



A Polylogarithmic Competitive Algorithm for Stochastic Online Sorting and TSP

Andreas Kalavas  

Carnegie Mellon University, Pittsburgh, PA, USA

Charalampos Platanos  

National Technical University of Athens, Greece
Archimedes/Athena Research Center, Greece

Thanos Tolias  

National Technical University of Athens, Greece
Archimedes/Athena Research Center, Greece

Abstract

In *Online Sorting*, an array of n initially empty cells is given. At each time step t , an element $x_t \in [0, 1]$ arrives and must be irrevocably placed in an empty cell without knowledge of future arrivals. We aim to minimize the sum of absolute differences between pairs of elements placed in consecutive array cells, seeking an online placement strategy that results in a final array close to a sorted one. An interesting multidimensional generalization, referred to as the *Online Traveling Salesperson Problem*, arises when the request sequence consists of points in the d -dimensional unit cube and the objective is to minimize the sum of Euclidean distances between points in consecutive cells. Motivated by the recent work of (Abrahamsen, Bercea, Beretta, Klausen and Kozma; ESA 2024), we consider the *stochastic version* of Online Sorting (*resp.* Online TSP), where each element (*resp.* point) x_t is an i.i.d. sample from the uniform distribution on $[0, 1]$ (*resp.* $[0, 1]^d$). By carefully decomposing the request sequence into a hierarchy of balls-into-bins instances, where the balls to bins ratio is large enough so that bin occupancy is sharply concentrated around its mean and small enough so that we can efficiently deal with the elements placed in the same bin, we obtain an online algorithm that approximates the optimal cost within a factor of $O(\log^2 n)$ with high probability. Our result comprises an exponential improvement over the previously best known competitive ratio of $\tilde{O}(n^{1/4})$ for Stochastic Online Sorting due to (Abrahamsen et al.; ESA 2024) and $O(\sqrt{n})$ for (adversarial) Online TSP due to (Bertram, ESA 2025).

2012 ACM Subject Classification Theory of computation \rightarrow Online algorithms; Theory of computation \rightarrow Computational geometry

Keywords and phrases sorting, online algorithm, balls-into-bins, TSP

Digital Object Identifier 10.4230/LIPIcs.STACS.2026.58

Related Version *Full Version*: <https://arxiv.org/pdf/2508.12527>

Funding This work has been partially supported by project MIS 5154714 of the National Recovery and Resilience Plan Greece 2.0 funded by the European Union under the NextGenerationEU Program.

Acknowledgements The majority of this work was carried out while Andreas Kalavas was an intern at the Archimedes Research Unit. The authors would like to thank Dimitris Fotakis for many valuable discussions and insightful comments on this paper. They are also grateful to Marina Kontalexi for her helpful discussions.

1 Introduction

In the *Online Sorting Problem* we are given a sequence of n real numbers $x_1, x_2, \dots, x_n \in [0, 1]$, revealed one by one in an online fashion. An array A of length n is initially empty. Denote by $A[1], A[2], \dots, A[n]$ its cells. Upon the arrival of each element x_j , the algorithm must



© Andreas Kalavas, Charalampos Platanos, and Thanos Tolias;
licensed under Creative Commons License CC-BY 4.0

43rd International Symposium on Theoretical Aspects of Computer Science (STACS 2026).

Editors: Meena Mahajan, Florin Manea, Annabelle McIver, and Nguyễn Kim Thăng
Article No. 58; pp. 58:1–58:17



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



immediately and irrevocably assign it to an empty cell of A . After all n elements have been placed, the cost is defined as the total variation between consecutive entries *i.e.*, $\text{Cost}(A) = \sum_{i=1}^{n-1} |A[i+1] - A[i]|$. The objective is to minimize this cost. The problem was introduced by Aamand, Abrahamsen, Beretta, and Kleist [1] as a technical tool for proving lower bounds in online strip packing, bin packing, and perimeter packing. They studied the adversarial setting, where the sequence is chosen by an adaptive adversary, and designed an $O(\sqrt{n})$ -competitive algorithm along with a matching lower bound for deterministic algorithms. Later, Abrahamsen, Bercea, Beretta, Klausen, and Kozma [2] showed that even randomized algorithms cannot asymptotically improve this guarantee in the worst case.

Stochastic Online Sorting. Motivated by the large $\Omega(\sqrt{n})$ lower bound in the adversarial setting, Abrahamsen et al. [2] introduced *Stochastic Online Sorting*. There, the input elements x_1, \dots, x_n are drawn *i.i.d.* from the uniform distribution $\mathcal{U}(0, 1)$ and the algorithm seeks to minimize the cost incurred. Note that in the stochastic online sorting problem it is known that the cost of the offline optimal solution OPT is approximately equal to one for large enough n . Throughout this paper, we say that an algorithm for stochastic online sorting is c -competitive if it achieves cost at most $c \cdot \text{OPT}$ with high probability¹. Abrahamsen et al. [2], designed a $\tilde{O}(n^{1/4})$ -competitive algorithm, demonstrating that probabilistic assumptions can lead to significant improvements over the adversarial baseline.

Online TSP. Abrahamsen et al. [2] further generalized the problem by increasing its dimensionality, introducing the *Online Traveling Salesperson Problem*. Here the input elements x_1, \dots, x_n are points in $[0, 1]^d$, and the objective is to minimize the total Euclidean distance between consecutive points in the array. In the adversarial setting, they gave a dimension-dependent algorithm with competitive ratio $\sqrt{d} \cdot 2^d \cdot O(\sqrt{n \log n})$. Shortly after, Bertram [6] showed that dimension-independent bounds are achievable in the adversarial model presenting an $O(\sqrt{n})$ -competitive algorithm. To the best of our knowledge, the competitive ratio of the stochastic version of this problem has not been studied.

1.1 Our Contributions

In this work, we present a unified algorithmic framework, Algorithm 1, that applies to both stochastic online sorting and its multidimensional generalization, stochastic online TSP. Our framework carefully decomposes the input sequence into a series of balls-into-bins instances to achieve an $O(\log^2 n)$ approximation with high probability in both settings. We formally state our two main theorems below.

► **Theorem 1.** *Algorithm 1 achieves a cost of at most $O(\log^2 n)$ with high probability for the stochastic online sorting problem.*

► **Theorem 2.** *Algorithm 1 achieves a cost of at most $O(\log^2 n) \cdot \text{OPT}$ with high probability for the stochastic online TSP problem.*

Independent Work. In a recent parallel and independent work, Hu [11] studied Stochastic Online Sorting and obtained a deterministic algorithm with expected cost at most $\log n \cdot 2^{O(\log^* n)}$ and an elegant lower bound of $\Omega(\log n)$ on the expected cost of any randomized algorithm. Regarding high-probability bounds, which we aim to derive in this work, Hu [11, Theorem 1.3] presented an $O(\log^{20} n)$ -competitive algorithm.

¹ Throughout this work, “with high probability” means with probability at least $1 - 1/n$.

1.2 Technical Overview

An $O(\log^2 n)$ -Competitive Algorithm for Stochastic Online Sorting. The first main idea that Abrahamsen et al. [2] introduced is to divide the array into buckets / bins, exploit the randomness of the input in order to assign each element to a bucket, and solve the problem separately in each bucket. Suppose that each bucket has C elements; thus, we have n/C buckets, and the bucket i contains elements in $((i-1)\frac{C}{n}, i\frac{C}{n}]$. Then, assuming that each bucket will receive exactly the expected number of elements, *i.e.* C elements, the cost of the algorithm is approximately $\text{ALG} = \text{number of buckets} \times \text{cost inside bucket} = O(\sqrt{C})$, assuming that the $O(\sqrt{n})$ adversarial algorithm of Aamand et al. [1] was employed in each bucket.

