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.

Rappel — la semaine dernière

Rappel — Préliminaires

- On s'intéresse à l'étude de la *complexité* de certains algorithmes : la croissance *asymptotique* de leur temps d'exécution en fonction de la taille des données d'entrée.
- Nous modélisons ce temps d'exécution (au pire des cas) par une fonction $T(n)$, où n désigne la taille de l'entrée, mais nous ne pouvons définir cette fonction que *modulo la multiplication par un facteur constant*.
- $T(n)$ est une fonction de $\mathbb{N}$ vers $\mathbb{R}$. Toute fonction abordée dans ce cours doit être considérée comme étant de cette forme.

Rappel — Les définitions théoriques

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

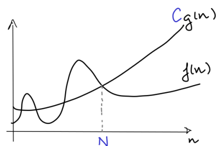
  L'ensemble des fonctions dont la **croissance** asymptotique **est dominée** par celle de g.

- $\Omega(g) = \{f \mid \exists C, N > 0 \text{ t.q. } \forall n > N,\ f(n) \geq C \cdot g(n)\}$,

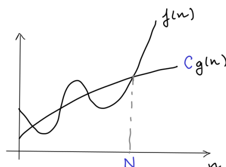
  L'ensemble des fonctions dont la **croissance** asymptotique **domine** celle de g.

- $\Theta(g) = \mathcal{O}(g) \cap \Omega(g)$,

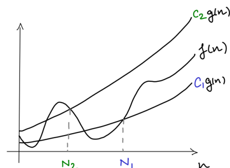
  L'ensemble des fonctions dont la **croissance** asymptotique **est équivalente** à celle de g.



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



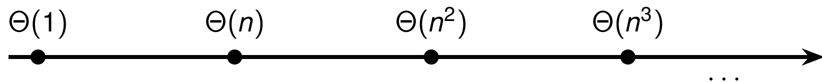
$f(n) = \Omega(g(n))$



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

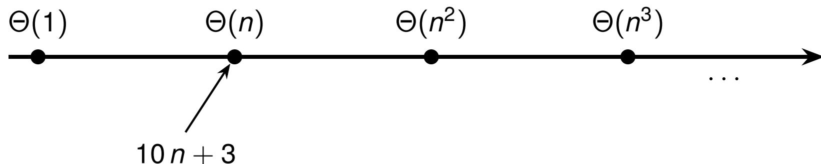
La hiérarchie de la complexité

- Si f et g sont telles que $f = \Theta(g)$, elles peuvent être considérée comme *équivalentes* en ce qui nous concerne.
- En considérons les fonctions modulo cette équivalence, nous obtenons la hiérarchie suivante :



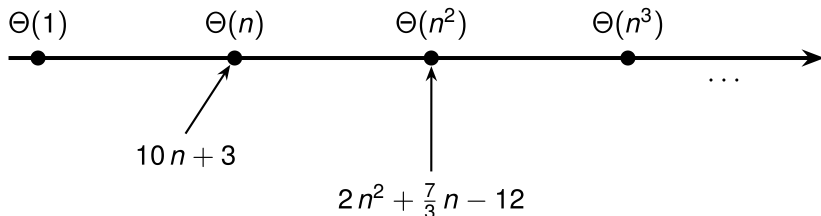
La hiérarchie de la complexité

- Si f et g sont telles que $f = \Theta(g)$, elles peuvent être considérée comme *équivalentes* en ce qui nous concerne.
- En considérons les fonctions modulo cette équivalence, nous obtenons la hiérarchie suivante :



