Notions de Bases

Bases d'Algorithmique

Ce document a pour objectif de vous familiariser avec l'algorithmique. Les tous premiers algorithmes que vous allez exécuter et écrire sont issus de connaissances communes vues lors de vos cours de l'enseignement primaire ou secondaire.
Pour exécuter les algorithmes en mode dit *pas à pas*, pensez à toujours avoir une feuille de brouillon et un stylo pour pouvoir noter le déroulé de l'algorithme à chaque instruction ou série d'instructions.

Définition informelle d'un algorithme [1] : « *procédure de calcul bien définie qui prend en entrée une valeur, ou un ensemble de valeurs, et qui donne en sortie une valeur, ou un ensemble de valeurs. Un algorithme est donc une séquence d'étapes de calcul qui transforment l'entrée en sortie* ».

1 Problèmes, Solutions, et Types de données

Les algorithmes sont donc des étapes successives permettant d'obtenir un résultat. Il s'agit littéralement de *comment* traiter un problème pour pouvoir le résoudre. Néanmoins, cette séquence d'étapes seule ne permet pas de savoir quel problème on souhaite traiter, il faut donc bien indiquer le contexte et l'objectif de l'algorithme. Ainsi, le problème à traiter, le *pourquoi*, est également extrêmement important.

Chercher et comprendre les problèmes rencontrés est donc très important pour pouvoir écrire les algorithmes les plus efficaces. Par exemple, si l'on cherche à trier des pierres selon leur taille, on utilisera des tamis de plus en plus large successivement pour récupérer tout d'abord les grains de sable, puis les cailloux les plus petits, et en dernier les pierres de plus grande taille (si l'on utilisait un tamis trop grand dès le début, toutes les tailles passeraient sans distinction). Dans cet exemple, il était nécessaire de constater que la taille des pierres était très importante, que des outils permettent de laisser passer ou non des pierres d'une certaine taille sont disponibles, et que l'on ne s'intéressait finalement pas à d'immenses rochers. Pour reprendre la comparaison avec les questions, les pierres et leurs tailles correspondent aux réponses du *quoi*. Ainsi :

- Pourquoi / Quel est l'objectif ? **Trier** des pierres par taille
- Quoi / Que manipule-t-on ? Des pierres de différentes tailles
- Comment ? En utilisant successivement des tamis avec des trous de plus en plus grands

Ces spécifications seront très importantes lorsque vous serez amenés à écrire des algorithmes : pensez toujours à bien vérifier les *spécifications* du problème avant d'essayer de répondre au problème (il peut arriver qu'en fait il n'y ait aucun problème).

En algorithmique, il existe quelques types fondamentaux permettant de représenter la plupart des informations du monde physique. En combinant ces types, on peut donc représenter quasiment tout ce qui existe et est mesurable (la taille d'une pierre, sa composition chimique, sa dureté, sa brillance et sa forme une fois taillée, etc).

- entier (*integer* en anglais) : il s'agit des entiers relatifs (positifs et négatifs)
- flottant (*float* ou *double* en anglais) : il s'agit des nombres à virgule (attention, ce type a des problèmes de *précision* : on ne peut pas toujours comparer correctement des flottants)
- caractère (*character* en anglais) : il s'agit des lettres ou caractères (à noter que ce type manipule une seule et unique lettre à la fois)
- chaîne de caractères (*string* en anglais) : il s'agit d'une suite de caractères

1. Introduction à l'Algorithmique. 2001 (2^e édition) T.Cormen et al.

2 Exécution pas à pas

1) Afin de bien comprendre comment fonctionne un algorithme, comment l'exécuter, et potentiellement comment le corriger, utilisez cet algorithme calculant la somme des n premiers entiers en l'exécutant à la main et en remplissant le tableau suivant pour $n = 5$.

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$$

```
algorithme fonction Somme : entier
  parametres locaux
    entier    n
  variables
    entier    i, sum

debut
i ← 1
sum ← 0
tant que (i <= n) faire
  sum ← sum + i
  i ← i + 1
fin tant que
retourner (sum)
fin algorithme fonction Somme
```

tour	i	sum
0	1	0
1	2	1
2	3	3
3	4	6
4	5	10
5	6	15
6		

Algorithme de la somme des N premiers entiers

Reprenez maintenant l'exemple de la multiplication égyptienne en l'exécutant cette fois-ci à la main en remplissant le tableau suivant.

2) Vous prendrez comme premières valeurs de test : $a = 4$ et $b = 5$.

3) Remplissez un tableau similaire sur un brouillon pour les valeurs $a = 3$ et $b = 13$.

```
algorithme fonction MultEgpytienne : entier
  parametres locaux
    entier    a
    entier    b
  variables
    entier    x, y, z

debut
x ← a
y ← b
z ← 0
tant que (y > 0) faire
  si (y EST IMPAIRE) alors
     z ← z + x
  fin si
  x ← 2 × x
  y ← y ÷ 2
fin tant que
retourner (z)
fin algorithme fonction MultEgpytienne
```

tour	x	y	z
0	4	5	0
1	8	2	4
2	16	1	4
3	32	0	20

Algorithme de la multiplication égyptienne

tour	x	y	z
0	3	13	0
1	6	6	3
2	12	3	3
3	24	1	15
3	48	0	39

Dans le domaine informatique, on appelle *traces d'exécution* le résultat d'exécution des programmes, avec si possible l'affichage de l'évolution de certaines variables et résultats. En remplissant à la main les tableaux avec l'évolution des valeurs, vous avez produit des traces d'exécution.
Vous l'aurez compris, les traces sont beaucoup plus utiles lorsqu'il y a beaucoup de valeurs intéressantes suivies (il faut donc que les développeurs prévoient l'affichage de ces valeurs, ou une option permettant d'afficher ces traces).
En anglais, l'activation des options ou modes *verbose* (qui pourrait être traduit par *verbeux*) impliquent d'afficher un peu plus de variables qu'initialement prévu. Ces affichages se font généralement dans des *logs* (traduit par *journaux*) ou à minima une sortie spécifique aux traces d'exécution pour ne pas les mélanger avec les résultats du comportement normal.

Effectuez maintenant l'algorithme de la division euclidienne. L'algorithme renverra le quotient.

4) Vous prendrez comme premières valeurs de test : $a = 19$ et $b = 3$.

```
algorithme fonction DivEuclideQuotient :
   entier
  parametres locaux
    entier    a
    entier    b
  variables
    entier    x, y

debut
x ← a
y ← 0
tant que (x > 0) faire
  x ← x - b
  y ← y + 1
fin tant que
retourner (y)
fin algorithme fonction DivEuclideQuotient
```

tour	x	y
0	19	0
1	16	1
2	13	2
3	10	3
4	7	4
5	4	5
6	1	6

Algorithme du quotient de la division euclidienne

5) Quelle variable faut-il renvoyer pour obtenir le reste ?

Il faut renvoyer $x + b$ pour obtenir le reste. Et on se rend également compte qu'il faut corriger l'algorithme en renvoyant $y - 1$ pour obtenir le vrai quotient dans certains cas. Cependant, on se rend compte que cet algorithme ne fonctionne que pour deux nombres positifs : il faudrait donc mieux préciser les spécifications.

6) Si le test dans le *tant que* était un >= plutôt qu'un > : quels changements à l'exécution cela produirait-il ?

Lorsque l'on donne deux nombres où x divise y, alors le quotient serait incorrect. Mais, grâce à cette condition corrigée, on peut effectivement retrouver le bon quotient dans tous les cas en appliquant $y - 1$ juste avant de renvoyer le résultat.

7) Quelles modifications faudrait-il apporter pour obtenir le bon quotient ? le bon reste ?

(Voir réponses précédentes)

8) Cet algorithme est incapable de gérer le cas où 0 est fourni en tant que diviseur. Comment pourrait-on corriger cela afin de protéger l'algorithme d'une boucle infinie ?

Il suffit d'insérer juste après le "début" une condition testant si b est à 0. Si oui, alors on renvoie une valeur spéciale, ou on déclenche une exception.

Dans la plupart des langages de programmation, vous verrez qu'un opérateur **`mod`** ou `%` existe et est assez fréquemment utilisée. Il s'agit simplement de l'opérateur calculant le reste de la division euclidienne.
$42 \% 10 = 2$ (lorsque l'on divise 42 par 10, le reste est 2)
$40 \% 10 = 0$ (lorsque l'on divise 40 par 10, le reste est 0)
$42 \% 2 = 0$ (lorsque l'on divise 42 par 2, le reste est 0)
$9 \% 10 = 9$ (lorsque l'on divise 9 par 10, le reste est 9)

3 Écriture d'algorithmes simples

Plusieurs opérations qui nous semblent évidentes sont en réalité bien plus complexes à réaliser dans la pratique.
La multiplication égyptienne est un exemple très concret de cela : nous savons multiplier car nous avons appris et compris ce qu'il se passait lors de cette opération, mais sans l'apprentissage, il est difficile de connaitre le résultat d'une multiplication. Tout comme il est difficile de multiplier de tête des nombres à virgules entre eux.
Les opérations simples sont cependant essentielles à l'écriture de programmes et d'algorithmes plus complexes.

9) Maintenant que vous savez lire, exécuter (y compris en mode pas à pas), et corriger un algorithme, écrivez l'algorithme de la multiplication classique à base d'additions ($N \times M$ = N additions de la valeur M) dans le cas de nombres positifs uniquement.

N'hésitez pas à utiliser un exemple général simple pour bien déterminer la boucle à écrire :

$$5 \times 3 = 5 + 5 + 5 = 15$$

On peut donc s'attendre à avoir une accumulation dans une variable pour le résultat :

$$0, 5, 10, 15$$

$$0(+5)$$
$$5(+5)$$
$$10(+5)$$
$$15$$

Mais, on doit également connaître le cas d'arrêt : lorsque le multiplicateur est à 0.

5	3	0
5	2	5
5	1	10
5	0	15

```
algorithme fonction MultClassique : entier
  parametres locaux
    entier    a
    entier    b
  variables
    entier    total

debut
total = 0
tant que (b > 0)
  total = total + a
  b = b - 1
fin tant que
retourner (total)
fin algorithme fonction MultClassique
```

10) Comment peut-on traiter les nombres négatifs ? Écrivez maintenant une fonction *MultRelatifs* permettant de multiplier des nombres entiers négatifs (n'hésitez pas à appeler la fonction de multiplication que vous avez précédemment écrite).

```
algorithme fonction MultRelatifs : entier
  parametres locaux
    entier    a
    entier    b
  variables
    entier    total, signe

debut
signe = 1
si (a < 0)
  signe = signe * (-1)
  a = a * (-1)
fin si
si (b < 0)
  signe = signe * (-1)
  b = b * (-1)
fin si
total = signe * MultClassique(a, b)
retourner (total)
fin algorithme fonction MultRelatifs
```

11) Écrivez l'algorithme calculant la puissance de x^n (pour n positif ou nul).

Au lieu d'utiliser les symbole $\times$ ou * pour multiplier, vous ferez un appel à votre dernière fonction *MultRelatifs* en lui donnant deux paramètres et en récupérant le résultat.

```
algorithme fonction Puissance : entier
  parametres locaux
    entier    a
    entier    b
  variables
    entier    total

debut
si (b == 0)
  retourner (1)
fin si
total = 1
pour (i = 0) jusqu'a (b)
  total = MultRelatifs(total, a)
fin pour
retourner (total)
fin algorithme fonction Puissance
```

12) Écrivez l'algorithme testant la parité d'un nombre n.

La parité est simplement la qualité d'un nombre d'être pair ou impair. Vous renverrez 0 en cas de nombre pair, et 1 en cas de nombre impair.

```
algorithme fonction Parite : entier
  parametres locaux
    entier    n
debut
si ((n % 2) == 0)
  retourner (0)
sinon
  retourner (1)
fin si
fin algorithme fonction Parite
```

4 Logique et écriture d'algorithmes

En mathématiques, un domaine étudie en particulier la *logique* de façon formelle (logique de premier ordre, ...). Ce domaine est directement appliqué en électronique numérique avec les portes logiques (*logic gates* en anglais) et les bascules (*flip-flops* en anglais).

La logique s'appuie sur deux valeurs qui s'opposent : **`vrai`** (1) et **`faux`** (0). Il est possible d'exprimer des assertions avec des *formules logiques* afin de vérifier si celles-ci sont vraies ou fausses dans certains cas selon des paramètres. Pour cela, plusieurs opérateurs logiques existent. Les trois opérateurs fondamentaux (faisant découler toute une série d'autres opérateurs utiles) existent : **`NOT`** (non), **`AND`** (et), **`OR`** (ou).

`NOT` est un opérateur unaire, il ne s'applique qu'à un seul paramètre : « - A ».
`AND` et **`OR`** sont des opérateurs binaires, ils s'appliquent à deux paramètres : « A et B », « A ou B ».
Nous représentons dans les tableaux suivants ce qui s'appelle des *tables de vérités* : les résultats des assertions logiques.