Hence, smaller buckets reduce cost but increase the chance of imbalance, while larger buckets stabilize the load at the expense of higher internal cost. This trade-off is central to their analysis. To handle the imbalances, a part of the array in the end is reserved, also called backyard, and its size depends on how large these imbalances are. Since elements inserted in the backyard are arbitrary, the algorithm incurs a cost equal to the square root of its size. Thus, the previous trade-off translates into a trade-off between the size of the backyard and C : the larger C is, the smaller the backyard and vice versa. Their first algorithm balances this trade-off and achieves $\tilde{O}(n^{1/3})$ complexity. Recursively using their algorithm inside the buckets, they improve their competitive ratio to $\tilde{O}(n^{1/4})$.

Our approach to Stochastic Online Sorting can be naturally described through the lens of the balls-into-bins paradigm. More specifically, the algorithm proceeds by maintaining a partitioning of a certain part of A into equally sized *bins*, while the arriving elements can be regarded as *balls* placed into bins. Each bin is associated with a certain subinterval of $[0, 1]$, which determines if a new ball / element is placed into the particular bin. The subintervals associated with the bins form a partitioning of $[0, 1]$ and are of equal length. Hence, since the elements are *i.i.d.* samples from the uniform distribution on $[0, 1]$, each new ball is placed uniformly at random (and independently of the other balls) in each bin.

Interpreting the $\tilde{O}(n^{1/3})$ -competitive algorithm of Abrahamsen et al. [2] through the balls-into-bins framework above, we realize that the main limitation of their approach is that they consider a single balls-into-bins instance, which has very large bins and is fixed at the beginning of the algorithm. Therefore, the backyard of A must be large enough, in order to accommodate the significant imbalances in the bins' occupancy.

Our key new technical insight is that we can decompose an instance of Stochastic Online Sorting into a hierarchy of balls-into-bins instances (*a.k.a.* *phases*) of geometrically decreasing size. Crucially, we first distinguish between *buckets* and *bins*. A bucket is a contiguous set of cells corresponding to a single interval of the domain, while a bin is a collection of buckets that together cover the same interval but need not be contiguous in the array. It is important, that we select the bucket sizes dynamically, when we proceed from one phase to the next, in order to create same sized bins to deal with any imbalances (*i.e.*, empty cells in some buckets) left from the previous phase.

Our algorithm (see also Section 2 and Algorithm 1 for the details) maintains that all bins in the same phase are of equal size (and have equal probability of receiving a new element) and all bins created by the algorithm have size $\Theta(\log^2 n)$ (though the bucket size may slightly vary between phases). In the first phase, we consider the first $n/2$ cells of A , which we partition into $n/(2\log^2 n)$ buckets (we also partition $[0, 1]$ into the same number of subintervals with equal length). The current phase ends when the first of its buckets becomes full. A standard concentration bound (Lemma 5) shows that *w.h.p.* before the first phase ends: (i) at least $(1 - o(1))\frac{n}{2}$ elements are successfully placed into its bins; and (ii) every bin

has received at least $(1 - o(1)) \log^2 n$ elements. Using the (adversarial) $O(\sqrt{n})$ -competitive algorithm of Aamand et al. [1] to deal with the exact placement of the elements in the same bin we get that *w.h.p.* the total cost of the algorithm during its first phase is $O(\log n)$.

In the second phase, we consider the next $n/4$ cells of A which are partitioned into $n/(4 \log^2 n)$ buckets (as before, we partition $[0, 1]$ into the same number of subintervals with equal length). The bin size is slightly larger than in the first phase (but again at most $(1 + o(1)) \log^2 n$), because in the bins we also include cells from buckets left empty in the first half of A from the previous phase. As above, *w.h.p.* before the second phase ends: (i) at least $(1 - o(1)) \frac{n}{4}$ elements are successfully placed into its bins; (ii) every bin has received at least $(1 - o(1)) \log^2 n$ elements; and (iii) the total cost of the algorithm for its second phase is $O(\log n)$. Moreover, we show that due to property (ii) (combined for the first and the second phase), *w.h.p.* before the second phase ends, all cells in the first half of A (*i.e.*, the cells considered in the first phase) are full. So, we maintain the invariant that any imbalances in bucket occupancy left from the first (*resp.* any) phase do not carry over to the phases after the second (*resp.* next one).

Using the steps and the invariants (and carefully defining the exact quantities hidden in the $o(1)$ notation) above, our algorithm proceeds from one phase to the next, for $O(\log n)$ phases, until we are left with a single bucket of size $\Theta(\log^2 n)$. The total cost is dominated by the algorithm's total cost for the different phases and is $O(\log^2 n)$ *w.h.p.*

Extension to Stochastic Online TSP. When extending to higher dimensions, three challenges arise. First, unlike in the one-dimensional setting, there is no simple closed-form expression for OPT , and we must instead rely on getting a good estimate of OPT using concentration bounds. Second, the domain must be partitioned into blocks that both (i) have equal probability mass, to preserve the balls-into-bins property, and (ii) contain points that are sufficiently close so that the intra-block cost is negligible compared to OPT . Third, there is no obvious ordering of the blocks that guarantees low inter-block cost. To overcome these difficulties, we exploit properties of the uniform distribution, which allows us to partition the space into hyperboxes of similar geometry. We then define a tour that visits the input points block by block, with consecutive blocks chosen to be adjacent. The ordering of blocks is inspired by space-filling curves, a tool that has been widely used in the study of universal TSP (see [14, 7, 10, 8]). We prove that this structured tour is within a constant factor of the optimal TSP tour *w.h.p.*, and we adapt our algorithmic framework to approximate it using Bertram's algorithm [6]. As a result, we retain the $O(\log^2 n)$ guarantee in higher dimensions.

1.3 Other Related Work

Online Sorting with Larger Arrays. Another interesting variant of the *Online Sorting Problem* is one where the size of the array m is longer than the input sequence, *i.e.*, $m > n$. This version was introduced by Aamand et al. [1], who designed a deterministic $2\sqrt{\log n \sqrt{\log \log n + \log(1/\epsilon)}}$ -competitive algorithm when $m = (1 + \epsilon)n$. They complemented this with a lower bound, showing that every deterministic algorithm with $m = \gamma n$ is at least $1/\gamma \cdot \Omega(\log n / \log \log n)$ -competitive. Later, Azar et al. [4] and Nirjhor et al. [13], in independent and concurrent work, improved the upper bound, nearly resolving this variant.

Hashing Schemes. The connection between stochastic online sorting and hashing was already observed by Abrahamsen et al. [2]. Two particularly relevant examples are *Filter Hashing*, introduced by Fotakis et al. [9], and *Transactional Multi-Writer Cuckoo Hashing*, proposed by Kuzmaul [12]. Both hashing schemes employ a multi-layered structure where each subsequent layer is tasked with handling the imbalances of the previous layers.

Variants of Online TSP. Online TSP has also been studied in different contexts. A notable line of work interprets it as a scheduling problem: points (or requests) appear online in a metric space, and the objective is to minimize the time until all points have been visited [3]. In contrast, our setting focuses on minimizing the total length of the tour, rather than the completion time.

