Improved Lower Bounds for Online Hypercube Packing

Sandy Heydrich[1,2] and Rob van Stee[3]

[1] Max Planck Institute for Informatics, Saarbrücken, Germany
[2] Graduate School for Computer Science, Saarbrücken, Germany
heydrich@mpi-inf.mpg.de
[3] University of Leicester, Leicester, UK
rvs4@le.ac.uk

Abstract. Packing a given sequence of items into as few bins as possible in an online fashion is a widely studied problem. We improve lower bounds for packing hypercubes into bins in two or more dimensions, once for general algorithms (in two dimensions) and once for an important subclass, so-called Harmonic-type algorithms (in two or more dimensions). Lastly, we show that two adaptions of the ideas from the best known one-dimensional packing algorithm [9] to square packing also do not help to break the barrier of 2.

1 Introduction

In this paper, we consider the problem of online hypercube packing. This problem is defined as follows: We receive a sequence of hypercubes $h_1, \ldots, h_n$ (called *items*) in d-dimensional space, and each item h_i has a certain edge length s_i. We furthermore have an infinite number of bins, which are hypercubes of edge length one. We have to assign each item h_i to a bin and a position $(x_1, \ldots, x_d)$ inside this bin, such that $0 \leq x_j \leq 1 - s_i$ for all $1 \leq j \leq d$ and no two items in the same bin are overlapping. Items must be placed parallel to the axes of the bins. We call a bin *used* if at least one item is assigned to it, and our goal is to minimize the number of used bins. The online setting requires us to assign an item to a bin immediately when it arrives, without knowledge of future items. We consider this problem in two or more dimensions.

For measuring the quality of a solution of the algorithm, we use the standard notion of *asymptotic performance ratio*. For an input sequence σ, let $\mathcal{A}(\sigma)$ be the number of bins algorithm $\mathcal{A}$ uses to pack the items in σ and let $OPT(\sigma)$ be the minimum number of bins in which these items can be packed. The asymptotic performance ratio for A is defined as

$$R_{\mathcal{A}}^{\infty} = \limsup_{n\to\infty} \sup_{\sigma} \left\{ \frac{\mathcal{A}(\sigma)}{OPT(\sigma)} \middle| OPT(\sigma) = n \right\}$$

If $\mathcal{O}$ denotes a class of packing algorithms, then the optimal asymptotic performance ratio for class $\mathcal{O}$ is defined as $R_{\mathcal{O}}^{\infty} = \inf_{\mathcal{A}\in\mathcal{O}} R_{\mathcal{A}}^{\infty}$. From now on, we will only talk about asymptotic performance ratios, although we omit the word asymptotic.

1.1 Previous Results

The classic online bin packing problem in one dimension was first considered by Ullman [17], and he also gave the FirstFit algorithm with performance ratio $\frac{17}{10}$ [11]. The NextFit algorithm was introduced by Johnson [10], who showed that this algorithm has a performance ratio of 2.

The Harmonic algorithm was introduced by Lee and Lee [12]. If we define $u_1 = 2, u_{i+1} = u_i(u_i - 1) + 1$, then this algorithm has performance ratio $h_\infty = \sum_{i=1}^{\infty} \frac{1}{u_i - 1} < 1.69104$. It uses bounded space (i.e. only a constant number of bins are open at a time, meaning that items can be added to them) and they showed that no algorithm with this property can perform better. Later, various improvements of this approach were given (using unbounded space), including RefinedHarmonic (performance ratio $\frac{373}{228} < 1.63597$) [12], ModifiedHarmonic (performance ratio < 1.61562) and ModifiedHarmonic2 (performance ratio < 1.61217) by Ramanan et al. [15], Harmonic++ (performance ratio < 1.58889) by Seiden [16], and finally SonOfHarmonic (performance ratio < 1.5815) by Heydrich and van Stee [9]. The best general lower bound of 1.54037 for online bin packing in one dimension was given by Balogh et al. [1].

Online bin packing of rectangles was first discussed by Coppersmith and Raghavan [2]. They gave an algorithm which has in two dimensions a performance ratio of $\frac{13}{4}$ for general rectangles and $\frac{43}{16}$ for squares. Additionally, they showed a lower bound of $\frac{4}{3}$ for square packing in any dimension $d \geq 2$. Csirik and van Vliet improved upon this by giving an algorithm that achieves h_∞^d performance ratio for any dimension $d \geq 2$ [3]. They also show that this is a lower bound for bounded space algorithms, although their algorithm uses unbounded space. Later, Epstein and van Stee provided a bounded space algorithm that matches this lower bound [5]. In the same paper, they also give an optimal online bounded space algorithm for box packing (i.e. items are not hypercubes anymore but can have different sizes in different dimensions), although they cannot provide the exact performance ratio. Finally, Han et al. [7] gave an upper bound of 2.5545 for the special case of $d = 2$, which is the best bound currently known.

The best known lower bounds for hypercube packing are 1.6406 for two dimensions and 1.6680 for three dimensions [4]. Regarding upper bounds, the best algorithm for square packing achieves a performance ratio of 2.1187 and the best algorithm for cube packing achieves 2.6161 [8].

1.2 Our Contribution

We improve the general lower bound for square packing in two dimensions to 1.6707. The previous lower bound [4] was constructed by using an input such that any item size in this input divides all larger item sizes. In this way, a lower bound could be proved analytically since it could be proved that only very few relevant patterns remained. The key idea behind our improvement is that any given input (with non-divisible item sizes) in a lower bound construction can be extended in a greedy manner, in such a way that the patterns can be easily

determined using a formula without the need for an exhaustive search. This exhaustive search is then only needed for the first part of the input sequence, which makes it feasible. This allows us to use inputs with up to ten different item sizes, including very small sizes, which were infeasible to determine patterns for using the known approaches. We also made several algorithmic improvements to the pattern search, dramatically improving the running time for many important patterns.

Furthermore, we improve the lower bound for Harmonic-type algorithms in any dimension $d \geq 2$. This uses a generalization of the method of Ramanan et al. [15]. In particular, we show that such an algorithm cannot break the barrier of 2 for $d = 2$, by giving a lower bound of 2.02 for this case. This shows that substantial new ideas will be needed in order to improve significantly on the current best upper bound of 2.1187 and get close to the general lower bound. Our lower bound tends to 3 for large numbers of dimensions.

Lastly, we also show that when incorporating two central ideas from the currently best one-dimensional bin packing algorithm [9] into two-dimensional square packing, similar lower bounds as those for Harmonic-type algorithms can be achieved.

1.3 Preliminaries

At several points in this paper, we use the notion of *anchor points* as defined by Epstein and van Stee [6]. We assign the coordinate $(0, \ldots, 0)$ to one corner of the bin, all edges connected to this corner are along a positive axis and have length 1. Placing an item at an anchor point means placing this item parallel to the axes such that one of its corners coincides with the anchor point and no point inside the item has a smaller coordinate than the corresponding coordinate of the anchor point. We call an anchor point *blocked* for type s items in a certain packing (i.e. in a bin that contains some items), if we cannot place an item of type s at that anchor point (without overlapping other items).

