Algorithmique et ComplexiTé
Rappels(?):
Preuve et complexité des algorithmes

Master1 Informatique

Sophie Tison
sophie.tison@univ-lille.fr

Université de Lille

Un premier rappel: comment prouver qu'un algorithme est correct

Université de Lille

PROUVER QU'UN PROGRAMME EST CORRECT?

Des estimations évaluent les pertes annuelles liées aux erreurs de programmation en Europe à plus de 100 milliards d'euros.

Les exemples de bugs célèbres sont nombreux: Sonde Mariner 1, Ariane V, Panne d'Electricité en Amérique du Nord en 2003, F22 Raptor en 2007, BNP Paribas en 2009, ...
Certains bugs ont eu des conséquences dramatiques comme celui du Therac-25.

PROUVER QU'UN PROGRAMME EST CORRECT???

TESTER?

Le test est un outil essentiel mais ne garantit pas l'absence de bugs.

"Program testing can be used to show the presence of bugs, but never to show their absence !" Edsger W. Dijkstra

Prouver qu'un programme est correct???

Preuve formelle?

Elle peut être:

- Manuelle
- Automatique – donc limitée
- Interactive comme par exemple avec Coq.

L'assistant de preuve Coq

Coq Recipient Of The ACM 2013 Software System Award: "Coq is a software tool for the interactive development of formal proofs, which is a key enabling technology for certified software. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs."
Coq a par exemple permis la réalisation de CompCert, compilateur certifié C.
Il a aussi permis de prouver le théorème des 4 couleurs - toute carte planaire peut être coloriée avec 4 couleurs.

Prouver qu'un programme est correct???

Preuve formelle?

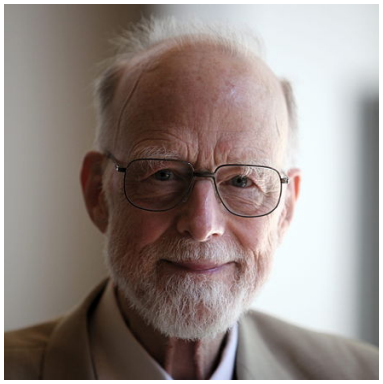
"One does not need to give a formal proof of an obviously correct program; but one needs a thorough understanding of formal proof methods to know when correctness is obvious." John C. Reynolds

Prouver la correction d'un programme,

c'est prouver qu'il correspond à la spécification donnée.

La Logique de Hoare-Floyd a été introduite à la fin des années 60 pour formaliser la relation entre langage de programmation (impératif) et langage de spécification.

Charles Antony Richard Hoare



Charles Anthony Richard Hoare, lauréat du prestigieux Prix Turing (1980), a eu de nombreuses contributions remarquables, comme la logique de Hoare mais aussi le Quicksort, un compilateur Algol 60 ou la définition de CSP, un modèle pour la programmation concurrente.

Triplets de Hoare

La logique de Hoare-Floyd se base sur la notion de *triplet de Hoare*, de la forme

$$\{P\}\ \mathtt{A}\ \{Q\}$$

P est la précondition,

Q la postcondition,

A une expression du langage de programmation.

Spécification

$$\{P\}\ \mathtt{A}\ \{Q\}$$

P et Q sont des assertions, i.e. des formules du langage de spécification décrivant les valeurs des variables, l'état de la mémoire ou du système.

En toute rigueur, le langage de spécification devrait être un langage mathématique comme la logique des prédicats, mais dans un objectif de documentation plus que de preuve de programmes, le langage de spécification peut être le langage naturel (en évitant les ambiguités!).

Interprétation

L'interprétation de $\{P\}\ \mathtt{A}\ \{Q\}$ est:

si le système vérifie $\{P\}$, on peut assurer qu'il vérifie $\{Q\}$ après l'exécution de `A`;

En fait, c'est un peu plus compliqué que cela; en effet, il se peut que l'exécution de `A` ne termine pas! On a donc deux interprétations possibles:

- si $\{P\}$ est vérifiée, après l'exécution de `A`, si celle-ci termine, on peut assurer $\{Q\}$; on parle alors de correction partielle.
- si $\{P\}$ est vérifiée, la terminaison de `A` est assurée et après son exécution, on peut assurer $\{Q\}$; on parle alors de correction totale.