A	**`NOT`**
0	1
1	0

A	B	**`AND`**
0	0	0
0	1	0
1	0	0
1	1	1

A	B	**`OR`**
0	0	0
0	1	1
1	0	1
1	1	1

Ces trois opérateurs sont très souvent utilisés pour tester des conditions. Si le résultat est vrai (1), alors la condition est validée. On peut combiner ces opérateurs pour tester des assertions logiques.

Cependant, ces opérateurs sont également appliqués dans le cadre de langages de programmation pour modifier des valeurs au niveau des bits les constituant (comme nous le verrons plus tard). En plus de **`NOT`**, **`OR`**, **`AND`**, vous pourrez rencontrer les opérateurs **`NAND`** (non et), **`NOR`** (non ou), et le très important **`XOR`** (ou exclusif) qui est extrêmement utilisé en cryptographie. Ces opérateurs sont simplement des combinaisons des trois opérateurs fondamentaux : **`NAND`** applique simplement un **`NOT`** au résultat du **`AND`**, et **`NOR`** applique simplement un **`NOT`** au résultat du **`OR`**.
`XOR` est légèrement plus complexe dans sa construction, mais sa logique est simple : il faut que les deux entrées soient dans des état différents pour que le résultat soit vrai (1). Vous pouvez néanmoins constater qu'en appliquant un **`AND`** aux sorties de **`OR`** et **`NAND`**, on obtient la même table de vérité.

A	B	**`NAND`**
0	0	1
0	1	1
1	0	1
1	1	0

A	B	**`NOR`**
0	0	1
0	1	0
1	0	0
1	1	0

A	B	**`XOR`**
0	0	0
0	1	1
1	0	1
1	1	0

4.1 Exercices de logique

13) Écrivez une fonction prenant 3 entiers en paramètre, et indiquant lequel est le plus petit/grand. *min(a, b, c) max(a, b, c)*

```
algorithme fonction Min3 : entier
  parametres locaux
    entier    a, b, c
debut
si (a <= b) et (a <= c)
  retourner (a)       # A plus petit que les 2
sinon si (b <= c)    # A plus grand que l'un des 2
  retourner (b)       #    [donc pas le plus petit]
sinon
  retourner (c)
fin si
fin algorithme fonction Min3
```

```
algorithme fonction Max3 : entier
  parametres locaux
    entier    a, b, c
debut
si (a >= b) et (a >= c)
  retourner (a)       # A plus grand que les 2
sinon si (b >= c)    # A plus petit que l'un des 2
  retourner (b)       #    [donc pas le plus grand]
sinon
  retourner (c)
fin si
fin algorithme fonction Max3
```

```
algorithme fonction Min3_v2 : entier
  parametres locaux
    entier    a, b, c
debut
min = a
si (b < min)
  min = b
si (c < min)
  min = c
fin si
retourner (min)
fin algorithme fonction Min3_v2
```

Somme des carrés des deux plus grandes valeurs parmi trois :

```
algorithme fonction SommeCarres1 : entier
  parametres locaux
    entier    a, b, c
debut
MAX = max3(a,b,c)
si (MAX == a)
  a = 0
sinon si (MAX == b)
  b = 0
sinon
  c = 0
MID = max3(a,b,c)
retourner (MAX * MAX + MID * MID)
fin algorithme fonction SommeCarres1
```

Version pas très maline, mais qui fonctionne

```
algorithme fonction SommeCarres2 : entier
  parametres locaux
    entier    a, b, c
debut
si (min3(a,b,c) == a)
  retourner (b * b + c * c)
sinon si (min3(a,b,c) == b)
  retourner (a * a + c * c)
sinon
  retourner (a * a + b * b)
fin algorithme fonction SommeCarres2
```

Version courte

```
algorithme fonction SommeCarres3 : entier
  parametres locaux
    entier    a, b, c
debut
MIN = min3(a,b,c)
a = a * a
b = b * b
c = c * c
MIN = MIN * MIN
sum = a + b + c - MIN
retourner (sum)
fin algorithme fonction SommeCarres3
```

Version logique

```
algorithme fonction SommeCarres4 : entier
  parametres locaux
    entier    a, b, c
debut
si (a >= b) et (a >= c)
  si (b > c)
    retourner (a * a + b * b)
  sinon
    retourner (a * a + c * c)
sinon si (b >= a) et (b >= c)
  si (a > c)
    retourner (b * b + a * a)
  sinon
    retourner (b * b + c * c)
sinon
  si (a > b)
    retourner (c * c + a * a)
  sinon
    retourner (c * c + b * b)
fin algorithme fonction SommeCarres4
```

Version sans appel à sous-fonction ["Super-Duper-Bourrine" - Le Magicien]

5 Récursivité

La récursivité est un principe très simple où une fonction se rappelle elle-même.

L'écriture d'algorithmes récursifs implique au moins deux choses dans cet ordre très précis : une condition d'arrêt où l'on retourne le résultat, puis, un appel récursif avec un paramètre modifié (si on ne modifie aucun paramètre, alors la récursion serait infinie : on rappellerait la fonction dans les mêmes conditions qu'actuellement, donc elle se rappellerait encore une fois avec strictement les mêmes paramètres).

Il est nécessaire d'écrire les conditions d'arrêts en premier, car il s'agit des cas exceptionnels où l'on doit arrêter la récursion. De même, il est hautement conseillé d'écrire les cas les plus génériques en dernier, car des cas partiellement exceptionnels pourraient être absorbés plus tôt par le cas général. Par exemple, si l'on souhaite dénombrer les N premiers entiers, et afficher à chaque dizaine un message particulier, on déclarera en premier la condition d'arrêt où un paramètre est à 0 ou 1, puis, on écrira la condition testant les valeurs notables particulières (ici si le paramètre est une dizaine), et en tout dernier on écrira le cas général qui concerne n'importe quelle valeur.

5.1 Exercices récursifs

Maintenant que vous avez écrit quelques algorithmes simples avec des boucles, nous allons passer à leurs versions récursives.

14) Commencez par exécuter l'algorithme de la somme des N premiers entiers en remplissant le tableau avec l'évolution des paramètres donnés dans un premier temps, puis des résultats. Vous effectuerez cette exécution avec 5 comme paramètre.

```
algorithme fonction SommeRec :
   entier
  parametres locaux
    entier    n

debut
si (n == 1) alors
  retourner (1)
sinon
  retourner (n + SommeRec(n - 1))
fin si
fin algorithme fonction SommeRec
```

appel	n
0	5
1	4
2	3
3	2
4	1
5	
6	

appel	retour	total
6		
5		
4	1	1
3	2	4
2	4	7
1	7	11
0	11	16

Somme des N premiers entiers (récursif)

Vous remarquerez que l'algorithme est beaucoup plus court en quantité d'instructions. Ceci est principalement dû au fait que le calcul que nous exécutons est déjà dans une forme adaptée (souvenez-vous du principe de récurrence, ou encore des suites) : la même opération est répétée avec un paramètre réduit ou augmenté de 1 (ou d'un pas bien défini).

15) Écrivez maintenant l'algorithme de la factorielle, mais de façon récursive. N'oubliez pas : on écrit d'abord la ou les conditions d'arrêt, et ensuite seulement on effectue l'opération avec l'appel récursif.

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

```
algorithme fonction Factorielle : entier
  parametres locaux
    entier    n
debut
si (n <= 1)
  retourner (1)
sinon
  retourner (n * Factorielle(n - 1))
fin si
fin algorithme fonction Factorielle
```

16) Écrivez l'algorithme récursif calculant la somme des N premiers entiers.

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$$

```
algorithme fonction SommeRec : entier
  parametres locaux
    entier    n
debut
si (n == 0)
  retourner (0)
sinon
  retourner (n + SommeRec(n - 1))
fin si
fin algorithme fonction SommeRec
```

17) Écrivez l'algorithme récursif calculant la multiplication de deux entiers positifs, en n'utilisant que des additions et des soustractions.

```
algorithme fonction MultiplicationRec0 : entier
  parametres locaux
    entier    a, b
debut
si (b == 0)
  retourner (b)
sinon
  retourner (a + MultiplicationRec0(a, (b - 1)))
fin si
fin algorithme fonction MultiplicationRec0
```

Cette version ne gère que les nombres positifs... comment faire les négatifs ? De plus, si A est à 0, mais B est énorme... alors on fera beaucoup de calculs pour rien

18) Améliorez l'algorithme de la multiplication pour qu'elle gère maintenant les nombres négatifs. Vous pouvez pour cela vous aider d'une fonction chapeau, c'est-à-dire une fonction qui prend les deux paramètres attendus (les deux nombres à multiplier) et fait différents tests avant d'appeler une autre fonction qui elle sera récursive.

```
algorithme fonction MultiplicationRec1 : entier
  parametres locaux
    entier    a, b
debut
si (a == 0) ou (b == 0)
  retourner (0)
sinon si (b < 0)
  retourner (-a + MultiplicationRec1(a, (b + 1)))
sinon
  retourner (a + MultiplicationRec1(a, (b - 1)))
fin si
fin algorithme fonction MultiplicationRec1
```

Le 1er test sera constamment exécuté à chaque appel... cela consommera du temps CPU pour rien : on peut sortir ce test dans ce que l'on appelle une "fonction chapeau"

```
algorithme fonction MultiplicationRec2Chapo : entier
  parametres locaux
    entier    a, b
debut
si (b == 0)
  retourner (0)
sinon si (b < 0)
  retourner (-a + MultiplicationRec2(a, (b + 1)))
sinon
  retourner (a + MultiplicationRec2(a, (b - 1)))
fin si
fin algorithme fonction MultiplicationRec2Chapo

algorithme fonction MultiplicationRec2 : entier
  parametres locaux
    entier    a, b
debut
si (a == 0) ou (b == 0)
  retourner (0)
sinon
  retourner (MultiplicationRec2(a, b))
fin si
fin algorithme fonction MultiplicationRec2
```

Le test de A et B à 0 n'est maintenant fait qu'une seule fois dans la fonction chapeau, avant d'appeler la fonction réellement récursive qui effectue les traitements

19) Écrivez l'algorithme récursif calculant le Nième terme d'une suite géométrique. Vous devriez avoir en paramètres : le terme u_0 désignant le premier terme de la suite, la raison q, et le numéro n du terme recherché.

$$u_n = u_0 \times q^n$$

```
algorithme fonction SuiteGeometriqueRec : entier
  parametres locaux
    entier    u0, q, n
debut
si (q == 0)
  retourner (0)
sinon si (n == 0)
  retourner (u0)
sinon
  retourner (q * SuiteGeometriqueRec(u0, q, (n-1)))
fin algorithme fonction SuiteGeometriqueRec
```

20) Écrivez une fonction récursive calculant le $n^{\text{ème}}$ terme de la suite de Fibonacci.

$$fibo(0) = fibo(1) = 1$$
$$fibo(n) = fibo(n-1) + fibo(n-2)$$

```
algorithme fonction FiboRec : entier
  parametres locaux
    entier    n
debut
si (n <= 1)
  retourner (1)
sinon
  retourner (FiboRec(n-1) + FiboRec(n-2))
fin si
fin algorithme fonction FiboRec
```

Le cas n = 0 n'est pas géré ici, mais on peut ajouter une condition supplémentaire pour le gérer

```
algorithme fonction FiboRecTerm1 : entier
  parametres locaux
    entier    n, acc
debut
si (n == 0)
  retourner (acc)
sinon si (n == 1)
  retourner (acc + 1)
sinon
  retourner (0, FiboRecTerm1(n-1, acc) + FiboRecTerm1(n-2, acc))
fin si
fin algorithme fonction FiboRecTerm1
```

```
algorithme fonction FiboRecTerm2 : entier
  parametres locaux
    entier    n, a, b
debut
si (n == 0)
  retourner (a)
sinon si (n == 1)
  retourner (b)
sinon
  retourner (FiboRecTerm2(n-1, b, a+b))
fin si
fin algorithme fonction FiboRecTerm2
```

21) Écrivez maintenant la version itérative de la suite de Fibonacci. [Astuce : on dispose de deux cas à valeurs fixes, et à chaque étape on doit se rappeler du résultat précédent.]

```
algorithme fonction FiboIter : entier
  parametres locaux
    entier    n
debut
si (n == 0)
  retourner (0)
var1 = 0
var2 = 1
tant que (n > 0)
  total = var1 + var2
  var1 = var2
  var2 = total
  print(total)   # Affiche la suite
fin tant que
retourner (total)
fin algorithme fonction FiboIter
```

Pour écrire la version itérative, il faut simplement constater que l'on doit se souvenir d'un terme, et de son prédécesseur... donc à chaque itération, on réutilise l'ancienne

22) Écrivez une fonction récursive calculant le $n^{\text{ème}}$ terme de la suite d'Ackermann.

$$\begin{aligned} A(0,n) &= n+1 && [n >= 0] \\ A(m,0) &= A(m-1,1) && [m > 0] \\ A(m,n) &= A(m-1, A(m,n-1)) && [m > 0 \ \& \ n > 0] \end{aligned}$$

```
algorithme fonction Ackermann : entier
  parametres locaux
    entier    m, n
debut
si (m == 0)
  retourner (n + 1)
sinon si (n == 0)
  retourner (Ackermann((m - 1), 1))
sinon
  retourner (Ackermann((m - 1), Ackermann(m, (n - 1)))
fin si
fin algorithme fonction Ackermann
```

Ne testez pas ce code avec des valeurs supérieures à 3 pour m et n...

