Exercice noté 1

Vous allez écrire un programme javascript qui va vérifier la validité d'un mot de passe entré par l'utilisateur.

Créez un fichier dans codeboot, renommez-le **pass.js**

Le programme devra

* Demander à l'utilisateur d'entrer un mot de passe, avec la fonction **prompt**, et stocker le résultat dans une variable nommée **mot***.*
* Afficher le message **Mot de passe:** suivi du mot entré, avec la fonction **print**.
* Demander à l'utilisateur d'entrer le mot de passe une deuxième fois, avec la fonction **prompt**, et stocker le résultat dans une variable nommée **motbis***.*
* Dans une variable nommée **identique**, calculer si les deux mots de passe entrés sont identiques (la variable contiendra un boolean **true** si identique et **false** sinon)
* Afficher le message ***Identiques:***suivi du contenu de la variable **identique**.
* Calculer **en une seule expression**, la validité du mot de passe (voir les conditions qu'il doit respecter ci-dessous), le résultat de cette expression sera de type boolean, stocké dans une variable nommée **valid** qui devra valoir **true** si le mot de passe est valide, et **false** sinon.
* Afficher le message ***Validite:***suivi du contenu de la variable **valid**.

**Attention: Pour cet exercice, vous n'êtes pas autorisé à utiliser des structures de contrôle (if, while, ...).**

Conditions de validité du mot de passe:

* il est identique à **motbis**
* sa longueur est entre 4 et 8 caractères
* son premier caractère est différent de son dernier
* son premier caractère est une lettre (soit majuscule ou miniscule)
* son deuxième caractère est non alphanumérique (ni lettre minuscule ou majuscule or chiffre)

Remettez votre fichier **pass.js** sur StudiUM.

Références utiles

* La fonction [**prompt**](https://developer.mozilla.org/en-US/docs/DOM/window.prompt)
* La classe [**String**](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String), particulièrement la section ***String instances*** (indice: voir **length**, **charAt**, et les opérateurs de comparaison entre String)

## Exercice noté 2

Vous devez coder un programme qui calcule une approximation du  
nombre pi en faisant la somme d'une série de nombres. Mathématiquement  
le nombre pi est égal à la somme infinie suivante :

pi = 4/1 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + ...

Remarquez l'alternance des signes d'un terme à l'autre de la série  
(positif, négatif, positif, ...) De plus, remarquez que les diviseurs  
sont les nombres impairs. L'approximation sera la somme des 20000  
premiers termes de cette série. Coder un programme qui fait la somme  
et affiche avec print la valeur obtenue. Votre programme doit être  
complet, avec des commentaires qui indiquent le nom du fichier,  
l'auteur (votre nom), une brève description de l'utilité du programme.  
Il doit aussi y avoir des commentaires qui expliquent à quoi  
correspondent chaque variable, et le fonctionnement du programme.  
Utilisez des énoncés de boucle appropriés, une indentation correcte,  
et des énoncés blocs dans le corps des ifs et boucles. Votre  
programme doit éviter les calculs redondants et répétitifs.

## Exercice noté 3

## Un example de decomposition fonctionelle

Vous devez concevoir et coder une fonction nommée "romain" qui fait la  
conversion d'un nombre entier entre 1 et 3999 en un texte  
correspondant à la numération romaine de ce nombre. Par exemple  
l'appel romain(439) doit retourner le texte "CDXXXIX" et romain(2948)  
doit retourner le texte "MMCMXLVIII". Vous devez faire une  
décomposition en sous-fonctions pour que votre code soit facile à  
comprendre et qu'il évite la duplication de code similaire. En  
particulier vous devez avoir une fonction nommée "repeter" qui prend  
deux paramètres, un entier n et un texte t, et qui retourne un texte  
composé de n répétitions du texte t (par exemple, l'appel  
repeter(3,"I") doit retourner "III"). D'autre part, vous devez coder  
une fonction testRomain qui fait des tests unitaires bien choisis des  
fonctions que vous avez codées. En plus de faire correctement la  
conversion demandée, votre code doit être correctement indenté, il  
doit contenir des commentaires explicatifs et des identificateurs  
significatifs.

Il est à noter que votre code ne fait que définir des fonctions et  
exécuter les tests unitaires. Votre code ne doit pas faire  
d'entrée-sortie (pas d'appel à print, alert ou prompt). Il faut  
s'imaginer que votre code sera utilisé par un autre programmeur dans  
un logiciel plus gros (par exemple un logiciel de traitement de texte  
qui doit numéroter les pages en numérotation romaine).