PROUVER UN PROGRAMME REVIENT DONC À PROUVER

{Précondition sur les données}
`Programme`
{condition exprimant les résultats attendus}

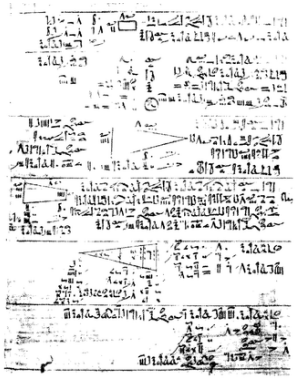
à partir des axiomes et des règles d'inférences de Hoare qui montrent comment inférer la correction d'instructions composées à partir de celles de base -données par la sémantique du langage-.

UN EXEMPLE

Input: Integers $x \geq 0$ and $b \geq 0$
Output: R=???
int $R = 0, a = x, b = y;$
while $b > 0$ **do**
  if *b impair* **then**
    $R = R + a;$
  end
  $a = 2 * a$;
  $b = b \div 2$;
end

Algorithm 1: Mystère

UN EXEMPLE: MULTIPLICATION À L'ÉGYPTIENNE (OU À L'ÉTHIOPIENNE OU À LA RUSSE, ...)



Le papyrus Rhind

MULTIPLICATION À L'ÉGYPTIENNE

Input: Integers $x \geq 0$ and $b \geq 0$
Output: $R = x * y$?
int $R = 0, a = x, b = y;$
while $b > 0$ **do**
  if *b impair* **then**
    $R = R + a;$
  end
  $a = 2 * a$;
  $b = b \div 2$;
end

Algorithm 2: Multiplication à l'égyptienne

Multiplication à l'égyptienne

Input: Integers $x \geq 0$ and $b \geq 0$
Output: $R = x * y$?
int $R = 0, a = x, b = y$;
while $b > 0$ **do**
  // ???
  if *b impair* **then**
    $R = R + a$;
  end
  $a = 2 * a$;
  $b = b \div 2$;
end

Algorithm 3: Multiplication à l'égyptienne

Multiplication à l'égyptienne

Input: Integers $x \geq 0$ and $b \geq 0$
Output: $R = x * y$?
int $R = 0, a = x, b = y$;
while $b > 0$ **do**
  // $R + a * b = x * y, b \geq 0$
  if *b impair* **then**
    $R = R + a$;
  end
  $a = 2 * a$;
  $b = b \div 2$;
end

Algorithm 4: Multiplication à l'égyptienne

Multiplication à l'égyptienne

Input: Integers $x \geq 0$ and $b \geq 0$
Output: $R = x * y$?
int $R = 0, a = x, b = y$;
while $b > 0$ **do**
  // $R + a * b = x * y, b \geq 0$
  // $R + 2 * a * (b \div 2) + a * (b \mod 2) = x * y$
  if *b impair* **then**
    $R = R + a$;
  end
  $a = 2 * a$;
  $b = b \div 2$;
end

Algorithm 5: Multiplication à l'égyptienne

Multiplication à l'égyptienne

Input: Integers $x \geq 0$ and $b \geq 0$
Output: $R = x * y$?
int $R = 0, a = x, b = y$;
while $b > 0$ **do**
  // $R + a * b = x * y, b \geq 0$
  // $R + 2 * a * (b \div 2) + a * (b \mod 2) = x * y$
  if *b impair* **then**
    $R = R + a$;
  end
  // $R + 2 * a * (b \div 2) = x * y$
  $a = 2 * a$;
  $b = b \div 2$;
end

Algorithm 6: Multiplication à l'égyptienne

Multiplication à l'égyptienne

Input: Integers $x \geq 0$ and $b \geq 0$
Output: $R = x * y$?
int $R = 0, a = x, b = y$;
while $b > 0$ **do**
  // $R + a * b = x * y, b \geq 0$
  // $R + 2 * a * (b \div 2) + a * (b \mod 2) = x * y$
  if *b impair* **then**
    $R = R + a$;
  end
  // $R + 2 * a * (b \div 2) = x * y$
  $a = 2 * a$;
  // $R + a * (b \div 2) = x * y$
  $b = b \div 2$;
end