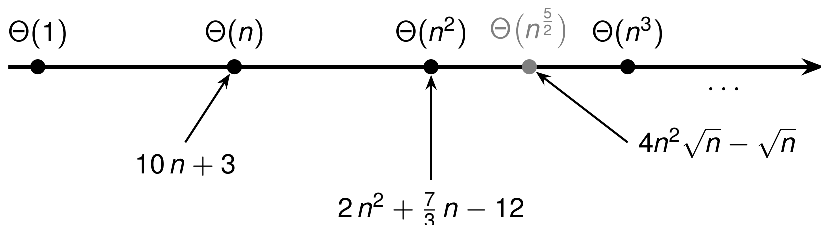
La hiérarchie de la complexité

- Si f et g sont telles que $f = \Theta(g)$, elles peuvent être considérée comme *équivalentes* en ce qui nous concerne.
- En considérons les fonctions modulo cette équivalence, nous obtenons la hiérarchie suivante :



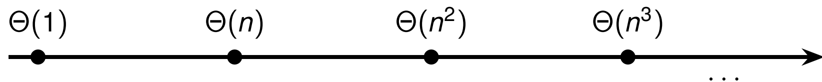
La hiérarchie de la complexité

- Si f et g sont telles que $f = \Theta(g)$, elles peuvent être considérée comme *équivalentes* en ce qui nous concerne.
- En considérons les fonctions modulo cette équivalence, nous obtenons la hiérarchie suivante :



La hiérarchie de la complexité

- Si f et g sont telles que $f = \Theta(g)$, elles peuvent être considérée comme *équivalentes* en ce qui nous concerne.
- En considérons les fonctions modulo cette équivalence, nous obtenons la hiérarchie suivante :

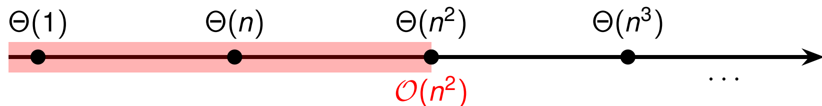


Supposons que $T(n)$ modélise le temps d'exécution d'un algorithme.

- Notre objectif est de comprendre la position de $T(n)$ dans cette hiérarchie.

La hiérarchie de la complexité

- Si f et g sont telles que $f = \Theta(g)$, elles peuvent être considérée comme *équivalentes* en ce qui nous concerne.
- En considérons les fonctions modulo cette équivalence, nous obtenons la hiérarchie suivante :

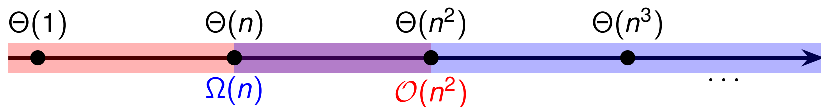


Supposons que $T(n)$ modélise le temps d'exécution d'un algorithme.

- Notre objectif est de comprendre la position de $T(n)$ dans cette hiérarchie.
  - Connaître une fonction g telle que $T = \mathcal{O}(g)$ donne une borne *supérieure* à sa position,

La hiérarchie de la complexité

- Si f et g sont telles que $f = \Theta(g)$, elles peuvent être considérée comme *équivalentes* en ce qui nous concerne.
- En considérons les fonctions modulo cette équivalence, nous obtenons la hiérarchie suivante :

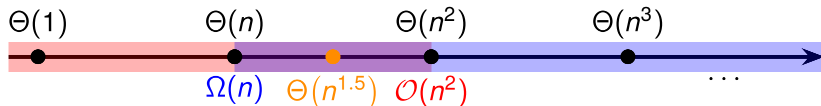


Supposons que $T(n)$ modélise le temps d'exécution d'un algorithme.

- Notre objectif est de comprendre la position de $T(n)$ dans cette hiérarchie.
  - Connaître une fonction g telle que $T = \mathcal{O}(g)$ donne une borne *supérieure* à sa position,
  - Connaître une fonction g telle que $T = \Omega(g)$ donne une borne *inférieure* à sa position,

La hiérarchie de la complexité

- Si f et g sont telles que $f = \Theta(g)$, elles peuvent être considérée comme *équivalentes* en ce qui nous concerne.
- En considérons les fonctions modulo cette équivalence, nous obtenons la hiérarchie suivante :



Supposons que $T(n)$ modélise le temps d'exécution d'un algorithme.