2 Lower Bound for General Algorithms in Two Dimensions

2.1 Van Vliet's Method

For deriving a general lower bound on the performance ratio of online hypercube packing algorithms, we extend an approach by van Vliet [18] based on linear programming. Problem instances considered in this approach are characterized by a list of items $L = L_1 \ldots L_k$ for some $k \geq 2$, where each sublist L_j contains $\alpha_j \cdot n$ items of side length s_j (we will also call such items "items of size s_j" or simply "s_j-items"). We assume $s_1 \leq \ldots \leq s_k$. The input might stop after some sublist. An online algorithm $\mathcal{A}$ does not know beforehand at which point the input sequence stops, and hence the asymptotic performance ratio can be lower

bounded by

$$R \geq \min_{\mathcal{A}} \max_{j=1,\ldots,k} \limsup_{n\to\infty} \frac{\mathcal{A}(L_1,\ldots,L_j)}{OPT(L_1,\ldots,L_j)}$$

For this approach, we define the notion of a *pattern*: A pattern is a multiset of items that fits in one bin. We denote a pattern by a tuple $(p_1,\ldots,p_k)$, where p_i denotes the number of s_i-items contained in the pattern (possibly zero). We call a pattern p *dominant* if the multiset consisting of the items of p and an additional item of the smallest item size that is used by p cannot be packed in one bin. The performance of an online algorithm on the problem instances we consider can be characterized by the number of bins it packs according to a certain pattern. Van Vliet denotes the set of all feasible patterns by T, which is the union of the disjoint sets $T_1,\ldots,T_k$ where T_j contains patterns whose first non-zero component is j (i.e., whose smallest item size used is s_j). We can then calculate the cost of an algorithm $\mathcal{A}$ by $\mathcal{A}(L_1,\ldots,L_j) = \sum_{i=1}^{j}\sum_{p\in T_i} n(p)$, where $n(p)$ denotes the number of bins $\mathcal{A}$ packs according to pattern p. Note that we only need to consider dominant patterns in the LP [18]. As the variables $n(p)$ characterize algorithm $\mathcal{A}$, optimizing over these variables allows us to minimize the performance ratio over all online algorithms with the following LP:

$$\begin{array}{lll}
\text{minimize} & R & \\
\text{subject to} & \displaystyle\sum_{p\in T} p_j \cdot x(p) \geq \alpha_j & 1 \leq j \leq k \\
& \displaystyle\sum_{i=1}^{j}\sum_{p\in T_i} x(p) \leq \lim_{n\to\infty} \frac{OPT(L_1,\ldots,L_j)}{n} R & 1 \leq j \leq k \\
& x(p) \geq 0 & \forall p \in T
\end{array}$$

In this LP, the variables $x(p)$ replace $n(p)/n$, as we are only interested in results for $n \to \infty$. Note that item sizes are always given in nondecreasing order to the algorithm. From now on, however, we will consider item sizes in *nonincreasing* order for constructing the input sequence and generating all patterns.

2.2 Greedy Extension

A common heuristic for generating difficult inputs for online algorithms is to choose the item sizes s_j and the values α_j so that a multiset containing exactly $\alpha_j \cdot n$ squares with sides s_j for all $j = 1,\ldots,k$ can be packed together in a single bin. Finding these values α_j for given sides $s_1,\ldots,s_k$ is a challenging problem in itself, on which more in the next section. This problem becomes much easier however if we restrict attention to item sizes that divide *all* previous item sizes. In this case we can use *canonical* packings to determine immediately exactly how many items can be packed into a bin containing also larger items.

We define the notion of canonical packings. Let $\hat{s}$ be the smallest item size. In a canonical packing, all items are placed at anchor points, which are defined as all points having all coordinates equal to $i \cdot \hat{s}$ for some $i \in \{0,\ldots,\lfloor \frac{1}{\hat{s}} \rfloor - 1\}$.

In total, we have $\left\lfloor \frac{1}{s} \right\rfloor^d$ such anchor points. If a multiset of items can be packed, it can also be packed by a canonical packing, as shown by [6].

For a given input containing j item sizes $s_1, ..., s_j$ with $s_1 \geq \ldots, \geq s_j$, we can greedily extend it as follows. Let s' be the largest value so that $s'|s_i$ and $s' < s_i$ for $i = 1, \ldots, j$. Then the next item size will be s', and the number of times this item will appear is given by the following lemma.

Lemma 1. *Let $(p_1, \ldots, p_j)$ be a pattern (not necessarily dominant) for item sizes $s_1, \ldots, s_j$ with $s_1 \geq \ldots \geq s_j$. Let s' be the largest number such that the sizes $s_1, \ldots, s_j$ are integer multiples of s' and $\forall 1 \leq i \leq j : s_i > s'$. Then, $(p_1, \ldots, p_j, p')$ is a dominant pattern for the sizes $s_1, \ldots, s_j, s'$ where*

$$p' = \left\lfloor \frac{1}{s'} \right\rfloor^d - \sum_{i=1}^{j} p_i \left(\frac{s_i}{s'} \right)^d$$

Proof. Let P be a canonical packing for the pattern $(p_1, \ldots, p_j)$. Every item of size s_i for $i = 1, \ldots, j$ blocks $\left(\frac{s_i}{s'}\right)^d$ anchor points. Moreover, as s' divides all the s_i and all items are placed at anchor points, such an item fills *exactly* the space between this number of anchor points in every dimension. At every unoccupied anchor point, we can place one item of size s'. Hence, in total we can add $\left\lfloor \frac{1}{s'} \right\rfloor^d - \sum_{i=1}^{j} p_i \left(\frac{s_i}{s'}\right)^d$ items of size s' to this packing. As we cannot add more items of type s', this pattern is dominant. □

We can inductively extend this lemma to obtain the following corollary:

Corollary 1. *Let $(p_1, \ldots, p_j)$ be a pattern for item sizes $s_1, \ldots, s_j$. We define additional item sizes $s_{j+1}, \ldots, s_k$ for $k > j$ such that for all $i = 1, \ldots, k-j$, the sizes $s_1, \ldots, s_{j+i-1}$ are integer multiples of s_{j+i}, and $s_{j+i} < \min\{s_1, \ldots, s_{j+i-1}\}$. Moreover, each s_{j+i} for $1 \leq i \leq k-j$ is the largest value that satisfies this condition. Then, $(p_1, \ldots, p_k)$ is a dominant pattern for the sizes $s_1, \ldots, s_k$ where*

$$p_{j+i} = \left\lfloor \frac{1}{s_{j+i}} \right\rfloor^d - \sum_{l=1}^{j+i-1} p_l \left(\frac{s_l}{s_{j+i}} \right)^d$$

For any given input with j different item sizes, we can extend it as described in Corollary 1; we call this a *greedy extension*. Additionally assume that when constructing a packing for the extension, we place the items in order of decreasing size. Let us call the items in the extension *small* items. Crucially, for every pattern which includes one or more item sizes of the extension, we can restrict our attention to dominant small-greedy patterns. A pattern is small-greedy if the largest *small* item in it appears as many times as it can fit in a bin together with all the preceding items, and this also holds for all smaller items.

