Introduction à lalgorithmique et aux structures de données

Cours L3 Algorithmique et programmation

Pierre Senellart

28 septembre 2017

Plan

Algorithmique & programmation

Complexité algorithmique

Structures de données élémentaires

Complexité amortie

Références

Algorithmique & programmation

- Vient du nom de محمد بن موسى خوارزمی

Algorithmique & programmation

- Vient du nom de محمد بن موسى خوارزمی (Muhammad ibn Musa al-Khwarizmi), un scientifique persan du IXe siècle

Algorithmique & programmation

- Vient du nom de محمد بن موسى خوارزمی (Muhammad ibn Musa al-Khwarizmi), un scientifique persan du IXe siècle
- ... qui a aussi donné le mot algèbre (الْجَبْر, la *restoration* en arabe), issu du titre de l'un de ses textes sur la résolution d'équations

Algorithmique & programmation

- Vient du nom de محمد بن موسى خوارزمی (Muhammad ibn Musa al-Khwarizmi), un scientifique persan du IXᵉ siècle
- ... qui a aussi donné le mot algèbre (الْجَبْر, la *restoration* en arabe), issu du titre de l'un de ses textes sur la résolution d'équations
- Un algorithme, c'est une spécification formelle de la manière dont résoudre un problème donné : à partir d'une entrée, comment produire la sortie correspondant à une solution du problème via des opérations élémentaires

Algorithmique & programmation

- Vient du nom de محمد بن موسى خوارزمی (Muhammad ibn Musa al-Khwarizmi), un scientifique persan du IXᵉ siècle
- … qui a aussi donné le mot algèbre (الْجَبْر, la *restoration* en arabe), issu du titre de l'un de ses textes sur la résolution d'équations
- Un algorithme, c'est une spécification formelle de la manière dont résoudre un problème donné : à partir d'une entrée, comment produire la sortie correspondant à une solution du problème via des opérations élémentaires
- L'algorithmique, c'est l'étude des algorithmes : conception d'algorithmes, analyse de leur performance, etc.

Algorithmique & programmation

- Vient du nom de محمد بن موسى خوارزمی (Muhammad ibn Musa al-Khwarizmi), un scientifique persan du IXe siècle
- ... qui a aussi donné le mot algèbre (الْجَبْر, la *restoration* en arabe), issu du titre de l'un de ses textes sur la résolution d'équations
- Un algorithme, c'est une spécification formelle de la manière dont résoudre un problème donné : à partir d'une entrée, comment produire la sortie correspondant à une solution du problème via des opérations élémentaires
- L'algorithmique, c'est l'étude des algorithmes : conception d'algorithmes, analyse de leur performance, etc.
- La programmation, c'est la manière dont transformer un algorithme en du code d'un langage informatique, afin de pouvoir exécuter l'algorithme sur des données concrètes

Algorithmique vs programmation

- Tout algorithme est implémentable : il doit être décrit en des termes suffisamment précis pour que la programmation de l'algorithme soit non ambiguë
- … mais ça ne veut pas dire que le programme implémentant cet algorithme soit simple à écrire, car le programmeur doit tenir compte des limites de la machine, du langage de programmation utilisé, des objets de bas niveau vs 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 usuels ont le même pouvoir d'expression (Turing-complets)
- … mais un langage de programmation peut avoir un impact sur la facilité (cf. `http://pierre.senellart.com/travaux/languages/languages.xml`) ou l'efficacité de l'implémentation (voir plus loin)

Structure de données

- Élément de base d'algorithmes plus complexes, réutilisées dans divers algorithmes pour résoudre divers problème
- Spécification formelle d'un objet mathématique abstrait (liste, ensemble, fonction, graphe, matrice, etc.), des opérations possible sur cet objet (insertion, parcours, inversion, etc.) et d'algorithmes les réalisant
- Brique de base 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, ayant des performances différentes

Plan

Algorithmique & programmation

Complexité algorithmique

Structures de données élémentaires

Complexité amortie

Références

Comment mesurer la performance d'un algorithme ?

