let $\mathcal{T}$ be the refinement tree of height $h$ for a run of InversionSort on instance $\mathcal{I}$, and let $V_\mathcal{T}$ be the set of nodes of $\mathcal{T}$. Then

$$\sum_{v \in V_\mathcal{T}} \text{OPT}(\mathcal{I}_v) \leq 2h \text{OPT}(\mathcal{I})$$

Proof. It is enough to show that for each level $L$ of $\mathcal{T}$ the inequality $\sum_{v \in L} \text{OPT}(\mathcal{I}_v) \leq 2 \text{OPT}(I)$ holds. Note that the subinstances $\mathcal{I}_v$ of the same polarity in any $L$ do not share vertices. Hence the decomposition Lemma 18 is applicable and the statement of the lemma follows because there are two polarities.

\end{proof}

Many unaffected subproblems.

A middle child with its three children currently being leafs is easy to analyse because the inversion that defines the boundaries between the children is described by Lemma 36. Note that we will use the following lemma on partial refinement trees.

Lemma 43. Let $T$ be a ternary tree of height $h$ and let $L = (v_1, \ldots, v_k)$ be part of the left-to-right leaf-traversal of $T$ of length $k \geq 4(h + 2)$. Then $L$ must contain a middle child whose 3 children are leafs.

Proof. Let $d(v)$ denote the depths of a node $v$ in the tree. If there are 5 consecutive vertices in $L$ of the same depth, 3 of them must be the children of the same vertex, showing the lemma. Otherwise, no such 5 consecutive vertices exist. Then, by the bound on the length of $L$, there must be at most 4 vertices that are a local maximum with depths $D$, i.e., the preceding and following element in $L$ have smaller depths. The first vertex with depths $D$ must be the left child of a node $v$, and it is a leaf. Hence, its two right siblings exist, and because the next vertices on $L$ have depths $\leq D$, the siblings cannot have children. This shows the statement of the lemma.

Observe that finished pairs of neighboring pivots on the backbone, i.e., subproblems where InversionSort verified that there are no further inversions, don’t spill out to their neighbors, and remember that over one pivot, spilling can only be in one direction.

Lemma 44. Let $L$ be a list of consecutive subproblems on the backbone, and assume there is no spill-in from the left to the first or from the right to the last (because of finished subproblems or the instance ending). Then at least one of the subproblems must be unaffected.

Proof. There are $|L| - 1$ internal pivots over which a spill-in into subproblems could happen, but there are $|L|$ subproblems. By the pigeon-hole principle, there must be a subproblem without spill-in, i.e., unaffected.

Lemma 45 (No long chain with only affected subproblems). Let $a$ be the number of active subproblems on the backbone at some stage of InversionSort and let $u$ be the number of the unaffected subproblems. There exists a constant $c$ such that with probability at least $1 - 1/N^2$ it holds $a \leq cu(\log N)^2$. 

30
Proof. Let $L$ be a list of consecutive active subproblems on the backbone. If we can show the statement of the lemma for each individual such list, the lemma follows.

If $a = |L| \leq c(\log N)^2$, the claim follows from Lemma 44 because there is at least one unaffected subproblem.

Otherwise, by Lemma 43, there are at least $a' = a/(4h + 2)$ many subproblems corresponding to a vertex whose children are leaves. Now, by Lemma 36, one of the boundaries is chosen uniformly at random, and its probability of having spill-over to the left is $1/4$, and so is for spill-over to the right. Now encode, from left to right, these spill-overs by a 1-bit if it is different from the previous. Let $k$ be the number of 1s in this sequence, and observe that the number of ones is a lower bound on the number of subsequences with the same direction of spill-over. Every other of these subsequences must have an unaffected subproblem.

