Complexité des problèmes

cyril.nicaud@univ-eiffel.fr

Déroulé du cours :

- 6 séances de cours
- 6+ séances de TD
- évaluation = un examen de 2h en janvier (feuille A4 recto-verso autorisée)

Cours 1 : Rappels d'algorithmique

1. Problèmes

Définition : Un **problème (algorithmique)** est défini par la caractérisation :

- des entrées (ou données) du problème
- du résultat (ou sortie) attendu pour chaque entrée, et/ou la transformation effectuée sur les entrées

Exemple. Le problème Sort consiste à trier un tableau d'éléments comparables. L'entrée est un tableau T et on peut décliner plusieurs variantes du problème en spécifiant le résultat / la transformation :

- *transformer* T de façon à ce que ses éléments soient dans l'ordre croissant
- *calculer* un nouveau tableau S qui contient les mêmes éléments que T en ordre croissant
- *calculer* la permutation des indices de T telle que si on suit les indices dans cet ordre, on voit les éléments de T en ordre croissant

2. Algorithmes

> ***Définition :*** *un **algorithme** est un procédé automatique pour résoudre un problème en un nombre fini d'étapes*

L'addition telle qu'apprise en primaire est un algorithme :

	4	3	7	8
+	1	6	4	1
---	---	---	---	---
	6	0	1	9

> ***Définition :*** *un **algorithme** est un procédé automatique pour résoudre un problème en un nombre fini d'étapes*

In [4]:

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

T = [2,3,5,7,11]
print(find(7,T))
print(find(8,T))
```

```
True
False
```

> ***Important :*** *un problème peut avoir plusieurs solutions (=algorithmes)*

Exemple : recherche d'un élément dans un tableau *trié*

- on peut utiliser l'algorithme `find(x,T)` précédent
- on peut faire une dichotomie pour utiliser le fait que le tableau est trié

Les deux méthodes sont valides, au sens où elles résolvent le problème

In [5]:

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

T = [2,3,5,7,11]
print(binary_search(7,T))
print(binary_search(8,T))
```

```
True
False
```

Quand on a plusieurs algorithmes pour résoudre un même problème on peut :

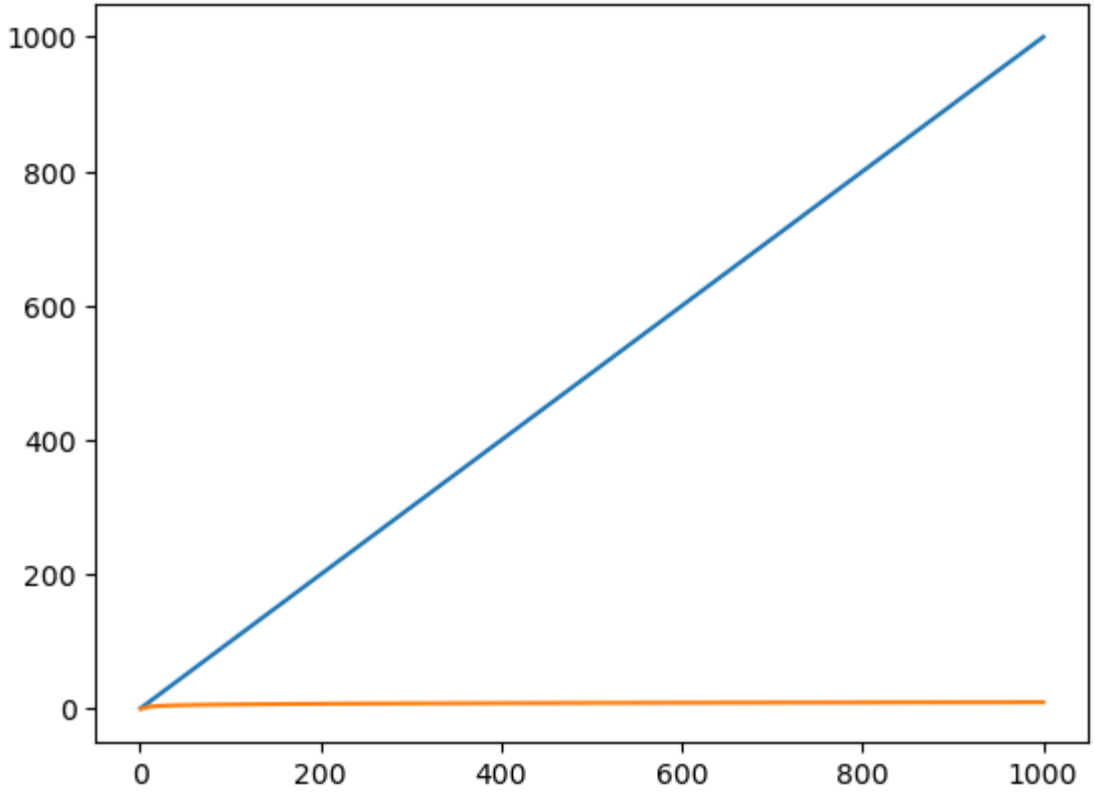
- les comparer expérimentalement si on a des benchmarks
- les comparer théoriquement en étudiant leur complexité

Exemples de complexité :