Algorithm 7: Multiplication à l'égyptienne

Multiplication à l'égyptienne

Input: Integers $x \geq 0$ and $b \geq 0$
Output: $R = x * y$?
int $R = 0, a = x, b = y$;
while $b > 0$ **do**
  // $R + a * b = x * y, b \geq 0$
  // $R + 2 * a * (b \div 2) + a * (b \mod 2) = x * y, b \geq 0$
  if *b impair* **then**
    $R = R + a$;
  end
  // $R + 2 * a * (b \div 2) = x * y, b \geq 0$
  $a = 2 * a$;
  // $R + a * (b \div 2) = x * y, b \geq 0$
  $b = b \div 2$;
  // $R + a * b = x * y, b \geq 0$
end

Algorithm 8: Multiplication à l'égyptienne

Multiplication à l'égyptienne

Input: Integers $x \geq 0$ and $b \geq 0$
Output: $R = x * y$
int $R = 0, a = x, b = y$;
while $b > 0$ **do**
  if *b impair* **then**
    $R = R + a$;
  end
  $a = 2 * a$;
  $b = b \div 2$;
end

Algorithm 9: Multiplication à l'égyptienne

L'algorithme s'arrête puisque b décroît strictement à chaque étape.

Les règles d'inférence de Hoare

La règle pour la Séquence

si
$\{P\}$ `A` $\{Intermede\}$ et $\{Intermede\}$ `B` $\{Q\}$

alors:

$\{P\}$ `A ; B` $\{Q\}$

La règle de la conditionnelle

si

$\{P \wedge cond\}$ `A` $\{Q\}$ et $\{P \wedge \neg cond\}$ `B` $\{Q\}$

alors:

$\{P\}$ `si cond alors A sinon B` $\{Q\}$

La règle pour le "tant que'

$\{P\}$ `tant que cond faire A` $\{Q\}$??

si on trouve une assertion i, qu'on appelle invariant, telle que:
$P \Rightarrow i$

$\{i \wedge cond\}$ `A` $\{i\}$

$\{i \wedge \neg cond\} \Rightarrow \{Q\}$

alors:

$\{P\}$ `tant que cond faire A` $\{Q\}$

Attention: pour avoir une preuve de correction totale, il faut aussi faire une preuve d'arrêt de la boucle!

L'affectation

$P[x \leftarrow exp]$ `x=exp` P

en supposant que exp ne contient pas de fonctions à effet de bord.

Par exemple:

$x + y < 5$ `x=x+y` $x < 5$

Le renforcement de la précondition

si

$P \Rightarrow Q$

et

$\{Q\}$ `A` $\{R\}$

alors

$\{P\}$ `A` $\{R\}$

L'AFFAIBLISSEMENT DE LA POSTCONDITION

si
$\{P\}$ A $\{Q\}$
et
$Q \Rightarrow R$
alors

$\{P\}$ A $\{R\}$

Soit: Qui peut le plus, peut le moins!

MULTIPLICATION À L'ÉGYPTIENNE: VERSION RÉCURSIVE

Input: $x \geq 0$ and $b \geq 0$ entiers
Output: retourne $x * y$
int Mult (int x, int y)
if *(y==0)* **then**
  return *0*
else
  return *Mult(2x, y div 2) +x* y mod 2;*
end

Preuve par induction (récurrence) sur y.

Un deuxième rappel: comment mesurer l'efficacité d'un algorithme

Complexité des algorithmes

Université de Lille

ANALYSER LA COMPLEXITÉ D'UN ALGORITHME?

Analyser la complexité d'un algorithme doit permettre de mesurer l'efficacité de l'algorithme.

On pourrait d'abord se poser la question:

Qu'est-ce qu'un algorithme efficace?

Qu'est-ce qu'un algorithme efficace?

Tentative de définition: Un algorithme est efficace si, une fois implémenté, il s'exécute rapidement sur toute donnée "réelle".

- Première remarque: attention à l'implémentation
- Deuxième remarque: on devrait pouvoir mesurer l'efficacité indépendamment de la plate-forme
- Troisième remarque: qu'est-ce qu'une donnée réelle? Souvent les problèmes d'efficacité surgissent quand la taille des données grandit...
- Quatrième remarque: ne prend en compte que la ressource "temps".

Qu'est-ce qu'un algorithme efficace?