Voici un indice pour la conception de vos fonctions. La numérotation  
romaine d'un nombre n se décompose en une partie qui représente (dans  
la numérotation arabe décimale) le chiffre des milliers, suivie d'une  
partie qui représente le chiffre des centaines, suivie d'une partie  
qui représente le chiffre des dizaines, et finalement une partie qui  
représente le chiffre des unités.

Ainsi la conversion du nombre 439, qui a 0, 4, 3, et 9 comme chiffres  
arabes des milliers, centaines, dizaines et unités, aura 4 parties :  
"", "CD", "XXX", "IX" qui seront concaténées. Chaque partie contient  
uniquement

- les lettres I, V et X pour les unités (par exemple "VIII" pour 8)  
- les lettres X, L et C pour les dizaines (par exemple "XL" pour 4)  
- les lettres C, D et M pour les centaines (par exemple "CM" pour 9)  
- la lettre M pour les milliers (par exemple "MM" pour 2)

## Exercice noté 4: enregistrement et tableaux

Vous allez construire un arbre généalogique (contenant seulement les mères et les filles) à l'aide des enregistrements et des tableaux.

1) Vous allez commencer par la définition d’un tableau vide appelé **personnes**. C.-à-d. :

var **personnes** = [];

2) Vous devez écrire une procédure nommée ajoutePersonne, qui prend deux paramètres, un nom et un age et créer un enregistrement avec trois champs: nom, age et enfants. Le champ enfant serait initialisé comme un tableau vide. La procédure ajoute l'enregistrement à la fin du tableau **personnes** (par exemple, en utilisant personne.push()). Donc **personnes** sera un tableau qui contient un enregistrement à chaque élément.

-- Utilisez ajoutePersonne pour créer un tableau **personnes** qui contient les données suivantes:

nom, age: "Julie", 100 ; "Sarah",83; "Jennifer", 82; "Olivia", 79; "Marge", 55; "Mathilde",48; "Joanne",45; "Isabelle", 47; "Celine", 23; "Caroline", 29; "Wendy", 24; "Kaliste", 26; "Karine", 22; "Sophie", 28; "Orianne", 25; "Alice", 21;

3) Vous devez écrire une procédure nommée **ajouteEnfant**, qui prend deux paramètres, le nom (chaîne de caractères) du parent et le nom d'un enfant, et ajoute (avec un .push) l'enregistrement associé avec l'enfant dans le champ **enfants** (un tableau) associé avec le parent. Vous pouvez supposer que les deux (parent et enfant) sont déjà dans le tableau **personnes**.

Si on veut indiquer que Sarah est l'enfant de Julie, et on commence avec  **personnes**[0].**nom** = "Julie", **personnes**[0].**enfants** = [] et **personnes**[1].**nom** = "Sarah", l'exécution de **ajouteEnfant**("Julie","Sarah") nous donne:

**personnes**[0].**nom** = "Julie", **personnes**[0].**enfants**[0].**nom**= "Sarah" et aussi **personnes**[0].**enfant**[0].**age** = 83.

Assurez-vous que personnes[0].enfants[0] et personnes[1] faire référence au même objet (leurs pointeurs pointent vers la même place en mémoire).

-- Utilisez **ajouteEnfant** pour ajouter aux champs **enfants** les informations suivantes:

Enfants de Julie: Sarah, Jennifer, Olivia;

Enfants de Sarah:  Marge, Mathilde;

Enfant de Jennifer:  Joanne;

Enfant de Olivia: Isabelle;

Enfant de Marge: Celine, Caroline, Wendy;

Enfant de Mathilde:  Kaliste, Karine, Sophie;

Enfant de Isabelle:  Orianne,  Alice;

4) Vous devez écrire une fonction nommée **quiEnfants**, qui prend comme paramètre le nom (en string) de quelqu'un dans le tableau **personnes** (par exemple, "Isabelle") et **retourne** (avec return) un tableau qui contient les noms (en string) de ses enfants par**ordre croissant de l'age**.

Exemple:

**quiEnfants**("Isabelle")  retourne: ["Alice", "Orianne"]

5) Vous devez écrire une fonction nommée **quiPetitsEnfants**qui prend comme paramètre le nom (en string) de quelqu'un dans le tableau **personnes** (par exemple, "Julie") et **retourne** (avec return) un tableau qui contient les noms (en string) des petits enfants par**ordre croissant de l'age**.

Exemple:

**quiPetitsEnfants**("Sarah")  retourne: [ "Karine", "Celine", "Wendy", "Kaliste", "Sophie", "Caroline" ]

La fonction quiPetitsEnfant devrait appeller quiEnfant.

Votre code ne doit pas faire d'entrée-sortie (pas d'appel à print, alert ou prompt). Il faut imaginer que votre code sera utilisé par un autre programmeur dans un logiciel plus gros.