- `find` est en $\mathcal{O}(n)$
- `binary_search` est en $\mathcal{O}(\log n)$

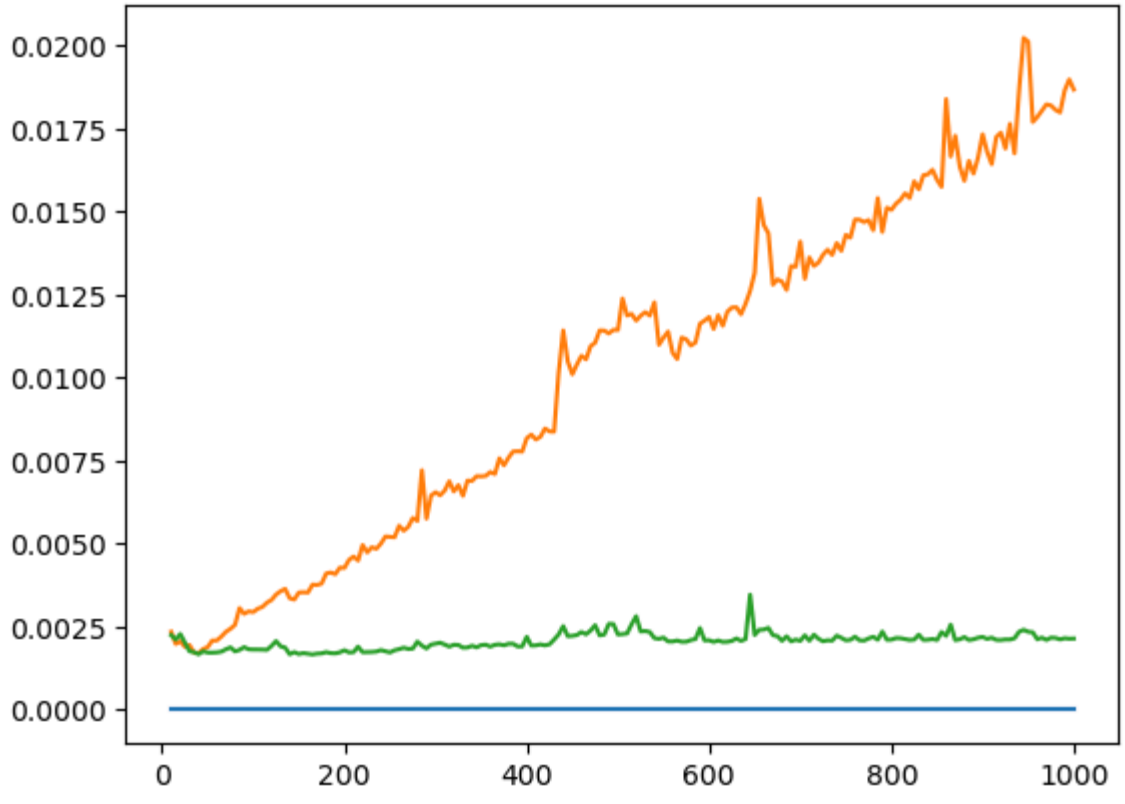
In [42]:

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

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



In [7]:

```
time_dicho, time_find, iterations = [], [], 1000
L = range(10, 1001, 5)
for n in L: # n de 10 à 1000
    T = list(range(n))
    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
        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]) # en bleu l'axe des x
plt.plot(L,time_find)        # en orange le temps de find
plt.plot(L,time_dicho)       # en vert le temps de binary_search
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

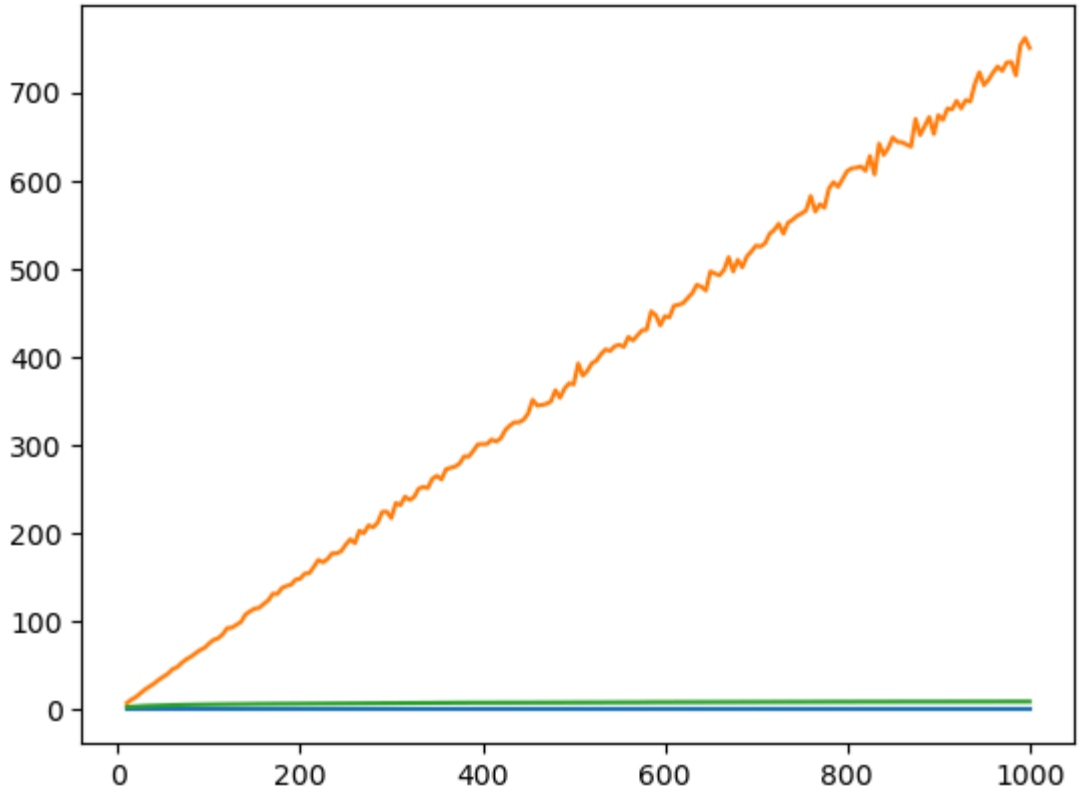
Mesurer le temps d'exécution = compliqué : on peut utiliser des compteurs

In [8]:

```python
def find_c(x,T):
    c = 0
    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 [9]:

```
iterations = 1000
L = list(range(10, 1001, 5))
c_dicho, c_find = [0] * len(L), [0] * len(L)
for i in range(len(L)): # n de 100 à 2000
    T = list(range(L[i]))
    for _ in range(iterations):
        if random() < .5: x = randrange(0, L[i]) + 0.1 # x pas dedans
        else: x = randrange(0, L[i]) # x dedans
        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]) # en bleu l'axe des x
plt.plot(L,c_find)        # en orange le temps de find
plt.plot(L,c_dicho)       # en vert le temps de binary_search
plt.show()
```


700
600
500
400
300
200
100
0
0
200
400
600
800
1000

2. Complexité

> ***Définition :*** *étudier la complexité d'un algorithme c'est estimer la quantité de ressources dont elle a besoin*

Les ressources typiques :

- temps
- espace: quantité mémoire
- énergie
- . . .

