

Mesurer l'efficacité d'un algorithme

CPES2: Algorithmique et Applications

Pierre Senellart



9 septembre 2022

Plan

Algorithmique et programmation

De l'algorithme au code machine

Complexité algorithmique

Références

Algorithmique

- Vient du nom de محمد بن موسى الخوارزمي

Algorithmique

- Vient du nom de محمد بن موسى الخوارزمي (Muhammad ibn Musa al-Khwarizmi), un scientifique perse du IX^e siècle

Algorithmique

- Vient du nom de محمد بن موسى الخوارزمي (Muhammad ibn Musa al-Khwarizmi), un scientifique perse du IX^e siècle
- ... qui est aussi à l'origine du mot algèbre (الجبر, *remise en place* (des os fracturés) en arabe), du titre de l'un de ses livres sur la résolution d'équations

Algorithmique

- Vient du nom de محمد بن موسى الخوارزمي (Muhammad ibn Musa al-Khwarizmi), un scientifique perse du IX^e siècle
- ... qui est aussi à l'origine du mot algèbre (الجبر, *remise en place* (des os fracturés) en arabe), du titre de l'un de ses livres sur la résolution d'équations
- Un algorithme est une **spécification formelle** de la manière dont résoudre un problème donné : à partir d'une **entrée**, comment produire la **sortie** correspondant à la solution d'un problème, en combinant des opérations élémentaires

Algorithmique

- Vient du nom de **محمد بن موسى الخوارزمي** (Muhammad ibn Musa al-Khwarizmi), un scientifique perse du IX^e siècle
- ... qui est aussi à l'origine du mot algèbre (**الجبر**, *remise en place* (des os fracturés) en arabe), du titre de l'un de ses livres sur la résolution d'équations
- Un algorithme est une **spécification formelle** de la manière dont résoudre un problème donné : à partir d'une **entrée**, comment produire la **sortie** correspondant à la solution d'un problème, en combinant des opérations élémentaires
- L'algorithmique est l'**étude des algorithmes** : conception de l'algorithme, analyse de sa performance, démonstration de sa correction, etc.

Algorithmique

- Vient du nom de محمد بن موسى الخوارزمي (Muhammad ibn Musa al-Khwarizmi), un scientifique perse du IX^e siècle
- ... qui est aussi à l'origine du mot algèbre (الجبر, *remise en place* (des os fracturés) en arabe), du titre de l'un de ses livres sur la résolution d'équations
- Un algorithme est une **spécification formelle** de la manière dont résoudre un problème donné : à partir d'une **entrée**, comment produire la **sortie** correspondant à la solution d'un problème, en combinant des opérations élémentaires
- L'algorithmique est l'**étude des algorithmes** : conception de l'algorithme, analyse de sa performance, démonstration de sa correction, etc.
- La **programmation** est la manière de transformer (on dit **implémenter**) un algorithme en un code dans un langage informatique, afin d'exécuter l'algorithme sur des données concrètes

Algorithmique et programmation

- Tout algorithme est **implémentable** : il doit être décrit en des termes suffisamment précis pour qu'il n'y ait pas d'ambiguïté sur comment le transformer en programme
- ... mais cela ne veut **pas** dire que le programme implémentant l'algorithme est **facile à écrire**, car le programmeur doit prendre en compte les limitations de la machine, les particularités du langage, des objets de bas de niveau et pas des concepts de haut niveau, etc.
- **Algorithme** : abstraction de ce qui est **implémentable**
- Le langage de programmation n'a pas d'influence sur ce qui est implémentable ; tous les langages de programmation ont le même **pouvoir d'expression** (on dit qu'ils sont Turing-complet, on expliquera)
- ... mais un langage a un impact sur la **facilité** (cf. <https://pierre.senellart.com/other/languages/languages.xml>) ou sur l'**efficacité** de l'implémentation

Structure de données

- Élément de **base** utilisé dans des algorithmes plus complexes, réutilisé dans des algorithmes divers pour résoudre différents problèmes
- Spécification formelle d'un **objet** mathématique abstrait (liste, ensemble, graphe, matrice, etc.), des **opérations** possibles sur cet objet (insertion, énumération, inversion, etc.) et des **algorithmes** les réalisant
- Élément **implémentable**, souvent sous la forme d'une **classe** en programmation orientée objet
- Souvent possible de concevoir différentes structures de données pour le même objet mathématique, plus ou moins **efficaces** pour une tâche donnée

Plan

Algorithmique et programmation

De l'algorithme au code machine

Complexité algorithmique

Références

Qui fait quoi ?

- L'**algorithmicien** conçoit, décrit et analyse un algorithme

Qui fait quoi ?

