Expression des algorithmes

un bon niveau d'abstraction

Groupe algorithmique de l'IREM de Grenoble
{Anne.Rasse,**Jean-Marc.Vincent**,Benjamin.Wack}@imag.fr
{Maryline.Althuser,Herve.Barbe}@ac-grenoble.fr

Grenoble Novembre 2015

EXPRESSION DES ALGORITHMES

ALGORITHME

Un algorithme c'est ...

1. un moyen de communiquer à propos d'un problème/solution ;
2. une manière de résoudre un problème donné ;
3. une "formalisation" d'une méthode (qui sera prouvée et évaluée)
4. le premier pas (obligatoire) vers une implantation dans un langage de programmation

ALGORITHME

Un algorithme c'est ...

1. un moyen de communiquer à propos d'un problème/solution ;
2. une manière de résoudre un problème donné ;
3. une "formalisation" d'une méthode (qui sera prouvée et évaluée)
4. le premier pas (obligatoire) vers une implantation dans un langage de programmation

Quelques règles

1. Il y a de nombreuses manières d'écrire un algorithme
   Trouver son propre style ! ... Mais rester cohérent
2. un algorithme prends des entrées (input) et produit des sorties (output)
   Celles ci doivent être définies précisément
3. un algorithme peut utiliser d'autres algorithmes
   Approche "top-down" ... mais ces algorithmes doivent également être présentés

inspiré de Louis-Noël Pouchet

EXPRESSION D'UN ALGORITHME

Le langage algorithmique est une **convention** qui permet d'exprimer à un **lecteur**

1. l'idée de l'algorithme (principe, déroulement,...)
2. lui permettre de faire la preuve de celui-ci et de pouvoir analyser sa complexité
3. de pouvoir le traduire facilement dans un langage de programmation

Le langage algorithmique est donc plus ou moins proche d'un langage de programmation.
Exemple : tri par insertion

EXPRESSION DES ALGORITHMES

TRI PAR INSERTION : TD D'ALGORITHMIQUE

On souhaite trier un tableau d'éléments comparables. Le tableau est de taille n et les cases sont indicées de 1 à n.

Une itération ($i = 2$ à n) à chaque pas de laquelle on insère à sa place l'élément d'indice i dans la séquence triée formée des $i - 1$ premiers éléments.

Initialement : l'élément 1 forme une séquence triée.

Finalement : les n éléments sont triés.

On effectue l'insertion par une recherche séquentielle de l'emplacement k de l'élément i, et un décalage vers la droite des éléments de k à $i - 1$.

L'algorithme classique effectue ces deux opérations ensemble, c'est-à-dire décale l'élément i vers la gauche (par un échange) jusqu'à ce qu'il atteigne sa "bonne" place.

TRI PAR INSERTION : OUVRAGE DE CORMEN ET AL.

```
TRI-INSERTION(A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la séquence triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← clé
```

seule l'idée est donnée,

il n'y a pas de déclaration de variables,

il est implicite que A est un tableau,

clé l'élément qui permet la comparaison, i,j des indices de tableau,...

TRI PAR INSERTION : OUVRAGE DE SEDGEWICK ET AL.

