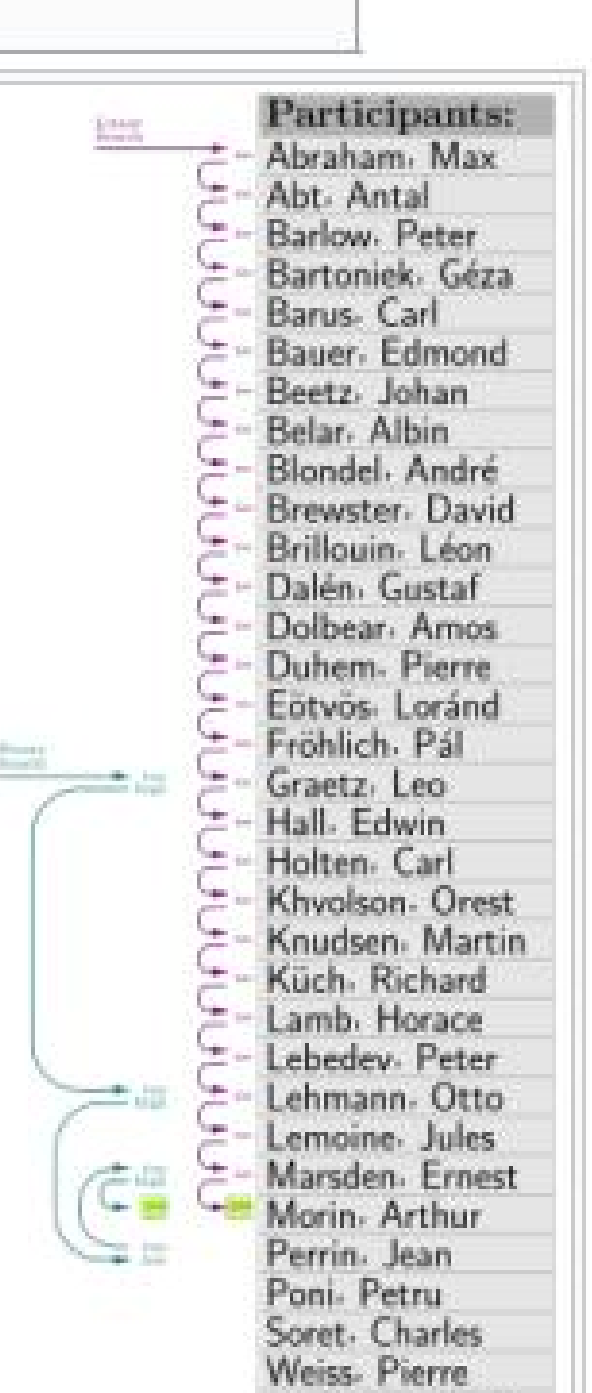
Analyse d'algorithmes

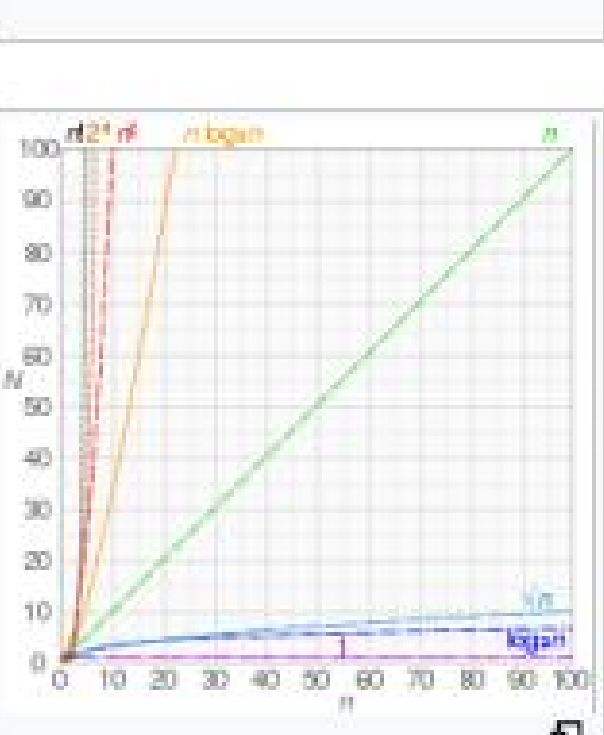
Un article de Wikipédia, l'encyclopédie libre

