

# Algorithmes et Pensée Computationnelle

## Consolidation 2

Les exercices de cette série sont une compilation d'exercices semblables à ceux vus lors des semaines précédentes. Le but de cette séance est de consolider les connaissances acquises lors des travaux pratiques des dernière semaines.

### Question 1: (🕒 10 minutes) Complexité

Analysez la complexité pour les deux codes suivants. Est-ce la même pour `fun()` et `fun2()` ? Pourquoi ?

```
1 def fun(n):
2     for i in range(n):
3         for j in range(n):
4             print(n)
5
6 def fun2(n):
7     for i in range(n):
8         print(n)
9     for j in range(n):
10        print(n)
```

#### >\_ Solution

Non, ils n'ont pas la même complexité. `fun()` a une complexité de  $O(n*n)$  car il s'agit de deux boucles imbriquées. En revanche, `fun2()` a une complexité de  $O(n+n)$  car il s'agit de deux boucles indépendante.

### Question 2: (🕒 10 minutes) Complexité

Quel est la complexité de ce code ?

```
1 def fun(k):
2     if k == 1:
3         print('Done')
4     else:
5         k = k/2
6     fun(k)
```

#### >\_ Solution

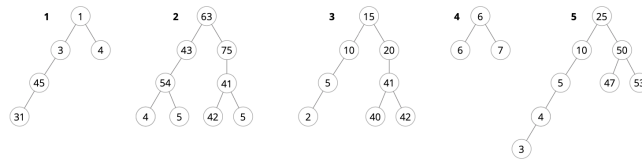
### Question 3: (🕒 10 minutes)

Ecrivez un programme python qui permet d'imprimer tous les nombres impairs à partir de 1 jusqu'à un nombre `n` défini par l'utilisateur (qui doit être supérieur à 1) : Exemple : si `n = 6`, résultat attendu : 1, 3, 5

#### >\_ Solution

```
1 def displayOddNumbers(limit):
2     for nb in range(limit+1):
3         if nb % 2 == 1:
4             print(nb)
5
6 limit = int(input("What is the limit of the function displayOddNumbers(limit)? "))
7
8 print("Odd numbers between 0 and " + str(limit) + ": ")
9 displayOddNumbers(limit)
```

**Question 4: (🕒 )** Lesquels (ou lequel) de ces arbres est un arbre binaire (binary tree) ? Donnez leur hauteur (height).



### >\_ Solution

1. Il ne s'agit pas d'un arbre binaire car il n'y a pas de condition qui est remplie à chaque noeud. Sa hauteur est de 3.
2. Il ne s'agit pas d'un arbre binaire car il n'y a pas de condition qui est remplie à chaque noeud. Sa hauteur est de 3.
3. Il s'agit d'un arbre binaire car une condition peut être appliquée à chaque noeud. Sa hauteur est de 3.
4. Il s'agit d'un arbre binaire car une condition peut être appliquée à chaque noeud. Sa hauteur est de 1.
5. Il s'agit d'un arbre binaire car une condition peut être appliquée à chaque noeud. Sa hauteur est de 4.

### Question 5: (🕒 10 minutes) Trie complexité

Trier la liste de fonctions suivante selon leur croissance asymptotique :

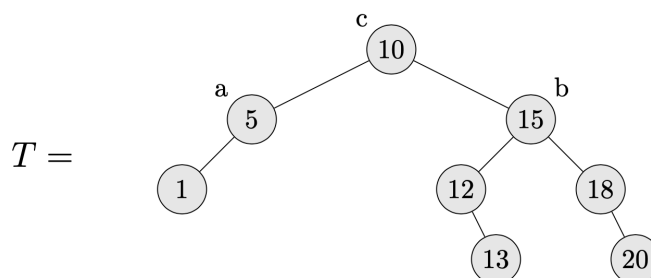
$$n^{\sqrt{n}}, n \cdot \log(n), n^{1/\log(n)}, \log(\log(n)), \sqrt{n}, 3^n/n^5, 2^n \quad (1)$$

