Informatique et Calcul Scientifique

Cours 9 : Quelques algorithmes

30.04.2025

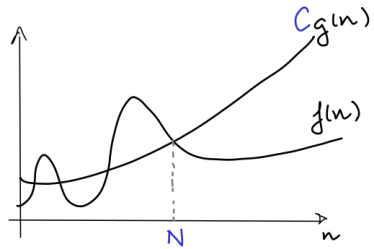
La fois passée, on a vu..

... Comment exprimer le temps de parcours $T(n)$ d'un algorithme en fonction de la taille n de l'instance d'entrée

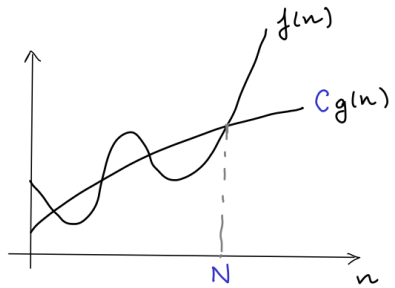
- En comptant le nombre d'opérations effectuées, celles-ci prenant habituellement un temps constant.

... Une manière d'exprimer le comportement asymptotique du temps de parcours d'un algorithme, et de comparer la performance de certains algorithme entre eux

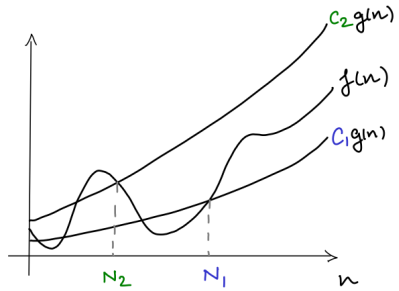
- On utilise la notation de Landau : $\mathcal{O}(\cdot)$, $\Omega(\cdot)$ et $\Theta(\cdot)$.



$f(n) = \mathcal{O}(g(n))$



$f(n) = \Omega(g(n))$



$f(n) = \Theta(g(n))$

... Une première application sur quelques algorithmes de recherche de maximum d'une liste.

Rappel : temps de parcours d'algorithmes en notation asymptotique

```
for i in range(n):
    #TEMPS CONSTANT
```

temps $\Theta(n)$

```
for i in range(n):
    for j in range(i+1, n):
        #TEMPS CONSTANT
```

temps $\Theta(n^2)$

```
for i in range(n):
    for j in range(i+1, n):
        for k in range(j+1, n):
            #TEMPS CONSTANT
```

temps $\Theta(n^3)$

But de la leçon

Aujourd'hui on...

Aujourd'hui, on verra quelques algorithmes fameux :

- Deux algorithmes de recherche
- Deux algorithmes de tri

Recherche dans une liste

Etant donnés une liste `L` quelconque de nombres et un nombre `x`, trouver `x` dans `L`.

- Retourner un indice `i` tel que `L[i] = x` si `x` apparaît dans `L`, sinon retourner `None`.
- Sans utiliser l'instruction `if x in L`, dont on ne connaît pas le temps de parcours !

```
def recherche(L, x):
    '''
    Entree: nombre x, liste L de nombres
    Sortie: i t.q. L[i]=x si un tel i existe
            None sinon
    '''
    n = len(L)

    for i in range(n):
        if L[i] == x:
            return i
```

Recherche dans une liste - temps de parcours

Que vaut le temps de parcours de cet algorithme ?

```
def recherche(L, x):
    n = len(L)
    for i in range(n):
        if L[i] == x:
            return i
```

Il existe deux cas extrêmes :

1. Si `x` est en tête de liste, $T(n) \sim \Theta(1)$ (temps constant)
2. Si `x` est en fin de liste ou n'apparaît pas dans la liste, $T(n) \sim \Theta(n)$

Rappel : le temps de parcours est défini **dans le pire des cas**, c'est-à-dire pour la pire instance imaginable. Le temps de parcours de cet algorithme est donc $\Theta(n)$ (linéaire en n).

Recherche dans une liste *triée*

Question : Peut-on améliorer cet algorithme ?

