# Задачи одномерной упаковки и двумерного замощения прямоугольниками и их применение в промышленности

В. В. Осокин, Р. Ф. Алимов, Т. Р. Сытдыков

В данной статье рассматривается решение двух задач: одномерная упаковка в одинаковые контейнеры и двумерное замощение прямоугольной области с программной реализацией на языке PHP. Для первой задачи будет получено приближенное решение с помощью эвристического BFD-алгоритма и метода ветвей и границ. Для второй задачи, в силу специфики ограничений на входные данные и способы замощения, будет использован метод ветвей и границ, позволяющий получить точное решение за премлемое время работы.

**Ключевые слова:** PHP, одномерная упаковка в контейнеры, двумерное замощение.

## Введение

В промышленной вентиляции при конструировании сложных вентиляционных установок, состоящих из множества блоков, широко используются специальные профили, с помощью которых соединяются составные части установок и образуется их каркас. В общем случае в одной установке может быть использовано несколько видов профилей. Если известно, из каких блоков состоит установка и как эти блоки расположены, то можно рассчитать набор требуемых профилей [2]. Поскольку профили, как правило, вырезаются из стандартных деталей с фиксированной длиной, то возникает задача минимизации количества стандартных деталей, требуемых для производства профилей.

Профили различных типов вырезаются из разных деталей, поэтому достаточно рассчитать минимальное количество стандартных деталей независимо для каждого типа профилей и скомбинировать результат. Поэтому в дальнейшем без ограничения общности будем считать, что все вырезаемые профили относятся к одному типу.

Единственным параметром вырезаемых профилей является длина. Следовательно, достаточно рассматривать профили как одномерные сущности. В этом случае полученная задача есть не что иное, как одномерная задача об упаковке в контейнеры [5].

В [3] рассматривался двумерный случай задачи, возникающий при производстве прямоугольных панелей, и применение BFDH-алгоритма для получения приближенного решения. Так же, как и двумерный вариант, одномерная задача является NP-трудной [4]. Тем не менее, наличие всего одного измерения упрощает задачу, поэтому для большей эффективности решение будет производиться в два этапа. Вначале с помощью BFD-алгоритма будет получено приближенное решение, использующее не более $\frac{11}{9} \cdot x + 1$ деталей, где $x$ — оптимальное (минимально возможное) число деталей [6]. Затем будет использовано ограниченное количество итераций метода ветвей и границ. Это позволит улучшить ответ в случае неоптимальной работы BFD-алгоритма и для малого числа панелей всегда получать точное решение.

Другой задачей, часто встречающейся при проектировании вентиляционных установок, является замощение прямоугольного воздуховода фильтрами определенных типоразмеров, каждый из которых характеризуется шириной и высотой. Математически это означает, что требуется замостить непересекающимися прямоугольниками стандартных размеров некоторый прямоугольник произвольных размеров.

Поскольку в общем случае полное замощение не всегда возможно, допустимо неполное замощение при условии, что площадь не замощенной части сечения воздуховода достаточно мала. Ясно, что помимо максимизации покрытой фильтрами площади сечения воздуховода важно минимизировать общую стоимость затраченных фильтров. В общем случае решения, оптимального по обоим параметрам (стоимость фильтров и площадь покрытой ими поверхности), не существует. В зависимости от требований к вентиляционной установке приоритет может отдаваться решению, покрывшую большую площадь, или решению, использовавшему более дешевый набор фильтров. Да-

лее будет подробно описан один из способов определения наилучшего замощения.

Ввиду конструктивных особенностей фильтры всегда должны располагаться в воздуховоде с фиксированной ориентацией (т. е. повороты, в том числе на углы, кратные $90^o$, не допустимы). Кроме того, для удешевления крепления фильтров в воздуховоде рассматривают только такие замощения, при которых фильтры расположены горизонтальными и вертикальными «ярусами» со следующими ограничениями. Во-первых, все горизонтальные и вертикальные ярусы имеют фиксированную длину (при этом длина горизонтального яруса может отличаться от длины вертикального яруса). Во-вторых, все фильтры горизонтальных ярусов имеют одинаковую высоту (но могут отличаться по ширине), а все фильтры вертикальных ярусов — одинаковую ширину (но могут отличаться по высоте). С учетом указанных ограничений будет использован метод ветвей и границ, позволяющий найти наилучшее решение за приемлемое число итераций для используемых в производстве наборов входных данных.