Pour analyser la complexité il faut une notion de taille sur les entrées : quand on dit que `find` est en $O(n)$, n est la taille du tableau

Dans la plupart des cas la taille est définie naturellement :

- la longueur d'une chaîne
- le nombre de cases d'un tableau
- le nombre de maillons d'une liste
- ...

Attention : la seule difficulté est sur la taille d'un entier!

- pour des besoins courants (indices, nombres dans un tableau, numéro des sommets, etc) on peut considérer que sa taille est constante
- pour des algorithmes manipulant des grands nombres (cryptographie, etc) il faut considérer que c'est proportionnel à l'encodage du nombre : la taille de la représentation de n est $\approx \log_2 n$

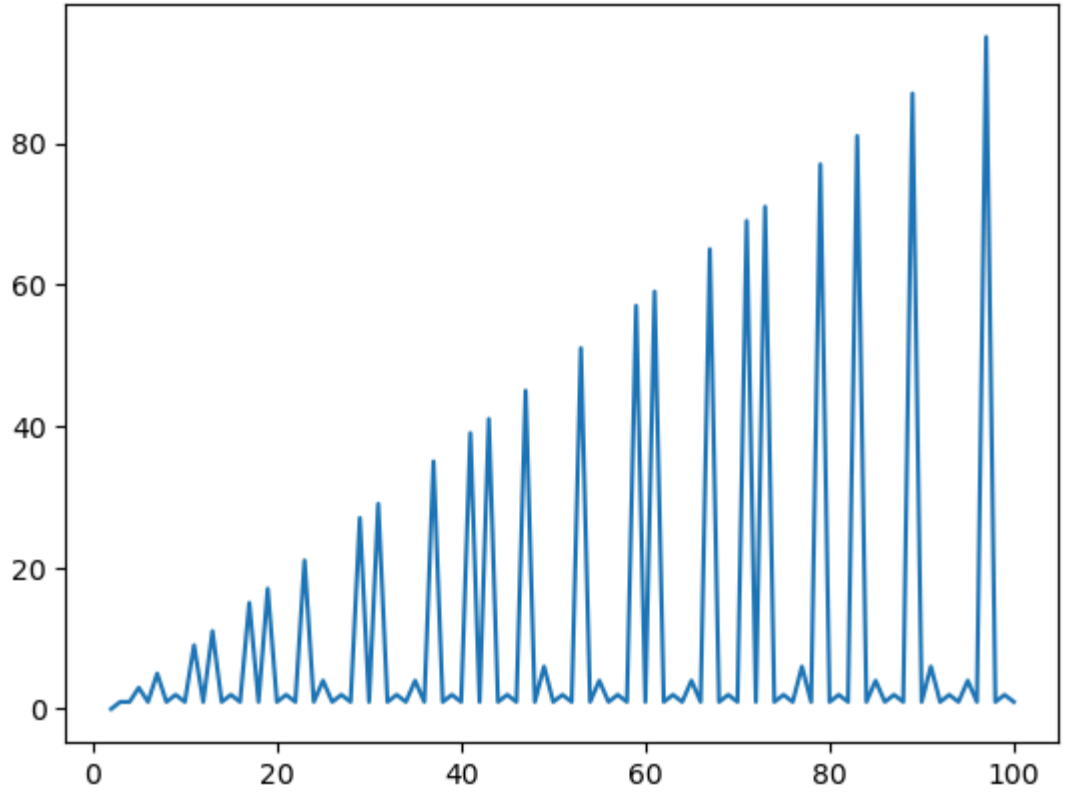
In [13]:

```
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 [14]:

```
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()
```



Si on veut faire un étude précise il faut également un modèle d'ordinateur

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

Ce n'est pas ce qui est exécuté par la machine !

- Python est interprété
- le C est compilé

L'ordinateur utilise du langage machine

Si on veut faire un étude précise il faut également un modèle d'ordinateur

- Machine de Turing (idée similaire aux automates finis)
- Random Access Machine (genre de processeur mathématique, avec un assembleur)
- . . .

Pour ce cours, on n'aura pas besoin d'aller à ce niveau de précision, on admet juste que même après interprétation/compilation :

- les instructions élémentaires se font en temps constant
- les accès mémoire se font en temps constant

> ***Définition :*** *pour une entrée E d'un algorithme, on note $C(E)$ la quantité de ressources (typiquement : temps, nombre d'instructions, ...) utilisée par l'algorithme pour traiter E.*

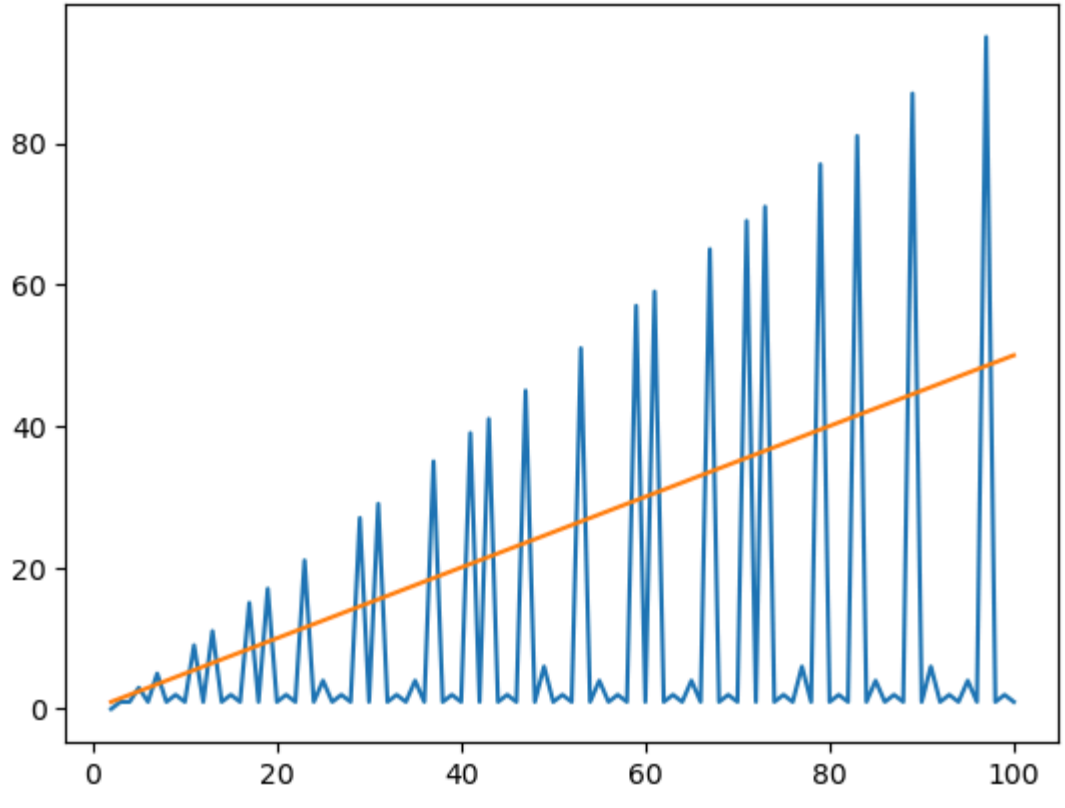
La fonction $C : E \to \mathbb{R}^+$ s'appelle la fonction de coût de l'algorithme

Remarque : c'est une fonction dont le domaine est *toutes* les entrées possibles :

- c'est beaucoup trop compliqué à décrire
- il est souvent impossible de comparer deux fonctions

In [15]:

```
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() # comparer le nombre d'itérations de is_prime avec x->x/2
           #(et encore l'entrée est juste un entier, pas un tableau ou un graphe)
```



On note E_n l'ensemble des entrées de taille n

On peut avoir une représentation simplifiée de la fonction de coût en n'associant qu'un seul coût à tous les éléments de E_n :

- dans le pire cas on choisit $c_n = \max_{E \in E_n} C(E)$
- en moyenne on choisit
  $c_n = \mathbb{E}_n[C] =$
  $\sum_{E \in E_n} \mathbb{P}(E) \cdot C(E)$

Si on choisit $\mathbb{P}(E) = \frac{1}{|E_n|}$ on la distribution uniforme, toutes les entrées de taille n ont même probabilité

c_n est une suite qui est une première simplification de la fonction de coût

In [18]:

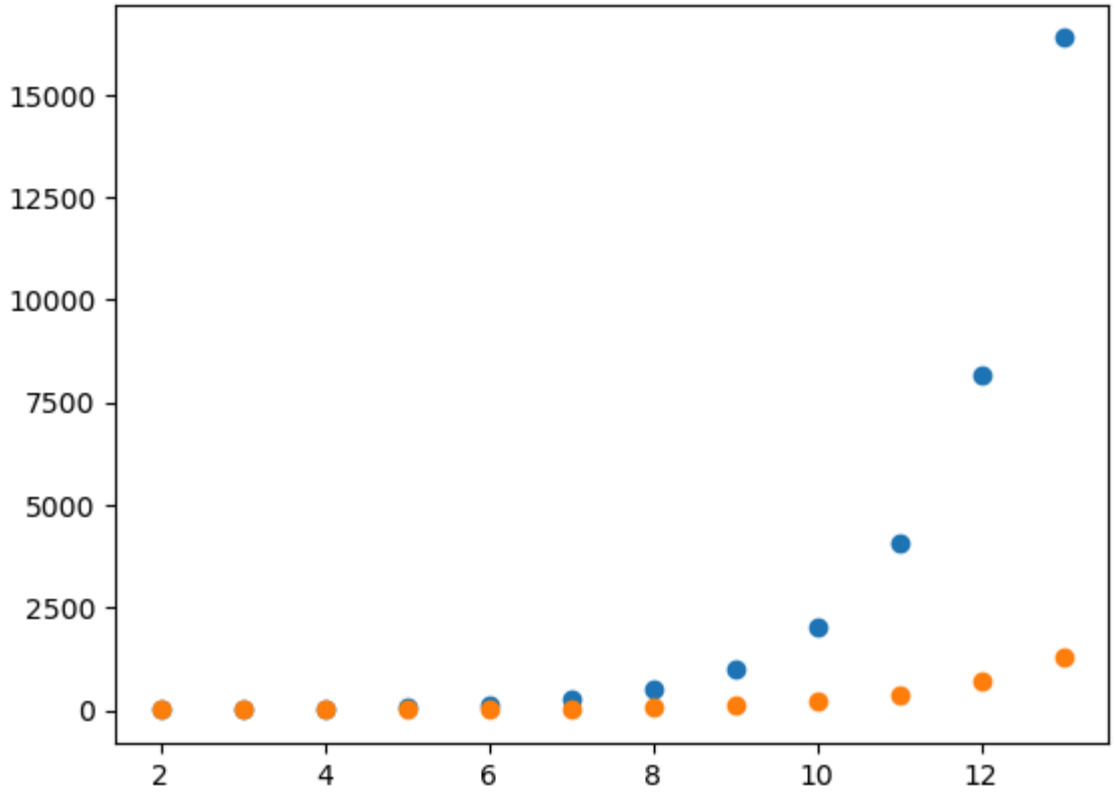
```
def list_k_bits(k):
    return list(range(2**k, 2**(k+1)))
list_k_bits(3)
```

Out[18]:

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

In [19]:

```
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

On note E_n l'ensemble des entrées de taille n

On s'intéressera principalement au pire cas avec

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

- Avantage : on certifie que tous les éléments de E_n utilise au plus c_n ressources
- Inconvénient : peut-être qu'il y a très peu d'entrées $E \in E_n$ pour lesquelles $C(E) = c_n$

> ***Important :*** *Le pire cas est le paradigme classique pour l'analyse d'algorithmes*

> ***Récapitulatif :***
> - *instructions et accès mémoire en temps* $\mathcal{O}(1)$
> - *on note* E_n *= entrées de taille* n
> - *on étudie la complexité dans le pire cas*

Reste difficile à comparer : si on a deux algorithmes $\mathcal{A}_1$ et $\mathcal{A}_2$ à comparer et qu'on trouve que

- la complexité pire cas de $\mathcal{A}_1$ est $\frac{n}{4^n}\binom{2n}{n}$
- la complexité pire cas de $\mathcal{A}_2$ est $\sum_{k=1}^{n} k \log k$

Quel est l'algorithme le plus efficace ?

