Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Bounds for online bounded space hypercube packing

L. Epstein, R. van Stee

REPORT SEN-E0417 SEPTEMBER 2004

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Bounds for online bounded space hypercube packing

ABSTRACT
In hypercube packing, we receive a sequence of hypercubes that need to be packed into unit hypercubes which are called bins. Items arrive online and each item must be placed within its bin without overlapping with other items in that bin. The goal is to minimize the total number of bins used. We present lower and upper bounds for online bounded space hypercube packing in dimensions 2,...,7.

2000 Mathematics Subject Classification: 68W25,68W40
1998 ACM Computing Classification System: F.2.2
Keywords and Phrases: approximation algorithms;bin packing;hypercube packing

Bounds for online bounded space hypercube packing

Leah Epstein* Rob van Stee†

September 28, 2004

Abstract

In hypercube packing, we receive a sequence of hypercubes that need to be packed into unit hypercubes which are called bins. Items arrive online and each item must be placed within its bin without overlapping with other items in that bin. The goal is to minimize the total number of bins used. We present lower and upper bounds for online bounded space hypercube packing in dimensions $2, \ldots, 7$.

1 Introduction

In the general multidimensional bin packing problem, we receive a sequence σ of *items* $h_1, h_2, \ldots, h_n$. Each item h has a fixed *size*, which is $s_1(h) \times \cdots \times s_d(h)$. I.e., $s_i(h)$ is the size of h in the ith dimension. We have an infinite number of *bins*, each of which is a d-dimensional unit hypercube. Each item must be assigned to a bin and a position $(x_1(h), \ldots, x_d(h))$, where $0 \leq x_i(h)$ and $x_i(h) + s_i(h) \leq 1$ for $1 \leq i \leq d$. Further, the positions must be assigned in such a way that no two items in the same bin overlap. A bin is *empty* if no item is assigned to it, otherwise it is *used*. The goal is to minimize the number of bins used. Note that for $d = 1$, the box packing problem reduces to exactly the classic bin packing problem.

In this paper, we consider this problem under the following restrictions:

- *items arrive online:* each item must be assigned in turn, without knowledge of the next items.
- *all items are hypercubes:* in the hypercube packing problem we have the restriction that all items are hypercubes, i.e. an item has the same size in every dimension.
- *bounded space:* an algorithm has only a constant number of bins available to accept items at any point during processing. The bounded space assumption is a quite natural one, especially so in online box packing. Essentially the bounded space restriction guarantees that output of packed bins is steady, and that the packer does not accumulate an enormous backlog of bins which are only output at the end of processing.

*School of Computer Science, The Interdisciplinary Center, Herzliya, Israel. lea@idc.ac.il. Research supported by Israel Science Foundation (grant no. 250/01).

†Centre for Mathematics and Computer Science (CWI), Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands. Rob.van.Stee@cwi.nl. Work supported by the Netherlands Organization for Scientific Research (NWO), project number SION 612-061-000.

The offline versions of these problems are NP-hard, while even with unlimited computational ability it is impossible in general to produce the best possible solution online. We consider bounded space online approximation algorithms.

The standard measure of algorithm quality for box packing is the *asymptotic performance ratio*, which we now define. For a given input sequence σ, let $\text{cost}_{\mathcal{A}}(\sigma)$ be the number of bins used by algorithm $\mathcal{A}$ on σ. Let $\text{cost}(\sigma)$ be the minimum possible number of bins used to pack items in σ. The *asymptotic performance ratio* for an algorithm $\mathcal{A}$ is defined to be

$$\mathcal{R}_{\mathcal{A}}^{\infty} = \limsup_{n\to\infty} \sup_{\sigma} \left\{ \frac{\text{cost}_{\mathcal{A}}(\sigma)}{\text{cost}(\sigma)} \middle| \text{cost}(\sigma) = n \right\}.$$

Let $\mathcal{O}$ be some class of box packing algorithms (for instance online algorithms or bounded space online algorithms). The *optimal asymptotic performance* ratio for $\mathcal{O}$ is defined to be $\mathcal{R}_{\mathcal{O}}^{\infty} = \inf_{\mathcal{A}\in\mathcal{O}} \mathcal{R}_{\mathcal{A}}^{\infty}$. Given $\mathcal{O}$, our goal is to find an algorithm with asymptotic performance ratio close to $\mathcal{R}_{\mathcal{O}}^{\infty}$.

We recently presented an optimal online bounded space algorithm for (general) multi-dimensional bin packing [1]. The asymptotic performance ratio of our algorithm is $(\Pi_\infty)^d$, where Π_∞ is the asymptotic performance ratio of the one-dimensional HARMONIC algorithm. We also presented an optimal online bounded space algorithm for (hyper-)cube packing in [1]. For this last algorithm, we know that the asymptotic performance ratio is $\Omega(\log d)$ and $O(\log d)$ but we have no explicit formulas for it. The current contribution explores the asymptotic performance ratio of our algorithm for low dimensions. In [1], we state some preliminary results for these values. In this paper, we describe the methods which we used to develop them, and some further methods, and give improved upper and lower bounds.

We summarize the results in the following table.

d	Lower bound	Upper bound
1	1.69103	1.69103
2	2.36288	2.3692
3	2.95642	3.0672
4	3.38984	3.7595
5	3.78744	4.4052
6	4.10246	5.0126
7	4.34946	5.7084

Table 1: Overview of results ($d = 1$ is included for completeness, and is given by [2])

In Section 2, we explain how we calculate upper bounds for our (optimal) algorithm. In Section 3, we give lower bounds for square packing. In Section 4, we give lower bounds for cube packing. Finally, in Section 5, we give lower bounds for general d, which improve on the best known lower bounds for small dimensions but do not improve the general result.