- Pour une instance la plus générale possible, non.
  Il faut parcourir toute la liste pour être sûr qu'un élément en fasse partie ou non.
- Et si la liste était triée ?

Exemple : recherche de l'élément 17 dans la liste de 23 éléments.

-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, 9, 9, 13, 17, 18, 24, 26, 27, 32, 38, 47, 51
-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, **9**, 9, 13, 17, 18, 24, 26, 27, 32, 38, 47, 51
-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, 9, 9, 13, 17, 18, 24, **26**, 27, 32, 38, 47, 51
-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, 9, 9, 13, **17**, 18, 24, 26, 27, 32, 38, 47, 51

Trouvé à l'index 14 de la liste en 3 itérations !

. Un autre exemple

Recherche dans une liste *triée*

Exemple : recherche de l'élément 31 dans la liste de 23 éléments.

-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, 9, 9, 13, 17, 18, 24, 26, 27, 32, 38, 47, 51
-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, **9**, 9, 13, 17, 18, 24, 26, 27, 32, 38, 47, 51
-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, 9, 9, 13, 17, 18, 24, **26**, 27, 32, 38, 47, 51
-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, 9, 9, 13, 17, 18, 24, 26, 27, 32, **38**, 47, 51
-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, 9, 9, 13, 17, 18, 24, 26, **27**, 32, 38, 47, 51
-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, 9, 9, 13, 17, 18, 24, 26, 27, **32**, 38, 47, 51
-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, 9, 9, 13, 17, 18, 24, 26, 27, 32, 38, 47, 51

Pas trouvé en 5 itérations !

Recherche par dichotomie

Cet exemple présente un type d'algorithme par **dichotomie**, auquel nous serons de nouveau confrontés dans la suite de ce cours. Il fonctionne de la manière suivante :

- Rechercher si une propriété est vérifiée dans un certain ensemble.
- Si oui :
  - Diviser cet ensemble par deux, et contrôler dans quel sous-espace la propriété est vérifiée
  - Répéter jusqu'à trouver le sous-espace le plus petit dans lequel celle-ci est vérifiée.
- Si non, conclure.

Un tel algorithme peut être implémenté de manière itérative.

Recherche binaire (recherche par dichotomie)

```
def recherche_binaire(L, x):
    '''
    Entree: nombre x, liste L de nombres triee
    Sortie: i t.q. L[i]=x s'il existe, None sinon
    '''
    n = len(L)
    bas = 0
    haut = n-1

    while haut >= bas:
        milieu = (bas + haut)//2
        if L[milieu] == x:
            return milieu
        elif L[milieu] > x:
            haut = milieu - 1
        else:
            bas = milieu + 1
```

Recherche binaire : temps de parcours

Calculons le temps de parcours de cet algorithme.

- Chaque itération de la boucle `while` prend un temps constant.
- On compte le nombre d'opérations effectuées.

Pour une entrée de taille n, quel est le nombre d'itérations de la boucle `while` ?

- La taille de la liste qu'on considère est à peu près coupée en deux à chaque itération.
- Lorsqu'on arrive à une liste de taille 1 (ou avant si l'élément est trouvé), l'algorithme s'arrête après cette itération.

Décomposition en puissances de 2

Question : Combien de fois faut-il diviser un entier n par 2 (division entière) pour arriver jusqu'à 1 ?

```
def decomposition(N):
    co = 0
    x = N/2
    while x >= 1:
        co +=1
        x /= 2
    return co
```

```
Output :
decomposition(1)=0
decomposition(3)=1
decomposition(4)=2
decomposition(8)=3
decomposition(13)=3
decomposition(16)=4
decomposition(25)=4
decomposition(32)=5
```

- Cet algorithme continue de diviser un nombre n par deux tant que le résultat est supérieur ou égal à 1
- Autrement dit, il permet d'obtenir k tel que $2^k \leq n < 2^{k+1}$.

Introduction à la fonction log

Soit n un entier strictement plus grand que 1. On suppose d'abord que n est une puissance de 2, i.e., il existe k in $\mathbb{N}$ tel que $n = 2^k$.