- L'**algorithmicien** conçoit, décrit et analyse un algorithme
- Le **programmeur** implémente cet algorithme dans un langage de programmation donné, pour un environnement donné

Qui fait quoi ?

- L'**algorithmicien** conçoit, décrit et analyse un algorithme
- Le **programmeur** implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages **interprétés** (comme Python) :

Qui fait quoi ?

- L'**algorithmicien** conçoit, décrit et analyse un algorithme
- Le **programmeur** implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages **interprétés** (comme Python) :
 - le **compilateur** transforme le programme en **bytecode** spécifique au langage

Qui fait quoi ?

- L'**algorithmicien** conçoit, décrit et analyse un algorithme
- Le **programmeur** implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages **interprétés** (comme Python) :
 - le **compilateur** transforme le programme en **bytecode** spécifique au langage
 - l'**interpréteur** exécute le bytecode instruction par instruction

Qui fait quoi ?

- L'**algorithmicien** conçoit, décrit et analyse un algorithme
- Le **programmeur** implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages **interprétés** (comme Python) :
 - le **compilateur** transforme le programme en **bytecode** spécifique au langage
 - l'**interpréteur** exécute le bytecode instruction par instruction
- Dans les langages **compilés** (comme C) :

Qui fait quoi ?

- L'**algorithmicien** conçoit, décrit et analyse un algorithme
- Le **programmeur** implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages **interprétés** (comme Python) :
 - le **compilateur** transforme le programme en **bytecode** spécifique au langage
 - l'**interpréteur** exécute le bytecode instruction par instruction
- Dans les langages **compilés** (comme C) :
 - le **compilateur** transforme le programme en code assembleur spécifique à la machine

Qui fait quoi ?

- L'**algorithmicien** conçoit, décrit et analyse un algorithme
- Le **programmeur** implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages **interprétés** (comme Python) :
 - le **compilateur** transforme le programme en **bytecode** spécifique au langage
 - l'**interpréteur** exécute le bytecode instruction par instruction
- Dans les langages **compilés** (comme C) :
 - le **compilateur** transforme le programme en code assembleur spécifique à la machine
 - l'**assembleur** transforme le code assembleur en code machine directement exécutable par le processeur

Exemple d'algorithme

Entrée : Tableau T avec n éléments distincts, un élément x

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for  
6: return pas trouvé
```

Programme Python

```
def find(x):  
    for i in range(0, n):  
        if T[i] == x:  
            return i  
    return -1
```

Bytecode Python

```
0 SETUP_LOOP                43 (to 46)
3 LOAD_GLOBAL                0 (range)
6 LOAD_CONST                 1 (0)
9 LOAD_GLOBAL                1 (n)
12 CALL_FUNCTION             2
15 GET_ITER
>> 16 FOR_ITER                 26 (to 45)
19 STORE_FAST                1 (i)
22 LOAD_GLOBAL                2 (T)
25 LOAD_FAST                 1 (i)
28 BINARY_SUBSCR
29 LOAD_FAST                 0 (x)
32 COMPARE_OP                 2 (==)
35 POP_JUMP_IF_FALSE         16
38 LOAD_FAST                 1 (i)
41 RETURN_VALUE
42 JUMP_ABSOLUTE             16
>> 45 POP_BLOCK
>> 46 LOAD_CONST                2 (-1)
49 RETURN_VALUE
```

Programme C

```
long find(int x) {  
    for(long i=0; i<=1000; ++i)  
        if(T[i]==x)  
            return i;  
    return -1;  
}
```

Code assembleur x86-64 (compilé depuis C)

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-20], edi
mov     QWORD PTR [rbp-8], 0
.L5:    cmp     QWORD PTR [rbp-8], 1000
        jg     .L2
mov     rax, QWORD PTR [rbp-8]
mov     eax, DWORD PTR [rax*4+0x404060]
cmp     DWORD PTR [rbp-20], eax
jne     .L3
mov     rax, QWORD PTR [rbp-8]
jmp     .L4
.L3:    add     QWORD PTR [rbp-8], 1
        jmp     .L5
.L2:    mov     rax, -1
.L4:    pop     rbp
        ret
```

Code machine x86-64

```
push    rbp                                #55
mov     rbp, rsp                            #48 89 e5
mov     DWORD PTR [rbp-20], edi             #89 7d ec
mov     QWORD PTR [rbp-8], 0                #48 c7 45 f8 00 00 00 00
.L5:   cmp     QWORD PTR [rbp-8], 1000      #48 81 7d f8 e8 03 00 00
jg     .L2                                  #7f 1d
mov     rax, QWORD PTR [rbp-8]              #48 8b 45 f8
mov     eax, DWORD PTR [rax*4+0x404060]     #8b 04 85 60 40 40 00
cmp     DWORD PTR [rbp-20], eax            #39 45 ec
jne     .L3                                  #75 06
mov     rax, QWORD PTR [rbp-8]              #48 8b 45 f8
jmp     .L4                                  #eb 0e
.L3:   add     QWORD PTR [rbp-8], 1         #48 83 45 f8 01
jmp     .L5                                  #eb d9
.L2:   mov     rax, -1                       #48 c7 c0 ff ff ff ff
.L4:   pop     rbp                            #5d
ret                                          #c3
```

Plan

Algorithmique et programmation

De l'algorithme au code machine

Complexité algorithmique

Références

Comment mesurer l'efficacité d'un algorithme ?

