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
retourne sum
fin algorithme fonction Somme
```

tour	i	sum
0		
1		
2		
3		
4		
5		
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
retourne z
fin algorithme fonction MultEgpytienne
```

tour	x	y	z
0			
1			
2			
3			

Algorithme de la multiplication égyptienne

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
retourne y
fin algorithme fonction DivEuclideQuotient
```

tour	x	y
0		
1		
2		
3		
4		
5		
6		

Algorithme du quotient de la division euclidienne

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

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

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

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 ?

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

debut









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).

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

debut









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
  variables
    entier

debut









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)*

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
  retourne (1)
sinon
  retourne (n + SommeRec(n - 1))
fin si
fin algorithme fonction SommeRec
```

appel	n
0	
1	
2	
3	
4	
5	
6	

appel	retour	total
6		
5		
4		
3		
2		
1		
0		

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.

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

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

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

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

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.

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.

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

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

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

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.]

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

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

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 :

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

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

25) 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)*

26) 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)*

27) 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)*

28) 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)*

29) 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.

30) 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 :

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

31) 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)*

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)

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

33) 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)*

34) 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)*

35) 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)*

36) 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)*

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).

37) 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)*

38) 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.

39) 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)*

40) 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)*

41) 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.

42) 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)*

43) 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)*

44) 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)*

45) 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)*

46) 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)*

47) 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)*

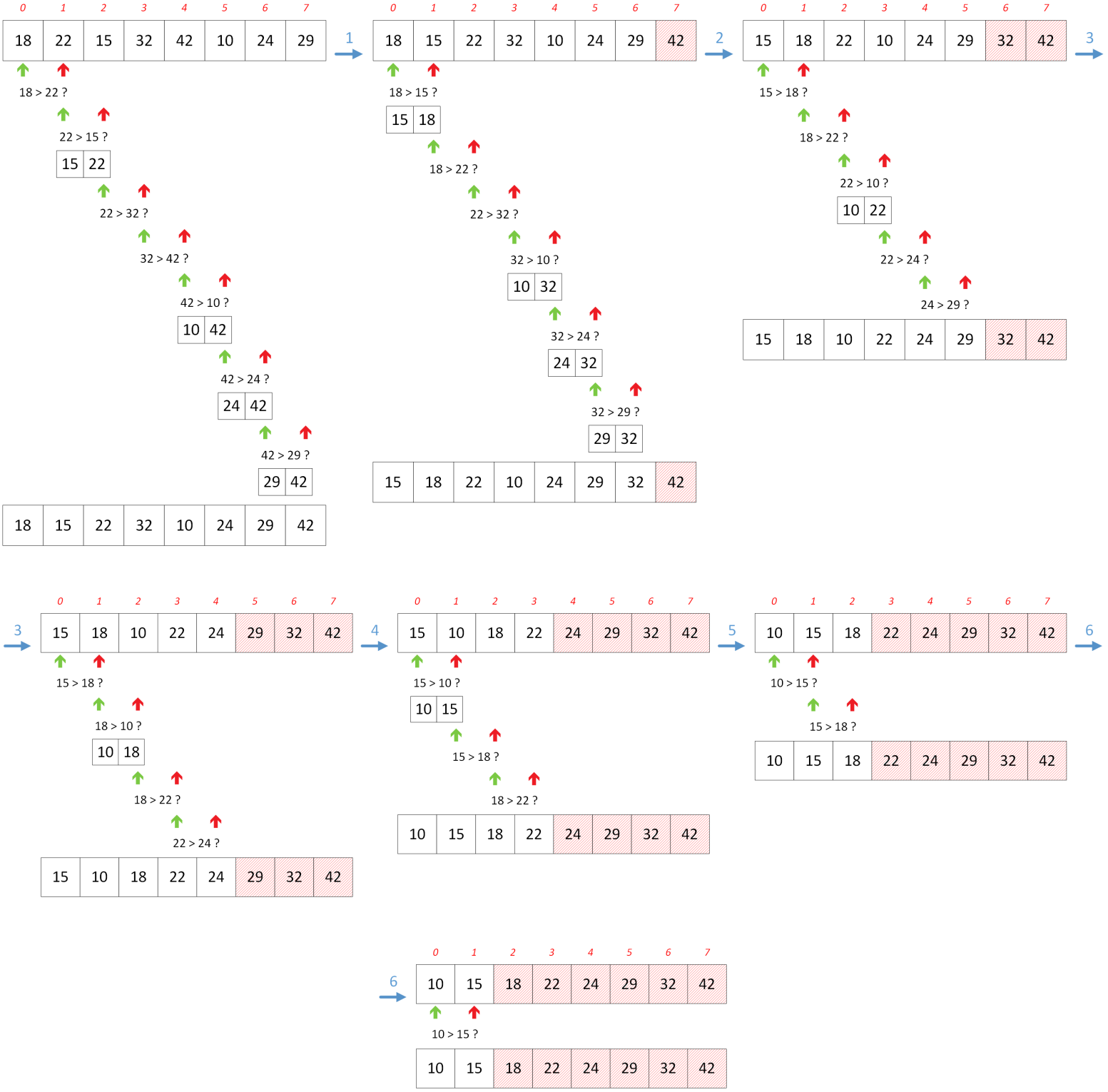
Attention, certains cas sont difficiles à détecter. Dans certains cas, il sera plus difficile de détecter "abc" dans "abcbc" que "abc" dans "ababc". N'oubliez pas de vérifier plusieurs cas complexes tels que : rechercher "abc" dans "abababc" ou "abcbcbc".