- Par définition, k est le **logarithme** en base 2 de n. On le dénote par $k = \log_2(n)$. Donc par définition, $n = 2^{\log_2(n)}$.
- $\log_2(n)$ est le nombre de fois qu'il faut diviser n par 2 pour arriver jusqu'à 1.

n	$\log_2(n)$
1	0
2	1
4	2
8	3
16	4

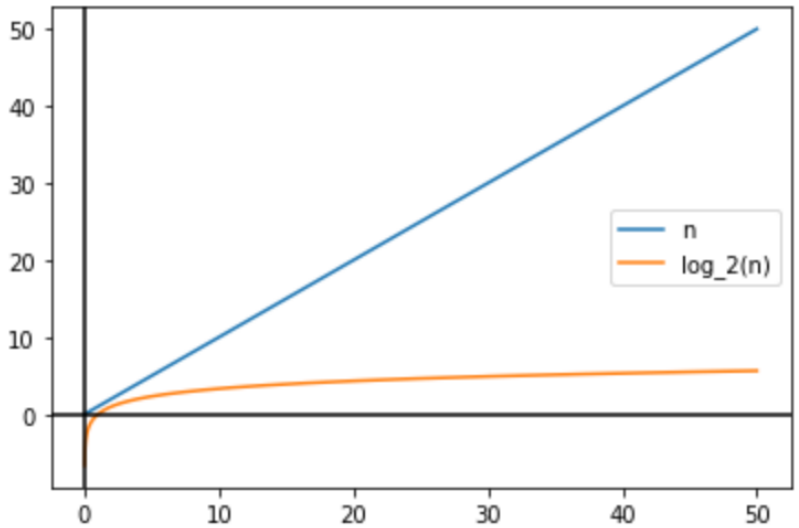
- Si n n'est pas une puissance de 2, alors k n'est pas un entier. Par exemple, $\log_2 10 = 3.32$, $2^{3.32} = 10$.

Comportement de la fonction log à l'infini

- $\lim_{n\to\infty}[\log_2(n)] = +\infty$
- $\log_2(n)$ croît vers l'infini quand n tend vers l'infini, mais **beaucoup plus lentement que n** :

$$\lim_{n\to\infty} \frac{\log_2(n)}{n} = 0.$$

- En particulier, $\log_2(n) = \mathcal{O}(n)$.