Cet article comprend une liste de références générales, mais elle reste largement non vérifiée car **elle manque de citations en ligne correspondantes suffisantes.** Aidez-nous à améliorer cet article en introduisant des citations plus précises. (Mars 2010) (Apprenez comment et quand supprimer ce modèle de message)

En informatique, l' **analyse des algorithmes** est le processus de recherche de la complexité de calcul des algorithmes - la quantité de temps, de stockage ou d'autres ressources nécessaires pour les exécuter. Habituellement, cela implique de déterminer une fonction qui relie la longueur de l'entrée d'un algorithme au nombre d'étapes qu'il prend (sa complexité temporelle) ou au nombre d'emplacements de stockage qu'il utilise (sa complexité spatiale). Un algorithme est dit efficace lorsque les valeurs de cette fonction sont petites ou croissent lentement par rapport à une croissance de la taille de l'entrée. Différentes entrées de la même longueur peuvent entraîner un comportement différent de l'algorithme, de sorte que les descriptions des meilleurs, des pires et des cas moyens peuvent toutes présenter un intérêt pratique. Sauf indication contraire, la fonction décrivant les performances d'un algorithme est généralement une limite supérieure, déterminée à partir des pires entrées de l'algorithme.



Pour rechercher une entrée donnée dans une liste ordonnée donnée, les algorithmes de recherche binaire et linéaire (qui ignorent l'ordre) peuvent être utilisés. L'analyse du premier et du second algorithme montre qu'il faut au plus log2(n) et n pas de vérification, respectivement, pour une liste de longueur n. Dans l'exemple de liste illustré de longueur 33, la recherche de "Morin, Arthur" prend 5 et 28 étapes avec une recherche binaire (indiquée en cyan) et linéaire (magenta) , respectivement.



Graphiques des fonctions couramment utilisées dans l'analyse des algorithmes, montrant le nombre d'opérations N par rapport à la taille d'entrée n pour chaque fonction

Le terme "analyse d'algorithmes" a été inventé par Donald Knuth. [1] L'analyse d'algorithmes est une partie importante d'une théorie plus large de la complexité informatique, qui fournit des estimations théoriques des ressources nécessaires à tout algorithme qui résout un problème informatique donné. Ces estimations donnent un aperçu des directions raisonnables de recherche d' algorithmes efficaces.

Dans l'analyse théorique des algorithmes, il est courant d'estimer leur complexité au sens asymptotique, c'est-à-dire d'estimer la fonction de complexité pour une entrée arbitrairement grande. La notation Big O, la notation Big oméga et la notation Big-thêta sont utilisées à cette fin. Par exemple, on dit que la recherche binaire s'exécute en un nombre d'étapes proportionnel au logarithme de la longueur de la liste triée recherchée, ou en O (log (n)), familièrement "en temps logarithmique". Habituellement, des estimations asymptotiques sont utilisées car différentes implémentations du même algorithme peuvent différer en efficacité. Cependant, les efficacités de deux implémentations "raisonnables" d'un algorithme donné sont liées par un facteur multiplicatif constant appelé constante cachée.

Des mesures exactes (non asymptotiques) de l'efficacité peuvent parfois être calculées, mais elles nécessitent généralement certaines hypothèses concernant l'implémentation particulière de l'algorithme, appelée modèle de calcul. Un modèle de calcul peut être défini en termes d' ordinateur abstrait, par exemple une machine de Turing, et/ou en postulant que certaines opérations sont exécutées en unité de temps. Par exemple, si la liste triée à laquelle nous appliquons la recherche binaire a n éléments, et que nous pouvons garantir que chaque recherche d'un élément dans la liste peut être effectuée en unité de temps, alors au plus log2 n + 1 unités de temps sont nécessaires pour retourner une réponse.

Sommaire [masquer]

Modèles de coût [modifier]

Les estimations d'efficacité temporelle dépendent de ce que nous définissons comme une étape. Pour que l'analyse corresponde utilement au temps d'exécution réel, il faut garantir que le temps nécessaire pour effectuer une étape soit majoré par une constante. Il faut être prudent ici; par exemple, certaines analyses comptent une addition de deux nombres comme une étape. Cette hypothèse peut ne pas être justifiée dans certains contextes. Par exemple, si les nombres impliqués dans un calcul peuvent être arbitrairement grands, le temps requis par une seule addition ne peut plus être supposé constant.

