TP4 : Découverte du JavaScript

Avant-Propos

Créer un dossier `TP4` dans le projet `WebStorm`.

Pour chacun des exercices suivants, créer un fichier `js` dans le dossier `TP4`.

Pour choisir un fichier `js` à exécuter, taper `Alt` + `Maj` + `F9` (ou dans le menu `Run > Debug...`).

Pour exécuter de nouveau le fichier sélectionné, taper `Maj` + `F9`.

Le mode strict est à utiliser obligatoirement pour ces exercices.

Exercice 1 : Volumes triés de cylindres

Le but de cet exercice est de trier des volumes de cylindres définis par leurs hauteurs et rayons respectifs.

1. Créer une fonction renvoyant le volume d'un cylindre pour une hauteur `h` et un rayon `r` donnés. Pour rappel, le volume est donné par la formule suivante :

   $$v = \pi r^2 h.$$

   Indication : utiliser `Math.PI`.

   ```
   function vol_cylindre(h, r) {
     // à compléter
   }
   ```

2. Afficher dans la console le volume d'un cylindre de hauteur `2` et de rayon `3`.
   Indications : utiliser la méthode `console.log`. Le résultat attendu dans la console est le suivant :

   ```
   56.548667764616276
   ```

3. Créer une fonction renvoyant un tableau contenant les objets ayant les propriétés `h` (hauteur), `r` (rayon) et `vol` (volume d'un cylindre de hauteur `h` et de rayon `r`), dont les hauteurs sont comprises entre les paramètres `h_min` et `h_max`, dont les rayons sont compris entre les valeurs des paramètres `r_min` et `r_max`, avec des pas respectifs `pas_r`

et pas_h. Si pas_r et pas_h ne sont pas définis lors de l'appel, ils devront prendre la valeur 1.

Ce tableau devra être trié en fonction du volume croissant.

Indication : à l'aide de deux boucles imbriquées, créer tous les objets de la forme {h : x, r : y, vol : z} avec les valeurs x, y et z décrites ci-avant. Utiliser ensuite la méthode sort.

```
function vol_cylindre_tab(h_min, h_max, r_min, r_max, pas_h, pas_r) {
  // à compléter
}
```

On appelle t le tableau obtenu pour h_min = r_min = 1 et h_max = r_max = 3.

4. Afficher dans la console le tableau t.
   Le résultat attendu dans la console est le suivant :

```
> Array(9) [Object, Object, Object, Object, Object, Object, Object, Obj
```

5. Afficher dans la console le tableau obtenu après avoir transformé le contenu de t en chaîne de caractères.
   Indications : utiliser les méthodes map et JSON.stringify. Le résultat attendu dans la console est le suivant :

```
> Array(9) ["{"h":1,"r":1,"vol":3.141592653589793}", "{"h":2,"r":1,"vol
```

6. Pour chaque valeur contenue dans t, afficher dans la console une chaîne de caractères de la forme "Le volume d'un cylindre de rayon rcm et de hauteur hcm est volcm³".
   Indication : utiliser la méthode forEach. Pour obtenir une expression concise, il est possible d'utiliser des littéraux de gabarits (ou template strings). Le résultat attendu dans la console est le suivant :

```
Le volume d'un cylindre de rayon 1cm et de hauteur 1cm est 3.141592653
Le volume d'un cylindre de rayon 1cm et de hauteur 2cm est 6.283185307
Le volume d'un cylindre de rayon 1cm et de hauteur 3cm est 9.424777960
Le volume d'un cylindre de rayon 2cm et de hauteur 1cm est 12.56637061
Le volume d'un cylindre de rayon 2cm et de hauteur 2cm est 25.13274122
Le volume d'un cylindre de rayon 3cm et de hauteur 1cm est 28.27433388
Le volume d'un cylindre de rayon 2cm et de hauteur 3cm est 37.69911184
```

```
Le volume d'un cylindre de rayon 3cm et de hauteur 2cm est 56.548667764
Le volume d'un cylindre de rayon 3cm et de hauteur 3cm est 84.82300164
```

Exercice 2 : mesures de performance (calcul de la factorielle)

Le but de cet exercice est de mesurer les performances de trois implantations différentes du calcul de la factorielle d'un nombre.

