

Informatique et Calcul Scientifique

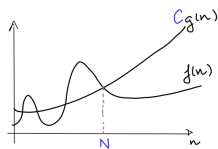
Cours 9 : Recherche dans une liste, introduction au logarithme

Aujourd'hui on verra

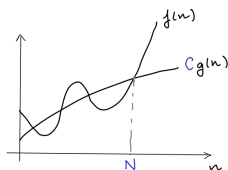
- ▶ Rappel notations $\mathcal{O}(\cdot)$, $\Omega(\cdot)$ et $\Theta(\cdot)$ et étude de la croissance asymptotique du temps de parcours d'un algorithme
- ▶ Recherche d'un élément dans une liste quelconque
- ▶ Recherche d'un élément dans une liste triée
- ▶ Logarithme de base 2.

- ▶ Notation $\mathcal{O}(\cdot)$, $\Omega(\cdot)$ et $\Theta(\cdot)$ pour comparer les vitesses de croissance de deux fonctions tendant vers l'infini
- ▶ Expression de l'ordre de croissance du temps de parcours d'un algorithme en notation $\mathcal{O}(\cdot)$, $\Omega(\cdot)$ et $\Theta(\cdot)$.

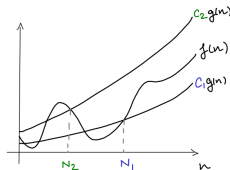
Rappel



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



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



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

Soit $n \in \mathbb{N}$, et f, g des fonctions positives¹ de n .

$$f(n) = \mathcal{O}(g(n)) \iff \exists C, N > 0 \text{ t.q. } \forall n > N \quad f(n) \leq C \cdot g(n).$$

1. Elles n'ont en fait besoin d'être qu'asymptotiquement positives, c'est-à-dire positives à partir d'un certain rang.

- ▶ Plus précisément : $\mathcal{O}(g(n))$ est **l'ensemble** des fonctions $f(n)$ telles que

$$\exists C, N > 0 \text{ t.q. } \forall n > N \quad f(n) \leq C \cdot g(n).$$

- ▶ La notation correcte serait $f \in \mathcal{O}(g)$, mais on emploie communément la terminologie " f est $\mathcal{O}(g)$ " et " $f = \mathcal{O}(g)$ ", qui sont des abus de notation.

Rappel - Notation $\mathcal{O}(\cdot)$

- ▶ Pour des puissances rationnelles $p \leq q$, $n^p = \mathcal{O}(n^q)$.
 - ▶ $n = \mathcal{O}(n^2)$, $n = \mathcal{O}(n)$, $\sqrt{n} = \mathcal{O}(n)$,...
 - ▶ Si $p < q$, n^p est $\mathcal{O}(n^q)$ mais n^q n'est pas $\mathcal{O}(n^p)$.
- ▶ Si f est un polynôme² de degré p et g un polynôme de degré q avec $p \leq q$, alors $f(n) = \mathcal{O}(g(n))$.
 - ▶ $n + 1 = \mathcal{O}(n^2)$, $n^2 + n + 2 = \mathcal{O}(n^3 + n)$,...
 - ▶ (plus général) $100n + \sqrt{n} = \mathcal{O}(n^2 + 3\sqrt{n})$

2. ou si f et g sont des sommes de puissances rationnelles dont les termes dominants sont respectivement an^p et bn^q . Par abus de notation, à partir de maintenant, "polynôme" indiquera aussi des sommes de puissances rationnelles.

Rappel - Notation $\Theta(\cdot)$

- ▶ Si f et g sont deux polynômes de même degré, alors $f(n) = \Theta(g(n))$
 - ▶ $n^2 + n + 1 = \Theta(3n^2 + 2n + 5)$
 - ▶ (plus général) $n + \sqrt{n} = \Theta(30n)$
- ▶ Pour des puissances $p < q$, n^p est $\mathcal{O}(n^q)$ mais n^p n'est pas $\Theta(n^q)$.
- ▶ La notation $\Theta(\cdot)$ permet de négliger les termes d'ordre inférieur et la constante multiplicative (positive) du terme dominant.

