PSEUDO-CODE ET LANGAGE JAVASCRIPT : LES STRUCTURES DE CONTRÔLES : LA REPTITION

INTRODUCTION

Nous avons vu dans le cours précédent l'utilisation des structures conditionnelles. Celles-ci ne suffisent pas dans bien des cas pour réaliser nos algorithmes.

Prenons un exemple : Prenons le cas d'une saisie au clavier (une lecture), où par exemple, le programme pose une question à laquelle l'utilisateur doit répondre par O (Oui) ou N (Non). Il faut, évidement, mettre en place un contrôle de saisie, afin de vérifier que les données entrées au clavier correspondent bien à celles attendues par l'algorithme.

```
Var Rep : Caractère
Début
        Afficher "Voulez vous un café ? (O/N)"
        Saisir Rep
        Si Rep <> "O" et Rep <> "N" Alors
                Afficher "Saisie erronée. Recommencez"
                Saisir Rep
        FinSi
Fin
```

C'est impeccable. Du moins tant que l'utilisateur a le bon goût de ne se tromper qu'une seule fois, et d'entrer une valeur correcte à la deuxième demande ! La solution consistant à aligner des SI… en pagaille est donc une impasse.

La bonne solution consiste ici à reposer la question **tant qu'**il n'apporte pas une réponse correcte.

Ces structures se montrent indispensables dans de très nombreux cas.

En voici d'autres exemples :

- Chercher dans un annuaire le numéro de téléphone d'une personne dont on connaît le nom. Le nombre de répétitions dépend de la longueur de la liste mais également de la position du nom dans la liste.
- Calculer la N^{ème} puissance d'un nombre "a" par multiplication successive du nombre a par lui-même.

Il existe 3 formes de répétitions en algorithmique, même chose en JS, qui s'utilisent selon des critères bien précis :

En ALGO	En JS	Critère d'utilisation
Le POUR :	for	S'utilise lorsque le nombre d'itération (n) est connu à l'avance, donc **fixe**
Le TANTQUE :	while	S'utilise lorsque le nombre d'itération (n) est inconnu (**de 0 à +∞)**
Le REPETER JUSQU'A:	do … while	S'utilise lorsque le nombre d'itération (n) est inconnu mais ce fait au moins 1 fois (**de 1 à +∞)**

Seule la boucle "TANT QUE" est fondamentale. Avec cette boucle, on peut réaliser toutes les autres boucles.

REPETION AVEC TANTQUE … FAIRE

La boucle TANT QUE … FAIRE permet de répéter un traitement tant qu'une **expression conditionnelle** est vrai. Si lors du tout premier test, la condition n'est pas vraie, le traitement ne sera pas exécuté. Le nombre d'itération est donc compris entre **0 .. +∞**

Donc attention, si la condition est toujours VRAI on fait une boucle infinie !

Pseudo langage:	Comportement :	EN JS:

TANTQUE condition évalué à vrai FAIRE
action1
action2] Séquence A
FINTANTQUE
		while(condition)
		{
		instruction1;
		instruction2;
		}

Que ce soit en pseudo-code ou en JS, la problématique principale est la même : il faut trouver la condition. La condition que l'on va mettre dans le TANTQUE exprime les raisons de continuer à boucler. Il faut donc se poser cette question :

1. **A quelle condition continu-t-on de boucler ?** réponse à la question **: je continue tant que …**

L'expérience montre que pour certaines personnes il est parfois plus facile de trouver la raison pour arrêter la boucle. Dans ce cas la question à se poser est :

2. **A quelle condition arrête-t-on de boucler ?** réponse à la question : **je m'arrête quand la condition …**

Prenons un exemple pour illustrer ses 2 questions : on veut écrire un algorithme permettant d'afficher les 11 premiers nombres (de 0 à 10). Pour résoudre ce problème on utilisera une variable "i"(compteur) comme variable de boucle.

- La réponse à la question 1 est **: je continue tant que** i <= 10
- La réponse à la question 2 est : **je m'arrête quand** i > 10

Ce que l'on mettra comme condition à notre boucle c'est la réponse à la question 1. Mais si on est plus à l'aise avec la question 2, il suffira d'inverser le résultat et notre condition deviendra : NON(i>10) ce qui revient à écrire i <= 10

Ce qui donne:

```
VAR i : entier
DEBUT
       i←0
       TANTQUE i <= 10 FAIRE
              Afficher i
              i ← i + 1
       FIN TQ
FIN
```

```
function exercice(){
       let i=0;
       while(i <= 10) {
              alert(i);
              i=i+1;
       }
}
```

Remarque : ici le nombre d'itérations est connues: le POUR est le plus adapté

Attention: Comme avec l'instruction if, il faut se méfier se méfier de l'opérateur d'affectation (=) qui ne doit pas être confondu avec l'opérateur de comparaison (==) ou égalité strict (===)

Exemple : ici on ne test pas si (i) est égale à 10, mais on affecte à (i) la valeur 10. Comme 10 est une valeur > 0, et que toute valeur positive est considérée comme VRAI, ce programme revient à remplacer (i=10) par (VRAI)

```
int i=0;
while(i=10) //boucle infinie => condition toujours vrai. C'est comme si nous avions écrit : while (true) …
{
       alert("bonjour");
       i=0;  //même si (i) repositionné à 0 on boucle quand même car (i) est systématiquement remis à 10
}
```

Exercice : Écrire un algorithme qui calcule le cube des nombres que l'utilisateur saisit. Pour arrêter, l'utilisateur doit entrer la valeur 0. Si l'utilisateur saisit 0 le programme s'arrête. Le cube de 0 n'est pas affiché. Dans cette exercice le nombre de répétitions n'est pas connu à l'avance. L'utilisateur à tout à fait le droit de saisir 0 dès la 1ère saisie. Il est donc logique que problème soit géré avec un TANTQUE.

Exercice : écrire un programme qui calcule la somme des entiers de 1 à 100.

Pour réaliser ce problème on a tendance à réfléchir comme on le ferait en math. Le résultat c'est 1+2+3+4+5+6+…+99+100.

Dans un programme on ne pourra faire comme cela. En réalité cet exercice correspond à un problème dit de CUMUL. Plutôt que de calculer en ligne, on procèdera par résultat intermédiaire. On calcul 1+2 ➔ 3, puis 3+3 ➔6, puis 6+4➔10, puis 10+5➔15…

Exercice : reprendre l'exercice précédent mais cette fois-ci l'utilisateur saisie la valeur limite (N). Donc on calculera la somme des entiers de 1 à N. Cette notation avec la lettre "N" est très souvent utilisée dans les algorithmes, pour indiquer que l'on fera N fois la boucle. N est ici une inconnue au moment où l'on écrit notre programme. Sa valeur ne sera connue qu'au moment de l'exécution du programme. Ici N dépend de la saisie de l'utilisateur

Exercice: écrire un programme qui permet à un utilisateur de saisir un code PIN (code à 4 chiffre). L'utilisateur à le droit à 5 essai maximum pour saisir son code. Au-delà un message de refus, et si le code est correct un message de bienvenue.

REPETITION AVEC REPETER…TANTQUE

Le **REPETER …**

permet de répéter un traitement jusqu'à ce qu'une **expression conditionnelle** soit évaluée à vrai. Le test se fait en fin de boucle, donc la séquence A est exécuté au moins une fois. Le nombre d'itération est donc compris entre **1 … +∞**. Parce qu'elles sont exécutées au moins une fois, les boucles REPETER … TANTQUE sont bien adaptés aux contrôles de saisie

Attention comme avec une boucle TANTQUE, si la condition est toujours VRAI ➔ boucle infinie.

Pseudo langage:	Comportement :	EN JS:
REPETER action1 action2 Séquence A TANTQUE condition évalué à vrai		do { instruction1; instruction2; }while (condition) ;

Au niveau algorithmique le REPETER … TANTQUE n'existe pas. On utilise une syntaxe qui est le REPETER … JUSQU'A. Pour simplifier les choses nous n'utiliserons pas cette dernière syntaxe.

Au niveau du langage JS, le mot clef utiliser et do {…} while(condition); On note les similitudes avec le while(condition) {…}, à ceci près la présence d'un ; à la fin du **while**. Celui-ci est obligatoire, et c'est la seule structure de contrôle qui a besoin de ce point-virgule de fin.

Attention 1 : ne pas mettre un ; à la fin d'une boucle TANTQUE, SI, SUIVANT, POUR. Dans le programme ci-contre un ; a été ajouté à la fin du while. Si aucune erreur n'est générée au niveau de la syntaxe et semble donc tout à fait cohérent, ce programme boucle à l'infini.

```
Let i = 0;
while (i < 10);
{
      alert(i);
      i = i + 1;
}
```

Explication dans le code en commentaire :

```
int i = 0;
while (i < 10)
    ;  // ici j'exécute une instruction vide. Comme i n'est pas modifié, la condition sera
       // toujours évaluée à VRAI

{      // le bloc d'instruction est indépendant, et n'est pas rattaché au while. Ce bloc ne
       // sera jamais exécuté
    alert(i);
    i = i + 1;
}
```

Attention 2 : Si au moins 2 instructions dans la boucle ajouter les { } délimitant le bloc , sinon risque de boucle ∞ comme dans l'exemple

```
int i = 0;
while (i < 10)
    alert(i);       // ici seul la fonction alert est rattaché au while. En l'absence des { }
                    // Comme i n'est pas modifié, la condition sera toujours évaluée à VRAI
    i = i + 1;      // l'instruction i=i+1 est en dehors du while, donc jamais exécuté.
```

Comme le montre ces 2 exemples, on peut facilement faire une boucle infinie. Pour éviter ses pièges on retient :

- **Pas de ; à la fin d'un while(), mais un ; à la fin d'un do{}while();**
- **Programmer simplement en n'oubliant pas les { } du bloc while, do while, for**

Exercice : reprendre l'exemple d'introduction (réponse O(Oui), N(Non)). Le gérer avec un REPETER … TANTQUE.

Exercice : écrire un programme qui simule une caisse enregistreuse. La caissière rentre le prix d'un article, sa quantité, et ceux pour tous les articles. Le prix total est affiché lorsqu'il n'y a plus d'article. Pour cela le programme pose la question "Encore des articles ? O=Oui, N=Non"

REPETITION AVEC LE POUR

Dans l'exercice précédent nous voyons bien que le nombre de produits n'est pas connu à l'avance (au moins un). Dans ces cas-là la boucle TANTQUE ou REPETER est bien adapté. Lorsque le nombre de répétitions est connu à l'avance on dispose de la boucle POUR.

Le principe de cette boucle est d'utiliser une variable de boucle qui est affecté à une valeur **début**. Par convention on utilise les lettres (i), (j) et (k) pour ces variables de boucle qui sont des compteurs. A chaque tour dans la boucle on incrémente la variable de boucle d'un **pas** définie dans le POUR. Ce pas est en générale de +1. On compte donc de 1 en 1, jusqu'à ce que (i) atteigne la valeur de **fin incluse**. Le pas peut être de 2,3,… mais aussi -1,-2 ,…

Pseudo langage:	Comportement :	EN JS:

Pour i ← début à fin **pas** incrément **faire**
action1
action2 — Séquence A
Fin Pour
		for(i=début ; i<= fin ; i=i+1)
		{ ❶ ❷ ❸
		inst1;
		inst2;
		}

Dans la syntaxe du JS on distingue 3 parties distinctes séparés par des ;

- ❶ : initialisation de la variable (i). Ne se fait qu'une fois, avant la boucle
- ❷ : test d'arrêt de la boucle. Se fait à chaque tour de la boucle. Si la condition est VRAI on continu la boucle.
- ❸ : incrémentation de la variable(i). Se fait à chaque tour de la boucle

Exemple : afficher les 10 premiers nombres pairs ➔ 0,2,4,6,8,10,12,14,16,18

Pseudo langage:	EN JS :

Pour i ← 0 à 18 **pas** +2 **faire**
Afficher(i)
Fin Pour
	let i;
	for(i=0 ; i<= 18 ; i=i+2)
	{
	alert(i);
	}

On pourrait faire la même chose avec une boucle TANTQUE, mais il faudrait initialiser la variable (i) et l'incrémenter explicitement. Le POUR fait la même chose mais en plus simple.

```
i ←  0
TANT QUE i <= 18 FAIRE
        Afficher( i )
        i ←  i + 2    // dans la boucle POUR cette ligne n'apparait pas. L'incrémentation est automatique.
FINTANTQUE
```

L'INSTRUCTION BREAK

Comme avec le switch, on peut utiliser l'instruction break pour sortir d'une boucle while, for et do while. Cette instruction permet de quitter à tous moment une boucle. Notons cependant que dans un while et un do{} while() on peut en général s'en passer.

Exemple :

```
char c; int x=0;
while( x<10){
        alert(x);
        c=prompt("Saisir c");
        if(c=='q'){
                break;  //pour sortir
        }
        x++;
}
```

FAIRE UNE TRACE D'EXECUTION

Il est souvent bien utile de réaliser ce que l'on appelle une trace d'exécution. Cela consiste à dérouler le programme à la main, en vérifiant son bon fonctionnement, ou bien pour comprendre ce qu'il fait.

Exemple: que fait ce programme ? Pour répondre nous allons faire une trace d'exécution en complétant le tableau ci-dessous.

```
PROGRAMME Pgr1
VAR
        a, resultat: réel
        n, i : entier
DEBUT
        resultat ← 1
        Lire a, n
        POUR i ← 1 à n  FAIRE
                    resultat ← resultat * a
        FINPOUR
        ECRIRE "Le resultat : ", resultat
FIN
```

Variable / Opérations	a	n	i	i <= n	resultat
Initialisation des variables	0	0	0		0
resultat ← 1	0	0	0		1
Lire a, n	10	3	0		1
Pour	10	3	1	True	1
resultat ← resultat * a	10	3	1	True	10
resultat ← resultat * a			2	True	100
			3	True	1000
	10	3	4	False	1000
Ecrire					

CONCLUSION

Grâce aux différentes structures de contrôles on peut résoudre une grande partie des problèmes informatiques. Le problème est de faire le bon choix parmi toutes ces structures. Pour cela on peut suivre l'algorithme suivant:

```
Si le nombre de répétition est connu alors
        Choisir un "POUR"
Sinon si le nombre de répétition >= 1 alors
        Choisir un "REPETER"
Sinon
        Choisir un "TANTQUE"
```

```
finsi
```