2 General Algorithmic Framework

Before we present our algorithm, we introduce some notation. For an array A we denote by $A[s : t]$ the subarray of A starting at s and ending at t . We also adopt a slight abuse of notation by using A to refer both to the array itself and to its length, with the intended meaning being clear from the context. We also define by $[k]$ the set $\{1, 2, \dots, k\}$. We call a subarray of a subarray of A a *sub-subarray* (or *bucket*).

2.1 Algorithm Description

We present an online sorting algorithm for the general setting in which we are given an array A of length n and a distribution \mathcal{D} over a domain \mathcal{S} , with sample access provided by `receive_sample`. We also assume access to the following subroutines.

Subroutines.

- **DomainPartitioning** $(\mathcal{D}, \mathcal{S}, \ell)$: This procedure takes as input a distribution \mathcal{D} over a domain \mathcal{S} and a positive integer ℓ . It outputs subsets $\{T_j^{(i)} \subseteq \mathcal{S} : i \in [\ell+1], j \in [2^{\ell-i+1}]\}$, collectively denoted by \mathcal{T} . The collection \mathcal{T} satisfies, for every level $i \in [\ell+1]$:
 1. *Covering*: $\bigcup_{j \in [2^{\ell-i+1}]} T_j^{(i)} = \mathcal{S}$.
 2. *Disjointness*: The sets $\{T_j^{(i)}\}_{j \in [2^{\ell-i+1}]}$ are pairwise disjoint.
 3. *Equal Mass*: $\mathbb{P}_{x \sim \mathcal{D}}[x \in T_j^{(i)}]$ is the same for all $j \in [2^{\ell-i+1}]$.
 4. *Laminarity*: \mathcal{T} forms a *laminar family* – every pair of sets in \mathcal{T} is either disjoint or one is contained in the other.
 Thus \mathcal{T} defines a hierarchical, binary-tree partition of \mathcal{S} , consisting of $2^{\ell+1} - 1$ subsets in total.
- **InBucketPlacement** (x, a) : This function takes as input an element $x \in \mathcal{S}$ and places it into an empty cell of sub-subarray a of A .
- **index** (\mathcal{T}, i, x) : This function takes as input a set \mathcal{T} outputted by **DomainPartitioning**, a positive integer i and an input element x and outputs the unique index k such that $x \in T_k^{(i)}$.
- **empty** (a) : This function takes as input a (sub-)subarray a of A and returns the number of empty cells in a at the current point of execution.

We also introduce the following definition:

► **Definition 3.** For a subarray A_i , let $N_i := \text{empty}(A_i)$ at the moment when the algorithm transitions from phase i to phase $i+1$, i.e., when some bucket of A_i becomes full and triggers an overflow.

The complete pseudocode is given in the Appendix of the full version; a more high-level version is presented in Algorithm 1. We now describe the algorithm step by step.

Algorithm 1 General Algorithmic Framework.

Data: Array $A[1 : n]$, Distribution \mathcal{D} over a domain \mathcal{S}
Result: success or fail

```

1  $i \leftarrow 0$ ;
2  $\ell \leftarrow \left\lceil \log \left( \frac{n}{2 \log^2 n} \right) \right\rceil$ ,  $K \leftarrow 2^\ell$ ;
3  $\mathcal{T} \leftarrow \text{DomainPartitioning}(\mathcal{D}, \mathcal{S}, \ell)$ ;
4 while  $A$  is not full do
5    $i \leftarrow i + 1$ ;  $K_i \leftarrow \frac{K}{2^{i-1}}$ ;
6   array  $A_i \leftarrow$  allocate  $\frac{n}{2^i}$  cells next to  $A_{i-1}$  ; /*  $A_0 \rightarrow$  array start */
7    $C_i \leftarrow \frac{A_i + N_{i-1}}{K_i}$  ; /* set the capacity of each bin */
8   for each  $j \in [K_i]$ ,  $c_j^{(i)} = C_i - \text{empty}(a_{2j-1}^{(i-1)}) - \text{empty}(a_{2j}^{(i-1)})$ 
9   if there exists  $j$ , s.t.  $c_j^{(i)} \leq 0$  then
10    return fail
11   divide  $A_i$  into  $K_i$  consecutive buckets  $a_j^{(i)}$  of sizes  $c_j^{(i)}$ ,  $j \in [K_i]$ ;
12   if remaining array places  $\leq 100 \log^2 n$  then
13     $K_i = 1$ ,  $A_i = a_1^{(i)} =$  remaining array places ; /* the last phase */
14   while there does not exist  $j$  s.t.  $a_j^{(i)}$  is full do
15      $x \leftarrow \text{receive\_sample}(\mathcal{D})$ ;
16      $k_1 \leftarrow \text{index}(\mathcal{T}, i-1, x)$ ,  $k_2 \leftarrow \text{index}(\mathcal{T}, i, x)$ ;
17     if  $a_{k_1}^{(i-1)}$  is not full then
18       InBucketPlacement( $x, a_{k_1}^{(i-1)}$ );
19     else
20       InBucketPlacement( $x, a_{k_2}^{(i)}$ );
21   if  $A_{i-1}$  is not full then
22     return fail
23 return success

```

Initialization. First, we define ℓ as the unique positive integer such that $\frac{n}{4 \log^2 n} < 2^\ell \leq \frac{n}{2 \log^2 n}$. The initial number of buckets for the first subarray and balls-into-bins instance is $K = 2^\ell$. Next, we partition the domain \mathcal{S} using the subroutine `DomainPartitioning`, which provides the partitioning of the domain into subsets (intervals / blocks) based on which we determine the bucket each element $x \in \mathcal{S}$ is inserted to. After completing the initializations, we move on to the first phase of the algorithm.

Phase 1. Let A_1 be the subarray containing the first $\lfloor \frac{n}{2} \rfloor$ elements of the original array. In the first phase we will place elements inside this subarray using a hash-based logic. We divide the A_1 elements into $K_1 = 2^\ell$ contiguous buckets, of size $C_1 = \frac{A_1}{K_1} = \Theta(\log^2 n)$ elements each. We place an arriving element $x \sim \mathcal{D}$ in the k -th bucket of A_1 if and only if $x \in T_k^{(1)}$, and then use the subroutine `InBucketPlacement` to position it within the bucket.

By definition of \mathcal{T} , each sample has equal probability of being placed into each bucket. Hence, placement of elements into the buckets of A_1 at the first phase of our algorithm can be thought as a balls-into-bins instance where we throw balls into K_1 bins, uniformly at random. When an element arrives and its designated bucket is full, we say that an overflow has occurred, triggering a transition to phase 2.

Phase 2. In order to handle overflows from subarray A_1 we allocate a subarray A_2 of size $\lfloor \frac{n}{4} \rfloor$ next to A_1 . Subarray A_2 is divided into $K_2 = K_1/2 = 2^{\ell-1}$ buckets. Regarding bucket sizes, crucially, the size of each bucket of A_2 is not the same, as discussed in the technical overview. To gain some intuition for the bucket sizes set in Line 8, we consider the following example.

