Задача «Удалить цифру из числа»

Краткое условие без легенды

Дано число n, требуется убрать из него одну цифру так, чтобы получилось максимальное число.

Пример

Например, если есть число 3223, из него можно получить четыре различных числа.

1. Если убрать цифру на первой позиции, мы получим число 223.
2. Если убрать цифру на второй позиции, мы получим число 323.
3. Если убрать цифру на третьей позиции, мы получим число 323.
4. Если убрать цифру на четвёртой позиции, мы получим число 322.

Наибольшим числом из этих всех является 323, именно оно и будет ответом.

Неверное решение

Первое решение, которое приходит в голову — попробовать удалить каждую цифру и затем проверить, какое число получится наибольшим.
К сожалению, это решение будет работать очень долго. Обычно на практике считают, что сравнение чисел работает за $O(1)$, но на самом деле это не так — оно работает за $O(n)$, где n — длина числа.
То есть такое решение будет работать за $O(n^2)$, что нам не подходит.

Верное решение

Чтобы найти верное решение, нужно заметить два важных факта.
1) В каком случае нам вообще выгодно удалять цифру?
2) Какое из удалений самое выгодное для нас?

Выгодно ли удаление

У нас возможны три варианта удаления цифры a в числе $*ab*$.

1. Мы удаляем a такую, что $a = b$. Такое удаление эквивалентно удалению b.

2. Мы удаляем a такую, что $a > b$. Такое удаление менее выгодно, чем удаление b, так как число $\star a\star > \star b\star$.

3. Мы удаляем a такую, что $a < b$. Такое удаление нам уже выгодно. Только такие удаления мы и будем рассматривать.

Вариант $*a$, то есть последний символ, мы всегда будем рассматривать.

Пример

Допустим, у нас есть число 3 233 423. Рассмотрим все позиции:

1. Рассмотрим первую позицию. $3 > 2$, а следовательно, такое удаление нам невыгодно. И в самом деле $233\,423 < 333\,423$.

2. Рассмотрим вторую позицию. $2 < 3$, а следовательно, такое удаление нам выгодно. И в самом деле $333\,423 > 323\,423$.

3. Рассмотрим третью позицию. $3 = 3$, а следовательно, такое удаление нам невыгодно. И в самом деле $323\,423 = 323\,423$.

4. Рассмотрим четвёртую позицию. $3 < 4$, а следовательно, такое удаление нам выгодно. И в самом деле $323\,423 > 323\,323$.

5. Рассмотрим пятую позицию. $4 > 2$, а следовательно, такое удаление нам невыгодно. И в самом деле 323 323 $<$ 323 343.

6. Рассмотрим шестую позицию. $2 < 3$, а следовательно, такое удаление нам выгодно. И в самом деле 323 343 $>$ 323 342.

7. Рассмотрим седьмую позицию. Она последняя, а следовательно, такое удаление нам выгодно.

Итого мы будем рассматривать варианты:

1. Удалим вторую позицию и получим число 333 423.

2. Удалим четвёртую позицию и получим число 323 423.

3. Удалим шестую позицию и получим число 323 343.

4. Удалим седьмую позицию и получим число 323 342.

Какое удаление нам выгоднее всего?

Таким образом, к сожалению, мы все еще можем получить линейное количество проверок (например, для чисел вида 1 234 567 890 123...).
Но это можно сделать быстрее. Давайте научимся находить лучшее удаление за $O(1)$.
Допустим, у нас есть число $\star_1 ac \star_2 bd \star_3$, в котором нам выгодно и удаление a, и удаление b.
Посмотрим на полученные числа $\star_1 c \star_2 bd \star_3$ и $\star_1 ac \star_2 d \star_3$. Так как префикс $\star_1$ в обоих случаях один и тот же, сравнение будет по следующему символу, но, так как $a < b$, первое число будет точно больше.
Таким образом, нам выгоднее удалять первое из выгодных удалений.

Пример

Допустим, у нас есть число 3 233 423. Рассмотрим все позиции.
В нём мы будем рассматривать варианты:

1. Удаляем 2 позицию и получим число 333 423.
2. Удаляем 4 позицию и получим число 323 423.
3. Удаляем 6 позицию и получим число 323 343.
4. Удаляем 7 позицию и получим число 323 342.