Deux modèles de coût sont généralement utilisésÿ:[2][3][4][5][6]

- le **modèle de coût uniforme,** également appelé **mesure de coût uniforme** (et variations similaires), attribue un coût constant à chaque opération de la machine, quelle que soit la taille des nombres impliqués le **modèle de coût logarithmique,** également appelé **mesure de coût logarithmique** (et variations similaires) ,
- attribue un coût à chaque opération machine proportionnel au nombre de bits impliqués

Ce dernier est plus lourd à utiliser, il n'est donc utilisé que lorsque cela est nécessaire, par exemple dans l'analyse d' algorithmes arithmétiques à précision arbitraire , comme ceux utilisés en cryptographie.

Un point clé qui est souvent négligé est que les limites inférieures publiées pour les problèmes sont souvent données pour un modèle de calcul qui est plus restreint que l'ensemble des opérations que vous pourriez utiliser dans la pratique et donc il existe des algorithmes plus rapides que ce qui serait naïf cru possible.[7]

Analyse d' exécution [modifier]

L'analyse du temps d'exécution est une classification théorique qui estime et anticipe l'augmentation du temps d'exécution (ou du temps d'exécution) d'un algorithme à mesure que sa taille d'entrée (généralement notée n) augmente. L' efficacité d'exécution est un sujet de grand intérêt en informatique : un programme peut prendre des secondes, des heures, voire des années pour terminer son exécution, selon l'algorithme qu'il implémente. Alors que les techniques de profilage logiciel peuvent être utilisées pour mesurer le temps d'exécution d'un algorithme dans la pratique, elles ne peuvent pas fournir de données de synchronisation pour toutes les infinités d'entrées possibles; ce dernier ne peut être atteint que par les méthodes théoriques de l'analyse du temps d'exécution.

Lacunes des métriques empiriques [modifier]

Étant donné que les algorithmes sont indépendants de la plate-forme (c'est-à-dire qu'un algorithme donné peut être implémenté dans un langage de programmation arbitraire sur un ordinateur arbitraire exécutant un système d'exploitation arbitraire), il existe d'autres inconvénients importants à l'utilisation d'une approche empirique pour évaluer les performances comparatives d'un ensemble donné d' algorithmes. algorithmes.

Prenons comme exemple un programme qui recherche une entrée spécifique dans une liste triée de taille n. Supposons que ce programme soit implémenté sur l'ordinateur A, une machine à la pointe de la technologie, utilisant un algorithme de recherche linéaire , et sur l'ordinateur B, une machine beaucoup plus lente, utilisant un algorithme de recherche binaire. Les tests de référence sur les deux ordinateurs exécutant leurs programmes respectifs peuvent ressembler à ceciÿ:

n (taille de la liste)	Temps d'exécution de l'ordinateur A (en nanosecondes)	Temps d'exécution de l'ordinateur B (en nanosecondes)
16	8	100 000
63	32	150 000
250	125	200 000
1 000	500	250 000

Sur la base de ces mesures, il serait facile de conclure que l'ordinateur A exécute un algorithme dont l'efficacité est bien supérieure à celle de l'ordinateur B. Cependant, si la taille de la liste d'entrée est augmentée à un nombre suffisant, cette conclusion est dramatique s'est avéré erroné :

n (taille de la liste)	Temps d'exécution de l'ordinateur A (en nanosecondes)	Temps d'exécution de l'ordinateur B (en nanosecondes)
16	8	100 000
63	32	150 000
250	125	200 000
1 000	500	250 000
...	...	...
1 000 000	500 000	500 000
4 000 000	2ÿ000ÿ000	550 000
16 000 000	8 000 000	600 000
...	...	...
63 072 × 1012	31 536 × 1012 ns, ou 1 an	1 375 000 ns, ou 1,375 millisecondes

L'ordinateur A, exécutant le programme de recherche linéaire, présente un taux de croissance linéaire . Le temps d'exécution du programme est directement proportionnel à sa taille d'entrée. Doubler la taille d'entrée double le temps d'exécution, quadrupler la taille d'entrée quadruple le temps d'exécution, et ainsi de suite. D'autre part, l'ordinateur B, exécutant le programme de recherche binaire, présente un taux de croissance logarithmique . Le fait de quadrupler la taille d'entrée n'augmente le temps d'exécution que d'une quantité constante (dans cet exemple, 50 000 ns). Même si l'ordinateur A est apparemment une machine plus rapide, l'ordinateur B dépassera inévitablement l'ordinateur A en termes d'exécution car il exécute un algorithme avec un taux de croissance beaucoup plus lent.