2 Upper bounds

We define weights for items based on the amounts of items that are placed together in one bin. Consider items that have size in the interval $(1/(i+1), 1/i]$ for $i = 1, \ldots, M$. The d-dimensional

version of our algorithm places i^d of them in one bin. We therefore define the weight of one such item to be $1/i^d$.

Finally, the smallest items are combined together in bins (in a way that only uses a constant number of open bins). It is proved in [1] that in bins used for this type, a volume of at least $(M^d-1)/(M+1)^d$ is occupied. We therefore define the weight of an item of this type as its size times $(M+1)^d/(M^d-1)$.

It is straightforward to see that with these definitions, the number of bins that our algorithm requires to pack a given input sequence is upper bounded by the total weight of the items in that input sequence. Since the optimal algorithm needs to pack the same amount (possibly combining items of very different sizes in one bin), it can be seen that the asymptotic performance ratio of our algorithm is upper bounded by the amount of weight that can be packed into a single bin (of any algorithm).

It is NP-hard to determine whether a given set of items fits into a single bin or not [3]. However, it is possible to give upper bounds for the amount of weight that can be packed into a single bin. This is what we have done to give upper bounds for the asymptotic performance ratio of our algorithm in several low dimensions.

To be precise, we repeatedly use the following claim from [1].

Claim 1 *Given a packing of hyperboxes into a bin, such that each component j is bounded in an interval $(1/(k_j+1), 1/k_j]$, where $k_j \geq 1$ is an integer, then this bin contains at most $\prod_{j=1}^{d} k_j$ hyperboxes.*

For instance, according to this claim at most 25 squares larger than 1/6 fit in a bin. A square larger than 1/2 can be partitioned into 9 squares larger than 1/6; a square larger than 1/3 can be partitioned into 4 squares larger than 1/6. Therefore, if we consider a packing with one item larger than 1/2 and three items larger than 1/3, then at most $25-9-3\cdot 4=4$ items larger than 1/6 fit with them inside one bin.

We can formulate similar constraints by expressing items as multiples of items of size $1/k$ for $k=1,\ldots,50$. This generates a linear program that can be solved by a computer. In this way we generated the upper bound for $d=3,\ldots,7$.

2.1 Square packing

For square packing, we have additionally created a computer program which can check for particular sets of items whether they can be packed inside a bin by trying all possible packings. This can be used in case the LP constraints do not give sufficiently good bounds. We denote this program by F. We begin by describing how this computer program works.

The computer program F In any feasible packing, items can be shifted down and to the left until every item touches another item or the edge of the bin.

We add coordinates to the bin in the obvious way, the origin is the lower left corner of the bin. For any given input, we are going to try and construct a feasible packing for this input by placing the items one by one in all possible orderings.

We maintain an array of "available positions" for items. At the start, this array contains only the point $(0,0)$ (lower left corner). Each time that an item is placed in an available position, this position is removed and two new available positions are formed: namely, at the top of the current

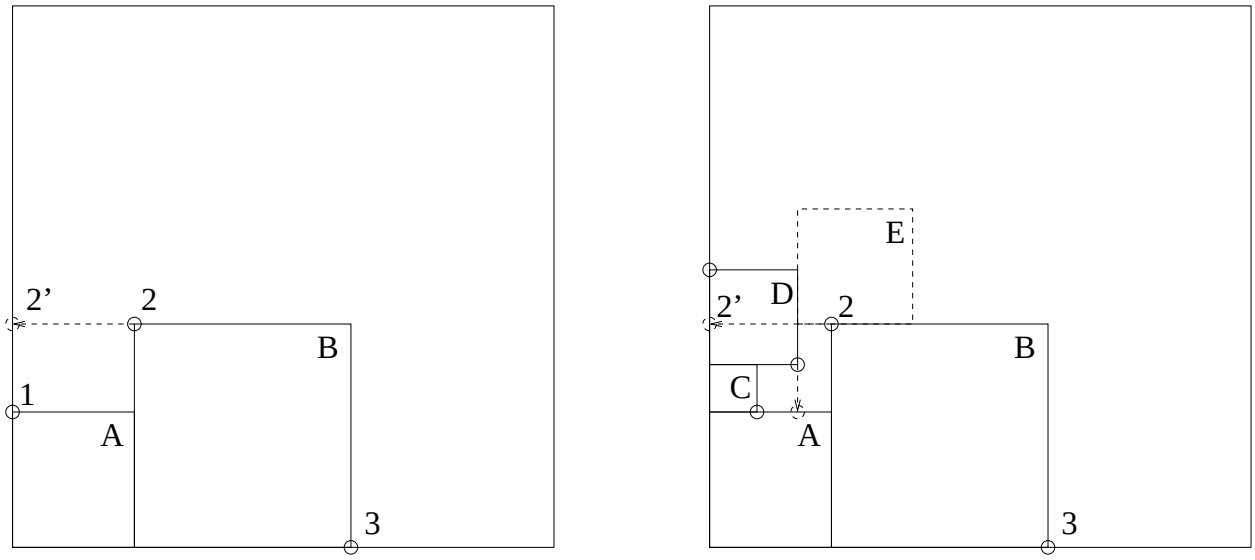

Figure 1: Illustration of available positions in partial packings

item and to the right of it. These positions are marked **top** and **right**. (Example: item of size 1/3 is placed at (x, y). New available positions are **right**: $(x + 1/3, y)$ and **top**: $(x, y + 1/3)$.)

In general, if an item cannot be placed at a certain position because there are other items to the right or on top of it that were placed earlier,we move to the next position.

