Сортировки — квадратичные и сортировка слиянием

- Квадратичные сортировки: пузырьком, выбором, вставками
- Сортировка подсчетом
- О-нотация
- Сортировка слиянием*
- Применение сортировки для решения задач

Мы считаем, что вы уже знаете и умеете следующие вещи:

- основы языков Python, C++ или Java (примеры в конспектах будут чаще всего на питоне)
- оператор if
- циклы for и while
- создание функций
- концепцию рекурсии
- поиск минимума в массиве

Квадратичные сортировки

Задача **сортировки** массива заключается в том, чтобы расставить его элементы в определённом порядке (чаще всего - по неубыванию. Это означает, что каждый элемент должен быть больше или равен всех предыдущих).

А именно мы хотим написать такую функцию `sort_array` , которая принимает массив в качестве аргумента и сортирует его элементы:

In [0]:
```
def sort_array(array):
    # надо придумать, что написать здесь

array = [1, -3, 7, 88, 7]
sort_array(array)
print(array)
```

```
[1, -3, 7, 88, 7]
```

Задание

Какие вы знаете алгоритмы сортировки?

Если вы не знаете ни одного алгоритма, то придумайте какой-нибудь разумный алгоритм сами.

Сортировка пузырьком

Это самый популярный алгоритм сортировки, хоть он и не самый очевидный.

Пусть N - длина массива. Сортировка пузырьком заключается в том, что мы просто N раз пройдемся по массиву и будем менять два соседних элемента, если первый больше второго.

Ссылка на красивую визуализацию: https://visualgo.net/nl/sorting (https://visualgo.net/nl/sorting)

Заметьте, как каждую итерацию максимальный элемент "всплывает как пузырек" к концу массива.

In [0]:
```
def bubble_sort(array):
    n = len(array) # длина массива
    for i in range(n): # n раз выполняем цикл
        for j in range(n - 1): # проходимся по всем элементам кроме последнего
            if array[j] > array[j + 1]: # сравниваем элемент по следующим
                a[j], a[j + 1] = a[j + 1], a[j] # меняем местами, если следующий меньше

a = [1, -3, 7, 88, 7]
bubble_sort(a)
print(a)
```

```
[-3, 1, 7, 7, 88]
```

Заметьте, что **после i шагов алгоритма сортировки пузырьком последние $i + 1$ чисел всегда отсортированы**. Именно поэтому алгоритм и работает.

И именно поэтому его можно немного ускорить.

Задание

Подумайте, какие лишние элементы мы перебираем. Как нужно изменить границы в двух циклах for, чтобы не делать никаких бесполезных действий?

Сортировка выбором

Более понятным и придумываемым способом является **сортировка выбором** минимума (или максимума).

Чтобы отсортировать массив, нужно просто N раз выбрать минимум среди еще неотсортированных чисел. То есть на i-ом шаге ищется минимум на отрезке $[i, n-1]$, и этот минимум меняется с i-ым элементом, теперь отрезок $[0, i]$ отсортирован.

Красивая визуализация (https://visualgo.net/nl/sorting) (вкладка SEL).

Сортировка вставками

Также существует **сортировка вставками**.

Префиксом длины i будем называть первые i элементов массива.

Тогда пусть на i-ом шаге у нас уже будет отсортирован префикс до i-го элемента. Чтобы этот префикс увеличить, нужно взять элемент, идущий после него, и менять с левым соседом, пока этот элемент наконец не окажется больше своего левого соседа. Если в конце он больше левого соседа, но меньше правого - это значит, что мы правильно вставили этот элемент в отсортированную часть массива.

Красивая визуализация (https://visualgo.net/nl/sorting) (вкладка INS).

Задание

Сдайте 4 первые задачи в контесте (https://informatics.msk.ru/mod/statements/view.php?id=33164):

1. Напишите сортировку пузырьком.
2. Напишите сортировку выбором максимума.
3. Напишите сортировку вставками.
4. Выберите любой алгоритм и примените его для сортировки пар.