- Notre objectif est de comprendre la position de $T(n)$ dans cette hiérarchie.
  - Connaître une fonction g telle que $T = \mathcal{O}(g)$ donne une borne *supérieure* à sa position,
  - Connaître une fonction g telle que $T = \Omega(g)$ donne une borne *inférieure* à sa position,
  - Connaître une fonction g telle que $T = \Theta(g)$ donne sa position exacte.

Pour un algorithme donner, comment trouver $T(n)$?

- En regardant le code ! Et en se rappelant qu'il faut considérer *le pire cas* d'une donnée d'entrée de taille n.
- La plupart des instructions usuelles prennent un temps de parcours constant (c'est notre modèle de computation).
- $T(n)$ dépend alors souvent du *nombre d'exécution* d'une instruction (ce qui est principalement lié aux boucles).

```
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 $\Theta\left(\frac{1}{2}n^2\right) = \Theta\left(n^2\right)$

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

temps $\Theta(n^3)$

Algorithmes de recherche

Recherche dans une liste

- Etant donnés 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 le temps de parcours n'est pas constant !

```python
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
```

Correctitude

```python
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]`.
    $\rightarrow$ À prouver chez vous !

Recherche dans une liste - temps de parcours

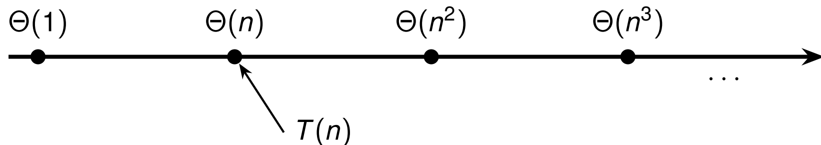
```python
def recherche(L, x):
    n = len(L)
    for i in range(n):
        if L[i] == x:
            return i
```

- 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, $\Theta(n)$

Recherche dans une liste - temps de parcours

```python
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, $\Theta(n)$
- Comme le temps de parcours est défini **dans le pire des cas**, le temps de parcours de cet algorithme est $\Theta(n)$.



Recherche dans une liste triée

- 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

. Un autre exemple

Recherche dans une liste triée

- 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

. Un autre exemple

Recherche dans une liste triée

- 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

. Un autre exemple

Recherche dans une liste triée

- 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

. Un autre exemple

Recherche dans une liste triée

- 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 !

. Un autre exemple

Recherche dans une liste triée

- 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

Recherche dans une liste triée

- 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

Recherche dans une liste triée

- 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

Recherche dans une liste triée

- 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

Recherche dans une liste triée

- 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

Recherche dans une liste triée

- 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 !

Recherche binaire (recherche par dichotomie)

```python
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
```

Correctitude – condition de terminaison

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

- Preuve que l'algorithme termine :
  - On considère le **variant de boucle** suivant : `haut - bas`.
  - À chaque itération de la boucle, `haut - bas` décroît d'au moins 1 (ou alors, on `return` une valeur).
  - La boucle termine lorsque `haut - bas` < 0.

Correctitude – preuve de validité

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

Pour prouver que l'algorithme rend la valeur correcte il y a deux conditions à vérifier :

- S'il retourne la valeur `i`, `i` satisfait bien `L[i]` = `x`.
  ↳ Trivial.

Correctitude – preuve de validité

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

Pour prouver que l'algorithme rend la valeur correcte il y a deux conditions à vérifier :

- S'il retourne la valeur `i`, `i` satisfait bien `L[i]` = `x`.
  ↳ Trivial.
- S'il retourne la valeur `None`, `x` n'est pas dans la liste.
  ↳ Plus dur, il nous faut un invariant de boucle :

Si `x` est dans `L`, alors il est dans `L[bas:haut + 1]`

  Terminaison. Comme, lorsque la boucle se termine, `haut < bas`, `L[bas:haut + 1]` sera vide et l'invariant de boucle impliquera bien que `x` n'est pas dans `L`.

Correctitude – preuve de l'invariant de boucle