И в самом деле первое число получается наибольшим.

Время работы и память

Мы пройдемся по числу и для каждой позиции за $O(1)$ проверим, выгодна ли она, вернув первую выгодную позицию. Таким образом, финальное время работы — $O(n)$.
Дополнительной памяти требуется $O(1)$.

Задача «YAML to INI»

Легенда

Необходимо реализовать конвертер из формата YAML в формат INI. Каждый набор входных данных является корректным yaml-файлом со следующими ограничениями:

- yaml-файл состоит только из словарей произвольной вложенности.
- Табуляция обозначается ровно четырьмя символами пробела.
- Каждая строка содержит объявление словаря (`section:`) или объявление пары «ключ-значение» (`key: value`), где ключ и значение разделены символами двоеточия и пробелом.
- Все имена словарей, ключи и значения непустые, состоят из строчных латинских букв и имеют длину не больше 10.
- В каждом словаре все имена ключей и непосредственно вложенных словарей уникальны.
- В каждом словаре содержится как минимум одна пара «ключ-значение», либо на некотором уровне вложенности этого словаря содержится пара «ключ-значение». Иными словами, пустых словарей нет.

Преобразуйте эти yaml-файлы в формат INI.

- Пара «ключ-значение» форматируется как `key = value`, то есть ключ и значение разделены символами пробела, равно и пробела.
- Порядок вывода пар «ключ-значение» должен оставаться **строго** таким же, как в исходном yaml-файле.

- Перед каждой парой «ключ-значение» следует вывести путь до неё, то есть имена словарей в квадратных скобках, разделённые точкой.
- Если две или более подряд идущие пары «ключ-значение» имеют одинаковый путь, его следует вывести ровно один раз для всей группы пар «ключ-значение».
- Если путь до некоторой пары «ключ-значение» (или группы подряд идущих пар) пустой, то есть не содержит ни одного словаря, путь следует опустить.
- Все описания групп пар «ключ-значение» должны быть разделены одной пустой строкой.
- После вывода очередного ini-файла следует вывести одну пустую строку.

Гарантируется, что размер входных и выходных данных не превосходит 10 Мб.

Разбор

В этой задаче вам нужно аккуратно реализовать переводчик из формата YAML в формат INI по формальным правилам, которые описаны в условии.
Давайте сначала решим более простую задачу, где не надо группировать в один блок пары «ключ-значение», которые находятся в одном словаре. Для этого будем читать входной файл построчно и поддерживать стек из названий словарей. Отличить строчку с объявлением словаря от строчки с парой «ключ-значение» легко: строчка с объявлением словаря заканчивается символом двоеточия. Для определения уровня вложенности очередного словаря посчитаем количество пробелов в начале строки и поделим на размер табуляции (4).

```
func simple() {
   n := ReadInt() // считать количество строк
   _ = ReadLine() // считать перенос строки

   stack := make([]string, 0) // стек с названиями словарей
   for range n {
      s := ReadLine() // считать очередную строку
      s = s[:len(s)-1] // отрезать символ переноса строки
      level := (len(s) - len(strings.TrimSpace(s))) / 4 // уровень
   вложенности словаря
      stack = stack[:level] // обрезать стек по текущей вложенности
      if s[len(s)-1] == ':' { // если словарь
         stack = append(stack, strings.TrimSpace(s[:len(s)-1])) //
   добавить словарь в стек
      } else { // если пара «ключ значение»
         if len(stack) > 0 { // если стек не пустой
            Println("[" + strings.Join(stack, ".") + "]") //
   напечатать все словари через точку
         }
         split := strings.Split(s, ":") // разбить пару «ключ значение»
   на ключ и значение
         Println(strings.TrimSpace(split[0]) + " = " + strings.
   TrimSpace(split[1])) // напечатать пару «ключ значение»
         Println() // напечатать пустую строку
      }
   }
}
```

