# 1. Dive into algorithm, speed and space

By [Xiaoou WANG](https://xiaoouwang.github.io)

Just like the math series, this tutorial is an introduction to algorithms and data structures. It targets students in the humanities and social sciences but not only because I think that learning by examples can benefit all those who like coding and top-down learning.

Let’s first consider a simple example for illustrative purpose. Suppose you want to write a function filtering all the even numbers smaller than and you have to choose one from the following two options.

%%time  
temp\_l = []  
def print\_evens\_fast(n):  
 i = 0  
 while i < n:  
 if i % 2 == 0:  
 temp\_l.append(i)  
 i += 2  
print\_evens\_fast(50)

CPU times: user 24 µs, sys: 0 ns, total: 24 µs  
Wall time: 27.9 µs

%%time  
temp\_l = []  
def print\_evens\_slow(n):  
 i = 0  
 while i < n:  
 if i % 2 == 0:  
 temp\_l.append(i)  
 i += 1  
print\_evens\_slow(50)

CPU times: user 27 µs, sys: 1 µs, total: 28 µs  
Wall time: 30 µs

In the output above, you should mainly focus on the wall time. Roughly speaking, wall time, also called real, represents actual elapsed time, while user and sys values represent CPU execution time.

As you see the print\_evens\_fast is slightly fast than the second one. The difference, however, seems trivial. If you have a critical mind, you will object, rightly so, that a single trial is quite subject to randomness.

A practical trick would be using the timeit%% magic command in Ipython. In this case I’ll use r1 n100 meaning 1 run of 100 loops.

%%timeit -r1 -n100  
temp\_l = []  
def print\_evens\_fast(n):  
 i = 0  
 while i < n:  
 if i % 2 == 0:  
 temp\_l.append(i)  
 i += 2  
print\_evens\_fast(50)

4.07 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)

%%timeit -r1 -n100  
temp\_l = []  
def print\_evens\_slow(n):  
 i = 0  
 while i < n:  
 if i % 2 == 0:  
 temp\_l.append(i)  
 i += 1  
print\_evens\_slow(50)

6.65 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)

The same conclusion still stands.

One major drawback of the present example is it’s artificial aspect and the trivial benefits of choosing a slightly faster algorithm.

You might wonder: are such tricks really useful/necessary in real-life scenarios?

Let’s take a look at another example, this time much more practical.

Websites often use ID to manage customs. Every day some IDs are used and some are freed. When some client tries to acquire a new ID, we want to always allocate it the smallest available one.

So how can you find the smallest free ID, which is 10, from the following list?

[18, 4, 8, 9, 16, 1, 14, 7, 19, 3, 0, 5, 2, 11, 6]

The most intuitive approach would be a brute-force search. However, this algorithm doesn’t scale well. For instance, let’s try a list of 50000 ids.

%%time  
def brute\_force(lst):  
 i = 0  
 while True:  
 if i not in lst:  
 print(f"The user can use the free id {i}")  
 break  
 i = i + 1  
  
import random  
# reproductibility  
random.seed(0)  
  
nb\_ids = 50000  
lst = list(range(nb\_ids))  
lst\_shuffled = random.sample(lst, len(lst))  
print(f"the first 6 ids of the shuffled list is {lst\_shuffled[:6]}")  
nb\_removed = random.randrange(nb\_ids)  
lst\_shuffled.pop(nb\_removed)  
brute\_force(lst\_shuffled)

the first 6 ids of the shuffled list is [25247, 49673, 27562, 2653, 16968, 33506]  
The user can use the free id 34838  
CPU times: user 9.42 s, sys: 10.2 ms, total: 9.43 s  
Wall time: 9.44 s

9.44 s, that’s a lot of time. If a user has to wait for 9.44 s before being attributed an ID. Chances are he will be long gone before the process is terminated.

Now if we use another algorithm based on the fact that:

for a series of n numbers , some of the must be outside the range [0, n) if there are free numbers, otherwise the list is exactly a permutation of and n should be returned as the minimum free number.

Let’s translate the above theorem into python：

%%time  
def min\_free(lst):  
 n = len(lst)  
 a = [0]\*(n+1)  
 for x in lst:  
 if x < n:  
 a[x] = 1  
 print(f'The id {a.index(0)} can be used.')  
  
min\_free(lst\_shuffled)

The id 34838 can be used.  
CPU times: user 4.44 ms, sys: 203 µs, total: 4.64 ms  
Wall time: 4.51 ms

As you can see the min\_free version spits the same ID, however it takes only 4.51 ms. Note that, given the huge gap, we can save ourselves from running multiple loops.

There remains a minor issue. Isn’t it tedious to run the program each time or even multiple times to have an idea of the speed ? Besides, the performance can differ quite significantly according to one’s computer configuration.

Luckily people figured out a easier and more objective way of measuring time called Big O notation (actually it measures the time complexity of an algorithm). For this particular case the time complexity of the brute force one is and the other one . However the faster uses more memory (consider the list a set aside as a bookkeeper), we say that the space complexity is .