Basics: Algorithms and Data Structures

cyril.nicaud@univ-eiffel.fr

- 4 courses + exercise sessions (on mondays)
- grade = small project in Python

Course 1: Algorithms and Complexity

1. Algorithms

Definition: an algorithm is an automatic process that solves a task in a finite number of steps

At primary school, we learned an algorithm to compute the sum of two numbers:

	4	3	7	8
+	1	6	4	1

	6	0	1	9

Définition: an algorithm is an automatic process that solves a task in a finite number of steps

In [2]:

```
def find(x,T):
    for y in T:
        if x == y:
            return True
    return False
```

Important: a given problem can have several solutions (algorithms)

Example: find an element in a *sorted* array

- the algorithm `find(x,T)` can be used
- the binary search is another solution, which uses the fact the array is sorted

Both algorithms are correct: they solve the problem.

In [1]:

```python
def binary_search(x, T):
    debut, fin = 0, len(T)-1
    while debut <= fin:
        m = (debut + fin) // 2
        if T[m] == x: return True
        elif T[m] < x: debut = m + 1
        else: fin = m - 1
    return False
```

When we have several solution for a given problem, we can:

- compare them experimentally if some benchmarks are available
- compare them theoretically by studying their complexity

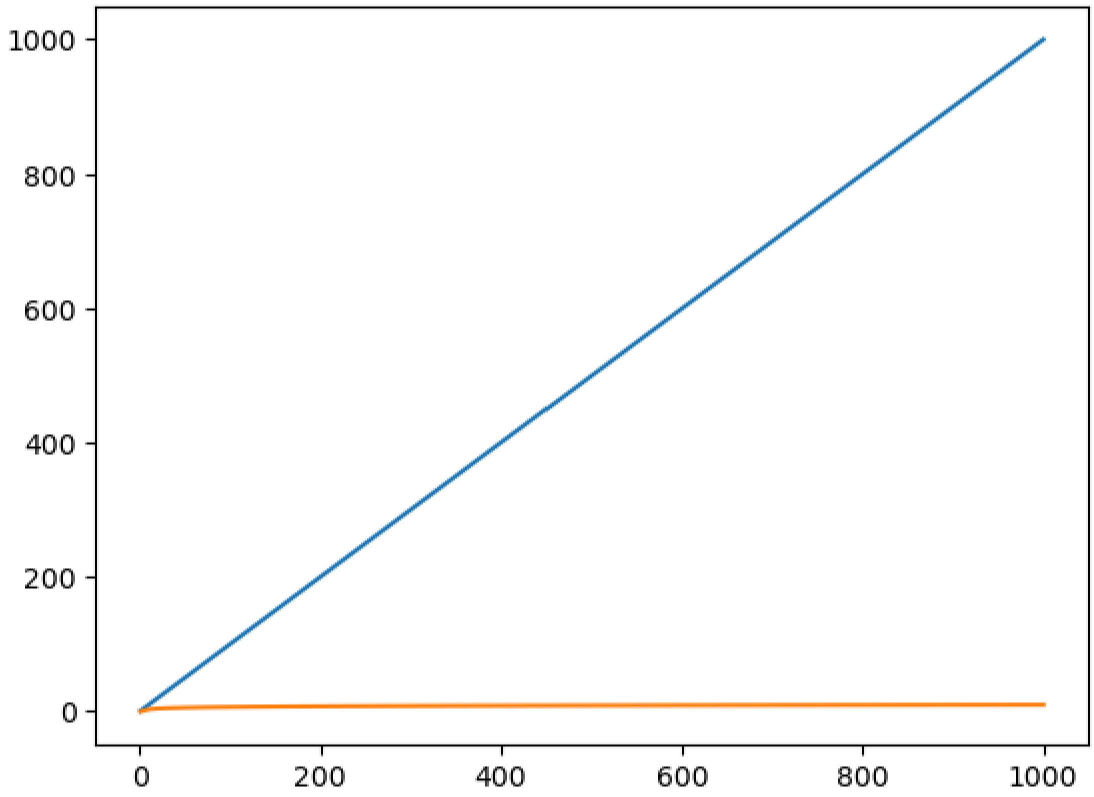
Examples of complexity:

`find` is in $\mathcal{O}(n)$

`binary_search` is in $\mathcal{O}(\log n)$

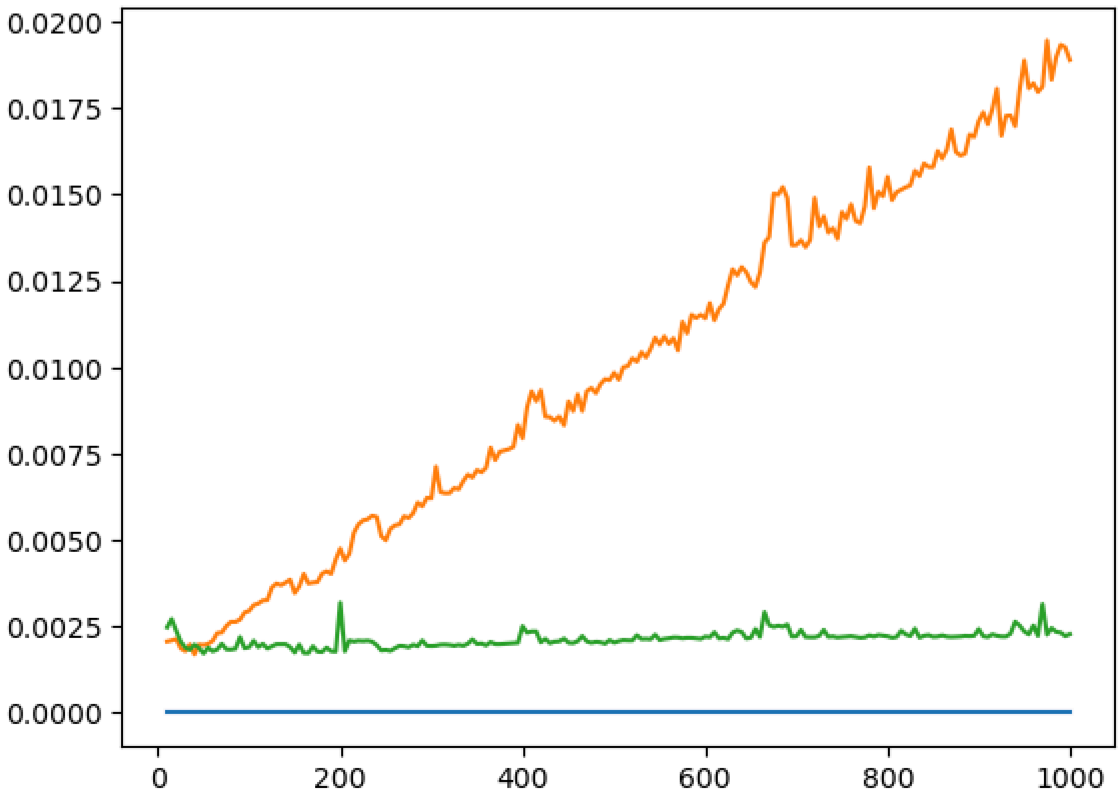
In [4]:

```
f = lambda x:x
g = lambda x:log(x,2)

draw_curves(1,1000,[f,g])
```



In [5]:

```python
time_dicho, time_find, iterations = [], [], 1000
L = range(10, 1001, 5)
for n in L: # taille du tableau
    T = list(range(n))
    t1 = time()
    for _ in range(iterations):
        if random() < .5: x = randrange(0,n) + 0.1
        else: x = randrange(0,n)
        find(x, T)
    time_find.append(time() - t1)

    t1 = time()
    for _ in range(iterations):
        if random() < .5: x = randrange(0,n) + 0.1 # x pas dedans
        else: x = randrange(0,n) # x dedans
        binary_search(x, T)
    time_dicho.append(time()-t1)
plt.plot([L[0],L[-1]],[0,0]) # axis in blue
plt.plot(L,time_find)        # find in orange
plt.plot(L,time_dicho)       # binary_search in green
plt.show()
```



