Chapitre 5 : Algorithmique (Part 1)

I. Introduction à l'algorithmique

1) Brève histoire de l'algorithmique

Dans l'imaginaire collectif, le mot « algorithme » est toujours associé à l'informatique. Toutefois, il existe une histoire complète et complexe de l'algorithmique, pré-existante à l'informatique.

La notion d'algorithme est très ancienne et pourrait être reliée au mode de raisonnement « casuistique » de la civilisation babylonienne (environ 1800 av. J.-C.) où chaque loi (conclusion) procède de conditions "simples" :

« Si un homme, avec le visage congestionné, a son œil droit proéminent : loin de chez lui, des chiens le dévoreront ».

« Si des gouttes d'huile versées dans une coupe remplie d'eau dérivent vers la gauche, le malade pour lequel on accomplit la divination mourra, si elles dérivent vers la droite il guérira. »

Ces "recettes" n'avaient bien sur pas valeur générale ! On avait donc une recette, un algorithme, à suivre scrupuleusement dans chaque cas rencontré.

Les Grecs ont développé leur vision mathématique autour de démonstrations et de constructions géométriques. Ces constructions reposaient sur la notion de recette à appliquer. Toutefois, à la différence des Babyloniens, ces "recettes" ont vocation à être général :

"Pour approcher pi, il suffit de calculer le périmètre du polygone circonscrit au cercle unité et celui du polygone inscrit. Quand le nombre de côté du polygone devient très grand, on trouve la valeur de pi."

Finalement, la notion d'algorithme est historiquement associée au nom du mathématicien perse du IXème siècle, Muhammad Ibn Mūsā al-Khuwārizmī. Celui-ci décrit la construction de certains objets mathématiques étape par étape.

L'idée d'algorithme est donc très antérieure à la création du premier ordinateur et est plutôt reliée aux mathématiques.

2) Définition

Définition :

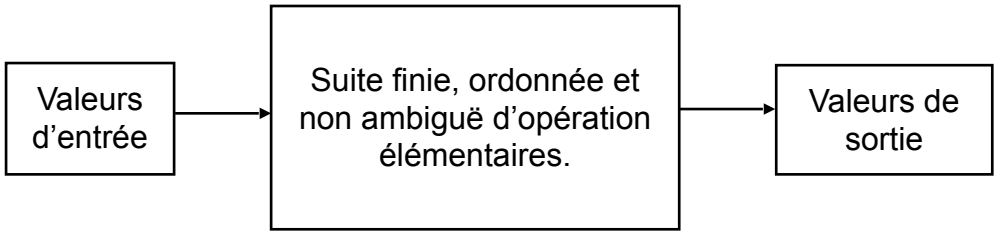
Un algorithme est une machine calculatoire réalisant une suite finie, ordonnée et non ambigüe d'instructions élémentaires permettant de résoudre une **classe** de problèmes à partir de données connues.

Remarques importantes :

- L'ordre a une importance ;
- Les instructions doivent être non ambigües (Casuistique…) ;
- Une instruction élémentaire est une **opération simple** se devant d'être **compréhensible** par un utilisateur (une opération simple pour vous est-elle nécessairement simple pour votre voisin ie. changer une chaudière à gaz est simple pour un plombier…) ;

Schéma simplificatif :

Cette définition peut se résumer comme suit :

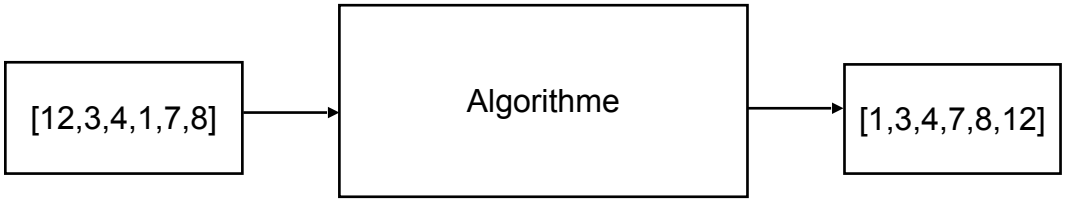


Algorithme

Exemple :