ALGORITHM 2.2 Insertion sort

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {  // Sort a[] into increasing order.
      int N = a.length;
      for (int i = 1; i < N; i++)
      {  // Insert a[i] among a[i-1], a[i-2], a[i-3]... ..
         for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
            exch(a, j, j-1);
      }
   }
   // See page 245 for less(), exch(), isSorted(), and main().
}
```

parti pris de décrire les algorithmes dans un langage de programmation ici Java

abstraction des opérateurs (less à la place de <),

syntaxe et propriétés du langage de programmation.

TRI PAR INSERTION : OUVRAGE DE DENENBERG ET AL.

procedure *InsertionSort*(**table** $A[0\,..\,n-1]$):
{Sort by inserting each item in position in the table of elements to its left}
    for i **from** 1 **to** $n-1$ **do**
        $j \leftarrow i$ {j scans to the left to find where $A[i]$ belongs}
        $x \leftarrow A[i]$
        while $j \geq 1$ **and** $A[j-1] > x$ **do**
            $A[j] \leftarrow A[j-1]$
            $j \leftarrow j-1$
        $A[j] \leftarrow x$

Algorithm 11.1 Insertion Sort.

mélange des approches

symboles et syntaxe spécifique

TRI PAR INSERTION : OUVRAGE DE KNUTH

5.2.1. Sorting by Insertion

One of the important families of sorting techniques is based on the "bridge player" method mentioned near the beginning of Section 5.2: Before examining record R_j, we assume that the preceding records $R_1, \ldots, R_{j-1}$ have already been sorted; then we insert R_j into its proper place among the previously sorted records. Several interesting variations on this basic theme are possible.

Straight insertion. The simplest insertion sort is the most obvious one. Assume that $1 < j \leq N$ and that records $R_1, \ldots, R_{j-1}$ have been rearranged so that

$$K_1 \leq K_2 \leq \cdots \leq K_{j-1}.$$

(Remember that, throughout this chapter, K_j denotes the key portion of R_j.) We compare the new key K_j with K_{j-1}, K_{j-2}, ..., in turn, until discovering that R_j should be inserted between records R_i and R_{i+1}; then we move records $R_{i+1}, \ldots, R_{j-1}$ up one space and put the new record into position $i+1$. It is convenient to combine the comparison and moving operations, interleaving them as shown in the following algorithm; since R_j "settles to its proper level" this method of sorting has often been called the *sifting* or *sinking* technique.

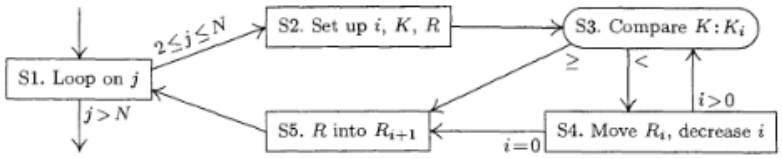


Fig. 10. Algorithm S: Straight insertion.

Algorithm S (*Straight insertion sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \leq \cdots \leq K_N$.

S1. [Loop on j.] Perform steps S2 through S5 for $j = 2, 3, \ldots, N$; then terminate the algorithm.

S2. [Set up i, K, R.] Set $i \leftarrow j-1$, $K \leftarrow K_j$, $R \leftarrow R_j$. (In the following steps we will attempt to insert R into the correct position, by comparing K with K_i for decreasing values of i.)

S3. [Compare $K : K_i$.] If $K \geq K_i$, go to step S5. (We have found the desired position for record R.)

S4. [Move R_i, decrease i.] Set $R_{i+1} \leftarrow R_i$, then $i \leftarrow i-1$. If $i > 0$, go back to step S3. (If $i = 0$, K is the smallest key found so far, so record R belongs in position 1.)

S5. [R into R_{i+1}.] Set $R_{i+1} \leftarrow R$. ∎

représentation graphique

schéma itératif (steps), non modulaire

TRI PAR INSERTION : OUVRAGE DE AHO ET AL.

Insertion Sorting

The second sorting method we shall consider is called "insertion sort," because on the i^{th} pass we "insert" the i^{th} element $A[i]$ into its rightful place among $A[1], A[2], \ldots, A[i-1]$, which were previously placed in sorted order. After doing this insertion, the records occupying $A[1], \ldots, A[i]$ are in sorted order. That is, we execute

for $i := 2$ **to** n **do**
move $A[i]$ forward to the position $j \leq i$ such that
$A[i] < A[k]$ for $j \leq k < i$, and
either $A[i] \geq A[j-1]$ or $j = 1$

To make the process of moving $A[i]$ easier, it helps to introduce an element $A[0]$, whose key has a value smaller than that of any key among $A[1], \ldots, A[n]$. We shall postulate the existence of a constant $-\infty$ of type keytype that is smaller than the key of any record that could appear in practice. If no constant $-\infty$ can be used safely, we must, when deciding whether to push $A[i]$ up before position j, check first whether $j = 1$, and if not, compare $A[i]$ (which is now in position j) with $A[j-1]$. The complete program is shown in Fig. 8.5.

```
(1)   A[0].key := -∞;
(2)   for i := 2 to n do begin
(3)       j := i;
(4)       while A[j] < A[j-1] do begin
(5)           swap(A[j], A[j-1]);
(6)           j := j-1
          end
      end
```

Fig. 8.5. Insertion sort.

TRI PAR INSERTION : WIKIPEDIA FR - EN

En français

```
procédure tri_insertion(tableau T, entier n)
    pour i de 1 à n-1
        x ← T[i]
        j ← i
        tant que j > 0 et T[j - 1] > x
            T[j] ← T[j - 1]
            j ← j - 1
        fin tant que
        T[j] ← x
   fin pour
fin procédure
```

En anglais version1