It is possible that some of these positions are "incorrect" in the sense that no item should be placed there. For instance, consider the packing in Figure 1, left. In this figure, the available positions are marked with circles and numbered. We are going to try these positions one by one with the next item in S. Suppose that placing item C in position 1 does not lead to a feasible packing. Then we would try item C in position 2. However, this item can be shifted towards the left, leaving more room for items at the right of the bin.

Therefore we would not put this item in position 2 but in fact in the shifted position indicated in the figure. It seems that this might block future items from being put in position 1, so that we are missing some packings and not trying all possibilities. However, it should be noted that if we do not find a feasible packing with item C in position 2' or in position 3, we will then try to place item D in position 1 and continue like this. Eventually, if we do not find any feasible packing, we will try all possible items in position 1 before going to position 2'. Thus it can be seen that we do check all the possibilities.

An important thing to note is that we cannot immediately modify position 2 (shift it left) when we create it. To see this, consider the packing in Figure 1, right. After placing items A through D, we would not consider placing item E as drawn, because this position is no longer available to us. Therefore we keep the unmodified positions in the array and shift items that are placed there, if possible.

We use some further heuristics to prune the positions:

- If a position coincides with an already available position, one of the pair in one location is marked unavailable.
- If a **top** position is too close to the top of the bin (the distance is smaller than the size of the

smallest item in the pattern), it is marked unavailable. Note that any item in such a position is not allowed to be shifted down, even if this were possible after first shifting it to the left: if we did allow this, the final position would always coincide with some other available position.

- If a **right** position is too close to the right of the bin, it is marked unavailable for the same reason.
- If a position has an item to its immediate left as well as immediately below it, then this is the actual position that will be used later to place an item and there will be no shifting. So, if even the smallest item does not fit here because of overlap with other items, mark the position unavailable, but remember which item causes the unavailability in case that item is removed later in the search process.
- If a position A does not have an item directy left or below it, it is still unavailable if there exists an item X that *covers* A (in this case, this position has been superseded by the two positions created when X was placed). By covering we mean in this case for a **top** position that there is an item placed at the same height and covering the x-value, and for a **right** position that there is an item at the same x-value and covering the y-value. In such cases, remember which item blocks the position.

The second and third heuristic explain why we need to distinguish between **top** and **right** positions.

We are now ready to describe the overall algorithm used in the program, which works recursively. It is called with a given possible pattern $(p_1, p_2, \ldots, p_k)$, where p_i denotes the number of items of size s_i for $i = 1, \ldots, k$, and the positions of previously placed items, plus the array of available positions. The main loop of the algorithm is over the item sizes $s_1, \ldots, s_k$. We try every size s_i for which $p_i > 0$. For a given available position A and item of size s_i, we construct a "real" position A': if A is **top**, we shift it left as far as possible; if A is **right**, we shift it down. How far it can be shifted possibly depends on s_i, so A' needs to be recalculated for every item.

We then try all available positions for the current item size. That is, we do the following:

- calculate A' depending on A and the size s_i
- put an item of size s_i in position A'
- if A' is **top** and the item can be shifted down, skip this position (we could shift the item down, but then it would coincide with another available position);
- similarly for a **right** position
- if the item is not completely inside the bin, skip this position;
- if the item overlaps with another item, skip this position;
- else, remove A from the array of available positions, and create new available positions as described above
- decrease p_i by 1
- try (recursively) to pack the remaining set