On ne veut pas des formules compliquées pour les complexité $\Leftarrow$ difficile à comparer

On simplifie à nouveau :

- on ne cherche qu'à estimer la croissance asymptotique (= quand n est grand)
- on ne veut utiliser que des combinaisons de fonctions simples : $n^\alpha, \log n, 2^n, \ldots$

On va donc introduire la notation $\mathcal{O}$ (et ses cousines : Ω et Θ)

3. Notations asymptotiques pour les suites

> ***Définition :*** *soient* $(u_n)_{n\geq 0}$ *et* $(v_n)_{n\geq 0}$ *deux suites à valeurs positives. On dit que* $u_n \in \mathcal{O}(v_n)$ *quand*
>
> $$\exists C > 0,\ u_n \leq C \times v_n$$
>
> *pour tout* n *assez grand*

- il s'agit donc d'une majoration *à constante multiplicative près*
- (techniquement, $\mathcal{O}(v_n)$ est un ensemble de suites)

> ***Définition :*** *soient* $(u_n)_{n\geq 0}$ *et* $(v_n)_{n\geq 0}$ *deux suites à valeurs positives. On dit que* $u_n \in \mathcal{O}(v_n)$ *quand*
>
> $$\exists C > 0,\ u_n \leq C \times v_n$$
>
> *pour tout* n *assez grand*

- (multiplication par constante) $\lambda u_n \in \mathcal{O}(u_n)$ pour tout λ
- (produit -> produit) si $u_n \in \mathcal{O}(v_n)$ et $u'_n \in \mathcal{O}(v'_n)$ alors $u_n \times u'_n \in \mathcal{O}(v_n \times v'_n)$
- (somme -> max) si $u_n \in \mathcal{O}(v_n)$ alors $u_n + v_n \in \mathcal{O}(v_n)$

Exemple :

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

Exemples :

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

Propriété : $u_n \in \mathcal{O}(v_n)$ *ssi* $\frac{u_n}{v_n}$ *est bornée*

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

In [ ]:

```
from math import cos,log
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))
```

Rappels : les fonctions $\ln x$ et e^x, et leur version binaire $\log_2 x$ et 2^x

- on a $y = \ln x \Leftrightarrow x = e^y$ et $y = \log_2 x \Leftrightarrow x = 2^y$
- $\log_2 x = \frac{\ln x}{\ln 2}$
- $\log n \in \mathcal{O}(n^\alpha)$ pour tout $\alpha > 0$: la fonction $\log$ croît très lentement
- $n^\alpha \in \mathcal{O}(e^n)$ pour tout $\alpha > 0$: la fonction $\exp$ croît très vite
- $\log(ab) = \log(a) + \log(b)$
- $\exp(a + b) = \exp(a) \times \exp(b)$

In [ ]:

```
f = lambda n:log(n)/n**1
g = lambda n:log(n)/n**2
h = lambda n:log(n)/n**.5

draw_curve(1,10000,h)
```

> ***Définition :*** *on dit que* $u_n \in \Omega(v_n)$ *quand*
>
> $$\exists c > 0,\ u_n \geq c \times v_n$$
>
> *pour* n *suffisamment grand*

- on a $u_n \in \Omega(v_n) \Leftrightarrow v_n \in \mathcal{O}(u_n)$:

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

Remarque : on va *beaucoup* utiliser Ω dans ce cours

Définition : *on dit que* $u_n \in \Theta(v_n)$ *quand*

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

pour n *suffisamment grand*

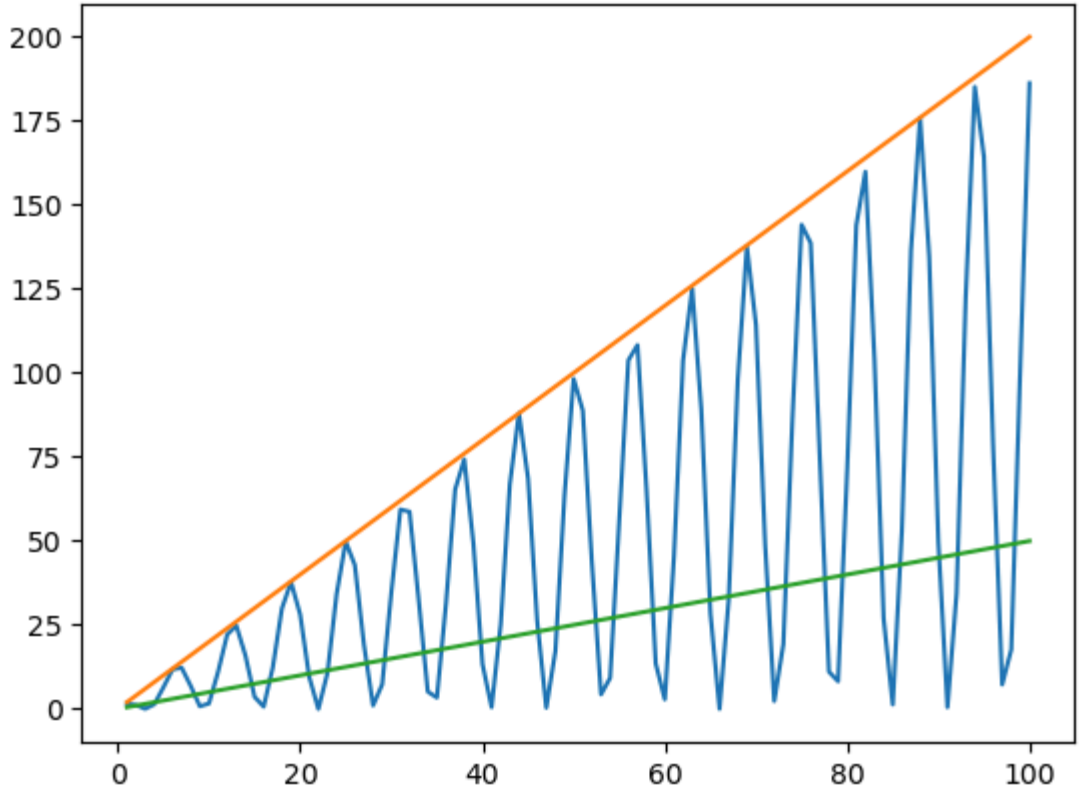
- on a

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

$n(3+\cos(n)) \in \Theta(n)$ mais $n(1+\cos(n)) \notin \Theta(n)$

In [21]:

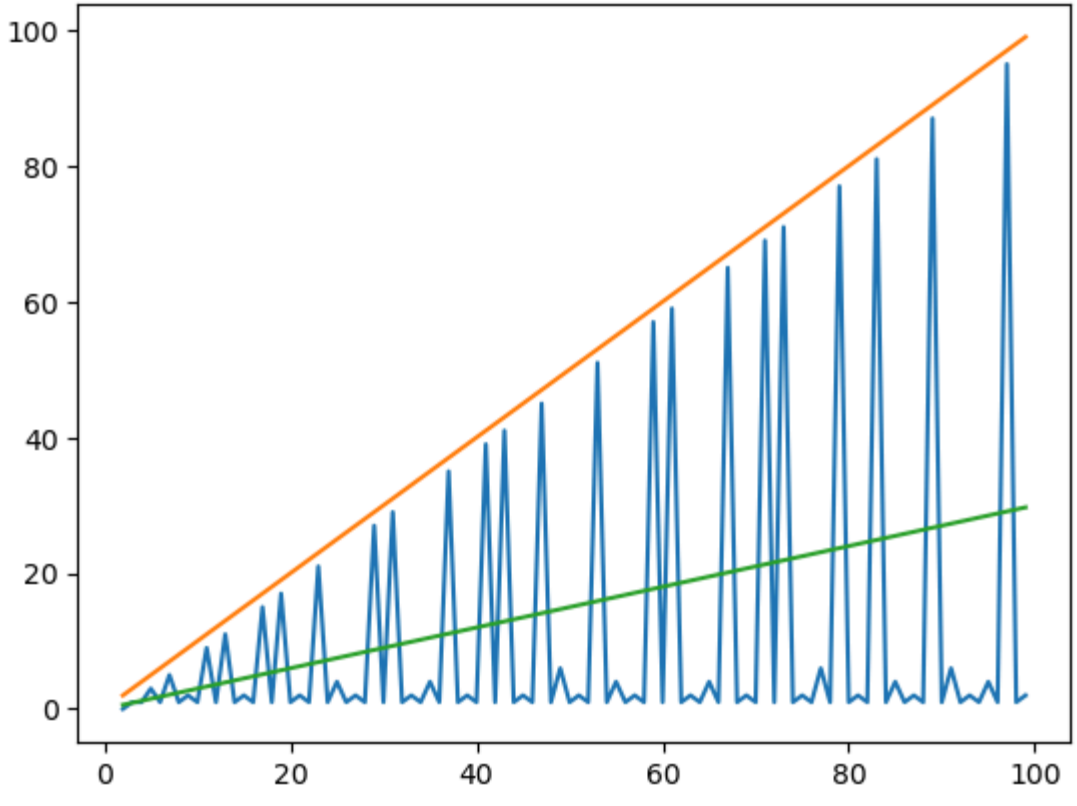
```
f, g = lambda n: n*(3+cos(n)), lambda n: n*(1+cos(n))

#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])
```



La suite du nombre d'itérations de `is_prime` est en $\mathcal{O}(n)$ mais pas dans $\Omega(n)$, donc pas dans $\Theta(n)$

In [22]:

```
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()
```



Récapitulatif :

- On mesure la quantité de ressources $C(E)$ pour exécuter l'algo sur l'entrée E
- On regroupe par entrées E_n de même tailles n, et on prend le pire cas

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

- On approxime asymptotiquement la suite c_n en utilisant la notation $\mathcal{O}$ (ou Θ si on peut) et des combinaisons de n^α, $\log n$, β^n, ...

Pourquoi c'est pertinent ?

On approxime beaucoup mais on veut des résultats *algorithmiques* qui ne dépendent pas :

- de la vitesse de l'ordinateur
- du langage utilisé
- de l'efficacité du compilateur / interpréteur
- etc.

L'information que l'on donne c'est sur la croissance du temps de calcul

- si $c_n = \Theta(n)$: si on double la taille de l'entrée, on double le temps

In [25]:
```
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 = 4000, time()
for _ in range(10**3): nb_a(rand_word(n))
print("temps : ", time()-t)
```

```
temps :  1.9503402709960938
```

L'information que l'on donne c'est sur la croissance du temps de calcul

- si $c_n = \Theta(n^2)$: si on double la taille de l'entrée, le temps est multiplié par 4 car

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

In [28]:

```
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

#print(doublons([3,4,4,1,4,3]))

n, t = 400, time()
for _ in range(10**3): doublons([choice(range(10)) for _ in range(n)])
print("temps : ", time()-t)
```

```
temps :  5.02341103553772
```

L'information que l'on donne c'est sur le taux de croissance du temps de calcul

- si $c_n = \Theta(\log n)$: si on double la taille de l'entrée, on rajoute un temps constant

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

In [29]:

```
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)
```

```
temps pour  1000 : 1.4149441719055176
temps pour 2x 1000 : 1.5591228008270264
temps pour 4x 1000 : 1.6547060012817383
```