- Tentative de caractériser, sur la description d'un algorithme, la performance d'un programme l'implémentant ; ou sur la description d'un problème, la performance d'un programme implémentant un algorithme résolvant ce problème
- Différentes notions de performances, différentes notions de complexité :

  Complexité en temps: temps de calcul d'un programme séquentiel

  Complexité en espace: espace mémoire utilisé par un programme

  Complexité de communication: volume de données échangées par un système distribué

  Complexité descriptive: longueur du programme le plus court

  Complexité de circuit: taille du circuit électronique implémentant l'algorithme
- Ici : les deux premiers uniquement (et surtout le premier !)

Calculer la complexité en temps

- On fait l'hypothèse que chacune des opérations de base apparaissant dans la description d'un algorithme :
  - arithmétique
  - affectation de variable
  - comparaisons
  - tests
  - etc.

  prend un temps élémentaire, que l'on borne par une constante C
- On somme le nombre d'opérations élémentaires réalisées, en fonction de la taille n de l'entrée, p. ex., $42 \times n$
- On déduit une borne, ici $42 \times n \times C$, sur le temps total de l'algorithme

Calculer la complexité en temps

- On fait l'hypothèse que chacune des opérations de base apparaissant dans la description d'un algorithme :
  - arithmétique
  - affectation de variable
  - comparaisons
  - tests
  - etc.

  prend un temps élémentaire, que l'on borne par une constante C
- On somme le nombre d'opérations élémentaires réalisées, en fonction de la taille n de l'entrée, p. ex., $42 \times n$
- On déduit une borne, ici $42 \times n \times C$, sur le temps total de l'algorithme
- Temps d'une opération élémentaire formalisable (avec les machines de Turing, ou avec les machines de Von Neumann), mais on ne le fera pas

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

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

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

Complexité en temps asymptotique

- On utilise les notations $O()$, $\Omega()$, $\Theta()$ et les bornes établies pour indiquer la complexité d'un algorithme et négliger C et les autres constantes
- Par exemple, si on a borné le temps élémentaire τ par :

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

- Et que sur toutes les entrées, l'algorithme A réalise $42 \times n$ opérations, alors :

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

d'où $T(\mathcal{A}, n) = \Theta(n)$

Complexité dans le pire des cas, cas moyen

- En général, on cherche à borner le temps d'un algorithme supérieurement, et on s'intéresse à la complexité dans le pire des cas : une borne supérieure qui est vraie sur n'importe quelle entrée
- Parfois trop restrictif, et on s'intéresse au cas moyen : en moyenne, sur toutes les entrées d'une taille donnée, quelle est une borne sur la complexité
- Fait l'hypothèse que toutes les entrées ont la même probabilité, ce qui est discutable

Exemple élémentaire : recherche dans un tableau

Entrée : Tableau T de n éléments distincts, un élément x présent dans 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 ?

Exemple élémentaire : recherche dans un tableau

Entrée : Tableau T de n éléments distincts, un élément x présent dans 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 des cas

(x est en dernière position)

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

$4n + 1$, soit $O(n)$

Exemple élémentaire : recherche dans un tableau

Entrée : Tableau T de n éléments distincts, un élément x présent dans 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 des cas

(x est en dernière position)

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

$4n + 1$, soit $O(n)$

Cas moyen

(x est en moyenne au milieu)