При сдаче задач нельзя пользоваться встроенной сортировкой!

Сортировка подсчетом

Предыдущие три алгоритма работали с массивами, в которых лежат абсолютно любые объекты, которые можно сравнивать. Любые числа, строки, пары, другие массивы, почти все что угодно.

Но в особых случаях, когда элементы принадлежат какому-то маленькому множеству, можно использовать другой алгоритм — **сортировку подсчетом**.

Пусть, например, нам **гарантируется, что все числа натуральные и лежат в промежутке от 1 до 100**.

Тогда есть такой простой алгоритм: создадим массив размера 100, в котором будем хранить на i-ом месте, сколько раз число i встретилось в этом массиве. Пройдемся по всем числам, и увеличим соответствующее значение массива на 1. Таким образом мы подсчитали, сколько раз какое число встретилось. Теперь можно просто пройтись по этому массиву и вывести 1 столько раз, сколько раз встретилась 1, вывести 2 столько раз, сколько встретилась 2, и так далее.

Красивая визуализация (https://visualgo.net/nl/sorting) (вкладка COU).

Задание

Сдайте 5, 6 и 7 задачи в контесте (https://informatics.msk.ru/mod/statements/view.php?id=33164):

1. Напишите сортировку подсчетом.
2. Придумайте, как преобразовать сортировку подсчетом, чтобы она работала быстро.
3. Придумайте, как использовать идею сортировки подсчетом в этой задаче.

При сдаче задач нельзя пользоваться встроенной сортировкой!

О-нотация

Очень часто требуется оценить, сколько времени работают эти алгоритмы. Но тут возникают проблемы:

- на разных компьютерах время работы всегда будет слегка отличаться;
- чтобы измерить время, придётся запустить сам алгоритм, но иногда приходится оценивать алгоритмы, требующие часы или даже дни работы.

Зачастую основной задачей программиста становится оптимизировать алгоритм, выполнение которого займёт тысячи лет, до какого-нибудь адекватного времени работы. Поэтому хотелось бы уметь предсказывать, сколько времени займёт выполнение алгоритма ещё до того, как мы его запустим.

Для этого давайте для начала попробуем оценить **число операций** в алгоритме.

Возникает вопрос: какие именно операции считать. Можно считать любые элементарные операции:

- арифметические операции с числами: `+, -, *, /`
- сравнение чисел: `<, >, <=, >=, ==, !=`
- присваивание: `a[0] = 3`

При этом надо учитывать, как реализованы некоторые отдельные вещи в самом языке. Например, в питоне срезы массива (`array[3:10]`) копируют этот массив, то есть этот срез работает за 7 элементарных действий. А `swap` , например, может работать за 3 присваивания.

Задание

Попробуйте посчитать точное число **сравнений** в сортировках пузырьком, выбором, вставками и подсчетом в худшем случае (это должна быть какая формула, зависящая от N - длины массива). Для сортировки подсчетом давайте считать, что все числа целые и лежат в промежутке от 1 до M

И также посчитайте точное число **присваиваний** в сортировках пузырьком, выбором, вставками и подсчетом в худшем случае. Давайте считать, что `swap` — это 3 присваивания.

Ниже будут ответы

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

В худшем случае алгоритмы работают за столько сравнений:

- Сортировка пузырьком (без улучшения): $N(N-1)$
- Сортировка пузырьком (с улучшением): $(N-1)+(N-2)+\ldots+1=\frac{N(N-1)}{2}$
- Сортировка выбором: $(N-1)+(N-2)+\ldots+1=\frac{N(N-1)}{2}$
- Сортировка вставками: $1+2+\ldots+(N-1)=\frac{N(N-1)}{2}$
- Сортировка подсчетом: нет сравнений

И столько присваиваний:

- Сортировка пузырьком (без улучшения): $3\frac{N(N-1)}{2}$
- Сортировка пузырьком (с улучшением): $3\frac{N(N-1)}{2}$
- Сортировка выбором: $3(N-1)+\frac{N(N-1)}{2}$
- Сортировка вставками: $3\frac{N(N-1)}{2}$
- Сортировка подсчетом: $N+M$ (M - это создание массива)

Но чтобы учесть все элементарные операции, ещё надо посчитать, например, сколько раз прибавилась единичка внутри цикла `for` . А ещё, например, строчка `n = len(array)` - это тоже действие.