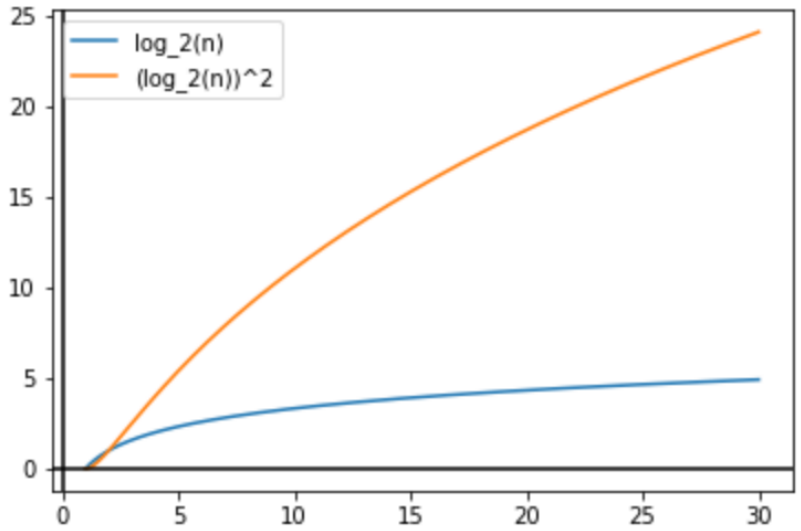


Comportement de la fonction log à l'infini

Pour des puissances rationnelles $p \leq q$,

$$[\log_2(n)]^p = \mathcal{O}([\log_2(n)]^q).$$

- $\log_2(n) = \mathcal{O}([\log_2(n)]^2)$



Comportement de la fonction log à l'infini

Pour toute puissance p, et pour toute puissance strictement positive q,

$$\lim_{n\to\infty} \frac{[\log_2(n)]^p}{n^q} = 0.$$

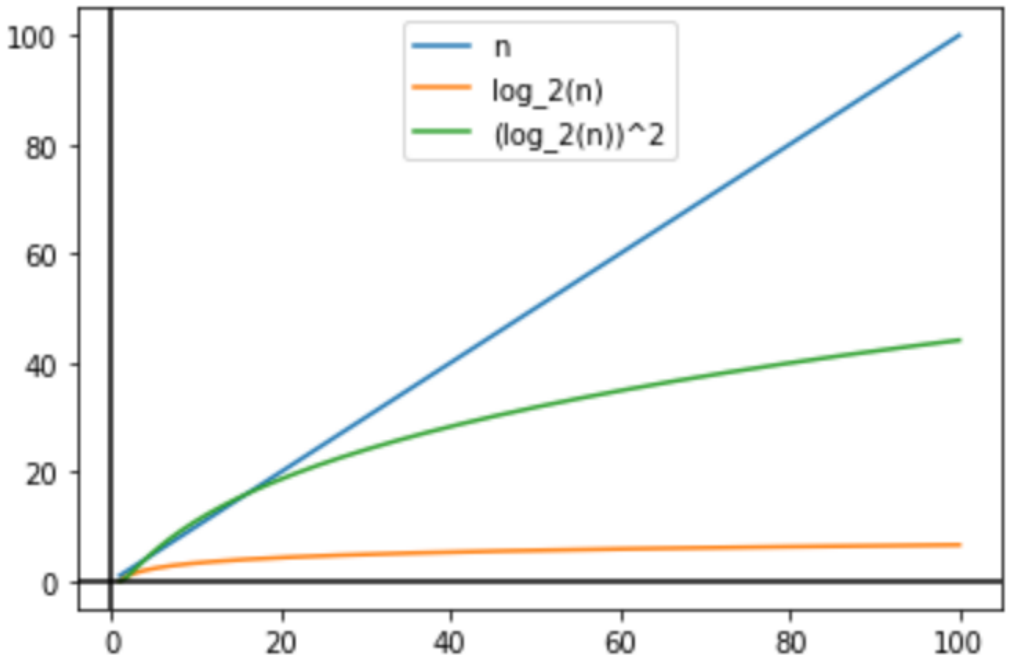
En particulier $[\log_2(n)]^p = \mathcal{O}(n^q)$. Par exemple,

- $[\log_2(n)]^2 = \mathcal{O}(n)$
- $[\log_2(n)]^{10} = \mathcal{O}(n)$
- $[\log_2(n)]^{1000} = \mathcal{O}(\sqrt{n})$

La fonction $\log_2(n)$ et ses puissances ont une **croissance logarithmique**, qui est dominée par la **croissance polynomiale** des puissances de n.

. si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, on dira que f est $o(g)$ ("f est petit o de g").

Comportement de la fonction log à l'infini



Recherche binaire - temps de parcours

La **pire** instance possible est une liste *triée* qui ne possède pas l'élément cherché, ou pour laquelle celui-ci est trouvé lorsqu'on arrive à une sous-liste de taille 1.

```
while haut >= bas:
        milieu = (bas + haut)//2
        if L[milieu] == x:
            return milieu
        elif L[milieu] > x:
            haut = milieu - 1
        else:
            bas = milieu + 1
```

- Si la tranche de liste considérée (`L[bas:haut+1]`) à une itération donnée est de taille ℓ, alors la tranche de liste considérée à la prochaine itération est de taille $\sim \ell/2$.
- Dans ce cas, la boucle **`while`** termine après $\Theta(\log_2(n))$ itérations.

Conclusion : L'algorithme de recherche binaire a donc temps de parcours $\Theta(\log_2(n))$ dans le pire des cas.

Comparaisons des deux algorithmes de recherche

La recherche simple a un temps de parcours linéaire $[T_1(n) \sim \Theta(n)]$ alors que l'algorithme de recherche binaire a un temps de parcours logarithmique $[T_2(n) \sim \Theta(\log_2(n)]$.