Comme toujours, votre programme doit être complet, avec des commentaires qui indiquent le nom du fichier, l'auteur (votre nom), une brève description de l'utilité du programme. Il doit aussi y avoir des commentaires qui expliquent à quoi correspond chaque variable, et le fonctionnement du programme

Vous devez implémenter une variante du tri insertion vu en cours où l'on s’intéresse à trier des valeurs entières en ordre décroissant et en place, c'est-à-dire en modifiant directement le tableau passé en argument. Le tri insertion classique (celui vu en cours) considère une liste de plus en plus grande et insère le nouvel élément de la liste considérée à la position où il doit se trouver par un glissement des valeurs inférieures. Le glissement implique que de nombreuses affectations sont effectuées sur le tableau à trier afin de déplacer certaines valeurs à leur place. La variante que vous devez écrire effectue plusieurs passages de l'algorithme de tri insertion, où chaque passage effectue les glissements des valeurs du tableau distantes d'une certaine valeur n appelée le régime.  
  
Par exemple, un glissement de régime 5 commençant à la position i devrait seulement modifier les valeurs du tableau aux positions *i*, *i-5*, *i-10*, *i-15*, ...  
  
L'idée est de commencer par un régime élevé (par exemple 11) afin de faire glisser des valeurs éloignées (indice 0, 11, 22, etc.) dans le tableau, puis de baisser progressivement le régime, pour terminer avec un régime 1 qui correspond à l'algorithme de tri insertion classique (où les valeurs successives sont glissées).  
  
Vous devez écrire une fonction **insertionShell** qui admet dans cet ordre deux arguments qui sont le **tableau à trier** et un **tableau de régimes**, c'est-à-dire, un tableau de valeurs entières strictement décroissantes et terminant par l'entier 1. Votre fonction doit retourner **un tableau qui contient le nombre d'affectations effectuées dans le tableau à trier à chaque régime**: la ieme valeur de ce tableau correspond donc au nombre d'affectations effectuées par votre algorithme au tableau à trier pour la ieme valeur dans le tableau de régime.  
  
Par exemple, si on reprend les noms de variables du tri insertion classique, **TOUT** *t[i] = memo* ou *t[i] = t[j]* comptent comme des affectations au tableau à trier.  
  
Voici deux exemples d'appels de votre fonction, ainsi que le détail des différentes étapes de l'algorithme. Dans le premier appel, les valeurs distantes de 5 commencent par être comparées et 78 affectations sont effectuées. Le régime 3 est ensuite considéré et 42 affectations sont effectuées; enfin le régime 1 est effectué (un tri insertion classique), ce qui requiert 46 affectations dans le tableau. Un total de 166 affectations ont donc été effectuées alors que le tri insertion classique requiert 332 affectations, tel que mesuré par le second appel.  
  
    print(insertionShell([1,3,2,5,4,6,7,8,10,41,43,49,22,13,24,40,20,9,9,10,1,13,45,25,43,12,48,47,25,7], [5,3,1])  
  
    regime 5:  
43,49,47,25,43,40,48,45,25,41,12,20,22,13,24,6,13,9,10,10,1,7,8,9,7,1,3,2,5,4  
    regime 3:  
48,49,47,43,45,40,41,43,25,25,13,24,22,13,20,10,12,9,7,10,9,7,8,4,6,5,3,2,1,1  
    regime 1:  
49,48,47,45,43,43,41,40,25,25,24,22,20,13,13,12,10,10,9,9,8,7,7,6,5,4,3,2,1,1  
78, 42, 46  
  
    print(insertionShell([1,3,2,5,4,6,7,8,10,41,43,49,22,13,24,40,20,9,9,10,1,13,45,25,43,12,48,47,25,7], [1]));  
    332  
       
    regime 1: 49,48,47,45,43,43,41,40,25,25,24,22,20,13,13,12,10,10,9,9,8,7,7,6,5,4,3,2,1,1  
  
Vous devez également écrire une fonction **best** qui prend en argument un t**ableau** ainsi qu'un **tableau de tableaux de régimes** et qui retourne le t**ableau de régimes le meilleur**, c'est-à-dire celui entrainant le moins d'affectations pour le tableau donné en argument. Voici un exemple non normatif (vous pouvez afficher les choses différemment, voire ne rien afficher) d'appel où le meilleur régime est [7,3,1]  
  
    var regimes = [[1],[5,4,3,2,1],[7,3,1],[11,9,7,3,1]];  
    var t = [2,4,5,7,8,2,5,8,9,12,13,2,4,56,12,5,7,23,12,24,34,56,3,6,1];  
    print(best(t,regimes));  
    7,3,1  
       
    tri selon [ 1 ] en 208  affectations  
    tri selon [ 5,4,3,2,1 ] en 158 affectations  
    tri selon [ 7,3,1 ] en 127 affectations  
    tri selon [ 11,9,7,3,1 ] en 129 affectations  
  
Votre programme ne sera pas noté sur la trace (ça peut cependant vous aider à vérifier le bon fonctionnement de votre algorithme). Vos fonctions doivent cependant correspondre aux niveaux des entrées et des sorties avec leur description.