Для решения основной задачи давайте явно сравнивать текущий стек с предыдущим. Если они различаются, надо печатать описание очередного блока в квадратных скобках.

```
func ineffective() {
   n := ReadInt() // считать количество строк
   _ = ReadLine() // считать перенос строки

   stack := make([]string, 0) // стек с названиями словарей
   prevStack := make([]string, 0) // стек который был напечатан на
   предыдущем шаге
   needLine := false // нужно ли печатать пустую строку перед блоком
   for range n {
      s := ReadLine() // считать очередную строку
      s = s[:len(s)-1] // отрезать символ переноса строки
      level := (len(s) - len(strings.TrimSpace(s))) / 4 // уровень
   вложенности словаря
      stack = stack[:level] // обрезать стек по текущей вложенности
      if s[len(s)-1] == ':' { // если словарь
```

```
14          stack = append(stack, strings.TrimSpace(s[:len(s)-1])) //
    добавить словарь в стек
15      } else { // если пара «ключ значение»
16          if !reflect.DeepEqual(stack, prevStack) { // если стек
    изменился
17              if needLine { // если нужно напечатать пустую строку перед
    блоком
18                  Println() // напечатать пустую строку
19              }
20              if len(stack) > 0 { // если стек не пустой
21                  Println("[" + strings.Join(stack, ".") + "]") //
    напечатать все словари через точку
22              }
23          }
24          needLine = true // теперь нужно печатать пустую строку перед
    каждым блоком
25          split := strings.Split(s, ":") // разбить пару «ключ значение»
    на ключ и значение
26          Println(strings.TrimSpace(split[0]) + " = " + strings.
    TrimSpace(split[1])) // напечатать пару «ключ значение»
27      }
28      prevStack = make([]string, len(stack)) // аллоцировать стек для
    сохранения
29      copy(prevStack, stack) // сохранить текущий стек
30  }
31  Println() // напечатать пустую строку
32 }
```

Сравнивать стеки явно не очень эффективно, так как сохранение стека работает за линейное время. Вместо этого нам достаточно сравнивать только их уровни вложенности.

```
1  func effective() {
2      n := ReadInt() // считать количество строк
3      _ = ReadLine() // считать перенос строки
4
5      stack := make([]string, 0) // стек с названиями словарей
6      prevLevel := 0 // вложенность стека который был напечатан на предыдущем
    шаге
7      needLine := false // нужно ли печатать пустую строку перед блоком
8      for range n {
9          s := ReadLine() // считать очередную строку
10         s = s[:len(s)-1] // отрезать символ переноса строки
11         level := (len(s) - len(strings.TrimSpace(s))) / 4 // уровень
    вложенности словаря
12         stack = stack[:level] // обрезать стек по текущей вложенности
13         if s[len(s)-1] == ':' { // если словарь
```

```
14          stack = append(stack, strings.TrimSpace(s[:len(s)-1])) //
    добавить словарь в стек
15      } else { // если пара «ключ значение»
16          if level != prevLevel { // если стек изменился
17              if needLine { // если нужно напечатать пустую строку перед
    блоком
18                  Println() // напечатать пустую строку
19              }
20              if len(stack) > 0 { // если стек не пустой
21                  Println("[" + strings.Join(stack, ".") + "]") //
    напечатать все словари через точку
22              }
23          }
24          needLine = true // теперь нужно печатать пустую строку перед
    каждым блоком
25          split := strings.Split(s, ":") // разбить пару «ключ значение»
    на ключ и значение
26          Println(strings.TrimSpace(split[0]) + " = " + strings.
    TrimSpace(split[1])) // напечатать пару «ключ значение»
27      }
28      prevLevel = level // запомнить текущий уровень вложенности
29  }
30  Println() // напечатать пустую строку
31 }
```

Время работы и память

Итого получаем эффективное решение, которое работает за линейное время относительно размера входных и выходных данных. Так как их размер ограничен 10 Мб по условию, решение на любом языке укладывается в ограничение по времени.

Задача «Крестики-нолики»

Краткое условие без легенды

У нас есть доска размером n x m, на которой уже размещены крестики (`X`), нолики (`0`) и пустые клетки (`.`). Задача состоит в том, чтобы определить, можно ли поставить **ровно один** крестик так, чтобы крестики победили, то есть образовалась последовательность хотя бы из k крестиков.

Пример

$k = 3$. Столбцов 3, строк 3.

.	X	.
.	X	0
0	.	X

Сначала мы проверим, нет ли победителей среди `0` в случае, если последовательность из `0` уже собрана — завершим решение, ответив `NO`. Далее посмотрим, есть ли `X` на доске, и проверим их расположение и количество, при условии что мы находим первый порядок расположения элементов, удовлетворяющий условию задачи. Завершим решение, ответив `YES`.