Definition 1. *Let $p = (p_1, \ldots, p_j, p_{j+1}, \ldots, p_k)$ be a pattern for item sizes $s_1 \geq s_2 \geq \ldots \geq s_k$, where for all $i = 1, \ldots, k-j$, the sizes $s_1, \ldots, s_{j+i-1}$ are integer multiples of s_{j+i} and $s_{j+i} < \min\{s_1, \ldots, s_{j+i-1}\}$. p is small-greedy if for all $i = 1, \ldots, k-j$ with $p_i > 0$ we have that the tuple $(p_1, \ldots, p_{j+i-1}, p_{j+i} + 1, 0, \ldots, 0)$ is not a feasible pattern.*

We can prove analogously to [4, Lemma 4] that any dominant pattern that is not small-greedy is a convex combination of dominant patterns that are small-greedy. The only property that is required is that each size of a small item divides all larger item sizes (including larger small item sizes).

Lemma 2. *For item sizes $s_1, \ldots, s_k$ where $s_{j+1}, \ldots, s_k$ are the greedy extension of $s_1, \ldots, s_j$ as described above, any dominant pattern that is not small-greedy is a convex combination of dominant patterns that are small-greedy.*

Proof. We define the notion of an *anchor point for a small item type i*: these are all points with coordinates equal to $l \cdot s_i$ for some $0 \leq l \leq \left\lfloor \frac{1}{s_i} \right\rfloor$. Note that these anchor points for any small type coincide with the corners of all larger items that were placed before.

We do induction in order to construct a convex combination of small-greedy patterns for a given dominant pattern p. The induction hypothesis is as follows: the vector that describes the numbers of items of the t smallest types which appear in the pattern is a convex combination of small-greedy vectors for these types. Call such a pattern t-greedy.

Definition 2. *Let $p = (p_1, \ldots, p_k)$ be a pattern for item sizes $s_1, \ldots, s_k$. Let $i_l \in \{1, \ldots, k-j\}$ be the largest index s.t. $p_{j+i_l} > 0$ and there are $t - l$ larger indices $i_{l'}$ with $p_{j+i_{l'}} > 0$.*

Let $i_1, \ldots, i_t \in \{1, \ldots, k-j\}$ be indices the largest indices with $p_{i_l} > 0$ for $l = 1, \ldots, t$. Let r be the vector that has p_{i_l} at index i_l for $l = 1, \ldots, t$ and zero elsewhere. p is called t-greedy if r is a convex combination of small-greedy patterns.

The base case is $t = 1$. We consider the items of the smallest type that occurs in p. Since p is dominant, for this type we have that as many items as possible appear in p, given the larger items. Thus p is 1-greedy.

We now prove the induction step. Suppose that in p, small items of type i appear fewer times than they could, given the larger items. Moreover, p contains items of some smaller type. Let i' be the largest smaller type in p (this is also a small item type, i.e. one added by the extension). By induction, we only need to consider patterns in which all the items of type less than i that appear, appear as many times as possible, starting with items of type i'. (All other patterns are convex combinations of such patterns.)

We define two patterns p' and p'' such that p is a convex combination of them. p' is defined as follows: modify p by removing all items i and adding the largest smaller item that appears in p, of type i', $\left(\frac{s_i}{s_{i'}}\right)^d$ times per each item i. When creating p', we thus add the maximum amount of items of type i' that can fit for each removed item of type i. p is greedy with respect to all smaller items, and $s_{i'}$ divides s_i as it is a small items size (one that was added by the extension). Therefore the multiset p' defined in this way is a pattern, and is $(t+1)$-greedy.

p'' on the other hand is created by adding items of type i to p and removing items of type i'. In particular, in the canonical packing for p, at each anchor point for type i that is not removed due to a higher-type item, we place an item of size s_i and remove all items that overlap with this item. Since all items smaller than s_i appear as many times as possible given the larger items, all the removed items are of the next smaller type i' that appear in p. This holds because the items are packed in order of decreasing size, and all corners of larger items are anchor points. Hence, if an anchor point is free, an item of type i' was put there.

In p'', the number of items of type i is now maximized given items of higher types. Only type i' items are removed, and only enough to make room for type i, so type i' remains greedy. Thus p'' is $(t+1)$-greedy. Each time that we add an item i, we remove exactly $\left(\frac{s_i}{s_{i'}}\right)^d$ items of type i'. So by adding an item i in creating p'', we remove exactly the same number of items of type i' as we add when we remove an item i while creating p'. Therefore, p is a convex combination of p' and p'', and we are done. □

It is therefore straightforward to list *all* the dominant patterns for an input with a greedy extension, as soon as we have determined the full set of patterns (including the non-dominant ones!) for the first (non-greedy) part of the input. This is the topic of the next section.

2.3 Finding Patterns

The main obstacle in executing van Vliet's method in more than one dimension is to find all the dominant patterns for a given input sequence. In general, it is NP-hard to determine whether a given set of squares can be packed into a single bin [13]. Recently, this was shown to be true for packing cubes as well [14]; we expect this to be true for higher dimensions as well.

Epstein and van Stee [6] describe a computer program called F that checks for a particular set of items whether they can be packed in a bin. They make the observation that in any feasible packing, we can shift all items to the left and to the bottom as far as possible, maintaining a feasible packing. This program F uses so-called *available positions*. Initially, there is only one such position, namely $(0, 0)$. Every time that we place an item at an available position, this position is removed and two new available positions are created: the top left corner and the lower right corner of the newly placed item. However, we do not necessarily place an item exactly at a given available position: if possible, we shift the item to the left and to the bottom as far as possible. This shifting must be performed for every item, as the distance that we can shift it might depend on the item size. The program follows essentially these steps for every available position and every item size that needs to be packed:

- Calculate the shifted position from an available position A and item size s_i
- Place the current item i at that shifted position
- If i is not completely inside the bin, overlaps with another item, or the item can be shifted further left or down (this depends on its size), skip this position

– Remove the available position A and create two new ones (on the top left corner and the bottom right corner of i)
– Decrease p_i by one (i.e., p always contains the number of items that still need to be packed, ignoring already packed items)
– Try recursively to pack the rest of the pattern
– If this is successful, return the packing found; otherwise, remove the item i, restore the old available positions, and add one to p_i

We extended and speeded up this computer program. Our modifications are listed below. The program finally outputs a matlab and/or maple file that contains the LP for all computed dominant patterns.

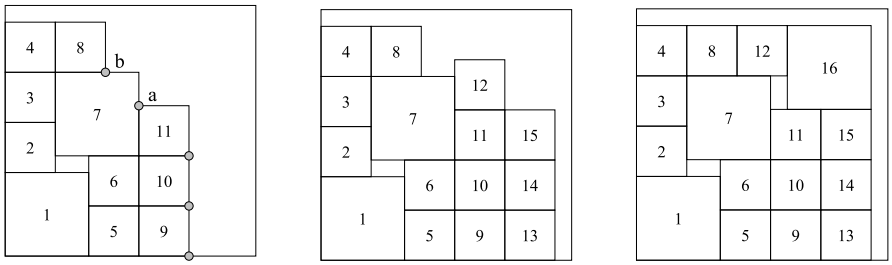
Parallelization The program was parallelized to give a speed up on multicore computers. Different patterns are now tested by different threads. We implemented a monitor, that computes which patterns have to be tested (i.e. which could be feasible due to simple constraints like total area), and a set of workers that get one such pattern at a time and try to pack it. Of course, one has to carefully avoid race conditions.