```
for i ← 1 to length(A) - 1
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
```

version 2

```
for i = 1 to length(A) - 1
   x = A[i]
   j = i
   while j > 0 and A[j-1] > x
       A[j] = A[j-1]
       j = j - 1
   A[j] = x
```

Autres sites

http://openclassrooms.com/courses/le-tri-par-insertion
http://rosettacode.org/wiki/Sorting_algorithms/Insertion_sort
http://www.sorting-algorithms.com/insertion-sort

EXPRESSION D'UN ALGORITHME : SYNTHÈSE

Conclusion ?

EXPRESSION DES ALGORITHMES

LE CRÊPIER

Crêpier-Itératif (T)

```
for i =Taille (T) to 2 do
    k = 1 ;
    for j = 2 to i
        if T[j] > T[k]
            k = j
    Retourne-préfixe (T,k)
    Retourne-préfixe (T,n)
```

LE CRÊPIER

Crêpier-Itératif (T)

for $i =$**Taille (**T**) to** 2 **do**
  $k = 1$;
  for $j = 2$ **to** i
    if $T[j] > T[k]$
      $k = j$
  Retourne-préfixe (T,k)
  Retourne-préfixe (T,n)

Retourne-préfixe (T,i)

$j = 1$ **while** $j < i - j + 1$
  Échange $(T,j,i-j+1)$
  $j = j + 1$

Échange (T,i,j)

$x = T[i]$
$T[i] = T[j]$
$T[j] = x$

LE CRÊPIER

Crêpier-Itératif (T)

Données: Un tableau T d'éléments comparables

Résultats: Les éléments rangés dans le même tableau par ordre croissant

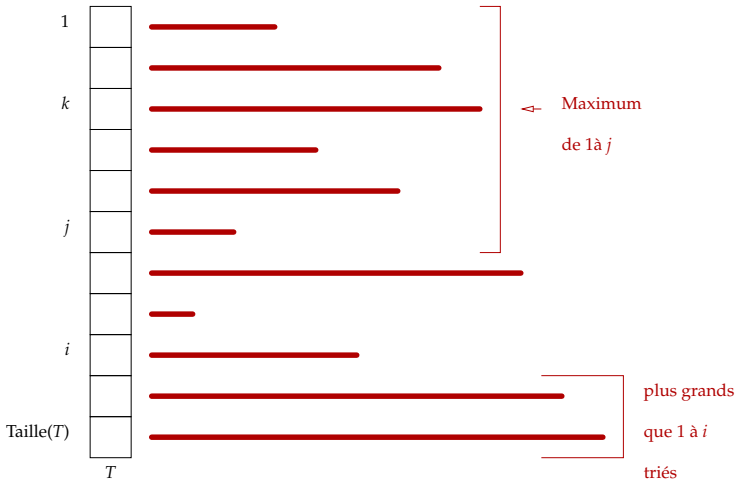
```
for i = Taille (T) to 2 do
    // Placer la bonne crêpe, la plus grande des i premières en position i
    k = 1; // indice de la plus grande crêpe
    for j = 2 to i
        // recherche de la plus grande crêpe
        if T[j] > T[k]
            k = j
    Retourne-préfixe (T,k)  // positionnement de la plus grande crêpe au sommet de la pile
    Retourne-préfixe (T,i)  // positionnement de la plus grande crêpe à sa position i
```

PREUVE DE L'ALGORITHME

Décrire l'état des variables de l'algorithme

Crêpier-Itératif (T)
Données: Un tableau T d'éléments comparables
Résultats: Les éléments rangés dans le même tableau par ordre croissant

for i = **Taille** (T) **to** 2 **do**
  // Placer la bonne crêpe, la plus grande des i premières en position i
  $k = 1$; // indice de la plus grande crêpe
  for $j = 2$ **to** i
    // recherche de la plus grande crêpe
    if $T[j] > T[k]$
      $k = j$
  ****
  Retourne-préfixe (T,k)
  // positionnement de la plus grande crêpe au sommet de la pile
  Retourne-préfixe (T,i)
  // positionnement de la plus grande crêpe à sa position i



LE CRÊPIER (PREUVE)

Crêpier-Itératif (T)
Données: Un tableau T d'éléments comparables
Résultats: Les éléments rangés dans le même tableau par ordre croissant