0.0200
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
0
200
400
600
800
1000

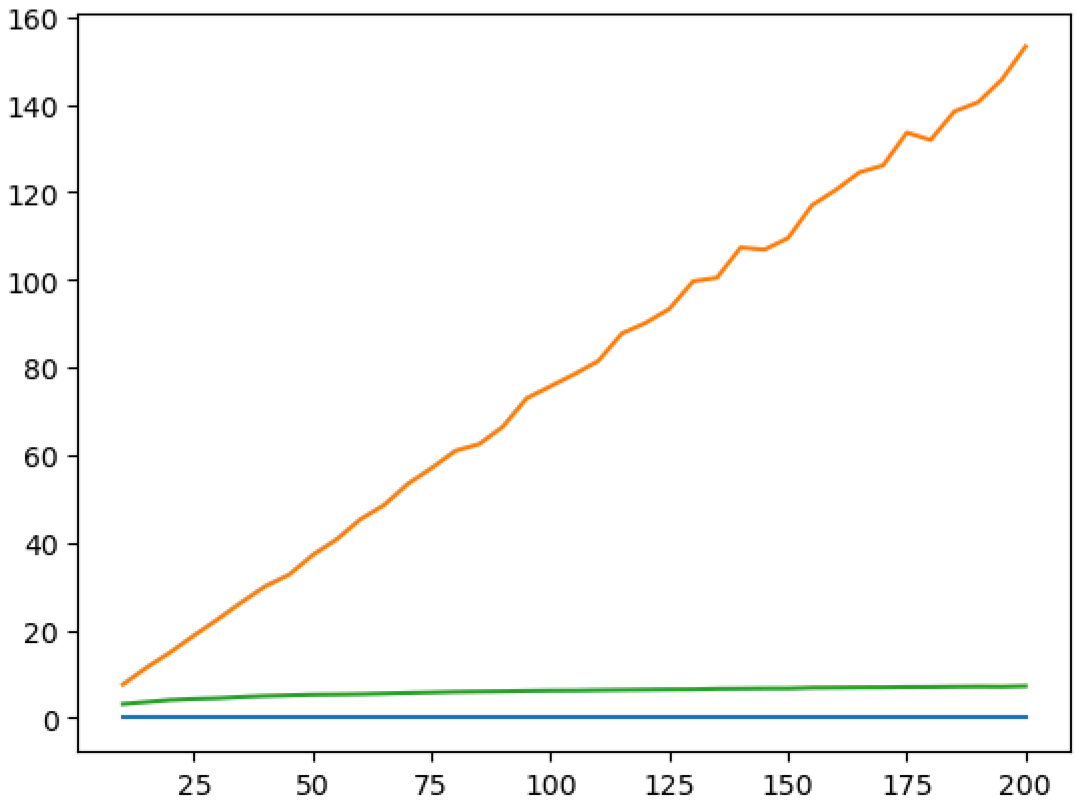
An other approach, as measuring time is complicated, is to use counters

In [7]:

```
def find_c(x,T):
    c = 0 # count the number of comparisons
    for y in T:
        c += 1
        if x == y:
            return c
    return c

def binary_search_c(x, T):
    debut, fin, c = 0, len(T)-1, 0
    while debut <= fin:
        c += 1
        m = (debut + fin) // 2
        if T[m] == x:
            return c
        elif T[m] < x:
            debut = m + 1
        else:
            fin = m - 1
    return c
```

In [8]:

```
iterations = 1000
L = list(range(10, 201, 5))
c_dicho, c_find = [0] * len(L), [0] * len(L)
for i in range(len(L)):
    T = list(range(L[i]))
    for _ in range(iterations):
        if random() < .5: x = randrange(0, L[i]) + 0.1
        else: x = randrange(0, L[i])
        c_find[i] += find_c(x, T) / iterations
        c_dicho[i] += binary_search_c(x, T) / iterations
plt.plot([L[0],L[-1]],[0,0]) # axis in blue
plt.plot(L,c_find)        # find in orange
plt.plot(L,c_dicho)       # binary_search in green
plt.show()
```



160
140
120
100
80
60
40
20
0
25
50
75
100
125
150
175
200

2. Complexity

Definition: studying the complexity of an algorithm consists in estimating the quantity of ressources that are needed

Typical resources

- time
- space
- energy
- ...

To study the complexity of an algorithm, we need a notion of size for the inputs: in the sentence "`find` is in $O(n)$", n is the size of the array

In most cases, the size is quite natural:

- the length of a string
- the number of cells of an array
- the number of elements of a list
- $\ldots$

Warning: the only commonly encountered difficulty is for integers!

- in most cases (indices, numbers in an array, labels for vertices, etc) the size of an integer can be considered as constant
- for algorithms using large numbers (cryptography, etc) it is proportionnal to the size of the binary encoding of the number: to encode n in binary, we need $\approx \log_2 n$ bits

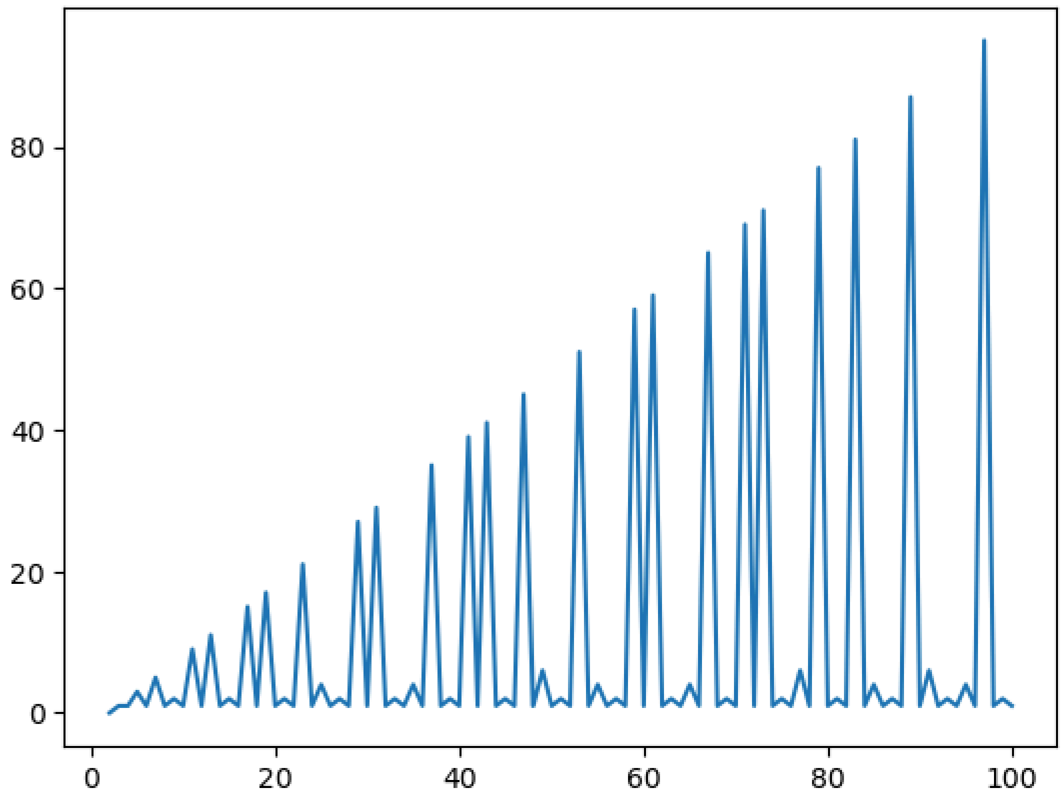
In [9]:

```
def is_prime(n):
    for i in range(2,n):
        if n%i == 0:
            return False
    return True

print([x for x in range(2,100) if is_prime(x)])
```