Разбор

Идея решения заключается в проверке всех возможных комбинаций расположения `0` и `X` на предмет выигрышной комбинации k.

```
    a[i] = make([]int8, m)
    s := ReadLine()
    for j := int32(0); j < m; j++ {
      switch s[j] {
      case '.':
        a[i][j] = 2
      case '0':
```

```
        a[i][j] = 0
      case 'X':
        a[i][j] = 1
      }
    }
  }
```

Считываем входные данные и формируем доску по заданным параметрам.

Основные функции решения задачи

```
  ok := func(x, y int32) bool {
    return 0 <= x && x < n && 0 <= y && y < m
  }
```

X и Y являются текущими позициями таргета. N и M — количество столбцов и строк в доске. Функция вернёт `true` в случае, если позиция таргета корректна, иначе вернёт `false`.

```
  win := func(x, y, dx, dy int32) bool {
    sum := make([]int32, 3)
    for ok(x, y) {
      sum[a[x][y]]++
      x += dx
      y += dy
      if ok(x-k*dx, y-k*dy) {
        if sum[0] == k || sum[1] == k {
          return true
        }
        sum[a[x-k*dx][y-k*dy]]--
      }
    }
    return false
  }
```

Функция `win` используется для проверки факта того, есть ли в заданных координатах последовательность из `X` или `0`. Направление проверки задается вектором (dx, dy).

Например:

- (1, 0) для проверки вертикального направления (сверху вниз);

- (1, 1) — для диагонали (сверху слева → вниз направо).

Функция подсчитывает количество элементов по заданным направлениям в скользящем окне и возвращает булево значение.

```
preWin := func(x, y, dx, dy int32) bool {
  sum := make([]int32, 3)
  for ok(x, y) {
    sum[a[x][y]]++
    x += dx
    y += dy
    if ok(x-k*dx, y-k*dy) {
      if sum[1] == k-1 && sum[2] == 1 {
        return true
      }
      sum[a[x-k*dx][y-k*dy]]--
    }
  }
  return false
}
```

Функция `preWin` — аналог функции выше с одним исключением: здесь происходит проверка того, можем ли мы поставить один `X` так, чтобы собрать выигрышную позицию.
Функция считает количество крестиков $sum[1]$ и пустых клеток $sum[2]$ по заданному k. Если есть одна пустая клетка и количество крестиков равно $(k-1)$, функция вернёт `true`, иначе вернёт `false`. Используем заданные функции в цикле для нахождения выигрышной комбинации:

```
type pair struct {
  f   func(int32, int32, int32, int32) bool
  ans string
}

for _, p := range []pair{{win, "NO"}, {preWin, "YES"}} {
  // right
  for i := int32(0); i < n; i++ {
    if p.f(i, 0, 0, 1) {
      Println(p.ans)
      return
    }
  }

```

```
15      // down
16      for j := int32(0); j < m; j++ {
17        if p.f(0, j, 1, 0) {
18          Println(p.ans)
19          return
20        }
21      }
22
23      // right down
24      for i := int32(0); i < n; i++ {
25        if p.f(i, 0, 1, 1) {
26          Println(p.ans)
27          return
28        }
29      }
30      for j := int32(0); j < m; j++ {
31        if p.f(0, j, 1, 1) {
32          Println(p.ans)
33          return
34        }
35      }
36
37      // left down
38      for j := int32(0); j < m; j++ {
39        if p.f(0, j, 1, -1) {
40          Println(p.ans)
41          return
42        }
43      }
44      for i := int32(0); i < n; i++ {
45        if p.f(i, m-1, 1, -1) {
46          Println(p.ans)
47          return
48        }
49      }
50    }
51    Println("NO")
52  }
```

Проверяем комбинации на предмет уже выигрышной комбинации из `X` или `O` через {`win, "NO"`}.
Если победителя нет, пытаемся найти выигрышную комбинацию путём вставки одного крестика через {`preWin, "YES"`}.

