Сортировки. Хеш-таблица

Семинар

Сортировка шелла

Shell sort

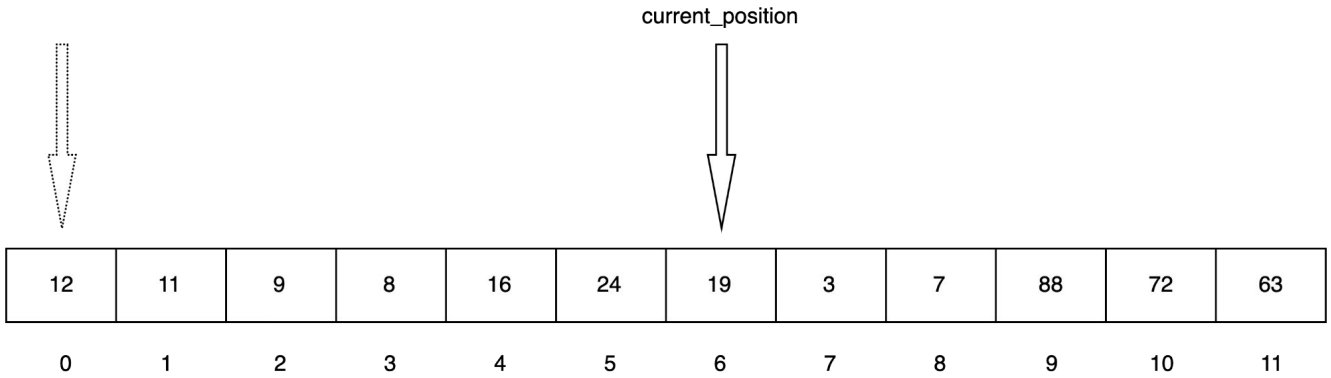
- Усовершенствованный алгоритм сортировки вставками (?)
- Сравниваем не рядом стоящие элементы, а элементы, которые располагаются на определенном удалении (шаге) друг от друга.
- На каждой итерации шаг уменьшается в два раза, пока не станет равным единицы
- На последнем проходе, когда gap = 1 сортировка вырождается в сортировку вставками.

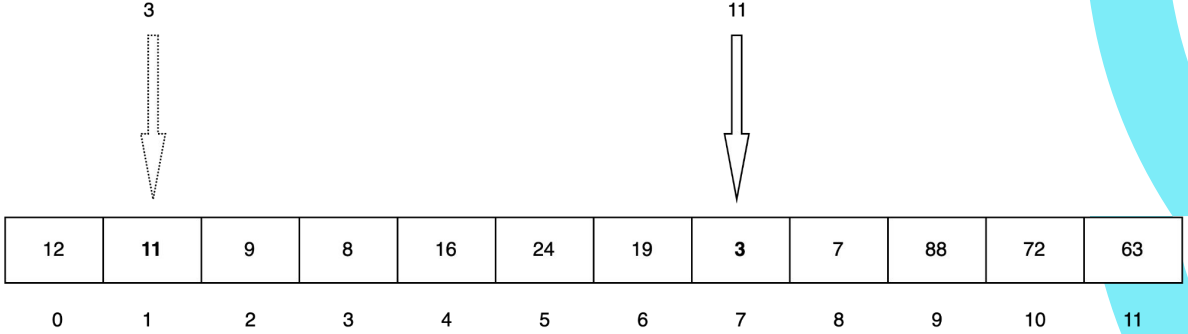
Описание

- Вычисляем первое значение шага **gap = len(array)/2**
- Находим элемент с индексом **gap** (**current_position**)
- Сравниваем этот элемент с элементом под индексом **current_position - gap**
- В случае, если элемент под получившимся индексом больше чем current_position меняем их местами
- Если нам понадобилась замена элементов, то проверяем элемент который теперь находится под индексом **current_position - gap** с элементом под индексом **current_position - 2gap**, если, конечно такой индекс существует
- Если перестановка не понадобилась двигаем **current_position** на **+1**
- После прохода уменьшаем **gap** вдвое

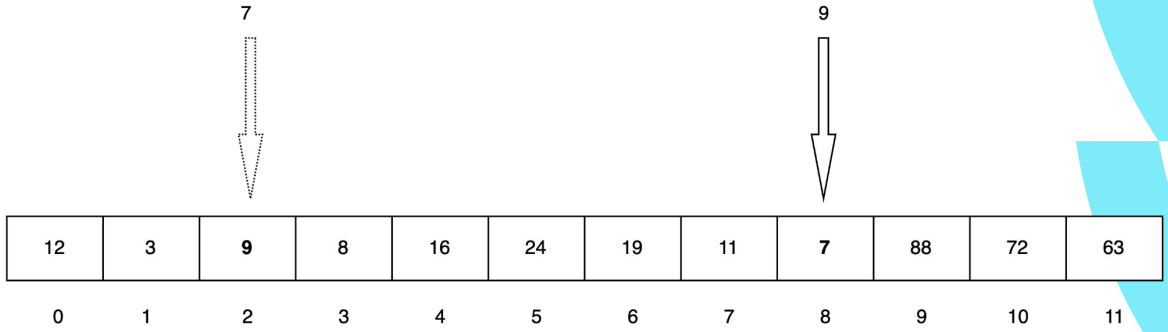
12	11	9	8	16	24	19	3	7	88	72	63
0	1	2	3	4	5	6	7	8	9	10	11