- Tentative de **caractériser**, à partir de la description d'un algorithme, l'**efficacité** d'un programme qui l'implémente ; ou, à partir de la description d'un problème, l'efficacité d'un programme qui implémente un algorithme résolvant ce problème
- **Différentes notions** d'efficacité, différentes notions de complexité :
 - Complexité en temps **temps** d'exécution d'un programme
 - Complexité en espace **espace** mémoire occupé par un programme
 - Complexité de communication volume des **données échangées** par un système distribué
 - Complexité descriptive **taille** du plus petit **programme**
 - Complexité de circuit **taille** du plus petit **circuit** électronique implémentant le programme
- Dans ce cours : seulement les deux premiers (et principalement le premier !) – **complexité algorithmique**

Comment calculer la complexité en temps

- On suppose que chaque **opération élémentaire** apparaissant dans la description d'un algorithme :
 - opérations arithmétiques
 - affectations de variable
 - comparaisons
 - tests
 - etc.

prend un **temps élémentaire**, borné par une constante C

- On calcule la somme du nombre d'opérations élémentaires effectuées, **en fonction de la taille de l'entrée n** , p. ex.,
 $42 \times n$
- On en déduit une borne, ici $42 \times n \times C$, sur le temps **total** de l'algorithme

Opérations élémentaires

- Pas défini formellement pour l'instant

Opérations élémentaires

- Pas défini formellement pour l'instant
- Mais on peut penser, par exemple :

Opérations élémentaires

- Pas défini formellement pour l'instant
- Mais on peut penser, par exemple :
 - au nombre d'instructions de bytecode que l'interpréteur exécute (en bornant le temps pris par une instruction par le **temps maximal** pris par toutes les instructions du bytecode)

Opérations élémentaires

- Pas défini formellement pour l'instant
- Mais on peut penser, par exemple :
 - au nombre d'instructions de bytecode que l'interpréteur exécute (en bornant le temps pris par une instruction par le **temps maximal** pris par toutes les instructions du bytecode)
 - au nombre d'instructions assembleur qui seront exécutées par le processeur en un **cycle processeur** (3 milliards par seconde pour un processeur 3 GHz) une fois assemblées

Opérations élémentaires

- Pas défini formellement pour l'instant
- Mais on peut penser, par exemple :
 - au nombre d'instructions de bytecode que l'interpréteur exécute (en bornant le temps pris par une instruction par le **temps maximal** pris par toutes les instructions du bytecode)
 - au nombre d'instructions assembleur qui seront exécutées par le processeur en un **cycle processeur** (3 milliards par seconde pour un processeur 3 GHz) une fois assemblées
- En pratique, **plus compliqué que ça** : pipelining, instructions nécessitant plusieurs cycles, exécution prédictive, etc.

Opérations élémentaires

- Pas défini formellement pour l'instant
- Mais on peut penser, par exemple :
 - au nombre d'instructions de bytecode que l'interpréteur exécute (en bornant le temps pris par une instruction par le **temps maximal** pris par toutes les instructions du bytecode)
 - au nombre d'instructions assembleur qui seront exécutées par le processeur en un **cycle processeur** (3 milliards par seconde pour un processeur 3 GHz) une fois assemblées
- En pratique, **plus compliqué que ça** : pipelining, instructions nécessitant plusieurs cycles, exécution prédictive, etc. Pas grave ! On peut toujours borner par une **constante suffisamment grande** les opérations élémentaires.

Opérations élémentaires

- Pas défini formellement pour l'instant
- Mais on peut penser, par exemple :
 - au nombre d'instructions de bytecode que l'interpréteur exécute (en bornant le temps pris par une instruction par le **temps maximal** pris par toutes les instructions du bytecode)
 - au nombre d'instructions assembleur qui seront exécutées par le processeur en un **cycle processeur** (3 milliards par seconde pour un processeur 3 GHz) une fois assemblées
- En pratique, **plus compliqué que ça** : pipelining, instructions nécessitant plusieurs cycles, exécution prédictive, etc. Pas grave ! On peut toujours borner par une **constante suffisamment grande** les opérations élémentaires.
- Pour les raisonnements théoriques : on peut rendre la notion d'opération élémentaire plus **formelle** (avec la notion de machine de Turing ou de machine de Von Neumann), mais on passe pour l'instant