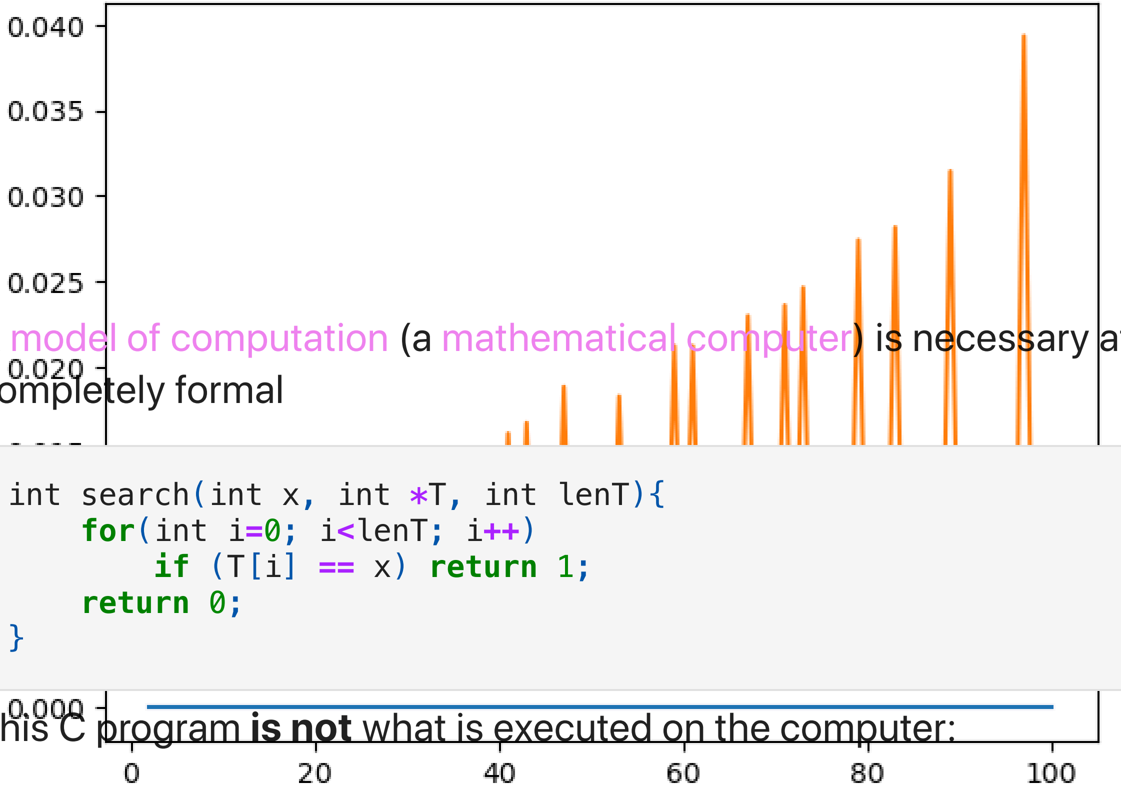
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 7
1, 73, 79, 83, 89, 97]
```

In [10]:

```
def is_prime_c(n):
    c = 0
    for i in range(2,n):
        c += 1
        if n%i == 0:
            return c
    return c

X = list(range(2,101))
Y = [is_prime_c(x) for x in X]
plt.plot(X,Y)
plt.show()

draw_time_curve(X, is_prime, 10**4)
```



0
20
40
60
80
0
20
40
60
80
100



A model of computation (a mathematical computer) is necessary at some point to be completely formal

In [ ]:

```
int search(int x, int *T, int lenT){
    for(int i=0; i<lenT; i++)
        if (T[i] == x) return 1;
    return 0;
}
```

This C program **is not** what is executed on the computer:

- Python is an interpreted language
- C is a compiled language

The computer uses machine code (machine language)

A model of computation (a mathematical computer) is necessary at some point to be completely formal

- Turing Machine (similar to finite state automata)
- Random Access Machine (kind of mathematical processor, with an assembly)
- ...

Fortunately, we won't need such details for this course, we consider that after being interpreted/compiled we have the following basic costs:

- an elementary instruction is performed in constant time
- a memory access is performed in constant time

Définition : for a given input E of an algorithm, let $C(E)$ be the quantity of ressources (time, number of elementary instructions, ...) used by the algorithm when running on E.

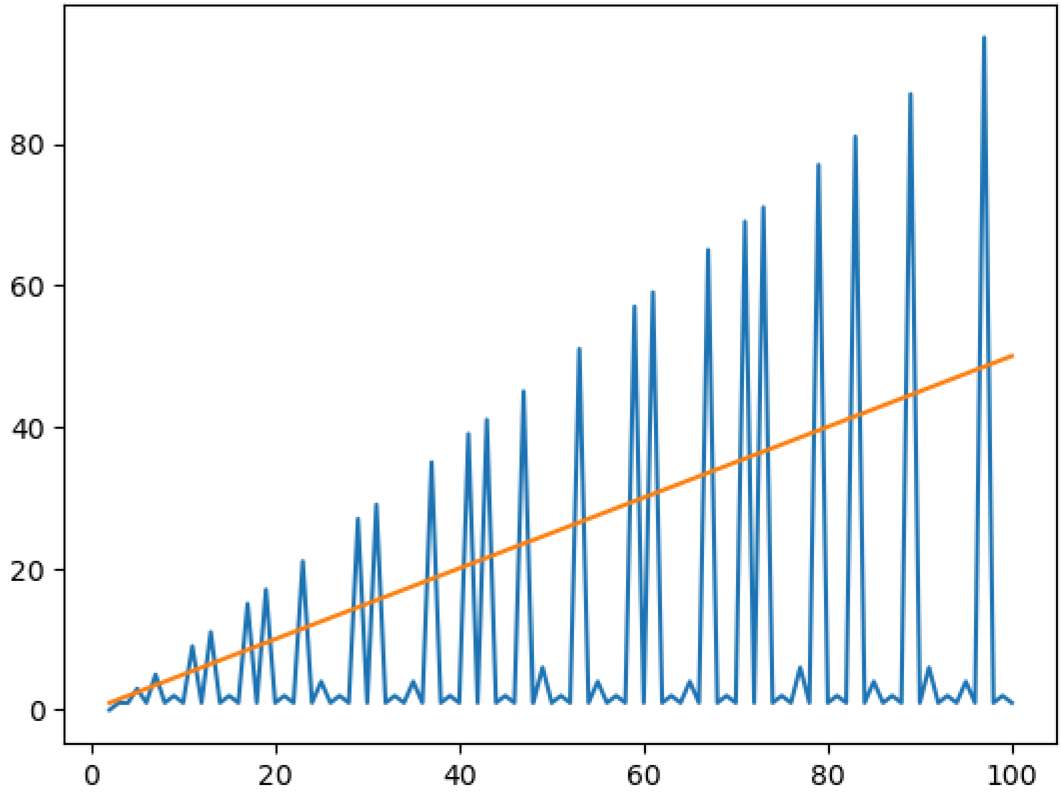
The mapping $C : E \to \mathbb{R}^+$ is called the cost function of the algorithm

Remark: it's a mapping defined on all possible inputs!

- it is way too complicated to describe it completely
- it is often impossible to compare two cost functions

In [11]:

```python
X = list(range(2,101))
Y = [is_prime_c(x) for x in X]
plt.plot(X,Y)
plt.plot(X,[x/2 for x in X])
plt.show() # compare the number of iterations of is_prime with x->x/2
```



Let E_n be the set of size-n inputs

We simplify the cost function by associating a unique cost to all the elements of E_n:

- for the worst case complexity we set $c_n = \max_{E \in E_n} C(E)$
- for the average case complexity we set $c_n = \mathbb{E}_n[C] = \sum_{E \in E_n} \mathbb{P}(E) \cdot C(E)$

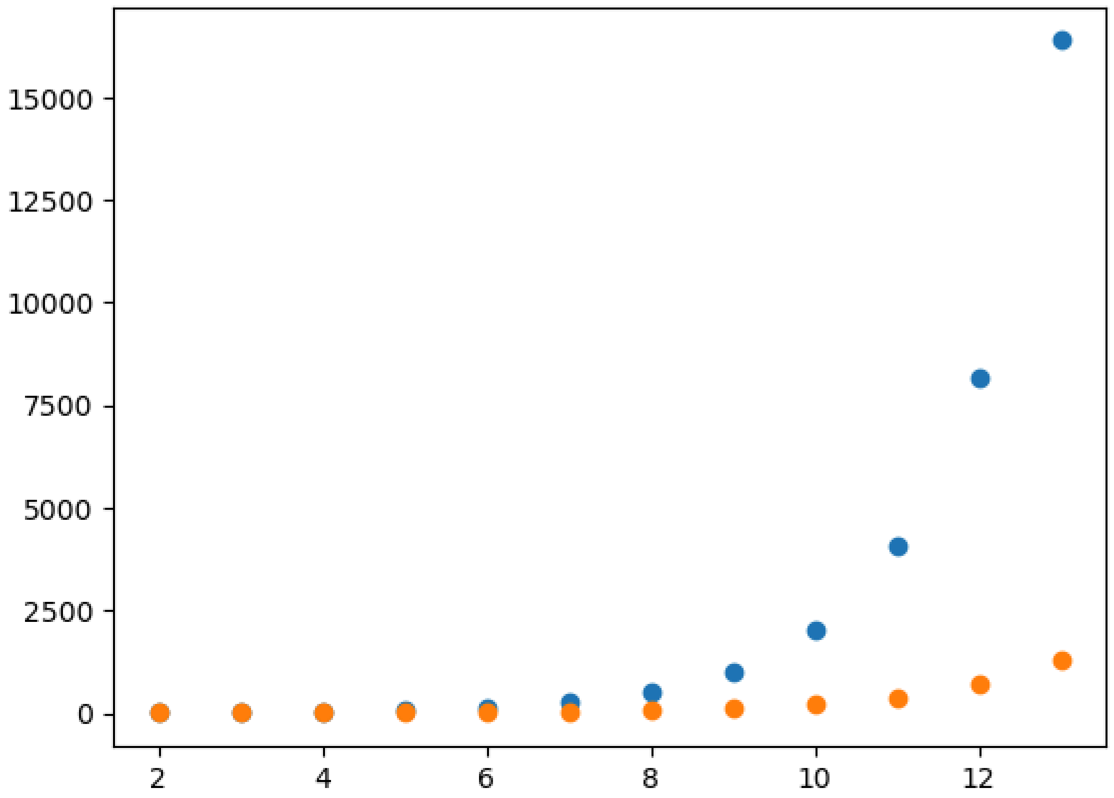
Remark: by choosing $\mathbb{P}(E) = \frac{1}{|E_n|}$, we obtain the uniform distribution, where all size-s inputs have same probability

c_n a **sequence**, which is a first simplification of the cost function

In [12]:
```
def list_k_bits(k):
    return list(range(2**k, 2**(k+1)))

print(list_k_bits(3))
```

```
[8, 9, 10, 11, 12, 13, 14, 15]
```

In [13]:
```
max_k = 13

worst_case = [max([is_prime_c(x) for x in list_k_bits(k)]) for k in range(2,max_k+1)]
average_case = [sum([is_prime_c(x) for x in list_k_bits(k)])/len(list_k_bits(k)) for k in ra

plt.plot(range(2,max_k+1), worst_case, 'o', linestyle="")
plt.plot(range(2,max_k+1), average_case, 'o', linestyle="")
plt.show()
```


15000
12500
10000
7500
5000
2500
0
2
4
6
8
10
12

Let E_n be the set of size-n inputs

In this course, we'll mostly focus on the worst case complexity, with

$$c_n = \max_{E \in E_n} C(E)$$

- **Pro:** we certify that all inputs in E_n use at most c_n resources
- **Con:** maybe there are not many inputs $E \in E_n$ such that $C(E) = c_n$

The worst-case complexity is the usual paradigm for analysing the efficiency of algorithm

Summary:
elementary instruction & memory acces in time
= size- inputs
we study the worst-case complexity

Summary:

elementary instruction & memory acces in time

 = size- inputs

we study the worst-case complexity

Still hard to compare two algorithms in general: if and are two algorithms for the same problem, and we find that

- the worst case complexity of is
- the worst case complexity of is

Which one is the more efficient?

We do not want complicated formula to describe the complexity! $\Leftarrow$ too difficult to compare them

So, we simplify again :

- we just aim at an asymptotic estimation of c_n (= when n is large)
- we just want to use simple functions: $n^\alpha, \log n, 2^n, \ldots$

We now need to introduce the notation $\mathcal{O}$ (and friends: Ω & Θ)

3. Asymptotic notations for sequences

Definition: let $(u_n)_{n\geq 0}$ and $(v_n)_{n\geq 0}$ be two positive sequences. We say that $u_n \in \mathcal{O}(v_n)$ when

$$\exists C > 0,\ u_n \leq C \times v_n$$

for n sufficiently large

- it is therefore an upper bound *up to a multiplicative constant*
- (formally, $\mathcal{O}(v_n)$ is a set of sequences)

Definition: let $(u_n)_{n\geq 0}$ and $(v_n)_{n\geq 0}$ be two positive sequences. We say that $u_n \in \mathcal{O}(v_n)$ when

$$\exists C > 0,\ u_n \leq C \times v_n$$

for n sufficiently large

- (multiplication by a constant) $\lambda u_n \in \mathcal{O}(u_n)$ for all λ
- (product -> product) if $u_n \in \mathcal{O}(v_n)$ and $u'_n \in \mathcal{O}(v'_n)$ then $u_n \times u'_n \in \mathcal{O}(v_n \times v'_n)$
- (sum -> max) if $u_n \in \mathcal{O}(v_n)$ then $u_n + v_n \in \mathcal{O}(v_n)$

Example:

$$(3n^2 + 2n + 4)(2n^3 + n) \Rightarrow \mathcal{O}(3n^2)\mathcal{O}(2n^3) \Rightarrow \mathcal{O}(n^2)\mathcal{O}(n^3) \Rightarrow \mathcal{O}(n^5)$$

Examples: which one are true?

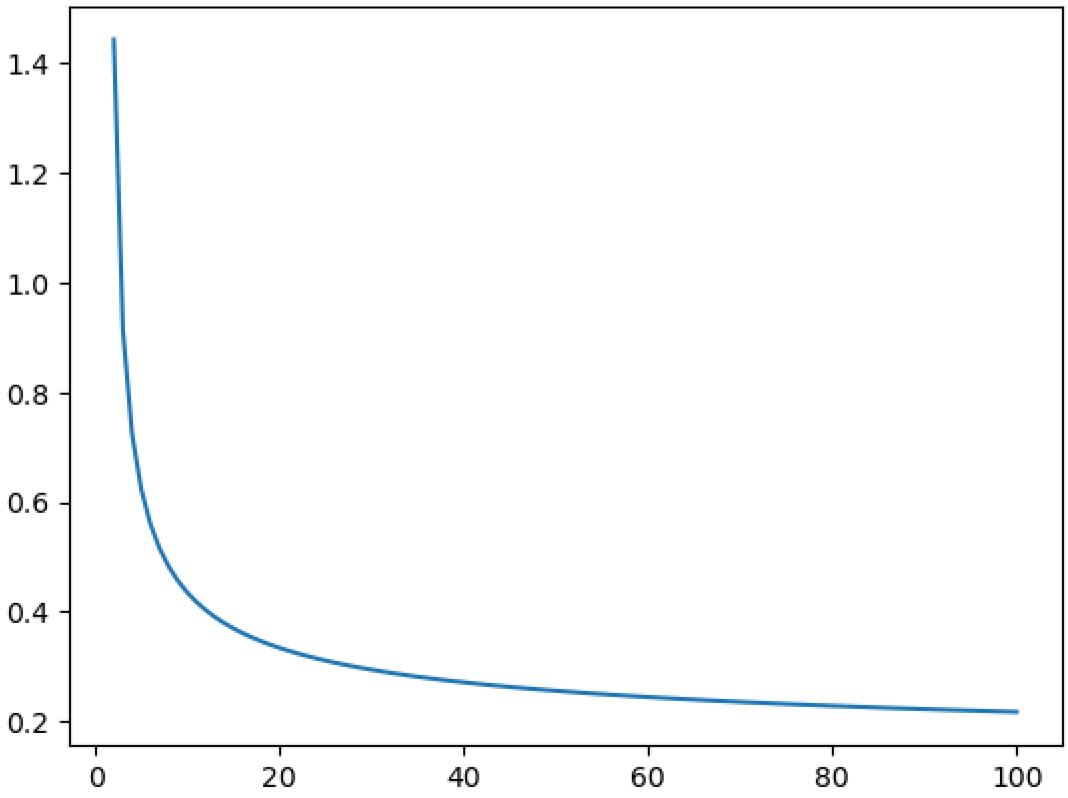
$$
\begin{array}{ccc}
n^2 \in \mathcal{O}(n^3); & n^3 \in \mathcal{O}(n^2); & n^2 \log n \in \mathcal{O}(n^3) \\
100n^2 + 99 \in \mathcal{O}(n^2); & 2^n \in \mathcal{O}(n^3); & 2^n \in \mathcal{O}(3^n) \\
3^n \in \mathcal{O}(2^n); & \sqrt{n} \in \mathcal{O}(n); & \sqrt{n} \in \mathcal{O}(\log n)
\end{array}
$$

Useful: $u_n \in \mathcal{O}(v_n)$ if and only if $\frac{u_n}{v_n}$ is bounded

$$n^2 \in \mathcal{O}(n^3); \quad n^3 \notin \mathcal{O}(n^2); \quad n^2 \log n \in \mathcal{O}(n^3)$$
$$100n^2 + 99 \in \mathcal{O}(n^2); \quad 2^n \notin \mathcal{O}(n^3); \quad 2^n \in \mathcal{O}(3^n)$$
$$3^n \notin \mathcal{O}(2^n); \quad \sqrt{n} \in \mathcal{O}(n); \quad \sqrt{n} \notin \mathcal{O}(\log n)$$

In [14]:

```
from math import cos
f, g, h = lambda n:n**2, lambda n:n**3, lambda n:(n**2)*log(n)
draw_curve(2, 100, lambda n: f(n)/h(n))
```

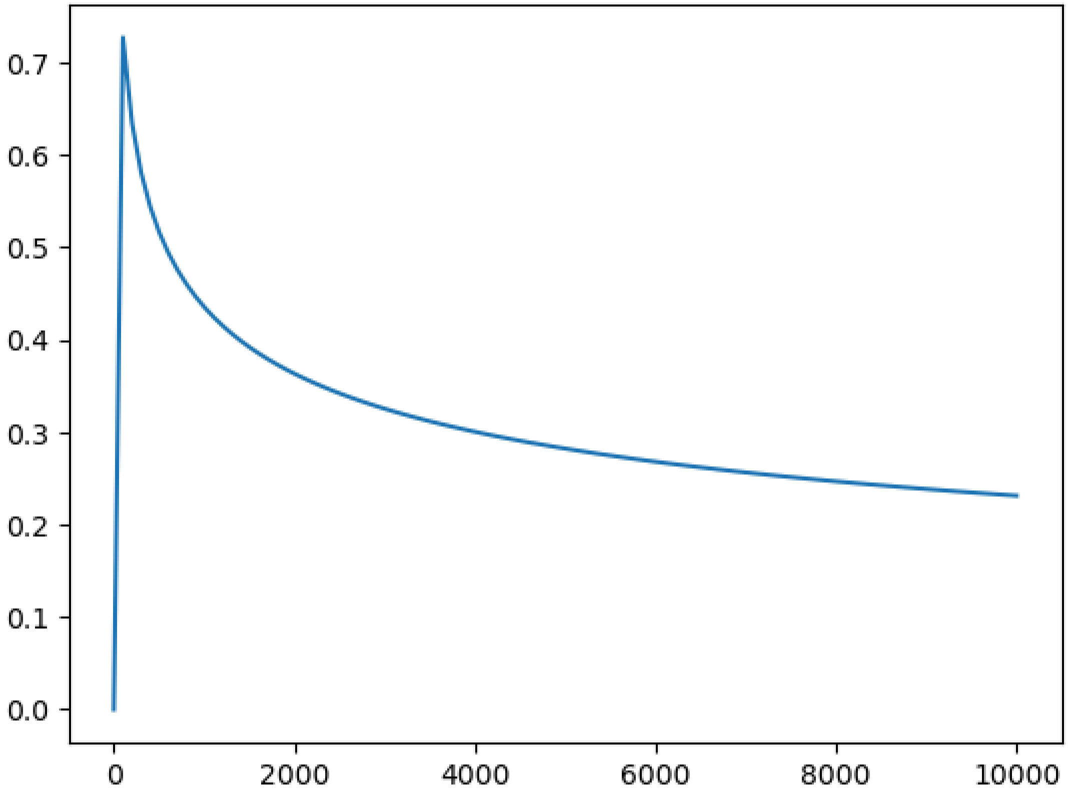


Refresher: the function $\ln x$ and e^x, and their binary versions $\log_2 x$ and 2^x

- we have $y = \ln x \Leftrightarrow x = e^y$ and $y = \log_2 x \Leftrightarrow x = 2^y$
- $\log_2 x = \frac{\ln x}{\ln 2}$
- $\log n \in \mathcal{O}(n^\alpha)$ for all $\alpha > 0$: the function $\log$ increases very slowly
- $n^\alpha \in \mathcal{O}(e^n)$ for all $\alpha > 0$: the function $\exp$ increases very quickly

In [15]:

```
f = lambda n:log(n)/n**1
g = lambda n:log(n)/n**2
h = lambda n:log(n)/n**.4

draw_curve(1,10000,h)
```



Definition: we write $u_n \in \Omega(v_n)$ when

$$\exists c > 0,\ u_n \geq c \times v_n$$

for n sufficiently large

Remark: $u_n \in \Omega(v_n) \Leftrightarrow v_n \in \mathcal{O}(u_n)$:

$$\underbrace{u_n \geq c\, v_n}_{u_n \in \Omega(v_n)} \Rightarrow v_n \leq \frac{1}{c} u_n \quad \text{and} \quad \underbrace{v_n \leq C\, u_n}_{v_n \in \mathcal{O}(u_n)} \Rightarrow u_n \geq \frac{1}{C} v_n$$

Definition: we write $u_n \in \Theta(v_n)$ when

$$\exists c, C > 0,\ c \times v_n \leq u_n \leq C \times v_n$$

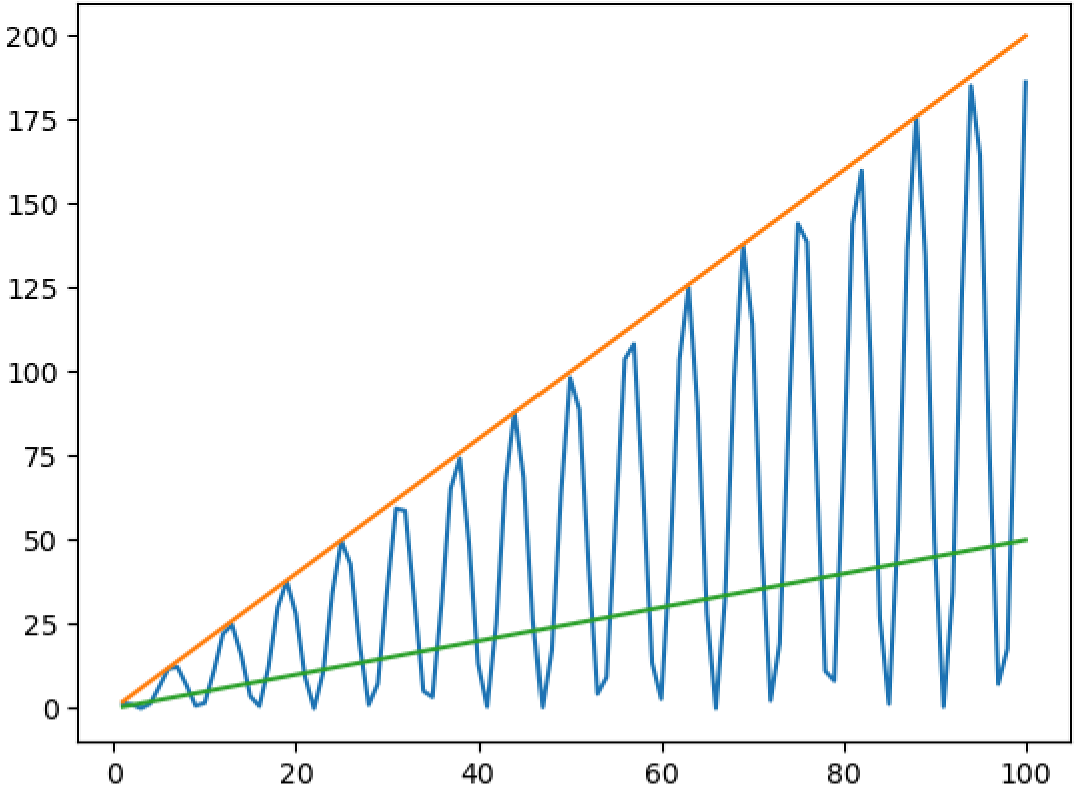
for n sufficiently large

Remark:

$$u_n \in \Theta(v_n) \Leftrightarrow \begin{cases} u_n \in \mathcal{O}(v_n) \\ u_n \in \Omega(v_n) \end{cases}$$

Examples: $n(3+\cos(n)) \in \Theta(n)$ but $n(1+\cos(n)) \notin \Theta(n)$

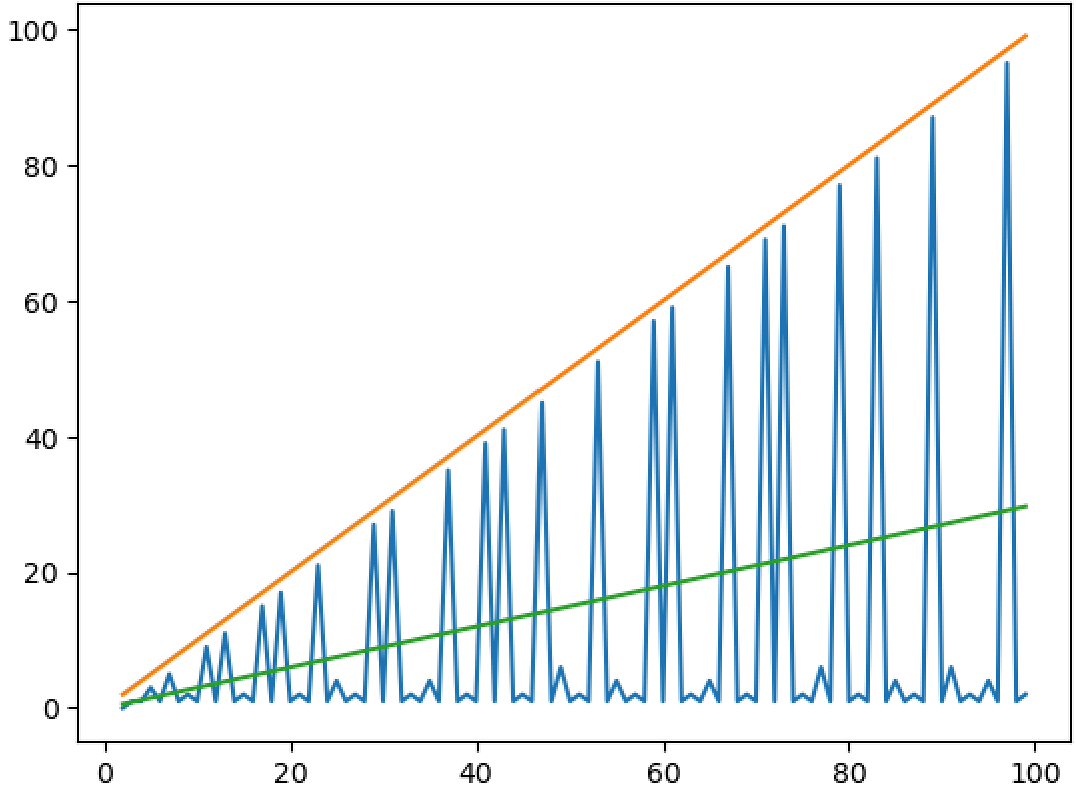
In [12]:
```
f, g = lambda n: n*(3+cos(n)), lambda n: n*(1+cos(n))

#draw_curves(1,100,[f, lambda n: 4*n, lambda n: 2*n])
draw_curves(1,100,[g, lambda n: 2*n, lambda n: .5*n])
```



The running time of `is_prime` is in $\mathcal{O}(n)$ but not in $\Omega(n)$, hence not in $\Theta(n)$

In [13]:

```
lx = list(range(2,100))
plt.plot(lx,[is_prime_c(n) for n in lx])
plt.plot(lx,[n for n in lx])
plt.plot(lx,[.3*n for n in lx])
plt.show()
```



Summary:

- we measure the amount of resources $C(E)$ required by the algorithm for the input E
- we group the element of E_n together by taking the worst case

$$c_n = \max_{E \in E_n} C(E)$$

- we approximate c_n asymptotically, using the notation $\mathcal{O}$ (or Θ if possible) and functions of the form n^α, $\log n$, β^n, $\ldots$

Why is it relevant?

We do make a lot of approximation, but we want theoretical result taht do not depend on:

- the processor's speed
- the programming language
- the efficiency of the compiler
- etc.

The information given by the complexity is the order of growth of the running time, how the running time scales with n.

Example: $c_n = \Theta(n)$ means that if we double n, the running time rougthly double too.

In [16]:

```
def nb_a(u):
    c = 0
    for x in u:
        if x == 'a':
            c += 1
    return c

def rand_word(n):
    return "".join([choice(['a','b']) for _ in range(n)])

n, t = 4*1000, time()
for _ in range(10**3): nb_a(rand_word(n))
print("temps : ", time()-t)
```

```
temps :  1.8199927806854248
```

The information given by the complexity is the order of growth of the running time, how the running time scales with n.

Example: $c_n = \Theta(n^2)$ means that if we double n, the running time is rougthly multiplied by 4:

$$(2n)^2 = 4n^2$$

In [18]:

```
def doublons(T):
    c = 0
    for i in range(len(T)):
        for j in range(i+1, len(T)):
            if T[i] == T[j]: c+=1
    return c


n, t = 2*200, time()
for _ in range(10**3): doublons([choice(range(10)) for _ in range(n)])
print("temps : ", time()-t)
```

```
temps :  5.213152885437012
```

The information given by the complexity is the order of growth of the running time, how the running time scales with n.

Example: $c_n = \Theta(\log n)$ means that if we double n, the running time is the same plus some constant:

$$\log(2n) = \log n + \log 2$$

In [31]:

```
def power(x,n):
    r = 1
    while n > 0:
        if n%2 == 1: r *= x
        x, n = x**2, n//2
    return r

n = 1000
t = time()
for _ in range(10**6): power(1.001, n)
print("temps pour ",n,":", time()-t)
t = time()
for _ in range(10**6): power(1.001, 2*n)
print("temps pour 2x",n,":", time()-t)
t = time()
for _ in range(10**6): power(1.001, 4*n)
print("temps pour 4x",n,":", time()-t)
t = time()
for _ in range(10**6): power(1.001, 8*n)
print("temps pour 8x",n,":", time()-t)
```

```
temps pour  1000 : 1.5708708763122559
temps pour 2x 1000 : 1.6300699710845947
temps pour 4x 1000 : 1.7099509239196777
temps pour 8x 1000 : 1.8543469905853271
```

4. Computing the complexity of iterative algorithms

Principle: look for the instruction that is executed the greater number of times

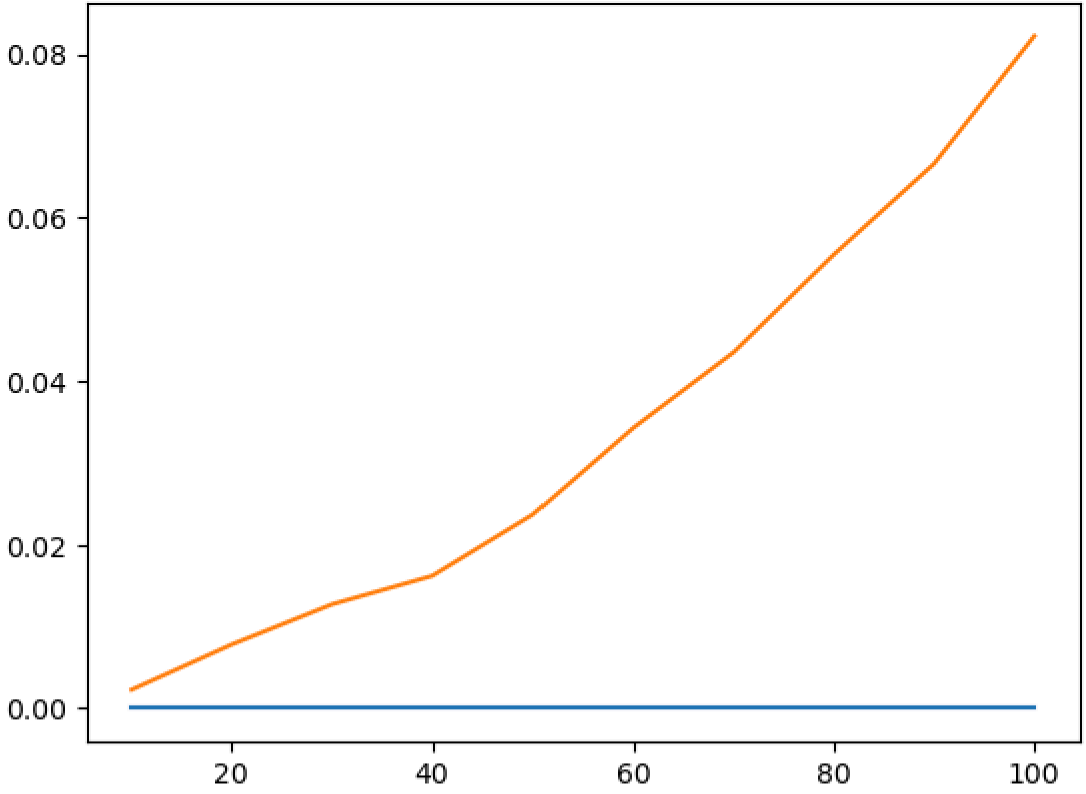
In [20]:

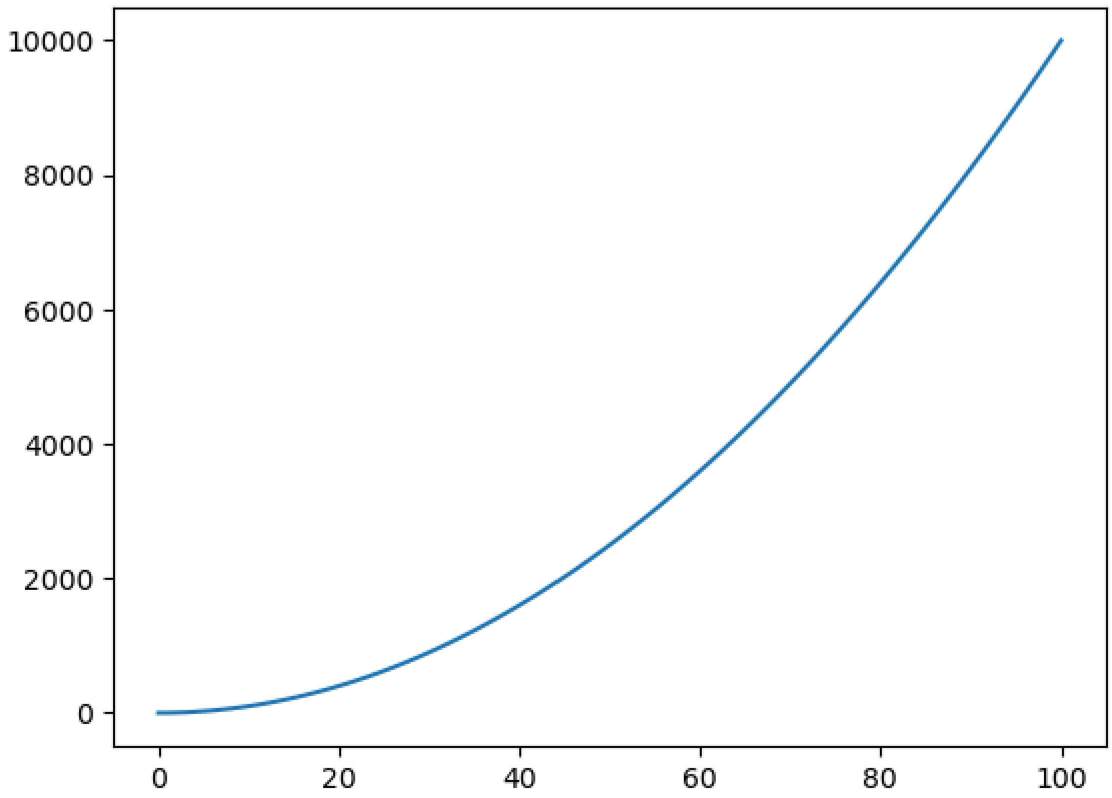
```
def bubble(T):
    for i in range(len(T)):
        for k in range(len(T)-1):
            if T[k] > T[k+1]:
                T[k],T[k+1] = T[k+1],T[k]
```

- the if instruction is the most executed, it is performed rougthly n^2 times for an array of size n
- the complexity is $\mathcal{O}(n^2)$

In [21]:

```
def bubble_random(n):
    T = [random() for _ in range(n)]
    bubble(T)
abscisses = list(range(10,101,10))
draw_time_curve(abscisses, bubble_random, 100)
draw_curve(0,100,lambda x:x**2)
```




10000
8000
6000
4000
2000
0
0
20
40
60
80
100

Warning: it is mathematically correct to say that `BubbleSort` is in $\mathcal{O}(n^{100})$, since $\mathcal{O}$ is just an upper bound

In [34]:
```
def bubble(T):
    for i in range(len(T)):
        for k in range(len(T)-1):
            if T[k] > T[k+1]:
                T[k],T[k+1] = T[k+1],T[k]
```

- but not acceptable in computer science: we want accurate informations

it is the same as, if asked for an integer greater than π*, one reply* $1.000.000$*; it is correct, but it is more relevant to answer* 4

When possible, it would be better to describe the complexity of an algorithm using the notation Θ, which is more precise.

But:

- in practice, people tend to use $\mathcal{O}$ for Θ
- if in a programming language (or software) documentation a $\mathcal{O}$ is given, it is usually the tightest known upper bound
- very often, it is in fact a Θ

$\Rightarrow$ `BubbleSort` is in $\Theta(n^2)$

Improvement of `BubbleSort`: at step i, the last i element are at their correct place, no need to scan them.

In [24]:
```
def bubble_optimized(T):
    for i in range(len(T)):
        for k in range(len(T)-1-i):
            if T[k] > T[k+1]:
                T[k],T[k+1] = T[k+1],T[k]
```

At step i, the instruction if is done $n-1-i$ times:

$$n-1\ +\ n-2\ +\ldots+\ 1=\sum_{j=1}^{n-1} j=\frac{n(n-1)}{2}\in\Theta(n^2)$$

The improvement *doesn't* change the complexity

Such sums

$$\sum_{i=1}^{n} c_i$$

often occurs when analyzing complexity.

In [35]:

```
def eval_polynom(P, x):
    r = 0
    for i in range(len(P)):
        r += P[i]*power(x,i) # O(log i)
    return r

P = [1,2,3] # P = 1 + 2x + 3x^2
print(eval_polynom(P,1), eval_polynom(P,2))
```

```
6 17
```

Theorem if $\alpha, \beta \geq 0$, then

$$\sum_{i=1}^{n} i^{\alpha}(\log i)^{\beta} \in \Theta(n^{\alpha+1}(\log n)^{\beta})$$

- this gives another proof that the improved `BubbleSort` is still in $\Theta(n^2)$:

$$\underbrace{\sum_{i=1}^{n} i}_{\alpha=1,\beta=0} \in \Theta(n^2)$$

- For `eval_polynom`, we get

$$\underbrace{\sum_{i=1}^{n} \log i}_{\alpha=0,\beta=1} \in \Theta(n \log n)$$

there is a $\mathcal{O}(n)$ algorithm for polynomial evaluation, called Horner's méthod

5. Computing the complexity of recursive algorithms

In [36]:

```python
def factorial(n):
    if n <= 1: return 1
    return n * factorial(n-1)

print([factorial(i) for i in range(10)])
```

```
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

In [37]:

```python
def fibo(n):
    if n <= 1: return n
    return fibo(n-1) + fibo(n-2)

print([fibo(i) for i in range(10)])
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Principle: estimate the number of recursive calls, and evaluate the cost of each call

In [38]:

```
NB = 0
def factorial(n):
    global NB
    NB += 1
    if n <= 1: return 1
    return n * factorial(n-1)

factorial(15)
print("number of calls =", NB)
```

```
number of calls = 15
```

Principle: estimate the number of recursive calls, and evaluate the cost of each call

In [39]:

```
NB = 0
def fibo(n):
    global NB
    NB += 1
    if n <= 1: return n
    return fibo(n-1) + fibo(n-2)

fibo(35)
```

Out[39]:

```
9227465
```

Principle: estimate the number of recursive calls, and evaluate the cost of each call

Define $A(n) =$ number of calls in the worst case for a size-n input, then find an induction formula for $A(n)$ by studying the algorithm

In [58]:
```
def factorial(n):
    if n <= 1: return 1
    return n * factorial(n-1)
```

We have

$$A(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 1 + A(n-1) & \text{sinon} \end{cases}$$

In [59]:

```
def fibo(n):
    if n <= 1: return n
    return fibo(n-1) + fibo(n-2)
```

we have

$$A(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 1 + A(n-1) + A(n-2) & \text{otherwise} \end{cases}$$

In [ ]:
```
def power(x,n):
    if n == 0: return 1
    if n%2 == 1: return power(x,n//2)**2 * x
    else: return power(x,n//2)**2

print([power(2,i) for i in range(11)])
#draw_time_curve(list(range(1,10))+list(range(10,1001,10)), lambda n:power(1.01,n), 10000)
```

on a

$$A(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 + A(\lfloor \frac{n}{2} \rfloor) & \text{sinon} \end{cases}$$

Theorem: if $A(n) = A(n-k) + \mathcal{O}(n^\alpha \log^\beta n)$ then

$$A(n) = \mathcal{O}(n^{\alpha+1} \log^\beta n)$$

In [60]:

```
def factorial(n):
    if n <= 1: return 1
    return n * factorial(n-1)
```

We have

$$A(n) = \underbrace{1}_{n^0} + A(n - \underbrace{1}_{k=1})$$

Thus, $A(n) = \mathcal{O}(n)$

Theorem: if $A(n) \geq A(n-k) + A(n-\ell)$ then

$$A(n) = \Omega(C^n),\ C > 1$$

Such an algorithm has exponential complexity (which is very bad)

In [355]:
```
def fibo(n):
    if n <= 1: return n
    return fibo(n-1) + fibo(n-2)
```

Since $A(n) = 1 + A(n-1) + A(n-2)$, the complexity is exponential

Master theorem $A(n) = a \cdot A(\frac{n}{b}) + \mathcal{O}(n^c)$ with $b > 1$ then

1. if $b^c < a$, then $A(n) = \Theta(n^{\log_b a})$

2. if $b^c = a$, then $A(n) = \Theta(n^c \log n)$

3. if $b^c > a$, then $A(n) = \Theta(n^c)$

Very useful for divide and conquer algorithms

Master theorem: $A(n) = a \cdot A(\frac{n}{b}) + \mathcal{O}(n^c)$ with $b > 1$ then

Case 2: if $b^c = a$, then $A(n) = \Theta(n^c \log n)$

In [357]:

```
def power(x,n):
    if n == 0: return 1
    if n%2 == 1: return power(x,n//2)**2 * x
    else: return power(x,n//2)**2
```

We have $A(n) = A(\frac{n}{2}) + \mathcal{O}(1)$ thus $a = 1, b = 2$ et $c = 0$

We have $b^c = 2^0 = 1 = a$, hence it is case 2 $A(n) = \Theta(\log n)$

Master theorem: $A(n) = a \cdot A(\frac{n}{b}) + \mathcal{O}(n^c)$ with $b > 1$ then

Case 2: if $b^c = a$, then $A(n) = \Theta(n^c \log n)$

In [358]:
```
def merge_sort(T):
    if len(T) < 2 : return T
    L, R = merge_sort(T[:len(T)//2]), merge_sort(T[len(T)//2:])
    return merge_sort(L,R) # in O(n), not implemented!
```

We have $A(n) = 2A(\frac{n}{2}) + \mathcal{O}(n)$ then $a = 2, b = 2$ et $c = 1$

We have $b^c = 2^1 = 2 = a$, hence it is case 2 again $A(n) = \Theta(n \log n)$

Master theorem $A(n) = a \cdot A(\frac{n}{b}) + \mathcal{O}(n^c)$ with $b > 1$ then

1. if $b^c < a$, then $A(n) = \Theta(n^{\log_b a})$

2. if $b^c = a$, then $A(n) = \Theta(n^c \log n)$

3. if $b^c > a$, then $A(n) = \Theta(n^c)$