1. Implanter le calcul de la factorielle d'un nombre n en trois fonctions différentes :
    1. Une version itérative ;
    2. Une version récursive (non récursive terminale) ;
    3. Une version récursive terminale.
2. Utiliser la fonction suivante pour mesurer les différences de performance de ces implantations :

```
function log_mesure(f, n) {
  const nb_repet = 100
  let debut = performance.now()
  for (let k = 0; k <= nb_repet; k++) f(n)
  let fin = performance.now()
  console.log(`${nb_repet} appels à ${f.name}(${n}) prennent ${fin - d
}
```

Exercice 3 : mesures de performance (suite de Padovan)

Le but de cet exercice est de mesurer les performances de quatre implantations différentes du calcul de valeurs de la suite de Padovan.

La suite de Padovan est définie récursivement comme suit :

$$P(n) = \begin{cases} 1 & \text{si } n \in \{0, 1, 2\}, \\ P(n-2) + P(n-3) & \text{sinon.} \end{cases}$$

1. Implanter deux fonctions itératives calculant un tableau contenant les `(n + 1)` premières valeurs de la suite de Padovan où la dernière valeur :
    1. est en tête de tableau *(utiliser `unshift`)* ; autrement dit, le résultat attendu est
    $$[P(n), P(n-1), \ldots, P(1), P(0)];$$
    2. est en queue de tableau *(utiliser `push`)* ; autrement dit, le résultat attendu est
    $$[P(0), P(1), \ldots, P(n-1), P(n)].$$
2. Implanter le calcul de la `(n + 1)`^{e} valeur P(n) de la suite de Padovan en quatre fonctions différentes :

    1. Une version itérative ;
    2. Une version récursive (non récursive terminale);
    3. Une version renvoyant la tête du tableau calculé dans la question 1.1 précédente ;
    4. Une version renvoyant la queue du tableau calculé dans la question 1.2 précédente.
3. Mesurer les performances à l'aide de la fonction `log_mesure` de l'exercice précédent.

Exercice 4 : calculs de notes

Le but de cet exercice est de calculer des valeurs statistiques à partir de tableaux de valeurs, en ne détruisant pas les données initiales.

1. Implanter une fonction renvoyant un **entier** aléatoire compris entre deux bornes.
   Indication : utiliser les méthodes `Math.ceil`, `Math.floor` et `Math.random`.

   ```
   function randomInt(min, max) {
     // à compléter
   }
   ```

2. Coder une fonction renvoyant une structure aléatoire composée d'un champ `note` (un entier aléatoire entre 0 et 20) et d'un champ `coeff` (un entier aléatoire entre 1 et 100).

   ```
   function alea_note() {
     // à compléter
   }
   ```

   Ainsi, l'instruction

   ```
   console.log(JSON.stringify(alea_note()));
   ```

   devrait produire un résultat de la forme :

   ```
   > {"note":11,"coeff":38}
   ```

3. Coder une fonction renvoyant un tableau contenant `n` notes aléatoires.

   ```
   function alea_tab_notes(n) {
     // à compléter
   }
   ```

   Ainsi, la séquence d'instructions

   ```
   let tab = alea_tab_notes(5);
   console.log(tab.map(JSON.stringify));
   ```

devrait produire un résultat de la forme :

```
> Array(5) ["{"note":11,"coeff":67}", "{"note":5,"coeff":66}", "{"note'
```

4. Coder une fonction renvoyant un tableau contenant n fois la même valeur, définie à partir des paramètres de la fonction.
   Indication : utiliser la méthode `fill`.

```
function const_tab_note(note, coeff, longueur) {
  // à compléter
}
```

Ainsi, la séquence d'instructions

```
let tab = const_tab_note(12, 2, 5);
console.log(tab.map(JSON.stringify));
```

devrait produire le résultat suivant :

```
> Array(5) ["{"note":12,"coeff":2}", "{"note":12,"coeff":2}", "{"note"
```

5. Implanter une fonction renvoyant la somme des coefficients d'un tableau de notes.
   Indication : utiliser la méthode `reduce`.

```
function poids_coeff(tab) {
  // à compléter
}
```