Example. Suppose for simplicity's sake, that we are on the classic online sorting case and subarray A_1 has 100 cells and is divided into $K = K_1 = 4$ buckets, with capacity $C_1 = 25$ each, such that $T_1^{(1)} = (0, 0.25)$, $T_2^{(1)} = (0.25, 0.5)$, $T_3^{(1)} = (0.5, 0.75)$, $T_4^{(1)} = (0.75, 1)$ respectively. Assume that bucket $a_1^{(1)}$ is overflowed and the number of elements in each bucket are $(25, 20, 10, 15)$ at the time exactly before the overflow. Then, the up to this point empty, subarray A_2 will be brought into action to take care of the overflows of subarray A_1 . By design, A_2 has 50 cells and $K_2 = K_1/2 = 2$ buckets that handle elements $T_1^{(2)} = (0, 0.5)$, $T_2^{(2)} = (0.5, 1)$ respectively. If we allocate 25 cells to each bucket in A_2 , then we expect the first one to overflow significantly faster than the second one since in A_1 there are less empty cells for elements lying in $(0, 0.5)$ than for elements lying in $(0.5, 1)$. Thus, in order to handle the imbalances in A_1 we will allocate more cells in the first bucket in A_2 . The total number of empty cells in A_1 is $\text{empty}(A_1) = 0 + 5 + 15 + 10 = 30$ and since we add 50 cells with A_2 we have in total 80 cells – this is the size of the second balls-into-bins instance. Specifically, we can consider that the instance has 2 bins that handle elements from $(0, 0.5)$ and $(0.5, 1)$ respectively. The first bin contains the first two buckets of A_1 and the first bucket of A_2 . The second bin contains the last two buckets of A_1 and the second bucket of A_2 . We would like the capacities of the bins to be close enough in order to have a balanced balls-into-bins instance, thus equal to $\frac{\text{empty}(A_1) + A_2}{K_2} = 40$. Thus, we allocate $c_1^{(2)} = 40 - \text{empty}(a_1^{(1)}) - \text{empty}(a_2^{(1)}) = 35$ cells for the first bucket in A_2 and $c_2^{(2)} = 40 - \text{empty}(a_3^{(1)}) - \text{empty}(a_4^{(1)}) = 15$ cells for the second one. As a result, we obtain a new balanced balls-into-bins instance: the bins have equal capacity, so the expected overflow occurs later than it would if A_2 were simply divided into two equal-sized buckets.

This illustrates how A_2 's bucket sizes compensate for imbalances in A_1 . We now return to the formal description of phase 2. The capacity of each bin in the second balls-into-bins instance is thus:

$$C_2 = \frac{\text{size of instance}}{\text{number of bins}} = \frac{A_2 + \text{empty}(A_1)}{K_2} = \Theta(\log^2 n)$$

(see also Lemma 4) and each bucket has capacity as defined in Line 8 in order to create a balanced instance, *i.e.* $c_j^{(2)} = C_2 - \text{empty}(a_{2j-1}^{(1)}) - \text{empty}(a_{2j}^{(1)})$. If we cannot allocate such bucket size we say that our algorithm *failed*.

Regarding element placement, each bucket j in A_2 accepts elements from the subset $T_j^{(2)}$. As elements arrive online, we first attempt to place them in their designated bucket in A_1 ; if it is full, we place them in the corresponding bucket in A_2 . When an element arrives and its designated bucket in A_1 and A_2 is full, we transition to phase 3 and we say that a bucket in array A_2 overflowed. Crucially, if A_1 is not full at this point, we say that the algorithm has *failed*. In Section 2.2, we show that our algorithm does not fail with high probability.

Phases 3 and Beyond. The same logic extends to all subsequent phases, while the main invariant remains: each subarray fills before a bucket of the next subarray overflows, otherwise our algorithm fails. When the remaining places in the array are less than $100 \log^2 n$ we transition to the last phase of our algorithm. Note that for C_i in each phase it holds (its proof is in the Appendix of the full version):

► **Lemma 4.** *For each phase i , it holds that the capacity C_i of the i -th balls-into-bins instance is $\Theta(\log^2 n)$.*

Final Phase. In the final phase, the last subarray, denoted by B , is a single bucket, and elements are inserted using `InBucketPlacement`. Due to the above-discussed invariant, when B overflows (*i.e.*, when it is completely filled, as it has only one bucket), all previous subarrays are full. Thus, we have placed all elements into the array and the algorithm returns **success**.

Number of Phases. Let R be the number of phases of our algorithm and let B be the last subarray A_R . For R it holds that $B + \sum_{i=1}^{R-1} \lfloor \frac{n}{2^i} \rfloor = n$ with $B \leq 100 \log^2 n$. Hence,

$$\sum_{i=1}^{R-1} \frac{n}{2^i} \geq n - 100 \log^2 n \Rightarrow n(1 - 2^{-R}) \geq n - 100 \log^2 n \Rightarrow R \geq \log \left(\frac{n}{100 \log^2 n} \right)$$

and since $n - \sum_{i=1}^{R-2} \lfloor \frac{n}{2^i} \rfloor > 100 \log^2 n$ we get that $R \leq 1 + \log \left(\frac{n}{100 \log^2 n} \right)$, thus $R = \Theta(\log \frac{n}{\log^2 n})$. Also, the size of A_{R-1} is $n/2^{R-1} = \Theta(\log^2 n)$ and for B it holds that $B \geq n - n(1 - 2^{-R}) = \Theta(\log^2 n)$, thus B and A_{R-1} are asymptotically tight, preserving the invariant.

2.2 Probability of failure

Note that sometimes our algorithm might fail. In this section we upper bound the probability of failure of our algorithm. There are two cases where our algorithm could fail. The first one is at line 10 of Algorithm 1 and the other is at line 23. To show that the failure probability $\mathbb{P}[\text{fail}]$ is small, we first present the following lemma, which is central to the analysis of the balls-into-bins instances arising in our problem.

► **Lemma 5.** *Suppose a balls-into-bins instance with K total bins, each with capacity $C = \Theta(\log^2 n)$, and let $M = K \cdot C \leq n$ be the total number of balls that all bins can collectively hold. If the probability that an element belongs to each bin is the same, then the first overflow of a bin happens after $M - \frac{M}{\Theta(\log^{1/2} n)}$ balls are thrown w.p. at least $1 - K/n^2$. Moreover, after throwing $M + \frac{M}{\Theta(\log^{1/2} n)}$ balls, all bins are full w.p. at least $1 - K/n^2$.*

The proof of the lemma is deferred to the Appendix of the full version. We proceed by defining three types of events such that, conditioning on them, our algorithm cannot fail. We will conclude our proof that $\mathbb{P}[\text{fail}]$ is small by showing that these events occur with high probability.

► **Definition 6.** *Consider the creation of the i -th balls-into-bins instance during the algorithm's execution. Denote by $M_i := A_i + N_{i-1}$ its size and by C_i its bins' capacity. We define the following events:*

- \mathcal{O}_i : A_i overflows after $M_i - \frac{M_i}{\Theta(\log^{1/2} n)}$ balls have been thrown in this instance
- \mathcal{F}_i : A_i is fully filled after $M_i + \frac{M_i}{\Theta(\log^{1/2} n)}$ balls have been thrown in this instance
- \mathcal{G}_i : each bin of the instance has received at least $C_i - \frac{C_i}{\Theta(\log^{1/3} n)}$ balls after $M_i - \frac{M_i}{\Theta(\log^{1/2} n)}$ balls have been thrown in this instance

and let $\mathcal{E}_i = \mathcal{O}_i \cap \mathcal{F}_i \cap \mathcal{G}_i$ and $\mathcal{E} = \cap_i \mathcal{E}_i$.

We now handle the first case of failure by proving the following lemma (the proof is in the Appendix of the full version).

► **Lemma 7.** *Conditioned on \mathcal{E} , it holds that $\forall j, C_i > \text{empty} \left(a_{2j-1}^{(i-1)} \right) + \text{empty} \left(a_{2j}^{(i-1)} \right)$, for each phase i of our algorithm.*

We now proceed by handling the second case of failure. First we introduce some definitions.