Rappel : temps de parcours d'algorithmes en notation asymptotique

Soit $T(n)$ le temps de parcours d'un algorithme pour une entrée de taille n **au pire des cas**.

- ▶ Si on donne une fonction $f_1(n)$ telle que $T(n) = \mathcal{O}(f_1(n))$, f_1 est une **borne supérieure** sur le temps de parcours de l'algorithme.
- ▶ Si on donne une fonction $f_2(n)$ telle que $T(n) = \Omega(f_2(n))$, f_2 est une **borne inférieure** sur le temps de parcours de l'algorithme.
- ▶ Si on donne une fonction $f(n)$ telle que $T(n) = \Theta(f(n))$, f est à la fois une borne supérieure et une borne inférieure sur le temps de parcours de l'algorithme : elle décrit le comportement asymptotique du temps de parcours.

Rappel : temps de parcours d'algorithmes en notation asymptotique

- ▶ Soit $T(n)$ le temps de parcours d'un algorithme pour une entrée de taille n au pire des cas. Comment exprimer $T(n)$ en fonction de n ?
- ▶ En regardant le code !
- ▶ Selon notre modèle de computation, la plupart des instructions usuelles prennent un temps de parcours constant (plus précisément, borné par une constante)
- ▶ Le temps de parcours de l'algorithme dépendra du nombre d'instructions exécutées.

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 $\Theta(n^3)$

Recherche dans une liste

- ▶ Etant donné une liste `L` 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 parcours³ !

```
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
```

3. en fait, son temps de parcours est linéaire en n et son implémentation est équivalente à notre algorithme de recherche.

Correctitude (idée de preuve)

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

- ▶ Si l'algorithme retourne i , ce i satisfait $L[i] = x$.
- ▶ Si l'algorithme retourne `None`, x n'apparaît pas dans la liste :
 - ▶ Invariant de boucle : au début de la i ème itération de la boucle `for`, on sait que x n'est pas dans $L[0:i]$.

Recherche dans une liste - temps de parcours

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

- ▶ Si `x` est en tête de liste, $\Theta(1)$ (temps constant)
- ▶ Si `x` est en fin de liste ou n'apparaît pas dans la liste, $\Theta(n)$
- ▶ **RAPPEL** : le temps de parcours est défini **dans le pire des cas** : le temps de parcours de cet algorithme est $\Theta(n)$ (linéaire en n).

- ▶ Et si la liste était triée ?
- ▶ Exemple : recherche de l'élément 17 dans la liste
-7, -3, -3, -1, 0, 0, 1, 2, 5, 6, 8, 9, 9, 13, 17, 18, 24, 26, 27, 32, 38, 47, 51

- ▶ Et si la liste était triée ?

- ▶ Exemple : recherche de l'élément 17 dans la liste

-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

- ▶ Et si la liste était triée ?

- ▶ Exemple : recherche de l'élément 17 dans la liste

-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

- ▶ Et si la liste était triée ?
- ▶ Exemple : recherche de l'élément 17 dans la liste

-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

- ▶ Et si la liste était triée ?
- ▶ Exemple : recherche de l'élément 17 dans la liste

-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 !

- ▶ Exemple : recherche de l'élément 31 dans la liste

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

- ▶ Exemple : recherche de l'élément 31 dans la liste

-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

► Exemple : recherche de l'élément 31 dans la liste

-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

► Exemple : recherche de l'élément 31 dans la liste

-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

► Exemple : recherche de l'élément 31 dans la liste

-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

► Exemple : recherche de l'élément 31 dans la liste

-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

► Exemple : recherche de l'élément 31 dans la liste

-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 !

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.

Un tel algorithme peut être implémenté de manière récursive et itérative.