```
for i = Taille (T) to 2 do
    // {Les éléments de T d'indice plus grand que i sont triés et plus grands que les
       éléments d'indice de 1 à i}
    // Placer la bonne crêpe, la plus grande des i premières en position i
    k = 1; // indice de la plus grande crêpe
    for j = 2 to i
        // {L'élément T[k] est plus grand que les éléments d'indice 1 à j − 1.}
        // recherche de la plus grande crêpe
        if T[j] > T[k]
            k = j
        // {L'élément T[k] est plus grand que les éléments d'indice 1 à j.}
    Retourne-préfixe (T,k) // positionnement de la plus grande crêpe au sommet de la pile
    Retourne-préfixe (T,i) // positionnement de la plus grande crêpe à sa position i
    // {Les éléments de T d'indice plus grand que i − 1 sont triés et plus grands que les
       éléments d'indice de 1 à i − 1}
```

- Les éléments **{ ..... }** sont appelées des **assertions** qui portent sur les valeurs des variables et ont une valeur logique vrai/faux.
- Dans notre exemple on montre que si l'assertion est vraie en début d'itération, elle sera vérifiée en fin d'itération (invariant d'itération).
- Il reste à montrer que l'assertion est vérifiée au début de l'itération.
- En fin d'itération, l'assertion vraie prouve la correction du programme.

LE CRÊPIER (APPROCHE RÉCURSIVE)

Crêpier-Récursif (T,n)
Données: Un tableau T d'éléments comparables de taille n
Résultats: Les éléments rangés dans le même tableau par ordre croissant

if $n \neq 1$
  $k =$ **Indice-du-max** (T,n)
  `// recherche de l'indice de la valeur maximale du tableau de 1 à n`
  Retourne-préfixe (T,k)
  `// positionnement de la plus grande crêpe au sommet de la pile`
  Retourne-préfixe (T,i)
  `// positionnement de la plus grande crêpe à la position n`
  Crêpier-Récursif $(T,n-1)$
  `// appel récursif pour trier le tableau de 1 à n-1`

Expression plus élégante, plus facile à prouver par induction (par récurrence)

LE CRÊPIER : PETITE HISTOIRE

Quel est le nombre minimal $f(n)$ de retournements pour trier un tas arbitraire de n crêpes ?

Références

- Dweighter, Harry ; Garey, Michael R. ; Johnson, David S. ; Lin, Shen (1977), *Solutions of Elementary Problem E2569*, Amer. Math. Monthly 84 : 296
  Pseudonyme de Jacob E. Goodman
- Gates W.H. ; Papadimitriou, C.H. *Bounds for sorting by prefix reversal*. Discrete Math. 27 (1979), 47–57.