6. Coder une fonction calculant la moyenne pondérée des notes contenues dans un tableau de notes.
   Indication : essayer de ne réaliser qu'un seul parcours des valeurs en utilisant une structure adaptée et la fonction `reduce`.

```
function moyenne(tab) {
  // à compléter
}
```

7. Calculer la note médiane d'un tableau de notes. **Attention, cette fonction ne doit pas modifier l'état du tableau.** *Indication : pour calculer la médiane, il suffit de trier le tableau, puis de sélectionner la case du milieu si la longueur du tableau est impaire, ou la moyenne des deux cases au centre du tableau sinon.*

```
function mediane(tab) {
  // à compléter
}
```

8. Vérifier les deux dernières fonctions à l'aide des deux fonctions de générations de tableau de notes.

Exercice 5 : heuristique des plus proches voisins

Dans cet exercice, on essaye de réduire la distance à parcourir pour se déplacer entre des points repérés par des coordonnées. Le but est de traverser tous les points depuis le premier, en essayant de réduire la distance totale. Ce problème est connu sous le nom de problème du voyageur de commerce.

Ici, nous utiliserons une heuristique gloutonne. Le successeur d'un point visité sera ainsi le plus proche des points restants à visiter.

1. Implanter une fonction renvoyant un nombre aléatoire compris entre deux bornes. **Le nombre généré n'est pas nécessairement entier.**

   ```
   function randomNumber(min, max) {
     // à compléter
   }
   ```

2. Coder une fonction permettant de générer aléatoirement une structure composée de deux champs x et y (une coordonnée), nombres compris entre deux bornes données en paramètres.

   ```
   function randomCoord(min, max) {
     // à compléter
   }
   ```

   Ainsi, l'instruction

   ```
   console.log(JSON.stringify(randomCoord(0,100)));
   ```

   devrait produire un résultat de la forme :

   ```
   > {"x":78.04020249032833,"y":15.883077167921144}
   ```

3. Implanter une fonction renvoyant un tableau contenant n coordonnées aléatoires.

   ```
   function randomCoords(min, max, length) {
     // à compléter
   ```

```
}
```

Ainsi, la séquence d'instructions

```
let tab = randomCoords(0, 100, 5);
console.log(tab.map(JSON.stringify));
```

devrait produire un résultat de la forme :

```
> Array(5) ["{"x":82.49216619367243,"y":16.59048497571074}", "{"x":92.
```

4. Coder deux fonctions `sqrDist` et `dist` qui, pour deux coordonnées `c1` et `c2`, renvoient respectivement :
    1. le carré de la distance entre `c1` et `c2` ;
    2. la distance entre `c1` et `c2`.
5. Coder une fonction `tab_dist` renvoyant longueur totale d'un trajet fixé selon un tableau de coordonnées, calculée comme la somme des distances entre chacun des points. Plus précisément, `tabDist([c1, c2, ..., cn]) === dist(c1, c2) + dist(c2, c3) + ... + dist(cn-1, cn)`.
6. Implanter une fonction `plus_pres_de` prenant trois coordonnées `c`, `c1` et `c2` et renvoyant :
    - `-1` si `c1` est plus près de `c` que `c2` ;
    - `1` si `c2` est plus près de `c` que `c1` ;
    - `0` si `c` est à égale distance de `c1` et `c2`.
7. Implanter une fonction triant un tableau de coordonnées de la plus proche à la plus lointaine d'une coordonnée `c` donnée.

```
function tri_plus_proche(c, tab) {
  // à compléter
}
```

8. Implanter une fonction triant un tableau de coordonnées de la plus lointaine à la plus proche d'une coordonnée `c` donnée.

```
function tri_plus_loin(c, tab) {
  // à compléter
}
```

9. Réaliser le tri glouton d'un tableau en considérant la méthode suivante :
    - la première case est la première coordonnée du tableau ;

- le successeur d'un point visité est le plus proche des points restants à visiter.

Cette fonction renvoie un nouveau tableau et ne doit pas modifier le tableau de départ.

10. Vérifier empiriquement que le tri glouton d'un tableau et celui de son inverse ne correspondent pas nécessairement à des trajets de même longueur (en considérant la fonction précédente `tabDist`).