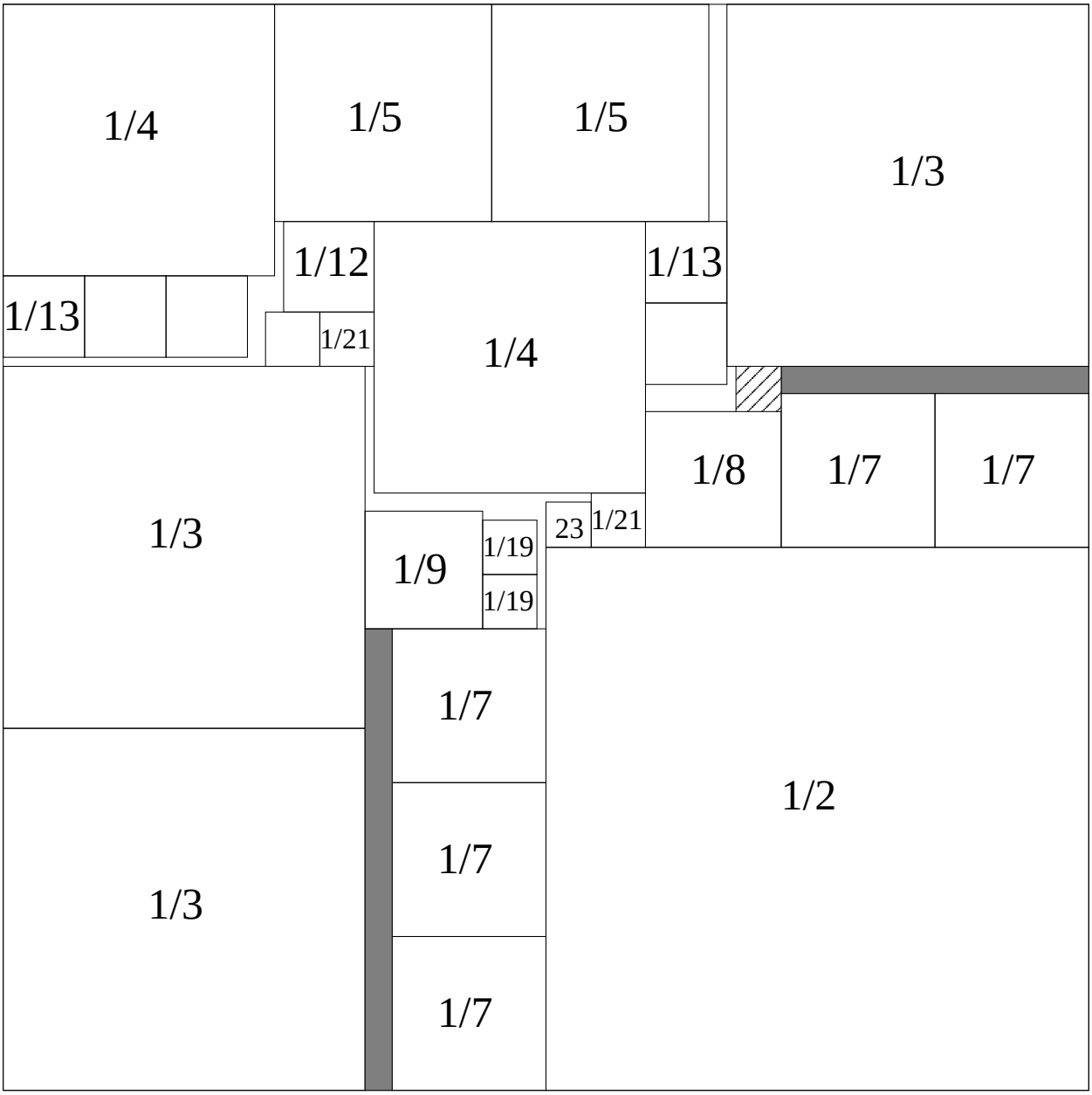

Figure 2: The lower bound for square packing. The arced item has size $1/25 + \varepsilon$. The grey areas contain items of size $1/43 + \varepsilon$ (18 bottom left, 12 top right, 30 in total).

- on failure, add 1 to p_i; remove the item of size s_i from position A' and restore the set of available positions to the previous set

As soon as we find a feasible packing, we break off the search and return true. If this never happens, we eventually return false.

This function is called originally with the set S that we are interested in, and the available position (0,0).

The upper bound Our reasoning is as follows, where in parentheses we mention whether the conclusion follows from a linear program (LP) or from this computer program (F).

We are looking for a bin with maximum weight. If there is no item larger than 1/2, the maximum weight that can be packed is at most 1.8862 (LP). Else, if there are at most two items larger than 1/3, the bound is 2.3235 (LP). Else, with at most one item larger than 1/4, the bound is 2.3519 (LP). Else, with at most one item larger than 1/5, the bound is 2.3632 (LP).

To get a bound which is higher than 2.3632, we need to pack one item larger than 1/2, three items larger than 1/3, two items larger than 1/4 and two items larger than 1/5. At most five items larger than 1/7 can be packed with them (F). (No items larger than 1/6 can be packed with them—see above.)

If there are only at most four items larger than 1/7, the upper bound becomes 2.3692 (LP). Else, there can be at most one item of size 1/8 (F), and the upper bound becomes 2.3678. We conclude that the asymptotic performance ratio of our algorithm is not worse than 2.3692.

3 Square packing

Consider the packing shown in Figure 2.

If an item is marked with $1/i$ for some integer i, it means that the size of the associated item (length of a side) is $(1+\varepsilon)/i$ for some very small $\varepsilon > 0$. It is straightforward to verify that the packing shown in Figure 2 is in fact a valid packing for these items, by checking that the total horizontal size of items along each horizontal line through the packing is strictly less than 1. This implies that all the items can be scaled up by a small amount and the items will still fit in the bin.

When many items of one type arrive, a bounded space algorithm will have to pack almost all of these items into bins of their own. Thus it needs $N/(i-1)^2 - O(1)$ bins to pack N items of size $(1+\varepsilon)/i$. We say that the weight of an item of size $(1+\varepsilon)/i$ is $1/(i-1)^2$. Then the total amount of bins needed by a bounded space algorithm is at least the total weight of the items minus a constant. We give an overview of the items used and the number of times that they occur in Figure 2 in the table below. In the first line containing the item sizes, we ignore the constant ε.

Size	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{7}$	$\frac{1}{8}$	$\frac{1}{9}$	$\frac{1}{12}$	$\frac{1}{13}$	$\frac{1}{19}$	$\frac{1}{21}$	$\frac{1}{23}$	$\frac{1}{25}$	$\frac{1}{43}$
Amount	1	3	2	2	5	1	1	1	5	2	3	1	1	30
Weight	1	$\frac{1}{4}$	$\frac{1}{9}$	$\frac{1}{16}$	$\frac{1}{36}$	$\frac{1}{49}$	$\frac{1}{64}$	$\frac{1}{121}$	$\frac{1}{144}$	$\frac{1}{324}$	$\frac{1}{400}$	$\frac{1}{484}$	$\frac{1}{576}$	$\frac{1}{1764}$

Each copy of each item that is in this packing is going to appear N times in the input for the bounded space algorithms. Then the offline cost to pack these items is exactly N. The total area of the items in one bin is less than 0.986934. We can therefore add sand (squares of infinitesemal size) of total volume $0.013066 \cdot N$. The total weight is then more than $2.36288 \cdot N$ which implies the lower bound.

4 Cube packing

On the next pages you will find a list of figures. These figures represent intersections of the unit cube at particular heights, starting at height 0. In these figures, for clarity it is assumed that all item sizes are exact inverses of integers. The reader should verify that in every figure, every horizontal and vertical line through the plane intersects some area without any items. This shows that if we replace the items by the real items (of size $1/2 + \varepsilon$ etc.), the items still fit in the bin.