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

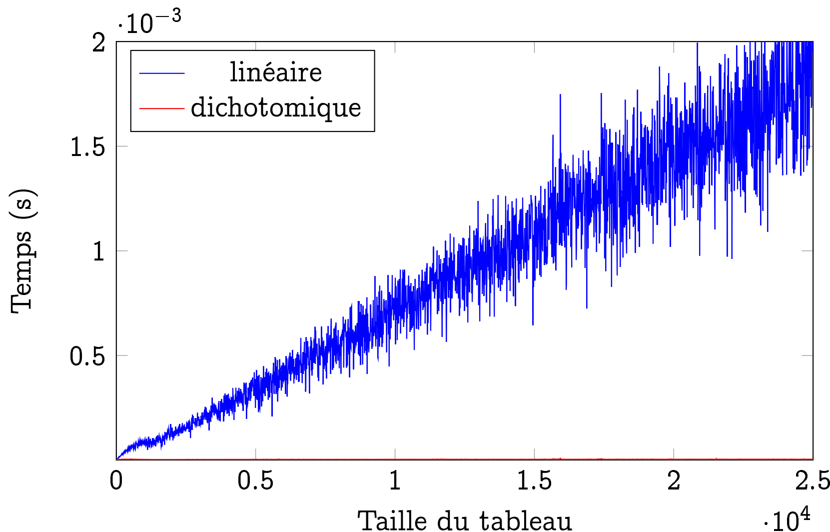
$2n + 1$, soit $O(n)$

Complexité asymptotique vs performances réelles

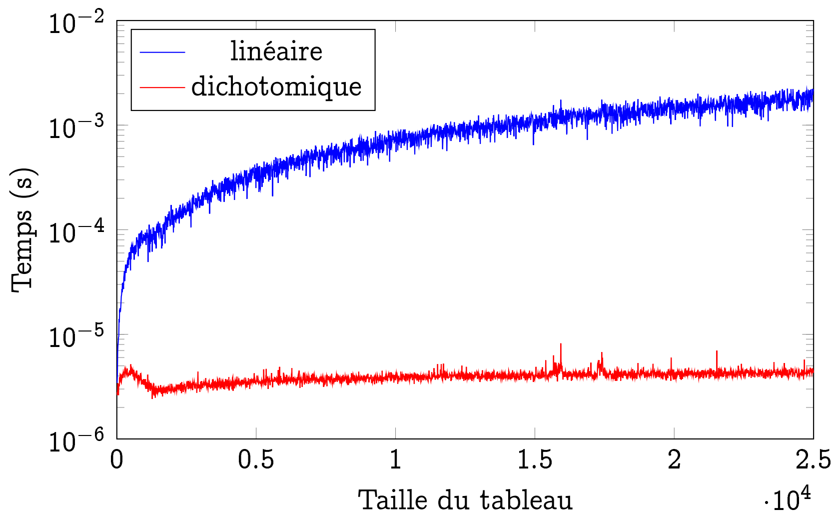
En pratique :

- La complexité asymptotique a un réel impact. Un algorithme en $O(n)$ est toujours plus lent qu'un algorithme en $O(\log n)$ *si* n *est suffisamment grand*
- Parfois, un algorithme en $O(n^2)$ (i.e., $\leqslant \alpha n^2$) peut être plus efficace qu'un algorithme en $O(n)$ (i.e., $\leqslant \beta n$) pour des tailles d'entrée n courantes, parce que $\alpha \ll \beta$
- Parfois, la complexité dans le pire des cas est mauvaise, mais la complexité en moyenne est bonne, et peut être celle qui compte en pratique
- Le langage de programmation utilisé a un réel impact sur les performances (mais en général, c'est 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

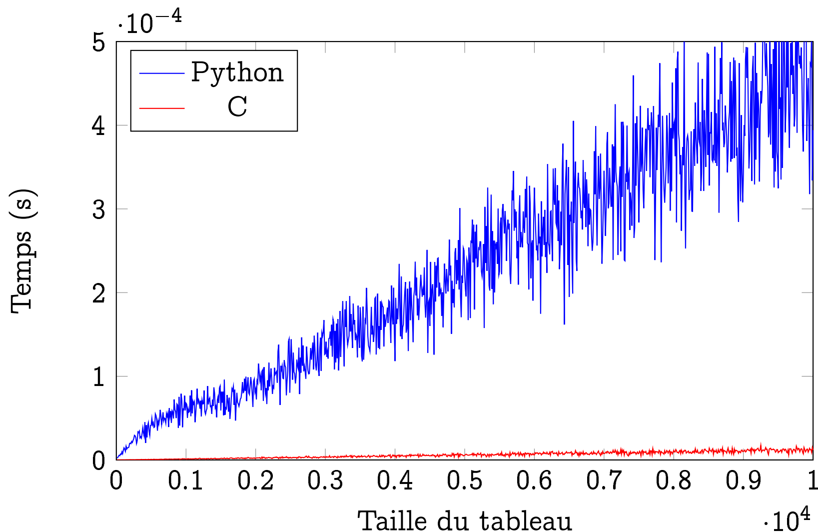
Recherche linéaire vs dichotomique dans tableau trié