Время работы и память

Для начала давайте возьмём вариант решения задачи через brute force — полный перебор всех возможных вариантов решения.
Для каждой клетки $O(n \cdot m)$ мы будем проверять каждое направление $O(k)$.
По сути, для одной проверки временная сложность будет занимать $O(n \cdot m \cdot k)$. Но если мы хотим проверить варианты возможной вставки крестика, временная сложность возрастет до $O((n \cdot m)^2 \cdot k)$.
В нашем случае мы проверяем уже существующие комбинации и не пытаемся на каждой итерации создать новую комбинацию.
Наше решение в функциях `win` и `preWin` просканирует каждую вертикаль, горизонталь и диагональ по одному разу. Таким образом, цикл с вызовами `win` и `preWin` работает за $O(n \cdot m)$.
В этом случае при увеличении доски и расположения комбинаций наш алгоритм будет кратно быстрее по сравнению с методом перебора вариантов. При этом мы проверяем все возможные ситуации на доске, уменьшаем число проверок путем обновления счетчиков в скользящем окне, учитываем возможный вариант раннего завершения решения.
Brute force имеет квадратичную зависимость от размера доски, тогда как наш алгоритм имеет линейную зависимость.

Задача «Галерея искусств»

Краткое условие без легенды

Дано n коробок — $[a_i, b_i]$ и m картин — $[c_i, d_i]$, где размеры — ширина и длина.
Считается, что картина помещается в коробку, когда минимальная и максимальная стороны картины меньше либо равны минимальной и максимальной сторонам коробки соответственно, при этом коробки могут уместить неограниченное количество картин.
Требуется найти такое минимальное количество коробок, чтобы в них поместились все картины.

Пример

Допустим, у нас есть 3 коробки:

1. 1 10
2. 3 3
3. 5 2

Допустим, у нас есть 3 картины:

1. 4 2
2. 9 1
3. 2 2

Тогда:

1. первую картину можно поместить в третью коробку.
2. вторую картину можно поместить в первую коробку.
3. третью картину можно поместить во вторую и третью коробки.

Следовательно, нам нужно взять 2 коробки: третью и первую.

Разбор

Первым делом давайте все коробки и все картины запишем так, чтобы ширина была всегда меньше, чем длина. Для этого в некоторых случаях перевернем коробки: если $a_i > b_i$, сделаем $swap(a_i, b_i)$. То же самое сделаем и с c_i, d_i.
Теперь отсортируем коробки и картины по убыванию ширины и длины, то есть по первому числу.

Пример

Если у нас снова есть 3 коробки:

1. 1 10
2. 3 3
3. 5 2

И есть 3 картины:

1. 4 2
2. 9 1
3. 2 2

После проделанных операций мы получим 3 коробки:

1. 3 3
2. 2 5
3. 1 10

И 3 картины:

1. 2 4
2. 2 2
3. 1 9

Продолжение разбора

Теперь общий подход к решению будет следующим.
Будем идти по картинам начиная от картины с наибольшей шириной к наименьшей.
Тогда для каждой картины мы можем взять либо какую-то имеющуюся коробку, либо новую.
Проверить имеющиеся коробки достаточно просто: у них всех ширина больше или равна нашей, так как мы ищем от наибольшей ширины к наименьшей. Следовательно, нас интересует взятая коробка с наибольшей длиной (а это можно просто запоминать).
Новую коробку же взять также достаточно просто. Нужно двумя указателями или бинпоиском найти первую коробку с шириной меньше, чем наша, и из всех коробок перед ней взять коробку с наибольшей длиной.
Если использовать бинпоиск, нужно дополнительно предварительно посчитать структуру для максимума на префиксе. Если использовать два указателя, нужно поддерживать максимум ширины из взятых и невзятых коробок.

Пример выбора коробок

Расмотрим тот же самый пример:

1. 3 3
2. 2 5
3. 1 10

И 3 картины:

1. 2 4
2. 2 2
3. 1 9

Решение будет работать следующим образом:

1. Мы еще не взяли никаких коробок, а следовательно, ищем новую коробку. Для первой картины по длине нам подходит либо первая, либо вторая коробка. Так как ширина второй коробки больше, берем коробку (2, 5).

2. Нам подходит уже взятая коробка (2, 5). Новые не берем.

3. Взятые коробки нам не подходят. По длине нам подходят все 3 коробки, ширина наибольшая у 3 коробки, следовательно, ее и возьмем.