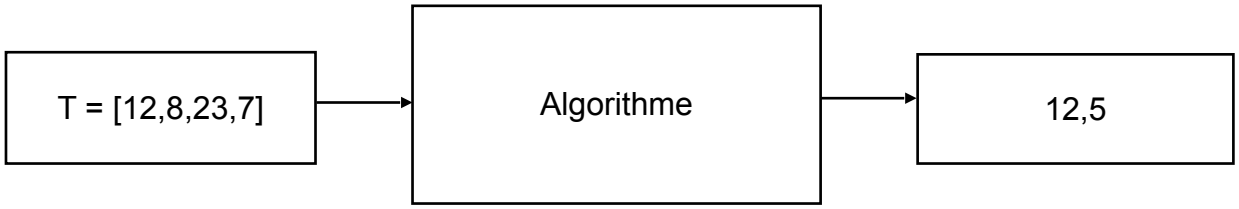
Un algorithme important en informatique est l'algorithme de tri. Nous allons étudier cette année, ainsi que l'année prochaine, différents algorithmes de tri pour les **tableaux**.

Nous avons en entrée un tableau non trié et nous obtenons en sortie un tableau trié :



II. Étude de cas

1) Étude de cas simple : calcul de la moyenne d'un tableau de nombres




```
Algorithme :

  Données
  T : tableau d'entiers
  n : entier
  moyenne : nombre à virgule

  n = taille(T)
  moyenne = 0
  pour i allant de 0 jusqu'à n-1, on fait :
    moyenne ⬅ moyenne + T[i] / n
  renvoyer moyenne
```

Définition : quand on écrit un algorithme, on utilise un langage dit « langage naturel » aussi appelé "pseudo-code" où les instructions sont issues du langage courant.

Rem : Ce langage naturel (« tant que », « si »…) permet de passer facilement à un langage de programmation respectant une syntaxe particulière (Python, Java...), on dit alors que l'on **implémente** l'algorithme.

Méthodologie :

Pour comprendre l'algorithme, on va le tester sur un cas particulier (ni trop court, ni trop long, ni trop facile, ni trop complexe). Les résultats se regroupent en général dans un tableau.

Initialisation : **T = [12, 8, 23, 7], n = 4, moyenne = 0**.
i parcourt tous les indices du tableau T, donc i va prendre pour toutes les valeurs de 0 à 3.

i	T[i] / n	moyenne
0	12/4 = 3	0+3 = 3
1	8/4 = 2	3+2 = 5
2	23/4 = 5,75	5+5,75 = 10,75
3	7/4 = 1,75	10,75+1,75 = 12,5

On renvoie 12,5 . Notre algorithme fonctionne sur cet exemple.

Exercice :
Tester l'algorithme suivant sur un exemple bien choisi :

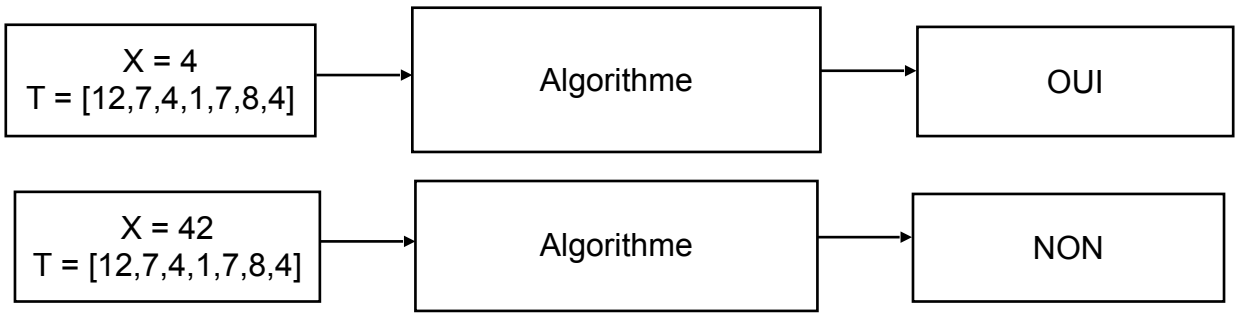
```
Données
T : tableau d'entiers
n : entier positif
moyenne : nombre à virgule

n = taille(T)
sigma = 0
pour i parcourant tous les indices du tableau T :
  sigma ⬅ sigma + (T[i]-moyenne(T))**2
renvoyer sigma/n
```