Deuxième Tentative de définition?
Un algorithme efficace est un algorithme qui se comporte de façon raisonnable indépendamment de la plate-forme même pour des données de taille assez grande...

On proposera une définition un peu plus précise tout à l'heure!

Analyser la complexité d'un algorithme c'est

Exprimer le coût de l'algorithme - la quantité de ressources utilisées - *en fonction de la taille des données*

Idée: on essaie de prévoir le comportement en fonction de la taille des données.

Quelles ressources?

la mémoire: *complexité spatiale*

le temps : *complexité temporelle*

ou le nb de processeurs, les communications,....

COMMENT MESURER LA TAILLE DES DONNÉES

. a priori la taille des données est **le nombre de bits de leur représentation en binaire**;

. même si dans certains cas, on étudie la complexité en fonction d'une autre notion de taille, comme le nombre d'éléments pour une liste, la dimension pour une matrice, . . .

COMMENT MESURER LA TAILLE DES DONNÉES

Quelle est la taille d'un entier n?
C'est le nombre de bits de sa représentation binaire.

Quelle est la "taille de 2017"?

$$2^{10} \leq 2017 < 2^{11}$$

Donc, la taille de la représentation en binaire de 2017 est 11.

La taille d'un entier n est environ $\log_2 n$ ($\lfloor(\log_2 n)\rfloor + 1$).

QUEL EST LE COÛT DE L'ALGORITHME A POUR UNE DONNÉE d - NOTÉ $cout_A(d)$-?

UN MODÈLE DE CALCUL?

. Pour évaluer ce coût indépendamment de la machine (physique) , on utilise un *modèle* de machines séquentielles "classiques", en général la RAM ("Random Access Machine").

.Chaque instruction "simple' ($+, *, -$, affectation, comparaison, appel, accès mémoire ...) a un coût uniforme de 1, i.e. indépendant de la taille de ses opérandes par opposition au coût logarithmique où on tient compte de la taille des données.

QUEL EST LE COÛT DE L'ALGORITHME A POUR UNE DONNÉE d - NOTÉ $cout_A(d)$-?

ON SIMPLIFIE!

On se borne souvent à "compter" certaines instructions: comparaisons pour un tri par comparaisons, accès à la mémoire externe pour un tri externe, opérations arithmétiques de base pour des produits de matrices...

Quelle complexité?

D'une donnée à l'autre!

Pour deux données de taille n, le coût peut être différent!

- *La complexité dans le meilleur des cas:*

$$Inf_A(n) = inf\{cout_A(d)/d \text{ de taille } n\}$$

- *La complexité dans le pire des cas:*

$$Sup_A(n) = sup\{cout_A(d)/d \text{ de taille } n\}$$

- *La complexité en moyenne:* pour la définir, il faut disposer pour tout n d'une mesure de probabilité p sur l'ensemble des données de taille n;

$$Moy_A(n) = \sum_{d \text{ de taille } n} p(d) * cout(d)$$

En résumé...

Quand on parle de la complexité d'un algorithme sans préciser laquelle, c'est souvent de la complexité temporelle dans le pire des cas qu'on parle.

Ordres de grandeur

On ne calcule pas en général la complexité exacte mais on se contente de calculer son ordre de grandeur asymptotique voire de borner celui-ci.

Par exemple, si on fait $n^2 + 2n - 5$ opérations, on retiendra juste que l'ordre de grandeur est n^2. On utilisera donc les notions classiques d'ordre de grandeur: Θ, O, o, Ω.

Ordres de grandeur: définitions

"Grand O"

Soient f et g deux fonctions de $\mathcal{N}$ dans $\mathcal{R}$:

$f \in O(g)$

Ssi_{def}

$\exists c \in \mathcal{R}^{+*}, \exists \mathcal{A}$ tels que $\forall n > A, f(n) \leq c * g(n)$

On dit que f est dominée asymptotiquement par g.
On notera souvent $f = O(g)$.

Si f/g a une limite finie en $+\infty$, alors $f = O(g)$.

ORDRES DE GRANDEUR: DÉFINITIONS
Ω

$f \in \Omega(g)$ Ssi_{def} $\exists C \in \mathcal{R}^{+*}$,tels que $\forall n > A, C * g(n) \leq f(n)$

On dit que f domine asymptotiquement par g.
On notera souvent $f = \Omega(g)$. On a alors $g = O(f)$.