Пример полного решения

Давайте теперь разберем полное решение. Авторское решение использовало два указателя, так что будем использовать именно его, а не бинпоиск.
Возьмем пример посложнее. Например, 5 коробок:

1. 1 8

2. 6 2

3. 3 4

4. 6 6

5. 10 2

5 картин:

1. 1 2

2. 2 7

3. 4 3

4. 3 5
5. 6 4

Приведем его к правильному виду, то есть поменяем местами длину и ширину и отсортируем.

5 коробок:

1. 6 6
2. 3 4
3. 2 10
4. 2 6
5. 1 8

5 картин:

1. 4 6
2. 3 5
3. 3 4
4. 2 7
5. 1 2

Поставим первый указатель в начале массива коробок, а второй — в начале массива картин. Изначально максимальная ширина из уже взятых коробок — -1 (так как еще ничего не взяли), из рассмотренных — -1 (так как еще ничего не рассмотрели).

$ptr_{boxes} = 0, ptr_{images} = 0, width\_taken = -1, width\_all = -1, count = 0$

1. Картина под указателем ptr_{images} по длине вмещается в коробку ptr_{boxes}. Следовательно, обновим максимальную ширину рассмотреных коробок до 6 и сдвинем ptr_{boxes}.

   $ptr_{boxes} = 1, ptr_{images} = 0, width\_taken = -1, width\_all = 6, count = 0$

2. Картина под указателем ptr_{images} по длине не вмещается в коробку ptr_{boxes}. Следовательно, нужно решить, брать ли новую коробку и сдвинуть ptr_{images}. Брать коробку нужно, так как $width\_taken < width_{box}, width\_all > width_{box}$.

   $ptr_{boxes} = 1, ptr_{images} = 1, width\_taken = 6, width\_all = 6, count = 1$

3. Картина под указателем ptr_{images} по длине вмещается в коробку ptr_{boxes}. Максимальная ширина не обновляется. Передвигаем указатель коробок.

   $ptr_{boxes} = 2, ptr_{images} = 1, width\_taken = 6, width\_all = 6, count = 1$

4. Картина под указателем ptr_{images} по длине не вмещается в коробку ptr_{boxes}. Следовательно, нужно решить, брать ли новую коробку и сдвинуть ptr_{images}. Брать коробку не нужно, так как $width\_taken > width_{box}$.

   $ptr_{boxes} = 2, ptr_{images} = 2, width\_taken = 6, width\_all = 6, count = 1$

5. Картина под указателем ptr_{images} по длине не вмещается в коробку ptr_{boxes}. Следовательно, нужно решить, брать ли новую коробку и сдвинуть ptr_{images}. Брать коробку не нужно, так как $width\_taken > width_{box}$.

   $ptr_{boxes} = 2, ptr_{images} = 3, width\_taken = 6, width\_all = 6, count = 1$

6. Картина под указателем ptr_{images} по длине вмещается в коробку ptr_{boxes}. Максимальная ширина обновляется до 10. Передвигаем указатель коробок.

   $ptr_{boxes} = 3, ptr_{images} = 3, width\_taken = 6, width\_all = 10, count = 1$

7. Картина под указателем ptr_{images} по длине вмещается в коробку ptr_{boxes}. Максимальная ширина не обновляется. Передвигаем указатель коробок.

   $ptr_{boxes} = 4, ptr_{images} = 3, width\_taken = 6, width\_all = 10, count = 1$

8. Картина под указателем ptr_{images} по длине не вмещается в коробку ptr_{boxes}. Следовательно, нужно решить, брать ли новую коробку и сдвинуть ptr_{images}. Брать коробку нужно, так как $width\_taken < width_{box}, width\_all > width_{box}$.

   $ptr_{boxes} = 4, ptr_{images} = 4, width\_taken = 10, width\_all = 10, count = 2$

9. Картина под указателем ptr_{images} по длине вмещается в коробку ptr_{boxes}. Максимальная ширина не обновляется. Передвигаем указатель коробок.

   $ptr_{boxes} = 5, ptr_{images} = 4, width\_taken = 10, width\_all = 10, count = 2$

10. ptr_{boxes} указывает за пределы массива. Следовательно, нужно решить, брать ли новую коробку и сдвинуть ptr_{images}. Брать коробку не нужно, так как $width\_taken > width_{box}$.