$E[k] = a'/2 \geq c(\log N)^2)/h \geq 2 \log N$ if the constant $c$ is sufficiently large. Hence, by a Chernoff bound, with probability $1 - N^2$, we get $u \geq a'/4 \geq a/(12h) \gg a/c(\log N)^2$ and $u \geq a/c(\log N)^2$ for sufficiently large $c$.  

\[  \square \]

D InversionSort is polylog-instance optimal for Bichromatic Sorting: Proof of Theorem 22

\[ \textbf{Theorem 22 (Instance Optimality of Bichromatic InversionSort). There exists a constant} \ c > 0, \ \text{such that for every instance} \ \mathcal{I}, \ \text{the cost of bichromatic InversionSort on} \ \mathcal{I}, \ \text{with probability at least} \ 1 - 1/n, \ \text{is at most} \ c(\log n)^4 \text{OPT}(\mathcal{I}), \ \text{where} \ \text{OPT}(\mathcal{I}) \ \text{is the cost of the Hamiltonian.} \]

Lemma 36 had a weaker hypothesis than what InversionSort for bipartite sorting provided (there both pivots are uniformly chosen, as opposed to just one in the lemma). This is because the modified version of InversionSort for bichromatic sorting satisfies this weaker hypothesis. Hence, only its proof needs to be adjusted.

\[ \textbf{Lemma 36 (Randomness in Inversion Finding). At any stage of the algorithm, consider} \ \text{a successful inversion finding procedure, which finds an inversion} \ y < x \ \text{on the part of the backbone between representatives} \ u_i < u_{i+1}, \ \text{with} \ y \in X_{i+1} \ \text{and} \ x \in X_i. \ \text{Say, w.l.o.g., that} \ u_i \ \text{is red and} \ u_{i+1} \ \text{is blue, and hence} \ x \ \text{is red and} \ y \ \text{is blue. Consider the list subinstances} \ I_{yu_{i+1}} \ \text{and} \ I_{ux}, \ \text{and denote by} \ I_{yu_{i+1}}^b \ \text{(resp.} \ I_{yu_{i+1}}^r \ \text{) the subset of blue (resp. red) elements in the subinstance} \ I_{yu_{i+1}}. \]

\[ \bullet \ \forall z \in I_{uyux}^b, \ \Pr(A \ \text{finds the inversion} \ (z, x) \ | \ x \ \text{is one of the keys in the inversion}) = \frac{1}{I_{uyux}^b}. \]

That is, conditioned on $x$ being in the inversion, $y$ is uniformly distributed among all the blue elements in $I_{uyux}$.

\[ \bullet \ \text{In contrast, conditioned on} \ y \ \text{being in the inversion,} \ x \ \text{is not uniformly distributed among all the red elements in} \ I_{yu_{i+1}}, \ \text{but there is a bias towards the outside, i.e., conditioned on} \ y \ \text{being fixed,} \ P(x = a) \leq P(x = b) \ \text{for} \ a < b. \]
Proof. If the inversion was found by a uniformly sampled pair \( x, y \), the original proof remains valid. Otherwise, assume w.l.o.g., that \( x \) is the maximum of the red sample. Hence, the distribution of \( x \) is biased towards the right, as stated in the lemma. Still, the distribution \( y \) is uniform among the inversions including \( y \).

\[ \square \]

D.1 Optimality of inversion finding including monochromatic queries

We show that our algorithm is optimal up to a constant factor by an exchange argument.

Consider an optimal deterministic algorithm OPT that finds an inversion with a total comparison cost \( C \) (note \( C \) is actually the cost of a certain number of comparisons of the three types). In OPT, the probability of one of the bi-chromatic comparisons finding an inversion is maximal, for the uniform distribution on the \( N_A \)-neighborhood. Let \( a \) be the number of red-red, \( b \) blue-blue, and \( c \) red-blue of the algorithm, i.e., its cost is \( C = \alpha a + \beta b + c \), and let \( p \) be its probability of finding an inversion. We will show that when our algorithm has done comparisons for a total cost of \( O(C) \), its probability of having found the inversion is at least \( p \). We run our algorithm until it did at least \( 2a \) red-red, \( 2b \) blue-blue, and afterwards \( c \) triples of red-blue comparisons. By standard arguments, this is within a constant factor of the cost of the assumed optimal algorithm.

Any element the optimal algorithm used in a monochromatic comparison is considered to be part of its respective samples. Because the naming (automorphism) of the elements is uniformly at random, these samples are completely random, and our algorithm has sampled at least as many elements in each color. Now we can distinguish between red-blue comparisons having none, one, or two elements from a sample.

A comparison without an element from the sample has probability \( p \), like in the inversion finding bound in Lemma 17.