- Etant donnée une liste non triée, comment la trier pour pouvoir la donner en entrée à `recherche_binaire` ?
- Quel est le coût de trier une liste ? A partir de combien d'appels à `recherche_binaire` sur une liste est-ce que cela vaut la peine de trier la liste auparavant ?

Algorithmes de tri

Il existe une multitudes d'algorithmes différents pour trier une liste ou un dictionnaire. Dans ce cours, nous allons nous concentrer sur les deux premiers.

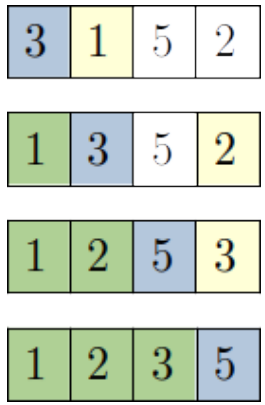
- tri par sélection
- tri par insertion
- tri à bulles
- tri par fusion
- tri rapide
- ...

On peut trier des objets selon beaucoup de critères différents. Ici, nous les trierons dans l'ordre croissant.

Tri par sélection

Idée : dans une liste triée, le premier élément est le plus petit, le deuxième est le 2^{e} plus petit, etc.

- On recherche le plus petit élément de la liste qu'on place en première position.
- Puis on recherche le deuxième plus petit élément (le plus petit de la sous-liste restante) qu'on place en deuxième position
- Ainsi de suite jusqu'à avoir parcouru toute la liste.



$\Longrightarrow$ On fait donc grandir une sous-liste triée, en insérant à chaque fois le minimum des éléments restants à la fin de cette sous-liste. [1]

1. Exemple interactif : https://visualgo.net/bn/sorting

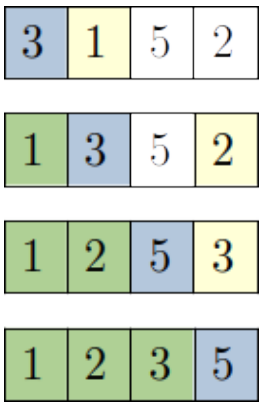
Tri par sélection

L'implémentation de cet algorithme en Python est la suivante.

```
def tri_par_selection(L):
    '''
    Entree: liste L de nombres
    Trie L
    '''
    n = len(L)
    for i in range(n):
        m = L[i]
        m_index = i
        for j in range(i+1,n):
            if L[j] < m:
                m = L[j]
                m_index = j
        L[i], L[m_index] = L[m_index], L[i]
```

Tri par sélection : temps de parcours

Que vaut son temps de parcours ?



- On commence par comparer le 1^{e} élément au $n-1$ éléments restants
- Puis on compare le 2^{e} élément aux $n-2$ éléments restants
- ...
- Finalement, on compare le $(n-1)^{\text{e}}$ au dernier.

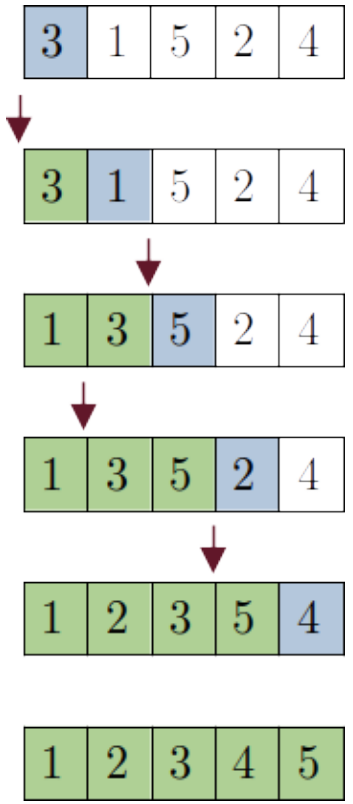
Y a-t-il une distinction entre le pire des cas, pour l'instance la plus défavorable, et d'autres cas ?

- Non, il y a *exactement* $(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}$ opérations qui prennent un temps constant, donc $T_{\text{sel}}(n)=\Theta(n^2)$.

