# Dark room

## Краткое условие без легенды

Дана комната размером $n$ в длину и $m$ в ширину. Нужно найти минимальное количество фонариков, необходимых для полного освещения этой комнаты, а также их координаты и направление. В одной координате не может быть больше одного фонарика.

## Решение

Можно догадаться, что для любой комнаты размером $n \cdot m$ хватит одного или двух фонариков для полного освещения. Давайте разберём каждый случай отдельно.

### Случай, когда нужен один фонарик

- Если $n = 1$ (одна строка) $\rightarrow$ ставим фонарик в $(1, 1)$, направляем вправо (`R`).
- Если $m = 1$ (один столбец) $\rightarrow$ ставим фонарик в $(1, 1)$, направляем вниз (`D`).

### Случай, когда нужны два фонарика

- Если $n > m$:
  - первый фонарик ставим в $(1, 1)$, направляем вниз (`D`);
  - второй фонарик ставим в $(n,\ m)$, направляем вверх (`U`).
- Если $m \geqslant n$:
  - первый фонарик ставим в $(1, 1)$, направляем вправо (`R`);
  - второй фонарик ставим в $(n, m)$, направляем влево (`L`).

Существует несколько возможных решений, это — одно из них.

## Время работы

Решение работает за $O(1)$, так как выполняется фиксированное количество сравнений, занимающее константное время.

# Обратный отсчёт

## Краткое условие без легенды

Дано натуральное число $n$ ($1 \le n \le 10^9$). Нужно найти количество цифр, которые понадобятся, чтобы представить каждое целое число от 0 до $n$ включительно.

## Разбор

### Решение, проходящее первые две группы тестов ($1 \le n \le 10^3$)

Чтобы пройти первые две группы тестов, достаточно перебрать каждое число из диапазона и для каждой цифры найти, какое максимальное количество этой цифры может понадобиться в числе.

Будем поддерживать массив из 10 чисел $count$, где $count_i$ — это минимальное необходимое количество цифры $i$ для ответа. Изначально все $count_i$ равны 0.

Теперь мы хотим понять, достаточно ли всех цифр, представленных в массиве $count$, чтобы представить какое-то число $x$, и если нет, то сколько их нужно. Ответ на задачу — это сумма $count_i$ после перебирания всех таких $x$ от 0 до $n$ включительно.

Посчитаем аналогичный массив $cx$, в котором $cx_i$ обозначает, сколько раз цифра $i$ встречается в числе $x$. Сравним для каждой цифры уже известное значение $count_i$ с необходимым значением $cx_i$. Если $cx_i > count_i$, значит, нам нужно взять больше цифр $i$, чтобы иметь возможность представить любое число от 0 до $n$ (в том числе $x$). А именно: хотя бы $cx_i$. В таком случае обновим $count_i$, присвоив туда значение $cx_i$.

Код на языке Go:

```go
func solve(n int) int {
    count := make([]int, 10)
    for x := 0; x <= n; x++ {
        xStr := strconv.Itoa(x)
        cx := make([]int, 10)
        for i := 0; i < len(xStr); i++ {
            cx[int(xStr[i]-'0')]++
        }
        for i := 0; i < 10; i++ {
            if cx[i] > count[i] {
                count[i] = cx[i]
            }
        }
    }
    sum := 0
    for i := 0; i < 10; i++ {
        sum += count[i]
    }
    return sum
}
```

У этого алгоритма высокая сложность ($O(n \cdot log_{10} n)$), из-за чего он проходит только для $n \leq 10^6$ (или примерно тысячу раз для $n \leq 10^3$).

## Решение, проходящее все группы тестов ($1 \leq n \leq 10^9$)

Можно заметить, что в общем случае ответ возрастает, когда в обратном отсчёте встречается либо очередное число, состоящее из одной цифры, повторённой несколько раз подряд, либо очередное круглое число. Действительно, для представления числа 1 достаточно только единицы, для числа 11 их уже нужно две, для числа 111 — три и так далее. Аналогично для двоек, троек, четвёрок и так далее. Единственное исключение — цифра 0. Её количество увеличивается на числах $0, 100, 1000, 10\,000$ и так далее, но не у числа 10.

Таким образом, если аккуратно перебрать числа, которые не превышают число $n$, но при этом увеличивают ответ, их количество и будет решением задачи.

Один из вариантов такого перебора на языке Go:

```go
func solve(n int) int {
    ans := 1 // всегда учитываем 0
    for x := 1; x <= n; x = x*10 + 1 { // x = 1, 11, 111, ...
        // перебираем множитель для x,
        // т.е. числа 111..., 222..., 333... и так далее
        for num := 1; num <= 9 && x*num <= n; num++ {
            ans++
        }
        if x > 1 && x*9 < n { // случай для круглых чисел, кроме 10
            ans++
        }
    }
    return ans
}
```

Сложность такого решения — $O(log_{10}n)$.