```
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
```

Invariant de boucle :

Si `x` est dans `L`, alors il est dans `L[bas:haut + 1]`

- **Initialisation.**
  ↳ Trivial, car au début `L[bas:haut + 1]` = `L`.

Correctitude – preuve de l'invariant de boucle

```python
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
```

Invariant de boucle :

Si `x` est dans `L`, alors il est dans `L[bas:haut + 1]`

- **Initialisation.**
  ↳ Trivial, car au début `L[bas:haut + 1]` = `L`.
- **Maintenance.**
  ↳ Si `L[milieu] > x`, alors `x` ne peut pas être dans `L[milieu:haut + 1]` (car `L` est triée). Donc si `x` est dans `L`, il est dans `L[bas:milieu]` = `L[bas:haut_new + 1]`, où `haut_new` est la valeur de `haut` après avoir été réaffectée.

Correctitude – preuve de l'invariant de boucle

```python
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
```

Invariant de boucle :

Si `x` est dans `L`, alors il est dans `L[bas:haut + 1]`

- **Initialisation.**
  ↳ Trivial, car au début `L[bas:haut + 1]` = `L`.

- **Maintenance.**
  ↳ Si `L[milieu] > x`, alors `x` ne peut pas être dans `L[milieu:haut + 1]` (car `L` est triée). Donc si `x` est dans `L`, il est dans `L[bas:milieu]` = `L[bas:haut_new + 1]`, où `haut_new` est la valeur de `haut` après avoir été réaffectée.
  ↳ Le cas où `L[milieu] < x` peut être traité de manière analogue.

Recherche binaire : temps de parcours

- 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 ?

La fonction log

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 $f(x) = \frac{1}{x}$.

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 considère toujours que le logarithme est en **base 2**, sauf indication contraire. On l'écrit $\log(n)$ ou $\log n$.

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 considère toujours que le logarithme est en **base 2**, sauf indication contraire. On l'écrit $\log(n)$ ou $\log n$.
- On donne une définition *combinatoire* de $\log n$ (en supposons que n est une puissance de 2) :

$$\log n = \boxed{\begin{array}{l}\text{le nombre fois qu'il faut diviser} \\ n \text{ par } 2 \text{ pour arriver à } 1.\end{array}}$$

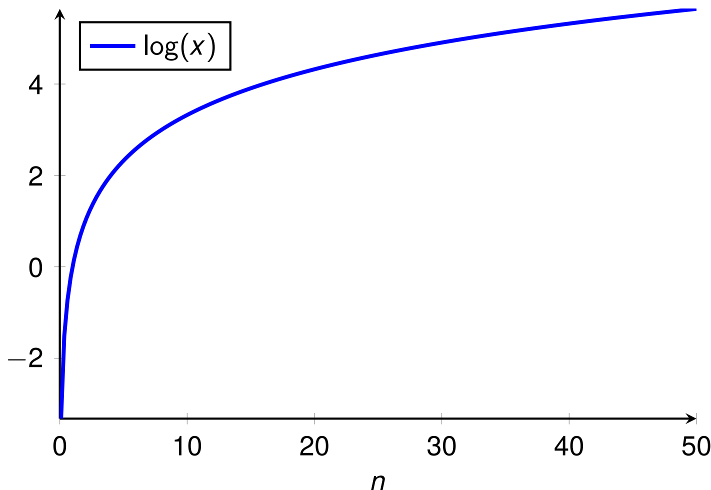
Introduction à la fonction log

- 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 $n = 2^k$.
- Remarquons que cet exposant k est justement le nombre de fois qu'il faut diviser n par 2 pour arriver à 1, autrement dit, $\log n$.
- Donc par définition, $n = 2^{\log n}$ et $\log n = \log(2^k) = k$.

n	$\log n$
1	0
2	1
4	2
8	3
16	4

Propriétés de la fonction log

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



Propriétés de la fonction log

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

$$x_1 < x_2 \Leftrightarrow \log(x_1) < \log(x_2).$$

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

$$\begin{aligned} \log(x_1 \cdot x_2) &= \log(x_1) + \log(x_2), \\ \log(x_1 / x_2) &= \log(x_1) - \log(x_2), \\ \log(x^p) &= p \log(x). \end{aligned}$$

Propriétés de la fonction log

