Efficacité d'un algorithme & Collections Python

CPES DAC: Introduction à l'algorithmique

Pierre Senellart

26 septembre 2023

Plan

Terminaison, correction, efficacité

Terminaison Un algorithme termine si le calcul finit en un nombre fini d'étapes, quel que soit l'entrée

Terminaison, correction, efficacité

Terminaison Un algorithme termine si le calcul finit en un nombre fini d'étapes, quel que soit l'entrée

Correction Un algorithme est correct si la valeur retournée est la solution du problème, quel que soit l'entrée

Terminaison, correction, efficacité

Terminaison Un algorithme termine si le calcul finit en un nombre fini d'étapes, quel que soit l'entrée

Correction Un algorithme est correct si la valeur retournée est la solution du problème, quel que soit l'entrée

Efficacité Comment définir ça ?

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 préditive, 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 préditive, 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 préditive, 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

Plan

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

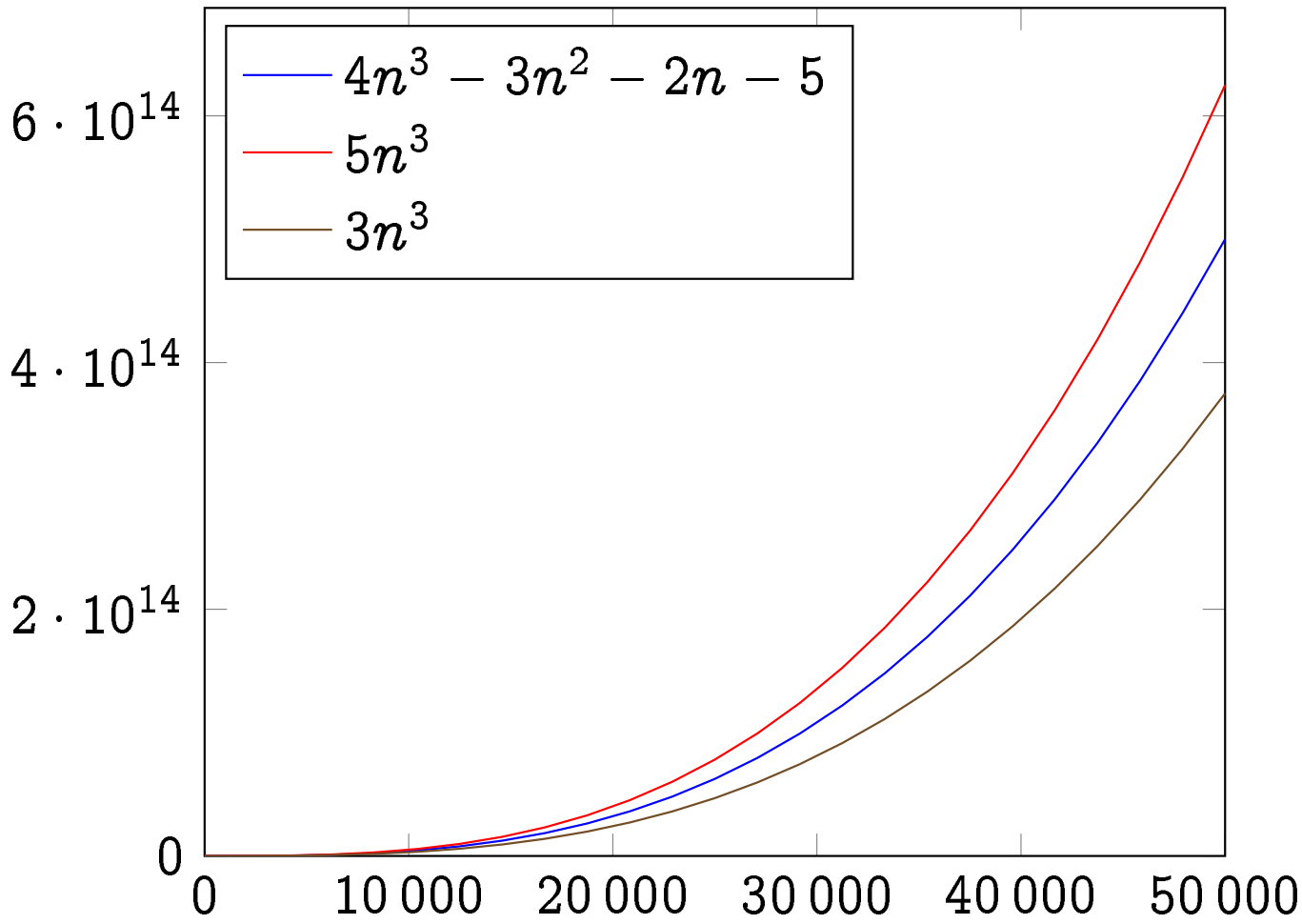
$$f(n) = O(g(n)) \quad \text{et} \quad f(n) = \Omega(g(n))$$