Recherche binaire (recherche par dichotomie)

```
def recherche_binaire(L, x):  
    '''  
    Entree: nombre x, liste L de nombres trie  
    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
```

Correctitude (idée)

- ▶ L'algorithme termine car :
 - ▶ Avant le début de la boucle `while`, `haut - bas` est positif (ou alors on ne rentre pas dans la boucle)
 - ▶ A chaque itération de la boucle, `haut - bas` décroît d'au moins 1 (ou alors on sort de la boucle)
 - ▶ La boucle termine lorsque `haut - bas` < 0 .
- ▶ L'algorithme rend la valeur correcte car :
 - ▶ S'il retourne la valeur `i`, `i` satisfait bien `L[i] = x`
 - ▶ S'il retourne la valeur `None`, c'est qu'on est sorti de la boucle avec `haut < bas`. Or a chaque itération de la boucle, on maintient la propriété
Si `x` est dans `L`, alors il est dans `L[bas:haut+1]`
A la sortie de la boucle, `L[bas:haut+1]` est une liste vide et donc `x` n'est pas dans `L`.

- ▶ Chaque itération de la boucle `while` prend un temps constant.
- ▶ 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.
- ▶ Combien de fois faut-il diviser un entier n par 2 (division entière) pour arriver jusqu'à 1 ?

Décomposition en puissances de 2

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

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 2 tant que le résultat est supérieur à 1.
- ▶ Autrement dit, il permet d'obtenir k tel que $2^k \leq n < 2^{k+1}$.

Introduction à la fonction log

Cet algorithme donne une première représentation de la fonction mathématique log qui apparaît souvent en informatique.

- ▶ En analyse, la fonction logarithme (de base e) sera définie de manière géométrique comme l'aire sous la courbe de la fonction $f(x) = \frac{1}{x}$.
- ▶ Dans ce cours, on définit le logarithme de base entière⁴ de manière combinatoire.

4. Usuellement base 2 ou 10

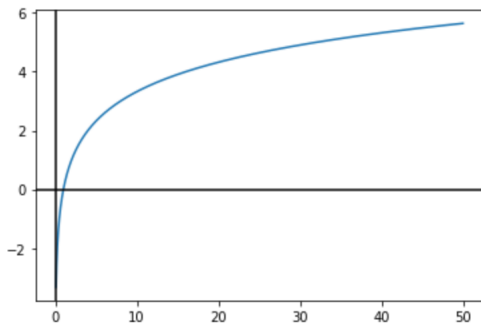
Introduction à la fonction log

- ▶ Soit n un entier strictement positif. 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

Propriétés de la fonction log

La fonction $\log_2(x)$ est en fait définie sur $\mathbb{R}_+^* =]0, \infty[$



- ▶ $\log_2(x)$ est **strictement croissante** : pour tous $x_1, x_2 \in]0, \infty[$

$$x_1 < x_2 \Leftrightarrow \log_2(x_1) < \log_2(x_2)$$

- ▶ Pour tous $x, x_1, x_2 \in]0, \infty[$, pour toute puissance p rationnelle :

$$\log_2(x_1 \cdot x_2) = \log_2(x_1) + \log_2(x_2)$$

$$\log_2(x_1/x_2) = \log_2(x_1) - \log_2(x_2)$$

$$\log_2(x^p) = p \log_2(x)$$

Propriétés de la fonction log

- ▶ On s'intéressera aux valeurs de $\log_2(n)$ uniquement pour n entier, et en particulier pour n tendant vers l'infini.
- ▶ Pour n puissance de 2, $\log_2(n)$ a un sens combinatoire.
- ▶ Pour n entier positif qui n'est pas une puissance de 2, soit k la plus grande puissance de 2 telle que $2^k < n$. On a donc

$$2^k < n < 2^{k+1}.$$

- ▶ Puisque $\log_2$ est croissante, on a

$$\log_2(2^k) < \log_2(n) < \log_2(2^{k+1})$$

et donc

$$k < \log_2(n) < k + 1.$$

- ▶ Par exemple, $10 < \log_2(2000) < 11$ (puisque $1024 < 2000 < 2048$).

Comportement de la fonction $\log$ à l'infini

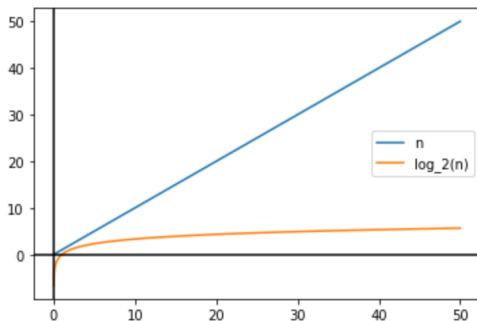
- ▶ $\lim_{n \rightarrow \infty} (\log_2(n)) = +\infty$
- ▶ $\log_2(n)$ croît vers l'infini quand n tend vers l'infini, mais **beaucoup plus lentement que n** :

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

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

n	$\log_2(n)$
1 024	10
1 048 576	20
1 073 741 824	30
$1.099511628 \times 10^{12}$	40

Comportement de la fonction log à l'infini

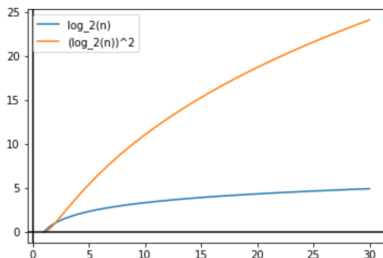


Comportement de la fonction log à l'infini

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

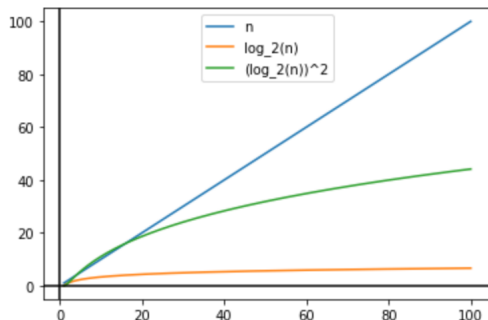
- Pour toute puissance p , et pour toute puissance strictement positive q ,

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

- En particulier $(\log_2(n))^p = \mathcal{O}(n^q)$. Par exemple,
 - $(\log_2(n))^2 = \mathcal{O}(n)$
 - $(\log_2(n))^{10} = \mathcal{O}(n)$
 - $(\log_2(n))^{10} = \mathcal{O}(\sqrt{n})$
- $\log_2(n)$ et ses puissances ont une **croissance logarithmique**, qui est dominée par la **croissance polynomiale** des puissances de n .

. si $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, 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

```
def recherche_binaire(L, x):  
    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
```

- ▶ Temps de parcours dans le pire des cas : lorsque l'élément n'est pas trouvé ou est trouvé lorsqu'on est arrivé à une liste de taille 1.
- ▶ Dans ce cas, la boucle `while` termine après $\Theta(\log(n))$ itérations.
- ▶ L'algorithme de recherche binaire a donc temps de parcours $\Theta(\log(n))$ dans le pire des cas.

Recherche binaire - nombre d'itérations (idée de preuve)

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

La boucle `while` termine
après $\Theta(\log(n))$ itérations.

► Idée de preuve :

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 $\leq \ell/2$.

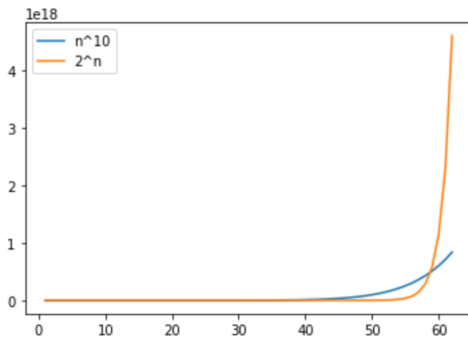
- ▶ Etant donné une liste non triée, comment la trier pour pouvoir la donner en entrée à `recherche_binaire` ?
- ▶ Quel est le coût de trier une liste ? A partir de combien d'appels à `recherche_binaire` sur une liste est-ce que cela vaut la peine de trier la liste auparavant ?
- ▶ Questions à méditer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

- Pour toute puissance rationnelle p , pour tout réel $a > 1$,

$$\lim_{n \rightarrow \infty} \frac{n^p}{a^n} = 0.$$

- En particulier, $n^p = O(a^n)$ (et a^n n'est pas $O(n^p)$). Par exemple,
 - $n = O(2^n)$
 - $n^{100} = O(2^n)$
 - ...
- Pour tout polynôme $f(n)$, la **croissance polynomiale** de f est dominée par la **croissance exponentielle** de a^n .
 - $n^2 + n\sqrt{n} + 1 = O(2^n)$
 - $n^{10} + n^8 + 3n^4 = O(2^n)$
 - ...

Croissance exponentielle



Croissance exponentielle

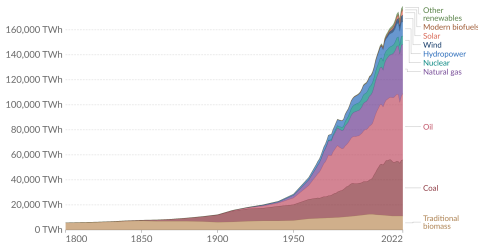
Une multitude de situations physiques et sociales ont en réalité une croissance exponentielle. On peut citer⁵ :

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

Global primary energy consumption by source

Primary energy is calculated based on the 'substitution method' which takes account of the inefficiencies in fossil fuel production by converting non-fossil energy into the energy inputs required if they had the same conversion losses as fossil fuels.

Our World in Data



Data source: Energy Institute Statistical Review of World Energy (2023); Vaclav Smil (2017)
[OurWorldinData.org/snsrev](https://www.ourworldindata.org/snsrev) | CC-BY

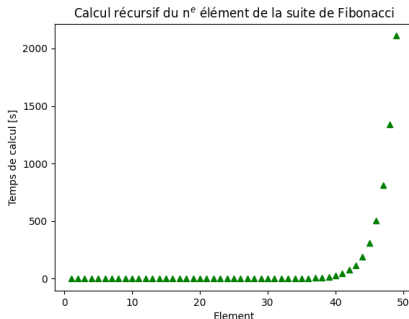
Mathématiquement, $f(n) = f_0 \cdot (1 + r)^n$

5. population,ressources

Algorithmes exponentiels

Certains algorithmes ont un temps de parcours $T(n)$ **exponentiel** en la taille n de l'entrée : $T(n) = \Theta(a^n)$ (ou $T(n) = \Omega(a^n)$ et $T(n) = \mathcal{O}(b^n)$ pour des constantes a, b).

- ▶ Exemple : on peut prouver que l'algorithme récursif `fib()` vu au Cours 7 pour calculer le n^{eme} nombre de Fibonacci a un temps de parcours exponentiel en n . On observe empiriquement la croissance de ce temps de parcours :



- ▶ Un autre exemple : le problème de la somme de sous-ensembles⁶ : étant donné une liste L de n nombres et une valeur cible V , existe-t-il un sous-ensemble des indices de L tel que la somme des éléments correspondants de L vaut V ?
 - ▶ Input : $L = [11, 2, 9, -5, 2, 7, -2, -3]$ et $V = 1$
 - ▶ Output : oui car $2 + 2 - 3 = 1$.
- ▶ L'algorithme naïf parcourt tous les sous-ensembles d'indices de L et vérifie la somme des éléments.
- ▶ Il y a 2^n tels sous-ensembles ! Le temps de parcours de cet algorithme a une borne inférieure de $\Omega(2^n)$.

6. https://en.wikipedia.org/wiki/Subset_sum_problem