7 Tris

48) 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)*

7.1 Tri à bulles

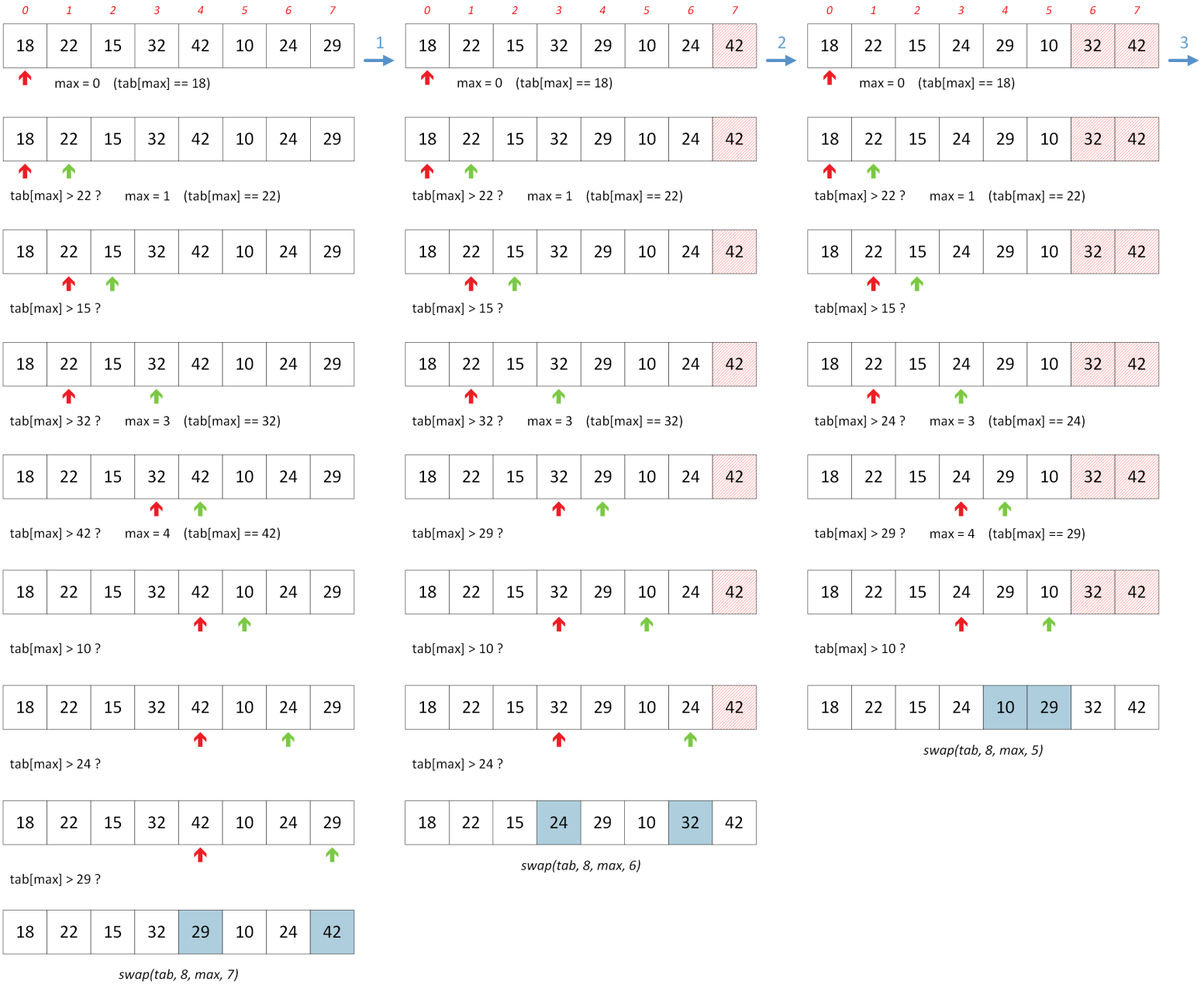
Le tri à bulles vise à faire remonter tour à tour les plus grandes valeurs vers la fin du tableau. Si deux valeurs côté à côté sont dans le désordre, on les inverse, et on teste les suivantes jusqu'à la fin du tableau. Avec cette méthode, dès que l'on trouve la plus grande valeur du tableau, celle-ci 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).