ORDRES DE GRANDEUR: DÉFINITIONS
Θ

Soient f et g deux fonctions de $\mathcal{N}$ dans $\mathcal{R}$:

$f \in \Theta(g)$ Ssi_{def} $\exists c, C \in \mathcal{R}^{+*}, \exists \mathcal{A}$,tels que
$\forall n > A, C * g(n) \leq f(n) \leq c * g(n)$

On dit alors que f et g sont de même ordre de grandeur asymptotique.

On notera souvent $f = \Theta(g)$.
On a $f = \theta(g)$ Ssi $f = O(g)$ et $g = O(f)$.

Si f/g a une limite finie non nulle en $+\infty$, alors $f = \Theta(g)$.

ORDRES DE GRANDEUR: DÉFINITIONS
"PETIT O"

$f \in o(g)$ Ssi_{def} $\forall \epsilon \in \mathcal{R}^{+*}, \exists \mathcal{A}$,tels que $\forall n > A, f(n) \leq \epsilon * g(n)$

On dit que $f$est négligeable asymptotiquement devant g.

On notera souvent $f = o(g)$

Si f/g a pour limite 0 en $+\infty$, alors $f = o(g)$.

ORDRES DE GRANDEUR: EXEMPLES

- $n^3 + 3n + 7 \in \theta(n^3)$
- $5 * n^3 + 3n + 7 \in \theta(n^3)$
- $5 * n^3 + 3n + 7 \in O(n^3)$
- $5 * n^3 + 3n + 7 \in O(n^4)$
- $5 * n^3 + 3n + 7 \in o(n^4)$

CLASSER PAR ORDRE DE GRANDEUR

On peut trier des fonctions par "ordre" asymptotique de grandeur: si $f = o(g)$, on dira que $f < g$.
Par exemple $\sqrt{n} < n^5 < n^8$.

Trier les fonctions suivantes:

$$n^3 \quad 3^n \quad \log n \quad n^2 \quad \sqrt{n} \quad n^n$$

$$? \quad ? \quad ? \quad ? \quad ? \quad ?$$

CLASSER PAR ORDRE DE GRANDEUR

On peut trier des fonctions par "ordre" asymptotique de grandeur: si $f = o(g)$, on dira que $f < g$.
Par exemple $\sqrt{n} < n^5 < n^8$.

Trier les fonctions suivantes:

$$n^3 \quad 3^n \quad \log n \quad n^2 \quad \sqrt{n} \quad n^n$$

$$\log n \quad ? \quad ? \quad ? \quad ? \quad ?$$

CLASSER PAR ORDRE DE GRANDEUR

On peut trier des fonctions par "ordre" asymptotique de grandeur: si $f = o(g)$, on dira que $f < g$.
Par exemple $\sqrt{n} < n^5 < n^8$.

Trier les fonctions suivantes:

$$n^3 \quad 3^n \quad \log n \quad n^2 \quad \sqrt{n} \quad n^n$$

$$\log n \quad \sqrt{n} \quad ? \quad ? \quad ? \quad ?$$

CLASSER PAR ORDRE DE GRANDEUR

On peut trier des fonctions par "ordre" asymptotique de grandeur: si $f = o(g)$, on dira que $f < g$.
Par exemple $\sqrt{n} < n^5 < n^8$.

Trier les fonctions suivantes:

$$n^3 \quad 3^n \quad \log n \quad n^2 \quad \sqrt{n} \quad n^n$$

$$\lg n \quad \sqrt{n} \quad n^2 \quad ? \quad ? \quad ?$$

CLASSER PAR ORDRE DE GRANDEUR

On peut trier des fonctions par "ordre" asymptotique de grandeur: si $f = o(g)$, on dira que $f < g$.
Par exemple $\sqrt{n} < n^5 < n^8$.

Trier les fonctions suivantes:

$$n^3 \quad 3^n \quad \lg n \quad n^2 \quad \sqrt{n} \quad n^n$$

$$\lg n \quad \sqrt{n} \quad n^2 \quad n^3 \quad ? \quad ?$$

CLASSER PAR ORDRE DE GRANDEUR

On peut trier des fonctions par "ordre" asymptotique de grandeur: si $f = o(g)$, on dira que $f < g$.
Par exemple $\sqrt{n} < n^5 < n^8$.

Trier les fonctions suivantes:

$$n^3 \quad 3^n \quad \lg n \quad n^2 \quad \sqrt{n} \quad n^n$$

$$\lg n \quad \sqrt{n} \quad n^2 \quad n^3 \quad 3^n \quad ?$$

CLASSER PAR ORDRE DE GRANDEUR

On peut trier des fonctions par "ordre" asymptotique de grandeur: si $f = o(g)$, on dira que $f < g$.
Par exemple $\sqrt{n} < n^5 < n^8$.

Trier les fonctions suivantes:

$$n^3 \quad 3^n \quad \log n \quad n^2 \quad \sqrt{n} \quad n^n$$

$$\log n \quad \sqrt{n} \quad n^2 \quad n^3 \quad 3^n \quad n^n$$

REMARQUE

Evaluer l'ordre de grandeur asymptotique du coût de l'algorithme en fonction de la taille des données... plutôt que la complexité exacte se justifie si les données manipulées sont de grande taille.
Il ne faut pas négliger trop vite les constantes .

Q?: A partir de quelle taille de données, un algorithme de complexité exacte $200 * n * \log_2 n$ sera-t-il plus intéressant qu'un algorithme en n^2 ?.

Un peu de vocabulaire: un algorithme est dit...

. en temps constant si sa complexité dans le pire des cas est bornée par une constante.

. linéaire (resp. linéairement borné) si sa complexité (dans le pire des cas) est en $\Theta(n)$ (resp. $O(n)$).

. quadratique (resp. au plus quadratique) si sa complexité (dans le pire des cas) est en $\Theta(n^2)$ (resp. en $O(n^2)$).

. polynomial ou polynomialement borné, si sa complexité (dans le pire des cas) est en $O(n^p)$ pour un certain p.

. au plus exponentiel si elle est en $O(2^{n^p})$, pour un certain $p > 0$.

Etre praticable ou ne pas être praticable?

Algorithme Praticable

Par "convention", un algorithme est dit praticable si il est polynomial, c.à.d. si sa complexité temporelle dans le pire des cas est polynomiale.

Les limites de la convention

- Si un algorithme est exponentiel dans le pire des cas, il peut être polynomial en moyenne; si les pires cas sont exceptionnels, il peut être praticable ...en pratique: certains algos impraticables selon la définition ci-dessus sont ... pratiqués tous les jours (comme lle simplexe).

-L'exécution d'un algorithme dont la complexité moyenne est de l'ordre de grandeur de n^5 microsecondes prendrait 30 ans si $n = 1000$.

Mais ce n'est pas une si mauvaise convention

+ En pratique, il s'avère que la plupart des algorithmes polynomiaux ont un comportement asymptotique équivalent à un polynôme de faible degré, 2 ou 3.

+ De plus, la classe des algorithmes polynomiaux a de bonnes propriétés de clôture (par exemple, une "séquence" de deux algorithmes polynomiaux est polynomiale).

+ Enfin, et surtout, elle est indépendante du modèle séquentiel "classique" choisi: un algorithme polynômial pour le modèle des machines de Turing, le sera pour une RAM et vice-versa .

ORDRES DE GRANDEUR: EXEMPLES

Exemples de temps d'exécution en fonction de la taille de la donnée et de la complexité de l'algorithme, si on suppose qu'une instruction est de l'ordre de la nanoseconde;

T.\C.	$\log n$	n	$n \log n$	n^2	2^n
10	$3ns$	$10ns$	$30ns$	$100ns$	$1\mu s$
100	$7ns$	$100ns$	$700ns$	$10\mu s$	$> 10^{11}$ *siècles*
1000	$10ns$	$1\mu s$	$1/100ms$	$1ms$	*astronomique*
10000	$13ns$	$10\mu s$	$1/7ms$	$1/10s$	*astronomique*
100000	$17ns$	$1/10ms$	$2ms$	$10s$	*astronomique*

QUELQUES EXEMPLES

Input: t tableau de n entiers, $n > 0$
for $(i = n-1; i > 0; i--)$ **do**
  for $(j = 0; j < i; j++)$ **do**
    if $(t[j+1] < t[j])$ **then**
      $echanger(t[j], t[j+1])$;
    end
  end
end