We had to be careful how to schedule the processing of different patterns. For example, in order to calculate how many items of a certain item type can be added to a pattern that does not contain this type, we use information from other patterns. To be precise, if we want to add items of size s_4 to a pattern $(p_1, p_2, p_3, 0)$ with $p_1, p_2, p_3 > 0$, and we have a pattern $(p_1, p_2, 0, p_4^*)$, then we know that we can add at most $p_4^* - p_3$ items of the smallest size to $(p_1, p_2, p_3, 0)$. Furthermore, if we also have a pattern $(p_1, p_2, p_3 - 1, p_4^*)$, we can reduce this bound to $p_4^* - \lfloor s_3/s_4 \rfloor^d$. That means, a worker might have to wait until the workers that test these other patterns are done, in order to avoid unnecessary work. The same is true when reusing packings from other workers (see below).

Special Patterns Patterns with only two non-zero components (i.e. patterns that use only items of two sizes; we call them *special patterns*) are packed differently. Say we have p_i items of size s_i and p_j items of size s_j that should be packed in one bin, and say w.l.o.g. $p_i \leq p_j$.

The order in which available positions are considered changes what the final packing looks like. Our program uses the following approach: The list of available positions initially contains only the point $(0, 0)$, and if we place an item of side length s at position (x', y') which is obtained by shifting an available position (x, y), we replace (x, y) by $(x', y' + s)$ and add $(x' + s, y')$ at the end of the list of available positions.

Any feasible pattern can be packed in the order in which our program does it, using the list of available positions. From now on, we only consider this fixed order. This only leaves the question of what to pack in every step. There are *three* options in every step: pack an item of size s_i, pack an item of size s_j, or pack nothing at all and move to the next available position. Obviously, sometimes an available position is too close to an edge of the bin and we see that no item can fit



(a) Situation after packing the first 11 items

(b) Continuing with the next available position

(c) Leaving out the next available position

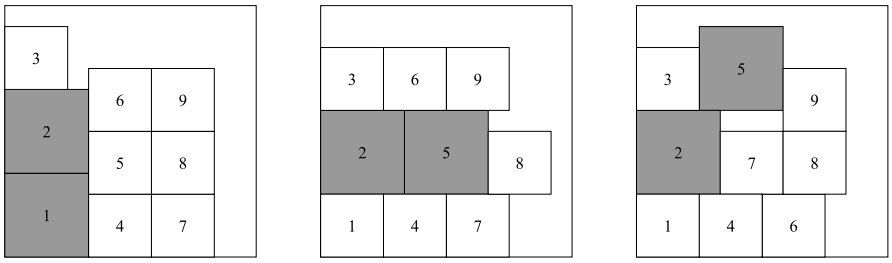
Fig. 1: An example where an available position needs to be left out even though an item could be placed there. We want to pack 3 items of size 1/3 and 13 items of size 1/5. The leftmost image depicts the situation after packing the first eleven items. Available positions are marked by gray circles. The next available position is the one labeled a. If we pack the next item (size 1/5) there, we see quite easily that we cannot add the third item of size 1/3 (middle). However, if we leave out this available position and instead continue with the next one (labeled b), we can pack all items (right).

there. However, see Fig. 1 for an illustration of an instance where it is necessary to skip an available position even though an item could be packed there.

For a feasible pattern, we can now specify its packing very succinctly, by only writing down when to use the least common item. That means, we can write down an ordered sequence $r = (r_1, \ldots, r_{p_i})$ containing all t such that the t-th item packed is an s_i-item. Such a sequence is called a position set for this pattern. For example, if we want to pack $p_1 = 2$ items of size $s_1 = 1/3$ and $p_2 = 7$ items of size $s_2 = 1/4$, the packing in Fig. 2a has the position set $r = (1, 2)$. Note that such a position set does not describe one unique packing, as it does not specify which available positions to leave out. In Fig. 2b and 2c, $r = (2, 5)$, although the packings are different (in Fig. 2b, an item is packed whenever it is possible; in Fig. 2c, we leave out the available position $(1/3, 1/4)$, which is the lower right corner of item 2).

We can now try to actually pack the items by using all specifications of this form. Here we also need to do backtracking since we still have two options for every available position: pack the next item or pack nothing. We use the analogous approach for patterns with three different item types, where we have to maintain two position sets for the two types with fewer items.

Reusing Packings We try to reuse packings of other patterns as much as possible. When we successfully pack a pattern, we store this packing so that we can start from it if we later try to pack a pattern which contains one additional item. We also output all packings as a human-readable text document, so that



(a) A packing for position set $(1, 2)$

(b) A packing for position set $(2, 5)$

(c) Another packing for position set $(2, 5)$

Fig. 2: Illustration of position sets for special patterns. Grey items have size 1/3, white items have size 1/4. Items are packed in the denoted order.

the packings can be verified by humans as well. Similarly, for special patterns as described above, we try to test promising position sets first, i.e. position sets that were used successfully for similar patterns earlier.

2.4 Results

We ran this program with different item sizes as input. The best bound was achieved for the input sequence $1/2, 1/4, 1/5, 1/10$ that was then extended greedily to $1/2, 1/4, 1/5, 1/10, 1/20, 1/40, 1/80, 1/160, 1/320, 1/640$ (each item size actually misses a $+\epsilon$ that we omitted for readability). Choosing α-values 1, 5, 7, 4, 8, 77, 157, 317, 637, 1277, we obtained the lower bound 1.6707. Note that these α values correspond to the pattern that contains for each size in decreasing order as many items as can be added to all larger items (i.e. start with one 1/2-item, add as many 1/4-items as possible (five), then add as many 1/5-items as possible (seven), and so on).

Theorem 1. *For two dimensional hypercube packing, no online algorithm can achieve an asymptotic performance ratio better than 1.6707.*

Table 1 gives an overview over some lower bounds we obtained with other inputs.

3 Lower Bound for Harmonic-Type Algorithms

Now, we consider the hypercube packing problem in d dimensions, for any $d \geq 2$. We define the class $C^{(h)}$ of Harmonic-type algorithms analogous to [15]. An algorithm $\mathcal{A}$ in $C^{(h)}$ for any $h \geq 1$ distinguishes, possibly among others, the following disjoint subintervals

Input sequence	Alpha values	Lower bound
1/2, 1/3, 1/4, 1/5, 1/60, 1/120, ..., 1/960	1, 3, 2, 2, 643, 237, ...	1.5839
1/2, 1/3, 1/4, 1/12, 1/24, ..., 1/768	1, 3, 2, 19, 45, ...	1.6277
1/2, 1/3, 1/6, 1/7, 1/42, 1/84, ..., 1/672	1, 3, 4, 11, 60, ...	1.6642
1/2, 1/3, 1/6, 1/12, ..., 1/1536	1, 3, 4, 21, 45, ...	1.6443
1/2, 1/4, 1/5, 1/20, 1/40, ..., 1/1280	1, 5, 7, 24, 77, ...	1.6593

Table 1: Other inputs tried with our program from section 2.3 and resulting lower bounds.