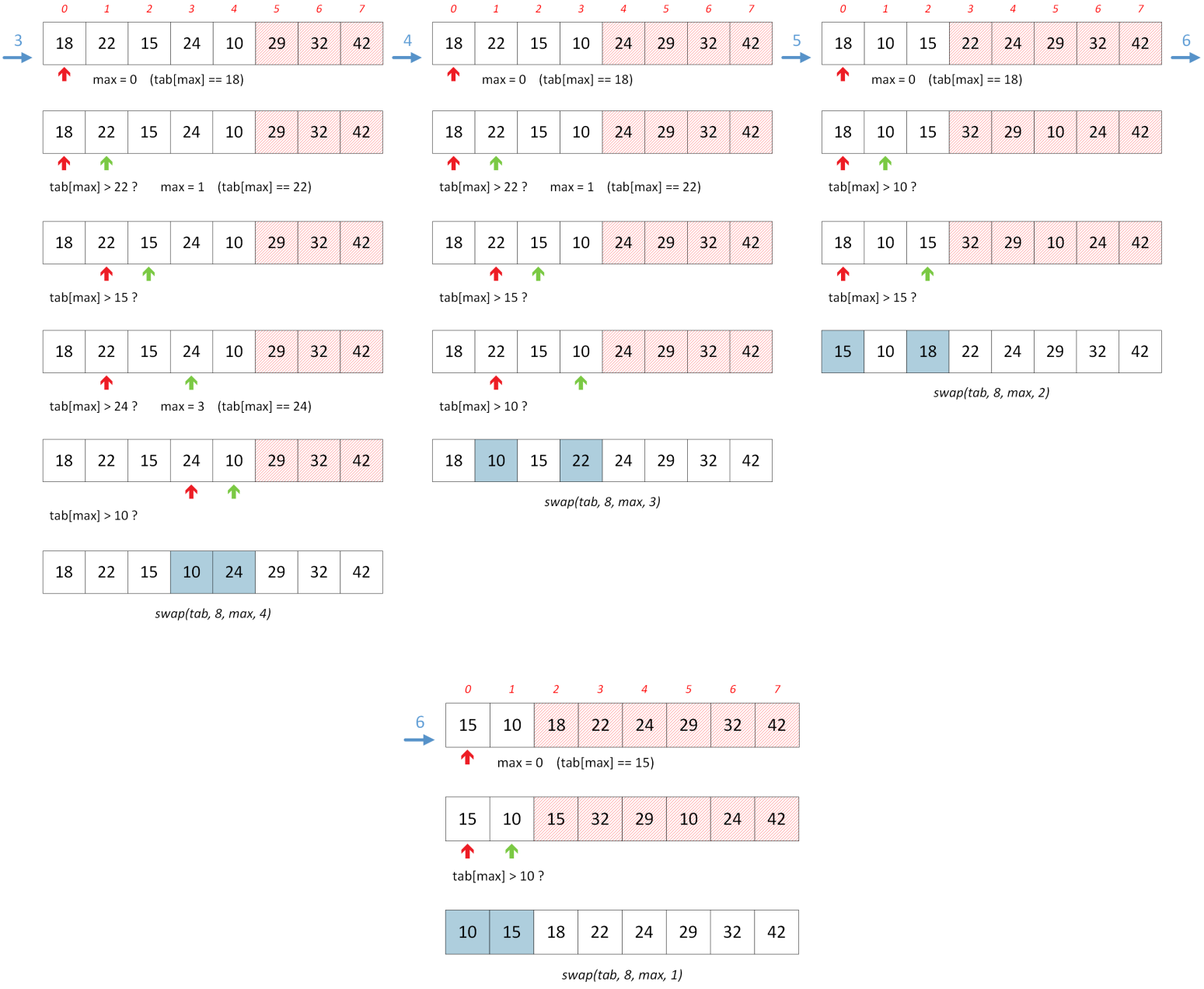


49) Écrivez une procédure de tri respectant l'algorithme du tri à bulles (ou *bubble sort* en anglais). Vous devez modifier la position des éléments dans le tableau créer de tableau supplémentaire. *BubbleSort(tab, len)*

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.