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
  $f(n) \times g(n) \in \Omega(\varphi(n) \times \psi(n))$

Remarques supplémentaires

- Comme $\log_b n = \frac{\ln n}{\ln b}$ par définition, $\log_b n \in \Theta(\ln n)$ pour tout b : on peut donc omettre la base des logs et écrire $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$

Remarques supplémentaires

- Comme $\log_b n = \frac{\ln n}{\ln b}$ par définition, $\log_b n \in \Theta(\ln n)$ pour tout b : on peut donc omettre la base des logs et écrire $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$
- Si $f(n) \in O(g(n))$, $g(n) \in \Omega(f(n))$

Remarques supplémentaires

- Comme $\log_b n = \frac{\ln n}{\ln b}$ par définition, $\log_b n \in \Theta(\ln n)$ pour tout b : on peut donc omettre la base des logs et écrire $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$
- Si $f(n) \in O(g(n))$, $g(n) \in \Omega(f(n))$
- Si $f(n) \in \Omega(g(n))$, $g(n) \in O(f(n))$

Remarques supplémentaires

- Comme $\log_b n = \frac{\ln n}{\ln b}$ par définition, $\log_b n \in \Theta(\ln n)$ pour tout b : on peut donc omettre la base des logs et écrire $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$
- Si $f(n) \in O(g(n))$, $g(n) \in \Omega(f(n))$
- Si $f(n) \in \Omega(g(n))$, $g(n) \in O(f(n))$
- Si $f(n) \in \Theta(g(n))$, $g(n) \in \Theta(f(n))$

Remarques supplémentaires

- Comme $\log_b n = \frac{\ln n}{\ln b}$ par définition, $\log_b n \in \Theta(\ln n)$ pour tout b : on peut donc omettre la base des logs et écrire $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$
- Si $f(n) \in O(g(n))$, $g(n) \in \Omega(f(n))$
- Si $f(n) \in \Omega(g(n))$, $g(n) \in O(f(n))$
- Si $f(n) \in \Theta(g(n))$, $g(n) \in \Theta(f(n))$
- Habituellement, on met dans les $O()$, $\Omega()$, Θ des expressions « aussi simples que possibles »

Remarques supplémentaires

- Comme $\log_b n = \frac{\ln n}{\ln b}$ par définition, $\log_b n \in \Theta(\ln n)$ pour tout b : on peut donc omettre la base des logs et écrire $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$
- Si $f(n) \in O(g(n))$, $g(n) \in \Omega(f(n))$
- Si $f(n) \in \Omega(g(n))$, $g(n) \in O(f(n))$
- Si $f(n) \in \Theta(g(n))$, $g(n) \in \Theta(f(n))$
- Habituellement, on met dans les $O()$, $\Omega()$, Θ des expressions « aussi simples que possibles »
- Écrire $f(n) = O(g(n))$ au lieu de $f(n) \in O(g(n))$ est un abus de langage (ce n'est pas une égalité)… mais tout le monde le fait

Plan

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 :

$$C_1 \leqslant \tau \leqslant C_2$$

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

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

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

Complexité asymptotique et efficacité réelle

- 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

- 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., $\geqslant \alpha n^2$) peut être plus efficace qu'un algorithme en $O(n)$ (i.e., $\leqslant \beta n$) pour une taille d'entrée n courante, parce que $\alpha \ll \beta$

Complexité asymptotique et efficacité réelle

- 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., $\geqslant \alpha n^2$) peut être plus efficace qu'un algorithme en $O(n)$ (i.e., $\leqslant \beta n$) 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, et c'est ce qui compte en pratique

Complexité asymptotique et efficacité réelle

- 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., $\geqslant \alpha n^2$) peut être plus efficace qu'un algorithme en $O(n)$ (i.e., $\leqslant \beta n$) 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, et c'est ce 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

- 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., $\geqslant \alpha n^2$) peut être plus efficace qu'un algorithme en $O(n)$ (i.e., $\leqslant \beta n$) 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, et c'est ce 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

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

Premier exemple : 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 ?