En déduire ce que calcule l'algorithme ci-dessus.

2) Étude de cas : recherche d'un entier dans une liste

Quelle est la suite d'opérations élémentaires permettant de trouver la réponse à la question :
« X est-il présent dans le tableau T ? »



Découverte de l'algo 1

T = , X =

L'idée est de passer en revue tout le tableau T

Regardez le premier element de T
Comparer à X
Si T == X
alors on marque que l'on a gagné !
Si T != X alors
on continue la recherche

Si on a passé tous les éléments en revue et qu'on n'a pas gagné, alors

Discuter avec les élèves les éléments présents dans l'algorithme.

```
Algorithme :


Données
T : tableau d'entiers
X : nombre entier
tr : booléen (VRAI ou FAUX)
i : nombre entier

i ← 0
tr ← FAUX
tant que i<taille(T) et que tr==FAUX:
  si T[i]==X:
    tr ← VRAI
  fin si
  i ← i+1
fin tant que
renvoyer la valeur de tr
```

Instruction élémentaire ?

bloc conditionnel

bloc d'itération

on a perdu!

i	i<taille(T)	tr	i<taille(T) et tr == FAUX	T[i]	T[i] == X
0	0<4 est VRAI	FAUX	VRAI	12	12 == 4 : FAUX
1	1<4 est VRAI	FAUX	VRAI	7	7 == 4 : FAUX
2	2<4 est VRAI	FAUX	VRAI	4	4 == 4 : VRAI tr devient VRAI
3	3<4 est VRAI	VRAI	FAUX	STOP	STOP

Application 1 :

1) Repérez quelle ligne permet de répondre à la question posée.
2) N'y-a-t'il que des instructions élémentaires dans cet algorithme ?

Étude d'un algorithme :

Voilà ce que cela pourrait donner avec l'algorithme que nous venons d'écrire :

T = [12,7,4,1,4]
X = 4

Remarque :
Ce tableau n'est pas unique mais permet de bien représenter les choses. Selon votre niveau de compréhension, vous pouvez réaliser plus ou moins d'étapes.

i = 1
tr = FAUX
i<=7 et tr est FAUX : **entrée dans la boucle**
t[1] (i.e. 12) n'est pas égal à 4 donc i ← i+1 (i=2)
retour au début boucle

i = 2
tr=FAUX
i<=7 et tr est FAUX : **entrée dans la boucle**
t[2] (i.e. 7) n'est pas égal à 4 donc i ← i+1 (i=3)
retour au début boucle

i = 3
tr=FAUX
i<=7 et tr est FAUX : **entrée dans la boucle**
t[3] (i.e. 4) est égal à 4 : entrée dans le si
tr = VRAI
sortie du si
i ← i+1 (i=4)
retour au début boucle

i = 4
tr=VRAI
tr est VRAI : **arrêt de la boucle**
renvoyer VRAI

Les entrées et les sorties ne sont pas forcément du même type.
Si l'on recherche un entier X dans un tableau T de nombres entiers, la sortie n'est pas forcément une liste de nombres.

Que peut-elle être ? *Oui/Non (booléen), la position de l'entier dans la liste (un indice), le nombre d'occurrences de l'entier, tout à la fois…*

Application 2 (en devoir à la maison) :
Quelle est l'algorithme permettant de répondre à la question : « Combien de fois X est-il présent dans le tableau T ? ». On l'écrira avec le même langage que ci-dessus.