- $\overline{I}_1 = (1 - y_1, 1]$
- $I_{1,j} = (1 - y_{j+1}, 1 - y_j]$, for every $j \in \{1, \ldots, h\}$
- $\overline{I}_2 = (y_h, 1/2]$
- $I_{2,j} = (y_{h-j}, y_{h-j+1}]$, for every $j \in \{1, \ldots, h\}$
- $I_\lambda = (0, \lambda]$

for some parameters y_j and λ, where $1/3 = y_0 < y_1 < \ldots < y_h < y_{h+1} = 1/2$ and $0 < \lambda \leq 1/3$. For convenience, we assume that all y_j are rational.

Algorithm $\mathcal{A}$ has to follow the following rules:

1. For each $j \in \{1, \ldots, h\}$, there is a constant m_j s.t. a $1/m_j$-fraction of the items of side length in $I_{2,j}$ is packed $2^d - 1$ per bin ("red items"), the rest are packed 2^d per bin ("blue items").
2. No bin contains an item of side length in $I_{1,i}$ and an item of side length in $I_{2,j}$ if $i + j \leq h$.
3. No bin contains an item of side length in $\overline{I}_1$ **and** an item of side length in $I_{2,j}$.
4. No bin contains an item of side length in $I_{1,j}$ **and** an item of side length in $\overline{I}_2$.
5. No bin that contains an item of side length in I_λ contains an item of side length in $I_{1,j}, I_{2,j}, \overline{I}_1$ or $\overline{I}_2$.

We will now define $2h+1$ input instances for the hypercube packing problem in d dimensions, and for each instance we derive a lower bound on the number of bins any $C^{(h)}$-algorithm must use to pack this input.

Every such input instance consists of three types of items. The input will contain N items of side length u, followed by $(2^d - 1)N$ items of side length v and finally followed by MN items of side length t, where u, v, t and M will be defined for each instance differently. We will then show, for every instance, that one u-item, $2^d - 1$ v-items and M t-items can be packed together in one bin, thus the optimal packing for this input uses at most N bins.

3.1 Instances $1, \ldots, h$

Let $\epsilon > 0$ be arbitrarily small. For every $j \in \{1, \ldots, h\}$, we define the following instance of the problem: Let $u = \frac{1+\epsilon}{2}, v = (1+\epsilon)y_{h-j}$ and $t = \frac{(1+\epsilon)y_{h-j}}{2K}$ for some large integer K such that $t \in I_\lambda$ and $\frac{K}{y_{h-j}} \in \mathbb{N}$. Clearly, $u \in I_{1,h}$ and $v \in I_{2,j}$.

In order to show that one u-item, 2^d-1 v-items and M t-items can be packed in one bin, we will define anchor points for each size and then place items at some of these such that no two items are overlapping.

There is only one anchor point for u-items, namely $(0,\dots,0)$, i.e. the origin of the bin. We place one u item there. For items of side length v, we define anchor points as all points having all coordinates equal to $(1+\epsilon)/2$ or $(1+\epsilon)/2-(1+\epsilon)y_{h-j}$. This defines 2^d anchor points, but an anchor point can only be used for a v-item if at least one coordinate is $(1+\epsilon)/2$. Hence, we can pack 2^d-1 v-items together with the u-item placed before.

For items of side length t, the anchor points are all points with coordinates equal to $i\frac{(1+\epsilon)y_{h-j}}{2K}$ for $i=0,\dots,\frac{2K}{y_{h-j}}-2$, i.e. we have $(\frac{2K}{y_{h-j}}-1)^d$ anchor points for these items. These anchor points form a superset of all previous anchor points for u- and v-items. Together with the fact that t divides u and v, we can conclude that all larger items take away an integer amount of anchor points for the t-items. To be precise, the u-item blocks $(u/t)^d=(K/y_{h-j})^d$ anchor points for t-items and each v-item blocks $\left(\frac{v}{t}\right)^d=(2K)^d$ anchor points for t-items. Hence, we can add $M:=\left(\frac{2K-y_{h-j}}{y_{h-j}}\right)^d-\left(\frac{K}{y_{h-j}}\right)^d-(2^d-1)(2K)^d$ t-items to the items packed before.

A Harmonic-type algorithm $\mathcal{A}$ packs a $1/m_j$-fraction of the $N(2^d-1)$ v-items 2^d-1 per bin, using $\frac{(2^d-1)N/m_j}{2^d-1}=\frac{N}{m_j}$ bins in total. The remaining $N(2^d-1)(1-1/m_j)$ v-items are packed 2^d per bin, adding another $N(1-1/m_j)\frac{2^d-1}{2^d}=N(1-1/m_j)\left(1-\frac{1}{2^d}\right)$ bins.

N/m_j of the u-items are added to bins with red v-items, the remaining $N(1-1/m_j)$ items of side length u must be packed one per bin.

Finally, an algorithm in the class $C^{(h)}$ needs at least $NM/\left(\frac{2K-y_{h-j}}{y_{h-j}}\right)^d$ bins to pack the t-items, giving

$$N\left(1-\left(\frac{K}{2K-y_{h-j}}\right)^d-(2^d-1)\left(\frac{2Ky_{h-j}}{2K-y_{h-j}}\right)^d\right)$$

bins for these items. If we let $K\to\infty$, this tends to $N\left(1-1/2^d-(2^d-1)y_{h-j}^d\right)$.

So, the total number of bins needed is at least

$$\begin{aligned}&N\left(\frac{1}{m_j}+\left(1-\frac{1}{m_j}\right)\left(1-\frac{1}{2^d}\right)+1-\frac{1}{m_j}+1-\frac{1}{2^d}-(2^d-1)y_{h-j}^d\right)\\&=N\left(2+\left(1-\frac{1}{m_j}\right)\left(1-\frac{1}{2^d}\right)-\frac{1}{2^d}-(2^d-1)y_{h-j}^d\right)\end{aligned}$$

As the optimal solution uses at most N bins, the performance ratio of any such algorithm $\mathcal{A}$ must be at least

$$R_{\mathcal{A}}\geq 2+(1-1/m_j)(1-1/2^d)-1/2^d-(2^d-1)y_{h-j}^d \qquad j=1,\dots,h \quad (1)$$