Ordres de croissance]

Article principal: notation Big O

De manière informelle, on peut dire qu'un algorithme présente un taux de croissance de l'ordre d'une fonction mathématique si au-delà d'une certaine taille d'entrée n, la fonction $f(n)$ fois une constante positive fournit une limite supérieure ou une limite pour le temps d'exécution de cet algorithme. Autrement dit, pour une donnée taille d'entrée n supérieure à un certain n0 et une constante c, le temps d'exécution de cet algorithme ne sera jamais plus grand que ce qui est $c \times f(n)$. Ce concept fréquemment exprimé en utilisant la notation Big O. Par exemple, puisque le temps d'exécution du tri par insertion croît de manière quadratique à mesure que sa taille d'entrée augmente, le tri par insertion peut être dit d'ordre O(n2).

La notation Big O est un moyen pratique d'exprimer le pire scénario pour un algorithme donné, bien qu'elle puisse également être utilisée pour exprimer le cas moyen - par exemple, le pire scénario pour le tri rapide est O (n2), mais le temps d'exécution moyen est O(n log n).

Ordres empiriques de croissance]

En supposant que le temps d'exécution suit la règle de puissance, *t ÿ k na*, le coefficient a peut être trouvé [8] en prenant des mesures empiriques du temps d'exécution et en cale $\{t_1, t_2\}$ à certains points problématiques $\{n_1, n_2\}$, $t_2/t_1 = (n_2/n_1)^a$ de sorte que $a = \log(t_2/t_1)/\log(n_2/n_1)$ Dans en d'autres termes, cela mesure la pente de la ligne empirique sur le tracé log-log du temps d'exécution en fonction de la taille du problème, à un certain point de taille. Si l'ordre de croissance suit effectivement la règle de puissance (et donc la ligne sur le tracé log-log est bien une ligne droite), la valeur empirique de a restera constante à différentes plages, et sinon, elle changera (et la ligne est une ligne courbe) - mais pourrait toujours servir à la comparaison de deux algorithmes donnés quant à leurs ordres locaux empiriques de comportement de croissance. Appliqué au tableau ci-dessusÿ:

n (taille de la liste)	Temps d'exécution de l'ordinateur A (en nanosecondes)	Ordre local de croissance (n^_)	Temps d'exécution de l'ordinateur B (en nanosecondes)	Ordre local de croissance (n^_)
15	...		100 000	
65	32	1.04	150 000	0,28
250	125	1.01	200 000	0,21
1 000	500	1,00	250 000	0,16
...	...		...	
1 000 000	500 000	1,00	500 000	0,10
4 000 000	2 000 000	1,00	550 000	0,07
16 000 000	8 000 000	1,00	600 000	0,06
...	...		...	

On voit clairement que le premier algorithme présente un ordre linéaire de croissance suivant bien la règle de puissance. Les valeurs empiriques du second diminuent rapidement, ce qui suggère qu'il suit une autre règle de croissance et qu'il a en tout cas des ordres de croissance locaux beaucoup plus faibles (et s'améliore encore), empiriquement, que le premier.

Évaluation de la complexité d'exécution [modifier]

La complexité d'exécution pour le pire scénario d'un algorithme donné peut parfois être évaluée en examinant la structure de l'algorithme et en faisant des hypothèses simplificatrices. Considérez le pseudo-code suivantÿ:

```
1    obtenir un entier positif n de l'entrée si n > 10 imprimer "Cela
2    peut prendre un certain temps..." pour i = 1 à n
3
4
5        for j = 1 to i print i * j
6            print "Done!"
7
```

Un ordinateur donné prendra un temps discret pour exécuter chacune des instructions impliquées dans l'exécution de cet algorithme. La durée spécifique d'exécution d'une instruction donnée variera en fonction de l'instruction en cours d'exécution et de l'ordinateur qui l'exécute, mais sur un ordinateur conventionnel, cette durée sera déterministe. [9] Supposons que les actions effectuées à l'étape 1 sont considérées comme consommant le temps T1, l'étape 2 utilise le temps T2, et ainsi de suite.