## Вырезание профилей

Пусть имеется неограниченное количество деталей длины $W$. Требуется вырезать $n$ профилей одного типа длины $w_i$, $i = 1, 2, \ldots, n$, затратив при этом минимальное количество деталей.

Будем решать задачу в два этапа. В первую очередь получим достаточно хорошее приближение, используя алгоритм Best Fit Decreasing (BFD). Затем попробуем улучшить полученное решение с помощью метода ветвей и границ.

Алгоритм BFD работает следующим образом. Во время работы поддерживается множество частично использованных деталей, упорядоченных по остаточной длине. Профили рассматриваются в порядке убывания длины и вырезаются из выбранной детали. Деталь выбирается следующим образом: среди всех деталей выбираем деталь с минимальной остаточной длиной, не меньшей длины профиля. Если подходящей детали нет, нужно использовать новую деталь.

Общее время работы алгоритма, очевидно, полиномиально относительно числа профилей $n$. При использовании сбалансированного двоичного дерева поиска асимптотика алгоритма составит $O(n \log n)$

[1]. Программная реализация на языке PHP структуры данных OrderedSet, выступающей в качестве дерева поиска, подробно рассмотрена в [3].

Создадим класс профиля Packer1DSegment. Добавим в класс следующие свойства: номер профиля, длина, номер детали и смещение левого конца при вырезании.

```
class Packer1DSegment {
    public $Index, $W, $X, $ContainerIndex;
    function __construct($_index, $_w){
        $this->Index = $_index;
        $this->W = $_w;
    }
```

Реализуем вспомогательную функцию сравнения отрезков для сортировки по убыванию длины.

```
    public static function compare($p1, $p2){
        if ($p1->W == $p2->W){
            return 0;
        }
        return ($p1->W < $p2->W) ? 1 : -1;
    }
}
```

Для использования двоичного дерева поиска необходимо создать вспомогательный класс для вершины дерева, добавить в него нужные свойства и функцию сравнения (компаратор).

```
class Packer1DSetItem {
    public $Index, $Len;
    function __construct($_index, $_len){
        $this->Index = $_index;
        $this->Len = $_len;
    }


    public static function compare($p1, $p2){
        if ($p1->Len == $p2->Len){
            if ($p1->Index == $p2->Index){
                return 0;
            }
```

```
                return ($p1->Index < $p2->Index) ? -1 : 1;
            }
            return ($p1->Len < $p2->Len) ? -1 : 1;
        }
    }
```

Создадим основной класс, который будет решать задачу о вырезании профилей. Он будет содержать массив вырезаемых профилей, размер стандартных деталей и другую вспомогательную информацию.

```
class Packer1D {
    public $W, $Segments;
    public $ContainersCount;
```

Реализуем алгоритм BFD, используя двоичное дерево поиска (OrderedSet). Отсортируем профили по убыванию длины и проинициализируем дерево.

```
private function BFD() {
    usort($this->Segments, "Packer1DSegment::compare");
    $orderedSet = new OrderedSet(array(), "
        Packer1DSetItem::compare");
    $this->ContainersCount = 0;
```

Для каждого профиля, используя дерево поиска, найдем подходящую деталь.

```
    foreach($this->Segments as $Segment) {
        $tmpSetItem = new Packer1DSetItem(-1, $Segment->W
            );
        $tmpElem = $orderedSet->LowerBound($tmpSetItem);
```

Если в дереве имеется подходящая деталь, будем использовать ее. Сохраним для профиля ее номер и смещение левого конца при вырезании. Удаляем деталь из дерева, уменьшим длину детали на длину профиля и, если длина все еще положительна, снова добавим деталь в дерево.

```
    if (isset($tmpElem)) {
        $Segment->X = $this->W - $tmpElem->Len;
        $Segment->ContainerIndex = $tmpElem->Index;
        $orderedSet->Erase($tmpElem);

    $tmpElem->Len -= ($Segment->W + $this->CutWidth);
    if ($tmpElem->Len > 0) {
        $orderedSet->Insert($tmpElem);
    }
```

Если в дереве нет подходящей детали, будем использовать новую деталь. Обновим свойства профиля и дерево аналогично случаю выше.

```
        } else {
            $Segment->X = 0;
            $Segment->ContainerIndex = $this->ContainersCount
                ;
            $tmpElem = new Packer1DSetItem($Segment->
                ContainerIndex, $this->W - $Segment->W - $this
                ->CutWidth);

                if ($tmpElem->Len > 0) {
                    $orderedSet->Insert($tmpElem);
                }
                $this->ContainersCount++;
            }
        }
    }
```

В результате полученное разрезание деталей на профили будет храниться в полях ContainerIndex и X элементов массива Segments. Общее количество деталей будет записано в свойство ContainersCount.

Использованное количество деталей, как указано в [6], будет удовлетворять соотношению

$$c \leq \frac{11}{9} \cdot x + 1,$$

где $x$ — минимально возможное (оптимальное) количество деталей.

Попробуем улучшить результат, используя метод ветвей и границ. Вначале проинициализируем вспомогательные массивы данных.

```
    private function PrepareForRecursiveSolution(){
        $this->TmpSegments = array();
        $this->SegmentUsed = array();

        for($i = 0; $i < count($this->Segments); $i++){
            $this->TmpSegments[] = clone($this->Segments[$i])
                ;
            $this->SegmentUsed[] = false;
        }
        $this->IterationCount = 0;
    }
```

Реализуем основную рекурсивную функцию метода ветвей и границ. Функция имеет 3 параметра — остаточную длину текущей детали CurContainerSegmentLength, оставшееся количество не вырезанных профилей CurSegmentCount и текущее количество использованных деталей CurContainersCount. Будем использовать ограничение на число рекурсивных вызовов.

```
private function RecursiveSolve(
    $CurContainerSegmentLength, $CurSegmentCount,
    $CurContainersCount){
     $this->IterationCount++;
     if ($this->IterationCount > self::MAX_ITERATION_COUNT
         ){
          return;
     }
```

Если все профили уже вырезаны, попробуем улучшить ответ и выходим из рекурсии.

```
if ($CurSegmentCount == 0){
     if ($CurContainerSegmentLength < $this->W){
          $CurContainersCount++;
     }

     if ($this->ContainersCount == 0 || $this->
         ContainersCount > $CurContainersCount){
          $this->ContainersCount = $CurContainersCount;
          for($i = 0; $i < count($this->Segments); $i++){
               $this->Segments[$i]->ContainerIndex = $this->
                   TmpSegments[$i]->ContainerIndex;
               $this->Segments[$i]->X = $this->TmpSegments[
                   $i]->X;

          }
     }
     return;
}
```

Если не все профили вырезаны и текущее число использованных деталей не меньше текущего минимального ответа, выходим из рекурсии, так как дальнейшие варианты вырезания не улучшат ответ.

```
if ($this->ContainersCount > 0 && $this->ContainersCount
    <= $CurContainersCount){
     return;
}
```

Далее пытаемся вырезать из текущей детали оставшиеся профили.

```
$Flag = false;
for($i = 0; $i < count($this->TmpSegments); $i++){
    if (!$this->SegmentUsed[$i] && $this->TmpSegments[$i
        ]->W <= $CurContainerSegmentLength){
        $Flag = true;
        $this->SegmentUsed[$i] = true;
        $this->TmpSegments[$i]->X = $this->W -
            $CurContainerSegmentLength;
        $this->TmpSegments[$i]->ContainerIndex =
            $CurContainersCount;
```

Рекурсивно вызываем функцию RecursiveSolve (уменьшая длину текущей детали и количество не вырезанных профилей).

```
        $this->RecursiveSolve($CurContainerSegmentLength
            - $this->TmpSegments[$i]->W - $this->CutWidth,
             $CurSegmentCount - 1, $CurContainersCount);
        $this->SegmentUsed[$i] = false;
        if ($CurContainerSegmentLength == $this->W){
            break;
        } else {
            for($j = $i; $j < count($this->TmpSegments)
                && $this->TmpSegments[$i]->W == $this->
                TmpSegments[$j]->W; $j++);
            $i = $j;
        }
    }
}
```

Наконец, если из текущей детали невозможно вырезать никакой профиль, рекурсивно вызываем функцию RecursiveSolve, используя новую деталь (увеличив количество использованных деталей на 1 и обновив длину текущей детали до исходной стандартной длины).

```
if (!$Flag){
    $this->RecursiveSolve($this->W, $CurSegmentCount,
        $CurContainersCount + 1);
}
```

Основная функция, решающая задачу о вырезании профилей, будет иметь следующий вид:

```
public function Pack1D(){
    $this->BFD();
    $this->PrepareForRecursiveSolution();
    $this->RecursiveSolve($this->W, count($this->
        TmpSegments), 0);
}
```

Как упоминалось выше, количество использованных деталей будет указано в свойстве ContainersCount, а само разбиение восстанавливается по свойствам ContainerIndex и X элементов массива Segments. Полученное решение для любого набора профилей будет достаточно близко в оптимальному (это следует из оценки для BFD-алгоритма), для небольших наборов профилей благодаря использованию метода ветвей и границ всегда будет найдено точное решение.

## Замощение воздуховода фильтрами

Пусть задан прямоугольник размера $W$ x $H$. Также задано множество типоразмеров, характеризующихся шириной и высотой $w_i$ x $h_i$, $i = 1, 2, \ldots, n$. В качестве замощений (или покрытий) прямоугольника $W$ x $H$ будем рассматривать множества четверок вещественных чисел $Z = \{(w, h, x, y)\}$, удовлетворяющие следующим условиям:

1) $\forall\ (w, h, x, y) \in Z\ \exists\ i \in 1, 2, \ldots, n$: $w = w_i, h = h_i$. Таким образом, в покрытии участвуют только прямоугольники заданных типоразмеров.

2) $\forall\ (w, h, x, y) \in Z\ 0 \leq x \leq W - w, 0 \leq y \leq H - h$. Прямоугольник $w$ x $h$ размещен в прямоугольнике $W$ x $H$ так, что сторона длины $w$ параллельна стороне длины $W$, сторона длины $h$ — параллельна стороне длины $H$. $(x, y)$ — координаты левого нижнего угла прямоугольника.

3) Для любых различных четверок $(w_1, h_1, x_1, y_1), (w_2, h_2, x_2, y_2) \in Z$ $max(x_1, x_2) \geq min(x_1 + w_1, x_2 + w_2)$ или $max(y_1, y_2) \geq min(y_1 + h_1, y_2 + h_2)$. Другими словами, никакие два прямоугольника в замощении не пересекаются.

4) $\forall\ (w_1, h_1, x_1, y_1), (w_2, h_2, x_2, y_2) \in Z$ $x_1 = x_2 \Rightarrow w_1 = w_2$. Таким образом, замощение состоит из вертикальных ярусов, в каждом

вертикальном ярусе у всех прямоугольников одинаковая ширина и $x$-координата левого нижнего угла.

5) $\forall\ (w_1, h_1, x_1, y_1), (w_2, h_2, x_2, y_2) \in Z\ y_1 = y_2 \Rightarrow h_1 = h_2$. Аналогично, можно считать, что замощение состоит из горизонтальных ярусов, в каждом из которых у всех прямоугольников одинаковая высота и $y$-координата левого нижнего угла.

6) $\forall\ (w_1, h_1, x_1, y_1), (w_2, h_2, x_2, y_2) \in Z\ (w_1, h_2, x_1, y_2) \in Z$, $(w_2, h_1, x_2, y_1) \in Z$. Таким образом, вертикальные и горизонтальные ярусы образуют своеобразную «сетку».

Далее нужно определить, каким способом из всех замощений будет выбираться наилучшее. Это можно сделать множеством способов, будем использовать один из наиболее простых. Добавим дополнительные условия, которым должно удовлетворять оптимальное покрытие.

7) Невозможно добавить вертикальный ярус к покрытию.

8) Площадь покрытой части прямоугольника $W$ x $H$ составляет не менее $k$ процентов от его общей площади.

9) Количество покрывающих прямоугольников минимально возможное.

Минимизация количества покрывающих прямоугольников может быть обобщена — вместо количества прямоугольникам можно присвоить веса, соответствующие стоимости фильтров, и минимизировать суммарный вес. Рассмотрение этого обобщения выходит за рамки данной статьи.

Будем использовать алгоритм, использующий особенности ярусного строения замощений. Заметим, что по составу прямоугольников одного горизонтального яруса и одного вертикального яруса легко восстановить все прямоугольники покрытия. Поэтому для генерации всех покрытий и выбора среди них оптимального будем строить два яруса — один горизонтальный и один вертикальный, восстанавливая затем все прямоугольники покрытия.