Notations $O()$, $\Omega()$, $\Theta()$

- Soient $f : \mathbb{N} \rightarrow \mathbb{R}_+$, $g : \mathbb{N} \rightarrow \mathbb{R}_+$ deux fonctions

Notations $O()$, $\Omega()$, $\Theta()$

- Soient $f : \mathbb{N} \rightarrow \mathbb{R}_+$, $g : \mathbb{N} \rightarrow \mathbb{R}_+$ deux fonctions
- On écrit $f(n) \in O(g(n))$ (ou $f(n) = O(g(n))$) si

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \leq \alpha g(n)$$

Notations $O()$, $\Omega()$, $\Theta()$

- Soient $f : \mathbb{N} \rightarrow \mathbb{R}_+$, $g : \mathbb{N} \rightarrow \mathbb{R}_+$ deux fonctions
- On écrit $f(n) \in O(g(n))$ (ou $f(n) = O(g(n))$) si

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \leq \alpha g(n)$$

- On écrit $f(n) \in \Omega(g(n))$ (ou $f(n) = \Omega(g(n))$) si

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \geq \alpha g(n)$$

Notations $O()$, $\Omega()$, $\Theta()$

- Soient $f : \mathbb{N} \rightarrow \mathbb{R}_+$, $g : \mathbb{N} \rightarrow \mathbb{R}_+$ deux fonctions
- On écrit $f(n) \in O(g(n))$ (ou $f(n) = O(g(n))$) si

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \leq \alpha g(n)$$

- On écrit $f(n) \in \Omega(g(n))$ (ou $f(n) = \Omega(g(n))$) si

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \geq \alpha g(n)$$

- On écrit $f(n) \in \Theta(g(n))$ (or $f(n) = \Theta(g(n))$) si

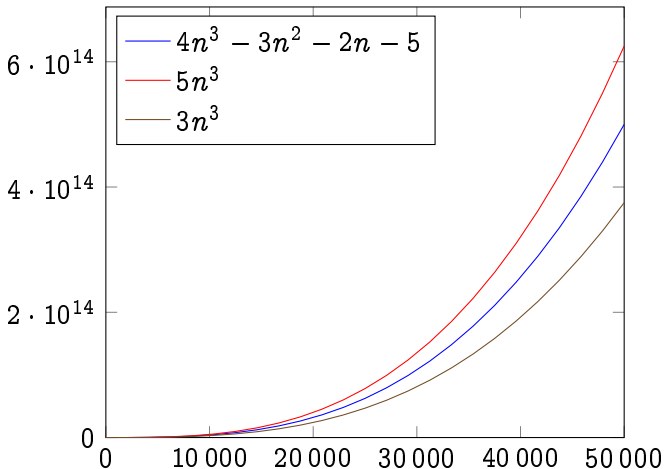
$$f(n) = O(g(n)) \quad \text{et} \quad f(n) = \Omega(g(n))$$

Exemples de $O()$, $\Omega()$, $\Theta()$ – 1/2

Si P est un polynôme de degré k , $P(n) \in \Theta(n^k)$

Exemples de $O()$, $\Omega()$, $\Theta()$ – 1/2

Si P est un polynôme de degré k , $P(n) \in \Theta(n^k)$



Exemples de $O()$, $\Omega()$, $\Theta()$ – 2/2

- $\log n \in O(n)$ mais $\log n \notin \Omega(n)$

Exemples de $O()$, $\Omega()$, $\Theta()$ – 2/2

- $\log n \in O(n)$ mais $\log n \notin \Omega(n)$
- $n \log n \in O(n^2)$, $n \log n \in \Omega(n)$, mais $n \log n \notin \Omega(n^2)$

Exemples de $O()$, $\Omega()$, $\Theta()$ – 2/2

- $\log n \in O(n)$ mais $\log n \notin \Omega(n)$
- $n \log n \in O(n^2)$, $n \log n \in \Omega(n)$, mais $n \log n \notin \Omega(n^2)$
- $42n^2 \log n + 23n^2 \in \Theta(n^2 \log n)$

Exemples de $O()$, $\Omega()$, $\Theta()$ – 2/2

- $\log n \in O(n)$ mais $\log n \notin \Omega(n)$
- $n \log n \in O(n^2)$, $n \log n \in \Omega(n)$, mais $n \log n \notin \Omega(n^2)$
- $42n^2 \log n + 23n^2 \in \Theta(n^2 \log n)$
- pour tout polynôme P , $P(n) \in O(2^n)$ mais $P(n) \notin \Omega(2^n)$