► **Remark 8.** Note that for phase i , the size of its balls-into-bins instance is $M_i = A_i + N_{i-1}$.

► **Definition 9.** For each phase i , define as T_i the number of elements inserted during phase i until the first overflow in A_i and T'_i as the number of elements inserted since the beginning of phase i until A_i becomes full.

We continue by proving the following lemma.

► **Lemma 10.** Conditioned on \mathcal{E} , for each phase i , when the i -th balls-into-bins instance overflows, subarray A_{i-1} is full.

Proof. We proceed using strong induction (see also Figure 1 for a visual representation of the proof).

Base case, $i = 1$. We aim to show that the first subarray A_1 is filled before any overflow occurs in the second balls-into-bins instance, that is, $T_2 \geq T'_1 - T_1$, conditioned on \mathcal{E} . For the first phase, it holds $T_1 = A_1 - N_1$ (there is no previous phase). Thus, the size of the second balls-into-bins instance is $A_2 + N_1$. Now, using Definition 6, observe that:

$$\begin{aligned} T_2 - (T'_1 - T_1) &\geq M_2 - \frac{M_2}{\Theta(\log^{1/2} n)} - \left(M_1 + \frac{M_1}{\Theta(\log^{1/2} n)} - A_1 + N_1 \right) \\ &\geq A_2 + N_1 - \frac{A_2 + N_1}{\Theta(\log^{1/2} n)} - \left(A_1 + \frac{A_1}{\Theta(\log^{1/2} n)} - A_1 + N_1 \right) \\ &= A_2 - \frac{A_2 + N_1}{\Theta(\log^{1/2} n)} - \frac{A_1}{\Theta(\log^{1/2} n)} \\ &> A_1/4 - o(A_1) \geq 0, \end{aligned}$$

since $A_2 > A_1/4$, thus conditioned on \mathcal{E} we obtain the desired result, proving the base case.

Induction step. Assume the statement holds for each $i \in [r-1]$. We aim to prove that it also holds for $i = r$. From induction hypothesis, since we condition on \mathcal{E} , every subarray A_j for $j \in [r-1]$ becomes completely filled before any bucket in A_{j+1} overflows. As a result, when phase $r+1$ starts, every subarray A_j for $j \in [r-1]$ is filled. This is crucial, since it implies that every new sample that arrives is inserted in this balls-into-bins instance; thus we can use Definition 6 (otherwise the elements inserted into the instance are not necessarily uniform in bins). The size of the balls-into-bins instance of phase $r+1$ is $A_{r+1} + N_r$. Now, using Definition 6 again, observe that:

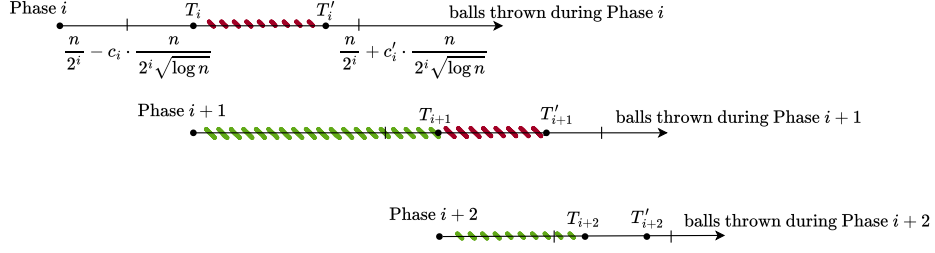
$$T_{r+1} - (T'_r - T_r) \geq M_{r+1} - \frac{M_{r+1}}{\Theta(\log^{1/2} n)} - \left(M_r + \frac{M_r}{\Theta(\log^{1/2} n)} - M_r + \frac{M_r}{\Theta(\log^{1/2} n)} \right) \quad (1)$$

$$= A_{r+1} + N_r - \frac{A_{r+1} + N_r}{\Theta(\log^{1/2} n)} - 2 \cdot \frac{A_r + N_{r-1}}{\Theta(\log^{1/2} n)} \quad (2)$$

$$\geq A_{r+1} - 6 \cdot \frac{A_{r-1}}{\Theta(\log^{1/2} n)} \quad (\text{since } A_{r+1} < A_r < A_{r-1} \text{ and } A_i > N_i) \quad (3)$$

$$\geq A_{r-1}/8 - o(A_{r-1}) \geq 0, \quad (4)$$

where we have used that $A_i = \Theta(A_{i-1})$, completing the proof of the induction step and thus proving the lemma. ◀



■ **Figure 1** A timeline of the phases of our algorithm during execution. Note how the inequality $T'_i - T_i \leq T_{i+1}$ is preserved.

We establish the following lemma, whose proof is deferred to the Appendix of the full version.

► **Lemma 11.** *It holds that $\mathbb{P}[\neg \mathcal{E}] \leq \frac{1}{n}$.*

Since \mathcal{E} implies that our algorithm will not fail we get that our algorithm does not fail with high probability since:

$$\mathbb{P}[\text{fail}] \leq \mathbb{P}[\neg \mathcal{E}] < \frac{1}{n}$$

► **Remark 12.** For the full Algorithm where we use \tilde{C}_i some buckets can have capacity $\lfloor C_i \rfloor$ and others $\lfloor C_i \rfloor + 1$, thus $\tilde{C}_i \geq \lfloor C_i \rfloor$. For the time of the first overflow, from Lemma 5 we get that w.p. $1 - \frac{K_i}{n^2}$: $T_i \geq K_i \lfloor C_i \rfloor - \frac{K_i \lfloor C_i \rfloor}{\Theta(\log^{1/2} n)} \geq K_i C_i - K_i - \frac{K_i C_i}{\Theta(\log^{1/2} n)} \geq M_i - \frac{M_i}{\Theta(\log^{1/2} n)}$.

Similarly, for the other event, the capacity of each bin in the i -th balls-into-bins instance is at most $1 + \lfloor C_i \rfloor$. Thus for the time where all buckets are full, from Lemma 5 we get that w.p. $1 - \frac{K_i}{n^2}$: $T'_i \leq K_i(1 + \lfloor C_i \rfloor) + \frac{K_i(1 + \lfloor C_i \rfloor)}{\Theta(\log^{1/2} n)} \leq M_i + \frac{M_i}{\Theta(\log^{1/2} n)}$.

3 Stochastic Online Sorting

In this section, we specialize Algorithm 1 for the classical stochastic online sorting case where $\mathcal{D} = \mathcal{U}(0, 1)$ and $\mathcal{S} = [0, 1]$. We first, define subroutines `DomainPartitioning` and `InBucketPlacement`.

■ `DomainPartitioning`($\mathcal{U}, [0, 1], \ell$): For $j \in [2^\ell]$, let:

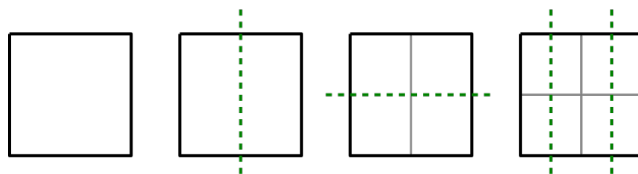
$$T_j^{(1)} = [(j-1)/2^\ell, j/2^\ell) \text{ and } T_j^{(i)} = T_{2j-1}^{(i-1)} \cup T_{2j}^{(i-1)}$$

■ `InBucketPlacement`(x, a): We use the deterministic adversarial algorithm \mathcal{A}_{adv} of [1] to place elements within each bucket.