Moreover, the reader should verify that when taking all the empty areas in all the figures together, they cover the unit square. Thus also in the vertical dimension, the packing still works with the real item sizes.

We give a table of the heights at which new items start, and specify these items. An item "starts" if its lower boundary is at the current height. See Table 2.

	Item size							
Height	1/2	1/3	1/4	1/5	1/7	1/8	1/9	1/13
0		4	5	2		2	1	12
1/13								12
1/9							1	
1/8						2		
2/13								12
2/9							1	
3/13								12
1/4			2	5				
4/13								8
1/3		3			11			
5/13								8
9/20				2				
10/21					5			
1/2	1					4		
11/20				2				
13/21					5			
5/8							6	
13/20					2			
2/3						4		32
3/4			5					
16/21					2			
4/5				7				
Totals	1	7	12	18	25	12	9	96
Weight	1	1/8	1/27	1/64	1/216	1/343	1/512	1/1728

Table 2: Overview of item placements

The total volume of these items is less than 0.868125. Therefore we can add sand of total volume 0.131875. This implies a lower bound of 2.95642.

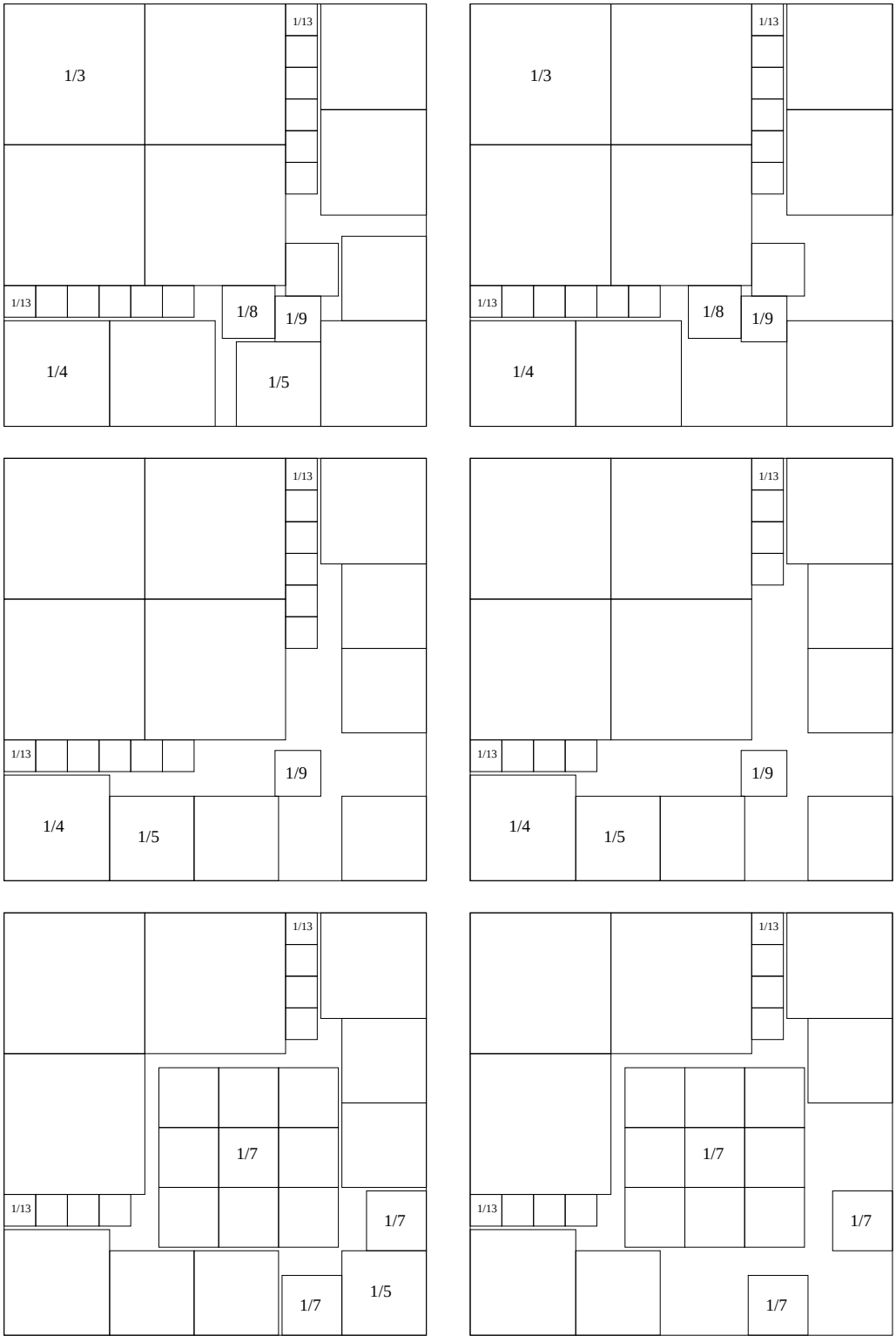

Figure 3: Intersections at heights 0, 1/5, 1/4, 4/13, 1/3 and $1/4 + 1/5 = 0.45$

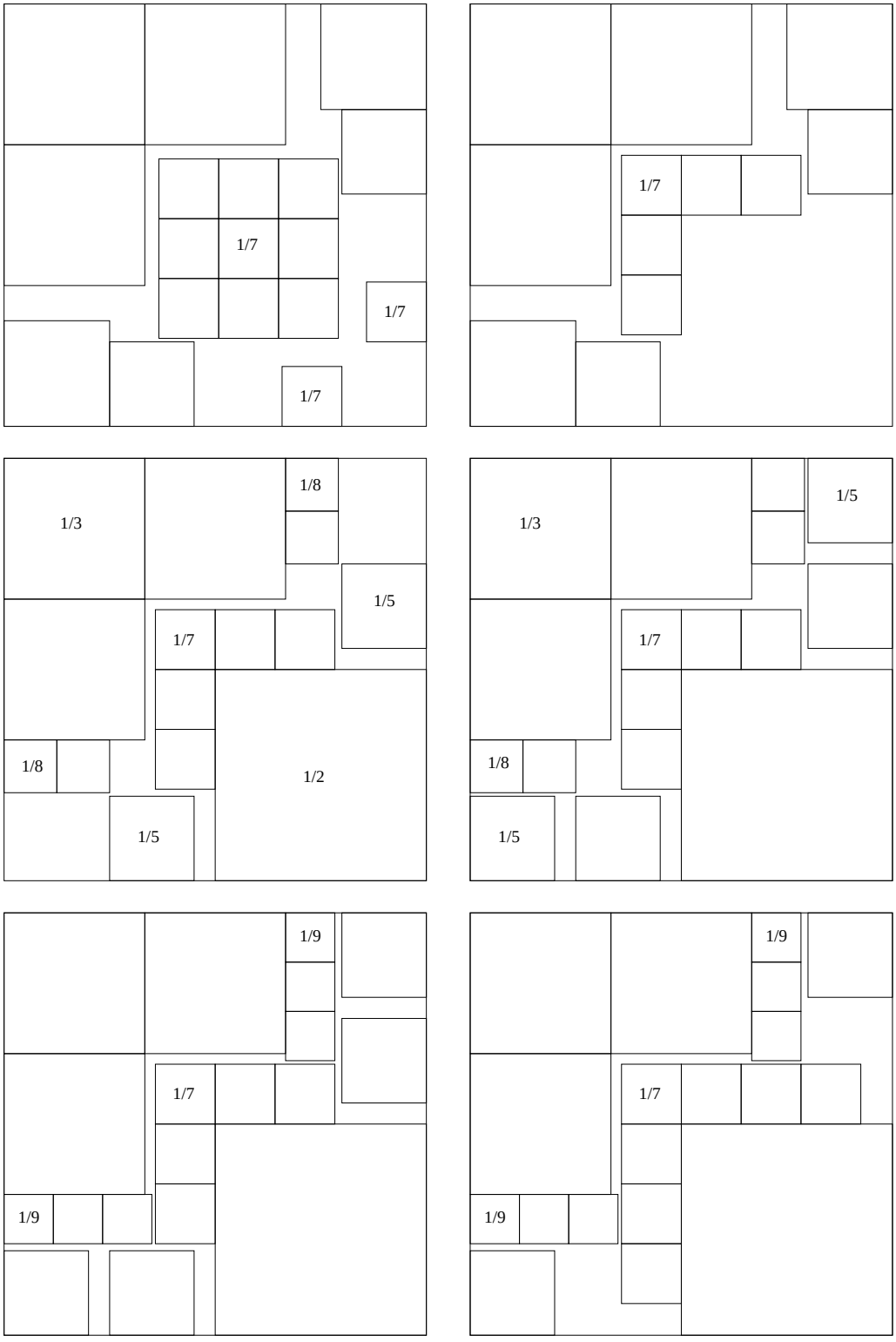

Figure 4: Heights $6/13 \approx 0.461$, $1/3 + 1/7 \approx 0.476$, $1/2$, $11/20$, $5/8$ and $1/4 + 2/5 = 0.65$

