Exponentiation rapide

Attention

Dans tout ce TP, on va calculer a^n sans utiliser `a**n`.

Exercice 1 ($\star$)**.** Écrire une fonction appelée `ExpoN(a:float,n:int)->float` qui a un nombre `a` et un entier naturel `n` renvoie a^n. Pour cela, utiliser la formule naïve : $a^n = a \times a \times \ldots \times a$. Déterminer la complexité de cet algorithme : on note $C(n)$ le nombre de multiplications. Donner un invariant de boucle utilisée dans cette fonction. Tester votre programme avec $3^{10} = 59\,049$.

Exercice 2 ($\text{𝄞}\star$)**.** Écrire une fonction **récursive** appelée `ExpoR(a:float,n:int)->float` qui renvoie a^n. Pour cela, utiliser avantageusement que si $n = 2q$ est pair, alors $a^n = (a^q) \times (a^q)$, et que si $n = 2q+1$ est impair, alors $a^n = (a^q) \times (a^q) \times a$. On note $C(n)$ le nombre de multiplications nécessaires, en fonction de si n est pair ou impair, exprimer $C(n)$ en fonction de $C(k)$ où k est un certain entier vérifiant $k < n$. Que vaut $C(n)$ si $n = 2^p$.

Exercice 3 ($\text{𝄞}\star\star$)**.** La fonction `ExpoR` est une version bien plus optimale que `ExpoN`. On se propose maintenant d'écrire `ExpoNR(a,n)` une version non récursive de `ExpoR`. Utilisons une boucle `while`, les variables `x`, `k` et `p` et l'invariant de boucle suivant :

$$x^k \times p = a^n$$

La variable `p` est initialisé à 1. Quelles doivent être les valeurs initiales de `x` et `k` pour que l'invariant soit initialement vrai. À chaque itération, on souhaite remplacer `k` par `k//2`. En séparant les cas suivant la parité de `k`, comment faire évoluer `x` et `p` pour que l'invariant reste vrai ? Quand `k` vaudra `0`, quelle variable aura la valeur recherchée ? Écrire alors la fonction `ExpoR`. Quel variant de boucle utiliser pour démontrer la terminaison de l'algorithme ?

Remarque 1. La méthode utiliser pour calculer a^n à l'exercice 2 ou à l'exercice 3 s'appelle l'**exponentiation rapide**. Ces deux fonctions ont un fonctionnement très similaire. L'exercice 3 étant la version non récursive de l'exercice 2. Cela repose dans tous les cas sur de la dichotomie. En effet, à chaque étape on divise [1] n par 2. Cette méthode s'applique aussi pour le calcul des puissances des matrices carrées.

Exercice 4 ($\star$)**.** On souhaite faire ce que l'on appelle un jeu de tests. C'est-à-dire, un script qui va essayer de trouver un bug dans nos fonctions. Pour cela, on peut, par exemple, 2^i à l'aide de ces trois fonctions pour i entre 0 et 100, et voir si le résultat diffère suivant les fonctions.

1. Plus ou moins suivant la parité de n en fait.

Exercice 5 ($\star$)**.** On souhaite mesurer le temps nécessaire par chacun des programmes pour exécuter ces calculs de puissance. Pour cela, importer une bibliothèque gérant le temps : `import time as time`. Pour mesurer le temps de un calcul, on fait :

```
tic=time.time()
#Un calcul
tac=time.time()
delai=tac-tic#Temps mis pour faire le calcul.
```

Quelle fonction est la plus rapide pour calculer 3^{10} et pour calculer 3^{10^6}

Remarque 2. L'exponentiation rapide ou la recherche dichotomique dans une liste triée ont pour complexité un $\mathcal{O}(\ln(n))$ où n est la longueur de la liste.

Et la suite

Exercice 6. Reprendre l'exercice 5 pour afficher, sur un graphe, en ordonnée le temps mis pour calculer 3^N en fonction de N en abscisse. On fera quatre courbes, une pour chaque fonction et la quatrième pour la commande Python `3**N`.

Retourner terminer le TP précédent (terminaison totale of course).