Programmation fonctionnelle

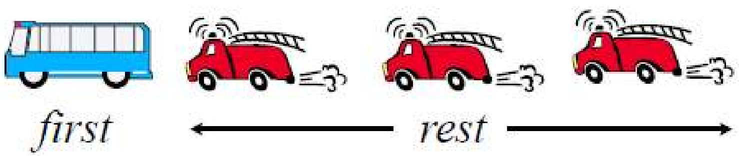
Dr. Mounir El Araki Tantaoui

Avec l’aimable autorisation du Professeur Jean Paul Roy

http://deptinfo.unice.fr/~roy/

Agenda

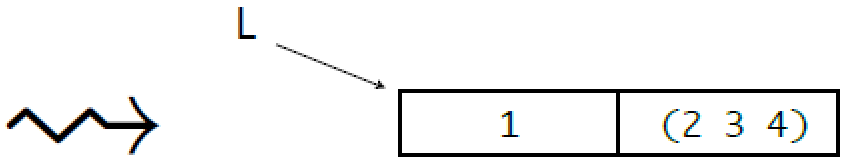
- Langage d'expressions préfixées
- Les fonctions
- Programmer avec des images / Animations
- Programmer par récurrence
- **Les listes chainées (Suite)**
- Les calculs itératifs
- Type abstraits et généralisation
- Les arbres binaires



Les listes "chaînées" de Scheme/Lisp

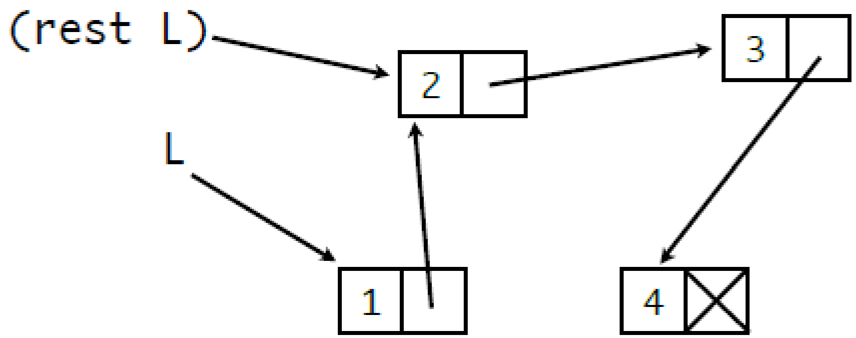
- Le mot **liste** recouvre deux structures de données distinctes suivant les langages de programmation. Les listes de Scheme (à la suite de Lisp) sont des **chaînages de doublets.**

```
> L
(1 2 3 4)
```

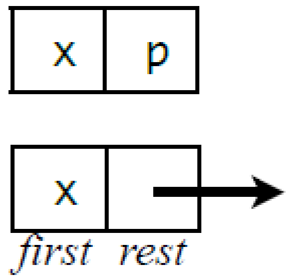


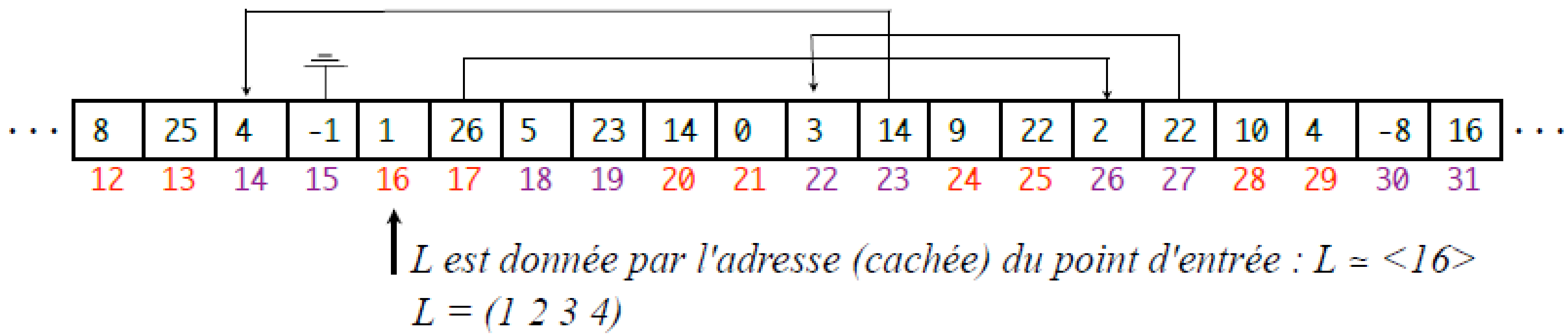
- RAPPEL : Une liste est *vide* ou bien est constituée :
  - d'un premier élément, accessible par la fonction **first**
  - et du reste de la liste, accessible par la fonction **rest**

```
> (rest L)
(2 3 4)
```



- Donc au fond, une liste non vide est un **couple** <x,p> formée du premier élément x et d'un *pointeur* p vers le reste de la liste. Ce pointeur représente une adresse mémoire. Un tel couple se nomme un **doublet**.
- Les doublets sont éparpillés dans la mémoire des listes. Chaque doublet connaît le doublet suivant (pas le précédent !)...





<u>DEFINITION</u> : Une liste est définie par récurrence :
- ou bien c'est la constante liste vide empty notée aussi ’()
- ou bien c'est un doublet dont le second élément (le reste) est une liste.

- Très bien, mais comment construire des doublets ?

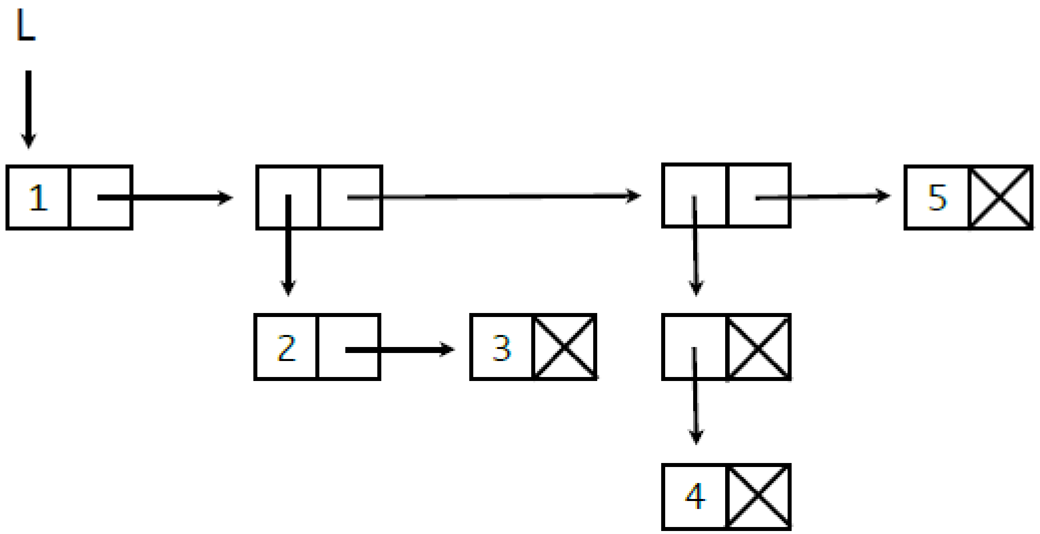
```
(define L (list 1 2 3 4))
                    ⇔
(define L (cons 1 (cons 2 (cons 3 (cons 4 empty)))))
(1 2 3 4) (2 3 4) (3 4) (4) ()
```

- L'unité d'occupation mémoire pour les listes est le doublet. La liste L contient 4 doublets (voir plus haut).
  - *The memory footprint of L is 4 pairs !*

- La complexité du tri par insertion était de $O(n^2)$ doublets. Il s'agit du nombre de doublets construits durant l'exécution du tri, et non pas le nombre de doublets du résultat. La plupart de ces doublets ne serviront à rien ensuite et seront recyclés automatiquement par le **Garbage Collector (GC).** Tous ces doublets rendus à la mémoire libre seront chaînés et placés dans une **liste libre**, dans laquelle cons va puiser !

- Les architectures de doublets peuvent être *ramifiées* : une liste peut contenir d'autres listes !

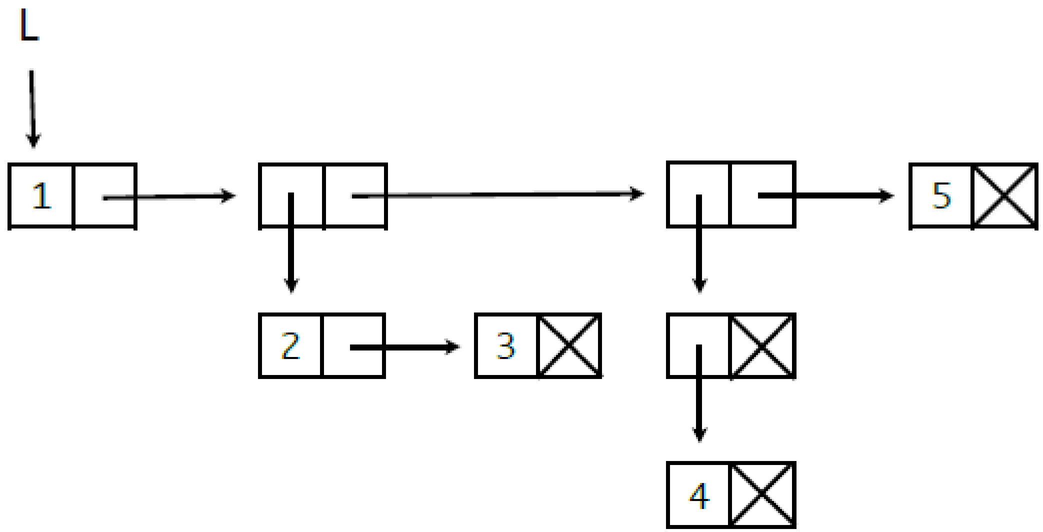
```
> (define L (list 1 (list 2 3) (list (list 4)) 5))
> L
(1 (2 3) ((4)) 5)
```



8 doublets
⇕
8 cons

```
(define L (cons 1 (cons (cons 2 (cons 3 empty))
                        (cons (cons (cons 4 empty) empty)
                              (cons 5 empty)))))
```

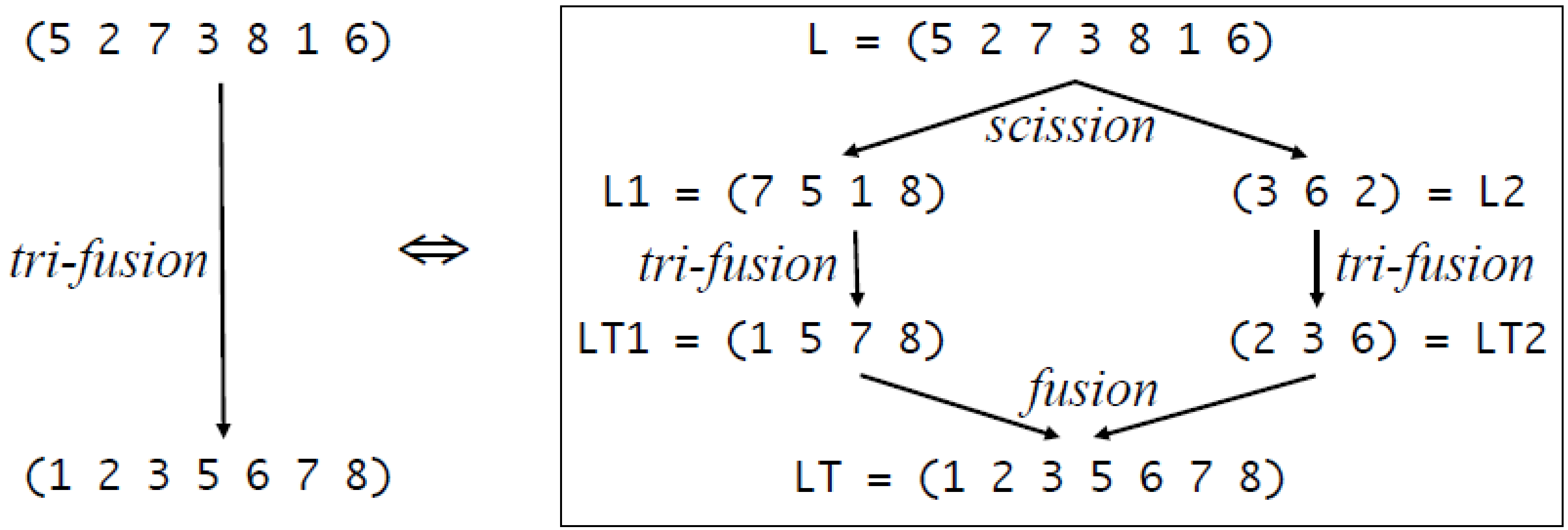
- Correspondance entre le dessin et la représentation parenthésée :
- une **flèche verticale** + une boîte ↝ une parenthèse ouvrante.
- un **élément dans le FIRST** ↝ on affiche le FIRST
- une **flèche horizontale** + une boîte ↝ un espace.
- une **croix dans un REST** ↝ on affiche une parenthèse fermante et on remonte



(1 ‿ (2 ‿ 3) ‿ ((4)) ‿ 5)

Le tri d'une liste par fusion

- Dans le cours 4 nous avions vu le *tri par insertion*, quadratique : $O(n^2)$.
- Nous allons étudier le **tri par fusion** d'une liste L, qui procède par **dichotomie**, et sera donc sans doute plus rapide :
  - je commence par scinder la liste L en deux sous-listes L1 et L2 de même longueur, ou presque. *Peu importe lesquelles...*
  - je trie par hypothèse de récurrence L1 et L2, pour obtenir LT1 et LT2.
  - je fusionne LT1 et LT2 en une seule liste triée.



- Au final, il y a trois fonctions à programmer :

```
(tri-fusion L)     ☞  LT
(scission L)       ☞  (LT1 LT2)
(fusion LT1 LT2)   ☞  LT
```

- Voici la scission. Le raisonnement - comme d'habitude - se fait par récurrence sur L. Je vais avancer de ***deux éléments à la fois*** à chaque étape, en plaçant le premier dans LT1 et le second dans LT2. S'il n'en reste qu'un, je le mettrai dans LT1 :
  - HR : Supposons que l'on connaisse (scission (rest (rest L))).

```
(define (scission L)
  (cond ((empty? L) (list empty empty))
        ((empty? (rest L)) (list L empty))
        (else (local [(define HR (scission (rest (rest L))))]
                (list (cons (first L) (first HR))
                      (cons (second L) (second HR)))))))
```

```
> (scission '(5 2 7 3 8 1 6))
((5 7 8 6) (2 3 1))
```

OK…

- En nombre d'appels à cons, le coût de (scission L) est clairement en O(n) puisqu'on parcourt linéairement la liste en demandant le même nombre de cons chaque fois.

```
(define (fusion LT1 LT2)    ; fusion triee de deux listes triees en croissant
  (cond ((empty? LT1) LT2)
        ((empty? LT2) LT1)
        ((<= (first LT1) (first LT2)) (cons (first LT1) (fusion (rest LT1) LT2)))
        (else (cons (first LT2) (fusion LT1 (rest LT2))))))
```

- La fusion se fait toujours en comparant les premiers de chaque liste, soit on insère le premier de la première liste dans la seconde liste, ou le premier de la seconde liste dans la première liste.

```scheme
(define (tri-fusion L)
  (if (or (empty? L) (empty? (rest L)))
      L
      (let ((S (scission L)))     ; S = (L1 L2)
        (fusion (tri-fusion (first S)) (tri-fusion (second S))))))
```

- Soit c_n le coût [en nombre d'appels à cons] de tri-fusion d'une liste de longueur n. L'algorithme récursif fournit l'équation :

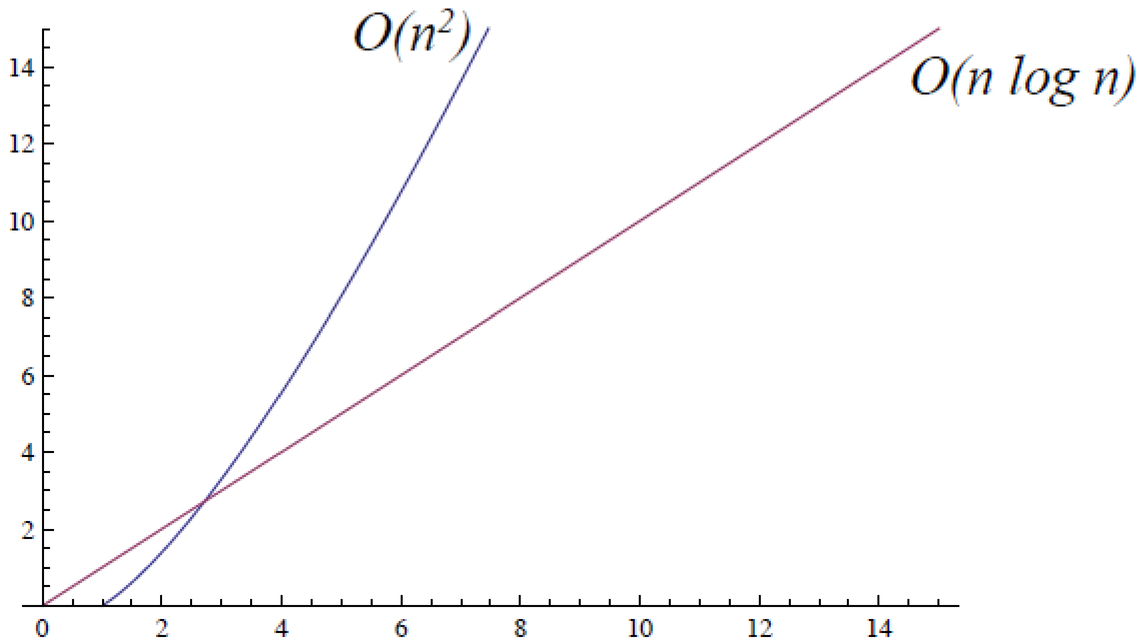
c_n = *<coût de scission>* + *<coût des HR>* + *<coût de fusion>*

$c_n = O(n) + 2\ c_{n/2} + O(n)$

$c_n = 2\ c_{n/2} + O(n)$

$c_n = 2\ c_{n/2} + n$ *pour simplifier !*

- Comment résoudre cette ***équation récursive*** $c_n = 2\ c_n/2 + n$?
- vous obtiendrez $c_n = O(n \log n)$.
- L'algorithme obtenu est quasi-linéaire, juste un peu moins efficace que O(n), mais vraiment très peu... Beaucoup plus rapide que $O(n^2)$!
- MORALE : La stratégie ***couper en deux*** est encore gagnante.



Un entier est-il premier ?

- n entier est **premier** si n ≥ 2 et si son plus petit (et seul) diviseur dans [2,n] est précisément n.
- Calculons donc (ppdiv n), le plus petit diviseur ≥ 2 de n.
- La récurrence brutale ne marche pas, je ne sais pas déduire (ppdiv n) à partir de (ppdiv (- n 1)). Je dois généraliser le problème.
- Cherchons le plus petit diviseur ≥ k de n, soit (ppdiv>= n k).

```
(define (ppdiv>= n k)
   ; le plus petit diviseur de n qui soit ≥ k
   .....)
```

- Alors je sais tester si un nombre est premier :

```
(define (premier? n)
  (and (>= n 2) (= (ppdiv>= n 2) n)))
```

```
> (premier? 1)
#f
> (premier? 2)
#t
> (premier? 2011)
#t
> (premier? 2013)
#f
```

- L'algorithme précédent s'effondre sur les grands entiers, mais on sait faire mieux avec plus de maths
  - On peut montrer facilement que la recherche peut être stoppée si $k^2>n$.

- Théorème d'Euclide : **il existe une infinité de nombres premiers.**

- ***PREUVE****. Supposons que cet énoncé soit faux. Il existerait alors un plus grand nombre premier N.*
- *Posons P = N! + 1. Puisque P > N, P ne peut pas être premier, il est donc divisible par un nombre premier q et nécessairement q ≤ N.*
- *Mais alors q serait à la fois un diviseur de N! et un diviseur de P = N! + 1, donc il diviserait leur différence 1, ce qui est impossible. L'hypothèse qu'il existe un plus grand nombre premier est donc absurde, CQED.*

- La distribution des nombres premiers reste mystérieuse. Ils se raréfient à l'infini (on peut montrer qu'il y a des trous aussi grands qu'on veut, par exemple 100000000000 entiers consécutifs tous non premiers). Jusqu'à n, il y en a environ *n/ln(n)*, démontré en 1896.
  - **Le plus grand nombre premier connu (janvier 2013) est $2^{57885161}-1$**

- **Liste de nombres premiers. Version 1**

- Calculons la liste (premiers n) des nombres premiers de [2, n]. Première tentative, récurrence brutale.

```
(define (premiers n)        ; les nombres premiers de [2,n]
  (if (<= n 1)
      empty
      (local [(define HR (premiers (- n 1)))]
        (if (premier? n) (cons n HR) HR))))
```

```
> (premiers 80)
(79 73 71 67 61 59 53 47 43 41 37 31 29 23 19 17 13 11 7 5 3 2)
> (length (premiers 10000))        ; time = 8.5 sec
1229
> (round (/ 10000 (log 10000)))
#i1086.0
```

- La liste est rendue à l'envers. On l'inverse. Mais attention !!!

- • Mais attention à inverser seulement à la fin !

```
(define (premiers n)                          ; dans l'ordre !
  (local [(define (aux n)                     ; produit la liste à l'envers
            (if (<= n 1)
                empty
                (local [(define HR (aux (- n 1)))]
                  (if (premier? n) (cons n HR) HR))))]
    (reverse (aux n))))
                                        ajout à gauche
```

```
> (premiers 80)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79)
```

- Et surtout résister à la tentation d'ajouter n à droite de HR par :
- (if (premier? n) (append HR (list n)) HR)
- car le coût serait O(1) + O(2) + ... + O(n) = O(n^2), complexité
- catastrophique ! On payerait 1000000 au lieu de 1000.

- Bon, c'est bien joli, mais peut-on obtenir la liste en ordre croissant **sans** payer reverse à la fin ? Réfléchissons :
- On obtient un mauvais ordre en descendant de n à n-1.
- Idée : suffirait-il de monter au lieu de descendre ?...
- Je ne peux pas passer de n à n+1, je filerais vers l'infini !
- Mais je peux prendre une seconde variable k, comme (ppdiv>= n k). Une variable qui va monter ! Cela revient encore à généraliser le problème :

Calculons la liste (premiers>= n k) des nombres premiers de [k, n].

```
(define (premiers>= n k)    ; les nombres premiers de [k,n]
  (cond ((> k n) empty)
        ((premier? k) (cons k (premiers>= n (+ k 1))))
        (else (premiers>= n (+ k 1)))))
```

```
(define (premiers n)    ; les nombres premiers de [2,n]
  (premiers>= n 2))     ; et zou ! On l'obtient dans l'ordre...
```

- **Liste de nombres premiers Version 2**

- Par le **CRIBLE D'ERATOSTHENE**.

1. On part de la liste des entiers de [2,n] :

   L = (2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ... n)

2. Le premier élément candidat est premier, je le garde et je supprime ses multiples. Je passe à l'élément suivant et continue ainsi jusqu'à ce que tous les éléments aient été traités.

- (2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 ...)
- (<u>2</u> 3 5 7 9 11 13 15 17 19 21 23 25)
- (<u>2</u> <u>3</u> 5 7 11 13 17 19 23 25)
- (<u>2</u> <u>3</u> <u>5</u> 7 11 13 17 19 23)
- (<u>2</u> <u>3</u> <u>5</u> <u>7</u> 11 13 17 19 23)
- *etc.*ers. Version 2

- **La liste des facteurs premiers de n**

- J'aimerais calculer la liste **(factor n)** de tous les facteurs premiers de n, mais la récurrence brutale ne fonctionne pas.
- Je vais calculer la liste **(factor>= n p)** de tous les facteurs premiers de n qui sont ≥ p.
- Or nous savons calculer le plus petit facteur premier de n qui est ≥ k avec (ppdiv>= n k). Nous allons donc nous promener sur les facteurs premiers de n et les purger de n au fur et à mesure :

- *plus petit facteur premier ≥ 2 de 6760 : **2**. Je divise par 2.*
- *plus petit facteur premier ≥ 2 de 3380 : **2**. Je divise par 2.*
- *plus petit facteur premier ≥ 2 de 1690 : **2**. Je divise par 2.*
- *plus petit facteur premier ≥ 2 de 845 : **5**. Je divise par 5.*
- *plus petit facteur premier ≥ 5 de 169 : **13**. Je divise par 13.*
- *plus petit facteur premier ≥ 13 de 13 : **13**. Je divise par 13.*
- *n = 1. STOP*

$$6760 = 2^3 \times 5 \times 13^2$$