Algorithme en $\Theta(n^2)$, donc quadratique.
Nombre de permutations? de 0 à n(n-1)/2

QUELQUES EXEMPLES

Input: t tableau de n entiers, $n > 0$
$intmax = t[0]$;
for $(i = 0; i < n-1; i++)$ **do**
  if $(max < t[i])$ **then**
    $max = t[i]$;
  end
end

Algorithme en $\Theta(n)$, donc linéaire

QUELQUES EXEMPLES

Input: t tableau de n entiers, $n > 0$
boolean $permute = true$;
int $last = n-1$;
while $permute$ **do**
  $permute = false$;
  for $(j = 0; j < last; j++)$ **do**
    if $(t[j+1] < t[j])$ **then**
      $permute = true$; $echanger(t[j], t[j+1])$;
    end
  end
  $last = last - 1$;
end

dans le meilleur des cas en $\Theta(n)$
dans le pire des cas en $\Theta(n^2)$
Algorithme en $\Theta(n^2)$, donc quadratique. On peut montrer qu'il est quadratique en moyenne.

Complexité de la Multiplication à l'égyptienne

Input: $x \geq 0$ and $b \geq 0$ entiers
Output: $R = x * y$
int $R = 0, a = x, b = y$;
while $b > 0$ **do**
  if b *impair* **then**
    $R = R + a$;
  end
  $a = 2 * a$;
  $b = b \div 2$;
end

Le nombre d'opérations élémentaires est $\Theta(log_2(y))$.

L'algorithme est linéaire en coût uniforme mais quadratique en coût logarithmique.

Quelques exemples

Soit l'algorithme suivant pour tester si un nombre est premier:
Input: $n > 1$ entier
boolean *Premier*(*int n*)
for $(int\ i = 2; i * i <= n; i++)$ **do**
  if $n \mod i = 0$ **then**
    return false;
  end
end
return true;

Cet algorithme est-il polynomial? Non!!!

Attention à la taille de la donnée!

Attention à la taille de la donnée!

Input: $n > 1$ entier
boolean *Premier*(*intn*)
for $(int\ i = 2; i * i <= n; i++)$ **do**
  if $n \mod i = 0$ **then**
    return false;
  end
end
return true;
Supposons que le coût d'une division ou multiplication soit constant, indépendant de la taille de la donnée, donc en $\Theta(1)$, alors la complexité dans le pire des cas de l'algo est $\Theta(\sqrt{(n)})$.

mais la taille de la donnée est $\log n$
donc le coût est en $\Theta(2^{|d|/2})$

TESTER SI UN ENTIER EST PREMIER?

Le premier test de primalité prouvé polynomial a été défini en 2002 par Manindra Agrawal, Neeraj Kayal et Nitin Saxena.

```
int T1(int n)
int c = 0;
for (int i = 0; i < n; i++) do
|  c++;
end
for (int i = 10; i < n + 10; i = i + 10) do
|  for (int j = n; j > 0; j = j − 10) do
|  |  c++;
|  end
end
return c;
```

en $\Theta(n^2)$

```
int T2(int n)
int c = 0;
for (int i = 100; i > 1; i = i − 2) do
|  for (int j = i; j < n; j = j ∗ 2) do
|  |  c++;
|  end
end
return c;
```

en $\Theta(\log n)$

A RETENIR (ENTRE AUTRES) SUR LA COMPLEXITÉ DES ALGORITHMES

. Quand on parle de la Complexité c'est souvent de la complexité temporelle dans le pire des cas

. Praticable = polynomial

. L'analyse de la complexité d'un algorithme est souvent asymptotique.

. Evaluer l'ordre de grandeur de la complexité d'un algorithme ... n'exempte pas de tester et d'expérimenter

. Attention à la taille de la donnée!

. Un QCM d'auto-évaluation sur Moodle

. La plate-forme d'entraînement à l'algorithmique est ouverte (sera pleinement fonctionnelle d'ici quelques jours): http://contest.fil.univ-lille1.fr/
Vous pouvez vous entraîner sur trois petits problèmes:

- . Que le meilleur gagne !
- . Equilibre
- . Majorité absolue

. Semaine prochaine: Cours, Td et TP