# Exercice 1. Exponentiation rapide

**Question 1.a** – On adopte la stratégie suivante :

$p \leftarrow 1$
**pour** $i = 1$ à $n$ **faire**
  $p \leftarrow p \times x$
**fin pour**
**renvoyer** p

**Question 1.b** –

```
def puiss(x, n):
    """
    puiss(x: float, n: int) -> float
    Hypothese: n >= 0
    """
    p = 1
    for _ in range(n):
        p = p*x
    return p
```

**Question 1.c** – Il y a $n$ tours de boucle et chaque tour s'exécute en temps constant, donc :

La complexité est linéaire en $n$.

**Question 2.a** – On adopte la stratégie suivante :

$p \leftarrow 1$, $y \leftarrow x$, $m \leftarrow n$.
**tant que** $m \neq 0$ **faire**
  **si** $m$ est impair **alors**
    $p \leftarrow p \times y$
  **fin si**
  $y \leftarrow y \times y$
  $m \leftarrow \left\lfloor \frac{m}{2} \right\rfloor$
**fin tant que**
**renvoyer** p

**Question 2.b** –

```
def puiss_dicho(x, n):
    """
    puiss_dicho(x: float, n: int) -> float
    Hypothese: n >= 0
    """
    p = 1; y = x; m = n
    while m != 0:
        if m % 2 == 1:
            p = p*y
        y = y*y
        m = m//2
    return p
```

**Question 2.c** – On a une complexité logarithmique en $n$

Pour le montrer, il suffit de compter le nombre de tours de boucle. On procède comme pour la recherche dichotomique dans une liste triée (voir cours). On numérote les tours de boucle par 1, 2, ..., $K$ où $K \in \mathbb{N}$ est le nombre de tours. Par une récurrence immédiate, au début du tour numéro $k \in \{1, 2, \ldots, K\}$, la valeur de la variable `m` notée $m_k$ vérifie :

$$m_k \leqslant \frac{n}{2^{k-1}}.$$

Pour $k_0 = \left\lceil \log_2(n) \right\rceil + 2$, on a $m_{k_0} \leqslant 1/2$. Donc il y a au plus $k_0 - 1$ tours de boucle.

**Question 3.a** – On doit choisir un $n$ relativement grand pour voir une différence entre les deux implémentations, mais il faut aussi que $|x^n|$ soit petit. En effet, si $|x^n|$ est grand (par exemple si on prend $x = 2$) alors le temps d'exécution sera altéré par le fait que la multiplication de deux grands entiers prend du temps.

On choisit de calculer $x^n$ où $x$ est un flottant aléatoire de $\left[\, 1 \,;\, 1 + 1/n \,\right]$. Notons que les quantités mises en jeu restent petites :

$$|x^n| \leqslant \left(1 + \frac{1}{n}\right)^n < e$$

```
import time, random
TPS_TEST = 1 # Chaque test dure au moins 1 seconde

def test_puiss(n, dicho): # dicho = True ou False
    tps_ini = time.perf_counter()
    nb_tests = 0
    while time.perf_counter() < tps_ini + TPS_TEST:
        nb_tests += 1
        x = random.uniform(1,1+1/n)
        if dicho:
            _ = puiss_dicho(x,n)
        else:
            _ = puiss(x,n)
    tps_moyen  = (time.perf_counter() - tps_ini)/nb_tests
    return tps_moyen, nb_tests
```

```
print("-"*30)
print("Fonction puiss")
print("valeur de n (temps moyen d'un test, nombre de tests effectués)")
for n in [10**i for i in range(2,9)]:
    print(n, test_puiss(n, False))

print("-"*30)
print("Fonction puiss_dicho")
print("valeur de n (temps moyen d'un test, nombre de tests effectués)")
for n in [10**i for i in range(2,9)]:
    print(n, test_puiss(n, True))
```

**Question 3.b** – On remarque que :

→ Lorsque $n$ est mutiplié par 10, le temps d'exécution de `puiss` est (approximativement) multiplié par 10. Ce qui correspond bien à une complexité linéaire.

→ Lorsque $n$ est mutiplié par 10, une quantité (approximativement) constante est ajoutée au temps d'exécution de `puiss_dicho`. Ce qui correspond bien à une complexité logarithmique.

## Exercice 2. Concaténations de chaînes de caractères

**Question 1.a** – Au début du tour de boucle numéro `k = 0`, on a `C[-1] = ""`. Pour `k > 0` :

$$\texttt{C[}-1\texttt{]} = \texttt{L[0]} + \texttt{L[1]} + \ldots + \texttt{L[k} - 1\texttt{]}.$$

**Question 1.b** – Dans la fonction, il y a `n = len(L)` tours de boucle. Au tour de boucle numéro `k`, on concatène une chaîne de taille `k` avec une chaîne de taille 1. Comme $\texttt{k} \leqslant \texttt{n}$, le temps d'exécution du tour de boucle numéro `k` est en $\mathcal{O}(\texttt{n})$.