Также не сразу очевидно, какой из этих алгоритмов работает быстрее. Сравнивать формулы сложно. Хочется придумать способ упростить эти формулы так, чтобы:

- не нужно было учитывать много информации, не очень сильно влияющей на итоговое время;
- легко было оценивать время работы разных алгоритмов для больших чисел;
- легко было сравнивать алгоритмы на предмет того, какой из них лучше подходит для тех или иных входных данных.

Для этого придумали O**-нотацию** - асимптотическое время работы вместо точного (часто его ещё называют асимптотикой).

Пусть $f(N)$ - это какая-то функция. Говорят, что алгоритм работает за $O(f(N))$, если существует число C, такое что алгоритм работает не более чем за $C \cdot f(N)$ операций.

В таких обозначениях можно сказать, что

- Сортировка пузырьком работает за $O(N^2)$
- Сортировка выбором работает за $O(N^2)$
- Сортировка вставками работает за $O(N^2)$
- Сортировка подсчетом работает за $O(N + M)$

Это обозначение удобно тем, что оно короткое и понятное, а также оно не зависит от умножения на константу или прибавления константы. Если алгоритм работает за $O(N^2)$, то это может значить, что он работает за N^2, за $N^2 + 3$, за $\frac{N(N-1)}{2}$ или даже за $1000 \cdot N^2 + 1$ действие. Главное, что функция ведет себя как N^2, то есть при увеличении N (в данном случае это длина массива) он увеличивается как некоторая квадратичная функция. Например, если увеличить N в 10 раз, время работы программы увеличится приблизительно в 100 раз.

Поэтому все эти рассуждения про то, сколько операций в `swap` или считать ли отдельно присваивания, сравнения и циклы - отпадают. Как бы вы ни ответили на эти вопросы, они меняют ответ на константу, а значит асимптотическое время работы алгоритма никак не меняется.

Первые три сортировки именно поэтому называют **квадратичными** - они работают за $O(N^2)$. Сортировка подсчетом работает намного быстрее - она работает за $O(N + M)$ действий, то есть если в задаче $M < N$, то это вообще линейная функция $O(N)$. Линейная функция растет гораздо медленнее, чем квадратичная, так что эта сортировка гораздо лучше, чем любая другая квадратичная. У нее есть лишь один недостаток - ее можно применять только если множество значений конечное и состоит из чисел от 1 до M.

Задание

Найдите асимптотику данных функций. Максимально упростите ответ (например, до $O(N)$, $O(N^2)$ и т. д.).

- $\frac{N}{3}$
- $\frac{N(N-1)(N-2)}{6}$
- $1 + 2 + 3 + \ldots + N$
- $1^2 + 2^2 + 3^2 + \ldots + N^2$
- $\log N + 3$
- 7
- 10^{100}

Задание

Найдите асимптотическое время работы данных функций:

In [0]:
```
def f(n):
    s = 0
    for i in range(n):
        for j in range(n):
            s += i * j
    return s

f(10)
```

Out[0]: 2025

In [0]:
```
def g(n):
    s = 0
    for i in range(n):
        s += i
    for i in range(n):
        s += i * i
    return s

g(10)
```

Out[0]: 330

In [0]:
```
def h(n):
    if n == 0:
        return 1
    return h(n - 1) * n

h(10)
```

Out[0]: 3628800

Задание

Найдите лучшее время работы алгоритмов, решающих данные задачи:

- Написать числа от 1 до N
- Написать все тройки чисел от 1 до N
- Найти разницу между максимумом и минимумом в массиве
- Найти число единиц в бинарной записи числа N

Сортировка слиянием* (для тех, кто всё успевает)

Возникает вопрос: а бывают ли сортировки, которые быстрее, чем квадратиные, и работают всегда?