Exemples de $O()$, $\Omega()$, $\Theta()$ – 2/2

- $\log n \in O(n)$ mais $\log n \notin \Omega(n)$
- $n \log n \in O(n^2)$, $n \log n \in \Omega(n)$, mais $n \log n \notin \Omega(n^2)$
- $42n^2 \log n + 23n^2 \in \Theta(n^2 \log n)$
- pour tout polynôme P , $P(n) \in O(2^n)$ mais $P(n) \notin \Omega(2^n)$
- Si $f(n) \in O(\varphi(n))$ et $g(n) \in O(\psi(n))$, alors
 $f(n) + g(n) \in O(\varphi(n) + \psi(n))$ et
 $f(n) \times g(n) \in O(\varphi(n) \times \psi(n))$

Exemples de $O()$, $\Omega()$, $\Theta()$ – 2/2

- $\log n \in O(n)$ mais $\log n \notin \Omega(n)$
- $n \log n \in O(n^2)$, $n \log n \in \Omega(n)$, mais $n \log n \notin \Omega(n^2)$
- $42n^2 \log n + 23n^2 \in \Theta(n^2 \log n)$
- pour tout polynôme P , $P(n) \in O(2^n)$ mais $P(n) \notin \Omega(2^n)$
- Si $f(n) \in O(\varphi(n))$ et $g(n) \in O(\psi(n))$, alors
 $f(n) + g(n) \in O(\varphi(n) + \psi(n))$ et
 $f(n) \times g(n) \in O(\varphi(n) \times \psi(n))$
- Si $f(n) \in \Omega(\varphi(n))$ et $g(n) \in \Omega(\psi(n))$, alors
 $f(n) + g(n) \in \Omega(\varphi(n) + \psi(n))$ et
 $f(n) \times g(n) \in \Omega(\varphi(n) \times \psi(n))$

Complexité (en temps) asymptotique

- On utilise les notations $O()$, $\Omega()$, $\Theta()$ et les bornes qui ont été établies pour indiquer la complexité d'un algorithme en **négligeant** C et les autres constantes
- Par exemple, si le temps τ de chaque opération élémentaire est **borné** par :

$$C_1 \leq \tau \leq C_2$$

- ... et si **sur toutes les entrées**, l'algorithme A fait $42 \times n$ opérations élémentaires, alors :

$$42 \times C_1 \times n \leq T(\mathcal{A}, n) \leq 42 \times C_2 \times n$$

et donc $T(\mathcal{A}, n) = \Theta(n)$

Complexité dans le pire cas, dans le cas moyen

- Habituellement, on cherche une borne supérieure sur le temps d'un algorithme, et on regarde la complexité **dans le pire des cas** : une borne supérieure qui est vraie sur n'importe quelle entrée
- Parfois trop restrictif, donc on regarde le **cas moyen** : en moyenne, pour toutes les entrées d'une taille donnée, quelle est une borne sur la complexité ?
- Ce cas moyen fait l'hypothèse que toutes les entrées ont la **même probabilité**, ce qui est **débatable**

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$
- n comparaisons de $T[i]$ avec x

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$
- n comparaisons de $T[i]$ avec x
- 1 retour

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$
- n comparaisons de $T[i]$ avec x
- 1 retour

$4n + 1$, c.-à-d., $O(n)$

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

Cas moyen

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$
- n comparaisons de $T[i]$ avec x
- 1 retour

$4n + 1$, c.-à-d., $O(n)$

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

Cas moyen

(x en moyenne en position $n/2$)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$
- n comparaisons de $T[i]$ avec x
- 1 retour

$4n + 1$, c.-à-d., $O(n)$

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$
- n comparaisons de $T[i]$ avec x
- 1 retour

$4n + 1$, c.-à-d., $O(n)$

Cas moyen

(x en moyenne en position $n/2$)

- $n/2$ affectations de i

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$
- n comparaisons de $T[i]$ avec x
- 1 retour

$4n + 1$, c.-à-d., $O(n)$

Cas moyen

(x en moyenne en position $n/2$)

- $n/2$ affectations de i
- $n/2$ comparaisons de i avec n

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$
- n comparaisons de $T[i]$ avec x
- 1 retour

$4n + 1$, c.-à-d., $O(n)$

Cas moyen

(x en moyenne en position $n/2$)

- $n/2$ affectations de i
- $n/2$ comparaisons de i avec n
- $n/2$ accès à $T[i]$

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$
- n comparaisons de $T[i]$ avec x
- 1 retour

$4n + 1$, c.-à-d., $O(n)$

Cas moyen

(x en moyenne en position $n/2$)

- $n/2$ affectations de i
- $n/2$ comparaisons de i avec n
- $n/2$ accès à $T[i]$
- $n/2$ comparaisons de $T[i]$ avec x

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$
- n comparaisons de $T[i]$ avec x
- 1 retour

$4n + 1$, c.-à-d., $O(n)$

Cas moyen

(x en moyenne en position $n/2$)

- $n/2$ affectations de i
- $n/2$ comparaisons de i avec n
- $n/2$ accès à $T[i]$
- $n/2$ comparaisons de $T[i]$ avec x
- 1 retour

Exemple simple : recherche dans un tableau

Entrée : Tableau T avec n éléments distincts, un élément x de T

Sortie : la position de x dans T

```
1: for  $i \leftarrow 0$  to  $n - 1$  do  
2:   if  $T[i] = x$  then  
3:     return  $i$   
4:   end if  
5: end for
```

Combien d'opérations élémentaires ?