## Exercice noté 6

Vous devez écrire une fonction récursive `change` qui prend en paramètre un nombre et un tableau de nombre en ordre décroissant. Le nombre donné en paramètre correspond au change à donner en cents et le tableau de nombre correspond aux devises disponibles en cents.  
  
Votre travail est d'énumérer (afficher) chacune des combinaisons de pièces pour le change demandé.  
  
Aucune combinaison ne devrait être affichée si aucune combinaison de devises ne  
permet de retourner le change exact.  
  
Les devises affichées doivent être ordonnées en ordre décroissant lors de l'affichage.  
Votre énumération doit être exhaustive et sans répétition de combinaisons.  
Votre fonction doit prioriser l'usage des devises les plus grandes dans l'énumération.  
  
Il est conseillé d'utiliser un interprète de ligne de commande pour éviter les pannes (crash) associées à des scripts demandant en performance à l'intérieur d'un navigateur.  
  
Exemple:  
```  
d8> change(11, [100, 25, 20, 10, 5, 1])  
10 1   
5 5 1   
5 1 1 1 1 1 1   
1 1 1 1 1 1 1 1 1 1 1   
```  
  
Exemple 2:  
```  
d8> change(36, [50, 12, 3])  
12 12 12   
12 12 3 3 3 3   
12 3 3 3 3 3 3 3 3   
3 3 3 3 3 3 3 3 3 3 3 3   
```  
  
Exemple 3:  
```  
d8> change(95, [100,25,10,5])  
25 25 25 10 10   
25 25 25 10 5 5   
25 25 25 5 5 5 5   
25 25 10 10 10 10 5   
25 25 10 10 10 5 5 5   
25 25 10 10 5 5 5 5 5   
25 25 10 5 5 5 5 5 5 5   
25 25 5 5 5 5 5 5 5 5 5   
25 10 10 10 10 10 10 10   
25 10 10 10 10 10 10 5 5   
25 10 10 10 10 10 5 5 5 5   
25 10 10 10 10 5 5 5 5 5 5   
25 10 10 10 5 5 5 5 5 5 5 5   
25 10 10 5 5 5 5 5 5 5 5 5 5   
25 10 5 5 5 5 5 5 5 5 5 5 5 5   
25 5 5 5 5 5 5 5 5 5 5 5 5 5 5   
10 10 10 10 10 10 10 10 10 5   
10 10 10 10 10 10 10 10 5 5 5   
10 10 10 10 10 10 10 5 5 5 5 5   
10 10 10 10 10 10 5 5 5 5 5 5 5   
10 10 10 10 10 5 5 5 5 5 5 5 5 5   
10 10 10 10 5 5 5 5 5 5 5 5 5 5 5   
10 10 10 5 5 5 5 5 5 5 5 5 5 5 5 5   
10 10 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5   
10 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5   
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

Exercice noté 7

Vous devez mettre en oeuvre une fonction récursive **postfix** fait retourne la chaine de caractères correspondant à l'affichage post-fixe d'une structure d'arbre binaire donnée en paramètre.

Un arbre est simplement une structure formée d'enregistrements (référés à l'avenir comme des noeuds) de la forme

{val: , gauche: , droite: }

où gauche et droite contiennent soit la valeur `null` soit un noeud.

Par exemple

var arbre = noeud( // 7

7, // / \

noeud(3, noeud(1), noeud(2)), // 3 6

noeud(6, noeud(4), noeud(5)) // / \ / \

}; // 1 2 4 5

// / \ / \ / \ / \

// # # # # # # # #

print(postfix(arbre)) // affiche: 1 2 3 4 5 6 7

L'idée d'un affichage post-fixe est relativement simple: il suffit de visiter récursivement d'abord l'attribut de gauche, puis celui de droite pour finir avec val.

On vous fournit le code suivant facilitant la création d'arbre.

Code fournis

function noeud(val, gauche, droite) {

if (typeof gauche === "undefined") gauche = null;

if (typeof droite === "undefined") droite = null;

return {val: val, gauche: gauche, droite: droite};

}