We are now ready to prove the first main theorem of this work.

► **Theorem 1.** *Algorithm 1 achieves a cost of at most $O(\log^2 n)$ with high probability for the stochastic online sorting problem.*

Proof. We begin by conditioning on the success of our algorithm, which ensures that each bucket in every subarray stores exactly the elements belonging to its designated interval. We distinguish four sources of cost: the *intra-bucket* cost, the *inter-bucket* cost, the *cost of connecting different subarrays* and the cost of the final *subarray B*.



■ **Figure 2** A visual representation of our splitting procedure on the plane. Uniform distribution ensures that splitting a block in half creates two blocks of equal probability mass.

- **Intra-bucket:** Consider subarray A_i . Since we use \mathcal{A}_{adv} algorithm for placement in each bucket, we incur total cost for the subarray: $\sum_{j=1}^{K_i} \sqrt{C_i} \cdot \frac{1}{K_i} = \Theta(\log n)$ and since we have $R - 1 = O(\log n)$ such subarrays in total we incur total cost $O(\log^2 n)$.
- **Inter-bucket:** Consider subarray A_i . Since, by definition, each bucket contains elements strictly smaller than the next, the total cost to “connect them” is at most $\sum_{j=1}^{K_i-1} \left(\frac{j+1}{K_i} - \frac{j}{K_i} \right) \leq 2$, thus in total for all subarrays: $O(\log n)$.
- **Between subarrays:** Between subarrays we incur cost at most 1 and since we have $R = O(\log n)$ transitions this cost is $O(\log n)$.
- **Subarray B :** The elements are inserted in B using \mathcal{A}_{adv} thus we incur total cost $\Theta(\log n)$, since B has $\Theta(\log^2 n)$ elements.

As a result, by conditioning on the success of our algorithm, we obtain total cost $O(\log^2 n)$, thus the total cost is $O(\log^2 n)$ with probability at least $1 - 1/n$, concluding the proof of the theorem. ◀

▶ **Remark 13.** Our algorithm is conceptually simple to state and analyze, and benefits from a hierarchical decomposition of Stochastic Online Sorting into balls-into-bins instances. The algorithm and the decomposition above can be naturally extended to non-uniform distributions. The only additional step we need to take care of is to partition the $[0, 1]$ interval into as many intervals as required in each phase so that there is equal probability that a new point arrives in each interval. Furthermore, by a more careful analysis and parameter selection, we can show an upper bound of $O(\log^{3/2+\epsilon} n)$ on the total cost of the algorithm *w.h.p.* for Stochastic Online Sorting. Finally, by choosing a sufficiently large constant c in the proof of Lemma 5, we obtain an even stronger high-probability guarantee, e.g. $1 - 1/n^{100}$.

4 Stochastic Online TSP

We now extend our approach for stochastic online sorting to the *Stochastic Online TSP* problem, obtained by increasing the dimensionality of the input. We show that Algorithm 1 can be adapted to preserve the one-dimensional guarantees in arbitrary dimension d .

▶ **Theorem 2.** *Algorithm 1 achieves a cost of at most $O(\log^2 n) \cdot OPT$ with high probability for the stochastic online TSP problem.*

4.1 DomainPartitioning

We partition the domain into hyperboxes, which we call *blocks*. Let ℓ be the unique positive integer such that $\frac{n}{4 \log^2 n} < 2^\ell \leq \frac{n}{2 \log^2 n}$. As in the one-dimensional case, the final block covers the entire domain, *i.e.*, $T_1^{(\ell)} = \mathcal{S}$. To construct the partition of $T^{(i)}$, we apply the split procedure from Algorithm 2 to $T^{(i+1)}$. Each split doubles the number of blocks, so after ℓ

splits, $T^{(1)}$ consists of 2^ℓ blocks. The splits proceed cyclically over the dimensions: the first along the first dimension, the second along the second, and so forth. After d splits, we return to the first dimension and continue this process until all ℓ splits have been performed.

■ **Algorithm 2** SPLIT(\mathcal{B}, i).

Data: Current set of blocks \mathcal{B} , splitting dimension i
Result: Updated set of blocks after splitting

```

1  $\mathcal{B}' \leftarrow \emptyset;$ 
2 for  $b \in \mathcal{B}$  do
3   Split  $b$  into two equal halves  $b_1, b_2$  along dimension  $i$ ;
4    $\mathcal{B}' \leftarrow \mathcal{B}' \cup \{b_1, b_2\};$ 
5 return  $\mathcal{B}'$ 

```

Elementary Blocks. We refer to the blocks $T^{(1)} = \{b_1, b_2, \dots, b_{2^\ell}\}$ at the lowest level of our hierarchy as *elementary blocks*. Within an elementary block, all points are treated identically by Algorithm 1: by construction, the algorithm's decisions do not depend on which specific point from the block is presented. Consequently, we may assume that the points drawn within an elementary block b are *i.i.d.* samples from the uniform distribution $\mathcal{U}(b)^2$.

Block order. At this stage, we impose an order on the blocks. Consecutive blocks are assigned to consecutive buckets/sub-subarrays. In addition, they are constructed so that they merge early during the execution of our algorithm. This motivates a further property: consecutive blocks must be neighbours, *i.e.*, they share a face (or, more generally, a hyperface). We present Algorithm 3, which traverses the blocks in such an order, and establish its correctness in Lemma 14 which is formally proven in the Appendix of the full version.

■ **Algorithm 3** ORDER(n_1, \dots, n_d).

Data: Number of blocks n_i along each dimension $i \in [d]$
Result: Traversal order of all blocks

```

1  $v \leftarrow (1, 1, \dots, 1);$  /* coordinates of the first block */
2  $L \leftarrow [v];$  /* list of visited block coordinates */
3  $m \leftarrow (1, 1, \dots, 1);$  /* move direction in each dimension */
4 while  $\neg(v[1] = 1 \wedge \dots \wedge v[d-1] = 1 \wedge v[d] = n_d)$  do
5    $j \leftarrow 1;$ 
6   while  $v[j] + m[j] \notin [1, n_j]$  do
7      $m[j] \leftarrow -m[j];$  /* reverse direction in dim.  $j$  */
8      $j \leftarrow j + 1;$ 
9    $v[j] \leftarrow v[j] + m[j]$ , Append  $v$  to  $L$ ;
10 return  $L$ ;

```

► **Lemma 14.** *Algorithm 3 visits every block exactly once, and any two consecutive blocks in the order are adjacent.*

² Note that $\mathcal{U}(b)$ is the restriction of the global distribution \mathcal{D} to block b .

4.2 InBucketPlacement

Once mapped to a bucket, an element is placed in an empty cell using Bertram’s adversarial algorithm [6].

4.3 Cost Analysis

Similarly to the stochastic online sorting case, we distinguish three sources of cost: the *intra-bucket* cost, the *inter-bucket* cost, and the *cost of connecting different subarrays*. Since absolute values are not informative, all bounds will be expressed in terms of OPT. Before estimating OPT and analyzing each cost source separately, we recall several key results that will be used throughout the analysis.