- Bulteau, L. ; Fertin, G. ; Rusu, I. *Pancake Flipping is Hard*. 37th International Symposium on Mathematical Foundations of Computer Science, Aug 2012, Bratislava, Slovakia. 7467, pp.247-258, 2012, Lecture Notes in Computer Science, Springer Verlag. Hal version

Valeurs de $f(n)$

- $f(n)$ est connu pour $n \leqslant 19$.
- Meilleur encadrement
$$\frac{15}{14}n \leqslant f(n) \leqslant \frac{18}{11}n + \mathcal{O}(1).$$
- le problème *MIN-SBPR* est $\mathcal{NP}$-dur
- Online Encyclopedia of Integer Sequences A058986

EXPRESSION DES ALGORITHMES

PRINCIPES POUR LA CONCEPTION D'ALGORITHMES

1. Déterminer les entrées et les sorties (spécification)
2. Trouver la structure de donnée adaptée pour ce problème
   - Ne pas hésiter à pré-traiter les entrées pour les mettre sous une forme adéquate
3. Essayer de réduire le problème à un problème connu
   - Tri, recherche, chemin dans un graphe,...
   - Vérifier si une solution existe déjà (livres, internet,...)
4. Décider de la manière d'approcher le problème : itératif/récursif/mix
   - Dépend de la manière de penser, de la facilité à trouver des invariants, de la manière de décomposer le problème en sous problèmes,...
5. **Écrire** l'algorithme
6. Faire tourner l'algorithme sur des exemples simples et des exemples "limite".
7. Donner les invariants de l'algorithme et faire la preuve
8. Évaluer la complexité de l'algorithme

PRINCIPES POUR L'EXPRESSION DES ALGORITHMES

Ma position personnelle (que certains de mes collègues ne partagent pas) :

- Un algorithme est toujours accompagné de schémas et de texte
  représentation de l'état des variables,
- utiliser les conventions
  (i,j sont des indices, x un réel, n un entier par exemple une taille de tableau,...).
- utiliser des identificateurs explicites, une variable un usage
  noms de variables, de fonctions,
- la forme est importante
  l'indentation doit permettre de comprendre la structure de l'algorithme
- mettre des commentaires dans le code et accompagner l'algorithme par une explication en français
- ne mettre que l'information importante : minimiser l'encre
- être cohérent
  utiliser le même formalisme pour toutes les présentations d'algorithmes

de manière plus générale

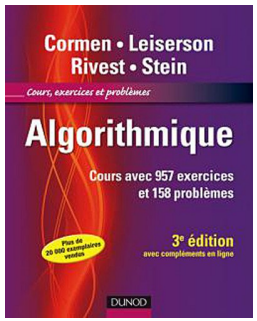
favoriser tout ce qui aide à la compréhension et éliminer ce qui peut perturber la lecture

EXPRESSION DES ALGORITHMES

BIBLIOGRAPHIE : OUVRAGES DE RÉFÉRENCE

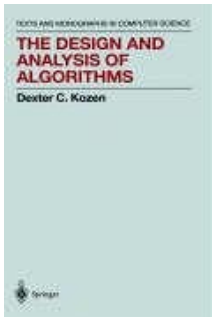
- **Algorithmique** *Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein.*. Dunod, 2010.