3.2 Instances $h+1,\dots,2h$

Another set of instances is given for any $j \in \{1,\dots,h\}$, if we use $u = (1+\epsilon)(1-y_{h-j+1}), v = (1+\epsilon)y_{h-j}$ and $t = \frac{(1+\epsilon)y_{h-j}(1-y_{h-j+1})}{K}$ for some large enough integer K such that $u \in I_{1,h-j}, v \in I_{2,j}, t \in I_\lambda$ and $\frac{K}{y_{h-j}}, \frac{K}{1-y_{h-j+1}} \in \mathbb{N}$. For these item sizes, the algorithm is not allowed to combine u-items with v-items in the same bin, although space for items in $I_{1,i}$ with $i > h-j$ is reserved in red bins containing v-items. We define the following anchor points: the point $(0,0)$ for type u; all points with all coordinates equal to $(1+\epsilon)(1-y_{h-j+1})$ or $(1+\epsilon)(1-y_{h-j+1})-(1+\epsilon)y_{h-j}$ for type v; and all points with all coordinates equal to $i\frac{(1+\epsilon)y_{h-j}(1-y_{h-j+1})}{K}$ for some $i \in \{0,\dots,\frac{K}{y_{h-j}(1-y_{h-j+1})}-2\}$ for type t. Again the anchor points for u- and v-items are a subset of the anchor points for t-items, and hence with the same argumentation as before we can pack one u-item together with 2^d-1 v-items and M t-items if we choose $M = \left(\frac{K-y_{h-j}(1-y_{h-j+1})}{y_{h-j}(1-y_{h-j+1})}\right)^d - \left(\frac{K}{y_{h-j}}\right)^d - (2^d-1)\left(\frac{K}{1-y_{h-j+1}}\right)^d$, as the u-item takes up $\left(\frac{K}{y_{h-j}}\right)^d$ anchor points of the t-items and each v-item takes up $\left(\frac{K}{1-y_{h-j+1}}\right)^d$ of these anchor points.

A similar calculation to before can be done: An algorithm in class $C^{(h)}$ needs $N/m_j + N(1-1/m_j)(1-1/2^d)$ bins for red and blue items of type v. It needs N bins for u-items, as they are packed one per bin, and finally

$$\begin{aligned}
&\frac{NM}{\left(\frac{K-y_{h-j}(1-y_{h-j+1})}{y_{h-j}(1-y_{h-j+1})}\right)^d} \\
&= N\left(1 - \left(\frac{K(1-y_{h-j+1})}{K-y_{h-j}(1-y_{h-j+1})}\right)^d - (2^d-1)\left(\frac{Ky_{h-j}}{K-y_{h-j}(1-y_{h-j+1})}\right)^d\right) \\
&\xrightarrow{K\to\infty} N\left(1-(1-y_{h-j+1})^d - (2^d-1)y_{h-j}^d\right)
\end{aligned}$$

bins are required to pack the t-items. Hence, we need at least

$$\begin{aligned}
&N\left(\frac{1}{m_j} + \left(1-\frac{1}{m_j}\right)\left(1-\frac{1}{2^d}\right) + 1 + 1 - (1-y_{h-j+1})^d - (2^d-1)y_{h-j}^d\right) \\
&= N\left(2 + \frac{1}{m_j} + \left(1-\frac{1}{m_j}\right)\left(1-\frac{1}{2^d}\right) - (1-y_{h-j+1})^d - (2^d-1)y_{h-j}^d\right)
\end{aligned}$$

bins in total. This gives the following lower bound for the performance ratio:

$$R_{\mathcal{A}} \geq 2 + 1/m_j + (1-1/m_j)\left(1-1/2^d\right) - (1-y_{h-j+1})^d - (2^d-1)y_{h-j}^d \quad (2)$$
$$j = 1,\dots,h$$

3.3 Instance $2h+1$

Let $u = \frac{1+\epsilon}{2}, v = (1+\epsilon)y_h$ and $t = \frac{(1+\epsilon)y_h}{2K}$ for some large enough integer K such that $u \in I_{1,h}, v \in \overline{I}_2, t \in I_\lambda$ and $\frac{K}{y_h} \in \mathbb{N}$. For these item sizes, the algorithm is

not allowed to combine u-items with v-items in the same bin. We define anchor points as follows: $(0,0)$ for type u; all points with coordinates equal to $\frac{1+\epsilon}{2}$ or $\frac{1+\epsilon}{2} - (1+\epsilon)y_h$ for type v; all points with coordinates equal to $i\frac{(1+\epsilon)y_h}{2K}$ for type t. As before, the anchor points for u and v-items are a subset of the t-items' anchor points, and so we can pack one u-item together with $2^d - 1$ v-items and M t-items if we choose $M = \left(\frac{2K-y_h}{y_h}\right)^d - \left(\frac{K}{y_h}\right)^d - (2^d-1)\,(2K)^d$.

For this input, any Harmonic-type algorithm uses at least N bins for u-items, $N\frac{2^d-1}{2^d} = N(1-\frac{1}{2^d})$ bins for v-items and $\frac{NM}{\left(\frac{2K-y_h}{y_h}\right)^d}$ bins for t-items. This gives in total

$$N\left(2 - \frac{1}{2^d} + 1 - \left(\frac{K}{2K-y_h}\right)^d - (2^d-1)\left(\frac{2Ky_h}{2K-y_h}\right)^d\right)$$
$$\xrightarrow{K\to\infty} N\left(3 - \frac{1}{2^{d-1}} - (2^d-1)y_h^d\right)$$

bins. We therefore can derive the following lower bound on the performance ratio:

$$R_{\mathcal{A}} \geq 3 - 1/2^{d-1} - (2^d-1)y_h^d \tag{3}$$

3.4 Combined Lower Bound

Given a certain set of parameters (y_j and m_j), the maximum of the three right sides of inequalities (1), (2) and (3) give us a bound on the competitive ratio of any Harmonic-type algorithm with this set of parameters. In order to get a general (worst-case) lower bound on $R_{\mathcal{A}}$, we need to find the minimum of this maximum over all possible sets of parameters.

This lower bound for $R_{\mathcal{A}}$ is obtained when equality holds in all of the inequalities (1), (2) and (3). To see this, consider the following: We have $2h+1$ variables and $2h+1$ constraints. For $j \in \{1,\ldots,h\}$, we see that (1) is increasing in m_j and (2) is decreasing in m_j. Next, let $c \in \{1,\ldots,h-1\}$. We see that (1) for $j = h-c \in \{1,\ldots,h-1\}$ is decreasing in y_c, and (2) for $j = h-c+1 \in \{2,\ldots,h\}$ is increasing in y_c. Finally, we have that (2) for $j=1$ is increasing in y_h and (3) is decreasing in y_h. This means, given certain parameters y_j and m_j, if e.g. (3) gives a smaller lower bound on $R_{\mathcal{A}}$ than (2) with $j=1$ does, we can decrease the value of y_h such that the maximum of the three lower bounds becomes smaller.

Setting the right hand side of (1) equal to the right hand side of (2), gives us $\frac{1}{m_j} = (1-y_{h-j+1})^d - \frac{1}{2^d}$ or alternatively $\frac{1}{m_{h-j+1}} = (1-y_j)^d - \frac{1}{2^d}$. Plugging this into (1) (replacing j by $h-j+1$), we find that

$$y_j = 1 - \left(\frac{-2^dR_{\mathcal{A}} + 2^dy_{j-1}^d - 4^dy_{j-1}^d - 1 + 3\cdot 2^d - 1/2^d}{2^d-1}\right)^{1/d} \tag{4}$$

Recall that we require $1/3 = y_0 < y_1$. From this, combined with (4) for $j = 1$, we obtain that

$$R_{\mathcal{A}} \geq 3 - 2\frac{2^d - 1}{3^d} - \frac{2^d + 1}{4^d}$$

We list some values of the lower bound for several values of d in Table 2.