► **Theorem 15** ([5]). *Given sufficiently large n , the expected length of a TSP tour with n points drawn i.i.d. from $\mathcal{U}([0, \Delta]^d)$ is $\beta_d \cdot n^{1-1/d} \cdot \Delta$, where $\beta_d \approx \sqrt{\frac{d}{2\pi e}}$. We denote this quantity by $TSP(n, d, \Delta)$.*

► **Corollary 16.** *Since OPT represents the length of the optimal tour of n points drawn i.i.d. from $\mathcal{U}([0, 1]^d)$ it is $\mathbb{E}[OPT] = TSP(n, d, 1) = \beta_d \cdot n^{1-1/d}$, where the randomness is over the drawn instance.*

► **Proposition 17.** *The following properties hold:*

1. *For all $m < n$ and all d, Δ , we have $TSP(m, d, \Delta) < TSP(n, d, \Delta)$.*
2. *For any convex $S \subseteq [0, \Delta]^d$, let $\text{tour}(S, n)$ denote the expected length of the optimal TSP tour on n points drawn uniformly i.i.d. in S . Then, $\text{tour}(S, n) \leq TSP(n, d, \max_{x, y \in S} \|x - y\|_\infty) \leq TSP(n, d, \Delta)$.*

► **Lemma 18** (Chapter 2, [15]). *Let T be the length of the TSP of n points drawn i.i.d. in $[0, \Delta]^d$. It holds for some constant c :*

$$\mathbb{P}[|T - TSP(n, d, \Delta)| \geq t] \leq 2 \exp\left(-\frac{ct^2}{n^{1-2/d}\Delta^2}\right)$$

Estimating the Optimal Cost. In contrast to the one-dimensional case, there is no simple closed-form expression for the optimal cost in higher dimensions. We therefore rely on asymptotic estimates and concentration bounds to control the value of OPT.

► **Lemma 19.** *With probability at least $1 - 2 \exp(-c'dn)$, the optimal TSP tour satisfies*

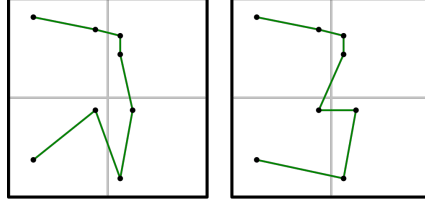
$$OPT \geq \frac{1}{2} \beta_d n^{1-1/d}.$$

We now proceed to the analysis of the three sources of cost.

Intra-Bucket Cost.

► **Lemma 20.** *Let OPT_i be the length of the optimal tour of $6 \log^2 n$ points drawn uniformly at random from elementary block b_i . Then,*

$$\mathbb{P}\left[OPT_i \geq 72\beta_d n^{-1/d} \log^2 n\right] \leq \frac{2}{n^2}.$$



■ **Figure 3** The domain $[0, 1] \times [0, 1]$ is partitioned into four blocks. The left panel shows the block-by-block tour produced by our algorithm, while the right panel shows the optimal TSP tour on the same instance. Although the two tours may differ, our partitioning ensures that their lengths remain close with high probability.

Proof.

Step 1: Block side length. Since the domain is split ℓ times, each dimension is split at least $\lfloor \ell/d \rfloor$ times³, thus the maximum side length of a block is at most $(1/2)^{\lfloor \ell/d \rfloor}$. As ℓ and d are integers, $(1/2)^{\lfloor \ell/d \rfloor} \leq (1/2)^{(\ell-d)/d} < \left(\frac{4 \log^2 n}{n}\right)^{1/d} \cdot 2 \leq 4 \left(\frac{\log^2 n}{n}\right)^{1/d}$. Hence, each side of b_i has length at most $4 \left(\frac{\log^2 n}{n}\right)^{1/d}$.

Step 2: Expected cost. The expected length of the optimal subtour inside b_i satisfies:

$$\mathbb{E}[\text{OPT}_i] \leq \text{TSP}(6 \log^2 n, d, \Delta) \leq \beta_d (6 \log^2 n)^{1-1/d} \Delta,$$

where Δ is the side length of b_i . Substituting $\Delta \leq 4 \left(\frac{\log^2 n}{n}\right)^{1/d}$ gives:

$$\mathbb{E}[\text{OPT}_i] \leq 24 \beta_d n^{-1/d} \log^2 n.$$

Step 3: Concentration. Applying Lemma 18 with $t = 48 \beta_d n^{-1/d} \log^2 n$, we bound the probability that OPT_i exceeds three times its expectation:

$$\mathbb{P}\left[\text{OPT}_i \geq 72 \beta_d n^{-1/d} \log^2 n\right] \leq 2 \exp\left(-\frac{c'' d n^{-2/d} (\log^2 n)^2 \cdot n^{2/d}}{\log^2 n}\right) \leq \frac{2}{n^2}. \quad \blacktriangleleft$$

► **Corollary 21.** *With probability at least $1 - \frac{1}{n \log^2 n}$, every one of the $2^\ell \leq \frac{n}{2 \log^2 n}$ elementary block tours has length at most $72 \beta_d n^{-1/d} \log^2 n$.*

So far, we proved that the length of the optimal tour of $6 \log^2 n$ points uniformly chosen in a block is bounded by $72 \beta_d n^{-1/d} \log^2 n$ with probability at least $1 - 2/n^2$. We want to transition from this to a bound of the optimal total intra bucket cost of a subarray j , denoted by $\text{INBUCKET}(j)$. For the first subarray, elementary blocks are mapped one to one to buckets and every bucket contains strictly less than $6 \log^2 n$ points therefore Corollary 21 implies that $\text{INBUCKET}(1) \leq 72 \beta_d n^{-1/d}$ with probability at least $1 - \frac{1}{n \log^2 n}$. We now show that any subsequent (fixed) subarray enjoys a comparable bound.

► **Lemma 22.** *Fix an arbitrary phase j and an arbitrary block $B_k^{(j)}$ of phase j . Block $B_k^{(j)}$ consists of some elementary blocks b_{k_1}, \dots, b_{k_m} . Fix any realization of points and let $r(B)$ denoted the realized points inside block B ,*

$$\text{INBUCKET}(j) = \sum_{i=1}^{2^{\ell-j+1}} \text{OPT}(r(B_i^{(j)})) \leq \sum_{i=1}^{2^\ell} \text{OPT}(r(b_i)) + \beta_d n^{1-1/d}.$$

where $\text{OPT}(S)$ denotes the length of the optimal tour of the points in set S .

³ We assume that $\ell \geq d$. In the full version we remove this assumption.

Proof. A feasible tour inside $r(B_k^{(j)})$ is obtained by concatenating the optimal tours of $r(b_{k_1}), \dots, r(b_{k_m})$ and adding at most $(r - 1)$ connecting edges between adjacent elementary blocks. Summed over all phase j blocks, the total number of such connectors is at most the number of elementary blocks. By Lemma 28, each connector has length at most $2\beta_d n^{-1/d} \log^2 n$, and by Corollary 30 the total intra-subarray connection cost is deterministically bounded by $\beta_d n^{1-1/d}$. This yields the claimed inequality. ◀

At this point we restate Lemma 4 slightly changing the phrasing to fit current context.