Pire cas

(x en dernière position)

- n affectations de i
- n comparaisons de i avec n
- n accès à $T[i]$
- n comparaisons de $T[i]$ avec x
- 1 retour

$4n + 1$, c.-à-d., $O(n)$

Cas moyen

(x en moyenne en position $n/2$)

- $n/2$ affectations de i
- $n/2$ comparaisons de i avec n
- $n/2$ accès à $T[i]$
- $n/2$ comparaisons de $T[i]$ avec x
- 1 retour

$2n + 1$, c.-à-d., $O(n)$

Complexité asymptotique et efficacité réelle

En pratique :

- La complexité asymptotique **importe**. Un algorithme en $\Theta(n)$ est plus lent qu'un algorithme en $O(\log n)$ *si n est suffisamment grand*

Complexité asymptotique et efficacité réelle

En pratique :

- La complexité asymptotique **importe**. Un algorithme en $\Theta(n)$ est plus lent qu'un algorithme en $O(\log n)$ *si n est suffisamment grand*
- Parfois, un algorithme en $\Omega(n^2)$ (c.-à-d., $\geq \alpha n^2$) peut être plus efficace qu'un algorithme en $O(n)$ (i.e., $\leq \beta n$) pour **une taille d'entrée n courante**, parce que $\alpha \ll \beta$

Complexité asymptotique et efficacité réelle

En pratique :

- La complexité asymptotique **importe**. Un algorithme en $\Theta(n)$ est plus lent qu'un algorithme en $O(\log n)$ *si n est suffisamment grand*
- Parfois, un algorithme en $\Omega(n^2)$ (c.-à-d., $\geq \alpha n^2$) peut être plus efficace qu'un algorithme en $O(n)$ (i.e., $\leq \beta n$) pour **une taille d'entrée n courante**, parce que $\alpha \ll \beta$
- Parfois, la complexité **dans le pire des cas** est haute, mais la complexité dans le cas moyen est faible, est la seule chose qui compte en pratique

Complexité asymptotique et efficacité réelle

En pratique :

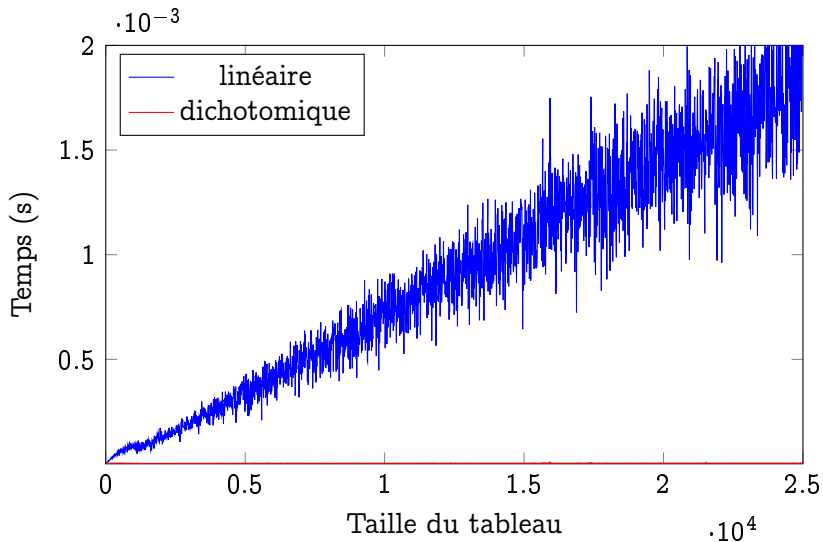
- La complexité asymptotique **importe**. Un algorithme en $\Theta(n)$ est plus lent qu'un algorithme en $O(\log n)$ *si n est suffisamment grand*
- Parfois, un algorithme en $\Omega(n^2)$ (c.-à-d., $\geq \alpha n^2$) peut être plus efficace qu'un algorithme en $O(n)$ (i.e., $\leq \beta n$) pour **une taille d'entrée n courante**, parce que $\alpha \ll \beta$
- Parfois, la complexité **dans le pire des cas** est haute, mais la complexité dans le cas moyen est faible, est la seule chose qui compte en pratique
- Le **langage de programmation utilisé** a un vrai impact sur les performances (mais, généralement, par un facteur constant, potentiellement grand)