Tri par insertion

Idée : trier une liste comme on trie une main à un jeu de cartes.

- On sélectionne le i^{e} élément, et on le compare avec l'élément $i-1$.
  $\rightarrow$ S'il est plus petit, on les échange puis le compare avec l'élément $i-2$, puis $i-3$, jusqu'à le comparer aux $i-1$ élément précédents (triés) si besoin.
  $\rightarrow$ Sinon, il est à sa bonne place.
- Par échanges successifs, on le met à la place $j \leq i$ telle que `L[j-1]` $\leq$ `L[j]` $\leq$ `L[j+1]`.



$\Longrightarrow$ On fait grandir une sous-liste triée, en insérant un élément à la fois **à la bonne place** dans cette sous-liste [2].

2. https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/visualize/

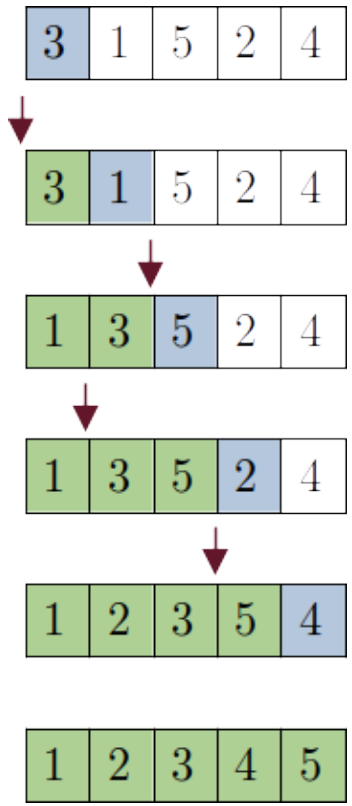
Tri par insertion

Voici une implémentation de l'algorithme par insertion.

```
def tri_par_insertion(L):
    '''
    Entree: liste L de nombres
    Trie L
    '''
    n = len(L)
    for i in range(n):
        j = i
        while j > 0 and L[j] < L[j-1]:
                L[j], L[j-1] = L[j-1], L[j]
                j -= 1
```

Tri par insertion : temps de parcours

Que vaut son temps de parcours ?



- A la i^{e} itération, on compare le i^{e} élément (bleu) avec les $(i-1)$ éléments précédents, au plus.
- Dans ce cas, les instructions dans la boucle `while` s'exécutent au plus

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} \text{ fois.}$$

Le temps de parcours de l'algorithme de tri par insertion vaut donc $T_{\text{ins}}(n) \sim \Theta(n^2)$.

Tri par insertion : temps de parcours

Y a-t-il une distinction entre le pire des cas, pour l'instance la plus défavorable, et d'autres cas ?

```
def tri_par_insertion(L):
    n = len(L)
    for i in range(n):
        j = i
        while j > 0 and L[j] < L[j-1]:
                L[j], L[j-1] = L[j-1], L[j]
                j -= 1
```

- Le meilleur des cas correspond à une liste déjà triée. On ne rentre pas dans la boucle `while`, et ne parcourt donc qu'une fois tous les éléments.
- Le pire des cas correspond à une liste triée à l'envers. On rentre alors $\frac{n(n-1)}{2}$ fois dans la boucle `while`.

Comparaison des algorithmes de tri

Ces deux algorithmes ne sont pas optimaux pour des instances de grande taille car $\Theta(n^2)$. Certains algorithmes non étudiés dans ce cours ont une complexité moins élevée :

- Tri par fusion : $\Theta(n \log_2 n)$
- Tri rapide : $\Theta(n^2)$

Question : Comment choisir quel algorithme utiliser ?

... ça dépend.

- de la disposition initiale de l'instance d'entrée,
- et nous n'avons abordé que le critère du temps de parcours !