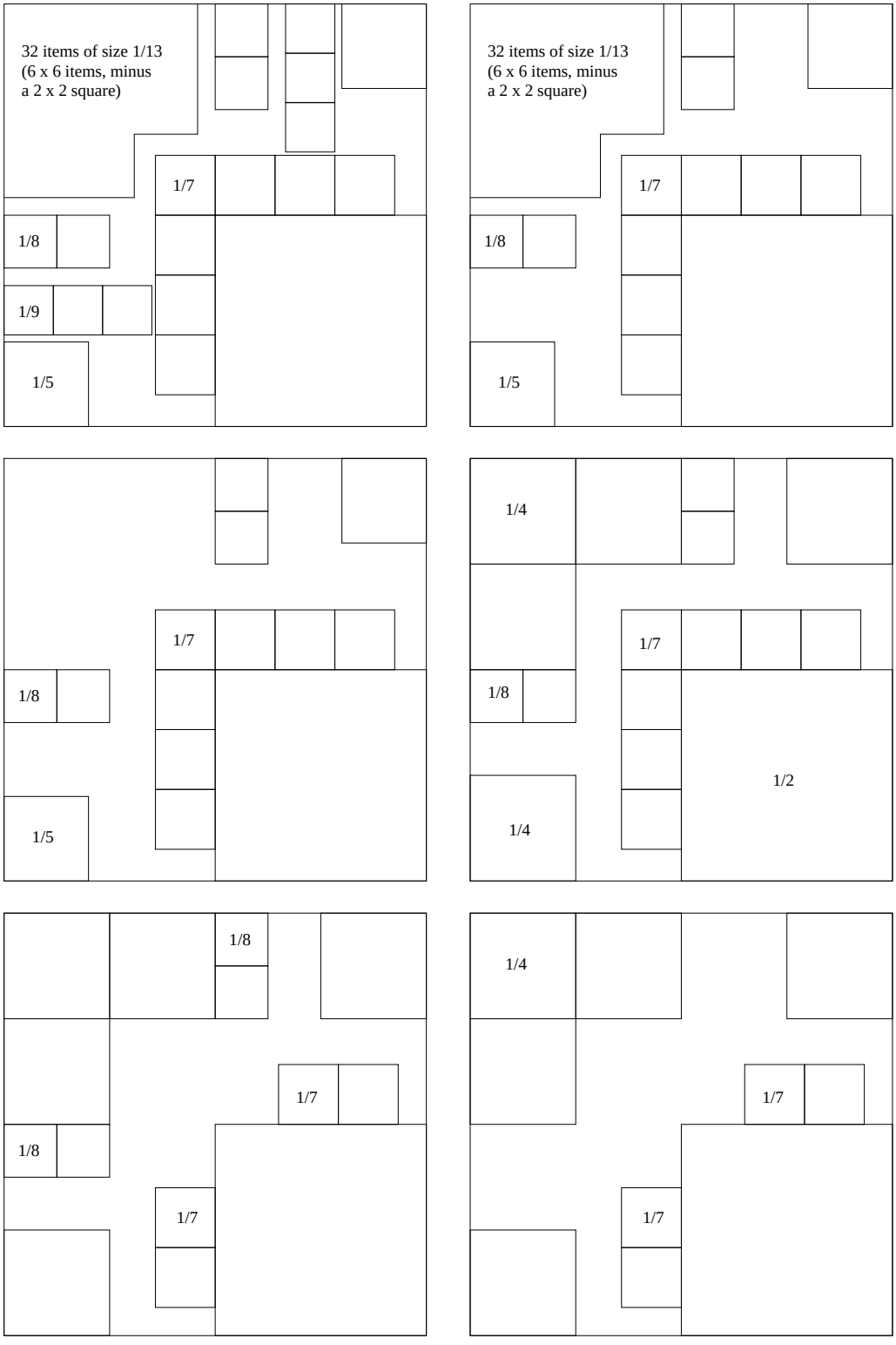

Figure 5: Heights $2/3$, $5/8 + 1/9 \approx 0.736$, $2/3 + 1/13 \approx 0.743$, $3/4$, $1/3 + 3/7 \approx 0.762$, $19/24$

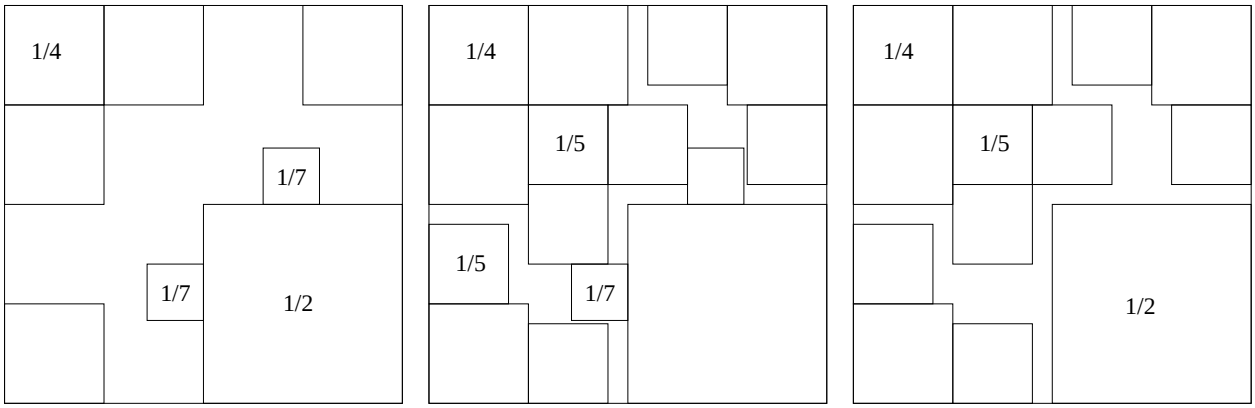

Figure 6: Intersections at heights $1/4 + 2/5 + 1/7 \approx 0.793$, $4/5$, $1/3 + 4/7 \approx 0.905$

5 Higher dimensions

We assign coordinates to the bin: one corner is the origin, and all edges connected to this corner are along a positive axis. All edges have length 1, so the point furthest from the origin is denoted by $(1, 1, \ldots, 1)$.

We now describe where to place the items of each size in the bin. Type i items have size $(1 + \varepsilon)/(i + 1)$.

We will describe anchor points for items of several types. If we say that an item is placed at an anchor point, we mean that one of its corners coincides with the anchor point and no point inside the item has a smaller coordinate than the anchor point. That is, the item is placed parallel to the coordinate axes and as far away as possible from the origin while still intersecting with the anchor point.

$(1 + \varepsilon)/2$ This item is placed with one corner in the origin.

$(1+\varepsilon)/3$ We define type 2 anchor points as follows. All coordinates of a anchor point are equal to $1 - \frac{1}{3}(1 + \varepsilon)$ or $1 - \frac{2}{3}(1 + \varepsilon)$. Since for each coordinate there are two choices, this defines 2^d anchor points in total. Of these, only one is contained in the type 1 item that was placed first: this is the anchor point with all coordinates equal to $1 - \frac{2}{3}(1 + \varepsilon)$. We place one item of type 2 at all other anchor points. It can be seen that no type 2 item overlaps with the type 1 item (since all points inside the type 1 item have at least one coordinate smaller than that of any anchor point).

$(1 + \varepsilon)/4$ Anchorpoints are points with all coordinates equal to 0, $(1 + \varepsilon)/4$ or $(1 + \varepsilon)/2$. There are 3^d such points.