Complexité asymptotique et efficacité réelle

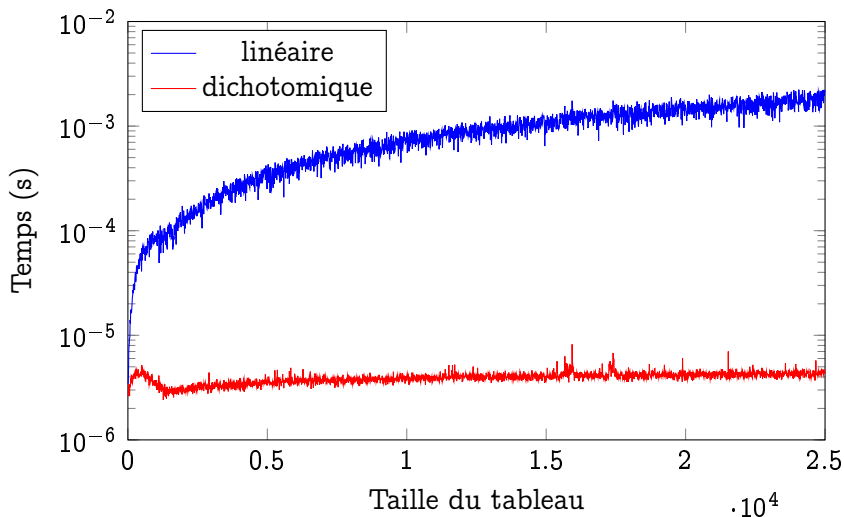
En pratique :

- La complexité asymptotique **importe**. Un algorithme en $\Theta(n)$ est plus lent qu'un algorithme en $O(\log n)$ *si n est suffisamment grand*
- Parfois, un algorithme en $\Omega(n^2)$ (c.-à-d., $\geq \alpha n^2$) peut être plus efficace qu'un algorithme en $O(n)$ (i.e., $\leq \beta n$) pour **une taille d'entrée n courante**, parce que $\alpha \ll \beta$
- Parfois, la complexité **dans le pire des cas** est haute, mais la complexité dans le cas moyen est faible, est la seule chose qui compte en pratique
- Le **langage de programmation utilisé** a un vrai impact sur les performances (mais, généralement, par un facteur constant, potentiellement grand)
- La complexité en temps ne dit rien sur le potentiel de parallélisation ou de distribution d'un algorithme – d'**autres** notions de complexité sont nécessaires

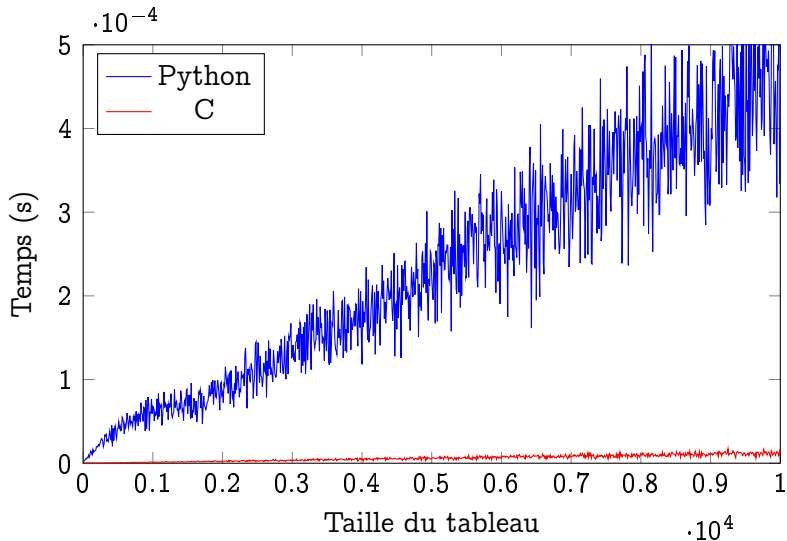
Recherche linéaire vs recherche dichotomique