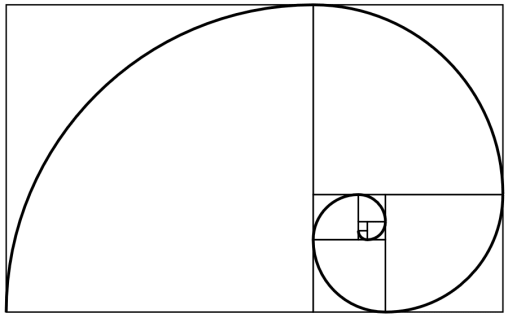
La méthode `sort()` de Python utilise un mélange du tri par insertion et du tri par fusion. Il a aussi une complexité $\Theta(n \log_2 n)$.

Bonus : complexité exponentielle

Une grandeur très répandue en cryptographie est la suite de Fibonacci.

Pour $n \in \mathbb{N}$, le $n$ème nombre de Fibonacci est défini comme

$$f_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$



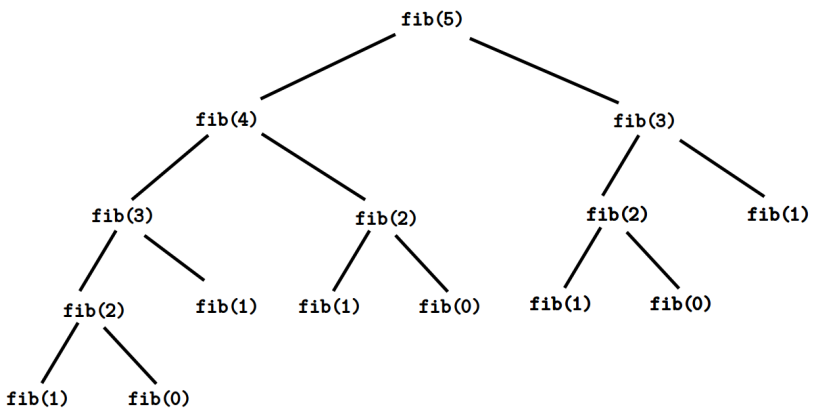
```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

Bonus : complexité exponentielle

Quelle serait la complexité de cet algorithme ?

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

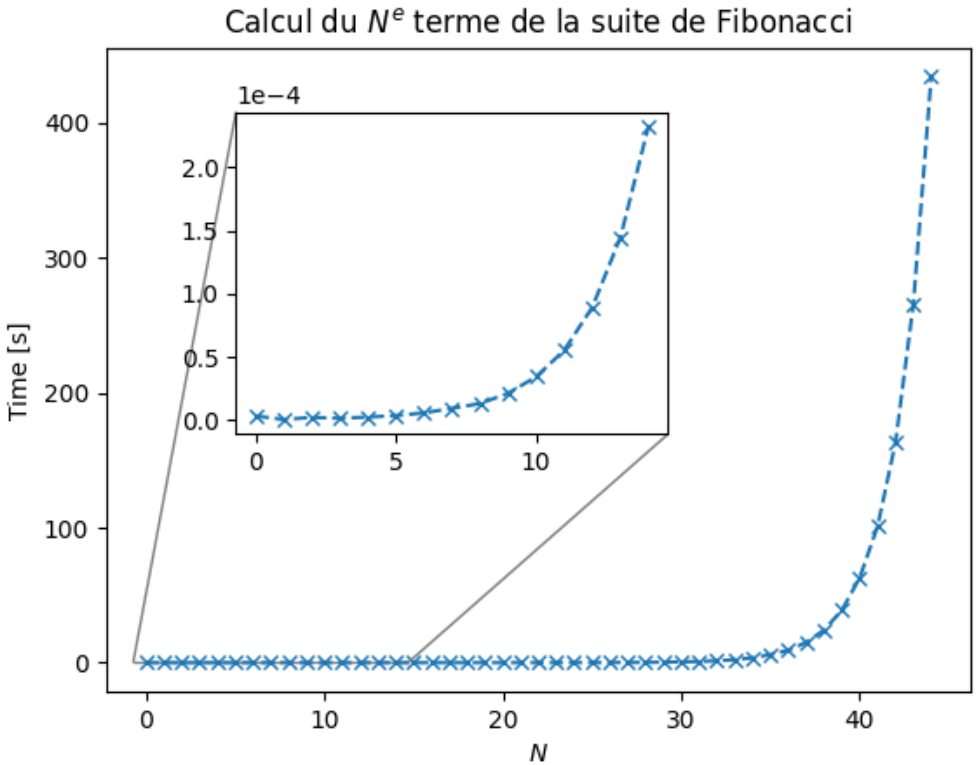
`fib(n)` appelle `fib(n-1)` et `fib(n-2)`, qui elles-mêmes appellent `fib(n-2)` et `fib(n-3)`, etc...



Le nombre d'opérations dans `fib(n+1)` vaut (environ) le double de `fib(n)`.
Le temps de parcours de cet algorithme suit une loi exponentielle : $T(n) \sim a^n,\ a > 1$

Bonus : complexité exponentielle