23) Écrivez l'algorithme récursif calculant le nombre de combinaisons de p dans n (C_n^p ou CPN), c'est-à-dire le nombre de parties à p éléments dans un ensemble E contenant n éléments.

Par exemple : pour $p = 2$, on recherche tous les **couples** possibles de différents éléments. Pour $p = 3$, on recherche tous les **triplets** possibles de différents éléments. En indiquant $n = 3$, on vise un ensemble composé de trois éléments distincts (par exemple : $E = \{1, 2, 3\}$, ou $E = \{A, B, C\}$, ou $E = \{\spadesuit, \heartsuit, \clubsuit\}$, il s'agit juste de trois éléments distincts).

Ainsi, pour $p = 2$ et $n = 3$, on recherche tous les couples possibles de trois éléments :

(A,B) (A,C)
(B,C)

$\Rightarrow C_3^2 = 3$ (3 couples possibles)

Pour $p = 2$ et $n = 4$, on recherche tous les couples possibles de quatre éléments :

(A,B) (A,C) (A,D)
(B,C) (B,D)
(C,D)

$\Rightarrow C_4^2 = 6$ (6 couples possibles)

Pour $p = 3$ et $n = 3$, on recherche tous les triplets possibles de trois éléments :

(A,B,C)

$\Rightarrow C_3^3 = 1$ (1 triplet possible)

Pour $p = 3$ et $n = 4$, on recherche tous les triplets possibles de quatre éléments :

(A,B,C) (A,B,D) (A,C,D)
(B,C,D)

$\Rightarrow C_4^3 = 4$ (4 triplets possibles)

Voici les axiomes pour votre implémentation :

$$\begin{array}{ll} C(0,n) = 1 & \\ C(n,n) = 1 & [n \neq 0] \\ C(p,n) = C(p,n-1) + C(p-1,n-1) & [n \neq p] \end{array}$$

```
algorithme fonction CombiRec : entier
  parametres locaux
    entier    p, n
debut
si (p == 0)
  retourner (1)
sinon si (p == n)
  retourner (1)
sinon
  retourner (CombiRec(p, n-1) + CombiRec(p-1, n-1))
fin algorithme fonction CombiRec
```

5.2 Exercices variés (récursif & itératif)

Les exercices dans cette section doivent plutôt être réalisés en itératif. Il est précisé lesquels peuvent aisément être réalisés en récursif. Il est interdit d'utiliser les tableaux ou pointeurs pour réaliser ces exercices.

24) Écrivez une fonction transformant un format horaire en un format uniquement composé de secondes. Cette fonction prendra 3 entiers en paramètre (les heures, les minutes, et les secondes) et les convertira en secondes. Par exemple, 1h 23m 45s deviendra 5025 secondes. *ConversionHoraire1(hh, mm, ss)*

```
algorithme fonction ConversionHoraire1 : entier
  parametres locaux
    entier    hh, mm, ss
debut
HH = hh * 3600
MM = mm * 60
SS = ss
total = HH + MM + SS
retourner (total)
fin algorithme fonction ConversionHoraire1
```

Version 1

```
algorithme fonction ConversionHoraire2 : entier
  parametres locaux
    entier    hh, mm, ss
debut
HH = hh * 60
MM = (mm + HH) * 60
SS = ss + MM
retourner (SS)
fin algorithme fonction ConversionHoraire2
```

Version 2

```
algorithme fonction ConversionHoraire3 : entier
  parametres locaux
    entier    hh, mm, ss
debut
retourner (hh * 3600 + mm * 60 + ss)
fin algorithme fonction ConversionHoraire3
```

Version courte

25) Écrivez une autre fonction de conversion horaire qui prend cette fois un unique entier qui respecte un format précis (hhmmss) pour le convertir en secondes. Par exemple le paramètre 153042 signifie 15h 30m 42s qu'il faut convertir en secondes. *ConversionHoraire2(hhmmss)*

```
algorithme fonction ConversionHoraireFormat1 : entier
  parametres locaux
    entier    hhmmss
debut
hh = hhmmss / 10000
mm = (hhmmss / 100) - (hh * 100)
ss = hhmmss - (hh * 10000) - (mm * 100)
retourner (hh * 3600 + mm * 60 + ss)
fin algorithme fonction ConversionHoraireFormat1
```

Version 1

```
algorithme fonction ConversionHoraireFormat2 : entier
  parametres locaux
    entier    hhmmss
debut
total = 0
tant que (hhmmss > 10000)
  hhmmss = hhmmss - 10000
  total = total + 3600
fin tant que
tant que (hhmmss > 100)
  hhmmss = hhmmss - 100
  total = total + 60
fin tant que
retourner (total + hhmmss)
fin algorithme fonction ConversionHoraireFormat2
```

Version 2

26) Écrivez une fonction qui transforme un nombre en son miroir. Cette fonction prend un entier en paramètre, et construit un autre entier qui est son miroir. Par exemple, pour 12034, son miroir sera 43021. Autre exemple : 2000 aura comme miroir 0002, c'est-à-dire 2. Attention aux nombres composés d'un nombre pair/impair de chiffres. Commencez par réaliser une fonction itérative. *MiroirIter(n)*

```
algorithme procedure NombreMiroirIterPrint
  parametres locaux
    entier    n
  variables
    entier var
debut
si (n == 0)
  ecrire(0)
sinon
  tant que (n > 0)
    var = n % 10
    n = n / 10
    ecrire(var)
  fin tant que
fin algorithme procedure NombreMiroirIterPrint
```

La fonctione "écrire" est l'équivalent d'un "print", c'est-à-dire qu'elle affiche à l'écran une valeur

```
algorithme fonction NombreMiroirIter : entier
  parametres locaux
    entier    n
  variables
    entier var, new
debut
si (n == 0)
  retourner (0)
new = 0
tant que (n > 0)
  var = n % 10
  new = (new * 10) + var
  n = n / 10
fin tant que
retourner (new)
fin algorithme fonction NombreMiroirIter
```

```
algorithme fonction NombreMiroirIterMagicien : entier
  parametres locaux
    entier    n
  variables
    entier mirror, neg
debut
mirror = 0
neg = 1
si (n < 0)
  neg = -1
  n = -n
tant que (n > 0)
  mirror = (mirror * 10) + (n % 10)
  n = n / 10
fin tant que
retourner (neg * mirror)
fin algorithme fonction NombreMiroirIterMagicien
```

Nombre miroir fait par Le Magicien : cette version gère même les nombres négatifs

27) Écrivez maintenant la version récursive du nombre miroir. Pour cette première version récursive, vous appelerez une fonction *écrire(x)* ou *print(x)* qui affiche un caractère ou un chiffre à la fois. *MiroirRec1(n)*

```
algorithme procedure NombreMiroirRecPrint1
  parametres locaux
    entier    n
debut
si (n < 0)
  ecrire("\n")
sinon
  ecrire(n % 10)
  NombreMiroirRec1(n / 10)
fin si
fin algorithme procedure NombreMiroirRecPrint1
```

La fonctione "écrire" est l'équivalent d'un "print", c'est-à-dire qu'elle affiche à l'écran une valeur

```
algorithme procedure NombreMiroirRecPrint2
  parametres locaux
    entier    n
debut
si (n > 0)
  ecrire(n % 10)
  NombreMiroirRecPrint2(n / 10)
fin si
fin algorithme procedure NombreMiroirRecPrint2
```

28) Écrivez maintenant la version récursive du nombre miroir. Pour cette deuxième version récursive, vous devrez renvoyer le nombre miroir et non pas juste l'afficher. *MiroirRec2(n)*

Astuce : vous pouvez utiliser un *accumulateur* comme deuxième paramètre, donc, écrire une fonction chapeau qui prendra un seul paramètre et préparera l'appel à la fonction récursive.

```
algorithme fonction NombreMiroirRecChapo : entier
  parametres locaux
    entier    n
debut
  retourner (NombreMiroirRec(n, 0))
fin algorithme fonction NombreMiroirRecChapo

algorithme fonction NombreMiroirRec : entier
  parametres locaux
    entier    n, acc
debut
si (n <= 0)
  retourner (acc)
sinon
  retourner (NombreMiroirRec((n / 10), (10 * acc + (n % 10))))
fin si
fin algorithme fonction NombreMiroirRec
```

La fonction chapeau initialise uniquement l'accumulateur à 0, et on n'oublie pas de "renvoyer" le résultat de la fonction réellement récursive

29) Écrivez une fonction récursive qui affiche les éléments successifs de la conjecture de Syracuse, mais qui renvoie également le nombre d'éléments produits avant d'atteindre 1. Utilisez les fonctions *écrire(x)* ou *print(x)* et une fonction chapeau si nécessaire. *Syracuse(n)*

Voici les axiomes pour votre implémentation :

$$\begin{array}{ll} Syracuse(0) = 1 & \\ Syracuse(1) = 1 & \\ Syracuse(n) = n/2 & \text{si } n \text{ est paire} \\ Syracuse(n) = 3n + 1 & \text{si } n \text{ est impaire} \end{array}$$

```
algorithme fonction SyracuseRecChapo : entier
  parametres locaux
    entier    n
debut
  retourner (SyracuseRec(n, 0))
fin algorithme fonction NombreMiroirRecChapo

algorithme fonction SyracuseRec : entier
  parametres locaux
    entier    n, acc
debut
ecrire(n)
si (n == 1)
  retourner (acc)
si (n % 2) == 0
  retourner (SyracuseRec((n / 2), acc + 1)
sinon
  retourner (SyracuseRec((3n + 1), acc + 1)
fin algorithme fonction NombreMiroirRec
```

30) Écrivez une fonction détectant si un nombre est un palindrome. La fonction renvoie *vrai* si c'est un palindrome, sinon elle renvoie *faux*. Un palindrome est simplement un mot ou un nombre composé des mêmes caractères ou chiffres sur sa première moitié par rapport à sa deuxième moitié. Par exemple, 27972 est un palindrome. 1331 est également un palindrome, mais 1664 n'en est pas un. Faites d'abord une version itérative, puis faites une version récursive. *PalindromeIter(n)* *PalindromeRec(n)*

```
algorithme fonction PalindromeIter1 : booleen
  parametres locaux
    entier    n
  variables
    entier   Dernier, Premier, len
debut
tant que (n > 0)
  len = 1
  Dernier = n % 10
  Premier = n
  tant que (Premier > 10)
    Premier = Premier / 10
    len = len + 1
  fin tant que
  si (Dernier != Premier)
    retourner (Faux)
  fin si
  n = n - (Premier * len)
  n = (n - (n % 10)) / 10
fin tant que
retourner (Vrai)
fin algorithme fonction PalindromeIter1
```

Version complète : on prend le 1er et le dernier chiffre, et on les compare

```
algorithme fonction PalindromeIter1bis : booleen
  parametres locaux
    entier    n
  variables
    entier   Dernier, Premier, len
debut
tant que (n > 0)
  len = 1
  Dernier = n % 10
  Premier = n
  tant que (Premier > 10)
    Premier = Premier / 10
    len = len + 1
  fin tant que
  si (Dernier != Premier)
    retourner (Faux)
  fin si
  n = (n - (Premier * len) - (n % 10)) / 10
fin tant que
retourner (Vrai)
fin algorithme fonction PalindromeIter1bis
```

```
algorithme fonction PalindromeIterMagique : booleen
  parametres locaux
    entier    n
  variables
    entier   temp, reverse, rem
debut
temp = n
reverse = 0
tant que (temp > 0)
  rem = temp % 10
  reverse = (reverser * 10) + rem
  temp = temp / 10
fin tant que
si (n == reverse)
  retourner (Vrai)
sinon
  retourner (Faux)
fin si
fin algorithme fonction PalindromeIterMagique
```

Version du Magicien : on construit un "miroir" et on le compare avec le nombre initial

6 Tableaux et Chaînes de caractères

Les tableaux en algorithmique, et dans la plupart des langages de programmation, sont simplement des vecteurs. C'est-à-dire qu'il s'agit de tableaux à une seule dimension. La plupart des langages de programmation font démarrer leurs tableaux à la position 0. Ceci implique qu'un tableau de taille 5 (donc qui contient 5 cases) démarre de la case 0 et finit à la case 4 (01234). Dans vos algorithmes, vous manipulerez donc souvent les cases de 0 à *longueur du tableau - 1*, avec comme condition *tant que (iterateur < longueur)* (on s'arrête lorsque l'itérateur atteint la longueur, donc après la dernière case du tableau). Il existe cependant quelques langages où les tableaux démarrent à la case 1, donc finissent à la case *longueur du tableau* : vérifiez toujours quel est l'index (le numéro de case) de la première case dans chacun des langages de programmation que vous utiliserez.