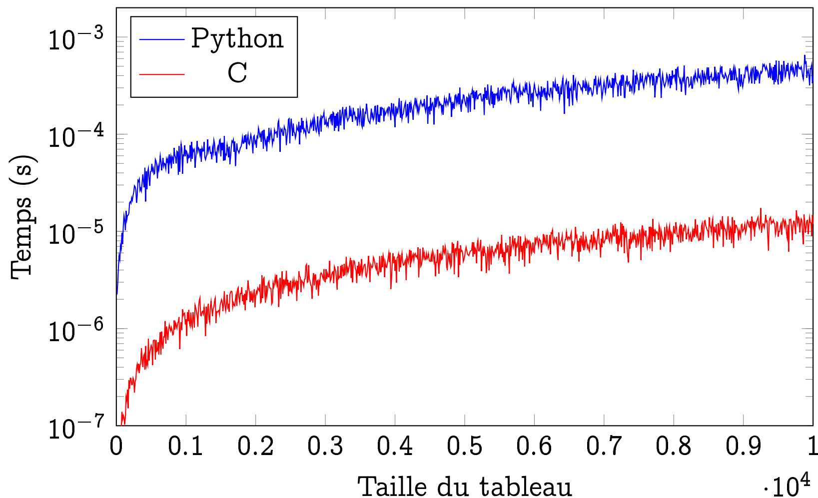
Recherche linéaire vs dichotomique dans tableau trié



Recherche linéaire : Python vs C



Recherche linéaire : Python vs C



Complexité en espace

- Pareil que complexité en temps, sauf qu'on compte les utilisations élémentaires de l'espace mémoire plutôt que les temps de calculs élémentaires
- On fait souvent des hypothèses simplificatrices, comme le fait que n'importe quel entier tient en espace mémoire constant
- On ne tient pas compte de l'espace nécessaire à représenter l'entrée
- On utilise aussi $O()$, $\Omega()$, $\Theta()$ pour résumer la complexité asymptotique
- Par exemple, recherche dans un tableau, complexité en espace en $O(1)$: on a juste besoin de stocker la variable i en mémoire (en plus des entrées T et x), ce qui occupe un espace élémentaire indépendant de la taille de l'entrée

Plan

Algorithmique & programmation

Complexité algorithmique

Structures de données élémentaires

Complexité amortie

Références

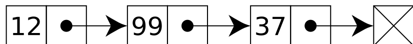
Conteneurs

- Structures de données stockant une liste $L = (l_1, \ldots, l_n)$ d'éléments (p. ex., entiers, nombres en virgule flottantes, objets complexes, etc.)
- Diverses spécifications d'une telle structure, permettant des opérations différentes, avec des efficacités différentes :
  - Accès aléatoire (*random access*) : étant donné i, accéder à l_i
  - Accès au début (l_1), à la fin (l_n)
  - Ajout au début (avant l_1), à un endroit arbitraire (entre l_i et l_{i+1}), à la fin (après l_n)
  - Suppression du premier élément (l_1), d'un élément arbitraire (l_i), du dernier élément (l_n)
  - Variante : on suppose un ordre sur les éléments, et on suppose que $l_1 \leqslant \ldots \leqslant l_n$. Accès à un élément ordonné, ajout en respectant l'ordre.