- On s'intéressera aux valeurs de $\log n$ uniquement pour n entier, et en particulier pour n tendant vers l'infini.

Propriétés de la fonction log

- On s'intéressera aux valeurs de $\log n$ uniquement pour n entier, et en particulier pour n tendant vers l'infini.
- Pour n puissance de 2, $\log n$ a un sens combinatoire.

Propriétés de la fonction log

- On s'intéressera aux valeurs de $\log n$ uniquement pour n entier, et en particulier pour n tendant vers l'infini.
- Pour n puissance de 2, $\log n$ 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
$$2^k < n < 2^{k+1}.$$

Propriétés de la fonction log

- On s'intéressera aux valeurs de $\log n$ uniquement pour n entier, et en particulier pour n tendant vers l'infini.
- Pour n puissance de 2, $\log n$ 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 est croissante, on a
$$\log(2^k) < \log n < \log(2^{k+1})$$
et donc $k < \log n < k + 1$.

Propriétés de la fonction log

- On s'intéressera aux valeurs de $\log n$ uniquement pour n entier, et en particulier pour n tendant vers l'infini.
- Pour n puissance de 2, $\log n$ 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 est croissante, on a
$$\log(2^k) < \log n < \log(2^{k+1})$$
et donc $k < \log n < k + 1$.
- Par exemple,
$$10 < \log 2000 < 11, \quad \text{(puisque } 1024 < 2000 < 2048\text{)}.$$

Propriétés de la fonction log

- On s'intéressera aux valeurs de $\log n$ uniquement pour n entier, et en particulier pour n tendant vers l'infini.
- Pour n puissance de 2, $\log n$ 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 est croissante, on a
$$\log(2^k) < \log n < \log(2^{k+1})$$
et donc $k < \log n < k + 1$.
- Par exemple,
$$10 < \log 2000 < 11, \quad \text{(puisque } 1024 < 2000 < 2048\text{)}.$$
- Pour n un entier positif quelconque, avec au plus $\log(n) + 1$ divisions par 2, on est sûrs d'arriver à un nombre ≤ 1.

Comportement de la fonction log à l'infini

- $\lim_{n \to \infty} \log n = +\infty$

Pour des puissances rationnelles $p < q$,

$$(\log n)^p = \mathcal{O}\big((\log n)^q\big) \text{ et } (\log n)^q \neq \mathcal{O}\big((\log n)^p\big).$$

Comportement de la fonction log à l'infini

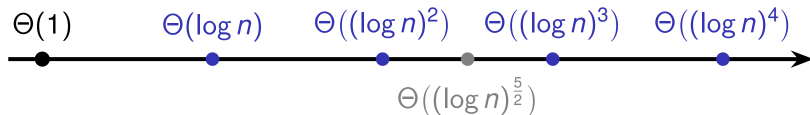
- $\lim_{n\to\infty} \log n = +\infty$

Pour des puissances rationnelles $p < q$,

$$(\log n)^p = \mathcal{O}\big((\log n)^q\big) \text{ et } (\log n)^q \neq \mathcal{O}\big((\log n)^p\big).$$

- Par exemple,

$$\log n = \mathcal{O}\big((\log n)^2\big) \text{ et } (\log n)^2 \neq \mathcal{O}\big(\log n\big).$$



Comportement de la fonction log à l'infini

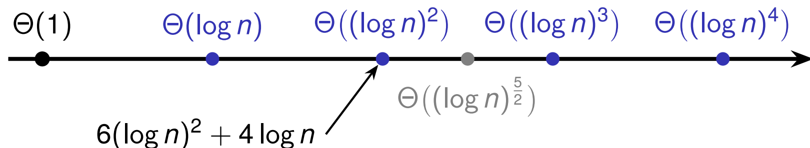
- $\lim_{n \to \infty} \log n = +\infty$

Pour des puissances rationnelles $p < q$,

$$(\log n)^p = \mathcal{O}\big((\log n)^q\big) \text{ et } (\log n)^q \neq \mathcal{O}\big((\log n)^p\big).$$

- Par exemple,

$$\log n = \mathcal{O}\big((\log n)^2\big) \text{ et } (\log n)^2 \neq \mathcal{O}\big(\log n\big).$$



Comportement de la fonction log à l'infini

- $\lim_{n \to \infty} \log n = +\infty$