Premier exemple : 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

Premier exemple : 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)

Premier exemple : 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

Premier exemple : 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

Premier exemple : 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 $T[i]$

Premier exemple : 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

Premier exemple : 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 ← 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

Premier exemple : 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 ← 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., $O(n)$

Premier exemple : 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 ← 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

Premier exemple : 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 ← 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 $n/2$)

Premier exemple : 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 ← 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

Premier exemple : 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 ← 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

Premier exemple : 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 ← 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 $T[i]$

Premier exemple : 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 ← 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

Premier exemple : 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 ← 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

Premier exemple : 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 ← 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., $O(n)$

Deuxième exemple : recherche dichotomique

Entrée: Tableau T trié avec n éléments, un élément x de T
Sortie: la position de x dans T

1: $deb \leftarrow 0$
2: $fin \leftarrow n$
3: **while** $deb < fin$ **do**
4: $\quad mid \leftarrow \left\lfloor \frac{deb+fin}{2} \right\rfloor$
5: $\quad$ **if** $T[mid] > x$ **then**
6: $\quad\quad fin \leftarrow mid$
7: $\quad$ **else if** $T[mid] < x$ **then**
8: $\quad\quad deb \leftarrow mid + 1$
9: $\quad$ **else**
10: $\quad\quad$ **return** mid
11: $\quad$ **end if**
12: **end while**

Preuve de terminaison et de correction

- Nombre fini d'étapes : à chaque fois qu'on passe dans la boucle, la quantité $fin - deb$ décroit strictement, jusqu'à ce que $deb = fin$ donc on ne peut pas partir dans une boucle infinie

Preuve de terminaison et de correction

- Nombre fini d'étapes : à chaque fois qu'on passe dans la boucle, la quantité $fin - deb$ décroit strictement, jusqu'à ce que $deb = fin$ donc on ne peut pas partir dans une boucle infinie
- On suppose qu'il existe i tel que $T[i] = x$, on veut montrer que la recherche dichotomique renvoie ce x

Preuve de terminaison et de correction

- Nombre fini d'étapes : à chaque fois qu'on passe dans la boucle, la quantité $fin - deb$ décroit strictement, jusqu'à ce que $deb = fin$ donc on ne peut pas partir dans une boucle infinie
- On suppose qu'il existe i tel que $T[i] = x$, on veut montrer que la recherche dichotomique renvoie ce x
- Invariant de boucle : à chaque étape de la boucle, on a $deb \leqslant i < fin$ (prouvé par récurrence)

Preuve de terminaison et de correction

- Nombre fini d'étapes : à chaque fois qu'on passe dans la boucle, la quantité $fin - deb$ décroit strictement, jusqu'à ce que $deb = fin$ donc on ne peut pas partir dans une boucle infinie
- On suppose qu'il existe i tel que $T[i] = x$, on veut montrer que la recherche dichotomique renvoie ce x
- Invariant de boucle : à chaque étape de la boucle, on a $deb \leqslant i < fin$ (prouvé par récurrence)
- Par conséquent, la condition du **while** $(deb < fin)$ ne peut jamais devenir fausse et le programme finit toujours par retourner une valeur

Preuve de terminaison et de correction

- Nombre fini d'étapes : à chaque fois qu'on passe dans la boucle, la quantité $fin - deb$ décroit strictement, jusqu'à ce que $deb = fin$ donc on ne peut pas partir dans une boucle infinie
- On suppose qu'il existe i tel que $T[i] = x$, on veut montrer que la recherche dichotomique renvoie ce x
- Invariant de boucle : à chaque étape de la boucle, on a $deb \leqslant i < fin$ (prouvé par récurrence)
- Par conséquent, la condition du **while** $(deb < fin)$ ne peut jamais devenir fausse et le programme finit toujours par retourner une valeur
- Quand la valeur mid est retournée, $T[mid] = x$, donc on retourne bien la bonne valeur

Complexité en temps (pire cas) de la dichotomie

- Les affectations, les tests, les accès à une case de T sont des opérations élémentaires ($O(1)$)

Complexité en temps (pire cas) de la dichotomie

- Les affectations, les tests, les accès à une case de T sont des opérations élémentaires ($O(1)$)
- Donc la complexité dépend uniquement du nombre de fois qu'on passe dans la boucle

Complexité en temps (pire cas) de la dichotomie

- Les affectations, les tests, les accès à une case de T sont des opérations élémentaires ($O(1)$)
- Donc la complexité dépend uniquement du nombre de fois qu'on passe dans la boucle
- Si au début d'une étape de la boucle $fin - deb = \ell$, à la prochaine étape, on a $fin' - deb' = \ell'$ avec l'un des deux cas suivant :

Complexité en temps (pire cas) de la dichotomie

- Les affectations, les tests, les accès à une case de T sont des opérations élémentaires ($O(1)$)
- Donc la complexité dépend uniquement du nombre de fois qu'on passe dans la boucle
- Si au début d'une étape de la boucle $fin - deb = \ell$, à la prochaine étape, on a $fin' - deb' = \ell'$ avec l'un des deux cas suivant :
  - $\ell' = \left\lfloor \frac{deb+fin}{2} \right\rfloor - deb \leqslant \frac{deb+fin}{2} - deb = \frac{\ell}{2}$