For the comparisons, where one endpoint is in a sample, observe that the number of inversions weakly increases, as we move to the extreme towards the other side. Observe that the number of inversions a sampled element is part of, is maximized for the minimum (if the representative of the bucket is to the right) or maximum points of the sample. Even if the assumed optimal algorithm compares with other elements of the sample, shifting to the extreme element of the sample only increases the success probability. Observe also that increasing the size of the sample only increases the probability of finding an inversion.

The last type of comparisons is to compare elements of the different samples. Observe that if this succeeds for any such elements, then the extreme points of the samples are an inversion. Because our samples are at least as large as these of the assumed optimal algorithm, our probability of finding an inversion this way is at least as large as for the optimal.

Hence, the cost of inversion finding in bichromatic InversionSort is within a constant of the inherent difficulty of finding an inversion in a certain subinstance. By the bound on the tree height and the limit on spill-in, the overall cost in these subroutines is hence within a polylog factor of optimal.

D.2 Overall proof of Theorem 22

Observe that in the bichromatic setting, there is a \( \Omega(n \log n) \) lower bound. Hence, the overall pivoting cost of InversionSort is actually within a constant factor of optimal. An
individual inversion finding is, as argued above, within a constant factor of the optimum for the corresponding subinstance, and the considerations about spill in and overlapping of subinstances lead, like for bipartite InversionSort, to a polylog factor in the instance optimality of determining the stripes of the instance. Because an optimal algorithm must also solve the subproblems of determining the stripes and sorting each stripe, these steps are a lower bound, and InversionSort is overall within a polylog factor of optimal, showing Theorem 22.

D.3 Variations of bichromatic costs

D.3.1 Bi-chromatic most expensive

In this case, it is natural to first sort both colors with monochromatic comparisons, to then do binary merging using exponential searches. It is easy to see, that this algorithm is within an $O(\log N)$ factor of the cost of the Hamiltonian, and one could leave it at that.

Still, with the framework of instance optimality developed in Section 2, we can show something stronger, namely that this algorithm is $O(1)$ instance optimal. Unlike for the other variants of the problem, here, providing the optimal algorithm with the information, where on the Hamiltonian the bichromatic edges are for this instance, is clearly too much: Then it can completely avoid expensive bichromatic comparisons, and just produce the corresponding output after the two cheap sorting steps. Hence, we should adapt the notion of neighborhood, and what seems natural is to only announce (or fix for the neighborhood) the number of bi-chromatic edges on the Hamiltonian, but not their position (and automorphisms). With this neighborhood, just for automorphisms, the sorting terms are, up to a constant, justified by information theoretic arguments that we have to determine the rank of an element within its color. For $k$ bi-chromatic edges on the Hamiltonian, there are respectively $(n-k')$ and $(m-k'')$ possibilities, where $k', k'' \in \{(k-1)/2, (k-1)/2\}$ and $k' + k'' = k - 1$ (and clarify what color the smallest element has). Since $\log \binom{n}{k} = O(k \log(n/k))$ and $\sum_k \log n_i \leq k \log n/k$ for $n = \sum_k n_i$, also the merging algorithm is optimal up to a constant factor, if it makes sure that the two cases of a single bichromatic edge on the Hamiltonian are detected with only constantly many comparisons.

D.3.2 Bi-chromatic middle expensive

This setting can be expressed as a monotone function in the framework of [15].

There is also a straight forward direct algorithm: W.l.o.g., assume $\alpha < 1 < \beta$. Sort the red elements using red-red comparisons at cost $O(\alpha|R| \log |R|)$. For each blue element, perform a binary search in these red, at total cost of $O(|B| \log |R|)$. Finally, sort the blue stripes with total cost $O(\beta \sum |B_i| \log |B_i|)$. Each of the three terms is justified by the need to determine the rank of a certain type of elements, and the above procedure uses the cheapest available comparisons for this task.

D.4 More than two colors

Definition 46 (Multichromatic Sorting). Multichromatic Sorting is universal sorting, where the elements are colored with $k$ different colors. All bichromatic comparisons have cost 1, and
for each color $i$, the monochromatic comparisons have a cost $\gamma_i > 1$, where $\gamma_i = \infty$ means that these comparisons are forbidden and stripes of this color are reported as a whole.