► **Lemma 23.** *For every subarray j , each bucket has capacity less than $6 \log^2 n$ points.*

► **Corollary 24.** *Each subarray gets at most $6 \log^2 n$ points from any elementary block.*

► **Lemma 25.** *For any $t \in \mathbb{R}$ and any subarray j , it holds $\mathbb{P}[OPT_i \geq t] \geq \mathbb{P}[OPT(r(b_i)) \geq t]$.*

Proof. The cardinality of $r(b_i)$ is a random variable. However, by Corollary 24, $r(b_i)$ contains fewer than $6 \log^2(n)$ points. In both cases, the points are independently drawn uniformly at random within the same block. The proof then follows from the observation that the probability of the optimal tour length of x i.i.d. points in some space S exceeding a threshold t is non-decreasing in the number of points x . ◀

► **Proposition 26.** *Fix any subarray j . With probability at least $1 - \frac{1}{n \log^2 n}$, it is $INBUCKET(j) \leq 73\beta_d n^{1-1/d}$.*

Proof. Lemma 25 combined with Lemma 20 implies $\mathbb{P}[OPT(r(b_i)) \geq 72\beta_d n^{-1/d} \log^2 n] \leq \frac{2}{n^2}$. Taking a union bound over different elementary blocks and adding the connecting term completes the proof. ◀

► **Corollary 27.** *By taking a union bound over the subarrays we get that, with probability at least $1 - \frac{1}{n \log n}$, for every subarray j it holds $INBUCKET(j) \leq 146OPT$.*

Inter-Bucket Cost.

► **Lemma 28.** *The cost of connecting two neighbouring blocks is at most*

$$2\sqrt{d} \cdot 4 \left(\frac{\log^2 n}{n} \right)^{1/d} \leq 2\beta_d n^{-1/d} \log^2 n.$$

► **Remark 29.** Lemma 28 applies to elementary blocks. When moving to higher phases, blocks are formed by merging pairs of blocks from the previous phase. As a result, the diameter of a block (and hence the cost of connecting two neighboring blocks) increases by at most a factor strictly smaller than 2 from one phase to the next. However, at the same time, the number of blocks to be connected decreases by a factor of 2 at each phase. Therefore, the total cost incurred by connecting all blocks within a subarray remains of the same order, which leads to Corollary 30.

► **Corollary 30.** *Using the observation above, the cost of connecting all blocks within a subarray is deterministically bounded by $\beta_d \cdot n^{1-1/d}$.*

Cost of Connecting Subarrays.

► **Lemma 31.** *The cost of connecting two subarrays is at most OPT .*

Proof. Connecting subarrays requires adding a single edge between two points, one from each subarray. Since the optimal tour OPT already spans all points in the domain, the additional cost of this connection is trivially bounded above by OPT . ◀

Putting Everything Together. We are now ready to prove Theorem 2. We decompose the total cost into four components:

- **Intra-bucket:** For each subarray A_j , placements inside buckets are handled by Bertram’s algorithm [6]. This incurs cost $\sum_{i=1}^{K_j} \sqrt{C_j} \cdot \text{OPT}(r(B_i)) \leq \Theta(\log n) \sum_{i=1}^{2^\ell} \text{OPT}_i = \Theta(\log n)\text{OPT}$. where the first inequality is derived from Lemma 22 and the second from Lemma 20. Since there are $R = O(\log n)$ subarrays, the total bucket cost is $O(\log^2 n)\text{OPT}$.
- **Inter-bucket:** By Lemma 28 and Corollary 30, the additional cost of connecting all buckets within a subarray A_i is at most $O(\text{OPT})$. Summing over all $k = O(\log n)$ subarrays yields a total of $O(\log n)\text{OPT}$.
- **Between subarrays:** Connecting consecutive subarrays requires at most one additional edge per transition. By Lemma 31, this incurs cost at most OPT per transition. With $k = O(\log n)$ subarrays, the total cost is $O(\log n)\text{OPT}$.
- **Subarray B :** The remaining elements are placed in B using Bertram’s algorithm [6], incurring total cost $O(\log n)\text{OPT}$, since $|B| = O(\log^2 n)$.

Conditioning on the high-probability event that Corollary 21 holds for all subarrays A_i and conditioning on the event that the algorithm does not fail, the total cost of the algorithm is $O(\log^2 n)\text{OPT}$. This completes the proof of Theorem 2.

5 Conclusion and Open Directions

In this work, we study the stochastic variants of Online Sorting and Online TSP, obtaining polylogarithmic upper bounds in both settings. Analyzing the problem through the lens of the balls-into-bins paradigm reveals exponential improvements over previous approaches. A natural direction is to extend our bounds to general known distributions in higher dimensions. At the same time, our framework also assumes full knowledge of the distribution. Relaxing this assumption prompts several questions: What guarantees are possible when points are drawn *i.i.d.* from an *unknown* distribution, or when inputs arrive in uniformly random order with no distributional assumptions?

References

- 1 Anders Aamand, Mikkel Abrahamsen, Lorenzo Beretta, and Linda Kleist. Online Sorting and Translational Packing of Convex Polygons. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 1806–1833. SIAM, 2023. doi:10.1137/1.9781611977554.CH69.
- 2 Mikkel Abrahamsen, Ioana O. Bercea, Lorenzo Beretta, Jonas Klausen, and László Kozma. Online Sorting and Online TSP: Randomized, Stochastic, and High-Dimensional. In Timothy M. Chan, Johannes Fischer, John Iacono, and Grzegorz Herman, editors, *32nd Annual European Symposium on Algorithms, ESA 2024, September 2-4, 2024, Royal Holloway, London, United Kingdom*, volume 308 of *LIPICs*, pages 5:1–5:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICs.ESA.2024.5.
- 3 Giorgio Ausiello, Esteban Feuerstein, Stefano Leonardi, Leen Stougie, and Maurizio Talamo. Algorithms for the on-line travelling salesman, August 1999.
- 4 Yossi Azar, Debmalya Panigrahi, and Or Vardi. Nearly tight bounds for the online sorting problem, 2025. doi:10.48550/arXiv.2508.14287.
- 5 Jillian Beardwood, J. H. Halton, and J. M. Hammersley. The shortest path through many points. *Mathematical Proceedings of the Cambridge Philosophical Society*, 55(4):299–327, 1959. doi:10.1017/S0305004100034095.
- 6 Christian Bertram. Online Metric TSP, 2025. doi:10.48550/arXiv.2504.17716.

- 7 Dimitris Bertsimas and Michelangelo Grigni. Worst-case examples for the spacefilling curve heuristic for the euclidean traveling salesman problem. *Operations Research Letters*, 8(5):241–244, 1989. doi:10.1016/0167-6377(89)90047-3.
- 8 George Christodoulou and Alkmini Sgouritsa. An improved upper bound for the universal TSP on the grid. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1006–1021. SIAM, 2017. doi:10.1137/1.9781611974782.64.
- 9 Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. In *STACS 2003, 20th Annual Symposium on Theoretical Aspects of Computer Science, Berlin, Germany, February 27 - March 1, 2003, Proceedings*, pages 271–282, February 2003. doi:10.1007/3-540-36494-3_25.
- 10 Mohammad T. Hajiaghayi, Robert Kleinberg, and Tom Leighton. Improved lower and upper bounds for universal tsp in planar metrics. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm, SODA '06*, pages 649–658, USA, 2006. Society for Industrial and Applied Mathematics.
- 11 Yang Hu. Nearly optimal bounds for stochastic online sorting, 2025. doi:10.48550/arXiv.2508.07823.
- 12 William Kuzmaul. Fast concurrent cuckoo kick-out eviction schemes for high-density tables, 2016. arXiv:1605.05236.
- 13 Jubayer Nirjhor and Nicole Wein. Improved online sorting, 2025. doi:10.48550/arXiv.2508.14361.
- 14 Loren K. Platzman and John J. Bartholdi. Spacefilling curves and the planar travelling salesman problem. *J. ACM*, 36(4):719–737, 1989. doi:10.1145/76359.76361.
- 15 J. Michael Steele. *Probability Theory and Combinatorial Optimization*. Society for Industrial and Applied Mathematics, 1997. doi:10.1137/1.9781611970029.