Complexité en temps (pire cas) de la dichotomie

- Les affectations, les tests, les accès à une case de T sont des opérations élémentaires ($O(1)$)
- Donc la complexité dépend uniquement du nombre de fois qu'on passe dans la boucle
- Si au début d'une étape de la boucle $fin - deb = \ell$, à la prochaine étape, on a $fin' - deb' = \ell'$ avec l'un des deux cas suivant :
  - $\ell' = \left\lfloor \frac{deb+fin}{2} \right\rfloor - deb \leqslant \frac{deb+fin}{2} - deb = \frac{\ell}{2}$
  - $\ell' = fin - \left\lfloor \frac{deb+fin}{2} \right\rfloor - 1 \leqslant fin - \frac{deb+fin}{2} = \frac{\ell}{2}$

Complexité en temps (pire cas) de la dichotomie

- Les affectations, les tests, les accès à une case de T sont des opérations élémentaires ($O(1)$)
- Donc la complexité dépend uniquement du nombre de fois qu'on passe dans la boucle
- Si au début d'une étape de la boucle $fin - deb = \ell$, à la prochaine étape, on a $fin' - deb' = \ell'$ avec l'un des deux cas suivant :
  - $\ell' = \left\lfloor \frac{deb+fin}{2} \right\rfloor - deb \leqslant \frac{deb+fin}{2} - deb = \frac{\ell}{2}$
  - $\ell' = fin - \left\lfloor \frac{deb+fin}{2} \right\rfloor - 1 \leqslant fin - \frac{deb+fin}{2} = \frac{\ell}{2}$
- Donc on décroit ℓ à chaque étape d'un facteur $\geqslant 2$

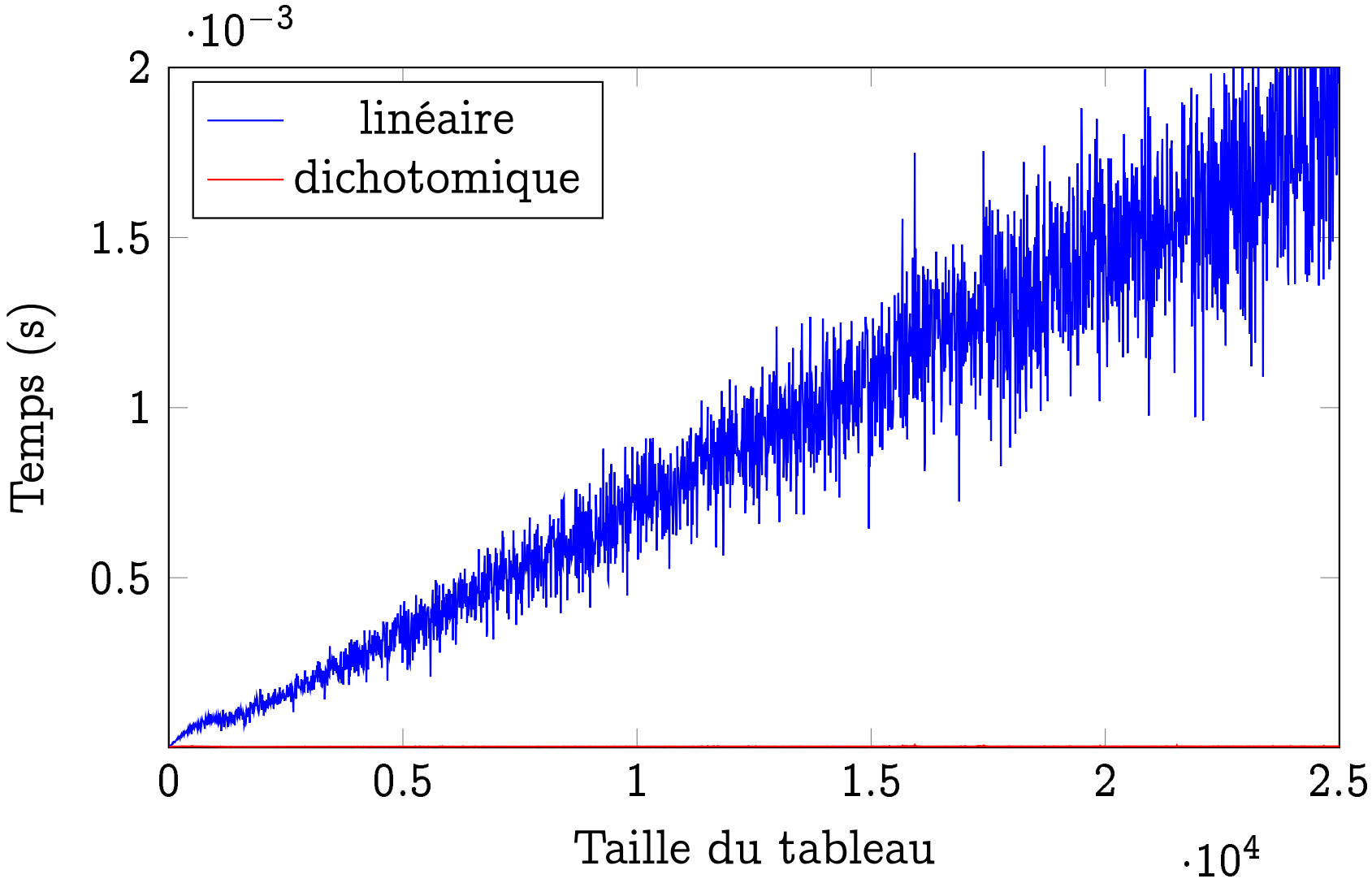
Complexité en temps (pire cas) de la dichotomie