La complexité finale est bien quadratique en `n`

**Question 2.a** – L'algorithme est le suivant :

**si** `L` est vide **alors**
  **renvoyer** `""`
**fin si**
**tant que** `L` n'est pas de taille 1 **faire**
  Soit `M` une liste vide
  **pour** $i = 0$ à `len(L)//2-1` **faire**
    Ajouter `L[2*i] + L[2*i+1]` à la fin de `M`
  **fin pour**
  **si** `len(L)` est impaire **alors**
    Ajouter `L[-1]` à la fin de `M`
  **fin si**
  Remplacer `L` par `M`
**fin tant que**
**renvoyer** `L[0]`

**Question 2.b** –

```
def concat_dicho(L):
    if len(L) == 0:
        return ""
    while len(L) > 1:
        M = []
        for i in range(len(L)//2):
            M.append(L[2*i] + L[2*i+1])
        if len(L) % 2 == 1:
            M.append(L[-1])
        L = M
    return L[0]
```

**Question 3.a** – Montrons que lorsque la liste `L` initiale est de taille $2^k$, le nombre de tours de boucle est $k$. Soit $N$ le nombre de tours de boucle et $n$ la taille initiale de la liste `L`. On numérote les tours de 0 à $N-1$. Au début du $i^{\text{ème}}$ tour, la taille de la liste est $n/2^i$ (se montre facilement par récurrence sur $i$). Ainsi, au début du tour $k-1$, la liste est de taille :

$$\frac{n}{2^{k-1}} = \frac{2^k}{2^{k-1}} = 2$$

Donc, à la fin du tour $k-1$, la liste est de taille 1.

Le nombre de tours de boucle est donc égal à $k$

**Question 3.b** – Soit `L` la liste donnée en argument de la fonction, soit `M` la liste créée lors d'un tour de la boucle `while`. Pour exécuter ce tour, on a besoin de créer les chaînes de caractères `M[0]`, `M[1]`, ..., `M[len(M)-1]`. Le temps d'exécution est donc proportionnel à :

$$\sum_{i=0}^{\texttt{len(M)}-1} \texttt{len(M[i])}$$

Or, on sait que les deux chaînes de caractères suivantes sont égales :

$$\texttt{M[0]} + \texttt{M[1]} + \ldots + \texttt{M[len(M)} - 1\texttt{]} \qquad \texttt{L[0]} + \texttt{L[1]} + \ldots \texttt{L[len(L)} - 1\texttt{]}$$

Donc :

$$\begin{aligned}
\sum_{i=0}^{\texttt{len(M)}-1} \texttt{len(M[i])} &= \texttt{len}\Big(\texttt{M[0]} + \texttt{M[1]} + \ldots + \texttt{M[len(M)} - 1\texttt{]}\Big) \\
&= \texttt{len}\Big(\texttt{L[0]} + \texttt{L[1]} + \ldots + \texttt{L[len(L)} - 1\texttt{]}\Big) \\
&= \sum_{i=0}^{\texttt{len(L)}-1} \texttt{len(L[i])} \\
&= \sum_{i=0}^{\texttt{len(L)}-1} 1 \\
&= \texttt{len(L)}
\end{aligned}$$

Finalement, le temps d'exécution d'un tour de la boucle `while` est linéaire en `len(L)`

**Question 3.c** – En combinant les résultats des deux questions précédentes :

> Le temps d'exécution d'un appel à `concat_dicho(L)` est en $\mathcal{O}(n\log(n))$ où $n$ est la taille de `L`

**Question 4** –

```
import time, random
TPS_TEST = 1 # Chaque test dure au moins 1 seconde

def test_concat(n, dicho): # dicho = True ou False
    tps_ini = time.perf_counter()
    nb_tests = 0
    i_min = ord("a"); i_max = ord("z")
    while time.perf_counter() < tps_ini + TPS_TEST:
        nb_tests += 1
        L = [chr(random.randint(i_min, i_max)) for _ in range(n)]
        if dicho:
            _ = concat_dicho(L)
        else:
            _ = concat(L)
    tps_moyen  = (time.perf_counter() - tps_ini)/nb_tests
    return tps_moyen, nb_tests
```

```
print("-"*30)
print("Fonction concat")
print("taille liste (temps moyen d'un test, nombre de tests effectués)")
for n in [10**i for i in range(2,6)]:
    print(n, test_concat(n, False))

print("-"*30)
print("Fonction concat_dicho")
for n in [10**i for i in range(2,6)]:
    print(test_concat(n, True))
```