Remarques :

- Dans l'algorithme ci-dessus, on part du principe qu'il existe une fonction « taille » qui prend en **paramètre** un tableau et qui renvoie le **nombre d'éléments** présents dans ce tableau. Vous noterez que déterminer le nombre d'éléments présents dans un tableau n'est pas vraiment une "opération élémentaire ». Pourtant, on considère l'utilisation de « longueur » comme une opération élémentaire car c'est une **opération commune** !

Application 3 :
Faites « tourner à la main » l'algorithme « X est-il présent dans le tableau T ? » avec T=[5,8,53] et X=42

Application 4 :
Soit T=[12,7,4,1,4] et X=4 .
En faisant « tourner à la main » votre algorithme répondant à la question « Combien de fois X est-il présent dans le tableau T ? », vérifiez que votre algorithme fonctionne.

Remarque :
Dans un cas réel, seul le coeur d'un algorithme demandera un travail au papier/crayon (à imager avec un algorithme cellulaire ie. Wireworld).

III. Analyse de la complexité d'un algorithme

1) Complexité dans le pire des cas : cas particulier

Pour une taille de tableau fixée, les deux algorithmes précédents ne font pas le même nombre d'opérations.

Question 1: Quelle sera la conséquence pour un utilisateur faisant tourner l'algorithme ?
Réponse : Le premier algorithme sera plus rapide. En effet, un système de calculs faisant un nombre fixé d'opérations par secondes, la rapidité d'exécution sera proportionnelle au nombre d'opérations.

Donc, l'un sera plus « rapide » à l'exécution que l'autre car :

Algo 1	12	7	4	1	4
Algo 2	12	7	4	1	4

Évaluons le nombre **d'opérations** élémentaires présentes dans l'algorithme 1 qui « recherche si un entier X est présent dans un tableau T ».

Deux cas doivent être considérés en fonction du tableau T :

- **Cas A :** L'entier X est à la k-ème position (index) dans le tableau T ;
- **Cas B :** L'entier X n'est pas dans le tableau T.

Cas A : Comptez le nombre d'opérations associées à chaque étape numérotée.

```
                                  Données :
                                  T : tableau d'entiers ;
                                  X : nombre entier ;
                                  tr : booléen (VRAI ou FAUX) ;
                                  i : nombre entier
                               1  début
1 opération                    2      i ← 1 ;
1 opération                    3      tr ← FAUX ;
k+1 comparaisons et k+1 tests  4      tant que i ≤ longueur(T) et tr == FAUX faire
k tests d'égalités             5          si T[i]==X alors
1 opération                    6              tr ← VRAI
                               7          fin
k opérations                   8          i ← i + 1
                               9      fin
1 opération                    10     retourner Valeur de tr
                               11 fin
```

Total : 1 + 1 + k + 1 + k + 1 + k + 1 + k + 1 = 4 k + 6

Exemple : Si l'élément est à la 1 000 000ème position, la machine fait 3000006 opérations.
Temps de calcul pour une machine cadencée à 4 GHz. ?

Cas B : Comptez le nombre d'opérations associées à chaque étape numérotée.

```
                                  Données :
                                  T : tableau d'entiers ;
                                  X : nombre entier ;
                                  tr : booléen (VRAI ou FAUX) ;
                                  i : nombre entier
                               1  début
                 1 opération   2      i ← 1 ;
                 1 opération   3      tr ← FAUX ;
n+1 comparaisons et n+1 tests  4      tant que i ≤ longueur(T) et tr == FAUX faire
          n tests d'égalités   5          si T[i]==X alors
                 0 opérations  6              tr ← VRAI
                               7          fin
                 n opérations  8          i ← i + 1
                               9      fin
                 1 opération   10     retourner Valeur de tr
                               11 fin
```

Total : 1 + 1 + n + 1 + n + 1 + n + n + 1 = 4 n + 5

Question 2: Quel est le cas le plus problématique ie. nécessitant le plus d'opérations élémentaires ?
Réponse : Le cas B car $n \geq j$: on doit parcourir tout le tableau dans le cas où X n'est pas dans le tableau.