- Les affectations, les tests, les accès à une case de T sont des opérations élémentaires ($O(1)$)
- Donc la complexité dépend uniquement du nombre de fois qu'on passe dans la boucle
- Si au début d'une étape de la boucle $fin - deb = \ell$, à la prochaine étape, on a $fin' - deb' = \ell'$ avec l'un des deux cas suivant :
  - $\ell' = \left\lfloor \frac{deb+fin}{2} \right\rfloor - deb \leqslant \frac{deb+fin}{2} - deb = \frac{\ell}{2}$
  - $\ell' = fin - \left\lfloor \frac{deb+fin}{2} \right\rfloor - 1 \leqslant fin - \frac{deb+fin}{2} = \frac{\ell}{2}$
- Donc on décroit ℓ à chaque étape d'un facteur $\geqslant 2$
- Donc il y a $\leqslant \log_2(n) + 1 = O(\log n)$ étapes

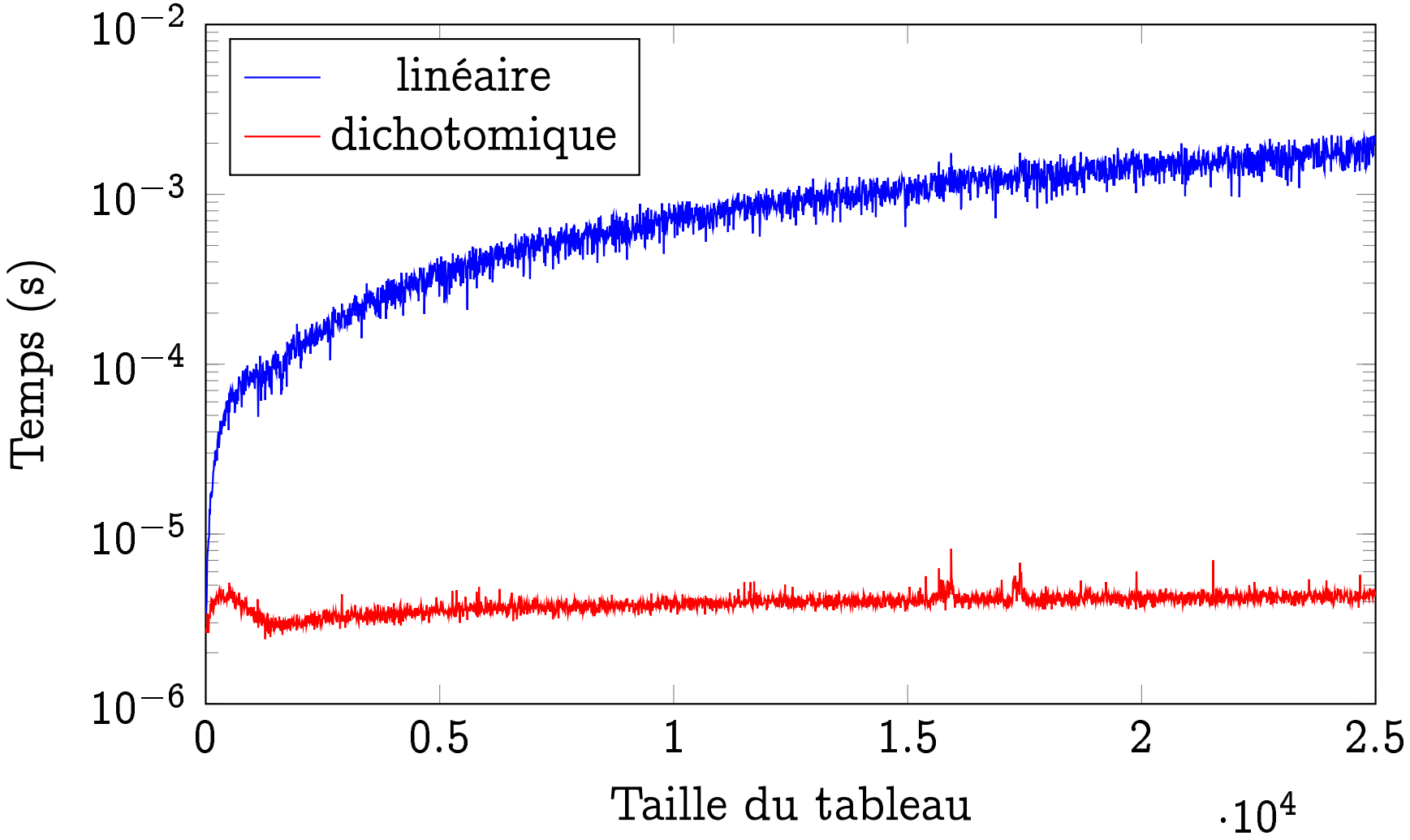
Complexité en temps (pire cas) de la dichotomie

