

## PARTIE 1 : Algorithmes

### 1.1 FizzBuzz

Objectif : Afficher une suite de nombres, mais en modifiant certains nombres selon les règles :

- Si le nombre est divisible par 3 → affiche Fizz
- Si le nombre est divisible par 5 → affiche Buzz
- Si le nombre est divisible par 3 ET 5 → affiche FizzBuzz
- Sinon, affiche le nombre lui-même



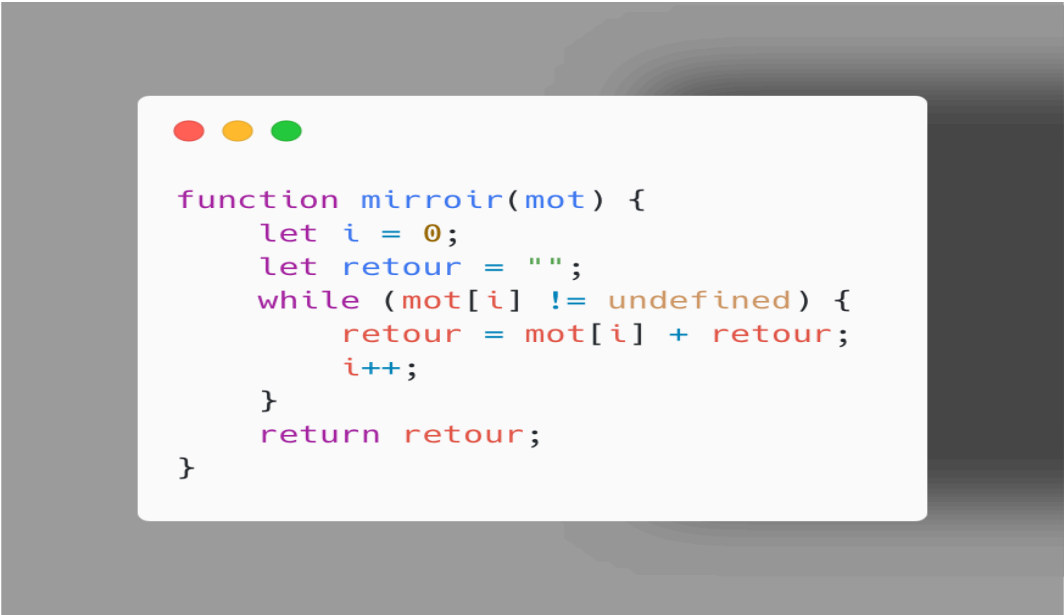
```
function fizzBuzz(n) {  
  for (let i = 1; i <= n; i++) {  
    if (i % 3 === 0 && i % 5 === 0) {  
      console.log("FizzBuzz");  
    } else if (i % 3 === 0) {  
      console.log("Fizz");  
    } else if (i % 5 === 0) {  
      console.log("Buzz");  
    } else {  
      console.log(i);  
    }  
  }  
}
```

## 1.2 Palindrome Checker

**Objectif :** Écrire une fonction qui vérifie si une chaîne de caractères est un palindrome.

Dans un premier temps, j'ai choisi de décomposer le problème pour bien structurer ma logique.

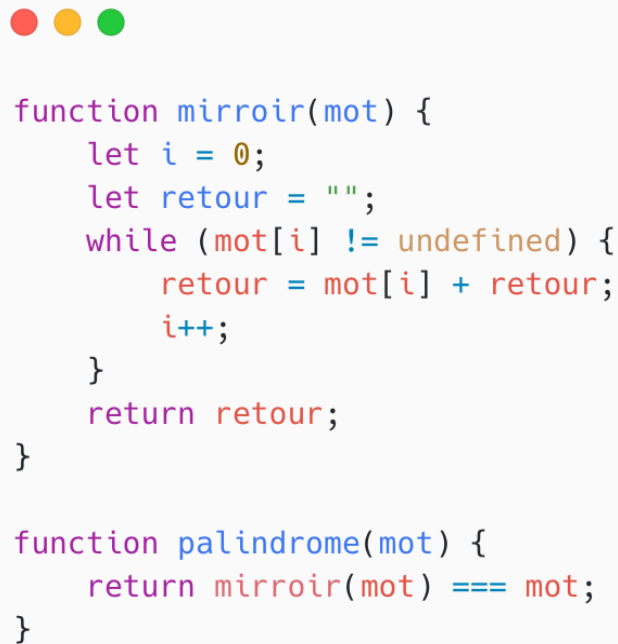
J'ai commencé par créer une fonction `mirroir(mot)`, dont le but est d'inverser une chaîne de caractères.



```
function mirroir(mot) {  
  let i = 0;  
  let retour = "";  
  while (mot[i] !== undefined) {  
    retour = mot[i] + retour;  
    i++;  
  }  
  return retour;  
}
```

Ensuite, j'ai utilisé cette fonction `mirroir` dans une seconde fonction appelée `palindrome(mot)`, qui permet de comparer le mot original avec sa version inversée.

Si les deux sont identiques, alors le mot est un palindrome, sinon ce n'en est pas un.



```
function mirroir(mot) {  
  let i = 0;  
  let retour = "";  
  while (mot[i] !== undefined) {  
    retour = mot[i] + retour;  
    i++;  
  }  
  return retour;  
}  
  
function palindrome(mot) {  
  return mirroir(mot) === mot;  
}
```

Enfin, pour aller plus loin, j'ai réécrit la même logique en JavaScript moderne, en utilisant les fonctions prédéfinies du langage comme **.split()**, **.reverse()** et **.join()** pour simplifier la version précédente.

Cette version permet de résumer et d'optimiser la logique développée manuellement avec **mirroir**.



```
function palindrome(mot) {  
  let clean = mot.toLowerCase().replace(/^[a-z0-9]/g, '');  
  let inverse = clean.split('').reverse().join('');  
  return clean === inverse;  
}
```

## 1.3 Quick Sort

**Objectif :** Implémenter l'algorithme QuickSort (tri rapide), un algorithme de tri récursif basé sur la stratégie de *divide-and-conquer* :



```
function quickSort(arr) {  
  if (arr.length <= 1) {  
    return arr;  
  }  
  
  let pivot = arr[0];  
  let left = [];  
  let right = [];  
  
  for (let i = 1; i < arr.length; i++) {  
    if (arr[i] <= pivot) {  
      left.push(arr[i]);  
    } else {  
      right.push(arr[i]);  
    }  
  }  
  
  return quickSort(left).concat([pivot], quickSort(right));  
}
```

**Question** : Que penses-tu de QuickSort ? Quels sont ses avantages/inconvénients ?  
L'utilises-tu dans tes projets en vrai ?

Je n'avais jamais utilisé l'algorithme QuickSort auparavant.  
C'est donc une belle découverte technique qui m'a permis de mieux comprendre comment fonctionne un tri récursif basé sur la stratégie du "divide and conquer".

J'ai appris que l'algorithme consiste à :

- Choisir un pivot (généralement le premier élément),
- Diviser le tableau en deux sous-listes :
  - Les éléments inférieurs ou égaux au pivot,
  - Les éléments supérieurs au pivot,

- Puis trier récursivement ces sous-listes,
- Et enfin reconstruire la liste triée en combinant les éléments triés avec le pivot.

Ce raisonnement m'a permis de mieux comprendre la puissance de la récursivité, ainsi que la différence entre des tris dits "naïfs" comme le tri à bulle.

C'était donc un exercice enrichissant, qui m'a aidé à progresser dans ma logique algorithmique.