Ответ — да. Есть несколько известных сортировок, работающих за $O(N \log N)$, и доказано, что асимптотически быстрее сортировок не бывает.

Давайте подробнее рассмотрим **сортировку слиянием**, она же **MergeSort**.

Ссылка на красивую визуализацию: https://visualgo.net/nl/sorting (https://visualgo.net/nl/sorting) (вкладка MER)

Для начала определим функцию **слияния** (**`merge`**) двух отсортированных массивов - она возвращает отсортированный массив, состоящий из элементов обоих массивов, и работает при этом за $O(N)$ (где N - общее число элементов).

In [0]:
```
def merge(a, b):
    # надо придумать, что написать здесь

merge([1, 3, 7, 10, 100], [2, 7, 7, 7, 11, 13, 18])
```

Out[0]:
```
[1, 2, 3, 7, 7, 7, 7, 10, 11, 13, 18, 100]
```

В сущности слить два массива просто - это делается с помощью двух указателей i и j. Изначально они равны 0 (то есть указывают на нулевые элементы массивов a и b). После этого достаточно смотреть на элементы $a[i]$ и $b[j]$ и минимальный из них класть в результирующий массив, после чего соответствующий указатель надо двигать дальше. Дальше нужно повторять этот процесс заново, пока мы не дошли до конца обоих массивов.

Когда функция `merge` уже написана, сам `merge_sort` писать уже легко - надо воспользоваться рекурсией. Чтобы отсортировать массив, достаточно отдельно отсортировать его левую и правую половины рекурсивно, и после этого эти половины слить.

Для удобства написания кода фунции можно сделать вот такими:

In [0]:
```
def merge(array, a, b, c):
    # функция сливает элементы массива array [a, b) и [b, c)
    # надо придумать, что написать здесь

arr = [1, 1, 7, 10, 100, 2, 7, 40, 78, 6, 13, 100]
merge(arr, 6, 9, 12)
print(arr)
merge(arr, 0, 6, 12)
print(arr)
```

```
[1, 1, 7, 10, 100, 2, 6, 7, 13, 40, 78, 100]
[1, 1, 2, 6, 7, 7, 10, 13, 40, 78, 100, 100]
```

In [0]:
```
def mergesort(array, a, c):
    # функция сортирует элементы массива array [a, c)
    # надо придумать, что написать здесь

arr = [1, 1, 7, 10, 100, 2, 7, 40, 78, 6, 13, 100]
mergesort(arr, 6, 12)
print(arr)

arr = [1, 1, 7, 10, 100, 2, 7, 40, 78, 6, 13, 100]
mergesort(arr, 0, 12)
print(arr)
```

Задание

Разобраться, реализовать сортировку слиянием и сдать последнюю задачу в этом контесте:

https://informatics.msk.ru/mod/statements/view3.php?id=33164
(https://informatics.msk.ru/mod/statements/view3.php?id=33164)

Чтобы сдать это задание, нужно писать красивый код - будет проведено ревью.

Применение сортировки для решения задач

Также сортировка очень часто применяется как часть решения олимпиадных задач. В таких случаях обычно используют встроенную сортировку sort. Она на разных языках может быть реализована по-разному, но везде она работает за $O(NlogN)$, и, обычно, неплохо оптимизирована.

In [1]:
```
# На питоне она пишется так
a = [1, 5, 10, 5, -4]
a.sort()
print(a)
```

```
[-4, 1, 5, 5, 10]
```

In [0]:
```
// на c++11 она пишется так

#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main() {
    vector<int> v = {1, 5, 10, 5, -4};
    sort(v.begin(), v.end());
    for (auto x : v) {
        cout << x << " ";
    }
}
```

Задание

Решить как можно больше задач из этого контеста:

https://informatics.msk.ru/mod/statements/view.php?id=33855
(https://informatics.msk.ru/mod/statements/view.php?id=33855)

В этом контесте можно и нужно использовать встроенную сортировку sort.