To solve the multichromatic setting, run a natural variant of (bichromatic) InversionSort: Pick one random element from each color, sort these, and take this as the backbone. Consider the remaining elements in random order. Place the element into the backbone using binary search. If this leads to a new element on the backbone (neighbors are of different colors), pivot the two current buckets. This maintains the invariants of the backbone and that three consecutive elements on the backbone have three different colors. Hence, all neighboring buckets are starting points of bi-chromatic problems, and we can use InversionSort on them. Here we interleave the computation as if the whole backbone was one backbone in the bichromatic setting.

The initial step of sorting the $k$ elements of all different colors, and continuing to create an initial multicolored backbone with $\ell$ elements on it, has expected cost $O(\ell \log \ell)$ by analogy to Quicksort. This is up to a constant factor optimal. The subproblems between these elements are solved with the $O(\log^4 n)$ instance optimality we showed for bichromatic / bipartite sorting in Theorem 21 and Theorem 22. Hence, by the decomposition lower bound, Lemma 18, we get the following.

**Theorem 47** (Instance Optimality of Multichromatic InversionSort). There exists a constant $c > 0$, such that for every instance $\mathcal{I}$, the cost of multichromatic InversionSort on $\mathcal{I}$, with probability at least $1 - 1/n$, is at most $c(\log n)^4 \text{OPT}(\mathcal{I})$, where $\text{OPT}(\mathcal{I})$ is the cost of the Hamiltonian.
E Bipartite Sorting with Hamiltonian

E.1 Backbonesort: Quicksort Adaptation for Perfectly Interleaved Instance

Consider the perfectly interleaved instance with \( n = m \): the smallest element is a red, followed by a blue, followed by a red, and so on. The canonical size vector is \((1, \ldots, 1)\).

**Definition 48 (Backbonesort).** Backbonesort has the same notion of a backbone as InversionSort. It runs in rounds that alternates between pivoting with red and pivoting with blue. In a round of pivoting with red (pivoting with blue is completely symmetrical), for each bucket of red elements, it chooses uniformly at random a red pivot. Now, each blue bucket is (attempted to be) split with the two red pivots of the neighboring red buckets, splitting it into 1 (not splitting), 2 or 3 blue buckets. For the new blue buckets that don’t have a pivot/representative on the backbone yet, choose one uniformly at random, and put them on the backbone. Finally, split all red buckets for which this makes sense using the newly chosen blue pivots.

**Theorem 49.** With probability at least \( 1 - 1/n \), Backbonesort runs in time \( O(n \log n) \) if the existence of a Hamiltonian path is promised, i.e., on the perfectly interleaved instance.

**Proof.** Consider an arbitrary red element \( r \). Consider one round of red pivoting, and let \( B \) be the bucket of \( r \) before the round, and \( B' \) after the round. Then, if \(|B| > 1\), with probability \( 1/8 \), \(|B'| \leq \frac{3}{4}|B|\): In the natural order of the elements, with probability \( 1/4 \), the randomly chosen pivot is in the middle, i.e. rank \( 1/4 \) to \( 3/4 \), of the bucket, and not in the same half as the current pivot/representative of the bucket. Now, the blue pivot is chosen uniformly between the old and the new pivot, and the probability of being in the middle half is at least \( 1/2 \) (the old pivot on the boundary, the new almost in the middle – everything else has an even higher probability, potentially even 1). Hence, with overall probability \( 1/8 \), \( B \) is split in the middle half, and \(|B'| \leq \frac{3}{4}|B|\).

From the claim follows from a proof similar to that of Theorem 39 that Backbonesort performs \( O(\log n) \) rounds. Because each round performs \( O(n) \) comparisons, the theorem follows.

E.2 InversionSort for bipartite sorting with Hamiltonian

While Theorem 21 shows only that InversionSort performs \( N(\log N)^{O(1)} \) comparisons on the perfectly interleaved instance, it is actually also optimal.

**Theorem 50 (InversionSort is optimal on perfectly interleaved).** With probability at least \( 1 - 1/n \), InversionSort runs in time \( O(n \log n) \) on the perfectly interleaved instance.