Если бы ширина и высота всех допустимых прямоугольников в покрытиях была различна, задача решалась бы тривиально, так как

любое замощение состояло бы из одинаковых прямоугольников (в противном случае оно не было бы ярусным). На практике, однако, такие простые случаи встречаются очень редко — чаще всего для каждого типоразмера имеется несколько других типоразмеров, у которых ширина или высота совпадает. Поэтому будем решать задачу методом ветвей и границ. Далее будет показано, что при определенных условиях, которым соответствуют встречающиеся на практике наборы входных данных, метод будет работать достаточно быстро.

Общая схема работы алгоритма такова. Вначале для каждой допустимой высоты типоразмера рекурсивно строим горизонтальный ярус. Далее для каждого построенного горизонтального яруса будем строить подходящий вертикальный ярус. Используя полученные ярусы, восстанавливаем покрытие и среди всех таких покрытий выбираем оптимальное.

Приведем программную реализацию описанного алгоритма на языке PHP. Основная функция, вызывающая рекурсивное решение, будет выглядеть следующим образом. Вначале инициализируются переменные, далее для каждого типоразмера вызывается рекурсивное построение горизонтального яруса соответствующей высоты.

```
public function Cover2D(){
    $this->ClearBestValues();
    $this->ClearCurValues();


    foreach($this->MapHW as $FirstH => $VectorWidths){
        if ($FirstH <= $this->H){
            $this->RecursiveGoWidth($FirstH);
        }
    }
}
```

Рекурсивное построение горизонтального яруса будет принимать в качестве параметра высоту яруса. Прямоугольники, образующие ярус, будут сохраняться в массив CurVectorW. Добавление и удаление прямоугольника в данном массиве будет осуществляться функциями PushBackWidth и PopBackWidth, для краткости опустим их код.

```
public function RecursiveGoWidth($FirstH){
    $LastW = $this->GetMinW();
```

После начальной инициализации переменных пытаемся добавить новый прямоугольник в горизонтальный ярус и рекурсивно достроить горизонтальный ярус.

```
foreach ($this->MapHW[$FirstH] as $NewW){
    if (($this->W >= $this->CurW + $NewW) && ($NewW
        <= $LastW)){
        $this->PushBackWidth($NewW);
        $this->RecursiveGoWidth($FirstH);
        $this->PopBackWidth();
    }
}
```

Теперь вызываем рекурсивное построение вертикального яруса. Вначале по прямоугольникам горизонтального яруса выясним, какие прямоугольники могут быть использованы для построения вертикального яруса.

```
$PossibleHeights = array();
foreach($this->MapHW as $NewH => $ArrayWidths){
    $CanBeUsed = true;
```

Прямоугольники с текущей высотой могут быть использованы, если для каждого прямоугольника из горизонтального яруса существует соответствующий прямоугольник с данной высотой.

```
    foreach($this->CurVectorW as $UsedW){
        $IsFound = $this->FindInArray($ArrayWidths,
            $UsedW);
        if (!$IsFound){
            $CanBeUsed = false;
            break;
        }
    }

    if ($CanBeUsed){
        $PossibleHeights[] = $NewH;
    }
}
```

Вызываем функцию построения вертикального яруса.

```
    $this->RecursiveGoHeight($PossibleHeights);
}
```

Построение вертикального яруса делается аналогично. Для ускорения будем запоминать результаты вычислений для набора высот.

```
public function RecursiveGoHeight($PossibleHeights){
```

Если набор высот уже использовался, то используем ранее вычисленные и сохраненные значения.

```
$Hash = $this->GetHashArray($PossibleHeights);
if (array_key_exists($Hash, $this->MapHH) && $this->CurH
    == 0){
    $this->CurH = $this->MapHH[$Hash];
    $this->CurVectorH = $this->MapHVec[$Hash];
    $this->UpdateBestCovering();
    $this->CurH = 0;
    $this->CurVectorH = array();
    return;
}
```