Ce dernier cas est appelé « **complexité dans le pire des cas** ». C'est la seule complexité qui nous intéressera dans la suite.
Notre exemple fait donc 4n+5 opérations dans le pire des cas.

Question 3: Supposons que l'on a deux algorithmes différents répondant à la même question. Comment faire pour comparer ces deux algorithmes ?
Réponse : En comparant leur complexité dans le pire des cas sur **des données de très grandes tailles**. En effet, sur des données de petites tailles, tous les algorithmes seront équivalents en rapidité d'exécution.

On donne donc un **O**rdre de grandeur (appelé asymptotique) lorsque n, la taille du tableau, devient très grand. Pour se faire, on abandonne les constantes dans notre estimation des opérations élémentaires pour garder les termes de plus haute puissance en n :
$4n + 5 = \mathcal{O}(n)$.

2) Définition théorique

Définition :

La complexité d'un algorithme indique comment augmente la **quantité** de ressources nécessaire à la résolution d'un problème de taille donné n lorsque cette taille est multipliée par 10, 100, 1000.

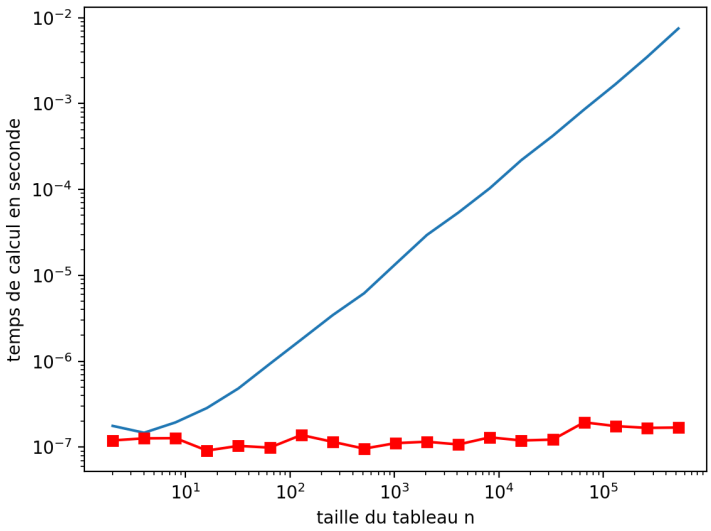
Il existe 2 types de complexité : une complexité en **temps** et en **mémoire**.

Exemple :

Nous avons vu que la recherche d'un élément dans un tableau de taille n a une complexité en temps linéaire. Cette recherche ne nécessite aucun stockage : la complexité mémoire est constante.

À l'inverse, en Terminale, on étudie la recherche d'un élément dans une structure appelée un dictionnaire. Cette recherche est immédiate : elle a une complexité en temps constante. Toutefois, cette recherche nécessite de stocker un nouveau tableau proportionnel à la taille du tableau initial : la complexité mémoire est linéaire.

Le tableau ci-dessous résume cela (tableau en bleu, dictionnaire en rouge) :



Méthodes pour calculer la complexité :

Méthode 1 : on compte le nombre d'opérations élémentaires (voir paragraphe 1) puis on simplifie de telle manière à obtenir la notation $\mathcal{O}$.

Par exemple, dans le cas où nous comptons un polynôme de degré 3 : $6n^3 + 3n + 10$
Il suffit de :

- garder uniquement le n qui possède l'exposant le plus grand (le plus haut degré) ;
- supprimer le coefficient devant le n .

On aura $6n^3 + 3n + 10 = \mathcal{O}(n^3)$.

On dit « n^3 est une bonne approximation de $6n^3 + 3n + 10$ quand n devient grand »

Exemple :
$n^2 + 3n + 4 = \mathcal{O}(n^2)$
$n^3 + 3n + 10^{12} = \mathcal{O}(n^3)$
$\sqrt{n} + 3n + 7 = \mathcal{O}(n)$ car n croit plus vite que $\sqrt{n}$.