## Вариант с предподсчётом

Поскольку число $n$ достаточно маленькое в этой задаче, можно найти все такие числа, которые увеличат ответ, если будут меньше переданного $n$. В пределах ограничений их немного, вот все подходящие:

$0, 1, 2, 3, 4, 5, 6, 7, 8, 9,$

$11, 22, 33, 44, 55, 66, 77, 88, 99, 100,$

$111, 222, 333, 444, 555, 666, 777, 888, 999, 1000,$

$1111, 2222, 3333, 4444, 5555, 6666, 7777, 8888, 9999, 10000,$

$11111, 22222, 33333, 44444, 55555, 66666, 77777, 88888, 99999, 100000,$

$111111, 222222, 333333, 444444, 555555, 666666, 777777, 888888, 999999, 1000000,$

$1111111, 2222222, 3333333, 4444444, 5555555, 6666666, 7777777, 8888888, 9999999, 10000000,$

$11111111, 22222222, 33333333, 44444444, 55555555, 66666666, 77777777, 88888888, 99999999, 100000000,$

$111111111, 222222222, 333333333, 444444444, 555555555, 666666666, 777777777, 888888888, 999999999, 1000000000$

Получив очередное число $n$, мы можем пройтись по этому числовому ряду и найти, сколько чисел не превышают $n$ — это и будет ответом на задачу:

```
var numbers = []int{0, 1, 2, 3, 4, 5, /* ... числа выше */}
func solve(n int) int {
    for i := 0; i < len(numbers); i++ {
        if numbers[i] > n {
            return i
        }
    }
    return len(numbers)
}
```

Получим такую же асимптотику $O(log_{10}n)$, только здесь меньше шансов ошибиться с получением очередного числа.

Можно пойти дальше и применить бинарный поиск на этом массиве, и в итоге получить константную асимптотику, так как размер массива `numbers` не меняется:

```
func solve(n int) int {
    idx := sort.SearchInts(numbers, n)
    if idx >= len(numbers) || numbers[idx] != n {
        return idx
    }
    return idx + 1
}
```

`sort.SearchInts` — это функция бинарного поиска из стандартной библиотеки Go, которая возвращает индекс первого элемента массива, который больше либо равен переданному числу. В других языках программирования реализация бинарного поиска может отличаться, но суть та же – нужно найти число, которое *строго больше* полученного $n$.

# Валидация

## Краткое условие без легенды

В этой задаче необходимо проверить корректность строки, которая была сгенерирована из списка товаров.

## Решение

Сначала разобьём строку по запятым. Сделать это можно функцией `strings.Split` в Go, методами строк `split` и `Split` в Python и C# соответственно.
После этого разобьём каждую часть строки по символу двоеточие, «`:`». Если какая-то из частей разбилась не на две части, то строка некорректна.

Осталось проверить:

- что все товары из строки находятся в списке товаров (с корректной ценой);
- что у каждого товара в строке уникальная цена;
- что нет цены из списка товаров, которая не встречается в строке.

Можно сделать это отдельно, но ниже мы приведём короткое решение со словарями и множествами.

Основная идея в том, что по списку товаров мы создадим словарь `dict[price` $\to$ `names]`, в котором ключами будут цены, а значениями — множества названий товаров с такой ценой. Составив такой словарь, пройдёмся по списку товаров в строке и рассмотрим очередной товар `name:price`.

- Если `dict[price]` пустой (то есть такого ключа в словаре нет), то строка некорректна — либо пары name:price в списке не существует, либо какой-то товар с ценой `price` уже был рассмотрен.

- Если `name` не входит в `dict[price]`, то строка некорректна, так как пары `name:price` в списке не существует.
- Иначе удалим из словаря ключ `price` (и все значения по этому ключу), чтобы больше не было возможности использовать цену `price`.

Если после прохода по строке словарь `dict` пустой, то строка корректна, другими словами, в строке присутствуют все цены из списка.

Обратите внимание, что нам необязательно переводить цену товара из строки в число.

## Время работы

Решение работает за линейное время относительно длины строки и длины списка товаров. Если вместо словарей и множеств использовать списки, то решение будет работать за квадратичное время.

# Похожие строки

## Краткое условие без легенды

Нам дано $t$ тестовых наборов, каждый из которых содержит $n$ строк. Строки состоят из строчных латинских букв.
Необходимо определить, сколько пар строк в каждом наборе являются похожими. Две строки считаются похожими, если у них совпадают все буквы на чётных позициях или все буквы на нечётных.

## Пример

$T = 1$
$N = 3$
`anca`
`abc`
`bac`

В наборе три строки: `anca`, `acb`, `bac`. Строки `anca` и `abc` имеют одинаковые буквы на чётных позициях (`a` и `c`).
Вердикт по данному набору — одна пара похожих строк.