**Proof.** (Sketch) From Theorem 39 follows that the pivoting cost is as claimed. To bound the inversion cost, compute for each backbone (in retrospect / knowing the instance) the quantity \( Q \) as the sum of the squares of the sizes of the subproblems. If we can show that, with constant probability, \( Q \) is reduced by a constant factor in each step of the algorithm, the Theorem follows. Actually, it would be enough to show that each summand of \( Q \) is reduced by a constant factor with constant probability. While this is true for the unaffected
subproblems, it can fail for affected subproblems, i.e., subproblems that have a significant spill-in. Hence, let’s bundle together, for an unaffected subproblem $p$, all subproblems that it spills in, and all subproblems these spill in and so on. These are at two chains starting at $p$, one to the left and one to the right. By the definition of spill-in, and because the instance is the perfectly interleaved one, the sizes of the spilled-in problems decrease by a factor of 4, and the squares of the sizes by a factor of 16. Hence the contribution of $p$ to $Q$ is dominating the contribution of the bundle, and its probability to get substantially reduced is enough to show that the whole bundle has a constant probability to reduce its contribution to $Q$ by a constant factor.

The following theorem shows that Backbonesort is quite far from being instance optimal, in contrast to InversionSort.

**Theorem 51.** For every $n$, there exists a family of instances $I_n$ with $n$ red keys and $n$ blue keys, such that the expected number of comparisons performed by Backbonesort is $\Omega(n^{1.5})$, whereas the expected cost of $\text{OPT}(I_n) = O(n \log n)$.

**Proof.** Consider the instance $I$ given by the size vector $(n - \sqrt{n}, 1, 1, \cdots, 1, 1, n - \sqrt{n})$. The first stripe of reds is size $n - \sqrt{n}$, followed by $\sqrt{n}$ perfectly interleaved reds and blues, followed by a blue stripe of size $n - \sqrt{n}$. We call the big stripes at either ends “balloons”, to distinguish them from the singleton stripes in the middle.

**Analyzing Backbonesort on $I$:** Backbonesort will see as input two sets $R$ and $B$ of size $n$. First, it will pick a random pivot $r$ in $R$, and pivot the $B$ using this. With probability $1 - 1/\sqrt{n}$, this pivot will be from the red balloon, and with probability $1/\sqrt{n}$, it will be one of the reds in the singleton stripe. It is easy to see that until Backbonesort ends up picking a red or blue pivot not in the respective balloon, all its comparisons come out as red < blue. Thus Backbonesort will select a pivot from the balloons $\Omega(\sqrt{n})$ times in expectation before it selects a useful red element that is not in the balloon. However, since pivoting costs $O(n)$ comparisons every time, at this point Backbonesort has already performed $\Omega(n^{1.5})$ comparisons in expectation.

**Analyzing OPT on $I$:** We upper bound OPT by analyzing the running time of InversionSort on this instance. Assume that the first comparison of a red with a blue gives $r < b$, a likely scenario since these elements are more likely to come from the balloon than the singleton stripes. Inversion sort will create a backbone with a red pivot $r$ and a blue pivot $b$, and will try to find an inversion between them. Now there are $\Theta(n)$ inversions between the $\sqrt{n}$ reds and the $\sqrt{n}$ blues in the middle, so the probability of finding an inversion is $\Theta(n/n^2) = \Theta(1/n)$. Thus in expected time $O(n)$ InversionSort will find an inversion $y < x$, where $y$ is blue and $x$ is red, to have $r < y < x < b$ on the backbone.

At this point, the balloon instances on the two sides get separated, but they each spill-in into the middle subproblem, and thus the middle subproblem will be affected. It can be shown that with constant probability the middle subproblem will not affect the two balloon subproblems.

Assume $y$ is at position $k_1$ from the red balloon and $x$ is at position $k_2$ from the red balloon. Then the expected time to find an inversion in the left subproblem is $(n - \sqrt{n + k_1})k_1/(k_1)^2 \approx n/k_1$. Similarly the expected time to find an inversion in the right subproblem is $n/k_2$. For
the middle subproblem, its slightly worse: it will be \( n^2/(\sqrt{n} - k_1 - k_2)^2 \), which could be as large as \( n \).