Une telle situation est à éviter absolument ! Ici, $T = 434$ s pour $N = 45$.

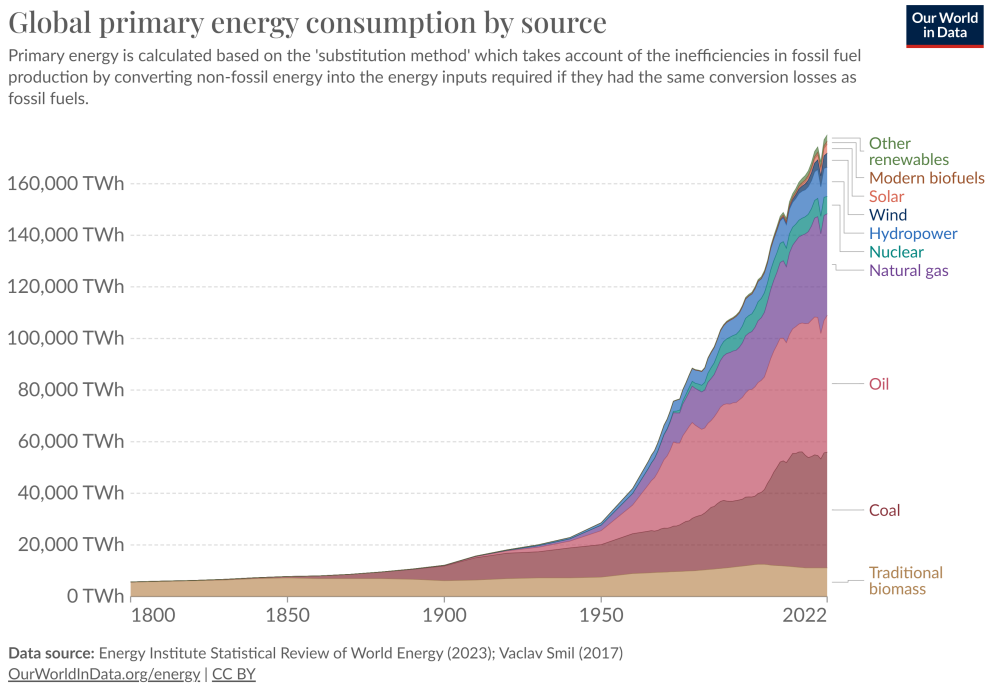


Heureusement, il existe généralement un moyen de contourner cette croissance exponentielle.

Bonus : complexité exponentielle

Une multitude de situations physiques et sociales ont en réalité une croissance exponentielle. On peut citer [3] :

- La croissance d'une population
- La propagation d'une maladie (covid)
- L'utilisation de ressources naturelles



Mathématiquement, $f(n) = f_0 \cdot (1 + r)^n$

3. population,ressources

Take Home Message

En faisant l'hypothèse que l'instance est *triée*, l'algorithme de recherche binaire [$\mathcal{O}(\log_2 n)$] est plus efficace que l'algorithme de force brute [$\mathcal{O}(n)$].
Comment trier une liste ?

- *Tri par sélection :* On insère le **bon** élément à la fin de la sous-liste triée.
- *Tri par insertion :* On insère chaque élément à la **bonne** place dans la sous-liste triée.

Ces deux algorithmes ont une complexité temporelle $\mathcal{O}(n^2)$