$d =$	1	2	3	4	5	6	∞
$R_{\mathcal{A}} >$	1.58333	2.02083	2.34085	2.56322	2.71262	2.81129	3

Table 2: Lower bounds for Harmonic-type algorithms in dimensions 1 to 6 and limit for $d \to \infty$.

Note that for $d = 1$, our formula yields the bound of Ramanan et al. [15]. Surprisingly, it does not seem to help to analyze the values of $y_2, \ldots, y_h$. Especially, equations involving y_j for $j > 1$ become quite messy due to the recursive nature of (4). If h is a very small constant like 1 or 2, we can derive better lower bounds for $R_{\mathcal{A}}$. For larger h, we can use the inequalities $y_1 < y_h, y_2 < y_h, y_3 < y_h$ (i.e. assuming that $h > 3$) to derive *upper* bounds on the best value $R_{\mathcal{A}}$ that could possibly be proven using this technique. These upper bounds are very close to 2.02 and suggest that for larger h, an algorithm in the class $C^{(h)}$ could come very close to achieving a ratio of 2.02 for these inputs. However, since the inequalities become very unwieldy, we do not prove this formally.

Theorem 2. *No Harmonic-type algorithm for two-dimensional online hypercube packing can achieve an asymptotic performance ratio better than* 2.0208.

4 Further Lower Bounds

Inspired by [9], one could try to improve online algorithms for packing 2-dimensional squares by incorporating two ideas from the one-dimensional case: combining large items (i.e. items larger than 1/2) and medium items (i.e. items with size in $(1/3, 1/2]$) whenever they fit together (ignoring their type), and postponing the coloring decision. The former is intuitive, while the idea of the latter would be the following: When items of a certain type arrive, we first give them provisional colors and pack them into separate bins (i.e. one item per bin). After several items of this type arrived, we choose the smallest of them to be red and all others are colored blue. With following items of this type, we fill up the bins with additional items. However, simply adding two more red items to the bin with a single red item might be problematic: When filling up the red bins with two more red items, it could happen that these later red items are larger than the first one - negating the advantage of having the first red item be relatively small. Alternatively, we could leave the red item alone in its bin. This

way, we make sure that at most 3/4 of the blue items of a certain medium type are smaller than the smallest red item of this type, but we have more wasted space in this bin.

For both approaches discussed above we will show lower bounds on the competitive ratio that are even higher than the lower bound established in Section 3 for Harmonic-type algorithms.

4.1 Always combining large and medium items

First, we consider algorithms that combine small and large items whenever they fit together. We define a class of algorithms B_1 that distinguish, possibly among others, the following disjoint subintervals (types):

- $I_m = (1/3, y]$ for some $y \in (1/3, 1/2]$
- $I_\lambda = (0, \lambda]$

These algorithms satisfy the following rules:

1. There is a parameter α s.t. an α-fraction of the items of side length in I_m are packed 3 per bin ("red items"), the rest are packed 4 per bin ("blue items").
2. No bin that contains an item of side length in I_λ contains an item of side length larger than 1/2 or an item of side length in I_m.
3. Items of type I_m are packed without regard to their size.

Let $a, b \in I_m, a < b$. We consider two different inputs, both starting with the same set of items: $\frac{\alpha}{3}N$ items of size b and $(1 - \alpha/3)N$ items of size a (i.e. in total N items of size a and b). By rule 3, the adversary knows beforehand which item will be packed in which bin, as they belong to the same type. Hence, the adversary can order these items in such a way that the items colored blue by the algorithm are all a-items, and in each bin with red items, there are two a- and one b-item. By rule 1, the online algorithm uses $(\frac{\alpha}{3} + \frac{1-\alpha}{4})N = \frac{3+\alpha}{12}N$ bins for items of this type.

The sizes a and b will tend towards 1/3, as this way the adversary can maximize the total volume of sand (infinitesimally small items) that can be added to any bin in the optimal solution while not changing the way the algorithm packs these items and increasing the number of bins the algorithm needs for packing the sand items. Therefore, we will assume that a and b are arbitrarily close to 1/3.

In the first input, after these medium items, $\frac{(1-\alpha/3)N}{3}$ items of size $1 - a$ arrive, followed by sand of total volume $\frac{24+7\alpha}{324}N$. In the optimal solution, we can pack $\frac{\alpha}{12}N$ bins with four b-items and sand of volume 5/9 each, and $\frac{(1-\alpha/3)N}{3}$ bins with three a-items, one $(1-a)$-item and sand of volume $\frac{\alpha}{12}N \cdot \frac{2}{9}$ each. Hence, the optimal solution uses $\frac{\alpha}{12}N + \frac{(1-\alpha/3)N}{3} = \frac{12-\alpha}{36}N$ bins.

The algorithm, however, cannot pack a large item into any of the bins with red medium items, as these always contain a b-item. Hence, in addition to the $\frac{3+\alpha}{12}N$ bins for medium items, the algorithm needs $\frac{(1-\alpha/3)N}{3}$ bins for large items

and at least $\frac{24+7\alpha}{324}N$ bins for sand. This gives in total at least $\frac{213-2\alpha}{324}N$ bins, and a competitive ratio of at least

$$\frac{\frac{213-2\alpha}{324}N}{\frac{12-\alpha}{36}N} = \frac{213-2\alpha}{9(12-\alpha)} \tag{5}$$

In the second input, after the medium items, $N/3$ items of size $1/2+\epsilon$ will arrive, followed by sand of total volume $\frac{5}{36}N$. The optimal solution packs all medium items three per bin, using $N/3$ bins, and adds one large item and sand of volume $15/36$ in each such bin. In the algorithm's solution, large items can only be added to the $\alpha N/3$ bins containing three red items, i.e. it needs additional $N/3-\alpha N/3$ bins for the remaining $N/3-\alpha N/3$ large items. Finally, the algorithm uses at least $5/36N$ bins for sand. The algorithm therefore uses in total at least $\frac{3+\alpha}{12}N + (1-\alpha)N/3 + 5/36N = \frac{26-9\alpha}{36}N$ bins. This gives a competitive ratio of at least

$$\frac{3(26-9\alpha)}{36} = \frac{26-9\alpha}{12} \tag{6}$$

Observe that (5) is increasing in α, while (6) is decreasing in α. Hence, the minimum over the maximum of the two bounds is obtained for the α-value that makes both bounds equal, which is $\alpha = \frac{197-\sqrt{36541}}{27} \approx 0.2164$. For this α, both bounds become larger than 2.0043.

Theorem 3. *No algorithm in class B_1 for two-dimensional online hypercube packing can achieve a competitive ratio of less than* 2.0043*.*

4.2 Packing red medium items one per bin, postponing the coloring

Now, consider the algorithm that packs red items alone into bins and makes sure that at most 3/4 of the blue items of a certain type are smaller than the smallest red item of this type. We define a new class of algorithms B_2 that distinguish, possibly among others, the following disjoint subintervals (types):

- $I_m = (1/3, y]$
- $I_\lambda = (0, \lambda]$

Furthermore, algorithms in B_2 satisfy the following rules:

1. There is a parameter α s.t. an α-fraction of the items of side length in I_m are packed 1 per bin ("red items"), the rest are packed 4 per bin ("blue items").
2. No bin that contains an item of side length in I_λ contains an item of side length larger than $1/2$ or an item of side length in I_m.
3. Items of side length in I_m are initially packed one per bin. At some regular intervals, the algorithm fixes some of these items to be red, and does not pack additional items of the same type witht them.