## Exercice 3. Nombre d'occurrences dans une liste triée

**Question 1 –**

```
def indice_geq_min(L, x, i_min, i_max):
    """
    indice_geq_min(L: list[int], x: int, i_min: int, i_max: int) -> int
    -----------------
    geq = greater or equal.
    Renvoie le plus petit indice i compris entre i_min et i_max tel que L[i] >=
    x. En particulier, si x apparait dans L, la fonction renvoie le plus petit
    indice tel que L[i] == x. Si tous les elements de L sont < x alors renvoie
    i_max+1.
    """
    i = i_min
    j = i_max
    while i <= j:
        m = (i+j)//2
        if L[m] >= x:
            j = m-1
        else:
            i = m+1
    return i
```

```
def indice_leq_max(L, x, i_min, i_max):
    """
    indice_leq_max(L: list[int], x: int, i_min: int, i_max: int) -> int
    -----------------
    leq = less or equal.
    Renvoie le plus grand indice i compris entre i_min et i_max tel que L[i] <=
    x. En particulier, si x apparait dans L, la fonction renvoie le plus grand
    indice tel que L[i] == x. Si tous les elements de L sont > x alors renvoie
    i_min-1.
    """
    i = i_min
    j = i_max
    while i <= j:
        m = (i+j)//2
        if L[m] > x: # Différence avec la fonction précédente
            j = m-1
        else:
            i = m+1
    return j         # Différence avec la fonction précédente
```

```
def nb_occ(L, x):
    """nb_occ(L: list[int], x: int) -> int"""
    i0 = indice_geq_min(L, x, 0, len(L)-1)
    i1 = indice_leq_max(L, x, 0, len(L)-1)
    return i1 - i0 + 1
```

**Question 2** – Soient `i0` et `i1` le plus petit indice et le plus grand indice tels que `L[i] == L[i0]` et `L[i] == L[i1]`. On va dans un premier temps calculer un encadrement de `i0` et de `i1`, puis calculer le nombre d'occurrences en utilisant les arguments `i_min` et `i_max` des fonctions `indice_geq_min` et `indice_leq_max`.

Pour trouver l'encadrement sur `i0`, on regarde la valeur de $L[i - 2^k]$ en augmentant la valeur de `k`. Lorsqu'on sort de la liste, ou bien que $L[i - 2^k] != L[i]$, on obtient notre encadrement.

```
def encadrement_i0(L, i):
    """
    encadrement_i0(L: list[int], i: int) -> int
    Renvoie un encadrement du plus petit indice i0 tel que L[i0] == L[i].
    Hypothese: i est un indice valide pour L
    """
    k = 0
    while True:
        i_min = i - 2**k
        if i_min < 0 or L[i_min] != L[i]:
            break
        k += 1
    if i_min < 0:
        i_min = 0
    else:
        i_min += 1
    if k == 0:
        return i_min, i
    else:
        return i_min, i - 2**(k-1)
```

```
def encadrement_i1(L, i):
    """
    encadrement_i1(L: list[int], i: int) -> int
    Renvoie un encadrement du plus grand indice i1 tel que L[i1] == L[i].
    Hypothese: i est un indice valide pour L
    """
    k = 0
    while True:
        i_max = i + 2**k
        if i_max >= len(L) or L[i_max] != L[i]:
            break
        k += 1
    if i_max >= len(L):
        i_max = len(L)-1
    else:
        i_max -= 1
    if k == 0:
        return i, i_max
    else:
        return i + 2**(k-1), i_max
```

```
def nb_occ_bis(L, i):
    """nb_occ_bis(L: list[int], i: int)"""
    i_min_i0, i_max_i0 = encadrement_i0(L, i)
    i_min_i1, i_max_i1 = encadrement_i1(L, i)
    i1 = indice(L, L[i], i_min_i0, i_max_i0, True)
    i2 = indice(L, L[i], i_min_i1, i_max_i1, False)
    return i2 - i1 + 1
```

## Exercice 4. Trichotomie (épreuve 2019 du concours e3A)

**Question 1** –

```
def appartient_tricho(L, x):
    """appartient_tricho(L: list[int], x: int) -> bool"""
    i = 0
    j = len(L)-1
    while i <= j:
        m1 = (2*i+j)//3
        m2 = (i+2*j)//3
        if L[m1] == x or L[m2] == x:
            return True
        if x < L[m1]:
            j = m1-1
        elif x < L[m2]:
            i = m1+1
            j = m2-1
        else:
            i = m2+1
    return False
```