## Разбор

Идея решения заключается в поиске аналогичных строк по чётным или нечётным позициям. Худший случай решения — перебор всех строк и поиск аналогичных.

### Основные функции решения задачи

Итак, для каждого тестового набора сформируем три хеш-таблицы:

- `evenMap` — для чётных позиций;
- `evenMap` — для нечётных позиций;

- `eqMap` — хеш-таблица для подсчёта асболютно совпадающих строк.

```
evenMap := make(map[string]int)
oddMap := make(map[string]int)
eqMap := make(map[string]int)
```

Объявим общий счётчик похожих пар:

```
result := 0
```

И объявим анонимную функцию для определения строк:

```
getEvenOdd := func(val string) ([]byte, []byte) {
  even := make([]byte, 0, len(val)/2)
  odd := make([]byte, 0, len(val)/2)
  for i := 0; i < len(val); i++ {
    if i%2 == 0 {
      even = append(even, val[i])
    } else {
      odd = append(odd, val[i])
    }
  }
  return even, odd
}
```

Для каждой строки определим срез байтов чётных и нечётных позиций символов — временная сложность $O(s)$, где $s$ — длина строки.

После получения байтовых срезов увеличим счётчики во всех хеш-таблицах, а также сразу сформируем конечный результат:

```
for i := 0; i < countN; i++ {
  readString := ReadSrt()

  if len(readString) == 1 {
      result += evenMap[readString]
      evenMap[readString]++
      continue
  }

```

```
        even, odd := getEvenOdd(readString)

        result += evenMap[string(even)]
        result += oddMap[string(odd)]
        result -= eqMap[readString]

        eqMap[readString]++
        evenMap[string(even)]++
        oddMap[string(odd)]++
    }
```

Стоит обратить внимание на заполнение таблицы `eqMap` и на то, как её значения влияют на конечный результат в тестовом наборе.

## Время работы

Обновление результатов хеш-таблиц происходит за $O(1)$. Поиск чётных и нечётных позиций за $O(s)$. Таким образом, решение имеет сложность $O(\sum |s|)$, где $\sum |s|$ — сумма длин всех строк.

# Коробки. Коробки. Коробки

## Краткое условие без легенды

Дана матрица из `ascii`-символов с именованными прямоугольниками внутри неё. Необходимо построить и вывести дерево вложенности прямоугольников в виде `json`-структуры.

## Разбор

### Решение, проходящее первую группу тестов

Для начала найдём и запишем в массив все прямоугольники, расположенные в матрице.

Последовательно, слева направо, сверху вниз, найдём все символы в матрице, из которых могут состоять имена коробок (латинские буквы и цифры). Обозначим позицию найденного символа как $(i, j)$. Тогда, так как по условию, имя коробки всегда стоит в левом верхнем углу прямоугольника, мы можем быть уверены, что символ в позиции $(i - 1, j - 1)$ является его левым верхним углом. А дальше от этой позиции мы легко сможем найти длины сторон этого прямоугольника и, соответственно, все его углы.

Теперь, имея полный список прямоугольников с их углами, найдём для каждой коробки *родителя* (коробку, в которой она находится) или же определим, что его нет. Заметим, что коробка $A$ вложена в коробку $B$, только если все углы $A$ лежат внутри $B$. Тогда *родителем* коробки $A$ будет такая коробка $B$, для которой $A$ является вложенной коробкой, и при этом площадь $B$ минимальна.

Теперь, имея список из $k$ коробок и найдя линейным списком для каждой *родителя*, мы построим дерево вложенности, где у каждого узла дерева мы знаем имя и позиции углов прямоугольника (отсюда, соответственно, и площадь), которое и необходимо вывести в ответ.

Итак, сложность этого частичного решения $O(k^2)$, где в худшем случае (замощение поля невложенными коробками) $k = (N/3) \cdot (M/3)$, следовательно, и сложность получается $O((N \cdot M)^2)$ времени и $O(N \cdot M)$ памяти.

### Полное решение

Для полного решения будем таким же образом искать коробки, но при этом искать очередной символ имени коробки будем не на всей входной матрице, а только внутри уже известной заранее родительской коробки.

Другими словами, после того как мы нашли первую коробку $A$ (слева направо, сверху вниз), мы сразу запустим такой же поиск, но только по символам внутри $A$. Тогда для найденных внутри неё коробок $A$ будет родителем. После того как мы нашли очередную коробку, пометим её как уже найденную (можно реализовать с помощью `bool`-матрицы или `set`'ов), чтобы избежать неоднозначности в виде того, что у $A$ будет несколько родителей.

Данный алгоритм призывает использовать рекурсивную функцию, которая будет принимать на вход лишь коробку-родителя, чтобы установить границы поиска для функции и сразу установить родителя для всех найденных коробок.

