ENSEIRB-Matmeca
2021-2022

1ère année - Filière Info
PG104

TD n°2 - Portée et récursivité

Exercice 1: Portée et chiffrement

Considérons la fonction suivante, permettant d'encoder des chaînes de caractère en utilisant un chiffrement de César (`https://en.wikipedia.org/wiki/Caesar_cipher`).

```
function cypher (str, offset) {
    let res = '';
    const codeA  = 'a'.charCodeAt(0); // character code of 'a'
    const extent = 26;                // number of letters in alphabet
    const realoffset = ((offset % extent) + extent) % extent;
    for (let i = 0; i < str.length; i++) {
        const code   = str.charCodeAt(i);
        if ((code >= codeA) && (code <= codeA + extent)) {
            const newcode = ((code-codeA+realoffset) % extent) + codeA;
            res += String.fromCharCode(newcode);
        } else {
            res += str[i];
        }
    }
    return res;
}
```

Le principe du chiffrement de César est excessivement simple, puisqu'il consiste à permuter de manière circulaire les lettres de l'alphabet un certain nombre de fois.

```
cypher('abcde', 1);  // → 'bcdef'
cypher('bcdef', -1); // → 'abcde'
cypher('abcde', 13); // → 'nopqr'
cypher('nopqr', 13); // → 'abcde'
```

1. Quelle sont les variables définies dans la fonction `cypher` ? Indiquer pour chacune d'entre elles leurs portées.
2. Rappeler la différence entre `let` et `const`. Sachant que les nombres sont des valeurs immutables, quel est l'intérêt de définir des variables comme `const` ?

Supposons vouloir réutiliser la fonction `cypher` pour produire une fonction `hidden_cypher` pour encoder des messages et dans laquelle la clé de chiffrement est invisible de l'extérieur. La clé serait ici un nombre entier compris entre 1 et 25. Une possibilité, en réutilisant le code précédent, consisterait à faire :

```
const key = 13;
function hidden_cypher(str) {
    return cypher(str, key);
}
```

3. Quelle est la portée de `key` dans ce code ? En quoi une telle portée peut-elle être gênante ?
4. Modifier le code, de manière à pouvoir faire que `key` soit inaccessible au reste du code.
5. Modifier le code de manière à renvoyer à la fois la fonction d'encodage et la fonction de décodage.

▷ Dans chacun des exercices suivants, l'idée est de construire un algorithme récursif pour résoudre le problème. Il est demandé, pour chacun de ces exercices, de décomposer le problème en un sous-problèmes de même nature.

Par exemple, si on considére le problème suivant : écrire une fonction `fact` qui calcule la factorielle de l'entrée passé en paramètre. Alors l'analyse se fait comme suit :

— le calcul de `fact(n)` peut se faire facilement à partir du calcul de `fact(n-1)`, qui est naturellement un problème de même nature, à travers le code suivant :

```
const res = n * fact(n-1);  // res is equal to fact(n)
```

— lorsque l'entier `n` est nul, alors le résultat de la fonction est évident et renvoie `1`.

Cela implique dont le code suivant pour `fact` :

```
// compute the factorial of the integer n
function fact(n) {
  if (n <= 1)
    return 1;
  else
    return n * fact(n-1);
}
fact(5);  // → 120
fact(10); // → 3628800
```

Exercice 2: Calcul du pgcd

Écrire une fonction `pgcd` donnant le plus grand diviseur commun de ses deux arguments (des entiers positifs), en se basant sur l'algorithme d'Euclide, rappelé ci-après :

- le PGCD de a et de b est égal au PGCD de b et a ;
- le PGCD de a et de 0 est égal à a ;
- lorsque $b > 0$, le PGCD de a et b est égal au PGCD de b et de (a mod b).

Exercice 3: Sommes de nombres

Soit $\text{sum}(x, y)$ la fonction définie par :

$$\text{sum}(x, y) = \sum_{k=x}^{y} k$$

1. Écrire l'équation reliant $\text{sum}(x, y)$ et $\text{sum}(x + 1, y)$, et la compléter de manière à définir entièrement sum par des équations.
2. Écrire une fonction récursive `sumInteger` qui, appelée avec deux entiers a et b comme arguments, renvoie $\text{sum}(a, b)$.

   Exemple :

```
sumInteger(1, 5); // → 15
sumInteger(5, 1); // → 0
```

3. De même, écrire une fonction récursive `sumSquares` qui calcule la somme des carrés des entiers situés entre ses deux arguments.

   Exemple :

```
sumSquares(1, 5); // → 55
sumSquares(5, 1); // → 0
```

4. Écrire une fonction récursive `sumGeneric` qui calcule la somme des $f(k)$ pour les entiers k situés entre ses deux arguments.

Exercice 4: Calculs de puissance

1. Définir une fonction récursive `powerLinear` calculant la puissance n-ème d'un nombre x en $n-1$ multiplications.

   Exemple :

```
powerLinear(2, 10); // → 1024
powerLinear(7, 18); // → 1628413597910449
```

2. Écrire la version améliorée de cette fonction `powerLogarithmic` faisant le calcul en au pire $\lceil \log_2(n) \rceil$ opérations.

Rappel : ne pas oublier de tester vos fonctions.

Exercice 5: Even / Odd

Supposons vouloir définir deux fonctions `myEven` et `myOdd` qui déterminent la parité de leur argument suivant le principe suivant :

- 0 est pair (et donc non impair),
- $n > 0$ est pair (respectivement, impair) si $n-1$ est impair (respectivement, pair).

1. Définir ces fonctions en Javascript grâce à des conditionnelles `if`.
2. Écrire ces fonctions à la main l'aide de formules logiques (utiliser les connecteurs booléens usuels de conjonction, disjonction et négation).

   Réécrire en Javascript une deuxième version sans instructions conditionnelles.

Exercice 6: Suite de Syracuse

En réutilisant les fonctions `myEven` et `myOdd` écrites dans l'exercice précédent, écrire la fonction `syracuse` qui donne la longueur d'un vol de la suite de Syracuse partant de la valeur n, sachant que cette longueur est donnée par :

- `syracuse`$(1) = 0$
- `syracuse`$(n) = 1 +$ `syracuse`$(n/2)$ si n est pair

- `syracuse`$(n) = 1 +$ `syracuse`$(1 + 3 \times n)$ si n est impair

Exemple :

```
syracuse(7); // → 16 as the flight is [7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]
```

Exercice 7: Bases numériques (récursif)

1. Écrire une fonction `convert` utilisant un algorithme *récursif* et transformant un nombre entier en la chaîne de caractères de sa représentation binaire.

   Exemple :

   ```
   convert(666) ;; // → '1010011010'
   ```

2. Étendre cette fonction pour qu'elle fonctionne en n'importe quelle base ≤ 9.

   Exemple :

   ```
   convert2Base(666, 2)  ;; // → '1010011010'
   convert2Base(666, 3)  ;; // → '220200'
   ```

3. (Bonus) Gérer les bases ≤ 35 (l'utilisation de la méthode `charCodeAt()` pourrait servir).

   Exemple :

   ```
   convert2Base(666, 16) ;; // → '29A'
   ```

4. Pour tester votre fonction, écrire la fonction `unconvert` qui prend une chaîne de caractères et une base et calcule la valeur entière correspondante.

Exercice 8: Anadromes

Écrire des fonctions *récursives* sur les chaînes de caractères permettant de décider si :

1. Une chaîne est un palindrome.
2. Une chaîne est l'anagramme d'une autre.