Méthode 2 : on compte le **nombre de boucles imbriquées** qui dépendent de la taille du problème n.

Quelques exemples classiques sur un tableau de taille n :

$\mathcal{O}(1)$
Pour i allant de 1 à 12 : (on parcourt les 12 premiers éléments)
…blablablablabla…

$\mathcal{O}(n)$
Pour i allant de 4 à n : (on commence à un nombre quelconque)
…blablablablabla…

$\mathcal{O}(\sqrt{n})$
Pour i allant de 1 à $\sqrt{n}$: (on fait $\sqrt{n}$ opérations)
…blablablablabla…

Pour i allant de 1 à n : (3 boucles imbriquées dépendent de n)
…blablablablabla…

$\mathcal{O}(n^3)$
Pour j allant de 1 à n :
…blablablablabla…
Pour k allant de 1 à n :
…blablablablabla…

Propriété : Comparaison des $\mathcal{O}$

Par ordre de complexité croissante et pour des données de taille n :

Complexité	Nom	taille 5	taille 10	taille 100	Exemple algo
$\mathcal{O}(1)$	Constante	0,001 s	0,001 s	0,001 s	Addition
$\mathcal{O}(\log n)$	Logarithmique	0,001 s	0,001 s	0,002 s	Recherche dichotomique
$\mathcal{O}(n)$	Linéaire	0,005 s	0,01 s	0,1 s	Parcours de liste
$\mathcal{O}(n \log n)$	Linéarithmique	0,003 s	0,01 s	0,2 s	Tri-fusion
$\mathcal{O}(n^2)$	Quadratique	0,025 s	0,1 s	10 s	Parcours tableau 2D
$\mathcal{O}(n^p), p \in \mathbb{N}$	Polynomiale	0,001 s avec p=3	0,0080 s avec p=3	8,0000 s avec p=3	Rotation écran smartphone
$\mathcal{O}(b^n), b \in \mathbb{N}$	Exponentielle de base b	0,001 s avec b=3	0,009 s avec b=3	3 486 784,401 s avec b=3	Problème du sac à dos
$\mathcal{O}(n!)$	Factorielle	0,001 s	0,002 s	154293,633192 329 s	Voyageur de commerce

Application directe : ex 5 fiche d'exos

Exercices récapitulatifs :

Exercice 1 ★ :

a) Que fait l'algorithme ci-contre ?
b) Déterminez sa complexité.

```
Données :
T : tableau d'entiers de taille 3 ;
i : nombre entier ;
somme : nombre entier
1  début
2      somme=0
3      pour i allant de 1 à 3 faire
4          somme ← somme + T[i]
5      fin
6      retourner Valeur de somme
7  fin
```

Exercice 2 ★ :

a) Écrivez un algorithme permettant de trouver le plus grand entier présent dans un tableau d'entiers positifs.
b) Vous testerez votre algorithme en utilisant le tableau T = [3,5,1,8,2].
c) Vous déterminerez ensuite la complexité de votre algorithme.

```
Données :
T : tableau bidimensionnel d'entiers ;
i, j : nombres entiers ;
somme : nombre entier
1  début
2      somme=0
3      pour i allant de 1 à 3 faire
4          somme ← somme + T[i]
5      fin
6      retourner Valeur de somme
7  fin
```

Exercice 3 ★ :

a) Écrivez un algorithme permettant de calculer la moyenne de tous les entiers présents dans un tableau d'entiers.
b) Vous ferez "tourner à la main" votre algorithme en utilisant le tableau T = [3,5,8,2].
c) Vous déterminerez ensuite la complexité de votre algorithme.

```
Données :
T : tableau bidimensionnel d'entiers ;
i, j : nombres entiers ;
somme : nombre entier
1  début
2      somme=0
3      pour i allant de 1 à longueur(n) faire
4          pour j allant de 1 à 3 faire
5              si ..... alors
6                  .....
7              fin
8          fin
9      fin
10 fin
```

Exercice 4 ★★★ :

a) Modifiez l'algorithme de l'exercice 1 afin de trouver les deux plus grands entiers présents dans un tableau d'entiers positifs. Votre algorithme devra être d'une complexité en $\mathcal{O}(n)$.
b) Vous ferez "tourner à la main" votre algorithme en utilisant le tableau T = [3,5,1,8,2,6].