42	14	18	666	1337
0	*1*	*2*	*3*	*4*

Les tableaux contiennent parfois des éléments de types différents (des entiers, des flottants, et des chaînes de caractères), mais il est fréquent qu'ils ne peuvent contenir qu'un seul et unique type à la fois (uniquement des entiers, ou uniquement des flottants, ou uniquement un type précis). Dans les exercices de ce sujet, nos tableaux ne pourront contenir qu'un seul et unique type à chaque fois. Si vous déclarez un tableau d'entiers, alors on ne peut y mettre que des entiers et rien d'autre.

Pour accéder à une case précise d'un tableau, on indique l'index entre crochets. Par exemple, pour un tableau d'entiers stocké dans la variable *tab*, on accède à la première case en écrivant : *tab[0]*. On peut récupérer le contenu de la case pour le mettre dans une variable en écrivant *var = tab[2]*. On peut écrire une valeur dans une case d'un tableau en écrivant *tab[2] = 42*.

L'index peut également être une variable, mais celle-ci doit être un entier (on ne peut pas accéder à une case dont l'index est un flottant ou un caractère). Ainsi, on ne peut ni faire *tab[0,42]* ni *tab['a']* ni *tab["trois"]*, mais si une variable n contient un entier (par exemple : $n = 3$), on peut écrire *tab[n]*.

Toujours concernant les index des cases, si on essaye d'accéder à une case inexistante (par exemple un index négatif comme -1 ou au delà de la taille du tableau), une erreur est renvoyée. Par exemple, pour un tableau de taille 5 (index de 0 à 4), on peut accéder à la case 4 (*tab[4]*) mais pas à la case 5 (*tab[5]*).

Les chaînes de caractères (c'est-à-dire les suites de caractères) sont en réalité des tableaux contenant des caractères. Le format standard des chaînes de caractères implique qu'une chaîne finisse par un unique caractère spécial : '*\0*'. Ainsi, la chaîne de caractères standard "lol" est en réalité un tableau de taille 4 dont la dernière case contient '\0'. Grâce à cette convention, on n'a plus besoin d'embarquer la taille de la chaîne, il suffit juste de chercher un '\0' pour comprendre qu'il n'y a rien après.

'l'	'o'	'l'	'\0'
0	*1*	*2*	*3*

Vous pouvez néanmoins rencontrer des chaînes de caractères non-standards, c'est-à-dire qu'elles ne finissent pas nécessairement par '\0', et peuvent même en contenir dans la chaîne elle-même. Dans ce cas très précis, on vous fournira toujours la taille de la chaîne de caractère en plus du tableau.

'A'	'b'	'C'	'\0'	'D'	'e'	'F'
0	*1*	*2*	*3*	*4*	*5*	*6*

6.1 Exercices variés (tableaux)

31) Écrivez maintenant une fonction palindrome fonctionnant sur un tableau dont la longueur est donnée en paramètre. *PalindromeTab(tab, len)*

```
algorithme fonction PalindromeTab1 : booleen
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier   ln, i
debut
ln = len - 1
i = 0
tant que (ln > 0)
  si (tab[i] != tab[ln])
    retourner (Faux)
  fin si
  i = i + 1
  ln = ln - 1
fin tant que
retourner (Vrai)
fin algorithme fonction PalindromeTab1
```

Version simple et complète : on teste chaque case 2 fois (une fois avec "i", et une fois avec "ln")

```
algorithme fonction PalindromeTab2 : booleen
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier   i
debut
i = 0
tant que (i < (len / 2))
  si (tab[i] != tab[len - 1 - i])
    retourner (Faux)
  fin si
  i = i + 1
fin tant que
retourner (Vrai)
fin algorithme fonction PalindromeTab2
```

2 optimisations : une seule variable (on déduit le décalage en utilisant l'opposé de l'itérateur), et on ne parcoure qu'une seule fois chaque case du tableau

```
algorithme fonction PalindromeTab2 : booleen
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier   i
debut
i = 0
tant que (tab[i] == tab[len - 1]) et (i < len)  # len - 1 ?
  i = i + 1
  len = len - 1
fin tant que
retourner (tab[i] == tab[len - 1])
fin algorithme fonction PalindromeTab2
```

Version optimisée et un peu plus courte encore

32) Écrivez une fonction palindrome fonctionnant sur une chaîne de caractères (le '\0' final ne sera bien entendu pas pris en compte dans le palindrome). *PalindromeStr(str)*

33) Écrivez une fonction comparant deux tableaux. Si les tableaux sont les mêmes, alors vous renverrez *vrai*, sinon vous renverrez *faux*. *CompareTab(tab1, tab2, len1, len2)*

```
algorithme fonction TabCmp1 : booleen
  parametres locaux
    entier[]  tab1, tab2
    entier    len1, len2
  variables
    entier   i
debut
  si (len1 != len2)
    retourner (Faux)
  fin si
i = 0
tant que (i < len1)
  si (tab1[i] != tab2[i])
    retourner (Faux)
  fin si
  i = i + 1
fin tant que
retourner (Vrai)
fin algorithme fonction TabCmp1
```

```
algorithme fonction TabCmp2 : booleen
  parametres locaux
    entier[]  tab1, tab2
    entier    len1, len2
  variables
    entier   i
debut
si (len1 != len2)
  retourner (Faux)
fin si
i = 0
tant que (tab1[i] == tab2[i]) et (i < (len - 1))
  i = i + 1
fin tant que
retourner (tab1[i] == tab2[i])
fin algorithme fonction TabCmp2
```

34) Écrivez une fonction qui renvoie la valeur la plus grande/petite du tableau. Vous ferez une version itérative et une version récursive pour Min et Max. *MinTabIter(tab, len) MaxTabIter(tab, len) MinTabRec(tab, len) MaxTabRec(tab, len)*

```
algorithme fonction MaxTabV1 : entier
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier   i, max   # min
debut
si (len < 1)
  retourner (-1)
fin si
max = tab[0]
pour i de 1 a (len - 1)
  si (tab[i] > max)     # Min : tab[i] < min
    max = tab[i]
  fin si
fin pour
retourner (max)
fin algorithme fonction MaxTabV1
```

Version simple et claire. Attention au "-1" au debut : peut être que "-1" correspond à une valeur utilisée... préférer une exception ou autre

```
algorithme fonction MaxTabV1.1 : entier
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier   i, max      # min
debut
si (len < 1)
  retourner (MIN_INT)    # Min : MAX_INT
fin si
max = MIN_INT            # Min : min = MAX_INT
pour i de 0 a (len - 1)
  si (tab[i] > max)      # Min : tab[i] < min
    max = tab[i]
  fin si
fin pour
retourner (max)
fin algorithme fonction MaxTabV1.1
```

Version qui ne fonctionne que dans les langages proposant une valeur représentant la plus grande/petite de toutes les valeurs (ou l'infini)

```
algorithme fonction MaxTabRec1 : entier
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier   i, max      # min
debut
si (len == 1)
  retourner (tab[0])
sinon si (tab[len - 1] > tab[len - 2])
  tab.pop(len - 2)       # Supprime l'element tab[len - 2]
  retourner (MaxRec(tab, len - 1))
sinon
  tab.pop(len - 1)       # Supprime l'element tab[len - 1]
  retourner (MaxRec(tab, len - 1))
fin si
fin algorithme fonction MaxTabRec1
```

Version qui modifie le tableau en le réduisant petit à petit (on regarde les 2 derniers éléments du tableau, et on supprime le plus petit des deux)

35) Écrivez une fonction qui calcule la somme de tous les éléments d'un tableau. Vous ferez une version itérative et une version récursive. *SommeTabIter(tab, len) SommeTabRec(tab, len)*

```
algorithme fonction SommeTabRecSimple : entier
  parametres locaux
    entier[]  tab
    entier    len
debut
si (len == 0)
  retourner (0)
sinon
  retourner (tab[len - 1] + SommeTabRecSimple(tab, len - 1))
fin si
fin algorithme fonction SommeTabRecSimple
```

Version récursive simple

```
algorithme fonction SommeTabRecChapo : entier
  parametres locaux
    entier[]  tab
    entier    len
debut
SommeTabRecTerm(tab, len, 0)
fin algorithme fonction SommeTabRecChapo

algorithme fonction SommeTabRecTerm : entier
  parametres locaux
    entier[]  tab
    entier    len
debut
si (len == 0)
  retourner (acc)
sinon
  retourner (SommeRecTerm(tab, len - 1, (acc + tab[len - 1])))
fin si
fin algorithme fonction SommeTabRecTerm
```

Version récursive terminale : on ne fait que renvoyer des résultats sans rien ajouter

```
algorithme fonction SommeTabIterV1 : entier
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier   i, somme
debut
somme = 0
pour i = 0 jusqu'a (len - 1)
  somme = somme + tab[i]
fin pour
retourner (somme)
fin algorithme fonction SommeTabIterV1
```

Version itérative simple et classique : on ajoute chaque élément en itérant sur chaque case du tableau

Améliorez l'algorithme pour n'utiliser qu'un seul itérateur tout en ajoutant à chaque fois le début et la fin du tableau (n'oubliez pas de vous arrêter là où il faut et de ne pas ajouter trop d'éléments).

36) Écrivez une fonction qui calcule la taille d'une chaîne de caractères (sans compter le '\0' final : "lol" a une taille de 3). Vous ferez une version itérative et une version récursive. *StrlenIter(str) StrlenRec(str)*

```
algorithme fonction SommeTabIterV2 : entier
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier   i, sum
debut
i = 0
sum = 0
si (len % 2) == 1
  pour i = 1 jusqu'a (len / 2)
    sum = sum + tab[i] + tab[len - 1 - i]
  i = i + 1
  fin pour
  sum = sum + tab[len / 2]
  retourner (sum)
sinon
  pour i = 1 jusqu'a (len / 2)
    sum = sum + tab[i] + tab[len - 1 - i]
  i = i + 1
  fin pour
  retourner (sum)
fin si
fin algorithme fonction SommeTabIterV2
```

Au lieu d'itérer sur chaque case, on ne fait que la moitié des itérations, car on ajoute à la fois l'élément en fin de tableau en plus de celui en début de tableau (comme pour le miroir). Le cas impaire omet l'élément du milieu

```
algorithme fonction SommeTabIterV2.1 : entier
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier   i, sum
debut
i = 0
sum = 0
pour i = 1 jusqu'a (len / 2)
  sum = sum + tab[i] + tab[len - 1 - i]
  i = i + 1
fin pour
si (len % 2) == 1
  sum = sum - tab[len / 2]
fin si
retourner (sum)
fin algorithme fonction SommeTabIterV2.1
```

Même chose que plus haut, mais comme dans le cas impaire l'élément central est ajouté 2 fois, on le supprime une fois avant de retourner le résultat

```
algorithme fonction StrlenIter : entier
  parametres locaux
    string    str
  variables
    entier   i
debut
i = 0
tant que (str[i] != '\0')
  i = i + 1
fin tant que
retourner (i)
fin algorithme fonction StrlenIter
```

On teste chaque caractère jusqu'à tomber sur le caractère final (si c'est le premier caractère, alors la chaîne est vide)

```
algorithme fonction StrlenRec : entier
  parametres locaux
    string    str
    entier    i
debut
si (str[i] == '\0')
  retourner (i)
sinon
  retourner (StrlenRec(str, (i + 1)))
fin si
fin algorithme fonction StrlenRec

algorithme fonction StrlenRecChapo : entier
  parametres locaux
    string    str
debut
retourner (StrlenRec(str, 0))
fin algorithme fonction StrlenRec
```

De la même manière que l'itératif, on teste chaque caractère jusqu'à tomber sur le caractère final

37) Écrivez une fonction qui compare deux chaînes de caractères et renvoie *vrai* si elles sont similaires et *faux* si elles diffèrent. Vous ferez une version itérative et une version récursive. *StrcmpIter(str1, str2) StrcmpRec(str1, str2)*

Essayez d'écrire une version itérative qui teste d'abord la longueur des chaînes, puis, une autre version sans ce test.

```
algorithme fonction StrcmpIterV1 : booleen
  parametres locaux
    string    str1, str2
  variable
    entier    i
debut
si (strlen(str1) != strlen(str2))
  retourner (Faux)
fin si
i = 0
tant que (i != strlen(str1))
  si (str1[i] != str2[i])
    retourner (Faux)
  fin si
  i = i + 1
fin tant que
retourner (Vrai)
fin algorithme fonction StrcmpIterV1
```

On teste d'abord si les deux chaînes sont les mêmes ou pas. Si elles sont de même taille, alors on va tester chaque caractère et quitter à la moindre différence. Si tous les caractères sont les mêmes, alors les deux chaînes sont les mêmes

```
algorithme fonction StrcmpIterV1.2 : booleen
  parametres locaux
    string    str1, str2
  variable
    entier    i
debut
si (strlen(str1) != strlen(str2))
  retourner (Faux)
fin si
i = 0
tant que (str1[i] == str2[i]) ET
         (str1[i] != '\0')
  i = i + 1
fin tant que
retourner (str1[i] == str2[i])
fin algorithme fonction StrcmpIterV1.2
```