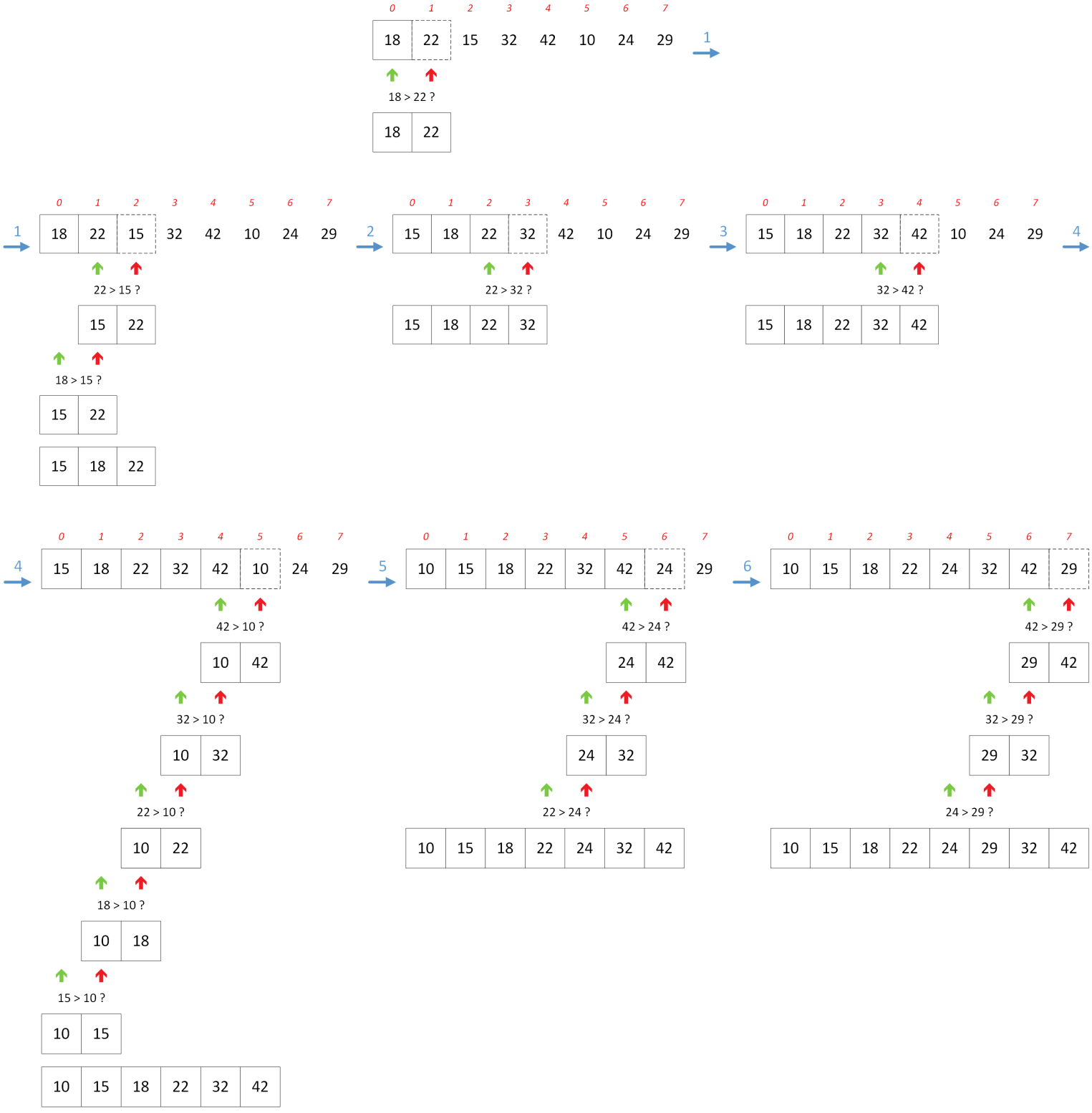




50) Écrivez une procédure de tri respectant l'algorithme du tri par sélection (ou *selection sort* en anglais). Vous devez modifier la position des éléments dans le tableau sans créer de tableau supplémentaire. *SelectionSort(tab, len)*

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.



51) 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)*

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.