  Ouvrage de référence internationale en algorithmique. Très pédagogique il peut être utilisé en autoformation, lorsque les bases sont acquises. Couvre l'ensemble du domaine.
- **Algorithms** *Robert Sedgewick and Kevin Wayne*. Addison Wesley, 2011.
  Une approche thématique permettant de reprendre les différents et paradigmes de l'algorithmique. La présentation est soignée, les détails des implémentations en Java sont très utiles.

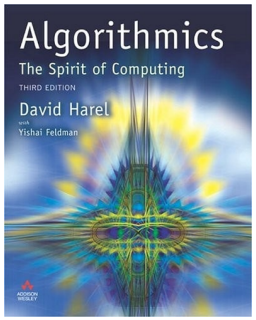
  Des versions précédentes en français : *Robert Sedgewick* Algorithmes en C ou Algorithmes en Java chez Dunod

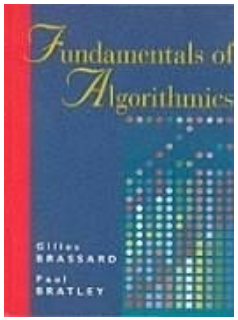


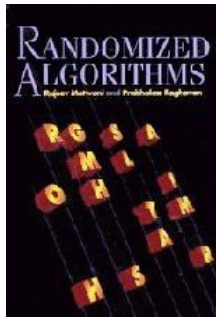
BIBLIOGRAPHIE : OUVRAGES PLUS AVANCÉS

- **The Design and Analysis of Algorithms** *Dexter C. Kozen* Springer, 1991.
  Excellent ouvrage pour de l'algorithmique avancée. Présenté sous forme de séquence de lectures "indépendantes" il va directement à l'essentiel. Les principes algorithmiques sont ainsi mis en valeur.
- **Algorithmics : The Spirit of Computing** *David Harel and Yishai Feldman* Addison Wesley, 2004.
  Orienté méthodologie, cet ouvrage propose une vue transversale en abordant successivement, méthode et analyse, limitations et robustesse, extensibilité... intéressant pour le recul pris.
- **Introduction à l'analyse des algorithmes** *Robert Sedgewick and Philippe Flajolet* Addison Wesley 1995
  Ouvrage théorique sur l'analyse de la complexité des algorithmes
- **Fundamental of Algorithms** *Gilles Brassard and Paul Bratley* Prentice Hall 1996
- **Randomized Algorithms**, *R. Motwani and P. Raghavan*, Cambridge University Press, 1995.



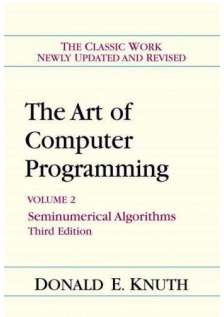


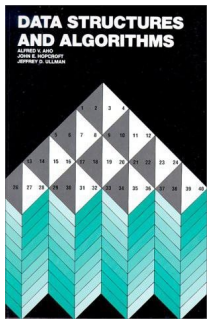


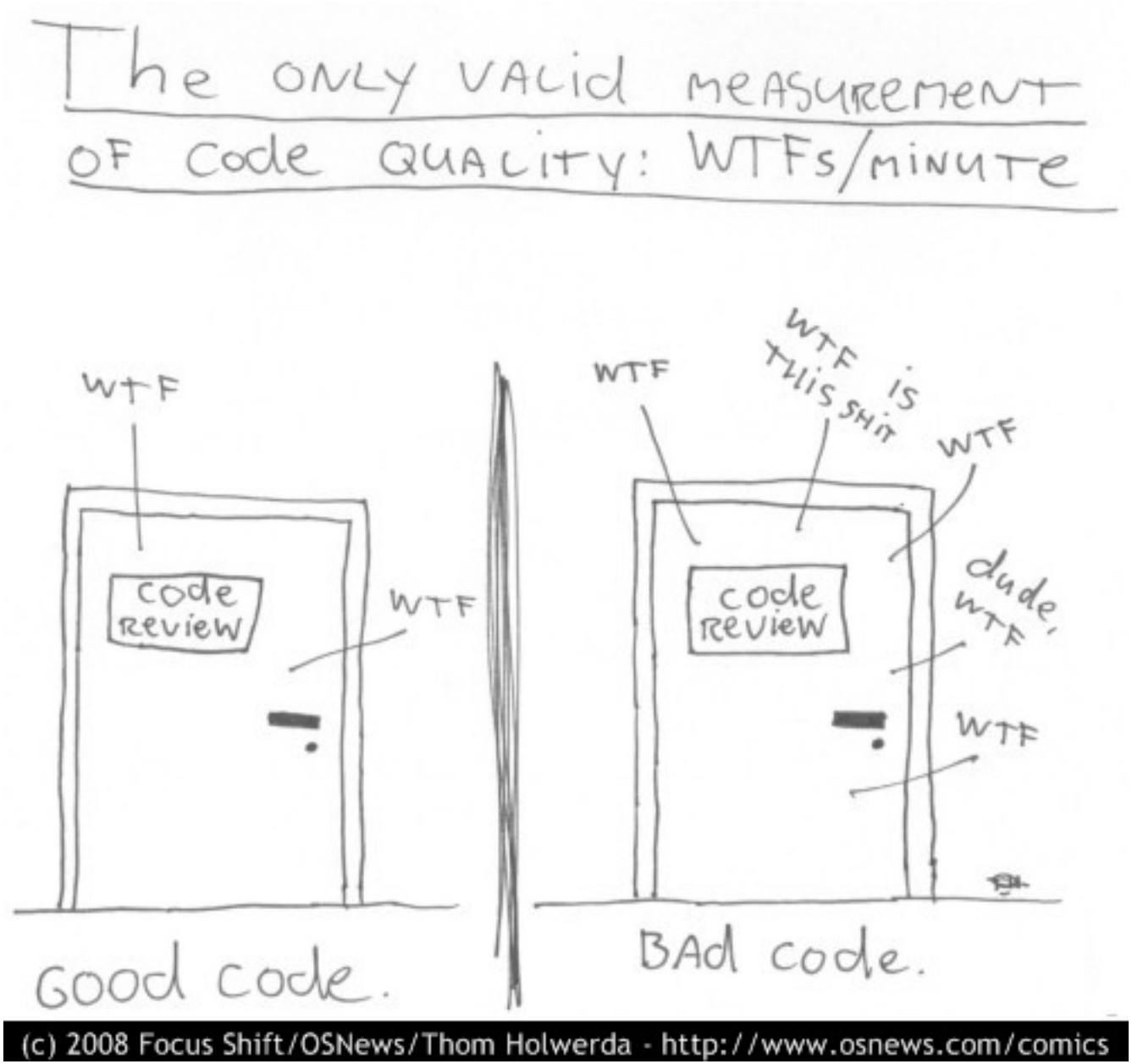


BIBLIOGRAPHIE : OUVRAGES HISTORIQUES DE RÉFÉRENCE

- **The Art of Computer Programming, Vol 1-4** *Donald E. Knuth*, Addison-Wesley, 1998.
  Ouvrage historique et encore d'actualité pour la conception et l'analyse d'algorithmes
- **Data Structures and Algorithms** *Alfred V. Aho, J.E. Hopcroft, et Jeffrey D. Ullman* Addison Wesley 1983
- *Jean-Luc Chabert et al.* **Histoires d'algorithmes** Belin 2010
  Une histoire des algorithmes avec un point de vue calcul et calcul numérique






The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF IS THIS SHIT
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics