PSEUDO-CODE ET LANGAGE JAVASCRIPT : - LES TABLEAUX -

INTÉRÊT DES TABLEAUX PAR L'EXEMPLE

Comment conserver un ensemble de notes pour effectuer des calculs de moyenne, de maximum, de minimum pour une classe de 20 élèves. La seule solution algorithmique dont nous disposons à l'heure actuelle :

- Déclarer 20 variables représentant les 20 notes nommées : n1, n2, n3,…, n20
- Saisir les 20 notes dans 20 SAISIR distinctes : saisir n1, saisir n2,…, saisir n20
- Effectuer le calcul de la moyenne avec l'instruction : (n1 + n2 +… + n20)/20

Imaginons que nous soyons dans un programme avec quelques centaines ou milliers de valeurs à traiter ➔**ingérable**

Imaginons que le nombre d'élèves augmente ou diminue ➔**ingérable**

Conclusion : il est difficile de gérer ce type de problème en utilisant de simples variables.

⇨ Pour gérer ce problème, on utilisera un TABLEAU : permets de rassembler toutes ces variables en une seule.

STRUCTURE D'UN TABLEAU EN PSEUDO-CODE

Un tableau est une structure de données permettant un accès direct aux éléments au travers d'un indice (ou numéro de case) dont voici les caractéristiques :

⇨ Chaque case est numérotée (ou indicé) de 0 (la première case) à N-1 (la dernière).
⇨ N entier et >1
⇨ Toutes les cases contiennent une valeur d'un type donné.

Représentation d'un tableau A de N cases contenant des entiers.

	0	1	2	3	4	…	N-2	N-1
Tableau A :	15	25	0	22	-10	…	56	33

DÉCLARATION D'UN TABLEAU EN JS

De nombreuses syntaxes sont possible en JavaScript pour créer un tableau. Pour simplifier les choses, nous verrons 2 façons de faire :

Façon 1 : Tableau vide

```
let A=new Array(100) ;
```

Ici A est un tableau de 100 cases. Ce tableau est construit (**new**) à l'aide de la classe Array. Chaque élément de ce tableau est initialisé à "**undefined**".

Dans l'exemple précédent, le nombre de cases est fixe, connu avant l'exécution du programme. Cela présente des inconvénients. Aussi il est préférable de définir la taille du tableau avec une variable N que l'utilisateur pourra choisir pendant l'exécution du programme.

Exemple :

```
let N=parseInt (prompt ("Nombre de cases du tableau ? "));
let B=new Array(N) ;
```

Façon 2 : Tableau prérempli

```
let A=new Array(10,20, 30,40, 50) ;
```

Dans cet exemple (A) est un tableau de 5 cases **réservé automatiquement.** La case d'indice 0 contient la valeur 10, et ainsi de suite.

En JS, comme dans beaucoup d'autres langages, il est possible qu'un tableau contienne des éléments de nature différente comme dans l'exemple ci-dessous. Nous avons ici un tableau de 6 éléments avec une chaine, un boolean, etc… Bien que possible, cela reste déconseillé.

```
let A=new Array("A", true, 10, undefined, null, 3.5) ;
```

L'ACCÈS AUX ÉLÉMENTS DU TABLEAU

L'accès aux éléments du tableau se fait de façon directe en donnant le numéro de case entre crochets **[ ]**.

L'accès se fait : soit en LECTURE, soit en ÉCRITURE.

ACCÈS EN LECTURE

Le tableau se trouve dans l'expression, à droite du signe d'affectation. Une opération de lecture consiste à récupérer la valeur d'une case.

Exemple :

```
let B=new Array(10,20, 30,40, 50,10, 20,30, 40,50) ;       //B est un tableau prérempli de 10 cases
let i=5 ;              //i est appelé indice. Par convention on utilise les lettres i, j,k pour cela.
let val = B[3] + 5 ;   //on accède à la case d'indice 3 du tableau,
                       //le contenu de cette case + 5  est affecté à val
val = B[i] ;           //on accède à la i-ème case du tableau, le contenu de cette case est affecté à val
```

ACCÈS EN ÉCRITURE

On souhaite modifier le contenu d'une case. Exemple :

```
const N= 10 ;
let A= new Array(N) ;
let i=5 ;      //i : indice.
let val = 8 ;
A[3] = A[2] + 10 ;     //on accède en écriture à la case 3 du tableau, le contenu de cette case est
                       //remplacé par la valeur 10 + A[2]
A [i] = val ;  //on accède à la ième case du tableau, le contenu de cette case est remplacé par val
```

ERREURS FRÉQUENTES

Dans de nombreux langages (en C, JAVA) un tableau est une suite de cases se suivant les unes à la suite des autres en mémoire.

Dans le cas d'un tableau A de 5 cases. On a la représentation mémoire suivante :

		@1ère case	@2ème cases	@3ème cases	@4ème case	@5ème case			
		4 octets	4 octets	4 octets	4 octets	4 octets			
A[-2]	A[-1]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

Notre tableau se trouve dans la mémoire. Cela signifie qu'il y a des données avant notre tableau et après. Une erreur fréquente consiste à accéder à un élément du tableau qui n'existe pas. Par exemple A[-1] ou A[5].

Ainsi il est possible d'accéder à des zones mémoires qui se trouvent avant le tableau ou après. Le problème c'est que cette zone mémoire est très certainement utilisée par une autre variable.

Dans ce cas, on vient modifier le contenu de cette variable. C'est ce qu'on appelle une opération à effet de bord ou plus connu sous le nom anglais de "stack overflow".

Le JavaScript est particulier dans ce domaine. Dans notre exemple, l'accès à la case 5 en écriture ajoute un nouvel élément.

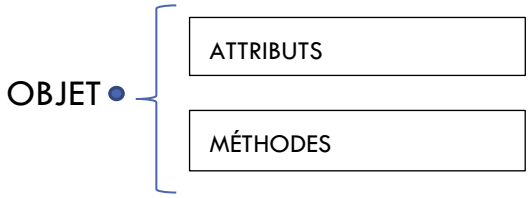
```
A[5]=10 ;//ajoute une nouvelle case.
```

```
document.write (A[6]) ;    //Accès en lecture une case qui n'existe pas : aucune erreur.
                          //La valeur affichée est undefined. Pas d'ajout de case au tableau
```

LA CLASSE ARRAY

Array est une classe, donc un modèle. Tous les tableaux définis à partir de ce modèle (new Array()) deviennent donc des **Objets**.

Un **Objet** rend des services au travers de ses méthodes et attributs. La figure ci-dessous montre comme accéder aux attributs et aux méthodes en utilisant l'opérateur (.) point appelé **accesseur de propriétés**



LA PROPRIÉTÉ LENGTH

La propriété length (longueur) est un entier non signé de 32 bits qui indique le nombre d'éléments présents dans le tableau.

Exemple : dans la console est affiché le nombre 6. Ce tableau contient 6 éléments.

```
let A = new Array(10, 20, 30, 40, 50, 60) ;
console.log (A.length) ;
```

Cette propriété sera particulièrement utile pour le parcours d'un tableau

LES MÉTHODES

Il existe beaucoup de méthode associée aux tableaux. Il est par exemple possible de faire le **tri** d'un tableau avec la méthode **sort ()**. C'est très pratique et très simple d'utilisation. Mais c'est une méthode "boîte noire".

Dans le cadre de l'apprentissage de l'algorithmique, il est utile de savoir comment programmer ce type d'algorithme. C'est ce que nous allons voir dans ce cours.

Aussi nous verrons seulement 2 méthodes. D'autres seront étudiés en TP et dans ce cours.

- La méthode **push** () ajoute un ou plusieurs éléments à la fin d'un tableau et retourne la nouvelle taille du tableau.

Exemple

```
let A = new Array(10, 20, 30, 40, 50, 60) ;
let taille = A.push (70) ;
console.log (taille) ;    //nouvelle taille : 7 éléments
console.table (A) ;
```

Dans l'exemple précédent, noter la méthode **table** () de l'objet **console**, qui affiche le tableau A dans la console comme illustré. Très pratique en mode débogage.

(index)	Value
0	10
1	20
2	30
3	40
4	50
5	60
6	70

- La méthode **pop** () supprime le dernier élément d'un tableau et retourne cet élément. Cette méthode modifie la longueur du tableau.

```
let A = new Array(10, 20, 30, 40, 50, 60) ;
let dernier = A.pop () ;
console.log (dernier) ;              //affiche 60
console.log (A.length) ;             //affiche la nouvelle taille : 5
```

LES ALGORITHMES SUR TABLEAUX

Les différentes opérations algorithmiques sur un tableau sont : parcours, recherche, suppression, ajout et tri.

La suite de ce cours présente une façon classique de programmer ces différents algorithmes et propose ensuite une approche

LE PARCOURS

Définition: nous parlons de parcours de tableau, lorsqu'on accède à tous les éléments en appliquant un même traitement sur chacun des éléments.

Algorithme général : on applique ce modèle :

```
let A = new Array(10, 20, 30, 40, 50) ;

let nb_elem = A.length ;      //Calcul du nombre d'éléments
for (let i = 0 ; i < nb_elem ; i++) {
      //traitement identique sur toutes les cases du tableau. : ici un affichage
      document.write (A [i] + " ") ;
}

//parcours inverse
for (i = nb_elem - 1 ; i >= 0 ; i --) {
    document.write (A [i] + " ") ;
}
```

Le parcours se fait de la 1ère case (0) jusqu'à la dernière (N-1).

LA RECHERCHE

Nous parlons de **recherche** dans un tableau, lorsqu'on cherche un élément vérifiant une certaine propriété. Cette recherche s'effectue jusqu'à la fin du tableau si l'élément cherché n'existe pas. Une recherche donne deux solutions possibles :

- L'élément cherché est trouvé
- L'élément cherché n'est pas trouvé

Une recherche est différente d'un parcours puisqu'aucun traitement systématique n'est effectué sur tous les éléments. On peut quitter avant la fin du tableau, si l'élément est trouvé.

```
let A = new Array(10, 20, 30, 40, 50) ;
let valCherche = 35 ;
let i ;
for (i = 0 ; i < A.length ; i++) {
    if (A [i] === valCherche)      //trouvé
        break;  //pour arrêter la boucle avant son terme.
}
if (i < A.length) {//on a trouvé
    document.write ("Touvé à l'indice " + i) ;

    //traitement cas trouvé
} else {
    document.write ("Pas Touvé -1") ;
    //traitement cas non trouvé
}
```

JavaScript propose dans la classe Array 2 fonctions permettant de faire une recherche dans un tableau :

- La méthode **indexOf**() renvoie le premier indice pour lequel on trouve un élément donné dans un tableau. Si l'élément cherché n'est pas présent dans le tableau, la méthode renverra -1.

```
let A = new Array(1, 25, 30, 25, 50) ;
let posElemCherche = A.indexOf (25) ; //vaut 1
console.log (posElemCherche) ;
```

- La méthode **lastIndexOf**() permet de renvoyer le dernier indice pour lequel une valeur donnée est présente dans un tableau. Si la valeur recherchée n'est pas présente, le résultat sera -1. Corresponds à une recherche en partant de la fin.

```
let A = new Array(1, 25, 30, 25, 50) ;
let dernierPosElemCherche = A.lastIndexOf (25) ;
console.log (dernierPosElemCherche) ;          //vaut 3
```

L'AJOUT ET L'INSERTION

L'AJOUT

Au niveau algorithmique, un tableau est déclaré avec un nombre de cases fixes. Si celui-ci est déclaré avec N cases, il n'est **pas possible d'ajouter** une case supplémentaire (N+1) à ce tableau.

En JavaScript les choses sont beaucoup plus souples. Il est possible de créer un tableau vide, puis d'ajouter dynamiquement (méthode **push**) des cases à notre tableau.

Exemple :

```
let A = new Array(10) ;         //A contient 10 cases vides
let B = new Array() ;  //B est un tableau avec aucune case (vide)

A.push (3) ;    //A contient maintenant 11 cases avec la valeur 3 dans la dernière.
                //Les 10 premières cases sont toujours vides (undefined)

B.push (10);B.push (5) ; //Ajout d'une case avec la valeur 10 puis 5. B à 2 cases
```

Dans le cas du tableau A celui-ci contient N cases (10 ici), chaque élément est initialisé à "undefined". On considère que ce tableau à 10 cases vides.

Tableau A :

0	1	2	3	4	...	N-2	N-1
undefined	undefined	undefined	undefined	undefined	undefined	undefined	undefined

Une opération classique consiste à remplir le tableau avec des valeurs. Elles peuvent être choisi par l'utilisateur ou bien de façon aléatoire. Il s'agit d'un parcours classique. Attention ici aucune case n'est ajoutée. Elles sont simplement remplies !!

Exemple :

```
let A = new Array(10) ;
for (let i = 0 ; i < A.length ; i++) {
    A [i] = parseInt (prompt ("Choisir une valeur pour la case " + i)) ;
}
document.write (A) ;
```

Parfois, le but n'est pas de remplir complètement le tableau. Une solution consiste à retenir le nombre d'éléments utilisé (nb) sur N existants.

Dans le tableau A suivant de (N) cases, seulement (3) sont utilisés. La variable (nb) est fixée à la valeur 3

Tableau A :

0	1	2	3	4	...	N-2	N-1
12	8	9	~~undefined~~	~~undefined~~	~~undefined~~	~~undefined~~	~~undefined~~

L'ajout dans ce cas-là consiste à remplir la première case libre se trouvant à l'indice (nb)

```
//on ajoute un élément à l'indice (nb) ➔1ère case vide
A [nb] = parseInt (prompt ("Choisir une valeur pour la case " + nb)) ;
nb++ ;   //on a rempli une case supplémentaire dans A, donc nb++
```

Si le tableau est rempli (cas où nb==N) 2 cas de figure sont possibles :

- Solution 1 : le problème interdit l'ajout d'une case au tableau : on ne fait rien
- Solution 2 : le problème posé autorise l'ajout de case. On utilise la méthode **push ()**

L'INSERTION

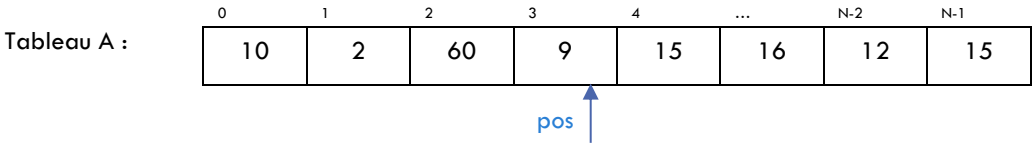
L'insertion correspond à un ajout d'une nouvelle case dans le tableau, mais pas forcément en fin de tableau comme avec **push.** L'insertion se fait à une position précise.

Cette position peut être :

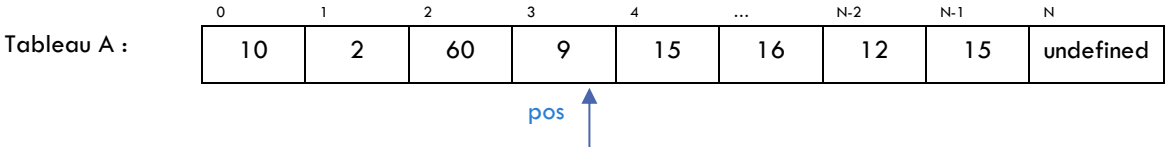
- Définie par le programme (donc recherche de la position d'insertion). Cela peut être le cas si on veut que le tableau resté trié.
- Ou bien choisie par l'utilisateur.
- Calculé, etc…

Il faut vérifier que la position d'insertion appartient bien à l'intervalle : [0… N-1]

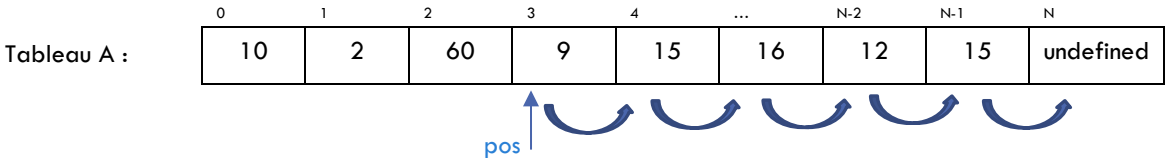
Exemple : dans ce tableau nous allons insérer la valeur **6**. La position d'insertion de trouve à l'indice pos (=3).



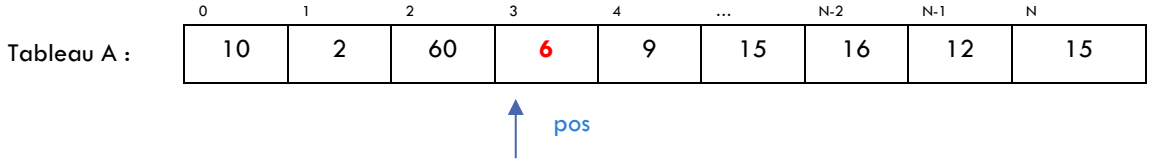
- On commence par ajouter une case vide au tableau A



- Ensuite il faut décaler les éléments en partant de la fin jusqu'à pos



- Il ne reste plus qu'à remplir la case pos avec la valeur à insérer



Code JavaScript correspondant :

```
let A = new Array(10,2, 60,9, 15,16, 12,15) ;
A.push (undefined) ;
let pos=3 ;
for (let i=A.length-1 ; i>= pos;i -- ){
    A[i]=A[i-1] ;
}
 A[pos]=6 ;
```

Attention :

Comme pour l'ajout, 2 cas de figure sont possibles :

- Solution 1 : le problème interdit l'ajout d'une case au tableau : l'insertion n'est pas possible donc on ne fait rien
- Solution 2 : le problème posé autorise l'ajout de case. On utilise la méthode **push () + décalage**

JavaScript propose la méthode la méthode **splice** () qui permet l'insertion ou la suppression d'éléments.

Exemple : on ajoute dans A la valeur 33 à la position 6. Dans le cas d'une insertion, le 2ème paramètre doit rester à 0. Voir la documentation ici : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/splice

```
A.splice (6,0, 33) ;
```

Il existe également la méthode **unshift** () qui insère (donc ajoute) un ou plusieurs éléments au début d'un tableau et renvoie la nouvelle longueur du tableau.

```
let A = new Array(6,8, 9,15, 12,15, 3,2) ;
let taille = A.unshift (1,2, 3);  //ajout des valeurs 1,2 et 3 en début de tableau
                                  //nouvelle taille : 3 cases de plus
```

LA SUPPRESSION

Nous parlons de suppression lorsqu'il s'agit de retirer un élément donné du tableau. Après cette opération un tableau de N cases en contiendra N-1.

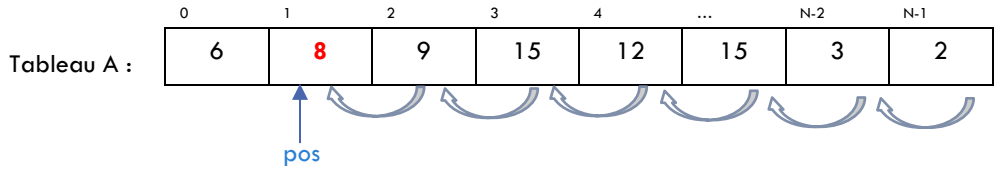
Comme pour l'insertion avant de supprimer un élément il faut connaître la position de cet élément.

Cette position (indice **pos**) peut être :

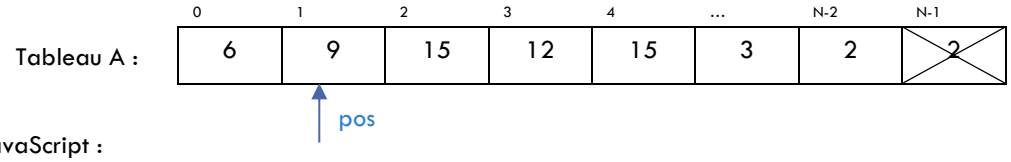
- Recherché par programme.
- Calculé
- Choisie par l'utilisateur

Dans tous les cas on pensera à vérifier que l'indice pos se trouve dans l'intervalle : [0... N-1]. Comme pour l'ajout la suppression se fait par décalage successive. Ces décalages se feront de la position de suppression jusqu'à la fin du tableau.

Exemple : dans cet exemple nous allons supprimer la valeur **8**. La position de suppression se trouve à l'indice pos (=1). Nous allons décaler les éléments en partant de l'indice **pos** jusqu'à la fin du tableau (case N-1).



Après les décalages la dernière case est supprimée (méthode **pop**)



Code JavaScript :

```
let A = new Array(6,8, 9,15, 12,15, 3,2) ;
let pos=1 ;

for (let i=pos ; i<A.length;i++){      //décalage
    A[i]=A[i+1] ;
}
A.pop () ;              //on supprime la dernière case
document.write (A) ;
```

La méthode **shift** () permet de retirer le premier élément d'un tableau et de renvoyer cet élément.

```
let A = new Array(6,8, 9,15, 12,15, 3,2) ;
let taille = A.shift ();  //la valeur 6 est supprimée. Une case de moins dans taille
```

Déjà présenté la méthode **splice** qui permet aussi de supprimer un élément à une certaine position

```
let A = new Array(6,8, 9,15, 12,15, 3,2) ;
A.splice (3,1) ;        //on supprime 1 case à partir de la position 3
```

LE TRI

Un algorithme de tri permet de ranger en ordre croissant ou décroissant un tableau. Ces relations d'ordre existent avec :

- Les entiers : -1 < 0 < 1 < 2 < 3 < … < 9
- Les réels
- Les caractères : 'A'< 'B'. Cette relation d'ordre est donnée par la table ASCII.

Ils existent de nombreux algorithmes de tris. Le choix de l'algorithme se fait selon ces critères :

- Efficace (rapidité) -> Complexe (niveau algorithmique)
- Facile à programmer (niveau algorithmique) -> moins efficace (rapidité)

Nous allons voir dans ce cours 2 algorithmes de tris faciles à programmer :

- Tri par recherche de minima, maxima
- Tri à bulle

TRI PAR RECHERCHE DE MINIMA

Le principe de cet algorithme est de rechercher le minimum d'un tableau, puis d'échanger la valeur trouvée avec la première case du tableau. On répète l'opération, mais cette fois-ci l'échange se fait entre le minimum et la deuxième case du tableau et ainsi de suite. Cet algorithme permet de faire un tri croissant. Pour un tri décroissant, il suffit de lire le tableau en sens inverse :

Algorithme :

```
A:tableau d'entier sur [1..N] ' on suppose ce tableau déjà initialisé avec N éléments
i, j,posMin : entier 'indice
temp : entier

debut
        pour i ← 0 à N-1 faire
                min ← A [i]
                pour j ← i à N-1 faire 'recherche du minimum
                        si A [j] < min alors
                                min ← A [j]
                                posMin ← j
                        finSi
                finPour
                temp ← A [posMin]     'échange
                A [posMin] ← A [i]
                A [i] ← temp
        finPour
fin
```

TRI À BULLE

Le principe de cet algorithme est le suivant pour un tri décroissant :

A. On compare le contenu de deux cases du tableau.
   - Si la première valeur est inférieure à la seconde, on effectue un échange.
   - Répéter l'opération pour chaque case du tableau jusqu'à la fin du tableau

Si des échanges ont eu lieu, refaire l'opération (retourner en A)

```
A:tableau d'entier sur [1..N]//déjà initialisé avec marque
i, j,posMax : entier  //indice
temp : entier
Echange : booléen

Début
Echange ← Vrai;  //a vrai lorsqu'il y a eu un échange
tantque Echange ←vrai faire
               Echange ←  faux       //si echange reste à faux le tableau est trié.
               pour i ← 0 à N-1 faire
                      si A [i] < A[i+1] alors           //échange
                             temp ← A [i] ;
                             A [i] ← A[i+1] ;
                             A[i+1] ← temp ;
                             Echange ← true ;
                      finsi
               finPour
       finTantque
fin
```