Tableau fixe (*fixed array*)

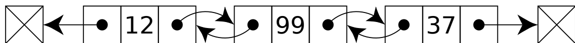
- Zone mémoire contiguë, de taille fixe, pré-allouée
- Correspond aux tableaux classiques des langages de programmation
- Accès aléatoire en $O(1)$
- Ajout, suppression impossibles
- Le plus compact possible, pas d'espace perdu

Liste chaînée (*linked list*)



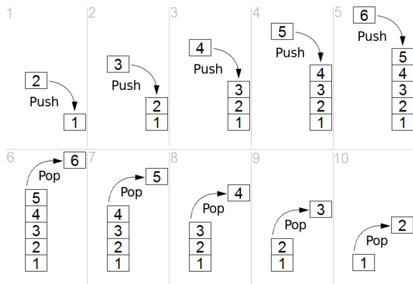
- Chaque élément est stocké dans un maillon (link), qui comporte également un pointeur vers l'élément suivant
- On conserve également un pointeur vers le premier maillon
- Accès au premier élément en $O(1)$
- Accès aléatoire ou au dernier élément en $O(n)$
- Ajout à un endroit arbitraire (une fois qu'on a accédé à l'élément précédent) en $O(1)$
- Suppression du premier élément en $O(1)$
- Suppression en $O(1)$ si on connaît l'élément précédent, en $O(n)$ sinon

Liste doublement chaînée (*doubly linked list*)



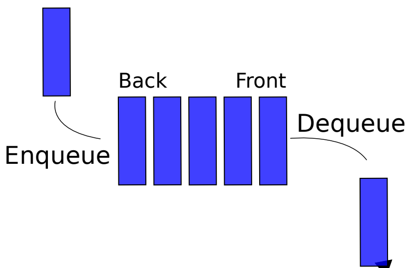
- Chaque élément est stocké dans un maillon (link), qui comporte également un pointeur vers l'élément suivant et l'élément précédent
- On conserve également un pointeur vers les premier et dernier maillons
- Accès au premier/dernier élément en $O(1)$
- Accès aléatoire en $O(n)$
- Ajout à un endroit arbitraire (une fois qu'on a accédé à l'élément) en $O(1)$
- Suppression à un endroit arbitraire (une fois qu'on a accédé à l'élément) en $O(1)$

Pile (*stack*, *LIFO* pour *last-in-first-out*)



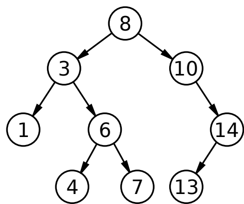
- Structure de données abstraite, peut être implémentée de différentes façons, par exemple, avec une liste simplement chaînée
- Seules opérations possibles :
  - Accès au premier élément en $O(1)$
  - Ajout d'un élément au début en $O(1)$
  - Suppression du premier élément en $O(1)$

File (*queue*, *FIFO* pour *first-in-first-out*)



- Structure de données abstraite, peut être implémentée de différentes façons, par exemple, avec une liste doublement chaînée
- Seules opérations possibles :
  - Accès au dernier élément en $O(1)$
  - Ajout d'un élément au début en $O(1)$
  - Suppression du dernier élément en $O(1)$

Arbre binaire de recherche non équilibré (*binary search tree*)



- Stocke une liste ordonnée d'éléments
- Arbre binaire : chaque nœud pointe vers au plus deux fils, on conserve un pointeur vers la racine
- Pour tout nœud stockant l'élément l, le sous-arbre enraciné au fils gauche (s'il existe) stocke des éléments $\leqslant l$, et le sous-arbre enraciné au fils droit des éléments $\geqslant l$
- Accès, ajout, suppression d'un élément ordonné : $O(p)$ où p est la profondeur de l'arbre
- Pas de borne a priori sur cette profondeur !

Plan

Algorithmique & programmation

Complexité algorithmique

Structures de données élémentaires

Complexité amortie

Références

Complexité amortie

- Jusqu'ici, on mesure séparément la complexité de chaque opération sur une structure de données
- Parfois, on ne peut pas borner le temps de chaque opération individuelle, mais, on peut borner le temps moyen dans une séquence d'opération
- On considère une séquence de n opérations $o_1, \ldots, o_n$ sur une structure de données, de coût $c_1, \ldots, c_n$
- On cherche à calculer la complexité moyenne d'une opération o_i, i.e.,

$$\frac{1}{n}\sum_{i=1}^{n} c_i$$

- On appelle cela la complexité amortie
- Ne fait de sens que si on définit précisément le type de séquence considéré