4. Complexité des algorithmes itératifs

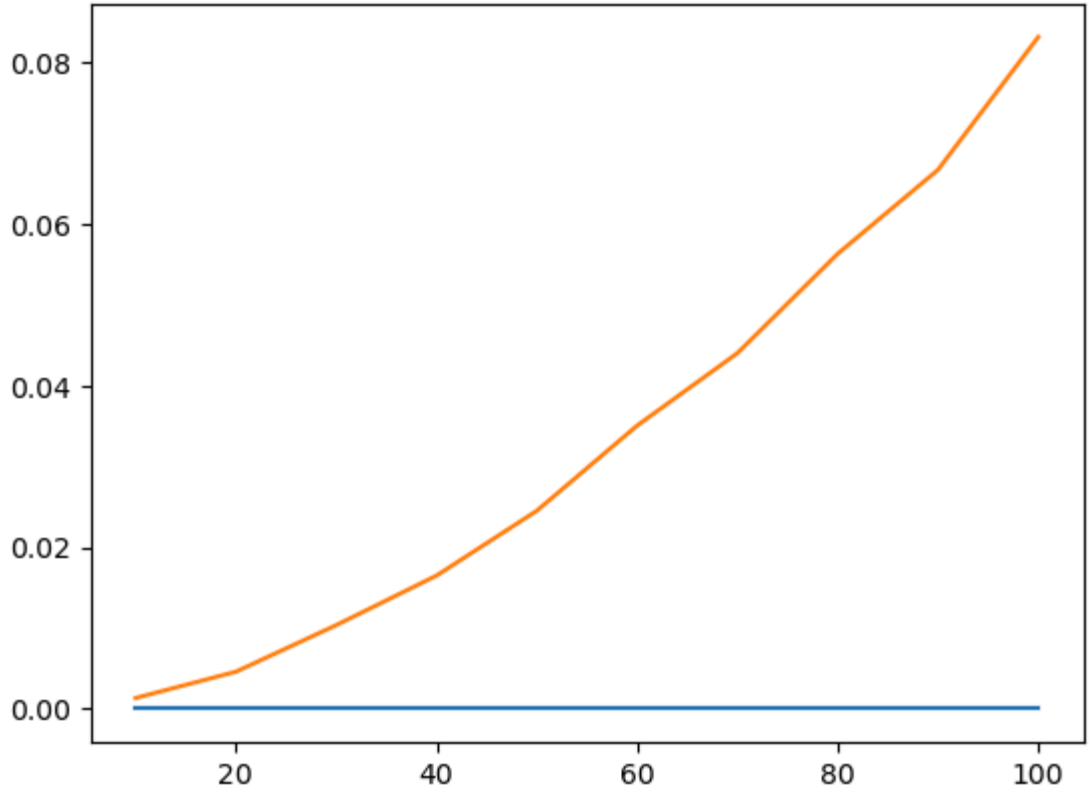
> ***Principe :*** *chercher la ligne la plus effectuée et majorer le nombre de fois qu'elle est effectuée*

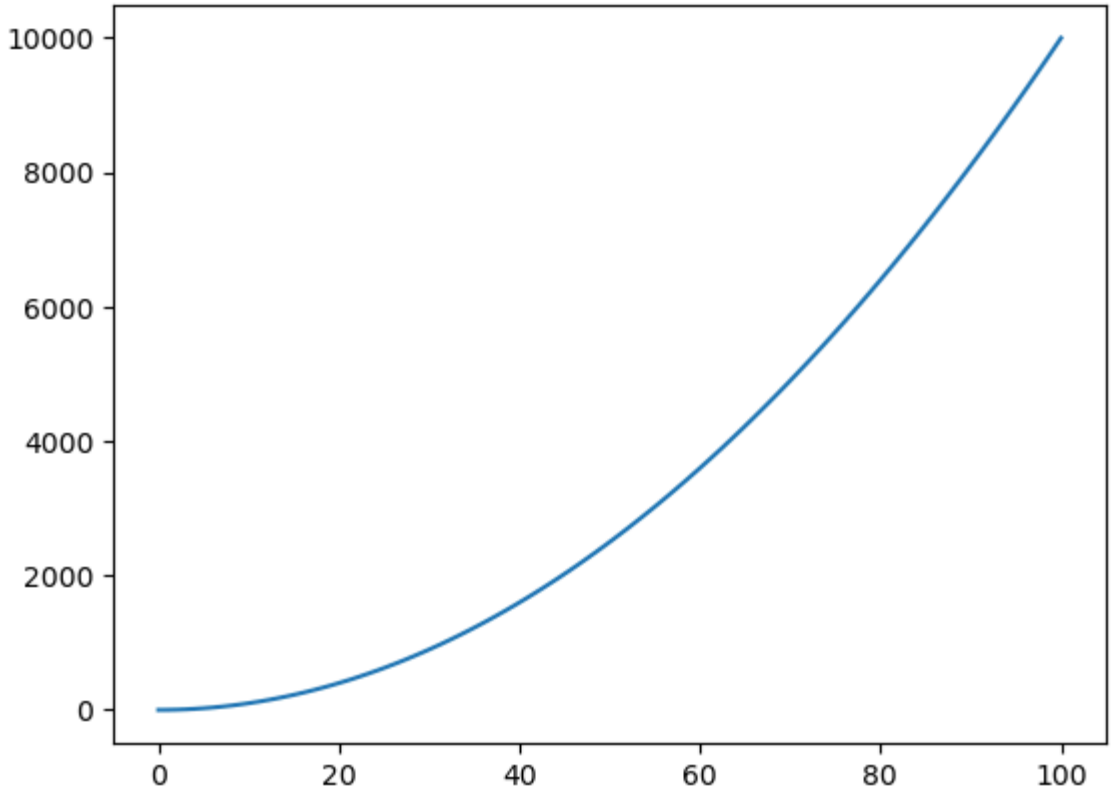
In [30]:
```
def bubble(T):
    for i in range(len(T)):
        for k in range(len(T)-1):
            if T[k] > T[k+1]:
                T[k],T[k+1] = T[k+1],T[k]
```

- C'est le if qui est le plus effectué, environ n^2 fois pour un tableau de taille n
- La complexité est en $\mathcal{O}(n^2)$