- Les affectations, les tests, les accès à une case de T sont des opérations élémentaires ($O(1)$)
- Donc la complexité dépend uniquement du nombre de fois qu'on passe dans la boucle
- Si au début d'une étape de la boucle $fin - deb = \ell$, à la prochaine étape, on a $fin' - deb' = \ell'$ avec l'un des deux cas suivant :
  - $\ell' = \left\lfloor \frac{deb+fin}{2} \right\rfloor - deb \leqslant \frac{deb+fin}{2} - deb = \frac{\ell}{2}$
  - $\ell' = fin - \left\lfloor \frac{deb+fin}{2} \right\rfloor - 1 \leqslant fin - \frac{deb+fin}{2} = \frac{\ell}{2}$
- Donc on décroit ℓ à chaque étape d'un facteur $\geqslant 2$
- Donc il y a $\leqslant \log_2(n) + 1 = O(\log n)$ étapes
- On conclut avec complexité de $O(\log n)$

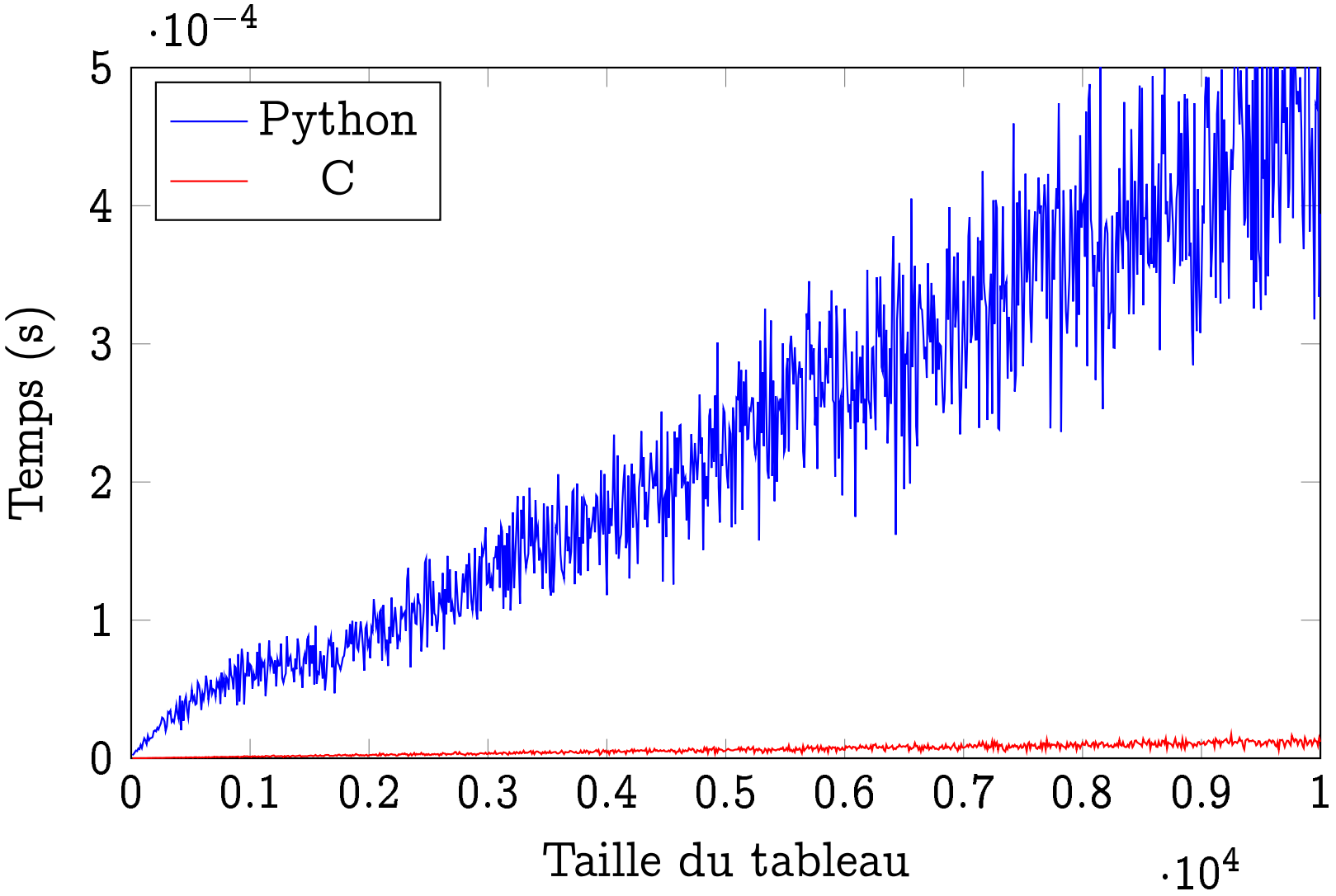
Recherche linéaire vs recherche dichotomique