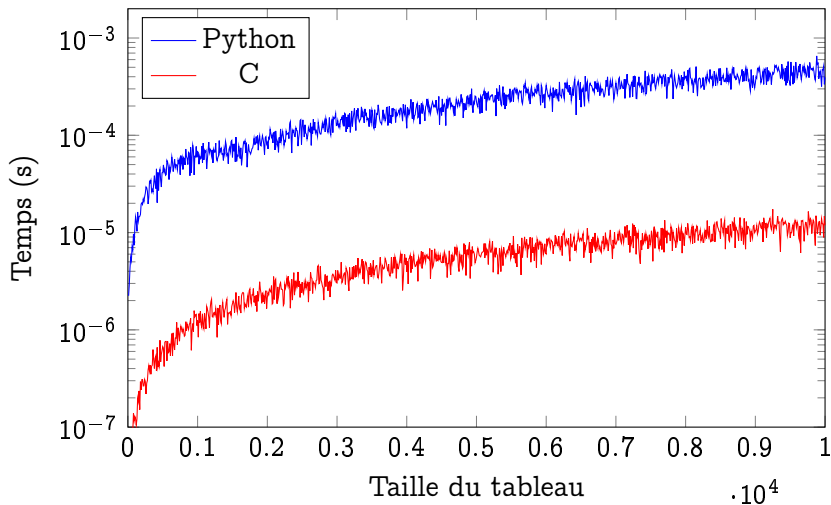
Recherche linéaire vs recherche dichotomique



Recherche linéaire : Python vs C



Recherche linéaire : Python vs C



En pratique : résoudre un problème efficacement

- **Concevoir** (dans sa tête ou sur papier) un algorithme le plus efficace possible, en utilisant les structures de données les plus appropriées

En pratique : résoudre un problème efficacement

- **Concevoir** (dans sa tête ou sur papier) un algorithme le plus efficace possible, en utilisant les structures de données les plus appropriées
- Calculer sa **complexité algorithmique** ($O()$ ou même $\Theta()$), dans le pire cas (et éventuellement dans le cas moyen)

En pratique : résoudre un problème efficacement

- **Concevoir** (dans sa tête ou sur papier) un algorithme le plus efficace possible, en utilisant les structures de données les plus appropriées
- Calculer sa **complexité algorithmique** ($O()$ ou même $\Theta()$), dans le pire cas (et éventuellement dans le cas moyen)
- **Implémenter** l'algorithme dans un langage de programmation le plus fidèlement possible, en tenant compte des particularités de l'environnement

En pratique : résoudre un problème efficacement

- **Concevoir** (dans sa tête ou sur papier) un algorithme le plus efficace possible, en utilisant les structures de données les plus appropriées
- Calculer sa **complexité algorithmique** ($O()$ ou même $\Theta()$), dans le pire cas (et éventuellement dans le cas moyen)
- **Implémenter** l'algorithme dans un langage de programmation le plus fidèlement possible, en tenant compte des particularités de l'environnement
- **Tester** la correction et l'efficacité de l'algorithme sur des petits exemples et des exemples réels, vérifier expérimentalement la complexité algorithmique

En pratique : résoudre un problème efficacement

- **Concevoir** (dans sa tête ou sur papier) un algorithme le plus efficace possible, en utilisant les structures de données les plus appropriées
- Calculer sa **complexité algorithmique** ($O()$ ou même $\Theta()$), dans le pire cas (et éventuellement dans le cas moyen)
- **Implémenter** l'algorithme dans un langage de programmation le plus fidèlement possible, en tenant compte des particularités de l'environnement
- **Tester** la correction et l'efficacité de l'algorithme sur des petits exemples et des exemples réels, vérifier expérimentalement la complexité algorithmique
- Faire du **profilage** de code pour déterminer les parties du programmes dans lesquelles le plus de temps est passé ; les optimiser (éventuellement en revenant à la conception de l'algorithme pour ces parties)

Complexité en espace

- Similaire à la complexité en temps, sauf que l'on mesure les **usages élémentaires de la mémoire** au lieu du temps des opérations élémentaires
- On fait souvent des **hypothèses simplificatrices**, comme le fait que n'importe quel entier tient en espace constant
- On **ne compte pas** l'espace dont on a besoin pour représenter l'entrée
- On utilise aussi $O()$, $\Omega()$, $\Theta()$ pour résumer la complexité **asymptotique**
- Par exemple, pour la recherche linéaire dans un tableau, la complexité en espace est $O(1)$: on a juste besoin de stocker la variable i en mémoire (en plus des entrées T et x), ce qui nécessite un espace élémentaire, indépendant de la taille de l'entrée

Plan

Algorithmique et programmation

De l'algorithme au code machine

Complexité algorithmique

Références

Références

- **Généralités** d'algorithmique : Chap. 1 de [Cormen et al., 2009, 2010]
- Bases d'**analyse de complexité** d'un algorithme : Chap. 2 et 3 de [Cormen et al., 2009, 2010]

Bibliographie

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. ISBN 978-0-262-03384-8. URL

<http://mitpress.mit.edu/books/introduction-algorithms>.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Algorithmique*. Dunod, 3rd edition, 2010. ISBN 978-2-100-54526-1. URL

<https://www.dunod.com/sciences-techniques/algorithmique-cours-avec-957-exercices-et-158-problemes>.