In [31]:

```
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

> ***Attention :*** *c'est mathématiquement correct de dire que le tri bulle est en $\mathcal{O}(n^{100})$, car $\mathcal{O}$ est juste une majoration*

In [ ]:
```
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]
```

- pas acceptable en informatique : on veut des informations les plus précises possibles
- analogie : si on demande un entier qui majore π, c'est correct de proposer $1.000.000$, mais plus informatif de proposer 4

> *Quand on peut, il vaudrait mieux exprimer avec un Θ qui est plus précis*

- en pratique informatique, on confond souvent $\mathcal{O}$ et Θ
- quand on annonce une complexité en $\mathcal{O}$ dans une doc, ça veut dire "la meilleure borne connue"
- c'est souvent un Θ

$\Rightarrow$ le tri bulle est en $\Theta(n^2)$

Amélioration : pas besoin d'aller jusqu'au bout à chaque étape dans le tri bulle, les i derniers sont à leur place

In [ ]:
```
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]
```

À l'étape i, le if est effectué $n-1-i$ fois :

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

Ne change pas la complexité

On a souvent à estimer des quantité de la forme

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

In [32]:

```
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
```

Theorème *si* $\alpha, \beta \geq 0$, *alors*

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

- On retrouve que le tri bulle amélioré est en $\Theta(n^2)$:

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

- Programmation dynamique

5. Complexité des algorithmes récursifs

In [33]:
```
def factorielle(n):
    if n <= 1: return 1
    return n * factorielle(n-1)

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

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

(On peut faire en temps linéaire pour les polynômes avec la méthode de Horner)

In [34]:
```
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]
```

> ***Principe :*** *estimer le nombre d'appels récursifs, et évaluer le coût de chaque appel (sans les appels à l'intérieur)*

In [36]:

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

factorielle(50)
print("nombre d'appels =", NB)
```

```
nombre d'appels = 50
```

> ***Principe :*** *estimer le nombre d'appels récursifs, et évaluer le coût de chaque appel (sans les appels à l'intérieur)*

In [41]:

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

fibo(35)
print("nombre d'appels =",NB)
```

```
nombre d'appels = 29860703
```

> ***Principe :*** *estimer le nombre d'appels récursifs, et évaluer le coût de chaque appel (sans les appels à l'intérieur)*

- Poser $A(n) =$ nombre d'appels dans le pire cas pour une entrée de taille n, trouver une récurrence sur $A(n)$ à partir de l'algorithme

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

on a

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

> ***Principe :*** *estimer le nombre d'appels récursifs, et évaluer le coût de chaque appel (sans les appels à l'intérieur)*

- Poser $A(n) =$ nombre d'appels dans le pire cas pour une entrée de taille n, trouver une récurrence sur $A(n)$ à partir de l'algorithme

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

on a

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

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

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

Théorème : *si* $A(n) = A(n-k) + \mathcal{O}(n^\alpha \log^\beta n)$ *alors*

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

In [ ]:

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

on a

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

Donc $A(n) = \mathcal{O}(n)$

Théorème : *si* $A(n) \geq A(n-k) + A(n-\ell)$ *alors*

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

L'algorithme est de complexité exponentielle (pas bon)

In [ ]:

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

Comme $A(n) = 1 + A(n-1) + A(n-2)$, la complexité est exponentielle

Théorème maître : $A(n) = a \cdot A(\frac{n}{b}) + \mathcal{O}(n^c)$ *avec* $b > 1$ *alors*

1. *si* $b^c < a$, *alors*
   $A(n) = \Theta(n^{\log_b a})$
2. *si* $b^c = a$, *alors*
   $A(n) = \Theta(n^c \log n)$
3. *si* $b^c > a$, *alors*
   $A(n) = \Theta(n^c)$

C'est un théorème très utile pour les algorithmes de type diviser pour régner

Théorème maître : $A(n) = a \cdot A(\frac{n}{b}) + \mathcal{O}(n^c)$ *avec* $b > 1$ *alors*

- *Cas 2 : si* $b^c = a$*, alors*
  $A(n)$
  $= \Theta(n^c \log n)$

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
```

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

on a $b^c = 2^0 = 1 = a$, c'est le cas 2 et donc $A(n) = \Theta(\log n)$

Théorème maître : $A(n) = a \cdot A(\frac{n}{b}) + \mathcal{O}(n^c)$ *avec* $b > 1$ *alors*

- *Cas 2 :* si $b^c = a$, alors
  $A(n)$
  $= \Theta(n^c \log n)$

In [ ]:
```
def tri_fusion(T):
    if len(T) < 2 : return T
    L, R = tri_fusion(T[:len(T)//2]), tri_fusion(T[len(T)//2:])
    return fusion(L,R) # en temps O(n), pas implantée !
```

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

on a $b^c = 2^1 = 2 = a$, c'est le cas 2 et donc $A(n) = \Theta(n \log n)$

Problème : multiplier deux polynômes $P = \sum a_i x^i$ et $Q = \sum b_j x^j$.

In [ ]:

```
def mult_naive(P,Q):
    R = [0] * (len(P)+len(Q)-1)
    for i in range(len(P)):
        for j in range(len(Q)):
            R[i+j] += P[i]*Q[j]
    return R
```

Si P et Q sont de degrés n, c'est en $\Theta(n^2)$

Algorithme de Karatsuba on écrit

$$P(X) = A(X) \cdot X^{n/2} + B(X)$$
$$Q(X) = C(X) \cdot X^{n/2} + D(X)$$

Et on a

$$R = AC \cdot X^n + \left((A-B)(D-C) + AC + BD\right) X^{n/2} + BD$$

on a juste besoin de faire :

- 3 multiplications de polynômes de degrés $\approx \frac{n}{2}$
- des sommes de polynomes en temps $\mathcal{O}(n)$
- des multiplication par X^k en temps $\mathcal{O}(n)$ (=décalage)

$$A(n) = 3A(n/2) + \mathcal{O}(n)$$

Théorème maître : $A(n) = a \cdot A(\frac{n}{b}) + \mathcal{O}(n^c)$ *avec* $b > 1$ *alors*

- *Cas 1 : si* $b^c < a$,
  $A(n)$
  $= \Theta(n^{\log_b a})$

Pour Karatsuba on a $A(n) = 3A(n/2) + \mathcal{O}(n)$ donc $a = 3, b = 2$ et $c = 1$

On a $b^c = 2 < 3 = a$, c'est bien le cas 1. Et donc la complexité de Karatsuba est

$$\Theta(n^{\log_2 3})$$

avec $\log_2(3) \approx 1.58$

In [ ]:

```
f = lambda x:x**2
g = lambda x:x**log(3,2)

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

Fin du cours 1