On commence à optimiser le code en faisant une boucle qui itère sur les chaînes et sort à la moindre différence ou lorsque l'on a atteint la fin de la chaîne (le test est effectivement en "et", car il faut que les deux conditions soient vraies pour itérer, donc, il faut que l'une "ou" l'autre soit fausse pour quitter la boucle). Dès que l'on sort de la boucle, il suffit de tester si la raison pour laquelle on a quitté la boucle : si on a quitté car on a atteint un caractère de fin, alors l'égalité sera vraie, mais si on a quitté car les caractères varient, alors l'égalité finale sera fausse

```
algorithme fonction StrcmpIterV1.3 : booleen
  parametres locaux
    string    str1, str2
  variable
    entier    i
debut
i = 0
tant que (str1[i] == str2[i]) ET
         (str1[i] != '\0') ET
         (str2[i] != '\0')
  i = i + 1
fin tant que
retourner (str1[i] == str2[i])
fin algorithme fonction StrcmpIterV1.3
```

Même chose que la v1.2, mais cette fois on se passe du test de longueur : on s'arrête dès que l'une des deux chaînes est arrivée à la fin, et on renvoie le résultat du test sur ces caractères

```
algorithme fonction StrcmpRecV1 : booleen
  parametres locaux
    string    str1, str2
    entier    len
debut
si (len <= 0)
  retourner (Vrai)
sinon si (str1[len - 1] != str2[len - 1])
  retourner (Faux)
sinon
  strcmpRec1(str1, str2, len - 1)
fin si
fin algorithme fonction StrcmpRecV1

algorithme fonction StrcmpRecChapoV1 : booleen
  parametres locaux
    string    str1, str2
debut
si (strlen(str1) != strlen(str2))
  retourner (Faux)
sinon
  strcmpRec1(str1, str2, strlenRec(str1))
fin si
fin algorithme fonction StrcmpRecChapoV1
```

La fonction chapeau a filtré les cas simples (strlen pourrait être en récursif), et on part du dernier caractère jusqu'au premier pour comparer. Si on a parcouru toute la chaîne, c'est qu'elles étaient similaires

```
algorithme fonction StrcmpRecV1.2 : booleen
  parametres locaux
    string    str1, str2
    entier    i
debut
si (str1[i] == '\0') ET (str2[i] == '\0')
  retourner (Vrai)
sinon si (str1[i] != str2[i])
  retourner (Faux)
sinon
  retourner (StrCmpRec(str1, str2, (i + 1)))
fin si
fin algorithme fonction StrcmpRecV1.2

algorithme fonction StrcmpRecChapoV1.2 : booleen
  parametres locaux
    string    str1, str2
debut
  retourner (StrCmpRec(str1, str2, 0))
fin algorithme fonction StrcmpRecChapoV1.2
```

Ici, on teste exactement comme dans le cas itératif V1.3 : si les chaînes sont terminées, tout va bien, si il y a la moindre variation, on quitte. La moindre variation implique qu'une chaîne finie avant l'autre, dans tous les autres cas, on continue d'avancer. (N.B. : on pourrait retirer un des tests de la V1.3)

38) Écrivez une fonction qui recherche un élément dans un tableau. Vous ferez une version itérative et une version récursive. *RechercheEltTabIter(tab, len, elt) RechercheEltTabRec(tab, len, elt)*

```
algorithme fonction RechercheEltTabIter : booleen
  parametres locaux
    entier[]  tab
    entier    len, elt
  variables
    entier    i
debut
pour i = 0 jusqu'a (len - 1)
  si (tab[i] == elt)
    retourner (i)
  fin si
fin pour
retourner (-1)
fin algorithme fonction RechercheEltTabIter
```

On itère simplement sur l'ensemble du tableau et on renvoie la position dès que le premier élément est trouvé. Si on a parcouru tout le tableau sans trouver l'élément, alors c'est qu'il n'y est pas.

```
algorithme fonction RechercheEltTabRec : booleen
  parametres locaux
    entier[]  tab
    entier    len, elt
debut
si (len <= 0)
  retourner (-1)
sinon si (tab[len - 1] == elt)
  retourner (len - 1)
sinon
  retourner (RechercheEltTabRec(tab, len - 1, elt))
fin si
fin algorithme fonction RechercheEltTabRec

algorithme fonction RechercheEltTabRecChapo : booleen
  parametres locaux
    entier[]  tab
    entier    len, elt
debut
retourner (RechercheEltTabRec(tab, len, elt))
fin algorithme fonction RechercheEltTabRecChapo
```

On teste si on a parcouru tout le tableau, si oui on quitte avec une erreur, sinon si dans la case courante se trouve l'élément, on renvoie la position courante, et dans tous les autres cas, on continue d'avancer et de chercher

39) Écrivez une fonction qui compare deux tableaux. Si les deux tableaux contiennent les mêmes éléments aux mêmes positions, vous renverrez *vrai*, sinon vous renverrez *faux*. Vous ferez une version itérative et une version récursive. *CompareTabIter(tab1, tab2, len1, len2) CompareTabRec(tab1, tab2, len1, len2)*

```
algorithme fonction TabCmpIter : booleen
  parametres locaux
    entier[]  tab1, tab2
    entier    len1, len2
debut
si (len1 != len2)
  retourner (Faux)
fin si
i = 0
tant que (tab1[i] == tab2[i]) et (i < len1)
  i = i + 1
fin tant que
si (i == len1)
  retourner (Vrai)
fin si
retourner (Faux)
fin algorithme fonction TabCmpIter
```

Si les tableaux sont de taille différentes, on n'exécute rien. Donc, quand les tableaux sont de taille identique, on avance tant que les éléments sont les mêmes ET que l'on n'a pas atteint la fin des tableaux. Si on quitte la boucle, c'est nécessairement car on a atteint la fin du tableau OU que les éléments sont différents. On va tester si les éléments sont différents (risque de dépassement !!!)... On va tester si on a dépassé la dernière case : si oui, c'est que les tableaux étaient identiques, sinon c'est qu'il y a eu une différence

```
algorithme fonction TabCmpRec : booleen
  parametres locaux
    entier[]  tab1, tab2
    entier    len
debut
si (len == 0)
  retourner (Vrai)
sinon si (tab1[len - 1] != tab2[len - 1])
  retourner (Faux)
sinon
  retourner (TabCmpRec(tab1, tab2, (len - 1)))
fin si
fin algorithme fonction TabCmpRec

algorithme fonction TabCmpRecChapo : booleen
  parametres locaux
    entier[]  tab1, tab2
    entier    len1, len2
debut
si (len1 != len2)
  retourner (Faux)
sinon
  retourner (TabCmpRec(tab1, tab2, len1))
fin si
fin algorithme fonction TabCmpRecChapo
```

La fonction chapeau élimine les cas où les tailles sont différentes, et prépare l'unique itérateur nécessaire : la longueur des deux chaînes. On va simplement itérer depuis la fin des tableaux jusqu'au début. Si on a consommé toutes les cases, alors il n'y avait aucune différence dans les tableaux, si on a une différence sur le caractère courant, on renvoie faux, et dans tous les autres cas, on avance

40) Écrivez une fonction qui teste si les éléments d'un tableau sont tous en ordre croissant. Si tous les éléments sont ordonnés du plus petit au plus grand, alors vous renverrez *vrai*, sinon vous renverrez *faux*. Si les éléments sont tous égaux, alors le résultat sera *vrai*. Vous ferez une version itérative et une version récursive. *TestCroissantTabIter(tab, len) TestCroissantTabRec(tab, len)*

Faites la même chose pour tester la décroissance.

```
algorithme fonction TestTabCroissantIter : booleen
  parametres locaux
    entier[]  tab
    entier    len
debut
si (len <= 2)                  # si (len < 2)
  retourner (Vrai)
fin si
pour i = 0 jusqu'a (len - 2)  #  1 jusqu'a (len - 1)
  si (tab[i + 1] < tab[i])    # (tab[i] < tab[i - 1]
    retourner (Faux)
  fin si
fin pour
retourner (Vrai)
fin algorithme fonction TestTabCroissantIter
```

On avance sur chaque case, et on regarde si la suivante est plus petite ou plus grande. Si la suivante est plus petite, on quitte car la suite n'est pas croissante. Attention : on va tester toutes les cases "et la suivante"... donc ne pas oublier de s'arrêter avant la dernière ! On va donc comparer "au plus" l'avant dernière et la dernière (d'où le len - 2). On peut également commencer par la case 1 au lieu de 0, et comparer avec l'élément précédent (dans ce cas on peut itérer jusqu'au dernier élément)

```
algorithme fonction TestTabCroissantRec : booleen
  parametres locaux
    entier[]  tab
    entier    len
debut
si (len == 0)
  retourner (Vrai)
sinon si (tab[len - 1] > tab[len])
  retourner (Faux)
sinon
  retourner (TestTabCroissantRec(tab, len - 1))
fin si
fin algorithme fonction TestTabCroissantRec
```

Si on a atteind le début de la chaîne (si on a consommé tous les caractères), alors le tableau était bien trié. Si un élément précédent est plus grand que son suivant, alors le tableau n'est pas trié. Dans tous les autres cas, on avance vers la début de la chaîne

41) Écrivez une fonction qui insère un élément dans un tableau à une position précise, et décale les éléments vers la fin. Le dernier élément qui devrait disparaître du tableau sera renvoyé par la fonction. Par exemple, pour un tableau contenant [A B C D], si l'on y insère 'Z' en position 1, le tableau doit devenir [A Z B C] et la fonction doit renvoyer D. *InsertionTab(tab, len, elt, pos)*

```
algorithme fonction InsertionTabV1 : entier
  parametres locaux
    entier[]  tab
    entier    len, pos
    entier    elt
  variables
    entier    i
    entier    var
debut
var = tab[len - 1]
i = len - 1
tant que (i != pos)
  tab[i] = tab[i - 1]
  i = i - 1
fin tant que
tab[i] = elt
retourner (var)
fin algorithme fonction InsertionTabV1
```

On sauvegarde tout d'abord l'élément qui va être expulsé/écrasé (c'est facile : c'est le dernier). On va ensuite décaler chaque élément vers la fin du tableau, donc mettre dans chaque case le contenu de la case précédente. La condition d'arrêt sera d'atteindre la case visée : dès que l'on arrive dessus, on ne fait plus rien

```
algorithme fonction InsertionTabV1.5 : entier
  parametres locaux
    entier[]  tab
    entier    len, pos
    entier    elt
  variables
    entier    i
    entier    var
debut
var = tab[len - 1]
i = len
tant que (i != (pos + 1))
  i = i - 1
  tab[i] = tab[i - 1]
fin tant que
tab[i] = elt
retourner (var)
fin algorithme fonction InsertionTabV1.5
```

Dans cette version légèrement modifiée, on va d'abord décaler le pointeur de la case actuelle, et ensuite seulement déplacer les valeurs. On a donc un certain décalage à compenser qui risque d'aller chercher des valeurs à l'index "-1" si on veut écraser l'élément en position "0". On limite donc le nombre de décalages en bornant à "pos + 1"

```
algorithme fonction InsertionTabV2 : entier
  parametres locaux
    entier[]  tab
    entier    len, pos
    entier    elt
  variables
    entier    i
    entier    tmp
debut
i = pos
tant que (i < len)
  tmp = tab[i]
  tab[i] = elt
  elt = tmp
  i = i + 1
fin tant que
retourner (elt)
fin algorithme fonction InsertionTabV2
```

Cette version effectue des copies successives en exploitant une variable : on prend le contenu de la case courante, on le stocke dans une variable temporaire, on écrit l'élément manipulé, puis, on considère l'élément de la variable temporaire en tant qu'élément manipulé. Ainsi, chaque élément du tableau au delà de la position visée devient un élément manipulé, et le dernier qui devrait être exclu devient manipulé à son tour, et donc, il est renvoyé. Dit d'une autre façon : on "swap"/échange d'une certaine façon les valeurs entre 2 variables au fur et à mesure de l'avancée

42) Écrivez une fonction qui supprime un élément dans un tableau à une position précise, et décale les éléments vers le début. L'élément supprimé du tableau sera renvoyé par la fonction. De plus, pour éviter que le dernier élément soit dupliqué, vous prendrez un élément en paramètre qui sera inséré à la fin. Par exemple, pour un tableau contenant [A B C D], si l'on supprime l'élément en position 1 tout en ajoutant 'Z', le tableau doit devenir [A C D Z] et la fonction doit renvoyer B. *SuppressionTab(tab, len, pos, elt)*

```
algorithme fonction SuppressionTab : entier
  parametres locaux
    entier[]  tab
    entier    len, pos
    entier    elt
  variables
    entier    i
    entier    var
debut
var = tab[pos]
i = pos
tant que (i != len - 1)
  tab[i] = tab[i + 1]
  i = i + 1
fin tant que
tab[i] = elt
retourner (var)
fin algorithme fonction SuppressionTab
```

Pour supprimer un élément et pousser les éléments de la fin jusqu'à la position visée, il suffit simplement de partir de la position supprimée, et y copier l'élément de la case suivante. La dernière case contiendra donc le même élément que l'avant dernière, on y écrit donc l'élément donné en paramètre [cet ajout est purement esthétique ici]

43) Écrivez une procédure qui inverse la position de tous les éléments. Vous ne devez pas construire de nouveau tableau, mais uniquement modifier en place le tableau (en utilisant des variables temporaires). Par exemple, pour un tableau contenant [A B C D], si l'on inverse la position des éléments, le tableau doit devenir [D C B A]. *InverserTab(tab, len)*

```
algorithme procedure InverserTabV1
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier    i
    entier    temp
debut
pour i = 0 jusqu'a ((len / 2) - 1)
  temp = tab[i]
  tab[i] = tab[len - 1]
  tab[len - 1] = temp
  len = len - 1
fin pour
fin algorithme procedure InverserTabV1
```

Pour inverser le tableau, on le parcoure uniquement jusqu'à la moitié des éléments avec un premier itérateur, on réduit le paramètre "longueur" de 1 à chaque tour afin de rester symétrique, et on inverse les éléments en les stockant dans une variable temporaire

```
algorithme procedure InverserTabV2
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier    i
    entier    tmp
debut
i = 0
tant que (i < (len / 2))
  tmp = tab[i]
  tab[i] = tab[len - 1 - i]
  tab[len - 1 - i] = tmp
  i = i + 1
fin tant que
fin algorithme procedure InverserTabV2
```

Le fonctionnement est parfaitement similaire, mais au lieu de mettre un second itérateur gardant le symétrique, on le déduit en faisant une soustraction entre le numéro de la dernière case et la position de l'itérateur courant (plus il grandit, plus la soustraction produit un résultat petit)

44) Écrivez une fonction qui vérifie sur une chaîne de caractères est bien un préfixe d'une autre chaîne de caractères. Cette fonction renvoie *vrai* si le prefixe est bon et *faux* si ce n'est pas le cas. Par exemple, "abc" est un préfixe à "abcdef", mais pas "bcd" ni "def". Vous ferez une version itérative et une version récursive. *PrefixStrIter(str, prefix) PrefixStrRec(str, prefix)*

```
algorithme fonction PrefixIterStrV1 : entier
  parametres locaux
    string    str, prefix
debut
si (SubStr(str, prefix) == 0)
  retourner (0)
sinon
  retourner (-1)
fin si
fin algorithme fonction PrefixIterStrV1
```

Version "ultra simple" dépendant de SubStr

```
algorithme fonction PrefixIterStrV2 : entier
  parametres locaux
    string    str, prefix
  variables
    entier    lenStr, lenPrefix
    entier    i, a
debut
lenStr = strlen(str)
lenPrefix = strlen(prefix)
si (lenPrefixe > lenStr)
  retourner (-1)
fin si
tant que (i < lenPrefix)
  si (str[i] == prefix[i]
    a = a + 1
  fin si
  si (a == lenPrefix)
    retourner (0)
  fin si
  i = i + 1
fin tant que
retourner (-1)
fin algorithme fonction PrefixIterStrV2
```

Si le préfixe est plus grand que la chaîne, on ne cherche rien de plus. Ensuite, on va chercher exclusivement sur les N premiers caractères (la taille du préfixe) de la chaîne principale si les caractères sont bien reconnus. Au moindre problème, le préfixe n'est donc pas reconnu, donc n'est pas un préfixe de la chaîne

```
algorithme fonction PrefixIterStrV3 : entier
  parametres locaux
    string    str, prefix
  variables
    entier    len, i
debut
len = strlen(prefix)
pour i = 0 jusqu'a len
  si (str[i] != prefix[i])
    retourner (-1)
  fin si
fin pour
retourner (0)
fin algorithme fonction PrefixIterStrV3
```

Version encore plus courte : on fait évoluer un itérateur jusqu'à la fin du préfixe, si on l'atteind alors le préfixe était bien là, sinon, lorsque les caractères du préfixe et de la chaîne varient, on quitte avec une erreur

45) Écrivez une fonction qui vérifie sur une chaîne de caractères est bien un suffixe d'une autre chaîne de caractères. Cette fonction renvoie *vrai* si le suffixe est bon et *faux* si ce n'est pas le cas. Par exemple, "def" est un suffixe à "abcdef", mais pas "cde" ni "abc". Vous ferez une version itérative et une version récursive. *SuffixStrIter(str, suffix) SuffixStrRec(str, suffix)*

```
algorithme fonction SuffixStrIter : entier
  parametres locaux
    string    str, suffix
  variables
    entier    lenStr, lenSuffix
debut
lenStr = strlen(str) - 1
lenSuffix = strlen(suffix) - 1
si (lenStr < lenSuffix)
  retourner (-1)
fin si
tant que (lenSuffix > 0)
  si (str[lenStr] == suffix[lenSuffix])
    lenSuffix = lenSUffix - 1
    lenStr = lenStr - 1
  sinon
    retourner (-1)
  fin si
fin tant que
retourner (lenStr)
fin algorithme fonction SuffixStrIter
```

On élimine d'abord les cas où le sufixe est plus long que la chaîne principale. Ensuite, on va reculer dans le suffixe et la chaîne, à la moindre différence entre le suffixe et la chaîne, on quitte. Si on arrive à consommer l'intégralité du suffixe, alors c'est que celui-ci a été reconnu

```
algorithme fonction SuffixStrRec : entier
  parametres locaux
    string    str, suffixe
  variables
    entier    lenStr, lenSuffixe
debut
si (lenSuffixe == 0)
  retourner (lenStr)
sinon si (str[lenStr - 1] != suffixe[lenSuffix - 1])
  retourner (-1)
sinon
  retourner (StrSuffixeRec(str, suffixe, lenStr - 1, lenSuffixe - 1))
fin sifin algorithme fonction SuffixStrRec

algorithme fonction SuffixStrRecChapo : entier
  parametres locaux
    string    str, suffixe
  variables
    entier    lenStr, lenSuffixe
debut
lenStr = strlen(str)
lenSuffixe = strlen(suffixe)
si (lenSuffixe > lenStr)
  retourner (-1)
sinon
  retourner (StrSuffixeRec(str, suffixe, lenStr, lenSuffixe))
fin si
fin sifin algorithme fonction SuffixStrRecChapo
```

On élimine d'abord les cas où le sufixe est plus long que la chaîne principale. Ensuite, on va reculer dans le suffixe et la chaîne, à la moindre différence entre le suffixe et la chaîne, on quitte. Si on arrive à consommer l'intégralité du suffixe, alors c'est que celui-ci a été reconnu

46) Écrivez une fonction qui vérifie sur une chaîne de caractères est contenue dans une autre chaîne de caractères. Cette fonction renvoie *vrai* si la sous-chaîne est contenue dans la chaîne principale et *faux* si ce n'est pas le cas. Par exemple, "abc" est contenue dans "ababc", mais pas "cde" ni "cba". Vous ferez une version itérative et une version récursive. *SubStrIter(str, sub) SubStrRec(str, sub)*

Attention, certains cas sont difficiles à détecter. Dans certains cas, il sera plus difficile de détecter "abc" dans "abcbc" que "abc" dans "ababc". N'oubliez pas de vérifier plusieurs cas complexes tels que : rechercher "abc" dans "abababc" ou "abcbcbc".

```
algorithme fonction SubStrIterFAUXv1 : entier
  parametres locaux
    string    str, sub
debut
si (strlen(str) < strlen(sub))
  retourner (-1)
fin si
i = 0
a = 0
tant que (i < strlen(str))
  si (str[i] == sub[a]) et (a < strlen(sub))
    a = a + 1
  sinon
    a = 0
  fin si
  si (a == strlen(sub))
    retourner (i - a - 1)
  fin si
  i = i + 1
fin tant que
retourner (-1)
fin algorithme fonction SubStrIterFAUXv1
```

Cette version naïve NE FONCTIONNE PAS mais est capable de chercher une sous-chaîne qui ne ressemble pas au reste de la chaîne.

```
algorithme fonction SubStrIterFAUXv1.1 : entier
  parametres locaux
    string    str, sub
debut
si (strlen(str) < strlen(sub))
  retourner (-1)
fin si
i = 0
a = 0
tant que (i < strlen(str))
  si (str[i] == sub[a]) et (a < strlen(sub))
    a = a + 1
  sinon
    a = 0
    continue
  fin si
  si (a == strlen(sub))
    retourner (i - a - 1)
  fin si
  i = i + 1
fin tant que
retourner (-1)
fin algorithme fonction SubStrIterFAUXv1.1
```

Cette version naïve "bis" NE FONCTIONNE TOUJOURS PAS mais est capable de chercher une sous-chaîne tant qu'un décalage suffisament grand existe entre la sous-chaîne et le contenu de la chaîne. On élimine tout d'abord les cas où la sous-chaîne est plus grande que la chaîne. Ensuite, on va rechercher chaque caractère de la sous-chaîne au fur et à mesure que l'on avance dans la chaîne principale (avec i). Lorsque l'on retrouve le 1er caractère de la sous-chaîne, on va augmenter un compteur dédié à la sous-chaîne (a). Si on tombe sur des caractères différents entre la chaîne et la sous-chaîne, on remet le compteur de la sous-chaîne à 0 pour rechercher de nouveau le premier caractère. On effectue un "continuer" qui est une instruction spéciale sautant toutes les instructions restantes dans la boucle, pour refaire une itération. Ici, on l'utilise pour pouvoir re-tester le début de la sous-chaîne immédiatement.

```
algorithme fonction SubStrRecFAUXv1 : entier
  parametres locaux
    string    str, sub
    entier    lenStr, lenSub
debut
si (lenSub == 0)
  retourner (lenStr)
sinon si (lenStr == 0)
  retourner (-1)
sinon si (str[lenStr - 1] == sub[lenSub - 1])
  retourner (SubStrRec(str, sub, lenStr - 1, lenSub - 1))
sinon
  retourner (SubStrRec(str, sub, lenStr - 1, strlen(sub)))
fin si
fin algorithme fonction SubStrRecFAUXv1

algorithme fonction SubStrRecChapoFAUXv1 : entier
  parametres locaux
    string    str, sub
debut
  retourner (SubStrRec(str, sub, strlen(str), strlen(sub), strlen(sub)))
fin algorithme fonction SubStrRecChapoFAUXv1
```

Cette version naïve NE FONCTIONNE PAS mais est capable de chercher une sous-chaîne tant qu'un décalage suffisament grand existe entre la sous-chaîne et le contenu de la chaîne. On teste d'abord le cas qui fonctionne où la sous-chaîne a été complètement consommée, car ce cas est plus spécifique (str ET sub ont atteint leur fin dans le cas où la sous-chaîne termine la chaîne). Puis on teste le cas où la chaîne est terminée seule mais pas la sous-chaîne (donc on n'a pas trouvé de sous-chaîne). Puis on teste les cas plus génériques : si on est sur le même caractère, on avance, si on a une différence de caractères, on redémarre le compteur à 0 en reprenant sur le caractère en cours

```
algorithme fonction SubStrRecFAUXv1.1 : entier
  parametres locaux
    string    str, sub
    entier    lenStr, lenSub
debut
si (lenSub == 0)
  retourner (lenStr)
sinon si (lenStr == 0)
  retourner (-1)
sinon si (str[lenStr - 1] == sub[lenSub - 1])
  retourner (SubStrRec(str, sub, lenStr - 1, lenSub - 1))
sinon
  si (lenSub != maxLenSub)
    retourner (SubStrRec(str, sub, lenStr, maxLenSub))
  sinon
    retourner (SubStrRec(str, sub, lenStr - 1, maxLenSub))
  fin si
fin si
fin algorithme fonction SubStrRecFAUXv1.1

algorithme fonction SubStrRecChapoFAUXv1.1 : entier
  parametres locaux
    string    str, sub
debut
  retourner (SubStrRec(str, sub, strlen(str), strlen(sub), strlen(sub)))
fin algorithme fonction SubStrRecChapoFAUXv1.1
```

Cette version naïve "bis" NE FONCTIONNE TOUJOURS PAS mais est capable de chercher une sous-chaîne tant qu'un décalage suffisament grand existe entre la sous-chaîne et le contenu de la chaîne. Pour affiner/améliorer et prendre en compte le "continue" qui n'était pas dans la version itérative initiale, on déifférencie le cas où l'on vient "juste" de tomber sur un caractère différent (on reprend au début de la sous-chaîne en restant sur le même caractère de la chaîne principale), du cas où l'on n'a pas encore trouvé le début de la sous-chaîne (on n'a pas consommé de caractère dans la sous-chaîne, donc on avance au caractère suivant de la chaîne)

