TP 1 – Vitesse d'un algorithme

L'objectif de cette activité est de comparer les vitesses d'exécution d'algorithmes différents réalisant une même tâche.

1 Recherche dans un tableau trié

Soit une liste L de nombres triés dans l'ordre croissant et un nombre x.
On veut déterminer si le nombre x est présent ou non dans la liste L.
Pour cela on réalise une fonction qui prend en variables la liste L et le nombre x, et qui renvoie une variable booléenne : VRAI *présent* ou FAUX *non présent*.

1. **Recherche séquentielle**
   Écrire et tester un algorithme qui réalise cette tâche en utilisant une boucle séquentielle.

```python
# Version optimale, utilise la liste directement
def Présence1(L,x):
    for valeur in L:
        if valeur==x:
            return True
    return False
# Version qui utilise le rang des éléments de la liste.
# Pas optimale, mais permet de se rafraichir la mémoire sur
# l'utilisation des listes
def Présence2(L,x):
    for rang in range(len(L)): # rang va de 0 à len(L)-1
        if L[rang]==x:
            return True
    return False
```

2. **Recherche dichotomique**
   Écrire et tester un algorithme qui réalise les opérations ci-dessous.
   a. définir les variables `début` et `fin` et les initialiser au rang du début et de la fin de la liste `L`
   b. lancer une boucle conditionnelle qui se termine quand `début` égale `fin`
   c. définir une variable `rang` qui prend la valeur de la moyenne entre `début` et `fin` arrondi à l'entier inférieur.
   d. définir la variable `valeur` qui prend la valeur de l'élément de la liste `L` de rang `rang`.
   e. Si la valeur `x` recherchée est égale à la variable `valeur`, retourner `True`
   f. Si la valeur `x` recherchée est inférieure à la variable `valeur`, adapter l'intervalle [`début`, `fin`] de sorte que la boucle conditionnelle s'applique à la partie de la liste qui peut contenir `x`.

g. Si au final la valeur `x` n'est pas dans la liste `L`, retourner `False`.

```
def Dichotomie(L,x):
    début=0
    fin=len(L)-1
    while début != fin:
        rang=(début+fin)//2
        valeur=L[rang]
        if valeur==x:
            return True
        elif x<valeur:
            fin=rang-1
        else:
            début=rang+1
        input()
    if L[rang+1]==x or L[rang-1]==x:
        return True
    return False
```

3. Comparaison du nombre d'opérations

Comparer le nombre d'opérations réalisées dans chacune des deux recherches en fonction du nombre n d'éléments contenus dans la liste.

Recherche séquentielle.
Le nombre de tests à réaliser est compris entre 1 (le nombre x est le premier de la liste) et n (le nombre x est le dernier de la liste, ou n'y est pas présent).
Si on considère que x est présent dans la liste et que chaque place pour x est équiprobable, le nombre moyen de tests est : $\frac{n}{2}$.
On dit que le nombre d'opérations à réaliser croît **linéairement** en fonction du nombre d'éléments de la liste.

Recherche dichotomique.
A chaque itération, le nombre d'éléments à tester est divisé par deux. Le nombre maximum d'itérations est donc le plus petit entier k tel que :

$$2^k \geqslant n.$$

En appliquant la fonction ln à chaque membre de l'inégalité on obtient :

$$k \geqslant \frac{\ln(n)}{\ln(10)}.$$

On dit que le nombre d'opérations à réaliser croît **logarithmiquement** en fonction du nombre d'éléments de la liste.

2 Exponentiation

Soit un réel a et un entier naturel n.
On cherche à réaliser une fonction pour calculer a^n. Il s'agit de programmer la fonction `a**n`.

1. **Calcul à l'aide d'une boucle séquentielle**
   Écrire et tester un algorithme qui réalise cette tâche en utilisant une boucle séquentielle.

```python
def Puissance(a,n):
    produit=1
    for k in range(n):
        produit=produit*a
    return produit
```

2. **Calcul par carrés**
   Écrire et tester un algorithme qui réalise les opérations ci-dessous.
   a. initialiser la variable `nombre` à `a`, la variable `exposant` à n, et la variable `produit` à 1.
   b. lancer une boucle conditionnelle qui se termine quand le cas où `exposant` vaut 1 a été traité
   c. si n est pair, le diviser par 2, et affecter la variable `nombre` à son carré
   d. traiter judicieusement le cas où n est impair.
   e. retourner le résultat a^n.

```python
def exponentiation_rapide(a, n):
    nombre = a
    exposant=n
    produit = 1
    while exposant>0:
        if exposant % 2: # si exposant est impair
            produit *= nombre # produit = produit * nombre
        nombre *= nombre # nombre = nombre * nombre
        exposant //= 2 # exposant = exposant // 2
    return produit
```

3. **Comparaison du nombre d'opérations**
   Comparer le nombre d'opérations réalisées dans chacun des deux calculs en fonction de la valeur de l'exposant n.

   La boucle séquentielle réalisera $n-1$ produits, soit une croissante **linéaire** du nombre de produits en fonction de la puissance à calculer.

   Le calcul par carrés.
   Si l'exposant n est une puissance de 2 alors on a :

$$a^n = a^{(2^p)} = \left(\left(\left(a^2\right)^2\right)\dots\right)$$

   Il faudra réaliser seulement $p = \dfrac{\ln(n)}{\ln(2)} = \log_2(n)$ multiplications.
   Si l'exposant n est un entier qui précède une puissance de 2, il faudra réaliser à peu près

deux fois plus de multiplications, soit

$$2\log_2(n).$$

Cet encadrement permet d'affirmer que la croissance du nombre de multiplications en fonction de la puissance à calculer est **logarithmique**.

$\sim$