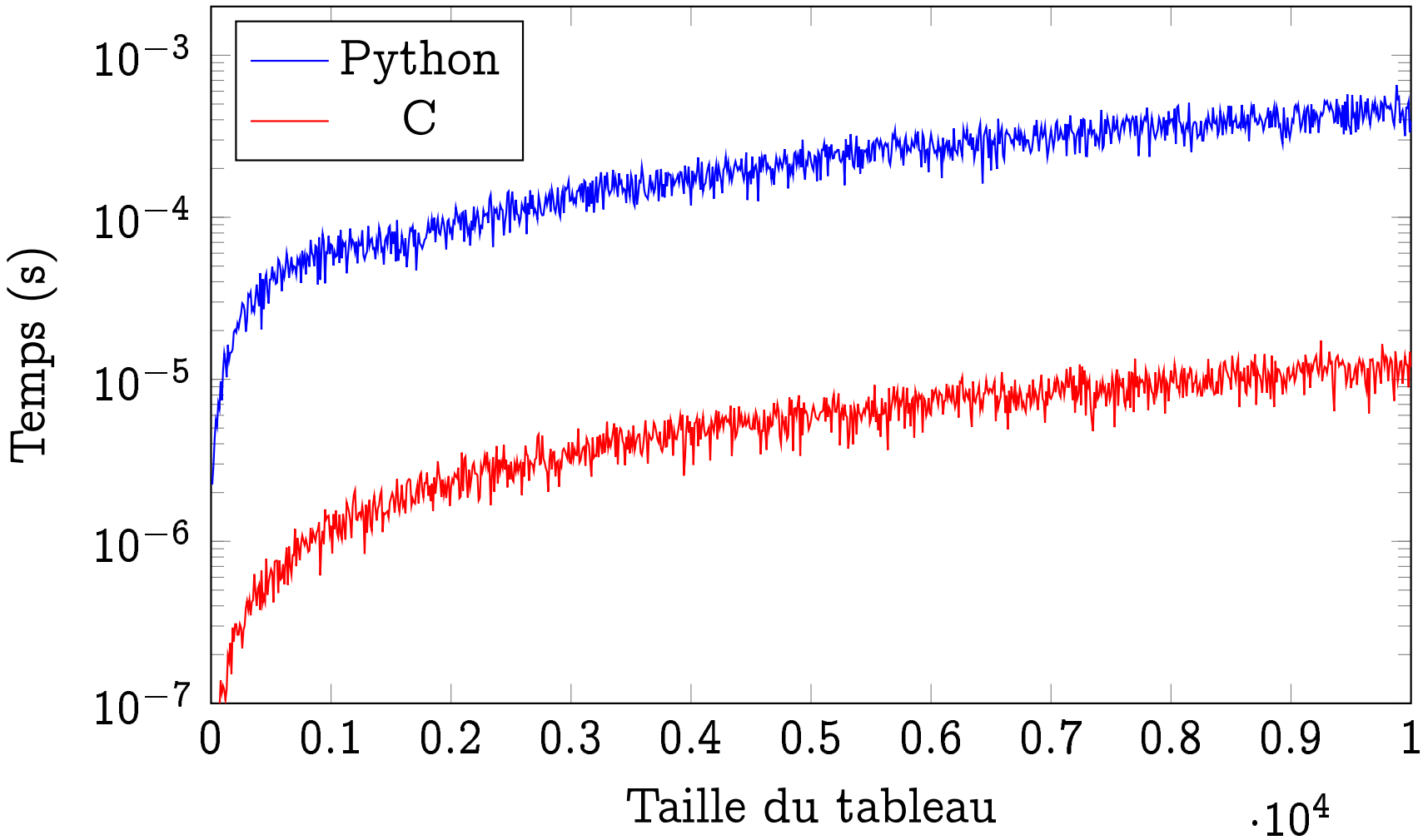
Recherche linéaire vs recherche dichotomique