7 Tris

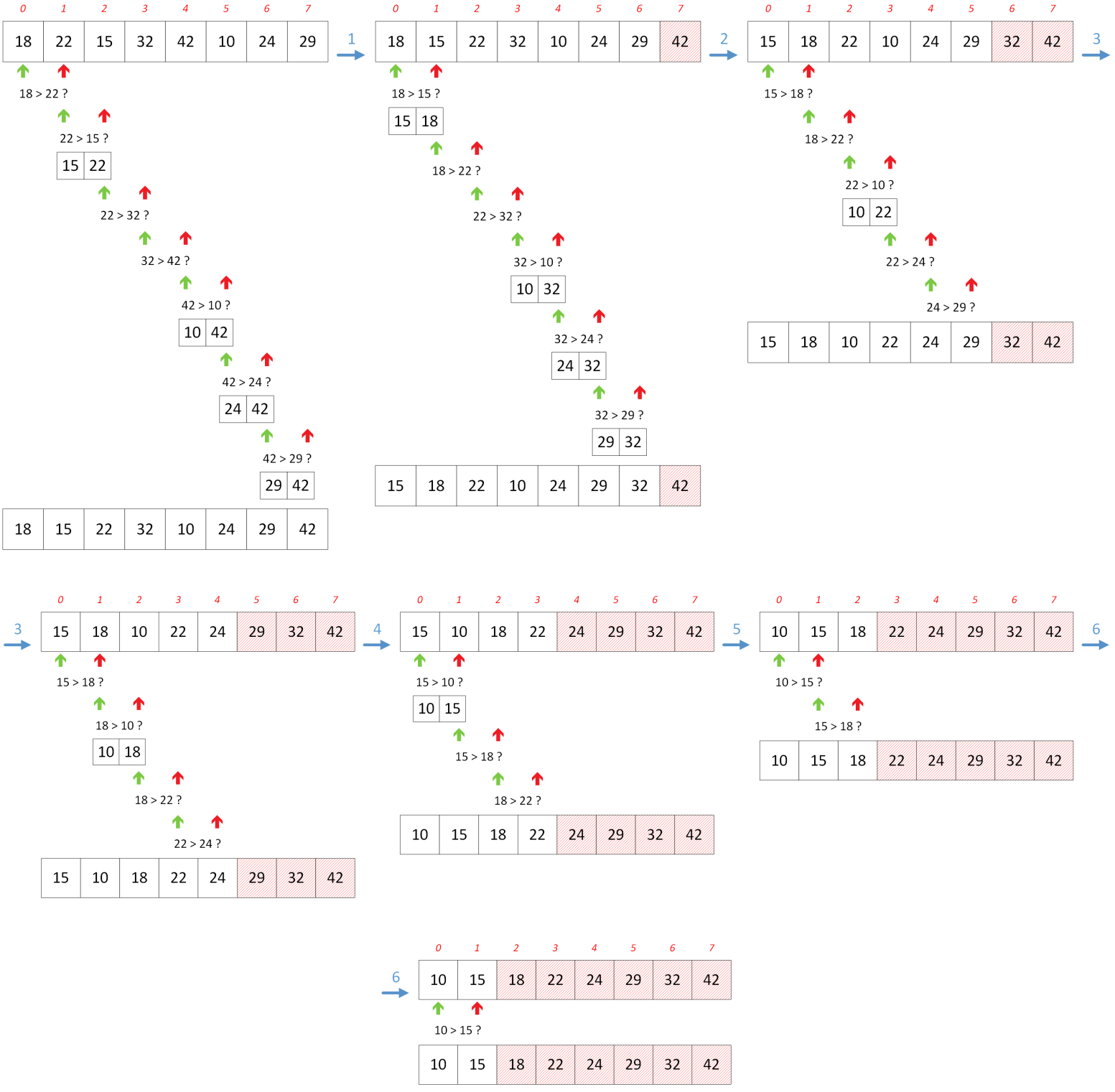
47) Écrivez une fonction qui inverse la position de deux éléments. Vous ne devez pas construire de nouveau tableau, mais uniquement modifier en place le tableau (en utilisant des variables temporaires). Par exemple, pour un tableau contenant [A B C D], si l'on inverse la position des éléments 0 et 1, le tableau doit devenir [B A C D]. *SwapEltTab(tab, len, pos1, pos2)*

```
algorithme fonction SwapEltTabV1 : booleen
  parametres locaux
    entier[]  tab
    entier    len, pos1, pos2
  variables
    entier    var
debut
si (pos1 > (len - 1)) ou (pos2 > (len - 1))
  retourner (Faux)
sinon si (pos1 < 0) ou (pos2 < 0)
  retourner (Faux)
sinon
  var = tab[pos1]
  tab[pos1] = tab[pos2]
  tab[pos2] = var
  retourner (Vrai)
fin si
fin algorithme fonction SwapEltTabV1
```

```
algorithme fonction SwapEltTabV2 : booleen
  parametres locaux
    entier[]  tab
    entier    len, pos1, pos2
  variables
    entier    var
debut
si (pos1 >= 0) et (pos1 < len) et
   (pos2 >= 0) et (pos2 < len)
  var = tab[pos1]
  tab[pos1] = tab[pos2]
  tab[pos2] = var
  retourner (Vrai)
sinon
  retourner (Faux)
fin si
fin algorithme fonction SwapEltTabV2
```

7.1 Tri à bulles

Le tri à bulles vise à faire remonter tour à tour les plus grandes valeurs vers la fin du tableau. Si deux valeurs côté à côté sont dans le désordre, on les inverse, et on teste les suivantes jusqu'à la fin du tableau. Avec cette méthode, dès que l'on trouve la plus grande valeur du tableau, celle-ci avancera progressivement jusqu'à la fin. Avant d'atteindre la plus grande valeur, on peut également faire remonter les autres valeurs plus grandes. Cependant, à chaque fois que l'on parcourt intégralement le tableau de gauche à droite, on est sûr que la valeur la plus à droite est à sa place finale. Ainsi, on réduit la zone à parcourir à chaque tour : une fois la plus grande valeur du tableau trouvée à la fin du premier tour et placée à la dernière case, on n'a plus besoin de chercher dans cette case (l'élément étant maintenant à sa place).



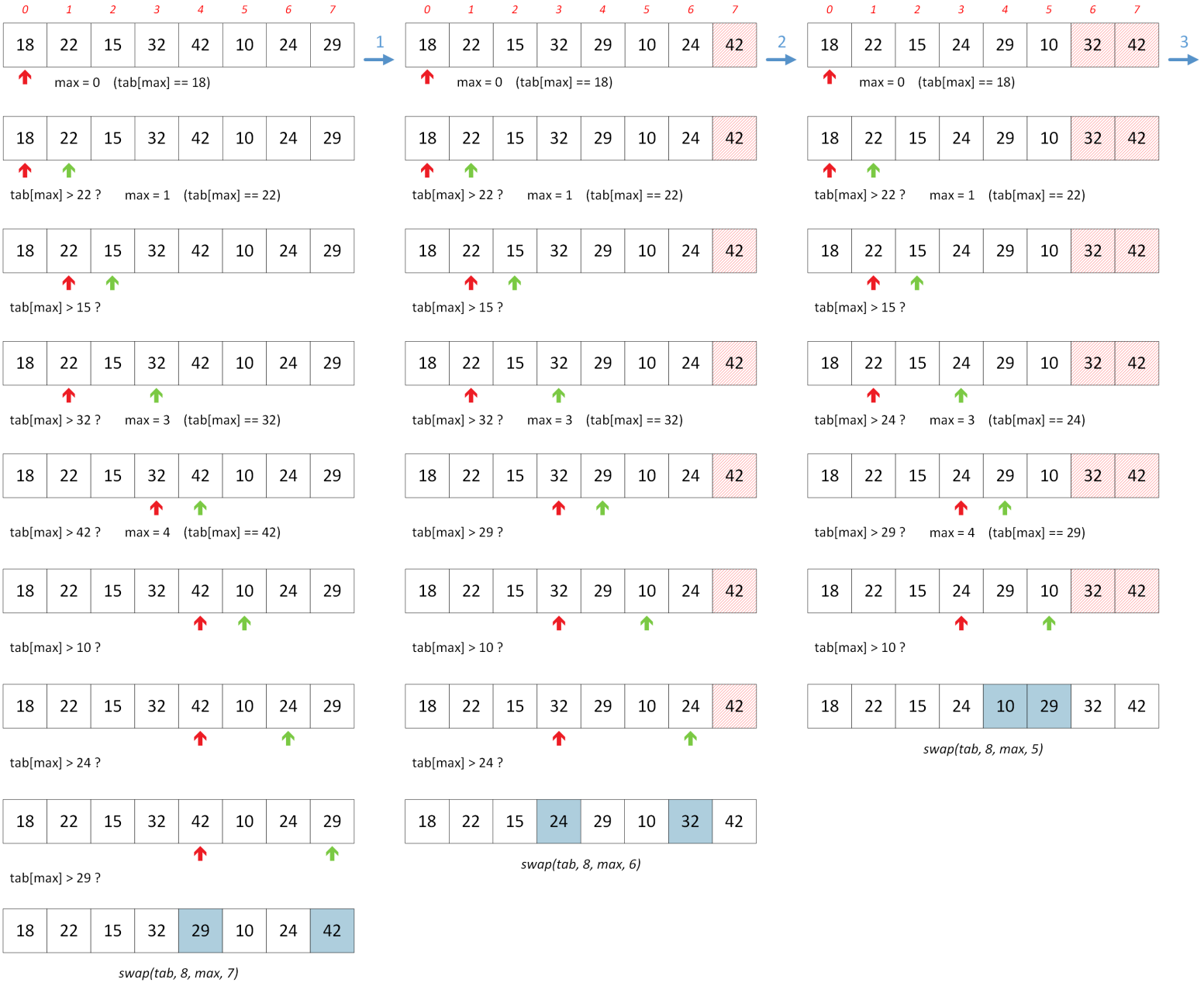
48) Écrivez une procédure de tri respectant l'algorithme du tri à bulles (ou *bubble sort* en anglais). Vous devez modifier la position des éléments dans le tableau créer de tableau supplémentaire. *BubbleSort(tab, len)*

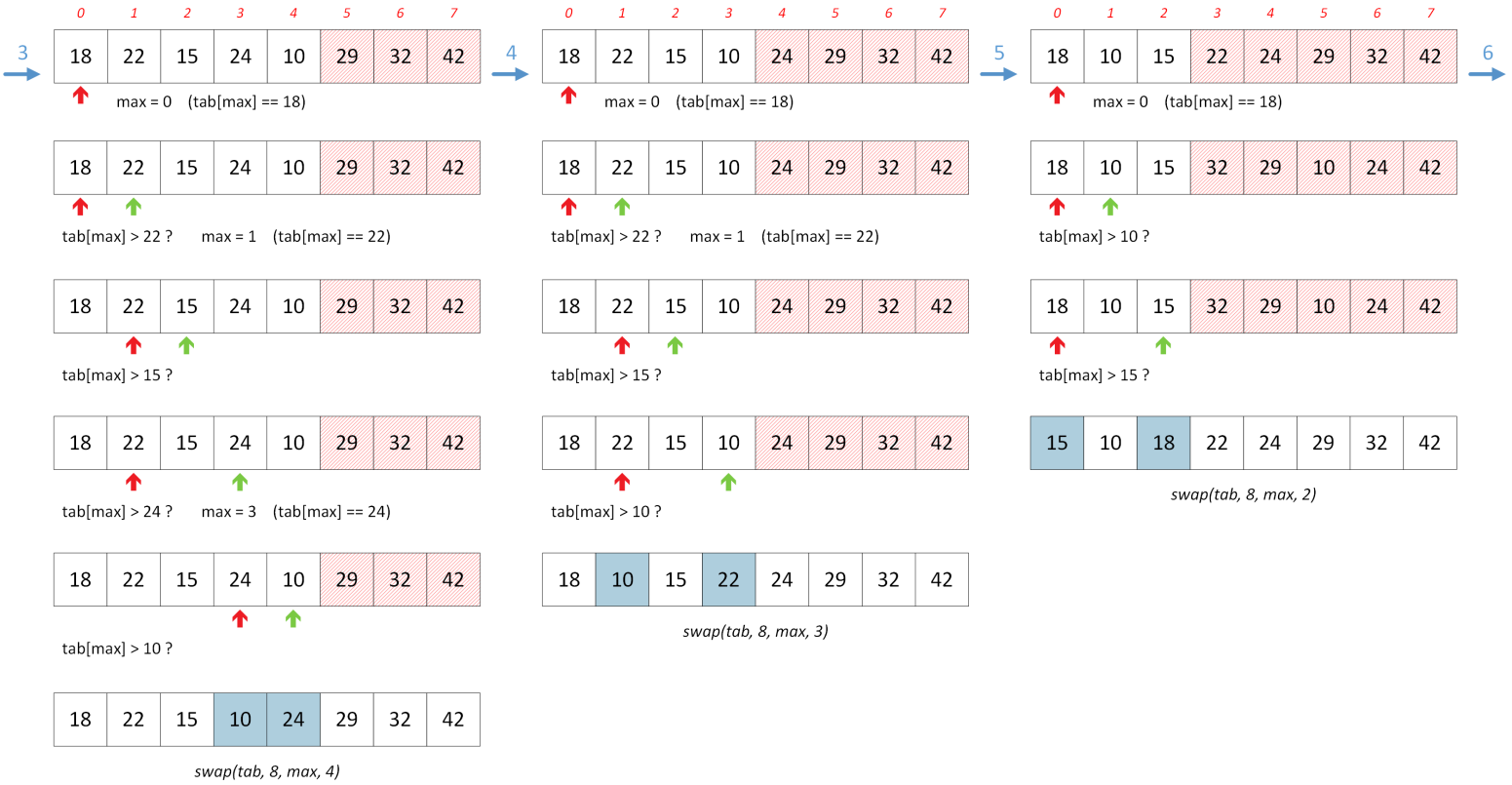
```
algorithme procedure BubbleSort
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier    i, j
debut
pour i = (len - 1) jusqu'a 1
  pour j = 0 jusqu'a (i - 1)
    si (tab[j] > tab[j + 1])
      swap(tab, len, j, j + 1)
    fin si
  fin pour
fin pour
fin algorithme procedure BubbleSort
```