Exercice 5 ★★ :

Dans cet exercice, on s'intéresse à un tableau bidimensionnel T de taille n par n. On accède à la i-ème ligne et à la j-ème colonne de ce tableau grâce à l'instruction :

$$T[i][j]$$

a) Soit le tableau $T = \begin{bmatrix} 5 & -2 & 1 \\ -1 & 4 & -3 \\ 7 & -9 & 5 \end{bmatrix}$. Quel nombre obtient-on si on demande $T[2][1]$? $T[1][3]$? $T[3][3]$?

b) L'algorithme page suivante fait la somme de tous les éléments

```
Données :
T : tableau bidimensionnel d'entiers ;
i, j : nombres entiers ;
somme : nombre entier
1  début
2      somme=0
3      pour i allant de 1 à longueur(n) faire
4          pour j allant de 1 à longueur(n) faire
5              si T[i][j] ≥ 0 alors
6                  somme ← somme + T[i][j]
7              fin
8          fin
9          retourner Valeur de somme
10     fin
11 fin
```

positifs du tableau T. Déterminez le nombre d'opérations effectué par l'algorithme puis donnez sa complexité.

c) Écrire un algorithme calculant la somme des éléments diagonaux du tableau T (on appelle cela la **trace, notée Tr(T)**). Dans notre exemple, la trace serait 5+4+5 = 14. Cet algorithme devra avoir une complexité **linéaire**.

Exercice 6 ★ :

Dans cet exercice, on s'intéresse à un tableau bidimensionnel T de taille n par 3. On accède à la i-ème ligne et à la j-ème colonne de ce tableau grâce à l'instruction :

$$T[i][j]$$

a) Compléter l'algorithme ci-contre afin que celui-ci remplace toutes les valeurs négatives du tableau T par des 0.
b) Déterminez la complexité de votre algorithme.

Exercice 7 ★★ :

Le problème du voyageur de commerce est le suivant :
« étant donné n villes et les distances séparant chaque ville, *trouver un chemin de longueur totale minimale qui passe exactement une fois par chaque ville et revienne au point de départ*. »

On dispose d'une instruction élémentaire « TrouverChemin » qui renvoie la longueur totale d'**un** chemin passant exactement une fois par chaque ville.

Cas particulier : calcul du nombre de chemins pour n = 4

a) Combien de villes peut-on choisir initialement ?
Une fois une ville choisie, combien de villes peut-on choisir ?
b) En déduire le nombre de chemins total dans le cas n = 4 puis le nombre de fois où on appellera l'instruction « TrouverChemin » . On pourra utiliser la notation factorielle définie par $n! = 1 \times 2 \times 3 \times 4 \ldots \times n$.

Cas général : calcul du nombre de chemins pour n grand

a) Combien de villes peut-on choisir initialement ?
Une fois une ville choisie, combien de villes peut-on choisir ?
Répéter l'opération…
b) En déduire le nombre de chemins total dans le cas général puis le nombre de fois où l'on appellera l'instruction « TrouverChemin ». On pourra utiliser la notation factorielle définie par $n! = 1 \times 2 \times 3 \times 4 \ldots \times n$.
c) Conclure sur la complexité de l'algorithme.
d) Dans le cas où l'on met 1 ns (10^{-9} s) pour calculer un chemin et où l'on dispose de 20 villes. Combien de temps va-t-on mettre pour calculer le chemin de longueur minimale?

Extension :

Le choix initial de la ville n'a en fait pas d'importance et nous avons compté les chemins deux fois (aller-retour).

Le nombre d'appels sera donc égal à $\frac{(n-1)!}{2}$. La complexité de l'algorithme va-t-elle changer?