- Дан массив из 12 элементов
- Вычисляем шаг, разделив длину массив пополам **gap = len(array)/2**
- Находим элемент с индексом **gap**, назовем его **current_position** и элемент с индексом **current_position - gap**
- Сравниваем два элемента arr[current_position] и arr[current_position - gap]
- При необходимости меняем их местами
- Если arr[current_position - gap] < arr[current_position], то просто инкрементируем **current_position**

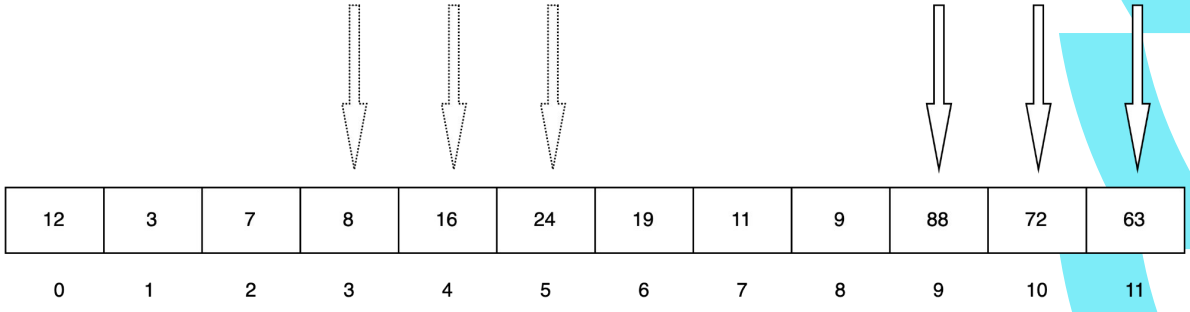




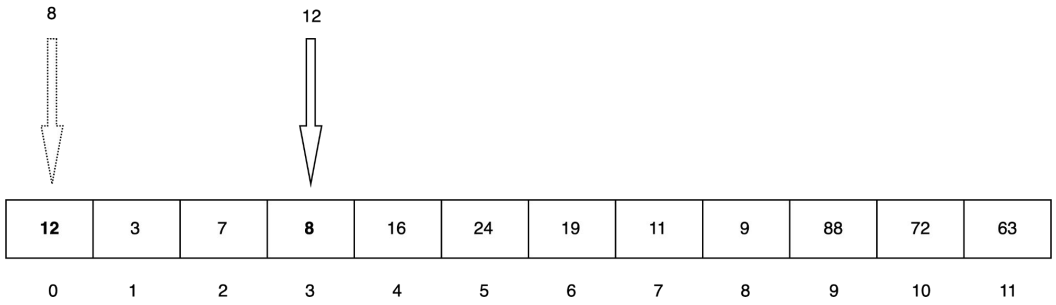
- В случае, когда понадобилась перестановка, продолжаем поиск элемент слева, чтобы сравнить arr[current_position - gap] и arr[current_position - 2gap]
- При необходимости идем влево на gap, пока не выйдем за пределы массива
- В данном случае это 1 - **gap**, получаем индекс -5 - такого элемента нет
- Просто инкрементируем **current_position**

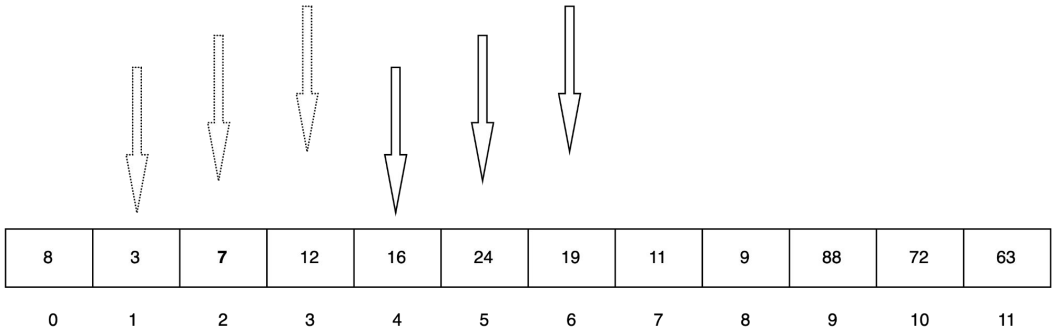


Продолжаем итерироваться с текущем шагом пока **current_position** не дойдет до конца массива

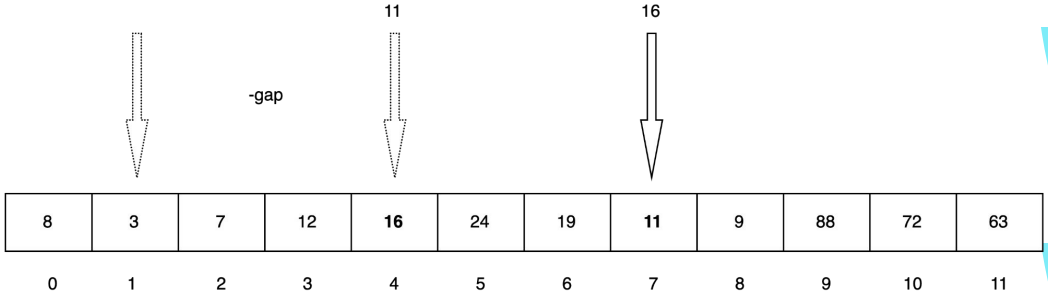


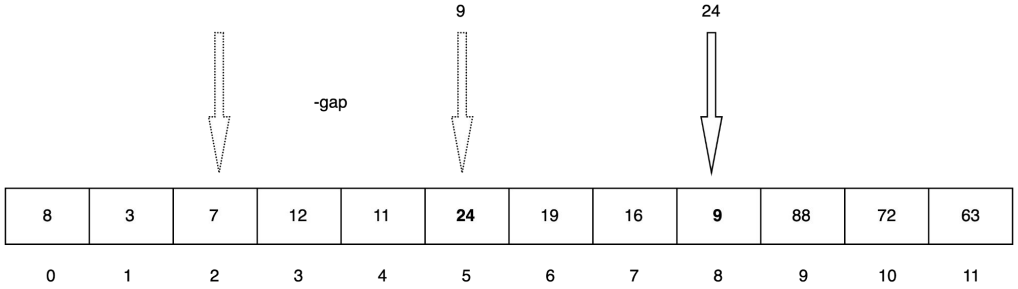
- Уменьшаем шаг в два раза
- Находим соответствующий элемент
- Продолжаем алгоритм

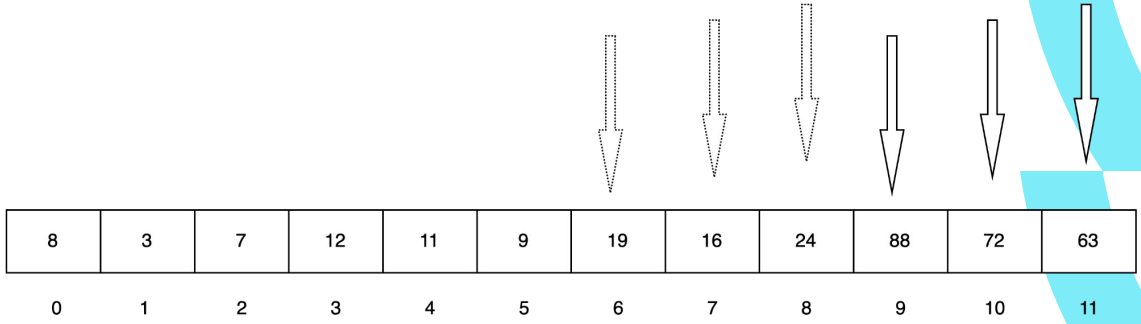




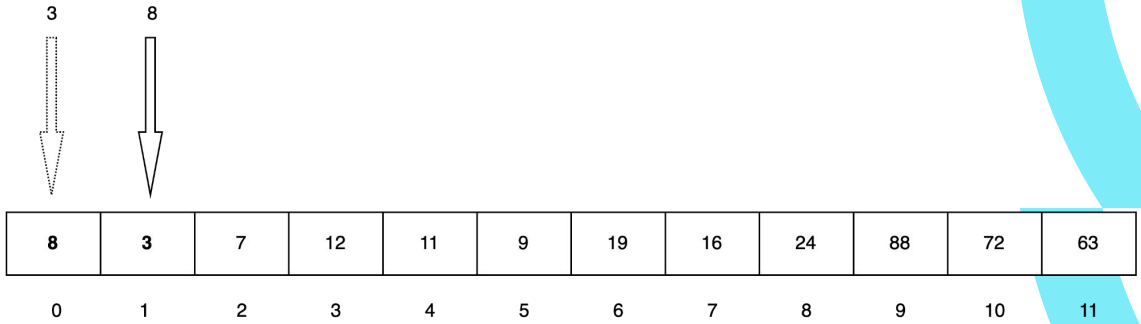
- Нам понадобилась перестановка
- arr[m_gap], arr[m_gap - gap] = arr[m_gap - gap], arr[m_gap]
- И теперь необходимо проверить следующий элемент под индексом **current_position - 2gap**
- Инкрементируем **current_position**

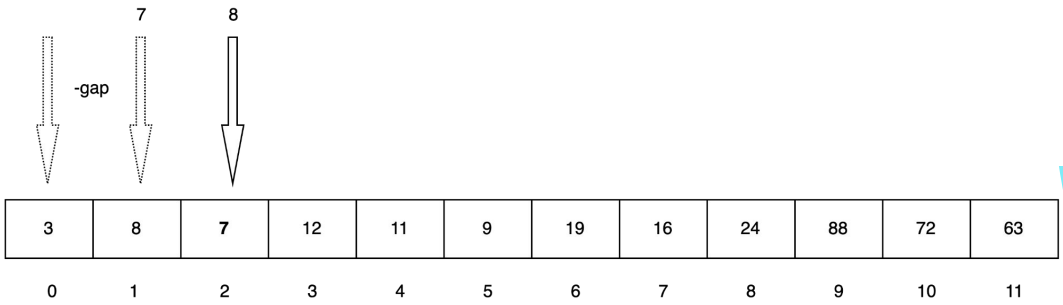


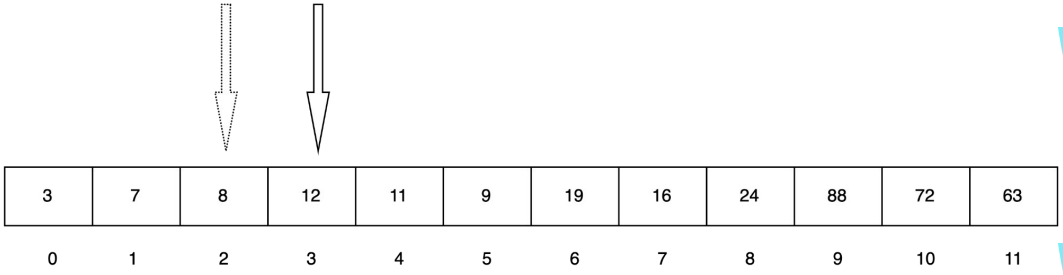


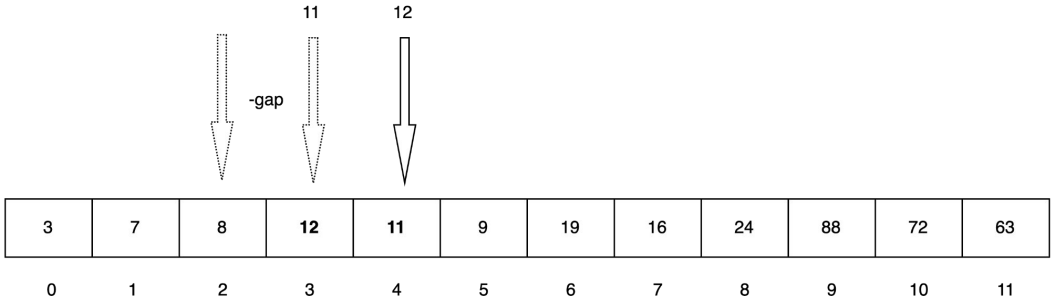


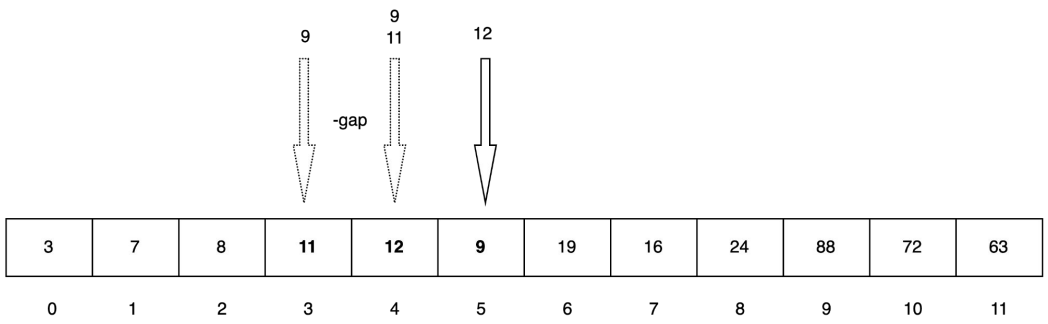
Дойдя до конца массива, снова уменьшаем **gap** в два раза

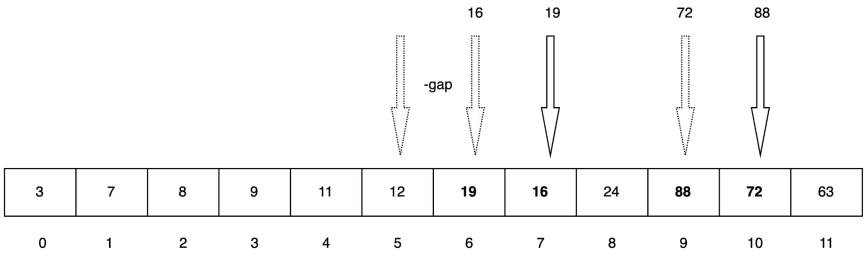


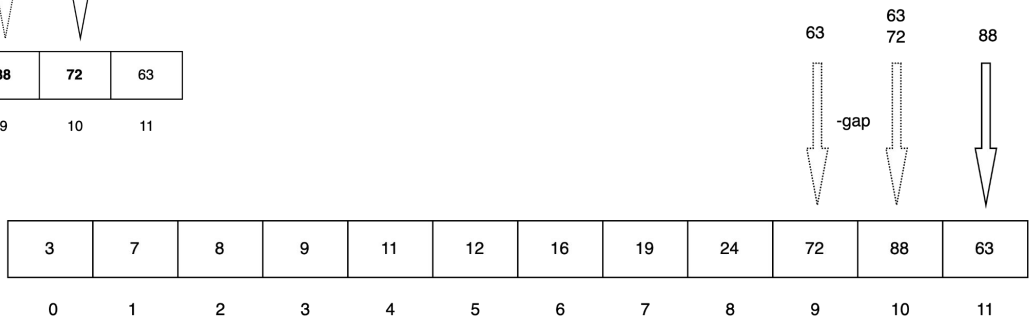

7
8
-gap
3
8
7
12
11
9
19
16
24
88
72
63
0
1
2
3
4
5
6
7
8
9
10
11

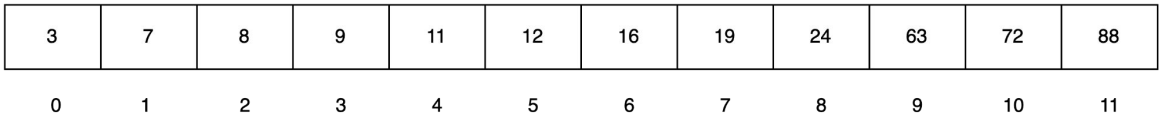

3
7
8
12
11
9
19
16
24
88
72
63
0
1
2
3
4
5
6
7
8
9
10
11


11
12
-gap
3
7
8
12
11
9
19
16
24
88
72
63
0
1
2
3
4
5
6
7
8
9
10
11


9
9
11
12
-gap
3 7 8 11 12 9 19 16 24 88 72 63
0 1 2 3 4 5 6 7 8 9 10 11


16
19
72
88
-gap
3 7 8 9 11 12 19 16 24 88 72 63
0 1 2 3 4 5 6 7 8 9 10 11


63
63
72
88
-gap
3 7 8 9 11 12 16 19 24 72 88 63
0 1 2 3 4 5 6 7 8 9 10 11

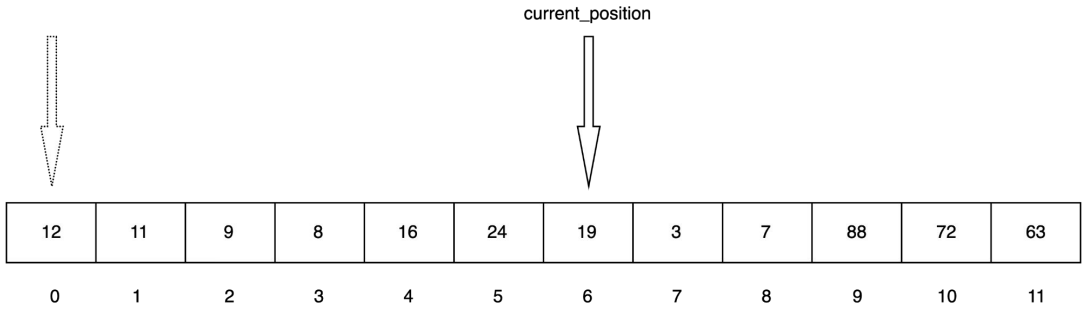

3 7 8 9 11 12 16 19 24 63 72 88
0 1 2 3 4 5 6 7 8 9 10 11

Пишем код

```
function shell_sort(arr) {
    n = len(arr)
    gap = len(arr) / 2
    // определяем внешний цикл
    // цикл должен быть определен значением gap
    return arr
}
```

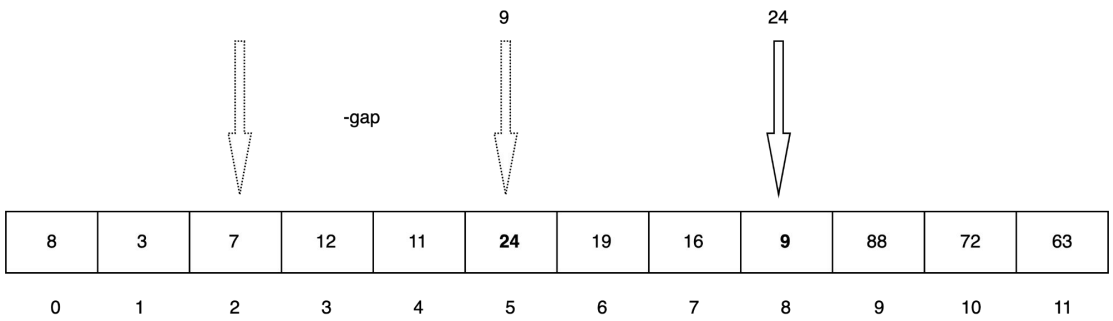
Пишем код

```
function shell_sort(arr) {
    n = len(arr)
    gap = len(arr) / 2
    while gap > 0 {
        // current_position на первой итерации равен gap
        // элемент под индексом current_position сравниваем с элементом
        // под индексом current_position - gap
        // при необходимости меняем их местами
        gap = gap / 2
    }
    return arr
}
```



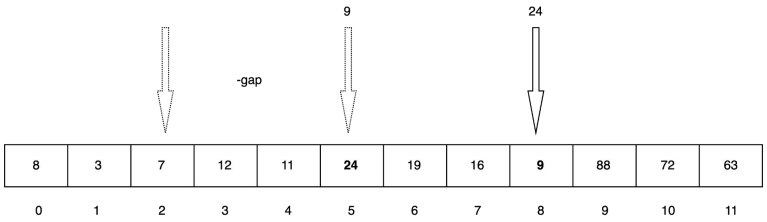
Пишем код

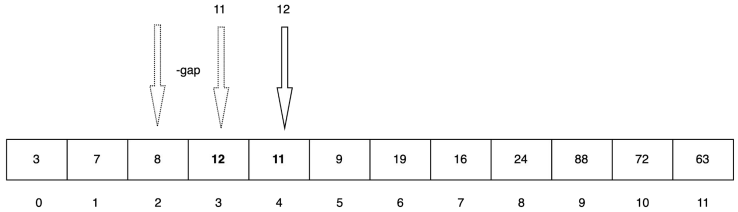
```
function shell_sort(arr) {
    n = len(arr)
    gap = len(arr) / 2
    while gap > 0 {
        // в цикле от current_position до n сравниваем элемент под индексом
        // current_position и current_position - gap если понадобилась перестановка,
        // продолжаем вычислять current_position - gap * m
        //  m - кол-во шагов влево, которые мы делаем при необходимости
        gap = gap / 2
    }
    return arr
}
```



Пишем код

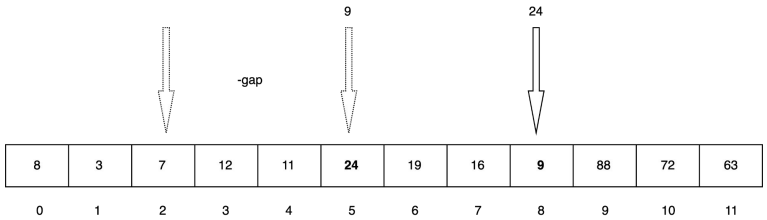
```
function shell_sort(arr) {
    n = len(arr)
    gap = len(arr) / 2
    while gap > 0 {
        for current_position = gap; current_position < n; current_position++ {
            // запоминаем current_position
            // в цикле проверяем значения под индексом current_position - gap
            // если нужна перестановка - проверяем значение по индексом
            // current_position - gap - gap ...
            // то есть декрементируем переменную в которой сохранили current_position
        }
        gap = gap / 2
    }
    return arr
}
```

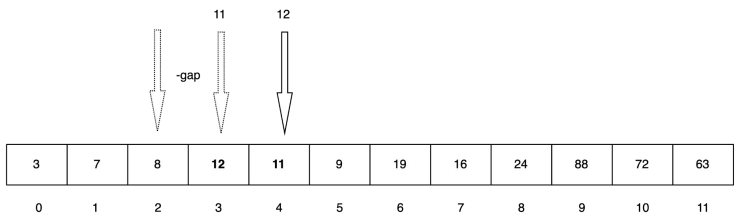




Пишем код

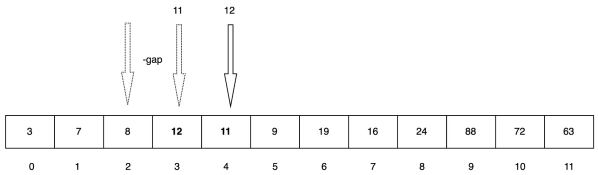
```
function shell_sort(arr) {
    n = len(arr)
    gap = len(arr) / 2
    while gap > 0 {
        for current_position = gap; current_position < n; current_position++ {
            m_gap = current_position
            // цикл до тех пор пока мы не вышли за границы массива слева и пока нам
            // нужна перестановка между элементами под индексами m_gap и m_gap - gap
            // делаем swap(arr[m_gap], arr[m_gap - gap])
            // декрементируем m_gap на gap
        }
        gap = gap / 2
    }
    return arr
}
```

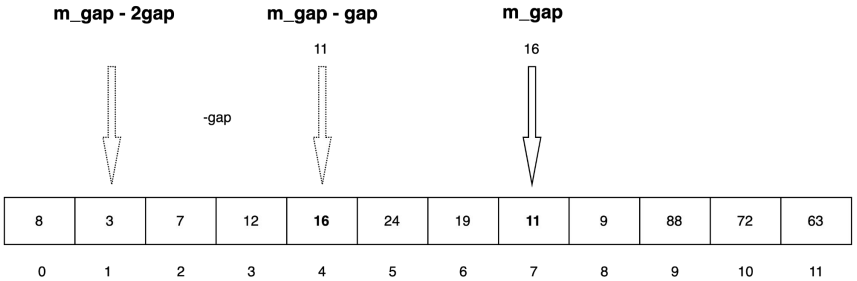




Пишем код

```
function shell_sort(arr) {
    n = len(arr)
    gap = len(arr) / 2
    while gap > 0 {
        for current_position = gap; current_position < n; current_position++ {
            m_gap = current_position
            while m_gap >= gap and arr[m_gap] < arr[m_gap - gap] {
                arr[m_gap], arr[m_gap - gap] = arr[m_gap - gap], arr[m_gap]
                m_gap -= gap
            }
        }
        gap = gap / 2
    }
    return arr
}
```





Варианты выбора шага

Хиббард предложил вычислять шаги в сортировке Шелла по формуле: **gap = 2^k - 1**, где k - целое число, начиная с максимального значения и уменьшая его на каждой итерации. Таким образом, на каждой итерации шаг уменьшается в два раза.

Сложность алгоритма сортировки Шелла зависит от выбранной последовательности шагов. Для последовательности шагов, вычисляемой по формуле Хиббарда, сложность алгоритма составляет примерно **O(n^(3/2))**, что делает его более эффективным по сравнению с сортировкой пузырьком или вставками, но менее эффективным по сравнению с современными алгоритмами сортировки, такими как быстрая сортировка или сортировка слиянием.

Варианты выбора шага

Пратт предложил вычислять шаги в сортировке Шелла по формуле: **gap = 2^i * 3^j**, где i и j - целые числа. На каждой итерации i и j уменьшаются на один, пока gap не станет равным 1. Сложность алгоритма сортировки Шелла, используя эту формулу, составляет примерно **O(n^(4/3))**, что делает его более эффективным, чем сортировка Хиббарда, но все равно менее эффективным, чем современные алгоритмы сортировки.

Варианты выбора шага

Эмпирическая последовательность Марцина Циура:
gap ∈ {1,4,10,23,57,132,301,701,1750};
является одной из лучших для сортировки массива ёмкостью приблизительно до 4000 элементов.

Накормить животных

В небольшом зоопарке есть некоторое количество животных.
Каждое животное потребляет какой-то объем еды, выраженный в целочисленном значении. Например, еноту нужна 1-порция еды, зебре 2, пантере 3, льву 4, жирафу 8 и т.д.
Каждый день, смотритель зоопарка привозит еду такими же порциями. То есть за раз он привозит 8, 3, 9, 1, 7. Порция на 8 может накормить одно животное один раз. То есть такая порция может накормить либо енота, либо льва, либо жирафа, но не может накормить, например зебру и енота. Только кого-то одного.
Надо написать функцию, которая определит, сколько из переданных животных может накормить заданное количество еды.

Накормить животных

Пример: массив потребностей в еде животных [3, 4, 7], массив привезенной еды: [8, 1, 2] может накормить одно животное.

Массив потребностей животных [3, 8, 1, 4] и массив еды [1, 1, 2] - накормленным будет лишь одно животное.

Массив потребностей животных [1, 2, 2] и массив еды [7, 1] - накормленным будет два обитателя зоопарка

Массив потребностей животных [8, 2, 3, 2] и массив еды [1, 4, 3, 8] - накормленным будет три обитателя зоопарка

На вход подается 2 массива целых чисел. Первый массив - потребности животных, второй - количество привезенной еды. Необходимо вернуть целое число - количество накормленных зверей.

```
function feedAnimals(animals, food) {
    if len(animals) == 0 or len(food) == 0 {
        return 0
    }

    return ...
}
```

Наивная реализация

food: 8, 1
animals: 1, 8
Одно животное останется голодным

```
function feedAnimals(animals, food) {
    if len(animals) == 0 or len(food) == 0 {
        return 0
    }
    count = 0
    for i = 0 ... len(food) {
        for j = 0 ... len(animals) {
            if food[i] >= animals[j] {
                count++
                // нужно запомнить индексы
                // накормленных животных и
                // использованной еды
                // при этом порции могут
                // расходоваться не эффективно
            }
        }
    }

    return count
}
```

Реализация с сортировкой

food: 1, 4, 3, 8 -> 1, 3, 4, 8

animals: 8, 2, 3, 2 -> 2, 2, 3, 8

Реализация с сортировкой

food: 1, 4, 3, 8 -> 1, 3, 4, 8

animals: 8, 2, 3, 2 -> 2, 2, 3, 8

```
function feedAnimals(animals, food) {
    if len(animals) == 0 or len(food) == 0 {
        return 0
    }
    sort(animals) // animals.sort()
    sort(food)    // food.sort()

    return ...
}
```

Реализация с сортировкой

food: 1, 4, 3, 8 -> 1, 3, 4, 8

animals: 8, 2, 3, 2 -> 2, 2, 3, 8

```
function feedAnimals(animals, food) {
    if len(animals) == 0 or len(food) == 0 {
        return 0
    }
    sort(animals) // animals.sort()
    sort(food)    // food.sort()

    // цикл по еде
    // смотрим, может ли кого-то накормить
    // текущая порция

    return ...
}
```

Реализация с сортировкой

food: 1, 4, 3, 8 -> 1, 3, 4, 8

animals: 8, 2, 3, 2 -> 2, 2, 3, 8

```
function feedAnimals(animals, food) {
    if len(animals) == 0 or len(food) == 0 {
        return 0
    }
    sort(animals) // animals.sort()
    sort(food)    // food.sort()

    count = 0
    for f in food {
        // сопоставляем текущую порцию
        // с потребностью животного
        // под индексом count
        // если животное удалось накормить
        // инкрементируем count и на следующей
        // итерации пытаемся накормить
        // следующее животное в массиве
    }

    return count
}
```

Реализация с сортировкой

food: 1, 4, 3, 8 -> 1, 3, 4, 8

animals: 8, 2, 3, 2 -> 2, 2, 3, 8

```
function feedAnimals(animals, food) {
  if len(animals) == 0 or len(food) == 0 {
    return 0
  }
  sort(animals) // animals.sort()
  sort(food)    // food.sort()

  count = 0
  for f in food {
    // проверяем, можем ли мы
    // обратиться по индексу count
    // к массиву animals
    if len(animals) > count {
      if f >= animals[count] {
        count += 1
      }
    }

    ... // условия выхода?
   }

  return count
}
```

Реализация с сортировкой

food: 1, 4, 3, 8 -> 1, 3, 4, 8

animals: 8, 2, 3, 2 -> 2, 2, 3, 8

```
function feedAnimals(animals, food) {
    if len(animals) == 0 or len(food) == 0 {
        return 0
    }
    sort(animals) // animals.sort()
    sort(food)    // food.sort()

    count = 0
    for f in food {
        if f >= animals[count] {
            count += 1
        }

        if count == len(animals) {
            break
        }
    }

    return count
}
```

Найти разницу между двух строк

На вход функции подается две строки: a и b. Строка b образована из строки a путем перемешивания и добавления одной буквы. Необходимо вернуть эту букву

Найти разницу между двух строк

- строка a: uio строка b: oeiu результат: e
- строка a: fe строка b: efo результат: o
- строка a: ab строка b: ab результат: “”
- строка a: bbb строка b: bbbb результат: b
- Как будем использовать хеш-таблицы?

```
function extraLetter(a, b) {
    …

    return ""
}
```

Найти разницу между двух строк

- Инициализируем хеш-таблицу для a
- Ключ таблицы это строка, буква из строки b
- Значение - кол-во повторений буквы в строке b

```
function extraLetter(a, b) {
    hashMapB = map[string]int

    // заполняем в цикле по b hashMapB

    return ""
}
```

Найти разницу между двух строк

- Инициализируем хеш-таблицу для a
- Ключ таблицы это строка, буква из строки b
- Значение - кол-во повторений буквы в строке b

```
function extraLetter(a, b string) {
        hashMapB := map[string]int{}
        for i := 0; i < len(b); i++ {
                hashMapB[b[i]]++
        }

        return ""
}
```

Найти разницу между двух строк

```
function extraLetter(a, b string) string {
        hashMapB := map[string]int{}
        for i := 0; i < len(b); i++ {
                hashMapB[b[i]]++
        }

        // итерируемся по строке a
        // на каждое вхождение буквы
        // из строки a в hashMapB
        // декрементируем счетчик
        // у соответствующего ключа

        return ""
}
```

Найти разницу между двух строк

Теперь в hashMapB содержится лишь одна буква у которой значение больше нуля

```
function extraLetter(a, b) {
        hashMapB := map[string]int{}
        for i := 0; i < len(b); i++ {
                hashMapB[string(b[i])]++
        }
        for i := 0; i < len(a); i++ {
                if contains(hashMapB, a[i]) {
                        hashMapB[a[i]]--
                }
        }

        ...

        return ""
}
```

Найти разницу между двух строк

```
function extraLetter(a, b string) string {
	hashMapB := map[string]int{}
	// O(n)
	for i := 0; i < len(b); i++ {
		hashMapB[string(b[i])]++
	}
	// O(n)
	for i := 0; i < len(a); i++ {
		if contains(hashMapB, a[i]) {
			hashMapB[a[i]]--
		}
	}
	// O(n)
	for letter, count = range(hashMapB) {
		if count > 0 {
			return letter
		}
	}

	return ""
}
```

Найти разницу между двух строк

```
function extraLetter(a, b) {
	hashMapA := map[string]int{}
	// O(n)
	for i := 0; i < len(a); i++ {
		hashMapA[a[i]]++
	}
	// O(n)
	for i := 0; i < len(b); i++ {
		if contains(hashMapA, b[i]) {
			hashMapA[b[i]]--
			// как только счетчик становится
			// равен 0 - удаляем элемент
			// из хеш-таблицы
			if hashMapA[b[i]] == 0 {
				delete(hashMapA, b[i])
				continue
			}
			continue
		}
		// если мы удалили ключ со значением b[i]
		// или же этой буквы никогда не было в таблицы
		return b[i]
	}

	return ""
}
```

Сумма двух элементов массива

Дан **не** отсортированный массив целых чисел и некоторое число target. Необходимо написать функцию, которая найдет два таких элемента в массиве, сумма которых будет равна target
Один элемент можно использовать лишь один раз. В случае если два таких элемента не нашлось, возвращаем пустой массив

Сумма двух элементов

- Мы уже решали похожую задачу, но там был отсортированный массив
- Мы можем отсортировать его методом Шелла и применить прежнее решение
- Делать мы так конечно же не будем

```
function twoSum(data, target) {
        cache := map[array_element]index_of_array_element

        return []
}
```

Сумма двух элементов

```
function twoSum(data, target) {
        cache := map[int]int
        for i = 0; i < len(data); i++ {
                // на каждой итерации вычисляем
                // возможный второй элемент
        }
        return []
}
```

Сумма двух элементов

```
function twoSum(data, target) {
        cache := map[int]int
        for i = 0; i < len(data); i++ {
                diff = target - data[i]
                // ищем diff в cache
        }
        return []
}
```

Сумма двух элементов

```
function twoSum(data, target) {
        cache := map[int]int
        for i = 0; i < len(data); i++ {
                diff = target - data[i]
                if contains(cache, diff) {
                    return [i, cache[diff]]
                }
        }

        return []
}
```

Сумма двух элементов

```
function twoSum(data, target) {
        cache := map[int]int
        for i = 0; i < len(data); i++ {
                if contains(cache, data[i]) {
                    return [i, cache[data[i]]
                }
                diff = target - data[i]
                cache[diff] = i
        }

        return []
}
```

Всем спасибо:)

И хорошего вечера!