Полное решение имеет сложность $O(N \cdot M)$ времени и памяти.

# Content delivery

## Краткое условие без легенды

Есть $n$ серверов с пропускной способностью $throughput_i$ (память/секунду) и $m$ изображений размером $weight_j$ (единицы памяти).

Время доставки изображения $j$ сервером $i$ рассчитывается как:

$$time_{ij} = weight_j / throughput_i$$

Требуется распределить изображения по серверам так, чтобы минимизировать разницу между максимальным и минимальным временем доставки среди всех изображений.

## Пример теста

Нам даны два сервера с пропускной способностью [3, 5], а также пять изображений, которые занимают [12, 13, 14, 15, 16] единиц памяти.

Если изображение 1 доставляется первым сервером, а 2, 3, 4, 5 — вторым, то время доставки будет соответственно:

$$[12/3, 13/5, 14/5, 15/5, 16/5] = [4, 3, 3, 3, 4]$$

Разность будет равна 1.

Можно показать и другое возможное решение: если изображения 1, 2, 3, 4 доставляются первым сервером, а 5 — вторым, то время доставки будет соответственно:

$$[12/3, 13/3, 14/3, 15/3, 16/5] = [4, 4, 4, 5, 4]$$

Разность также будет равна 1.

## Разбор

Давайте пройдёмся по всем серверам и изображениям, вычисляя время доставки каждого изображения на каждом сервере:

$$time_{ij} = weight_j / throughput_i$$

Все пары (время доставки, номер изображения, номер сервера) сохраняются в массив `delivery`.

```go
for server := 0; server < serversCount; server++ {
  for image := 0; image < imagesCount; image++ {
    time := (imageWeight[image] + serverThroughput[server] -
    1) / serverThroughput[server]
    delivery = append(delivery, Delivery{
      Time:   time,
      Image:  image,
      Server: server,
    })
  }
}
```

Теперь давайте массив `delivery` отсортируем от самого быстрого до самого медленного варианта доставки.

```go
sort.Slice(delivery, func(i, j int) bool {
  return delivery[i].Time < delivery[j].Time
})
```

И давайте используем метод двух указателей (*sliding window*), чтобы найти наименьший отрезок в отсортированном массиве `delivery`, покрывающий все изображения.

Создадим вспомогательные структуры:

- `imageCounter[j]` — сколько раз изображение $j$ встречается в текущем диапазоне.
- `processedImagesCount` — сколько разных изображений покрыто в текущем диапазоне.

```go
imageCounter := make([]int, imagesCount)
processedImagesCount := 0
```

Функции для добавления и удаления изображений в текущий диапазон:

- inc(image) — увеличивает счётчик изображения, отмечая его как обработанное.
- dec(image) — уменьшает счётчик, и, если изображение больше не покрывается, оно удаляется из диапазона.

```go
inc := func(image int) {
  if imageCounter[image] == 0 {
    processedImagesCount++
  }
  imageCounter[image]++
}

dec := func(image int) {
  imageCounter[image]--
  if imageCounter[image] == 0 {
    processedImagesCount--
  }
}
```

Запускаем метод двух указателей:

- right движется вперёд, добавляя варианты доставки.
- Если все изображения покрыты (processedImagesCount == imagesCount), пытаемся минимизировать разницу между delivery[right].Time и delivery[left].Time.
- Если нашли лучший вариант, обновляем minDeliveryGap.
- Затем двигаем left, убирая вариант доставки.

```go
minDeliveryGap := math.MaxInt32
var rangeStart, rangeEnd int

for left, right := 0, 0; right < len(delivery); right++ {
  inc(delivery[right].Image)
```

```
  for left <= right && processedImagesCount == imagesCount {
    deliveryGap := delivery[right].Time - delivery[left].Time
    if minDeliveryGap > deliveryGap {
      minDeliveryGap = deliveryGap
      rangeStart, rangeEnd = left, right
    }
    dec(delivery[left].Image)
    left++
  }
}
```

После нахождения оптимального диапазона, каждое изображение `imageServer[i]` закрепляется за сервером, который его обработает в найденном диапазоне.

```
for i := rangeStart; i <= rangeEnd; i++ {
  imageServer[delivery[i].Image] = delivery[i].Server
}
```

## Время работы и память

Этот алгоритм:

1. Перебирает все возможные варианты доставки изображений на серверах $O(n \cdot m)$.
2. Сортирует их по времени $O(n \cdot m \cdot log(n \cdot m))$.
3. Находит минимальный диапазон методом двух указателей $O(n \cdot m)$.

Итоговая сложность $O(n \cdot m \cdot log(n \cdot m))$ по времени и $O(n \cdot m)$ памяти для хранения массива `delivery`.