Algorithmique et programmation

Année 2008–2009

Matmeca

9 mars 2009

Modalités

- 6 cours.
- 6 TD.
- 6 TP.
- Note finale : 2/3 Examen + 1/3 TP.

Bibliographie


A. Aho, J.E. Hopcroft, J. D. Ullman.
Structures de données et algorithmes.
InterÉditions, 1989.


D. Beauquier, J. Berstel, Ph. Chrétienne.
Éléments d'algorithmique.
Masson, 1992.
Épuisé, mais disponible en bibliothèque. Version électronique (gratuite) :
`http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.html`


Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein.
Introduction à l'algorithmique.
Dunod, 2002.

Plan du cours

1. Algorithmes et programmes : généralités
2. Complexité
3. Appels de fonction & Récursivité
4. Les tris
5. Structures linéaires
6. Arbres binaires de recherche

1 – Algorithmes et programmes : généralités

Généralités sur les algorithmes

Mohamed *Al-Khwarizmi*,
Perse, vers 780-850

- Un algorithme est une méthode systématique, une recette, pour résoudre un problème donné.
- Il se compose d'un suite d'opérations simples à effectuer pour résoudre le problème. Cette méthode peut donc être appliquée par une machine, un ordinateur.
- L'algorithme doit donner une réponse après un nombre fini d'opérations.

Description des algorithmes

- Un algorithme peut être décrit en utilisant un ensemble réduit d'instructions.
  - une séquence d'instructions comprenant :
  - l'affectation de valeurs à des variables : $x \leftarrow$ valeur,
  - des instructions conditionnelles Si C Alors S FinSi.
  - des instructions conditionnelles Si C Alors S sinon T FinSi.
  - des instructions répétitives TantQue C Faire S FinTantQue.
  - des instructions répétitives Répéter S Jusqu'à C FinRepeter.
  - des instructions itératives Pour Tout x dans L Faire S FinPourTout.
  - Une instruction de Retour.

Exemple d'algorithmes

Problème : Étant donné un entier $x \geqslant 2$, est-ce que x est premier ?

Algorithme `Prem1` :

1. tester, pour tous les entiers $i \in [2, x-1]$, si i ne divise pas x.
2. si oui, répondre OUI, sinon répondre NON.

- On exprime un algorithme en langage naturel. L'objectif est que l'algorithme soit compréhensible par un humain, pour l'appliquer ou le prouver par exemple.
- La suite des opérations doit être suffisamment claire et explicite pour que la méthode soit systématique.

Programme

- L'ordinateur ne peut exécuter qu'un programme. Il ne comprend pas un algorithme.
- Un programme est un texte écrit dans un langage obéissant à des règles très précises, traduisant les opérations simples de l'algorithme en instructions.
- Il existe de nombreux langages : C, C++, Java, Lisp, Caml, Prolog, Smart Talk, Python...
- Dans cette UE on utilisera Fortran (en TD/TP).

Programme Fortran correspondant à l'algorithme `Prem1`

```
LOGICAL FUNCTION  Premier(n)
    IMPLICIT  NONE
    INTEGER :: n, i
    Premier = .TRUE.
    do i=2, n-1
       if (Mod(n,i) == 0) then
          Premier = .FALSE.
       endif
    enddo
  end FUNCTION Premier
```


- Il y a plusieurs programmes pour le même algorithme.
- Les opérations dictées par l'algorithme sont les mêmes, leur traduction dans le langage est différente.
- On parle d'implantation d'un algorithme.
- Un autre programme pour le même algorithme :

```
LOGICAL FUNCTION  Premier(n)
    IMPLICIT  NONE
    INTEGER :: n, i
    Premier = .TRUE.
    do i=n-1, 2, -1  ! On teste les diviseurs en descendant.
       if (Mod(n,i) == 0) then
          Premier = .FALSE.
       endif
    enddo
  end FUNCTION Premier
```


Qu'est-ce qu'un problème ?

C'est la description d'une entrée et d'une sortie. On utilise parfois les termes *donnée* (ou *instance*), et *résultat*.

PREMIER :

Entrée : un entier $x \geqslant 2$

Sortie : OUI, si x est premier, NON sinon

CONNEXE :

Entrée : un graphe G

Sortie : OUI, si G est connexe, NON sinon

Exemples de problèmes

VALEUR_POLYNÔME

Entrée : un polynôme $P(X) = a_0 + a_1X + \cdots + a_pX^p$ donné par ses coefficients entiers $a_0, \ldots, a_p$, et une valeur entière x_0.

Sortie : la valeur $P(x_0)$.

PGCD

Entrée : deux entiers positifs x et y.

Sortie : la valeur du plus grand diviseur commun de x et y.

COLORATION

Entrée : un graphe fini.

Sortie : une coloration du graphe avec le minimum de couleurs.

Algorithmique et programmation
└Algorithmes et programmes : généralités

Exemples de problèmes

POSITION_DANS_PERMUTATION

Entrée : une permutation σ des entiers entre 1 et n.

Sortie : la position de l'entier 1.

INTERSECTION

Entrée : deux suites finies d'entiers.

Sortie : OUI si les deux suites ont un élément commun, NON sinon.

INTERSECTION_BIS

Entrée : deux suites finies d'entiers compris entre 0 et 20.

Sortie : OUI si les deux suites ont un élément commun, NON sinon.

Algorithmique et programmation
└Algorithmes et programmes : généralités

- On a vu plusieurs programmes implémentant un même algorithme.
- Il y a aussi plusieurs algorithmes pour un même problème.

Algorithme `Prem1bis` :

1. pour tous les entiers $i \in [2, \sqrt{x}]$
2. si i divise x, alors répondre NON.
3. si tous les tests précédents ont échoué, répondre OUI.

Algorithmique et programmation
└Algorithmes et programmes : généralités

Des questions naturelles

- Quelles sont les propriétés souhaitées pour un algorithme ?
  - correct, et
  - efficace.
- Comment s'assurer qu'un algorithme est correct ?
  On verra que c'est une question difficile. On verra une méthode permettant, dans certain cas, de la résoudre.
- Lorsqu'on a plusieurs algorithmes pour résoudre un problème, comment choisir entre l'un ou l'autre ?
  On peut comparer l'efficacité des algorithmes sur différentes instances de grande taille.
- Comment évaluer l'efficacité d'un algorithme ?

Représentation des données

Pour évaluer l'efficacité d'un algorithme, on a besoin de spécifier la taille des entrées et des sorties, et donc leur représentation.

ADD :

Entrée : deux entiers naturels x et y

Sortie : l'entier $x + y$

Un algorithme qui résoud le problème ADD sera différent si x et y sont écrits en unaire, en binaire, en écriture décimale ou en chiffres romains !

Par défaut, il s'agira toujours d'une représentation en binaire, celle utilisée par l'ordinateur.

Taille des entrées

ADD2 :

Entrée : deux entiers x et y écrit en binaire

Sortie : l'entier $x + y$ écrit en binaire

La taille d'une entrée pour un problème donné est la longueur de sa représentation. [unité=*bit*=*binary digit*, poids des fichiers de données].

Pour ADD2 c'est le nombre total de chiffres binaires pour écrire x et y.

La taille est généralement exprimée en bits, mais pas toujours ! Parfois c'est un nombre de sommets ou d'arêtes par exemple si l'on suppose que les opérations sur les sommets ou les arêtes sont élémentaires.

Base 2

- Les entiers sont représentés par une suite de chiffres (digit).
- Par exemple en base 10 : $180 = 1 \times 10^2 + 8 \times 10^1 + 0 \times 10^0$.
- En base 2, on utilise le même principe.
  - $5 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ est représenté par 101.
  - $180 = 2^7 + 2^5 + 2^4 + 2^2$ est représenté par 10110100.

180 :

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	1	0	1	0	0

- En base 1 : 180 représenté par $\underbrace{111\cdots111}_{180 \text{ fois}}$.
- La taille d'un entier est le nombre de chiffres qui le compose.

Comparaisons des bases

- En informatique, on utilise souvent la représentation en base 2 : c'est la façon dont les entiers sont représentés en machine.
- Un entier $x > 0$ est représenté
  - en base 2 avec $\lceil \log_2(x+1) \rceil$ bits.
  - en base 10 avec $\lceil \log_{10}(x+1) \rceil$ bits.
- Ces deux quantités sont proportionnelles.
- En base 1 par contre, il faut x chiffres, ce qui est exponentiellement plus élevé.

Qu'est-ce qu'une opération simple (1/2) ?

- Malheureusement, dans un algorithme, cela dépend du niveau d'abstraction auquel on se situe, et des objets que l'on manipule.
- Parfois il s'agit d'opérations arithmétiques ($+, -, *, /, \%$) sur des entiers, d'opérations ensemblistes ($\cup, \cap, \setminus$), d'opérations sur des graphes...
- Par contre, quand on s'intéresse à l'algorithme d'addition de 2 entiers, (opération pourtant plus simple que la division), on ne considère pas que l'addition est « simple » : on se place à un niveau plus réaliste.

Algorithme PGCD1 et PGCD2 (d'Euclide)

Algorithme PGCD1(x, y)

1. pour tout entier d allant de $\min(x, y)$ à 1 par pas de -1 :
2. si d divise x et si d divise y
3. retourner d

Algorithme PGCD2(x, y) (d'Euclide)

1. $r \leftarrow$ reste de la division de y par x
2. Tant que $r \neq 0$, faire :
3. $y \leftarrow x$
4. $x \leftarrow r$
5. $r \leftarrow$ reste de la division de y par x
6. retourner x.

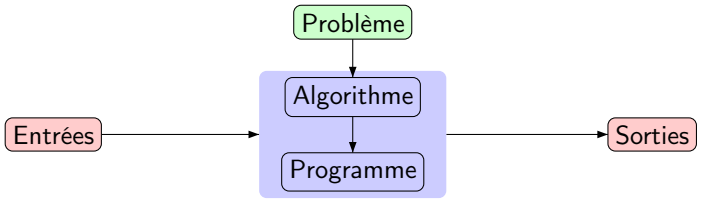
Qu'est-ce qu'une opération simple (2/2) ?

- Algorithme `Est-Connexe`(G) :
  - Tester s'il existe une chaîne entre toute paire de sommets de G
  - Si oui, répondre OUI, sinon NON
- Ici l'opération simple sous-entendue est une fonction `chaîne(G,s,t)` qui répond `True` s'il existe une chaîne entre s et t dans G, et `False` sinon.
- Dans tous les cas il doit s'agir d'opérations que l'on peut décomposer en une suite d'opérations **élémentaires**, "comprises" et exécutables par l'ordinateur.
- Dans l'exemple précédent, `chaîne(G,s,t)` peut utiliser une fonction `est_voisin(G,s,t)`.

Instructions élémentaires

- L'ordinateur ne peut exécuter que des instructions dites élémentaires, qui prennent un temps constant, indépendant des opérandes.
- Ex : addition, multiplication ou comparaison ($<, >, ==$) de variables de type `int`.
- A un niveau d'abstraction usuel, on considère donc que ces opérations arithmétiques prennent un temps constant. En toute rigueur, ce n'est pas exact, si on calcule sur des entiers arbitrairement grands par exemple.

Résumé



Taille des entrées

Opération simple

Instruction élémentaire

Notion de complexité

- **Rappel.** Il existe plusieurs algorithmes pour résoudre un même problème.
- Pour le problème PREMIER par exemple, encore un algorithme :
  Algorithme `Prem2` :
  - tester si x est pair
  - tester si tous les entiers **premiers** $\leqslant \sqrt{x}$ ne divisent pas x.
- Certains sont plus faciles que d'autres à programmer ...
- Certains sont plus efficaces que d'autres ...

2 – Complexité

Complexité : définition

- La complexité d'un algorithme A résolvant le problème P est le nombre maximum $C(n)$ d'instructions élémentaires utilisées par A pour résoudre P sur une entrée de taille n.

$$c(x) = \text{Nombre d'instructions élémentaires de } A \text{ sur l'entrée } x.$$
$$C(n) = \max_{x \text{ de taille } n} c(x).$$

- On peut définir de manière similaire la complexité d'un programme.
- On parle aussi de complexité dans le **pire des cas**.
- $C(n)$ est donc une fonction de n.
- Souvent on s'intéresse à $C(n)$ lorsque n est grand. On parle de complexité asymptotique.

Complexité : exemple

- Algorithme Trouver_1 pour POSITION_DANS_PERMUTATION :
  - Pour chaque i entre 1 et n.
  - si $\sigma(i)$ vaut 1, retourner i.
- On compte le nombre de comparaisons de $\sigma(i)$ à 1.
- On considère que la taille de la permutation est n (ce qui n'est pas tout à fait exact).
- Complexité dans le cas le pire : n comparaisons.
- On peut aussi définir la complexité dans le meilleur cas, ici 1. Elle est peu utilisée en raison de son manque d'intérêt.
- On peut aussi définir la complexité en moyenne, en se donnant une distribution sur les permutations.

Recherche de la valeur 1 dans une permutation : complexité en moyenne

- On suppose une distribution de probabilité uniforme sur l'ensemble $\mathfrak{S}_n$ des permutations de $\{1, \ldots, n\}$.
- Comme il y a $n!$ permutations, le coût est

$$\frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} c(\sigma)$$

- Le coût pour σ est $c(\sigma) = k$ si 1 apparaît en position k dans σ.
- Il y a $(n-1)!$ permutations telles que $\sigma(k) = 1$.
- La complexité en moyenne Trouver_1 est donc

$$\frac{1}{n!} \sum_{k=1}^{n} k \times (n-1)! = \frac{1}{n} \sum_{k=1}^{n} k = \frac{n+1}{2}.$$

Incrémentation en binaire

INCRÉMENTER

Entrée : un entier $x = x_{n-1} \cdots x_0$, de n bits (x_i est soit 0 soit 1).

Sortie : l'entier $x+1$ si $x < 2^n - 1$, l'entier 0 si $x = 2^n - 1$.

Algorithme Incrémenter

- $k \leftarrow 0$
- Tant que $x_k = 1$ et $k \neq n$ faire
- $\quad x_k \leftarrow 0$
- $\quad k \leftarrow k+1$
- Si $k \neq n$, alors faire $x_k \leftarrow 1$

Complexité de l'incrémentation (1/3)

- On compte le nombre $c(x)$ d'affectations à un x_k.
- Dans le cas le pire, on trouve n, si $x = x_{n-1} \underbrace{111 \cdots 111}_{n-1 \text{ fois}}$.
  La retenue va propager le calcul jusqu'à x_{n-1}
- Que vaut $c(x)$ en moyenne, en supposant les entrées équiprobables ?
- On remarque que l'écriture en base 2 d'un nombre pair finit par 0.
- Donc, quand x représente un nombre pair, le coût $c(x)$ vaut 1 (seul le dernier 0 est changé en 1 et l'algorithme s'arrête).

Complexité de l'incrémentation (2/3)

- Soit $E(n)$ l'ensemble des entrées de taille n.
- Par exemple $E(2) = \{00, 01, 10, 11\}$.
- On note $|E(n)|$ le nombre d'éléments de $E(n)$. On a $|E(n)| = 2^n$.
- Par définition, le nombre moyen d'affectations à x_k pour $x \in E(n)$ est

$$C_{\text{moy}}(n) = \frac{1}{|E(n)|} \sum_{x \in E(n)} c(x).$$

- On compte le nombre d'entrées pour lesquelles $c(x) = k$.
- $c(x) = k$ pour les entrées de la forme $x = x_{n-1} \cdots x_k \underbrace{0111 \cdots 111}_{k-1 \text{ fois}}$.

  (sauf pour $k = n$, ou cela arrive pour 2 entrées).

Complexité de l'incrémentation (3/3)

- Pour $1 \leqslant k < n$, il y a 2^{n-k} entrées x telles que $c(x) = k$.
  ... sauf pour $k = n$ où il y a 2 telles entrées. Donc

$$\begin{aligned} C_{\text{moy}}(n) &= \frac{1}{|E(n)|} \sum_{x \in E(n)} c(x) \\ &= \frac{1}{2^n} \sum_{x \in E(n)} c(x) \\ &= \frac{1}{2^n} \left[2n + \sum_{k=1}^{n-1} k \cdot 2^{n-k} \right] \end{aligned}$$

- On calcule facilement $\left[2n + \sum_{k=1}^{n-1} k \cdot 2^{n-k} \right] = 2^{n+1} - 2$.
- On a donc

$$C_{\text{moy}}(n) = \frac{2^{n+1} - 2}{2^n} = 2 - \frac{1}{2^{n-1}}$$

- La complexité en moyenne de INCRÉMENTATION est inférieure à 2 !

Complexité d'un problème

- Dans certains cas, on parle de complexité d'un problème P pour parler de la plus petite complexité possible d'un algorithme résolvant P.
- Déterminer la complexité d'un problème, ou même son ordre de grandeur, est en général très difficile : il faut trouver le meilleur algorithme possible !

Complexité inconnue (1/3)

- La multiplication de deux entiers de n bits ?
- L'algorithme que l'on apprend à l'école a une complexité proportionnelle à n^2.
- Le meilleur algorithme date de 2007, et a une complexité de

$$(n \log n) \cdot 2^{\log^* n}$$

où $\log^* n$ est le nombre de fois que l'on doit successivement appliquer la fonction $\log_2$ pour obtenir un nombre $\leqslant 1$. Si

$$n = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^2}}}}}}$$

alors $\log^* n$ vaut le nombre de "2". Cette fonction a une croissance extrêmement faible.

Complexité inconnue (2/3)

- Peut-on colorier les sommets d'un graphe G en 3 couleurs ?
  (sans avoir deux voisins adjacents de la même couleurs)
- Savoir si ce problème a une complexité polynomiale fait partie des 7 questions mathématiques sélectionnées par le CLAY Mathematical Institute pour leur difficulté.
- La question est ouverte depuis 1971 et vaut 1 million $.

Complexité inconnue (3/3)

- Est-ce qu'un graphe G possède un triangle ?
  (3 sommets x, y, z deux à deux voisins)

Le meilleur algorithme a une complexité proportionnelle à $n^{2,376}$ où n est le nombre de sommets de G, l'algorithme naïf ayant une complexité proportionnelle à n^3 :

Algorithme `Triangle-Naïf`(G) :

1. tester pour tous les triplets $\{x, y, z\}$ de sommets si x, y, z forment un triangle
2. si oui, OUI, sinon NON

Complexité : linéaire, exponentielle, *etc.*

- L'algorithme A a une complexité *linéaire* si, lorsque n est grand, $C(n)$ est proportionnelle à n. Mathématiquement, il faut qu'il existe une constante $a \neq 0$ telle que :

$$\limsup_{n \to +\infty} C(n)/n \;=\; a$$

- Terminologie (a, b sont des constantes, $a > 0$ et $b > 1$) : Pour une fonction $f(n)$ telle que $\limsup_{n \to +\infty} C(n)/f(n) = a$:

$f(n)$	type de complexité
n	linéaire
n^2	quadratique
b^n	exponentielle
$\log_b n$	logarithmique
n^b	polynomiale

- Les fonctions telles que $\log_b n$ ou $\sqrt[b]{n}$ sont **sous-linéaires** en n.

Temps d'exécution

Comme une instruction élémentaire prend un temps constant (disons a secondes), indépendant des opérandes et donc de la taille des entrées, la complexité peut donner une idée du temps d'exécution du programme T qui s'exécuterait sur une entrée de taille n.

$$\text{temps}(T, n) \;\leqslant\; a \cdot C(n)$$

Complexité de `Prem1`

La complexité de `Prem1` est proportionnelle à x.

1. tester, pour tous les entiers $i \in [2, x-1]$, si i ne divise pas x.
2. si oui, répondre OUI, sinon répondre NON.

La complexité de `Prem1` n'est pas linéaire ! Car la taille de x n'est pas x mais $n = \lceil \log_2(x+1) \rceil$, qui représente la longueur de l'écriture binaire d'un entier $x > 0$.

$$x = 2^{\log_2 x} \approx 2^n$$

(en négligeant la partie entière supérieure $\lceil \cdot \rceil$ et le $+1$)

La complexité de `Prem1` est donc **exponentielle !**

Complexité de PREMIER

En 2004, il a été trouvé un autre algorithme résolvant PREMIER de complexité $a \cdot (\log_2 x)^6$, soit $a \cdot n^6$ pour un nombre de n bits. La complexité de PREMIER est donc **polynomiale**.

Différence entre $2^{n/2}$ et n^6

La différence est énorme ! Si l'on suppose qu'un ordinateur peut exécuter une instruction élémentaire en 10^{-9} s (1 GHz), soit 1 milliard d'instructions par seconde, alors :

n (bits)	temps pour $2^{n/2}$		temps pour n^6	
32	$2^{32/2} \times 10^{-9} \approx$	65 μs	$32^6 \times 10^{-9} \approx$	1 s
64	$2^{64/2} \times 10^{-9} \approx$	4 s	$64^6 \times 10^{-9} \approx$	1 min 8 s
128	$2^{128/2} \times 10^{-9} \approx$	**584 ans**	$128^6 \times 10^{-9} \approx$	**1 h 13**
256	$2^{256/2} \times 10^{-9} \approx$	10^{11} ma.	$256^6 \times 10^{-9} \approx$	3,5 j
512	$2^{512/2} \times 10^{-9} \approx$	10^{52} ma.	$512^6 \times 10^{-9} \approx$	6,8 mois
1024	$2^{1024/2} \times 10^{-9} \approx$	10^{128} ma.	$1024^6 \times 10^{-9} \approx$	36 ans

1 μs = 1 millionième de seconde
1 ma. = 1 milliard d'années
512 bits = 320 chiffres décimals

- Dans la pratique, on cherche bien sûr des algorithmes de complexité la plus faible possible.
- Les complexités exponentielles sont donc à éviter.
- Parfois, ce n'est pas possible ! Voici un problème de complexité doublement exponentielle, où tout algorithme a une complexité $C(n) \geqslant 10^{10^n}$.
  Entrée : un entier x (écrit en base 10).
  Sortie : tous les entiers de x chiffres.
- y a 10^x entiers de x chiffres, donc la complexité est au moins 10^x. Or, la taille de l'entrée est ici de $n \approx \log_{10} x$. Autrement dit $x \approx 10^n$.
- La complexité de ce problème est au moins $10^x = 10^{10^n}$.

Problèmes très difficiles

- Contrairement à ce qu'on pourrait croire, il existe des problèmes qui ne possèdent pas d'algorithme pour les résoudre, quelque soit leur complexité !
- On parle de problème **indécidable**.

Exemple de problèmes indécidables (1/2)

HALTE :

Entrée : un programme Fortran (fichier de caractères)

Sortie : OUI si le programme s'arrête au bout d'un moment, NON sinon (s'il boucle)

Notez qu'à chaque programme Fortran, une réponse existe belle et bien : c'est OUI ou NON. Il n'y a simplement pas de méthode systématique pour **calculer** la réponse. Une machine ne pourra jamais résoudre ce problème.

Exemple de problèmes indécidables (2/2)

DIOPHANTIEN [Hilbert, 1900] :

Entrée : une équation polynomiale à coefficients entiers.

Sortie : OUI s'il existe une solution en entiers > 0, NON sinon.

Ex :

$x^2 + y^2 = z^2$ (OUI, $3^2 + 4^2 = 5^2$)

$x^5 + y^5 = z^5$ (NON, Théorème de Fermat-Wiles)

$x^2y + 2xyw - y^3w^4 = 3z^5$ (???)

Théorème de Matiyasevich (1970)

Il n'existe aucun algorithme répondant au problème DIOPHANTIEN.

Notation grand $O(\cdot)$

- Pour simplifier l'expression de certaines complexités, on utilise la notation $O(f(n))$ où $f(n)$ est une fonction de n.
- Par exemple si la complexité de l'algorithme `A` est $C_A(n) = 3n - 2$ et celle de l'algorithme `B` est $C_B(n) = \frac{1}{2}n^2 + \sqrt{n} - 1$, alors on notera $C_A(n) = O(n)$ et $C_B(n) = O(n^2)$.

  C'est une façon de dire que la complexité de `A` est linéaire et celle de `B` quadratique.

- Dans la notation $O(\cdot)$ on ne retient que le terme dominant (lorsque $n \to +\infty$) et on supprime le facteur multiplicatif.
- On parle d'ordre de grandeur et de complexité asymptotique.
- L'idée est qu'il ne sert à rien de calculer la complexité d'un algorithme à l'unité près. Par exemple, sur une machine à 1 GHz et pour $n = 10^9$, l'exécution de n ou $n+1$ instructions élémentaires prendrons 1 s dans les deux cas, la différence étant en pratique indiscernable !

On supprime le facteur multiplicatif, car il est courant qu'une suite de plusieurs instructions élémentaires sur une machine A puissent être regroupées en une seule sur une machine B plus puissante.

Ex : calcul de "A=B+4"

```
Z80:    LD A,4      ; charger 4 dans l'accumulateur
        ADD B       ; A=A+B

i86:    ADD A,B,4   ; A=B+4
```

Sur les machines récentes (processeurs Intel DualCore), on exécute plusieurs instructions élémentaires en parallèle ... Cela fait donc n ou $\frac{1}{2}n$ instructions élémentaires suivant la machine.

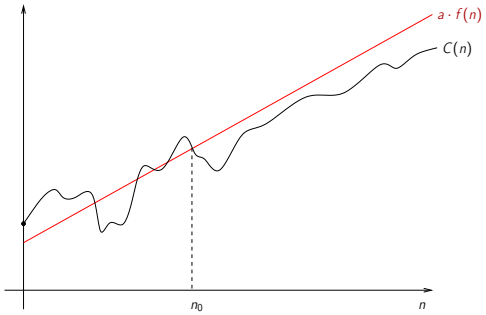
La complexité d'un algorithme est une mesure inhérente à l'algorithme (la méthode), elle ne doit pas dépendre de la machine sur laquelle on l'implante.

Définition formelle

Soient C et f deux fonctions $\mathbb{N} \to \mathbb{R}^+$.

On note "$C(n) = O(f(n))$", et l'on dit "$C(n)$ est en grand-O de n", pour dire qu'il existe $n_0 \in \mathbb{N}$ et $a \in \mathbb{R}^*$ tels que

$$\forall\, n \geqslant n_0, \quad C(n) \leqslant a \cdot f(n)\ .$$



Attention !

$C(n) = O(f(n))$ est une **notation** !

- Si $C_1(n) = O(n)$ et $C_2(n) = O(n)$, alors il est faux de dire que $C_1(n) = C_2(n)$ ou encore que $C_1(n) - C_2(n) = 0$!
- Mais il est correct d'écrire : $C_1(n) + C_2(n) = O(n)$.
- Dans certains ouvrages on note (plus justement ?) "$C(n) \in O(f(n))$" au lieu de $C(n) = O(f(n))$.

Complexité constante

- La notation $C(n) = O(1)$ signifie (par définition) qu'il existe $n_0 \in \mathbb{N}$ et une constante $a > 0$ telles que $C(n) \leqslant a$ $(= a \cdot 1)$ pour tout $n \geqslant n_0$
- On dit que la fonction C est bornée par une constante, sous-entendu lorsque n est assez grand.

Calcul précis de complexité

On utilise aussi la notation $O(\cdot)$ car dans la pratique il est souvent difficile de calculer exactement la complexité d'un algorithme (ou d'un programme), c'est-à-dire le nombre d'instructions élémentaires.

```
do i=1, n
    s=s+i
enddo
```

Combien d'instructions élémentaires ?
n ? $5n - 1$?
$\Rightarrow$ La complexité de cette boucle est $O(n)$.

Retenons qu'une instruction élémentaire est une instruction qui s'exécute en un temps (constant) indépendant de la taille de ses opérandes. Donc `s=s+i` peut être considérée comme une instruction élémentaire.

Notations $O(\cdot)$, $\Omega(\cdot)$ et $\theta(\cdot)$

Soient $f, g : \mathbb{N} \to R_+$ à valeurs positives.

- $f = O(g)$ s'il existe $a > 0$, $n_0 \in \mathbb{N}$ tels que

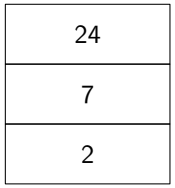
$$\forall n, \quad n \geqslant n_0 \Longrightarrow f(n) \leqslant a \cdot g(n).$$

- $f = \Omega(g)$ si $g = O(f)$.
- $f = \theta(g)$ si $f = O(g)$ et $g = O(f)$.

3 – Appels de fonction & Récursivité

Appels de fonction : pile d'exécution

- Pour gérer les appels de fonction pendant l'exécution d'un programme, et en particulier les appels récursifs, on utilise une pile.
- La pile sert en particulier à mémoriser les variables des fonctions appelées. Les piles seront vues plus tard en détail.
- Une pile est une zone contiguë de la mémoire pouvant contenir des valeurs (par exemple entières) :

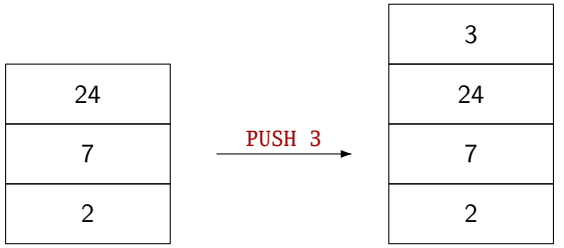


Pile contenant les valeurs 2, 7, 24

- Les deux opérations principales sur une pile sont l'ajout d'une valeur (qui fait grandir la pile) et la suppression de la valeur la plus haute (qui fait diminuer la pile).

Empilement

- L'ajout d'une valeur à la pile se fait toujours en haut de pile.
- L'ajout de `x` se note `PUSH x`, ou `EMPILER x`.

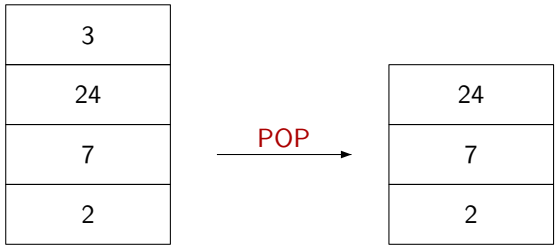


Pile contenant les valeurs 2, 7, 24

Après `PUSH 3` : pile contenant les valeurs 2, 7, 24, 3

Dépilement

- La suppression d'une valeur à la pile se fait toujours en haut de pile.
- La suppression se note `POP`, ou `DÉPILER` et permet de récupérer la valeur qui a été supprimée.



Pile contenant les valeurs 2, 7, 24, 3

Après `POP` : pile contenant les valeurs 2, 7, 24

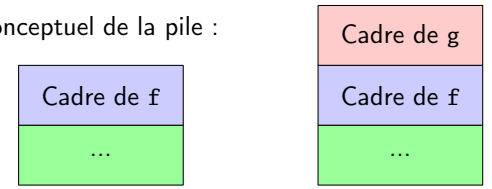
`POP` « retourne » la valeur 3

Appel de fonction : fonctionnement

```
Algorithme f():
    var x ← 3, y ← 6
    z ← g(x,y)
    retourner z
```

```
Algorithme g(x1, x2):
    y ← 15
    retourner y - x1 * x2
```

- Schéma conceptuel de la pile :



Avant appel de g

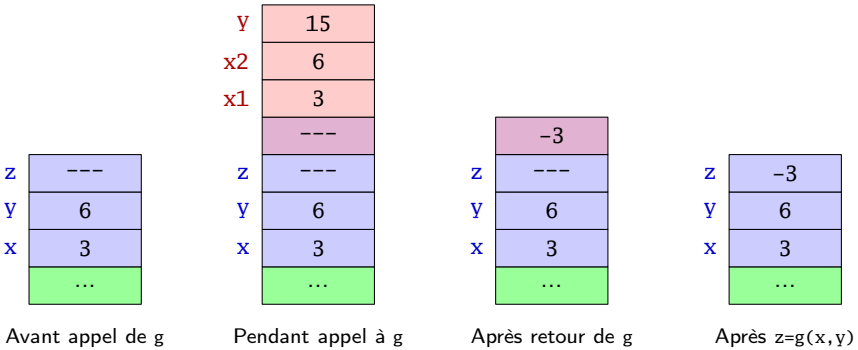
Pendant l'appel à g

- Un cadre d'appel de fonction permet de mémoriser
  - les valeurs des arguments et variables locales.
  - la valeur retour transmise à la fonction appelante.

Appel de fonction : fonctionnement

```
Algorithme f():
    x ← 3, y ← 6
    z ← g(x,y)
    retourner z
```

```
Algorithme g(x1, x2):
    y ← 15
    retourner y - x1 * x2
```



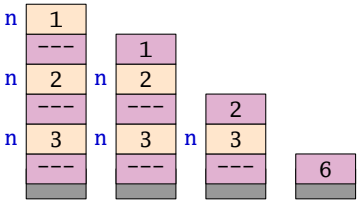
Avant appel de g | Pendant appel à g | Après retour de g | Après z=g(x,y)

Récursivité

- On peut, dans une définition de fonction `f`, appeler la fonction `f`.
- Les règles d'appel de fonction s'appliquent normalement.
- Bien adapté aux définitions de fonctions récursives.
- Exemple :
  - $0! = 1! = 1$,
  - $n! = n \times (n-1)!$ pour $n \geqslant 1$.

```
Algorithme fact(n):
    Si (n ⩽ 1)
        Retourner 1
    Retourner n * fact(n-1)

fact(3)
```



Récursivité

Pour définir une fonction récursive,

- on a un ordre $\leqslant$ sur les entrées d'un problème.
  - Bien fondé : il n'y a pas de suite infinie strictement décroissante.
  - Il y a un nombre fini d'éléments minimaux pour $\leqslant$.

Sous ces hypothèses, pour écrire un algorithme, on peut

- décrire son comportement sur les éléments $\leqslant$-minimaux,
- spécifier son comportement sur les autres entrées en fonction de son comportement sur des entrées strictement plus petites.

Appel de fonction : récapitulatif

1. Un algorithme récursif s'appelle lui-même sur des entrées « plus petites ».
   $\rightsquigarrow$ bien adapté pour des définitions mathématiques récursives.
2. Ne pas oublier de traiter les « cas de base ».
3. Pour tout algorithme récursif, il existe un algorithme non récursif effectuant le même calcul.

   Il suffit pour cela de savoir gérer « manuellement » la pile d'exécution.

   On verra dans la suite que l'on peut effectivement le faire.

Exemple 2 : algorithme d'Euclide

Algorithme PGCD3(x, y) (d'Euclide, version récursive)

{ Suppose $x, y > 0$ }
$r \leftarrow$ reste de la division de y par x
Si $r = 0$ alors
  retourner x.
FinSi
retourner PGCD3(r, x)

Comment montrer que l'algorithme

- va bien s'arrêter ? Utiliser un ordre adapté sur les entrées.
- calcule bien le résultat attendu ?
- Au passage : montrer que la complexité est $O(\log(\max(x, y)))$.

Récursivité : efficacité

- Exemple : calcul de 2^x pour $x \geqslant 0$.

```
Algorithme pow2(x: entier)
   Si (x ⩽ 0) alors
      Retourner 1
   Retourner pow2(x-1) + pow2(x-1)
```

- Quelle est la complexité de ce calcul de 2^x ?
- Comptons le nombre $A(x)$ d'opérations '+' effectuées par `pow2(x)`
- $A(0) = 0$, et $A(x) = 2 \cdot A(x-1) + 1$.
- Donc $A(x) = 2^x - 1$ [Ce n'est pas étonnant : on a calculé 2^x uniquement avec des additions et des 1].
- Comme la taille de la donnée est $n = \lceil \log_2(x+1) \rceil$, on a une complexité $C_1(n) = O(A(x)) = O(2^{2^n})$.

Récursivité : efficacité

```
Algorithme pow2bis(x: entier)
   Si (x ⩽ 0)
      Retourner 1
   Retourner 2 * pow2bis(x-1)
```

- Quelle est la complexité de ce calcul de 2^x ?
- Comptons le nombre $M(x)$ d'opérations '$*$' effectuées
- $M(0) = 0$, et $M(x) = M(x-1) + 1$.
- Donc $M(x) = x$, d'où une complexité $C_2(n) = O(M(x)) = O(2^n)$
  On a gagné une exponentielle, parce qu'on ne fait qu'1 appel récursif au lieu de 2.

Récursivité : efficacité

- On peut encore améliorer cette complexité.
- On utilise le fait que $2^x = (2^{x/2})^2 \cdot 2^{x\,\%\,2}$.

```
Algorithme1 pow2ter(x):
   Si (x ⩽ 0)
      Retourner 1
   y = pow2ter(x/2)
   Si (x modulo 2 == 0)
     Retourner y * y
   Sinon:
     Retourner y * y * 2
```

- La complexité $B(x)$, en nombre de '$*$' est telle que $B(0) = 0$ et $B(x) \leqslant B(x/2) + 2$.
- On en déduit par induction que la complexité est $C_3(n) = O(B(x)) = O(n)$.

Questions

- Peut-on faire mieux (asymptotiquement) que la complexité précédente pour calculer 2^x ?
- Comment adapter la technique pour le calcul du $x^{\text{ème}}$ nombre de Fibonacci ? Pour toute récurrence linéaire à 2 termes ?
  - Par exemple, pour Fibonacci, l'algorithme naïf

```
Algorithme Fib(x: entier)
   Si (x ⩽ 1) alors
      Retourner 1
   Retourner Fib(x-1) + Fib(x-2)
```

  a une complexité $\approx \phi^x$, soit double exponentielle, où $\phi = \frac{1+\sqrt{5}}{2}$.
  - Peut-on trouver une version en $O(x)$? En $O(\log x)$?
- Donner une version non récursive de la fonction `pow2ter`.

Coût et récurrences

- L'analyse de coûts produit souvent des récurrences sur la complexité de la forme

$$C(n_0) = d$$
$$C(n) = aC(n/b) + f(n)$$

- où t/n est interprété comme $\lfloor t/n \rfloor$ ou $\lceil t/n \rceil$.
- Elles apparaissent lorsqu'on subdivise un problème en a sous-problèmes de taille b fois plus petite.
- La fonction f mesure le coût de recomposition des solutions des sous-problèmes.

Récurrences de partitions

- Soit $C : \mathbb{N} \to \mathbb{R}_+$ croissante à partir d'un certain rang, telle qu'il existe des entiers $n_0 \geqslant 0$, $b \geqslant 2$ et $k \in \mathbb{R}_+$, $a, c \in \mathbb{R}_+^*$ tels que

$$C(n_0) > 0, \text{ et}$$
$$C(n) = aC(n/b) + cn^k \text{ si } n/n_0 \text{ est une puissance de } b$$

Alors

$$C(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta(n^k \log_b n) & \text{si } a = b^k \\ \theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Récurrences de partitions

- Remarque : l'analyse précédente ne couvre pas tous les cas.
- On peut par exemple avoir $C(n) = aC(n/b) + cn^k \log(n)^\ell$. On peut alors affiner le résultat : lorsque $= ab^k$, il faut distinguer les cas $\ell > -1$, $\ell = -1$ et $\ell < -1$.
- Souvent, on ne dispose que d'une inégalité

$$C(n) \leqslant aC(n/b) + cn^k$$

- On en déduit que la fonction de complexité est majorée par l'une des fonctions obtenues précédemment.

4 – Les tris

Algorithmes de tri

Le problème TRI est le suivant

Entrée : une liste $L = T[1], \ldots, T[n]$ d'éléments d'un ensemble totalement ordonné.

Sortie : Une permutation $T[\sigma(1)], \ldots, T[\sigma(n)]$ de L telle que

$$\forall i < n, \quad T[\sigma(i)] < T[\sigma(i+1)].$$

- Les éléments de la liste sont parfois appelés clés.
- On suppose les clés accessibles en mémoire (tri interne). On a un accès direct au i-ème élélment (coût unitaire).
- Tri par comparaison : Pour produire la sortie voulue, un algorithme de tri par comparaison n'effectue que des comparaisons d'éléments $<, \leqslant \ldots$
- On peut évaluer la complexité d'un tel algorithme de tri en fonction du nombre de comparaisons effectuées.

Tris par comparaison : borne inférieure

Théorème. Tout algorithme de tri par comparaisons effectue dans le cas le pire au moins $\Omega(n \log_2 n)$ comparaisons sur une liste de longueur n.

- Soit A un algorithme de tri par comparaisons.
- On représente l'ensemble des comportements de A sur les entrées de n clés par un arbre de décision, binaire.
- Si toutes les clés sont distinctes, il y a $n!$ permutations que l'algorithme doit distinguer : l'arbre a donc $n!$ feuilles.
- Le nombre de comparaisons dans le cas le pire est la longueur maximale d'une branche.
- Un arbre binaire ayant $n!$ feuilles a une branche de longueur $\log_2(n!) = \Omega(n \log_2 n - n) = \Omega(n \log_2 n)$.

Tris linéaires

- Dans le cas où les éléments à trier sont des entiers $\leqslant K$ où K est connu et $\leqslant n$, on peut trier en temps $O(n + K) = O(n)$.
- Une des technique consiste :
  - à allouer un tableau `C` de K éléments remplis par $0 \rightsquigarrow O(K)$.
  - à parcourir le tableau `T` à trier. Lorsqu'on rencontre l'élément `i`, on incrémente `C[i]` $\rightsquigarrow O(n)$.
  - À la fin de ce parcours, chaque case `C[i]` contient le nombre d'éléments valant `i`.
  - Il suffit de parcourir le tableau `C` pour reconstruire la liste triée $\rightsquigarrow O(n)$.
- Ces techniques de tri ne sont pas des tris par comparaison.

Tri bulle

Le principe de ce tri est de parcourir le tableau plusieurs fois en échangeant les éléments consécutifs qui ne sont pas dans l'ordre.

```
Algorithme TriBulle (T[1..n] : tableau entiers)
var i,j : entier
Début
  Pour i ← 1 à n-1 faire
    Pour j ← n-1 à i par pas -1 faire
      Si T[j] > T[j+1] alors
        Echanger(t,j,j+1)
      FinSi
    FinPour
  FinPour
Fin
```


Tri bulle : correction

- Pour prouver que le tri bulle est correct, on utilise un invariant.
- Inv_k : Après l'itération numéro k de l'itération sur i, les éléments se trouvant dans T[1],...,T[k] sont correctement placés.
- On vérifie Inv_k inductivement.

Tri bulle : cas le pire et le meilleur

- Le nombre de comparaisons effectué par le tri bulle est facile à obtenir : il est indépendant de l'entrée.
- Le nombre d'échanges minimum et maximum sont également faciles à obtenir.
- Si toutes les clés sont distinctes, le nombre d'échanges est le nombre d'inversions de la permutation définie par T.
- Le nombre maximal d'inversions pour une permutation est $\frac{n(n-1)}{2}$.
- En supposant une distribution uniforme sur les permutations, on calcule la complexité en moyenne du tri bulle en nombre d'échanges : $n(n-1)/4$.

Tri par sélection

Le tri par sélection consiste à placer l'élément minimal en première position, puis à recommencer récursivement sur le reste de la liste.

```
Algorithme Tri-selection(T; i; j)
Début
  Si j > i alors
    Trouver k tel que T[k] = min(T[i],...,T[j])
    Echanger(T; k; i)
    Tri-selection(T; i + 1; j)
 FinSi
Fin
```

- Prouver la correction de l'algorithme.
- Calculer le nombre d'échanges effectués par l'algorithme, ainsi que le nombre minimum, maximum et moyen de comparaisons.

Tri par insertion

L'algorithme consiste à trier récursivement la liste sauf la dernière valeur `T[n]`, et à insérer celle-ci à la bonne place. Pour trier la liste T entre i et j (compris) :

```
Algorithme Tri-insertion(T; i; j)
Début
  Si j > i alors
    Tri-insertion(T; i; j - 1)
    k ← j
    Tant que k > i et T[k] < T[k - 1] faire
      Echanger(T; k - 1; k)
    FinTantQue
  FinSi
Fin
```

- Correction et nombre de comparaisons effectuées dans le cas le pire ?
- Nombre moyen de comparaisons (probabilité uniforme et clés distinctes) ?
  Important : montrer que si les permutations initiales sont équiprobables, il en est de même pour celles définies par les n-1 premières clés.

Tri fusion

- Le principe du tri par fusion est de trier récursivement les deux moitiés de L : $T[1..\lfloor n/2 \rfloor]$ et $T[\lceil n/2 \rceil..n]$, puis de fusionner les listes obtenues.
- Pour trier T entre les positions i et j :

```
Algorithme Tri-fusion(T; i; j)
Début
  Si i < j alors
    milieu ← i + (j - i + 1)/2 - 1
    Tri-fusion(T; i; milieu)
    Tri-fusion(T; milieu + 1; j)
    Fusionner(T; i; milieu; j)
  FinSi
Fin
```


Fusion de deux listes triées

- L'algorithme `Fusionner(T ; i ; milieu ; j)` suppose que les deux listes suivantes sont triées :
  - T[i],...,T[milieu]
  - T[milieu + 1],...,T[j]
- Utilise un tableau temporaire `Tmp` de taille n.
- Il construit une liste triée dont l'ensemble des éléments est T[1],...,T[j].
- Sa complexité est $O(j - i + 1)$.

Fusion de deux listes triées

```
Algorithme Fusionner(T; i; milieu; j)
Début
  Pour k ← i à milieu faire Tmp[k] ← T[k] FinPour
  Pour k ← milieu+1 à j faire Tmp[k] ← T[j+milieu+1-k] FinPour
  s ← i
  t ← j
  Pour k ← i à j faire
    Si Tmp[s] < Tmp[t] alors
      T[k] ← Tmp[s]; s ← s+1
    Sinon
      T[k] ← Tmp[t]; t ← t-1
    FinSi
  FinPour
Fin
```

Fusion de deux listes

- Exemple pour `i=1` et `j=8`, `milieu=4` et `T=[4,8,9,13,2,5,7,12]`.
- Les deux sous-listes `T[1..4]` et `T[5..8]` sont bien triées.
- Après les deux premières itérations Pour, `Tmp=[4,8,9,13,12,7,5,2]`. (Noter l'inversion dans la seconde partie qui rend la dernière itération plus simple).
- A l'issue de la dernière itération Pour, on a `T=[2,4,5,7,8,9,12,13]`
- La correction de `Fusionner` se prouve à nouveau à l'aide d'invariants.

Cas le pire et en moyenne

- L'algorithme a la même complexité sur toute liste de taille n.
- Le nombre de comparaisons de `Fusionner` est compris entre $\lfloor n/2 \rfloor$ et n.
- Le nombre de comparaisons $C(n)$ effectuées par le tri fusion sur une liste de taille n vérifie :

$$\lfloor n/2 \rfloor + C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) \leqslant C(n) \leqslant n + C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil).$$

- En résolvant cette récurrence de partition, on obtient une complexité en $\theta(n \log n)$.
- Cette complexité est donc asymptotiquement optimale.
- En pratique, il y a plus efficace.

Tri rapide

- Le principe du tri rapide est
  1. de placer une clé élément, appelé pivot, à sa place définitive,
  2. de placer à sa gauche les clés $\leqslant$, à sa droite les clés $>$.
  3. de trier récursivement les deux listes gauches et droite.
- On a le choix pour la valeur de la clé-pivot. On prend ici celle du premier élément du tableau.
- L'algorithme `Pivoter` place la clé-pivot, réarrange le tableau selon 2., et renvoie la position à laquelle elle a été placée.

Tri rapide : principe

Trie les éléments du tableau `T` placés entre les positions `i` et `j`.

```
Algorithme Tri-rapide(T; i; j)
Début
  Si i < j alors
    k ← Pivoter(T; i; j)
    (* Ici, T[m] ⩽ T[k] si et seulement si m ⩽ k  *)
    Tri-rapide(T; i; k - 1)
    Tri-rapide(T; k + 1; j)
  FinSi
Fin
```

Tri rapide : pivot

- L'algorithme pour trouver le pivot a plusieurs variations.
- On prendra comme valeur de pivot `v = T[i]`, le premier élément.
- Il faut mettre cette valeur à la bonne place, tous les éléments inférieurs à gauche, tous les éléments supérieurs à droite, et retourner la position `k` du pivot.
- Un couple d'indices (s, t) est inversé par rapport au pivot si
  - $i + 1 \leqslant s < t \leqslant j$
  - $T[s] > v$
  - $T[t] \leqslant v$.

Pour placer le pivot et réarranger le tableau,

- On recherche un couple inversé (s, t) avec s minimal et t maximal : l'algorithme `couple-inverse(T,g,d,v)` renvoie un tel couple avec $g \leqslant s < t \leqslant d$.
- Si un tel couple existe, on l'inverse et on recommence à partir de s et t.

```
Algorithme Pivoter(T; i; j)
  debut
  v ← T[i] ; s ← i + 1; t ← j
  Tant que s ⩽ t faire
    (s, t) ← couple-inverse(T; s; t; v)
    Si s < t alors
      (* couple inversé trouvé *)
      Echanger(T; s; t) ; s ← s + 1; t ← t - 1
    FinSi
  FinTantQue
  Pivoter ← t
  Echanger(T; i; t)
```


tri rapide : correction

- On montre qu'après k ← `Pivoter(T; i; j)`, on a

$$\forall m \in [i, j] : \quad i \leqslant m \leqslant k \iff T[m] \leqslant T[k].$$

- On montre la correction de l'algorithme par induction sur $j - i$.

Tri rapide : cas le pire

- L'algorithme `Pivoter(T; i; j)` effectue au plus $j - i + 2$ comparaisons de clés.
- Soit $c(n)$ le nombre maximal de comparaisons effectués par l'algorithme sur un tableau de taille n, sans compter les comparaisons faites récursivement.
- Si sur un tableau de taille n, l'algorithme se rappelle récursivement sur des tableaux de tailles n_1 et n_2, avec $n = n_1 + n_2 + 1$, on a $c(n) \leqslant c(n_1) + c(n_2)$.
- On en déduit que le cas le pire
  - est atteint pour une liste déjà triée.
  - que dans ce cas, $\theta(n^2)$ comparaisons sont effectuées.

Tri rapide : analyse en moyenne

La complexité en moyenne, supposant les clés distinctes et une distribution uniforme sur $\mathfrak{S}_n$, est plus difficile.

Lemme clé. Pour une position du pivot k fixée, si π_1, k, π_2 est la permutation obtenue après pivotage, les permutations $\pi_1 \in \mathfrak{S}_{k-1}$ et $\pi_2 \in \mathfrak{S}_{n-k}$ sont équiprobables.

Preuve On compte le cardinal l'ensemble des permutations qui donnent après pivotage la permutation $\pi = \pi_1(k)\pi_2$.

Le calcul donne

$$\sum_{i=0}^{\min(k-1,n-k)} C_{k-1}^i C_{n-k}^i.$$

Complexité du tri rapide

- La complexité moyenne, en nombre de comparaisons, vérifie donc

$$f(n) \leqslant 1/n \sum_{k=1}^{n} (\underbrace{n+1}_{Pivoter} + f(k-1) + f(n-k))$$

et

$$1/n \sum_{k=1}^{n} (\underbrace{n-1}_{Pivoter} + f(k-1) + f(n-k)) \leqslant f(n)$$

car les valeurs du pivot sont équiprobables.

- On en déduit une complexité en $\theta(n \log n)$, en résolvant ces récurrences.

En pratique

- Le tri rapide se comporte mieux que le tri fusion.
- En particulier, la mémoire nécessaire est faible.
- Si les données ne sont pas aléatoire, on peut essayer de trouver un pivot médian.
- Versions probabilistes.
- Exercice Montrer que dans le cas le meilleur, la complexité est $\Omega(n \log(n))$.

5 – Structures linéaires

Types abstraits

- Un type abstrait définit des objets aux caractéristiques communes, par
  - un modèle de ces objets,
  - des opérations qui sont applicables aux objets du type,
  - des propriétés (axiomes) que doivent vérifier ces opérations.
- On peut définir plusieurs types abstraits pour représenter un même type d'objet. Exemple : graphes.‘

Structures de données

- Une structure de donnée décrit la représentation machine d'un type abstrait.
- Cette représentation ne suppose pas un langage de programmation particulier.
- Parmi les structures de données classiques :
  - structures linéaires séquentielles : listes, piles, files.
  - structures arborescentes : arbres binaires, arbres, forêts.
  - structures relationnelles (graphes).
- Pour un même type abstrait, on a le choix entre plusieurs structures de données, qui peuvent différer en efficacité.

Listes : définition

- Une liste est une suite finie d'éléments.

$$(e_1, \ldots, e_n)$$

- n est la longueur de la liste.
- La liste peut être vide : $n = 0$. On la note `()`.
- Chaque élément a un rang : l'élément e_i a rang `i`.
- On dit que e_{i+1} est l'élément suivant e_i ($0 \leqslant i < n$).
- On dit que e_{i-1} est l'élément précédant e_i ($0 < i \leqslant n$).
- L'emplacement où se trouve un élément est une cellule.

Type abstrait liste

- `LISTE_VIDE` : $\longrightarrow$ Liste.
- `EST_VIDE` : Liste $\longrightarrow$ Booléen.
- `TÊTE` : Liste $\longrightarrow$ Cellule.
- `FIN` : Liste $\longrightarrow$ Liste.
- `CONTENU` : Cellule $\longrightarrow$ Élément.
- `SUCC` : Cellule $\longrightarrow$ Cellule.
- `CONS` : Élément $\times$ Liste $\longrightarrow$ Liste.

Par exemple, `CONS` prend un élément et une liste et retourne une liste.

Type abstrait liste

Des axiomes traduisent les propriétés des listes.

- FIN(CONS(e,L)) = L.
- CONTENU(TÊTE(CONS(e,L)))=e.
- SUCC(TÊTE(L)) = TÊTE(FIN(L)).

Si `L = [4,2,6,8,3,8]`, la tête de `L` est la cellule contenant l'élément `4`, et la fin de `L` est `[2,6,8,3,8]`.

Type abstrait liste

- On peut définir à partir de ce type abstrait une fonction de longueur, ou bien une fonction permettant d'accéder au *i*-ème élément d'une liste.
- On aurait pu définir le type abstrait en se basant sur une fonction d'accès et la longueur.
- Exercice Définir un autre type abstrait pour les listes, et exprimer les opérations d'un des types abstraits grâce aux opérations de l'autre.

Listes : structures de données (1)

- Plusieurs structures de données conviennent bien pour les listes.
- Dans un tableau : allocation continue en mémoire.
- La liste contient la position de la tête de liste.
- Chaque « cellule » est représentée par une ou plusieurs cases contiguës.
  - cellule suivante (sauf dernier élément).
  - éventuellement : cellule précédante (si double chaînage).
  - valeur de l'élément contenu dans la cellule.

Listes : structures de données (2)

- On peut également utiliser des pointeurs comme structure de donnée.
- La liste vaut `NIL` si elle est vide.
- Sinon, elle pointe sur une structure, représentant la 1ère cellule, qui contient :
  - l'élément en tête de liste,
  - un pointeur vers la cellule suivante.

Listes : structures de données

- Les structures de données ont des caractéristiques différentes.
- Cela concerne par exemple la complexité des opérations.
  - rechercher un élément dans une liste.
  - insérer un élément.
- Avec la forme tableau, on doit réallouer (étendre le tableau) s'il est plein.

Piles

Une pile est une liste séquentielle dans laquelle les ajouts et suppressions se font à la même « extrémité », appelée *haut de pile*.
LIFO : Last In, First Out.

- `CRÉER_PILE` : $\longrightarrow$ Pile.
- `EST_PILE_VIDE` : Pile $\longrightarrow$ Booléen.
- `VALEUR` : Pile $\longrightarrow$ Élément.
- `DÉPILER` : Pile $\longrightarrow$ Pile.
- `EMPILER` : Élément $\times$ Pile $\longrightarrow$ Pile.

où `VALEUR` et `DÉPILER` ne sont valides que sur pile non vide.

Piles : type abstrait

- `DEPILER`(`EMPILER`(P,e)) = P
- `VALEUR`(`EMPILER`(P,e)) = e
- `EST_PILE_VIDE`(`CRÉER_PILE`()) = Vrai
- `EST_PILE_VIDE`(`EMPILER`(P,e)) = Faux

Piles : utilisation

- Les piles interviennent souvent en informatique.
- On en a déjà vu pour gérer les appels de fonctions, leurs variables locales et arguments, et la transmission de valeurs appelant/appelé.
- Autres exemple typique : calculatrice.
  - notation Polonaise inverse.
  - notation habituelle.

Files

- Une file est une liste séquentielle dans laquelle on insère d'un côté et on supprime de l'autre côté
- FIFO : First In, First Out.
  - `CRÉER_FILE` : $\longrightarrow$ File.
  - `EST_FILE_VIDE` : File $\longrightarrow$ Booléen.
  - `VALEUR` : File $\longrightarrow$ Élément.
  - `DÉFILER` : File $\longrightarrow$ File.
  - `ENFILER` : Élément $\times$ File $\longrightarrow$ File.

  où `VALEUR` et `DÉFILER` ne sont valides que sur file non vide.
- Exercice Axiomes ?

Piles et files : structures de données

- On représente souvent les piles et les files dans un tableau.
- On gère le tableau de façon circulaire.
- Les éléments sont représentés de façon contiguë.
- Pour les files, on a besoin en plus de la taille (pourquoi ?).

Une application des listes : tri racine

- Le tri racine (*radix-sort*) est utilisé pour trier une liste de mots $u_1, \ldots, u_k$ dans l'ordre lexicographique.
- Sur un alphabet X, sa complexité est

$$O(|X| + \sum_{i=1}^{k} u_i).$$

6 – Arbres binaires de recherche

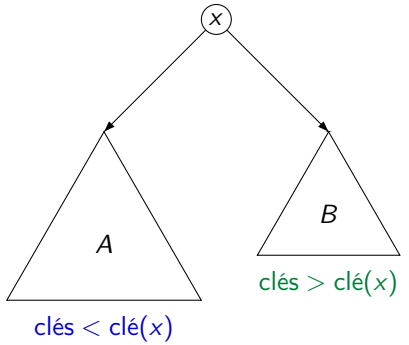
Arbres binaires

- Dans un arbre binaire, chaque sommet a 0, 1 ou 2 fils.
- Chaque sommet peut porter une information, appelée clé, élément d'un ensemble totalement ordonné.
- Un type abstrait (possible) :
  - EST_VIDE(a : arbre) : Booléen
  - RACINE(a : arbre) : sommet
  - CLÉ(s : sommet) : élément
  - SOUS_ARBRE_GAUCHE(a : arbre) : arbre
  - SOUS_ARBRE_DROIT(a : arbre) : arbre
  - ARBRE_VIDE() : arbre
  - CRÉER_ARBRE(x : élément ; g : arbre ; d : arbre) : arbre
- En utilisant ce type abstrait, on peut écrire des opérations pour récupérer la clé à la racine, changer la clé de la racine ou les sous-arbres gauches ou droits, tester si un sommet est une feuille, créer un arbre ne contenant qu'une feuille, etc.

Arbres binaires de recherche

- Soit $T = (x, A, B)$ un arbre binaire, de racine x, sous-arbre gauche A et sous arbre droit B.
- T est un arbre binaire de recherche (ABR) si, pour tous sommets y de A et z de B, on a

$$\text{cle}(y) < \text{cle}(x) < \text{cle}(z)$$



Arbres binaires de recherche : intérêt

- L'intérêt des ABR est que
  - les opérations de recherche, insertion et suppression de clés peuvent se faire en temps $O(h)$, où h est la hauteur de l'arbre...
  - ...et on peut faire en sorte que cette hauteur soit en $O(\log n)$, où n est le nombre de sommets dans l'arbre.
  - Dans la suite, on va donc voir comment travailler avec des arbres « compacts », de hauteur faible

Recherche dans un ABR

```
Algorithme Chercher(x : élément; A : arbre) : Booléen
Début
  Si EST_VIDE(A) alors
     Retourner Faux
  Sinon si x = CLÉ(RACINE(A)) alors
     Retourner Vrai
  Sinon si x < CLÉ(RACINE(A)) alors
     Retourner chercher(x, SOUS_ARBRE_GAUCHE(A))
  Sinon
     Retourner chercher(x, SOUS_ARBRE_DROIT(A))
  FinSi
Fin
```

La recherche est dans le cas le pire $O(h)$, où h est la hauteur de l'arbre.

Insertion dans un ABR

```
Algorithme insérer(x : élément ; A : arbre) ;
Début
  si EST_VIDE(A) alors
    A ← CRÉER_ARBRE(x, ARBRE_VIDE(), ARBRE_VIDE())
  Sinon si x < CLÉ(RACINE(A)) alors
    insérer(x, SOUS_ARBRE_GAUCHE(A))
  Sinon
    insérer(x, SOUS_ARBRE_DROIT(A))
Fin
```

Le coût est à nouveau en $O(h)$, où h est la hauteur de A.

Suppression dans un ABR

Pour supprimer un sommet dont la clé est donnée, en supposant ce sommet présent dans l'arbre :

- On navigue d'abord dans l'arbre jusqu'à trouver la clé.
- Lorsque le sommet à supprimer est trouvé,
  - s'il n'a qu'un fils, on le remplace par ce fils ;
  - s'il a deux fils, on échange sa clé avec celle la précédant, associée au sommet t « maximal » de son sous-arbre droit. On supprime ensuite le sommet t.

Arbres équilibrés

- Les opérations de recherche, d'insertion et de suppression ont une complexité, dans le cas le pire, en $O(h)$, où h est la hauteur de l'ABR.
- Sur une classe d'arbres « filiformes » (par exemple, si tous les fils gauche sont vides), on peut avoir $h = \Omega(n)$, où n est le nombre d'éléments. Dans ce cas, on ne gagne rien a priori à utiliser la structure d'ABR.
- Si au contraire l'arbre est « compact », il peut avoir $2^{h+1} - 1$ sommets, et $h = O(\log n)$. Dans ce cas, chacune des opérations va se faire en temps logarithmique.
- Soit $\mathcal{C}$ une classe d'arbres binaires. On dit que les arbres de $\mathcal{C}$ sont équilibrés s'il existe $k > 0$ tel que, pour tout arbre de $\mathcal{C}$ à n sommets et de hauteur h, on a
$$h \leqslant k.\log(n)$$
- Comment maintenir des ABR équilibrés ?

AVL

- Les AVL sont une classe d'arbres équilibrés, inventée par Adelson-Velskii et Landis (1962).
- Ce sont les arbres binaires ayant la propriété suivante :
$$\text{pour tout sommet x : } -1 \leqslant \delta(x) \leqslant 1.$$
où $\delta(x) = h(G(x)) - h(D(x))$ est la différence de hauteurs entre les sous-arbres gauche et droit.
- Exemple : arbres de Fibonacci.

Propriété d'équilibre

Pour tout AVL de hauteur h assez grand à n sommets, on a

$$h \leqslant 1,5 \log_2(n)$$

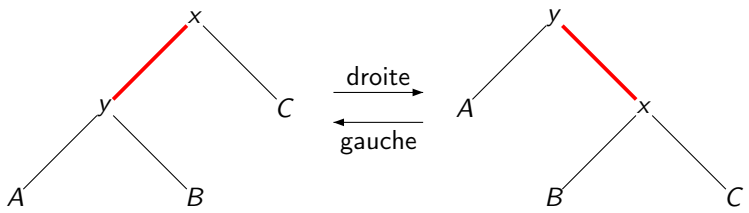
Preuve Calculer par induction le nombre minimal de sommets pour un arbre de hauteur fixée h. Ce nombre est atteint pour les arbres de Fibonacci.

AVL

- La propriété d'être un AVL peut être détruite par les opérations d'insertion et de suppression.
- Pour maintenir cette propriété après une insertion ou suppression, il faut transformer l'arbre.
- Pour être intéressantes, ces transformations doivent être faites rapidement, par exemple en $O(\log n)$.
- Il existe d'autres sortes d'arbres équilibrés (arbres rouge-noir, $a - b$ en particulier).

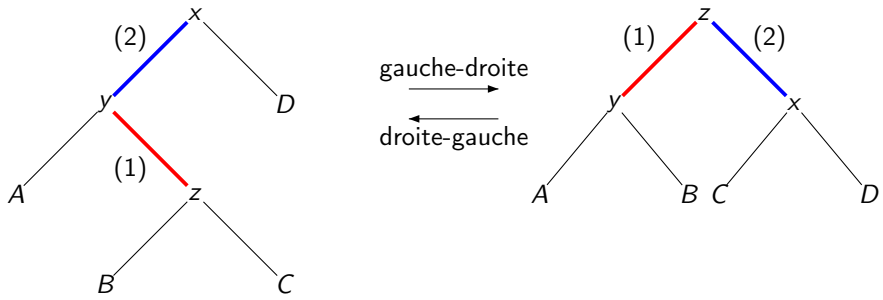
Rotations

- Les rotations sont des transformations locales des arbres.
- Elles peuvent s'implanter en temps constant et ne changent pas le fait que les clés restent triées dans l'ordre infixe.
- La figure suivante montre les rotations simples, où x et y sont des sommets et A, B, C des sous-arbres.



Rotations

- Les rotations doubles sont une suite de deux rotations simples.



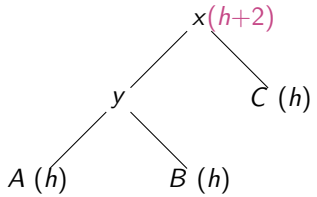
Exercice Montrer qu'on peut passer d'un arbre binaire de recherche à un autre sur le même ensemble de clés $\{0, 1, \ldots, n\}$ en temps $O(n)$.

Insertion

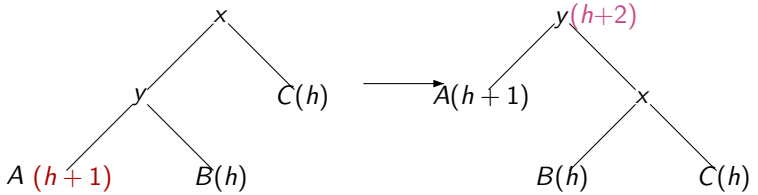
- On insère in sommet s.
- On considère sur le chemin entre s et la racine le sommet x le plus bas à être déséquilibré : $|\delta(x)| \geqslant 2$.
- On rééquilibre localement par une séquence de rotations.
- Au pire, 2 rotations sont nécessaires $\rightsquigarrow O(\log n)$.

Insertion : cas 1, insertion dans A

Avant



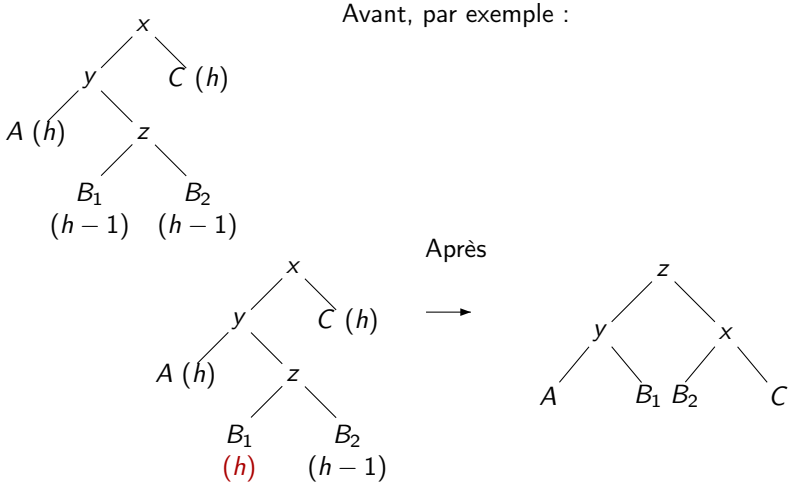
Après



On retrouve l'équilibre et la même hauteur qu'avant l'insertion.

Insertion : cas 2, insertion dans $B = (z, B_1, B_2)$

Avant, par exemple :



On retrouve l'équilibre et la même hauteur qu'avant l'insertion.

Suppression dans un AVL

- On a vu que l'insertion dans un AVL demande, en plus de l'insertion elle-même, 2 rotations additionnelles.
- $\Rightarrow$ complexité $O(\log n)$, où n est le nombre de sommets.
- Pour la suppression, le raisonnement est identique.
- Seule différence : le déséquilibre se propage, et l'on doit ainsi effectuer une suite de rotations jusqu'à la racine...
- ... mais $O(\log n)$ rotations fournissent la même complexité.

AVL : résumé

- L'insertion nécessite, pour maintenir la propriété AVL, au maximum 2 rotations (simples).
- La suppression nécessite au plus $O(\log n)$ rotations.
- La complexité d'une recherche, insertion ou suppression est donc $O(\log n)$.