Dans l'algorithme ci-dessus, les étapes 1, 2 et 7 ne seront exécutées qu'une seule fois. Pour une évaluation dans le pire des cas, il convient de supposer que l'étape 3 sera également exécutée. Ainsi, la durée totale d'exécution des étapes 1 à 3 et de l'étape 7 estÿ:

$$T_1 + T_2 + T_3 + T_7.$$

Les boucles des étapes 4, 5 et 6 sont plus délicates à évaluer. Le test de boucle externe à l'étape 4 s'exécutera (n + 1) fois (notez qu'une étape supplémentaire est nécessaire pour terminer la boucle for, donc n + 1 et non n exécutions), ce qui consommera le temps T4(n + 1). La boucle interne, en revanche, est régie par la valeur de j, qui itère de 1 à i. Lors du premier passage dans la boucle externe, j itère de 1 à 1ÿ: la boucle interne effectue une passe, donc l'exécution du corps de la boucle interne (étape 6) consomme du temps T6 , et le test de la boucle interne (étape 5) consomme du temps 2T5 .
Lors du prochain passage dans la boucle externe, j itère de 1 à 2ÿ: la boucle interne effectue deux passages, donc l'exécution du corps de la boucle interne (étape 6) consomme 2T6 temps, et le test de la boucle interne (étape 5) consomme 3T5 temps.

Au total, le temps total nécessaire pour exécuter le corps de la boucle interne peut être exprimé sous la forme d'une progression arithmétiqueÿ:

$$T_6 + 2T_6 + 3T_6 + \cdots + (n-1)T_6 + nT_6$$

qui peut être factorisé[10] comme

$$T_6\left[1 + 2 + 3 + \cdots + (n-1) + n\right] = T_6\left[\frac{1}{2}(n^2 + n)\right]$$

Le temps total requis pour exécuter le test de boucle externe peut être évalué de la même manièreÿ:

$$\begin{aligned}&2T_5 + 3T_5 + 4T_5 + \cdots + (n-1)T_5 + nT_5 + (n+1)T_5\\ =\ &T_5 + 2T_5 + 3T_5 + 4T_5 + \cdots + (n-1)T_5 + nT_5 + (n+1)T_5 - T_5\end{aligned}$$

qui peut être factorisé comme

$$\begin{aligned}&T_5\left[1 + 2 + 3 + \cdots + (n-1) + n + (n+1)\right] - T_5\\ =&\left[\frac{1}{2}(n^2 + n)\right]T_5 + (n+1)T_5 - T_5\\ =&T_5\left[\frac{1}{2}(n^2 + n)\right] + nT_5\\ =&\left[\frac{1}{2}(n^2 + 3n)\right]T_5\end{aligned}$$

Par conséquent, le temps d'exécution total de cet algorithme estÿ:

$$f(n) = T_1 + T_2 + T_3 + T_7 + (n+1)T_4 + \left[\frac{1}{2}(n^2 + n)\right]T_6 + \left[\frac{1}{2}(n^2 + 3n)\right]T_5$$

qui se réduit à

$$f(n) = \left[\frac{1}{2}(n^2 + n)\right]T_6 + \left[\frac{1}{2}(n^2 + 3n)\right]T_5 + (n+1)T_4 + T_1 + T_2 + T_3 + T_7$$

En règle générale, on peut supposer que le terme d'ordre le plus élevé dans une fonction donnée domine son taux de croissance et définit ainsi son ordre d'exécution. Dans cet exemple, n2 est le terme d'ordre le plus élevé, on peut donc conclure que f(n) = O(n2). Formellement, cela peut être prouvé comme suit :

Prouve-le $\left[\frac{1}{2}(n^2 + n)\right]T_6 + \left[\frac{1}{2}(n^2 + 3n)\right]T_5 + (n+1)T_4 + T_1 + T_2 + T_3 + T_7 \leq cn^2,\ n \geq n_0$

$$\begin{aligned}&\left[\frac{1}{2}(n^2 + n)\right]T_6 + \left[\frac{1}{2}(n^2 + 3n)\right]T_5 + (n+1)T_4 + T_1 + T_2 + T_3 + T_7\\ \leq&(n^2 + n)T_6 + (n^2 + 3n)T_5 + (n+1)T_4 + T_1 + T_2 + T_3 + T_7\ (\text{for } n \geq 0)\end{aligned}$$

Soit k une constante supérieure ou égale à [T1..T7]

$$\begin{aligned}&T_6(n^2 + n) + T_5(n^2 + 3n) + (n+1)T_4 + T_1 + T_2 + T_3 + T_7 \leq k(n^2 + n) + k(n^2 + 3n) + kn + 5k\\ =&2kn^2 + 5kn + 5k \leq 2kn^2 + 5kn^2 + 5kn^2\ (\text{for } n \geq 1) = 12kn^2\end{aligned}$$

Donc

$$\left[\frac{1}{2}(n^2 + n)\right]T_6 + \left[\frac{1}{2}(n^2 + 3n)\right]T_5 + (n+1)T_4 + T_1 + T_2 + T_3 + T_7 \leq cn^2, n \geq n_0 \text{ for } c = 12k, n_0 = 1$$

Une approche plus élégante pour analyser cet algorithme serait de déclarer que [T1..T7] sont tous égaux à une unité de temps, dans un système d'unités choisi de sorte qu'une unité soit supérieure ou égale aux temps réels de ces étapes . Cela signifierait que le temps d'exécution de l'algorithme se décompose comme suitÿ:[11]

$$4 + \sum_{i=1}^{n} i \leq 4 + \sum_{i=1}^{n} n = 4 + n^2 \leq 5n^2\ (\text{for } n \geq 1) = O(n^2).$$

Analyse du taux de croissance des autres ressources [modifier]

La méthodologie d'analyse d'exécution peut également être utilisée pour prédire d'autres taux de croissance, tels que la consommation d' espace mémoire. À titre d'exemple, considérons le pseudocode suivant qui gère et réaffecte l'utilisation de la mémoire par un programme en fonction de la taille d'un fichier géré par ce programmeÿ:

```
tant que le fichier est toujours ouvert : soit
    n = taille du fichier pour chaque 100
    000 kilo -octets d'augmentation de la taille du fichier double de la quantité de mémoire
        réservée
```

Dans ce cas, à mesure que la taille du fichier n augmente, la mémoire sera consommée à un taux de croissance exponentiel , qui est d'ordre O(2n). Il s'agit d'un taux de croissance extrêmement rapide et très probablement ingérable pour la consommation de ressources mémoire.

Pertinence [modifier]

L'analyse d'algorithme est importante dans la pratique car l'utilisation accidentelle ou involontaire d'un algorithme inefficace peut avoir un impact significatif sur les performances du système. Dans les applications sensibles au temps, un algorithme prenant trop de temps à s'exécuter peut rendre ses résultats obsolètes ou inutiles. Un algorithme inefficace peut également finir par nécessiter une quantité non économique de puissance de calcul ou de stockage pour fonctionner, le rendant à nouveau pratiquement inutile.

Facteurs constants _]

L'analyse des algorithmes se concentre généralement sur la performance asymptotique, en particulier au niveau élémentaire, mais dans les applications pratiques, les facteurs constants sont importants et les données du monde réel sont en pratique toujours limitées en taille. La limite est généralement la taille de la mémoire adressable, donc sur les machines 32 bits 232 = 4 Gio (plus grande si la mémoire segmentée est utilisée) et sur les machines 64 bits 264 = 16 EiB. Ainsi étant donné une taille limitée, un ordre de croissance (temps ou espace) peut être remplacé par un facteur constant, et en ce sens tous les algorithmes pratiques sont O(1) pour une constante suffisamment grande, ou pour des données suffisamment petites.

Cette interprétation est principalement utile pour les fonctions qui croissent extrêmement lentement : le logarithme itéré (binaire) (log*) est inférieur à 5 pour toutes les données pratiques (265 536 bits) ; le log-log (binaire) (log log n) est inférieur à 6 pour pratiquement toutes les données pratiques (264 bits) ; et le log binaire (log n) est inférieur à 64 pour pratiquement toutes les données pratiques (264 bits). Un algorithme à complexité non constante peut néanmoins être plus efficace qu'un algorithme à complexité constante sur des données pratiques si la surcharge de l'algorithme à temps constant se traduit par un facteur constant plus grand, par exemple, on peut avoir $K > k \log\log n$ tant que $K/k > 6$ et $n < 2^{2^6} = 2^{64}$.

Pour les grandes données, les facteurs linéaires ou quadratiques ne peuvent pas être ignorés, mais pour les petites données, un algorithme asymptotiquement inefficace peut être plus efficace. Ceci est particulièrement utilisé dans les algorithmes hybrides, comme Timsort, qui utilisent un algorithme asymptotiquement efficace (ici le tri par fusion, avec des données de complexité temporelle $n \log n$) mais passent à un algorithme asymptotiquement inefficace (ici le tri par insertion, avec une complexité temporelle n^2) pour les petites données

Voir aussi [modifier]

- Analyse amortie
- Analyse d'algorithmes parallèles
- Complexité de calcul asymptotique
- Meilleur, pire et cas moyen
- Notation grand O
- Théorie de la complexité computationnelle
- Théorème maître (analyse d'algorithmes)
- NP-Complet
- Analyse numérique
- Temps polynomial
- Optimisation du programme
- Profilage (programmation informatique)
- Évolutivité
- Analyse lissée
- Analyse de terminaison - le sous-problème consistant à vérifier si un programme se terminera du tout
- Complexité temporelle - comprend un tableau des ordres de croissance pour les algorithmes courants
- Complexité basée sur l'information

Remarques [modifier]

1. ^ "Knuthÿ: Nouvelles récentes" 28 août 2016. Archivé de l' original le 28 août 2016
2. ^ Alfred V. Aho; John E. Hopcroft; Jeffrey D. Ullman (1974). La conception [redacted] sley. Co., section 1.3
3. ^ Juraj Hromkovic (2004). Informatique théorique : introduction aux automates, calculabilité, complexité, algorithmique, randomisation, communication et cryptographie . Springer. p. 177–178. ISBN 978-3-540-14015-3.
4. ^ Giorgio Ausiello (1999). Complexité et approximationÿ: problèmes d'optimisation combinatoire et leurs propriétés d'approximation . Springer. p. 3–8. ISBN 978-3-540-65431-5.
5. ^ Wegener, Ingo (2005), Théorie de la complexité : explorer les limites of efficient algorithms , Berlin, New York : Springer-Verlag, p. 20, ISBN 978-3-540-21045-0
6. ^ Robert Endre Tarjan (1983). Structures de données et algorithmes de réseau . SIAM. p. 3–7. ISBN 978-0-89871-187-5.
7. ^ Des exemples du prix de l'abstraction ? , cstheory.stackexchange.com
8. ^ Comment éviter les abus et les pots-de-vin archivés 2017-03-08 à la Wayback Machine, sur le blog "Gödel's Lost Letter and P=NP" de RJ Lipton, professeur d'informatique à Georgia Tech, racontant l'idée de Robert Sedgewick
9. ^ Cependant, ce n'est pas le cas avec un ordinateur quantique 10. ^ On peut prouver par induction que $1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$
11. ^ Cette approche, contrairement à l'approche ci-dessus, néglige la temps constant consommé par les tests de boucle qui terminent leurs boucles respectives, mais il est trivial de prouver qu'une telle omission n'affecte pas le résultat final

Références [modifier]

- Sedgewick, Robert; Flajolet, Philippe (2013). Une introduction à l'analyse des algorithmes (2e éd.). Addison-Wesley. ISBN 978-0-321-90575-8.
- Greene, Daniel A.; Knuth, Donald E. (1982). Mathématiques pour l'analyse des algorithmes (deuxième éd.). Birkhauser. ISBN 3-7643-3102-X.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. & Stein, Clifford (2001). Introduction aux algorithmes. Chapitre 1ÿ: Fondations (deuxième éd.). Cambridge, MAÿ: MIT Press et McGraw-Hill. p. 3–122. ISBN 0-262-03293-7.
- Sedgewick, Robert (1998). Algorith[redacted]: Addison-Wesley Professional. ISBN 978-0-201-31452-6.
- Knuth, Donald. L'art de la programmation informatique. Addison-Wesley.
- Goldreich, Oded (2010). Complexité computationnelleÿ: une perspective conceptuelle. La presse de l'Universite de Cambridge. ISBN 978-0-521-88473-0.

Liens externes [modifier]

- Médias liés à l' analyse des algorithmes sur Wikimedia Commons

V · T · E L'informatique [Afficher]

Catégories : Analyse d'algorithmes Théorie de la complexité computationnelle