Recherche linéaire : Python vs C



Recherche linéaire : Python vs C



Plan

Complexité des collections

- On s'intéresse aux opérations possibles sur les collections (listes, ensembles, etc.) Python et leur complexité
- Complexité non spécifiée par le langage et peut dépendre de l'interpréteur Python ! Mais en pratique, pas de différence
- Complexités parfois amorties (voir cours ultérieur)
- On verra comment certaines de ces structures de données peuvent être définies dans un cours ultérieur

Tuples

Type	`tuple`	
Définition	Séquence ordonnée de taille n fixe, éléments non modifiables	
Littéraux	`()` `(1,)` `(1,2,3)` `('a',1,3.14)`	
Accès	`t[i]`	$O(1)$
Concaténation	`t1 + t2`	$O(n_1 + n_2)$
Longueur	`len(t)`	$O(1)$
Boucle	`for x in t:`	$O(n)$

« Listes » Python

Type	`list`	
Définition	Séquence ordonnée de taille n variable, éléments modifiables	
Littéraux	`[]` `[1]` `[1,2,3]` `['a',1,3.14]`	
Accès	`l[i]`	$O(1)$
Modification	`l[i]=42`	$O(1)$
Ajout à la fin	`l.append(42)`	$O(1)$
Ajout ailleurs	`l.insert(16,"toto")`	$O(n)$
Suppression à la fin	`l.pop()`	$O(1)$
Suppression ailleurs	`del l[16]`	$O(n)$
Concaténation	`l1 + l2`	$O(n_1 + n_2)$
Longueur	`len(l)`	$O(1)$
Boucle	`for x in l:`	$O(n)$

Tranches de listes

Python permet aussi d'accéder à une « tranche » d'une liste, une sous-liste de la liste : `l[i:j]` désigne tous les éléments de la liste entre les position i (inclus) et j (exclus). Si i est omis, il vaut 0 ; si j est omis, il vaut `len(l)`

Accès	`l[i:j]`	$O(j-i)$
Modification	`l1[i:j]=l2`	$O(n_1+n_2)$
Suppression	**del** `l[i:j]`	$O(n)$

Chaînes et tableaux d'octets

- Les chaînes de caractère (`str`) et tableaux d'octets (`bytes`) se comportent de manière très similaire aux listes
- Mais ce sont des séquences non modifiables
- Opérations d'accès, de tranchage : mêmes opérations, mêmes complexités que les listes
- Il existe un type `bytearray` pour des séquences modifiables d'octets

Compréhension de liste

- Compréhension : manière de définir une collection en ne donnant pas la liste de ses éléments (extension) mais une caractérisation de ceux-ci
- En Python, expression définissant des listes complexes
- Syntaxe :
  `[expression for x in collection if condition]`
- Plus concis que de faire une boucle !
- Même complexité que la boucle correspondante

```
mariages = {"Titi": "Tata", "Toto": None, "Tata": "Titi"}
celibataires = [p for p in mariage if mariage[p] == None]

# Tableau de multiplication en sautant une ligne sur 2
[(x,y,x*y) for x in range(10) for y in range(10) if x % 2 == 0]
```

Ensembles

Type	`set`	
Définition	Séquence non ordonnée de taille n variable	
Littéraux	`set()` `{1}` `{1,2,3}` `{'a',1,3.14}`	
Accès	`x in s`	$O(1)$
Ajout	`s.add(42)`	$O(1)$
Suppression	`s.remove(42)`	$O(1)$
Union	`s3=s1.union(s2)`	$O(n_1 + n_2)$
Intersection	`s3=s1.intersection(s2)`	$O(n_1 + n_2)$
Longueur	`len(s)`	$O(1)$
Boucle	`for x in s:`	$O(n)$

Dictionnaires

Type	`dict`	
Définition	Tableau associatif de taille n variable, reliant des clefs à des valeurs	
Littéraux	`{}`, `{'a':1, 'b': 2}`	
Accès	`d['a']`	$O(1)$
Ajout	`d['c']=42`	$O(1)$
Suppression	`del d['b']`	$O(1)$
Taille	`len(d)`	$O(1)$
Boucle	`for clef in d:`	$O(n)$

Plan

Références

- 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`.