When the side balloon get refined further, only their neighbors will get their spill-in, and the middle problems at the core start to become unaffected. At the extreme case, the balloons on the size only have \( O(1) \)-sized interleaving in the subproblem attached to them, and their expected cost to find an inversion becomes \( O(n) \).

However, at any point in time, if one considers the work done searching for an inversion\(^9\) on a layer of the decomposition tree, then the work on any subproblem does not exceed \( O(n) \) in expectation by the above argument. Using our general lemma on the depth of the tree being \( O(\log n) \) (Lemma 39) for any run of InversionSort, we get that the expected cost of InversionSort on \( I \) is \( O(n \log n) \). Furthermore, it is easy to see that the cost of InversionSort on all instances in \( \mathcal{N}_{AE}(I) \) is also \( O(n \log n) \) in expectation; these instances have roughly the same number of singleton stripes, and roughly the same sized-balloons. Thus we get that \( OPT(I) = O(n \log n) \) in expectation, proving the theorem.

\[ \]  

### E.3 Allowing Matches (Equality) as the outcome of a query

If one takes the classical nuts and bolts problem as the starting point of the considerations, it would be very natural to allow for the possibility of comparisons resulting in equality, a situation we call a match. Intuitively, such an outcome reveals much more information than a strict inequality, in particular because it allows, by transitivity, to find other equal elements of the same color. Here is a short sketch of the small adjustments to InversionSort needed for this setting. Note that the overall quality of InversionSort is not impacted. In a pivoting step we might find an equal element of the other color. We would use such pairs (or more) of equal elements to split the current subproblem on both colors. Such a split on a match is more benign than the splits by the normal representatives because there can never be a spill-in over such a pivot. Similarly, with respect to limiting the depth of the refinement tree, the possibility of matching elements does not disturb the considerations about balance. Therefore, the analysis of InversionSort becomes easier in this setting, and the same guarantees hold.

\[ ^9\text{As we showed in Theorem 40, the pivoting cost is always } O(n \log n). \]
Lower bound for finding the maximum with priced information

The instance from [15] is the following: there are two red nodes 1 and 2, and \( n - 2 \) blue nodes. There are \( n - 2 \) cost 0 edges that show 1 is greater than all the blue nodes. There are cost 1 edges between 2 and all blue nodes. There is a cost \( n \) edge between 1 and 2.

The instance distribution is, with equal probability, \( 1/n \), one of the following cases:

- 2 is the maximum (then 1, then all blue)
- 1 is maximum, then one of the blues (\( n - 1 \) cases), then 2, then all other blues

In the first case, the proof has cost \( n \), in each of the second cases, the proof has cost 1, it consist of the query that the special blue is greater than 2.

Any deterministic algorithm has the following costs: If it queries the edge between 1 and 2, it has cost \( n \). If it queries the edges between 2 and a blue in any order, the expected number of queries to find the special blue is \( n/2 \).

The expected competitive ratio can be calculated as follows. If the deterministic algorithm queries the 1 - 2 edge, the ratio is \( n/n = 1 \) in the first case, and \( n/1 = n \) in all other cases, giving an expected ratio of \( 1/n + (n - 1) \). If the algorithm does not query the 1-2 edge (and relies on the promise of the distribution that if all blue are smaller than 2, then 2 is the maximum), then the ratios are 1, 2, 3, \ldots, \( n - 1, n - 1 \), depending on which of the edges to the blues is the special one. This gives an expected ratio of roughly \( n/2 \). This implies that the competitive ratio of any deterministic algorithm on the distribution above is always \( \Omega(n) \). The same holds for any randomized algorithm too, by Yao’s principle.

Extension to sorting While this example shows that the competitive ratio to find the maximum is \( \Omega(n) \), this is not true, if we switch to sorting with the Hamiltonian promise. First, in order to extend this as a valid input to sorting, the blue elements need to be comparable at some cost, the same for all pairs. If we set the cost to 0, then the cost of the algorithm and the proof become 1, namely if there is a special blue, it is the maximum of the blue. If we set the cost to 1, the Hamiltonian has cost at least \( n - 3 \) between blue vertices, and this dominates the cost of finding the maximum; even more so if the blue-blue edge cost is set to \( n \). Hence, in all these cases and settings, the example is not a strong lower bound.