Méthode du potentiel

- On associe un potentiel $\Phi(X)$ (nombre réel) à une structure de données X
- On considère une séquence de n opérations $o_1, \ldots, o_n$, de coût réel $c_1, \ldots, c_n$, et les structure de données $X_0, \ldots, X_n$ correspondantes
- On définit $\hat{c}_i = c_i + \Phi(X_i) - \Phi(X_{i-1})$
- On a alors : $\frac{1}{n}\sum_{i=1}^{n} \hat{c}_i = \frac{1}{n}\sum_{i=1}^{n} c_i + \frac{1}{n}(\Phi(X_n) - \Phi(X_0))$
- Si on obtient $\hat{c}_i = O(f(|X|))$, alors on a aussi $\frac{1}{n}\sum_{i=1}^{n} c_i = O(f(|X|))$ si pour tout n, $\Phi(X_n) \geqslant \Phi(X_0)$
- On considère souvent que X_0 est la structure de données vide et on pose $\Phi(X_0) = 0$, ce qui donne la condition $\Phi(X_n) \geqslant 0$

Application : tableau dynamique (*array, vector, table*)

- Pointeur vers un tableau classique de capacité c + entier n stockant la taille réellement utilisée
- Accès aléatoire en $O(1)$
- Suppression du dernier élément en $O(1)$ en décrémentant n
- Insertion à la fin du tableau :

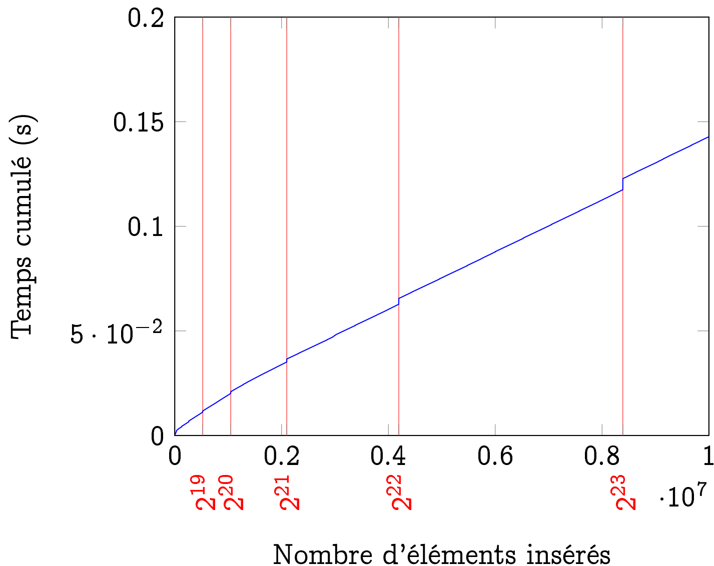
Entrée : tableau T avec capacité c, taille n, élément x
Sortie : tableau T de taille $n+1$ avec $T[n] = x$

```
if n = c then
    allouer nouveau tableau T' de taille 2 × c
    for i = 0 to n − 1 do
        T'[i] ← T[i]
    end for
    faire pointer T vers T'
    c ← 2 × c
end if
T[n] ← x
n ← n + 1
```

Analyse de la complexité amortie

Au tableau ! Séquence de n insertions

Insertion dans un vector en C++



Plan

Algorithmique & programmation

Complexité algorithmique

Structures de données élémentaires

Complexité amortie

Références

Références

- Généralités sur l'algorithmique : chap. 1 de [1, 2]
- Bases de l'analyse de la complexité d'un algorithme : chap. 2 et 3 de [1, 2]
- Structures de données élémentaires : chap. 10 de [1, 2]
- Complexité amortie : chap. 17 de [1, 2]

Bibliographie

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
Introduction to Algorithms.
MIT Press, 3rd edition, 2009.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
Algorithmique.
Dunod, 3rd edition, 2010.

Ressources utilisées

L’image de file est due à Vegpuff (Wikimedia), CC-BY-SA-3.0.