    $ptr_{boxes} = 5, ptr_{images} = 5, width\_taken = 10, width\_all = 10, count = 2$

Таким образом, ответ: 2.

Корректность

На самом деле корректность можно доказать и более строго.
На каждом шаге мы кладём нашу картину либо в какую-то из коробок, которые уже взяты, либо в подходящую по ширине коробку с максимальной длиной.
Давайте докажем это.

1. Если мы кладём картину в непустую коробку, мы не увеличиваем ответ, следовательно, нам всегда выгодно класть картины в еще не занятые коробки.

2. Почему же нам выгодно взять подходящую по ширине коробку с максимальной длиной?
   Пусть мы в нашем решении для картины (x, y) возьмём коробку (a, b), а наиболее эффективное — (c, d), при этом мы точно знаем, что $b \geq d$.
   Существует некоторый набор картин $[id_1, \ldots, id_n]$, таких, что $width_{id_1} < \cdots < width_{id_n}$, которые в эффективном решении лежат в коробке (c, d), при этом $id_1 = (x, y)$.
   Но тогда они также поместятся в коробку (a, b), так как
   $a > x \geq width_{id_1} \geq \cdots \geq width_{id_n}$, а $b > d > y \geq length_{id_1} \geq \cdots \geq length_{id_n}$.
   А значит, наше решение не хуже.

Время работы и память

Предобработка массива будет работать за $O(n \log n)$, так как мы сначала используем линейный проход, а затем сортируем массив.
Само решение будет работать за $O(n)$, если использовать два указателя, или за $O(n \log n)$, если использовать бинпоиск.
Следовательно, суммарно решение работает за $O(n \log n)$.
Дополнительная память требуется только на хранение вспомогательных переменных, а следовательно, память — $O(1)$.

Задача «Галерея искусств» для контеста QA

Первая ошибка

```
for i := 0; i < n; i++ {
    ar := ReadInts(2)
    boxes = append(boxes, [2]int{ar[0], ar[1]})
    boxes = append(boxes, [2]int{ar[1], ar[0]})
}
```

```
for (int i = 0; i < n; i++) {
    int[] ar = readInts(scanner, 2);
    boxes.add(new int[]{ar[0], ar[1]});
    boxes.add(new int[]{ar[1], ar[0]});
}
```

```
for _ in range(n):
    a, b = map(int, input().split())
    boxes.append((a, b))
    boxes.append((b, a))
```

```
for (int i = 0; i < n; i++)
{
    var ar = ReadInts(2);
    boxes.Add((ar[0], ar[1]));
    boxes.Add((ar[1], ar[0]));
}
```

Пример теста

1 коробка: (2, 5)
2 картины: (2, 5), (5, 2)
Правильный ответ: 1.
Ответ решения: 2.

Ход размышлений

В задаче требовалось проверить, подходит ли картина под коробку. В легенде задачи подходящей считалась картина, в которой выполнялось следующее условие на размер: «минимальная и максимальная стороны картины должны быть меньше либо равны минимальной и максимальной сторонам коробки соответственно».
Следовательно, участникам нужно было придумать тест, в котором чётко проверяется именно это условие.
Ошибку также можно было вывести из кода. Неверный код, который был дан участникам, сравнивал коробки иным способом. Значит, либо автор решения придумал другое сравнение, либо ошибся — это нужно проверить.

Вторая ошибка

```
pointWithBestY := boxes[0]
```

```
int[] pointWithBestY = boxes.get(0);
```

```
point_with_best_y = boxes[0]
```

```
var pointWithBestY = boxes[0];
```

Пример теста

1 коробка: (2, 4)
1 картина: (4, 4)
Правильный ответ: -1.
Ответ решения: 1.

Как это можно найти

Крайние тесты очень важны, никогда не стоит про них забывать.

В задаче важными крайними тестами будут:

1. 1 коробка и 1 картина, ответ: можно.

2. 1 коробка и 1 картина, ответ: нельзя.

Второй тест и ломал решение.

Важное уточнение

Нужно обратить внимание, что вторая ошибка появляется только после исправления первой.
Очень важно, чтобы при каждом изменении разработчиком кода тестирование было проделано полностью еще раз. Эта ошибка и проверяла то, что вы не забыли протестировать все тесты с нуля.