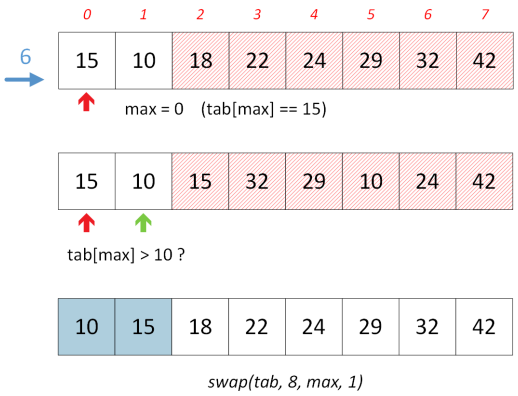
Dans cette version du Bubble Sort, on va chercher à pousser les plus gros éléments à la fin à chaque fois en faisant avancer chaque élément plus grand vers la droite. Une fois que le plus grand élément a été trouvé, il est naturellement poussé vers la droite. Une fois que ce plus grand élément est poussé à la fin du tableau, plus aucun élément ne sera plus grand, donc inutile de les comparer avec. Ainsi, la dernière case la plus à droite devient immuable. La première boucle avec i permet de déterminer quelle colonne va servir de fin pour les balayages successifs, et à chaque itération, on n'ira pas plus loin. La boucle utilisant j permet d'effectuer les balayages successifs et d'échanger la position entre un élément courant et son suivant s'il est plus petit. Ainsi, en tombant sur le plus grand élément, on va le déplacer progressivement jusqu'à la fin du tableau. On recommence avec les éléments restants, donc avec le 2e plus grand, et ainsi de suite.

7.2 Tri par sélection

Le tri par sélection est le tri le plus évident humainement parlant : on cherche le plus grand élément dans le tableau en parcourant toutes les cases, puis, on échange la place de cet élément avec le tout dernier du tableau (pour le placer à sa place définitive). On recommence ce traitement (sans lire la dernière case à chaque fois) jusqu'à avoir ordonné tous les éléments.




3
4
5
6
0 1 2 3 4 5 6 7
18 22 15 24 10 29 32 42
max = 0 (tab[max] == 18)
tab[max] > 22 ? max = 1 (tab[max] == 22)
tab[max] > 15 ?
tab[max] > 24 ? max = 3 (tab[max] == 24)
tab[max] > 10 ?
18 22 15 10 24 29 32 42
max = 0 (tab[max] == 18)
tab[max] > 22 ? max = 1 (tab[max] == 22)
tab[max] > 15 ?
tab[max] > 10 ?
18 10 15 22 24 29 32 42
max = 0 (tab[max] == 18)
tab[max] > 10 ?
tab[max] > 15 ?
15 10 18 22 24 29 32 42
swap(tab, 8, max, 2)
18 10 15 22 24 29 32 42
swap(tab, 8, max, 3)
18 22 15 10 24 29 32 42
swap(tab, 8, max, 4)


6
0 1 2 3 4 5 6 7
15 10 18 22 24 29 32 42
max = 0 (tab[max] == 15)
15 10 15 32 29 10 24 42
tab[max] > 10 ?
10 15 18 22 24 29 32 42
swap(tab, 8, max, 1)

49) Écrivez une procédure de tri respectant l'algorithme du tri par sélection (ou *selection sort* en anglais). Vous devez modifier la position des éléments dans le tableau sans créer de tableau supplémentaire. *SelectionSort(tab, len)*

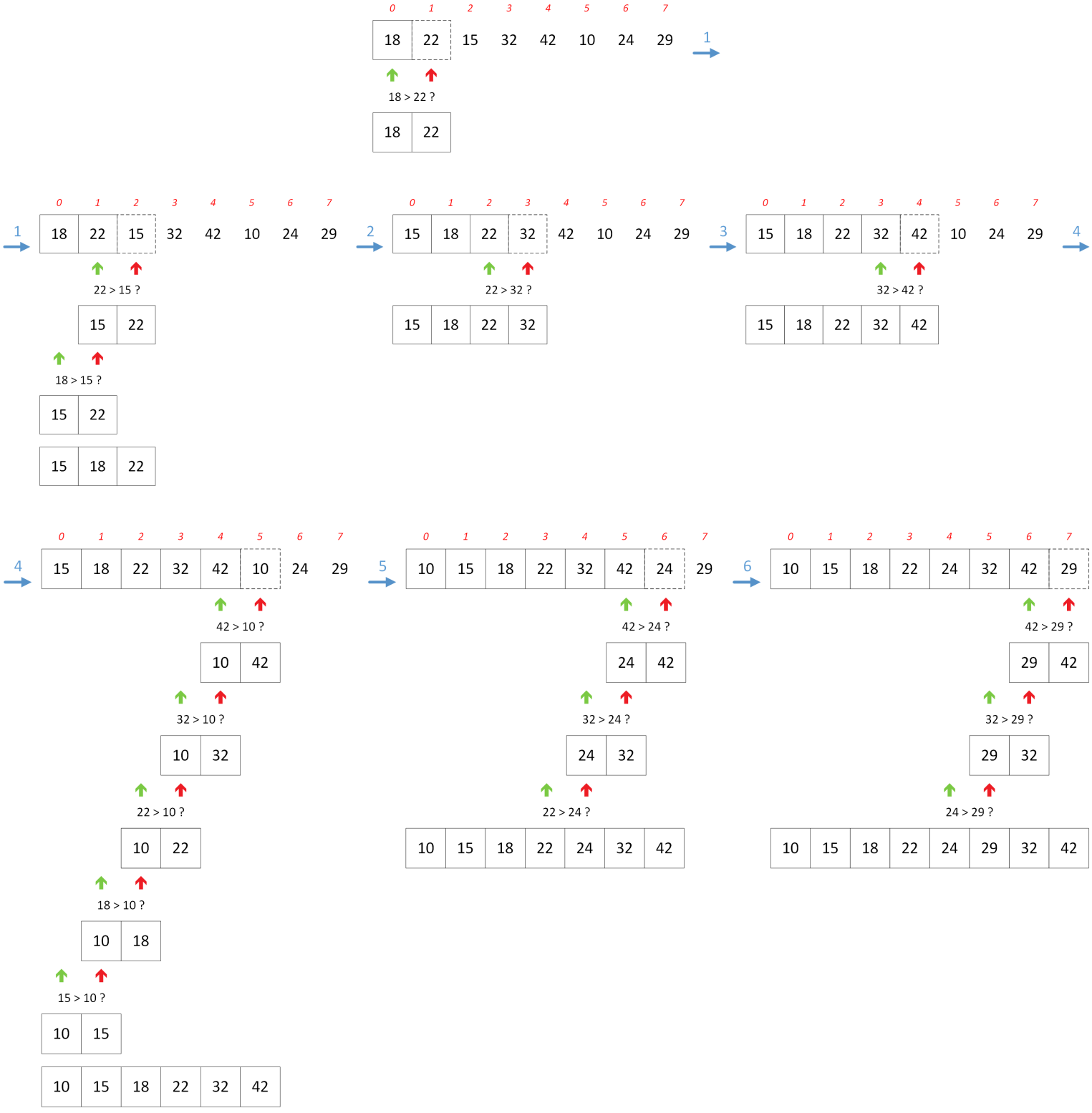
```
algorithme procedure SelectionSort
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier    max_range, max
debut
pour max_range = (len - 1) jusqu'a 2
  max = 0
  pour i = 1 jusqu'a (max_range)
    si (tab[i] > tab[max])
      max = i
    fin si
  fin pour
  swap(tab, len, max, max_range)
fin pour
fin algorithme procedure SelectionSort
```

Cas général permettant de parcourir le tableau plusieurs fois. À chaque parcours, on cherche la valeur la plus grande, et on la place à la fin. Étant donné que le plus grand élément est placé en fin, on va réduire le parcours d'un élément à chaque fois. Ainsi, le premier parcours fera la longueur du tableau, le dernier parcours ne fera que comparer les 2 premiers éléments pour placer correctement les deux plus petits éléments du tableau.
Cet algorithme a l'inconvénient de nécessiter au moins 3 éléments dans le tableau pour fonctionner correctement. Ainsi, on peut soit faire une fonction chapeau testant les cas où la longueur est inférieure à 3, soit ajouter un pré-traitement dans la fonction.

```
algorithme procedure PreProcessingSelectionSort
  parametres locaux
    entier[]  tab
    entier    len
debut
si (len >= 2)
  si (len == 2)
    si (tab[0] > tab[1])
      swap(tab, len, 0, 1)
    fin si
  sinon
    SelectionSort(tab, len)
  fin si
fin si
fin algorithme procedure PreProcessingSelectionSort
```

7.3 Tri par insertion

Le tri par insertion considère que le tableau donné en paramètre n'est pas trié, et seuls les éléments qu'il a manipulé successivement le sont. Ainsi, chaque élément du tableau est comparé à tous ceux déjà triés, et on le place là où il devrait être. Pour le premier élément, celui-ci est considéré comme déjà trié, on peut donc passer au deuxième. Le deuxième est comparé avec l'unique élément déjà trié : on échange leurs deux places si nécessaire. Le troisième élément est comparé au plus grand des deux éléments triés, s'il est plus grand ou égal, il reste à sa place, sinon on le décale vers la gauche d'un cran, et on le compare à l'élément suivant (et ainsi de suite). Chaque élément du tableau est donc *inséré* à sa place parmi les éléments considérés comme triés.



50) Écrivez une procédure de tri respectant l'algorithme du tri par insertion (ou *insertion sort* en anglais). Vous devez modifier la position des éléments dans le tableau créer de tableau supplémentaire. *InsertionSort(tab, len)*

```
algorithme procedure InsertionSort
  parametres locaux
    entier[]  tab
    entier    len
  variables
    entier    max_range, elt
debut
pour max_range = 1 jusqu'a (len - 1)
  elt = max_range
  tant que (elt > 0) et (tab[elt] < tab[elt - 1])
    swap(tab, len, elt, (elt - 1))
    elt = elt - 1
  fin tant que
fin pour
fin algorithme procedure InsertionSort
```

Cette fois, l'algorithme s'appuie sur l'idée que l'on va "insérer" des valeurs successivement. Le tableau est en réalité déjà rempli, mais on va considérer qu'il est intégralement non-trié, et que l'on va petit à petit trier les éléments depuis le premier jusqu'au dernier. Pour cela, on va "reconstruire" un tableau trié en déplaçant chaque élément à la place où il devrait être. Le premier élément du tableau étant seul, il est donc trié. On peut passer au deuxième élément et le comparer à l'unique élément déjà présent : doit-il être placé avant ou après le 1er élément ? On échange éventuellement les places. Ensuite, on peut passer au troisième élément : est-il plus grand ou plus petit que le deuxième ? S'il est plus grand, il reste à sa place et on peut s'occuper du quatrième élément. S'il est plut petit, on inverse leurs positions, et on se repose la question concernant le nouvel élément et le premier : lequel est le plus grand ? Ainsi, on itère du 1er élément jusqu'au dernier, et on effectue tous les échanges nécessaires pour placer le nouvel élément au bon endroit. Lorsque l'on analyse l'algorithme, on se rend compte que les éléments "avant" celui traité sont tous ordonnés, et les éléments "après" celui traité ne sont pas encore connus.
Comme pour le tri par sélection, on doit absolument avoir un tableau de trois éléments ou plus pour pouvoir exécuter cet algorithme. Ainsi, on doit également ajouter un pré-traitement dans l'algorithme même, ou écrire une fonction chapeau.

Ce document et ses illustrations ont été réalisés par Fabrice BOISSIER en octobre 2022.
La plupart des exercices sont inspirés du cahier d'algo de Nathalie "Junior" BOUQUET et Christophe "Krisboul" BOULLAY.