- Mais quand n tend vers l'infini, $\log n$ croît vers l'infini **beaucoup plus lentement que n** :

$$\lim_{n \to \infty} \frac{\log n}{n} = 0.$$

- En particulier, $\log n = \mathcal{O}(n)$, et $n \neq \mathcal{O}(\log n)$. [1]

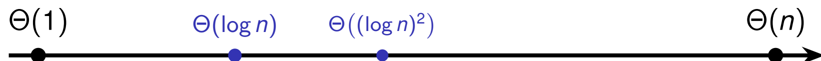
1. Preuve dans la série.

Comportement de la fonction log à l'infini

- $\lim_{n \to \infty} \log n = +\infty$
- Mais quand n tend vers l'infini, $\log n$ croît vers l'infini **beaucoup plus lentement que n** :

$$\lim_{n \to \infty} \frac{\log n}{n} = 0.$$

- En particulier, $\log n = \mathcal{O}(n)$, et $n \neq \mathcal{O}(\log n)$. [1]



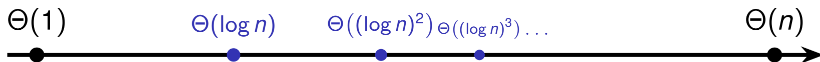
1. Preuve dans la série.

Comportement de la fonction log à l'infini

- $\lim_{n \to \infty} \log n = +\infty$
- Mais quand n tend vers l'infini, $\log n$ croît vers l'infini **beaucoup plus lentement que n** :

$$\lim_{n \to \infty} \frac{\log n}{n} = 0.$$

- En particulier, $\log n = \mathcal{O}(n)$, et $n \neq \mathcal{O}(\log n)$. [1]



Après combien de temps atteindrons-nous $\Theta(n)$?

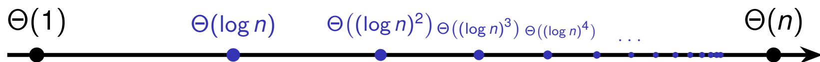
1. Preuve dans la série.

Comportement de la fonction log à l'infini

- $\lim_{n\to\infty} \log n = +\infty$
- Mais quand n tend vers l'infini, $\log n$ croît vers l'infini **beaucoup plus lentement que n** :

$$\lim_{n\to\infty} \frac{\log n}{n} = 0.$$

- En particulier, $\log n = \mathcal{O}(n)$, et $n \neq \mathcal{O}(\log n)$. [1]



Après combien de temps atteindrons-nous $\Theta(n)$?

Jamais !

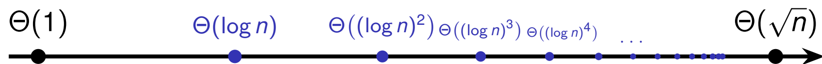
1. Preuve dans la série.

Comportement de la fonction log à l'infini

- $\lim_{n\to\infty} \log n = +\infty$
- Mais quand n tend vers l'infini, $\log n$ croît vers l'infini **beaucoup plus lentement que n** :

$$\lim_{n\to\infty} \frac{\log n}{n} = 0.$$

- En particulier, $\log n = \mathcal{O}(n)$, et $n \neq \mathcal{O}(\log n)$. [1]



Après combien de temps atteindrons-nous $\Theta(n)$?

Jamais !

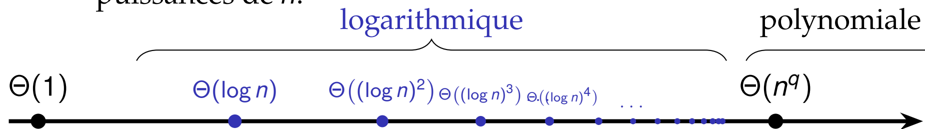
1. Preuve dans la série.

Comportement de la fonction log à l'infini

- Pour toute puissance p, et pour toute puissance strictement positive q,
$$\lim_{n\to\infty} \frac{(\log n)^p}{n^q} = 0.$$
- En particulier $(\log n)^p = \mathcal{O}(n^q)$ et $n^q \neq \mathcal{O}\big((\log n)^p\big)$.
Par exemple,
  - $(\log n)^2 = \mathcal{O}(n)$
  - $(\log n)^{10} = \mathcal{O}(n)$
  - $(\log n)^{10} = \mathcal{O}(\sqrt{n})$

Comportement de la fonction log à l'infini

- Pour toute puissance p, et pour toute puissance strictement positive q,
$$\lim_{n\to\infty} \frac{(\log n)^p}{n^q} = 0.$$
- En particulier $(\log n)^p = \mathcal{O}(n^q)$ et $n^q \neq \mathcal{O}\big((\log n)^p\big)$.
Par exemple,
  - $(\log n)^2 = \mathcal{O}(n)$
  - $(\log n)^{10} = \mathcal{O}(n)$
  - $(\log n)^{10} = \mathcal{O}(\sqrt{n})$
- $\log n$ et ses puissances ont une **croissance logarithmique**, qui est dominée par la **croissance polynomiale** des puissances de n.



De retour à la recherche binaire...

Recherche binaire - temps de parcours

```python
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 au 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

```
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 $\leq \ell/2$.
  - Une fois que cette liste est de taille 0 ou 1 (ce qui prend, au plus, $\log(n) + 1$ itérations), on sort de la boucle dans au plus une itération.

Liste non triée

- 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...

Croissance exponentielle

La fonction a^n

- Pour $a > 1$, on définit la fonction exponentielle de base a :

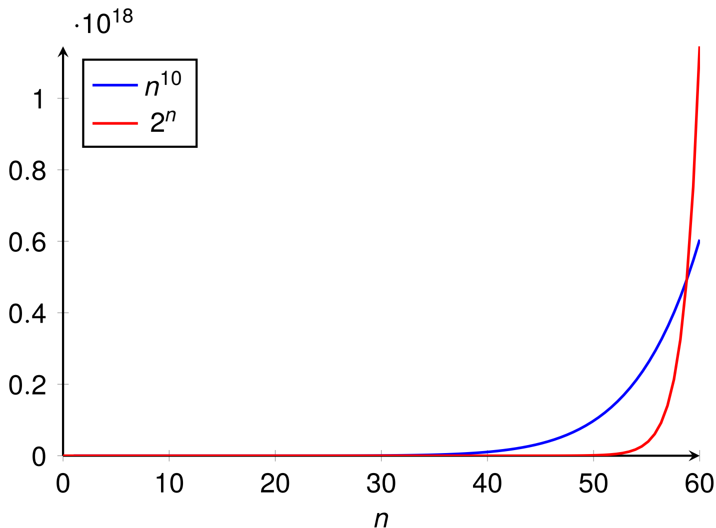
$$a^n = \underbrace{a \cdot a \cdots a}_{n \text{ fois}}$$

- Pour toute puissance rationnelle p,

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

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

Croissance exponentielle

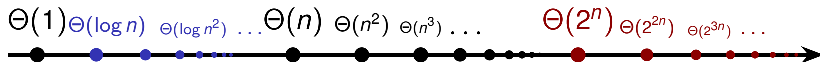


La hiérarchie de la complexité II

Comme précédemment, nous pouvons considérer les puissances de 2^n. Ici, cela a pour effet de multiplier n par une constante dans l'exposant, ce qui crée une nouvelle classe de complexité.

$$(2^n)^k = 2^{k \cdot n}.$$

Les classes de complexité vues jusqu'à présent peuvent donc être résumées dans la hiérarchie suivante :

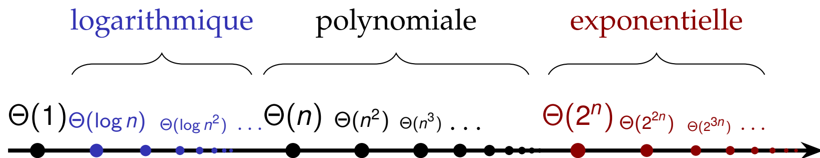


La hiérarchie de la complexité II

Comme précédemment, nous pouvons considérer les puissances de 2^n. Ici, cela a pour effet de multiplier n par une constante dans l'exposant, ce qui crée une nouvelle classe de complexité.

$$(2^n)^k = 2^{k \cdot n}.$$

Les classes de complexité vues jusqu'à présent peuvent donc être résumées dans la hiérarchie suivante :

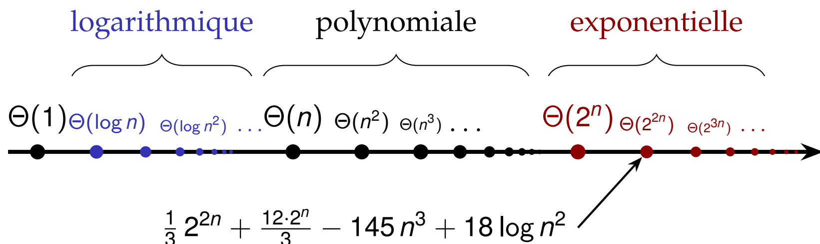


La hiérarchie de la complexité II

Comme précédemment, nous pouvons considérer les puissances de 2^n. Ici, cela a pour effet de multiplier n par une constante dans l'exposant, ce qui crée une nouvelle classe de complexité.

$$(2^n)^k = 2^{k \cdot n}.$$

Les classes de complexité vues jusqu'à présent peuvent donc être résumées dans la hiérarchie suivante :

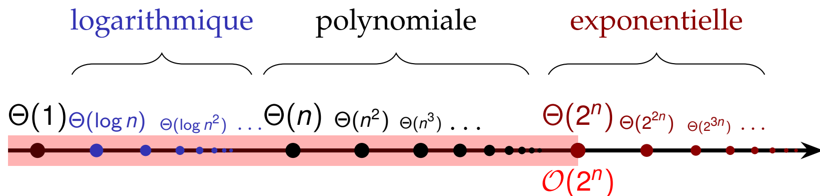


La hiérarchie de la complexité II

Comme précédemment, nous pouvons considérer les puissances de 2^n. Ici, cela a pour effet de multiplier n par une constante dans l'exposant, ce qui crée une nouvelle classe de complexité.

$$(2^n)^k = 2^{k \cdot n}.$$

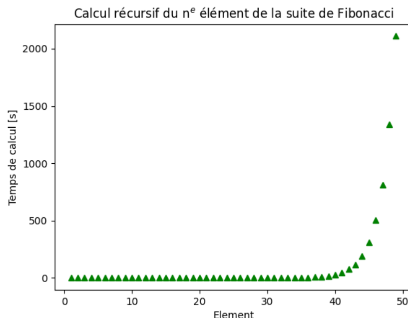
Les classes de complexité vues jusqu'à présent peuvent donc être résumées dans la hiérarchie suivante :



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 :



Algorithmes exponentiels - somme de sous-ensembles

- Un autre exemple : le problème de la somme de sous-ensembles[2] : 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 $\Omega(2^n)$.

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