Homework #1

Introduction, measuring code ...

Anastasiia Konoplina

---

1. Generate random data consisting of unsigned 64bit integers. Implement some sorting algorithm (any) on your own. Repeat measurements many times (100+) to study how much the times will vary. Increase the array size (n) and plot the measured times so that you can visually explore the timings. Discuss which data sizes are meaningful to analyze in practice.

For this homework I used python. As I understood from this article unsigned 64bit integers are "positive" integers in range [0; (2^64)-1], so that I generated an 10000 elements array using following code:

```
import random

lim = 2**64 - 1
array = [random.randint(1, lim) for _ in range(10000)]
```

The implementation of sorting algorithm:

```
def sort_func(array):

    k = 1
    while k < len(array):
        for i in range(len(array)-k):
            if array[i] > array[i+1]:
                array[i],array[i+1] = array[i+1],array[i]
        k += 1
    return array
```

I measured time of sorting using this algorithm 150 times using following code:

```
for i in range(1,150):

    start_time = time.time()
    res2 = sort_func(array)
    res_time1 = time.time() - start_time
    manual.append(res_time1)
```

So as result I got an array of times of execution. Min is 3.25329113007s, max is 6.55651092529s, mean is 3.32587398459.

To plot the results I used mean values. Result is in the end of second task.

---

2. Now measure the language-specific built-in sort algorithm. Plot the execution times as in the first. Try to also overlay some function f(x) that would approximate the built-in sort timings well.

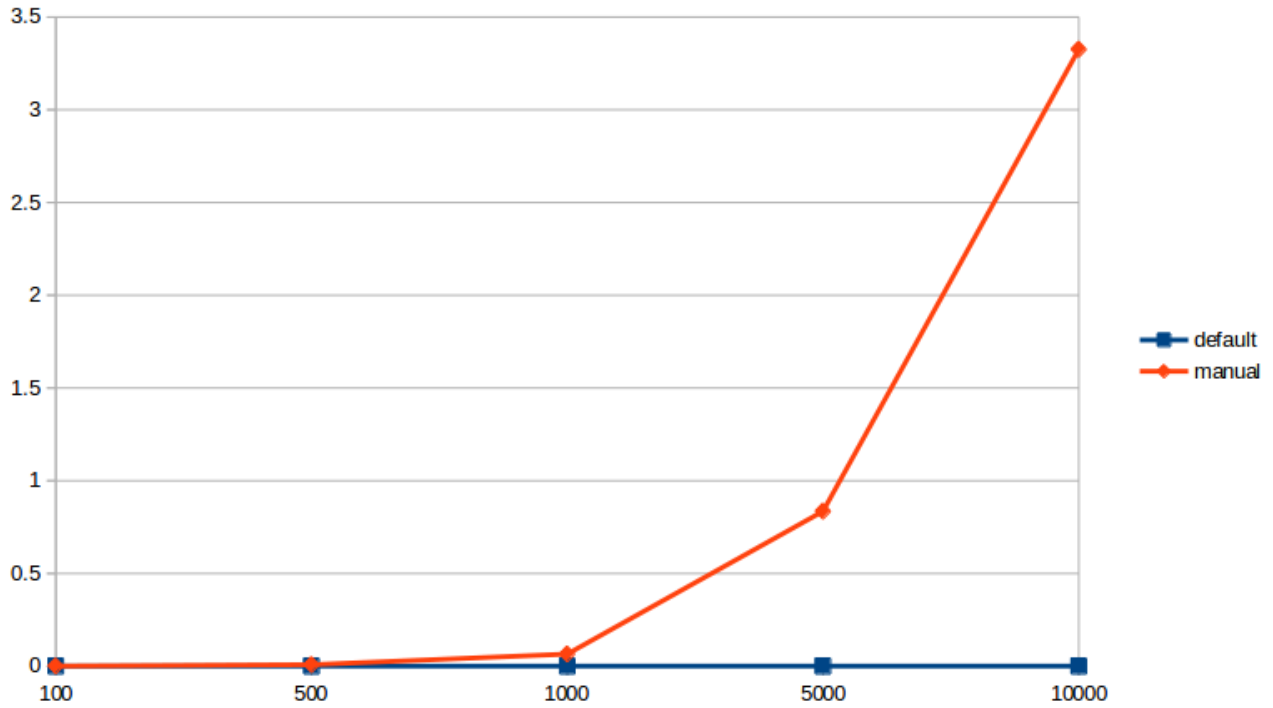
Time of execution for 150 times:

Min - 0.000248908996582

Max - 0.00417804718018

Mean - 0.000294926982598

Following graph show run time of both sorting algorithms: custom(red) and built-in(blue).



We can see that increasing array size leads to exponential increasing of run time of manual algorithm (red).

Number of elements in array	Built-in, seconds	Custom, seconds
100	0.00000405798153001	0.000458556778577
500	0.0000117749583964	0.00799235519098
1000	0.0000273208228909	0.0652229785919
5000	0.000124145674225	0.835148018234
10000	0.000294926982598	3.32587398459

3. Measure how big a table can you roughly sort within 1 minute for both your sort (1) and with built-in sort (2)? Predict what would happen if you increased data 100 fold for 1 and 2 by extrapolating from the measurements on smaller input sizes.

For sort1 (my sort) – appx array of 32 000 elements

For sort2 (built-in) – appx array of 500 000 000 elements

5. Prove that n! ∈ ω(2^n)

Let's assume:

f(n) = n^2
g(n) = n * logn

f(n) belongs to Ω(g(n)) if there is c > 0 and n0 > 0 so that:
f(n) >= c * g(n) for every n >= n0 so that:
n*n >= c*n*logn
n >= c*logn
If we take c = 1 and n0 = b^b, b > 1 (log base):
n >= logn if n>=n0 (because b^b >= b for b > 1)

Just want to say that basically it is not my solution, I have found it in stackoverflow, but I do understand why it is like this.