We define two disjoint subsets of this set of anchor points which we will not use. First, the anchor points for which all coordinates are 0 or $\frac{1}{4}(1 + \varepsilon)$ are contained in (or at the edge of) the type 1 item. There are 2^d such points.

Second, the anchor points for which all coordinates are $\frac{1}{4}(1 + \varepsilon)$ or $\frac{1}{2}(1 + \varepsilon)$ and for which at least one coordinate is $\frac{1}{2}(1 + \varepsilon)$ are not suitable because any item placed there would overlap with an item of type 2. (There are $2^d - 1$ such points.) This can be seen as follows: adding the vector

with all coordinates equal to $\frac{1}{4}(1+\varepsilon)$ gives a corner point of the hypothetical type 3 item placed at this anchor point with all coordinates equal to at least $\frac{1}{2}(1+\varepsilon)$ and at least one coordinate $\frac{3}{4}(1+\varepsilon)$. All such points are inside type 2 items.

We place one type 3 item at each of the remaining $3^d - 2^{d+1} + 1$ anchor points. In this way there is no overlap with any previously placed item.

$(1+\varepsilon)/5$ Consider anchor points with all coordinates equal to a multiple of $(1+\varepsilon)/4$ between 0 and 3, and at least one coordinate equal to $\frac{3}{4}(1+\varepsilon)$.

There are $4^d - 3^d$ such anchor points. (Compare to the anchor points we used in the previous step.)

We will only use anchor points of this set that have at least one coordinate equal to 0. There are 3^d points with all coordinates equal to a multiple of $(1+\varepsilon)/4$ between 1 and 3, but 2^d of them have all coordinates equal to $(1+\varepsilon)/4$ or $(1+\varepsilon)/2$ and are thus not in our set of anchor points. In summary, we exclude $3^d - 2^d$ anchor points in this way (they are not suitable).

It can be seen that at all remaining anchor points, an item of type 4 can be placed: consider the coordinate that is 0. Adding $(1+\varepsilon)/5$ to it gives a point where that coordinate is $(1+\varepsilon)/5$. It can be seen that the type 1 item and the type 3 items could not possibly overlap with any anchor point used in this step (or with a point which has all coordinates at least equal to the coordinates of an anchor point), because all those items are placed closer to the origin. Furthermore, all points inside a type 2 item have coordinates that are all at least $(1+\varepsilon)/3$, whereas we showed that points inside a type 4 item placed at an anchor point have one coordinate at most $(1+\varepsilon)/5$.

This shows that we can place $4^d - 2 \cdot 3^d + 2^d$ items of type 4 inside the bin.

$(1+\varepsilon)/61$ Anchorpoints are points with all coordinates equal to multiples of $(1+\varepsilon)/60$ between 0 and 59. It can be seen that all previously defined anchorpoints are in this set, apart from the ones defined for the items of size $(1+\varepsilon)/3$. For each of the $2^d - 1$ anchorpoints for those items, there exists an anchorpoint in the current set which is less than a vector $(\varepsilon, \ldots, \varepsilon)$ away from it.

Moreover, for all previously placed items, each corner point coincides with some anchor point, apart from the items of size $(1+\varepsilon)/3$, for which all corner points (except for ones with one coordinate equal to 1) are less than a vector $(\varepsilon, \ldots, \varepsilon)$ away from some anchor point.

In an empty cube, we could place 60^d items of size $(1+\varepsilon)/61$ by using all possible anchor points. It is straightforward to verify that the larger items take away the following amounts of items of size $(1+\varepsilon)/61$:

- $(1+\varepsilon)/2$: 30^d
- $(1+\varepsilon)/3$: $(2^d - 1)20^d$
- $(1+\varepsilon)/4$: $(3^d - 2^{d+1} + 1)15^d$
- $(1+\varepsilon)/5$: $(4^d - 2 \cdot 3^d + 2^d)12^d$

Let the amount of items of size $(1+\varepsilon)/61$ that we can place be $A = 60^d - 30^d - (2^d - 1)20^d - (3^d - 2^{d+1} + 1)15^d - (4^d - 2 \cdot 3^d + 2^d)12^d$.

This concludes the description of the placement of all the items. Ignoring ε, the remaining empty volume if all these items are placed inside a bin is

$$V = 1 - \frac{1}{2^d} - \frac{2^d - 1}{3^d} - \frac{3^d - 2^{d+1} + 1}{4^d} - \frac{4^d - 2 \cdot 3^d + 2^d}{5^d} - \frac{A}{61^d} \ .$$

Thus we can add sand of almost this volume to the bin.

The weight of type i items is $1/i^d$. Hence the total weight packed in the bin is (for $\varepsilon \to 0$)

$$1 + \frac{2^d - 1}{2^d} + \frac{3^d - 2^{d+1} + 1}{3^d} + \frac{4^d - 2 \cdot 3^d + 2^d}{4^d} + \frac{A}{60^d} + V.$$

The values of this expression for $d = 4, \ldots, 7$ are given in Table 1. It can be seen that this expression tends to 6 for $d \to \infty$, so for large d, the general lower bound of $\Omega(\log d)$ is better.

References

[1] Leah Epstein and Rob van Stee. Optimal online bounded space multidimensional packing. In *Proc. of 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, pages 207–216. ACM/SIAM, 2004.

[2] C. C. Lee and D. T. Lee. A simple online bin packing algorithm. *Journal of the ACM*, 32:562–572, 1985.

[3] J.Y.T. Leung, T.W. Lam, C.S. Wong, G.H. Young, and F.Y.L. Chin. Packing squares into a square. *Journal on Parallel and Distributed Computing*, 10:271–275, 1990.