Пытаемся добавить прямоугольник в вертикальный ярус.

```
$LastH = $this->GetMinH();
$Flag = false;
foreach ($PossibleHeights as $NewH){
    if (($this->H >= $this->CurH + $NewH) && ($NewH <=
        $LastH)){
        $Flag = true;
        $this->PushBackHeight($NewH);
        $this->RecursiveGoHeight($PossibleHeights);
        $this->PopBackHeight();
    }
}
```

Если вертикальный ярус невозможно увеличить, пытаемся обновить ответ.

```
if (!$Flag){
    $this->UpdateBestCovering();
    $this->MapHH[$Hash] = $this->CurH;
    $this->MapHVec[$Hash] = $this->CurVectorH;
}
```

Функция UpdateBestCovering обновляет ответ с учетом процента покрытой поверхности и количества прямоугольников.

```
public function UpdateBestCovering(){
    if ($this->CurW * $this->CurH * 100 >= $this->W *
        $this->H * $this->k
         && (count($this->CurVectorW) * count($this->
             CurVectorH) < count($this->BestVectorW) *
             count($this->BestVectorH)
         || $this->CurW * $this->CurH > $this->BestW *
             $this->BestH
         && count($this->CurVectorW) * count($this->
             CurVectorH) == count($this->BestVectorW) *
             count($this->BestVectorH))){

         $this->BestW = $this->CurW;
         $this->BestH = $this->CurH;
         $this->BestVectorW = $this->CurVectorW;
         $this->BestVectorH = $this->CurVectorH;
    }
}
```

Оценим время работы алгоритма. Количество рекурсивных вызовов RecursiveGoWidth соответствует суммарному количеству прямоугольников во всех возможных способах построения горизонтальных ярусов, при которых прямоугольники упорядочены по ширине. Количество прямоугольников в горизонтальном ярусе зависит от ширины прямоугольников. Как правило, ширина воздуховодов не превышает 10000 мм, а ширина фильтров различных типоразмеров колеблется от 100 мм до 2000 мм. Таким образом можно считать, что количество прямоугольников в горизонтальных ярусах не превосходит 100 (и в среднем значительно меньше 100).

Количество различных горизонтальных ярусов фиксированной высоты, в которых прямоугольники упорядочены по ширине, соответствует количеству способов разбиения чисел, не превосходящих $W$, на слагаемые, равные $w_i$, и, соответственно, экспоненциально зависит от количества различных типоразмеров фильтров с одинаковой высотой. На практике, как правило, бывает не более 3-4 фильтров различной ширины и одинаковой высоты. С учетом ограничений на размеры фильтров и воздуховодов получаем, что количество различных вариантов строения горизонтальных ярусов не превосходит нескольких миллионов. Аналогичное верно и для количества различных вариантов строения вертикальных ярусов. Соответственно, суммарное количество итераций в рекурсии ограничено несколькими миллиона-

ми. Подробный анализ количества операций при каждом рекурсивном вызове, и также время работы функций обновления ответа и инициализации данных выходит за рамки данной статьи. Отметим лишь, что на реальных входных данных, использованных при проектировании вентиляционных установок (до 13 типоразмеров фильтров, до 4 типоразмеров с одинаковой шириной и высотой, размеры воздуховодов до 3 метров) время работы алгоритма не превосходило 50 миллисекунд.

## Список литературы

[1] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. — 2-е изд.: Пер. с англ. — М.: «Вильямс», 2005. — 1296 с.

[2] Кудрявцев В. Б., Осокин В. В. ВЕНТИЛЯЦИЯ [программирование расчетов промышленного оборудования]. — М.: Интеллектуальные системы, 2016. — С. 212–214.

[3] Кудрявцев В. Б., Осокин В. В. ВЕНТИЛЯЦИЯ [программирование расчетов промышленного оборудования]. — М.: Интеллектуальные системы, 2016. — С. 218–227.

[4] А. В. Смирнов. О задаче упаковки в контейнеры. — УМН, 46: 4(280). — 1991. — С. 173–174

[5] Cofman E.G., Garey M. R., Johnson D. S. Approximation algorithms for bin-packing. — An updated survey // Algorithm Design for Computer System Design. CISM courses and lectures. — 1984. — № 284. — P. 49–106.

[6] Johnson, D. S., Demers, A., Ullman, J. D., Garey, M. R., and Graham, R. L. Worst-case Performance Bounds for Simple One-Dimensional Packing Algorithms. — SZAM Journal onComputing, 3(4), 299–325 (1974).