### >\_ Solution

- D'abord la constante :  $n^{1/\log(n)} = (2^{\log(n)})^{1/\log(n)}$
- Ensuite le  $\log(\log(n))$
- Puis  $n^{\sqrt{n}} = 2^{\sqrt{n} \cdot \log_2(n)}$
- Puis  $2^n$
- Enfin  $3^n/n^5$

### Question 6: (🕒 20 minutes) Théorie

1. Donner en pseudo code un algorithme ayant une complexité temporelle de  $O(\log n)$  qui prend comme argument un tableau trié  $A[1, \dots, n]$  de  $n$  nombres et une clé  $k$  et qui retourne "OUI" si  $A$  contient  $k$  et "NON" sinon.
2. Quelle est la hauteur maximum et la hauteur minimale d'un arbre binaire de recherche ayant  $n$  éléments ? Quel arbre est meilleur ? Justifier.
3. Considerer l'arbre binaire suivant :  
Dessiner les arbres obtenus après exécutions de chacune des opérations suivantes (chaque opération



est exécutée en commençant par l'arbre ci-dessus - les opérations ne sont pas exécutées séquentiellement).

- (a) TREE-INSERT(T, z) avec z.key = 0
- (b) TREE-INSERT(T, z) avec z.key = 17
- (c) TREE-INSERT(T, z) avec z.key = 14
- (d) TREE-DELETE(T, a)
- (e) TREE-DELETE(T, b)
- (f) TREE-DELETE(T, c)

## >\_ Solution

1. Etant donné que les nombres du tableau A sont triés, nous utilisons l'algorithme de recherche binaire. L'algorithme de recherche binaire prend comme argument un tableau A, une clé k, des indices p et q et retourne "OUI" si A[p . . . q] contient la clé k et "NON" autrement. Comme A[p . . . q] est trié, nous pouvons comparer k avec l'élément du milieu  $mid = \lfloor (p + q) / 2 \rfloor$  et :

- Si A[mid] = k return "OUI"
- Si A[mid] > k, alors cherchons k dans le tableau A[p . . . (mid-1)] en appelant récursivement l'algorithme BINARY-SEARCH(A, k, p, mid-1)
- Si A[mid] < k, alors cherchons k dans le tableau A[(mid + 1) . . . q] en appelant récursivement l'algorithme BINARY-SEARCH(A, k, mid+1, q)

The pseudo-code of the procedure is as follows :

```

1 def binary_search(A, k, p, q):
2   if (q < p):
3     return "NON" # array is empty so it doesn't contain k
4   else:
5     mid = (p+q)//2
6     if (A[mid] == k):
7       return "OUI"
8     elif (A[mid] > k):
9       return binary_search(A, k, p, mid-1)
10    else: # A[mid] < k
11      return binary_search(A, k, mid+1, q)

```

Note that we solve the original problem by calling BINARY-SEARCH(A, k, 1, n).

2. Hauteur minimale et maximale d'un arbre binaire.
  - La hauteur maximum d'un arbre binaire est atteinte quand l'arbre n'est constitué d'une seule branche.
  - La hauteur minimum est atteinte lorsque l'arbre binaire est "complet" : nous ne pouvons pas ajouter un noeud sans augmenter la hauteur de l'arbre hauteur de un.
3. Après execution de chaque opération, nous obtenons :
  - (a) Figure A : TREE-INSERT(T, z) avec z.key = 0
  - (b) Figure B : TREE-INSERT(T, z) avec z.key = 17
  - (c) Figure C : TREE-INSERT(T, z) avec z.key = 14
  - (d) Figure D : TREE-DELETE(T, a)
  - (e) Figure E : TREE-DELETE(T, b)
  - (f) Figure F : TREE-DELETE(T, c)