From rule 3 we can conclude that the algorithm gives the following guarantee: 3/4 of the blue items with size in I_m are not smaller than the smallest red item with size in I_m.

Let $a, b \in I_m, a < b$ as before. We again consider two different inputs, both starting with the same set of items: $\alpha N + \frac{1-\alpha}{4}N$ items of size b, and $\frac{3(1-\alpha)}{4}N$ items of size a. They arrive in such an order that all red items are b-items, and all bins with blue items contain one b- and three a-items. We require the b-item in the blue bins because of the postponement of the coloring: If the first blue item in a bin was an a-item, the algorithm would choose this item to become red and not one of the b-items. By rule 1, the algorithm needs $\frac{1-\alpha}{4}N + \alpha N = \frac{1+3\alpha}{4}N$ bins for these N items.

In the first input, after the medium items arrived, we get $\frac{1-\alpha}{4}N$ large items of size $1-a$, followed by sand of total volume $\frac{13+7\alpha}{144}N$. The optimal solution can pack the a-items three per bin together with one $(1-a)$-item, using $\frac{1-\alpha}{4}N$ bins for these items. The b-items are packed four per bin, using $(\frac{\alpha}{4}+\frac{1-\alpha}{16})N$ bins. Note that the empty volume in all bins of these two types is $\frac{1-\alpha}{4}N \cdot \frac{2}{9} + (\frac{\alpha}{4}+\frac{1-\alpha}{16})N \cdot \frac{5}{9} = \frac{13+7\alpha}{144}N$, i.e. it equals exactly the volume of the sand, so the sand can be filled in these holes without using further bins. Hence, the optimal number of bins is $\frac{1-\alpha}{4}N + (\frac{\alpha}{4} + \frac{1-\alpha}{16})N = \frac{5-\alpha}{16}N$.

The algorithm uses, as discussed before, $\frac{1+3\alpha}{4}N$ bins for the medium items of size a and b. The large items cannot be added to red medium items, as they do not fit together, thus the algorithm uses $\frac{1-\alpha}{4}N$ additional bins for the large items. Finally, according to rule 2, at least $\frac{13+7\alpha}{144}N$ additional bins are needed to pack the sand. This gives in total at least $\frac{1+3\alpha}{4}N + \frac{1-\alpha}{4}N + \frac{13+7\alpha}{144}N = \frac{85+79\alpha}{144}N$ bins. We find that the competitive ratio is at least

$$\frac{\frac{85+79\alpha}{144}N}{\frac{5-\alpha}{16}N} = \frac{85+79\alpha}{9(5-\alpha)} \tag{7}$$

In the second input, $N/3$ items of size $1/2+\epsilon$ arrive after the medium items, followed by sand of total volume $5/36N$. The algorithm packs this input the same way as a B_1 algorithm, so the analysis carries over. We get a competitive ratio of at least

$$\frac{\frac{26-9\alpha}{36}N}{N/3} = \frac{26-9\alpha}{12} \tag{8}$$

It can be seen that (7) is a function increasing in α, while (8) is decreasing in α, hence the minimum over the maximum of two bounds is reached when they are equal. In that case, $\alpha = \frac{529-\sqrt{274441}}{54} \approx 0.0950$, and the lower bound for the competitive ratio becomes larger than 2.0954.

Theorem 4. *No algorithm in class B_2 for two-dimensional online hypercube packing can achieve a competitive ratio of less than* 2.0954.

Note here that this is an even higher lower bound than the one shown in the previous Subsection 4.1, although we use postponement of the coloring here. This indicates that the space we waste by packing red medium items separately

outweighs the advantage we get by having a guarantee about the size of the red item.

References

1. János Balogh, József Békési, and Gábor Galambos. *Approximation and Online Algorithms: 8th International Workshop, WAOA 2010, Liverpool, UK, September 9-10, 2010. Revised Papers*, chapter New Lower Bounds for Certain Classes of Bin Packing Algorithms, pages 25–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
2. Don Coppersmith and Prabhakar Raghavan. Multidimensional on-line bin packing: Algorithms and worst-case analysis. *Oper. Res. Lett.*, 8(1):17 – 20, 1989.
3. János Csirik and André van Vliet. An on-line algorithm for multidimensional bin packing. *Oper. Res. Lett.*, 13(3):149 – 158, 1993.
4. Leah Epstein and Rob van Stee. Online square and cube packing. *Acta Inf.*, 41(9):595–606, 2005.
5. Leah Epstein and Rob van Stee. Optimal online algorithms for multidimensional packing problems. *SIAM J. Comput.*, 35(2):431–448, 2005.
6. Leah Epstein and Rob van Stee. Bounds for online bounded space hypercube packing. *Discrete Optim.*, 4(2):185–197, 2007.
7. Xin Han, Francis Y. L. Chin, Hing-Fung Ting, Guochuan Zhang, and Yong Zhang. A new upper bound 2.5545 on 2d online bin packing. *ACM Transactions on Algorithms*, 7(4):50, 2011.
8. Xin Han, Deshi Ye, and Yong Zhou. A note on online hypercube packing. *Cent. Europ. J. Oper. Re.*, 18(2):221–239, 2010.
9. Sandy Heydrich and Rob van Stee. Beating the harmonic lower bound for online bin packing. *CoRR*, abs/1511.00876, 2015. http://arxiv.org/abs/1511.00876.
10. David S. Johnson. Fast algorithms for bin packing. *J. Comput. Syst. Sci.*, 8(3):272–314, 1974.
11. David S. Johnson, Alan J. Demers, Jeffrey D. Ullman, Michael R. Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput.*, 3(4):299–325, 1974.
12. Chung-Chieh Lee and Der-Tsai Lee. A simple on-line bin-packing algorithm. *J. ACM*, 32(3):562–572, 1985.
13. Joseph Y.-T. Leung, Tommy W. Tam, C. S. Wong, Gilbert H. Young, and Francis Y. L. Chin. Packing squares into a square. *J. Parallel Distrib. Comput.*, 10(3):271–275, 1990.
14. Yiping Lu, Danny Z. Chen, and Jianzhong Cha. Packing cubes into a cube is np-complete in the strong sense. *J. Comb. Optim.*, 29(1):197–215, 2015.
15. Prakash V. Ramanan, Donna J. Brown, Chung-Chieh Lee, and Der-Tsai Lee. On-line bin packing in linear time. *J. Algorithms*, 10(3):305–326, 1989.
16. Steven S. Seiden. On the online bin packing problem. *J. ACM*, 49(5):640–671, 2002.
17. Jeffrey D. Ullman. The performance of a memory allocation algorithm. Technical Report 100, Princeton University, Princeton, NJ, 1971.
18. André van Vliet. *Lower and upper bounds for online bin packing and scheduling heuristics*. PhD thesis, Erasmus University, Rotterdam, The Netherlands, 1995.