TP1

October 23, 2015

1 Complexité

On peut utiliser le notebook pour mesurer le temps d'exécution de programmes, on va s'en servir pour vérfier en direct les complexités théoriques que nous avons vues en cours.

Pour mesurer le temps d'exécution, on utilise la commande *time*.

```
In [ ]: def ajout(n):
            t = []
            for i in xrange(1000000):
                t.append(i)
```

```
In [ ]: time ajout(1000000000)
```

1.1 Un peu d'échauffement

sur le modèle de la fonction ci-dessous qui simule une complexité $O(n)$, écrivez des fonctions qui simulent les complexité n^2, n^3, $\sqrt(n)$, et $log(n)$. Vous n'avez besoin que de la multiplication et de l'addition.

```
In [ ]: def complexiteN(n):
            i = 0
            while i < n:
                i+=1
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

Utiliser la commande *time* pour comparer les temps d'exécution pour n valant 100, 1000, 10 000, 100 000, 1 000 000.

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

1.2 Recherche dichotomique

Compléter les deux fonctions suivantes qui effectue une recherche dans un tableau. Dans le second cas, on on suppose que le tableau est trié et on effectue donc une recherche dichotomique.

```
In [ ]: """
        Recherche classique dans un tableau
        Input :
            - l, une liste d'entiers
            - v, une valeur
        Output : True si v est dans le tableau, False sinon
        """
        def recherche(l, v):
            # écrire le code
```

```
In [ ]: assert(recherche([5,2,4,2,6,4,6], 4))
        assert(recherche([5,2,4,2,6,4,6], 5))
        assert(recherche([5,2,4,2,6,4,6], 2))
        assert(recherche([5,2,4,2,6,4,6], 6))
        assert(not recherche([5,2,4,2,6,4,6], 10))
```

```
In [ ]: """
        Recherche dichotomique dans un tableau trié
        Input :
            - l, une liste d'entiers
            - v, une valeur
        Output : True si v est dans le tableau, False sinon
        """
        def rechercheDicho(l, v):
            # écrire le code
```

```
In [ ]: assert(rechercheDicho([2,2,4,4,5,6], 4))
        assert(rechercheDicho([2,2,4,4,5,6], 5))
        assert(rechercheDicho([2,2,4,4,5,6], 2))
        assert(rechercheDicho([2,2,4,4,5,6], 6))
        assert(not rechercheDicho([2,2,4,4,5,6], 10))
```

La fonction suivante retourne un tableau d'entiers alétoire de taille donnée (non trié).

```
In [ ]: def random_tableau(n):
            import random
            return [random.randint(1,n) for i in xrange(n)]
```

Comme nous n'avons pas encore étudié les algorithmes de tris, nous allons utiliser la méthode *sort* de python comme ci dessous.

```
In [ ]: t = random_tableau(1000)
        t.sort()
```

Ecrire une fonction qui prend en paramètre un entier n et utilise *random_tableau* et *sort* pour retourner un tableau trié.

```
In [ ]:
```

Dans les cellules suivantes, faites dans l'ordre pour chacune des tailles 100, 1000, 10 000, 1 000 000 * Tirez un tableau trié de taille n * mesurez le temps de recherche classique pour une valeur quelconque * mesurez le temps de recherche dichotomique pour la même valeur (dans le même tableau !)

Vous pouvez recommencer plusieurs fois pour chacune des valeurs et observer les résultats

```
In [ ]: # tirez le tableau trié

In [ ]: # mesurez le temps d'une recherche classique

In [ ]: # mesurez le temps d'une recherche dichotomique
```

Comme python possède la fonction *sort*, on se dit que peut-être, on pourrait modifier la recherche dans les tableaux non triés pour d'abord trié le tableau puis effectuer une recherche dichotomique.

Complétez la fonction suivante

```
In [ ]: """
        Trie le tableau en entrée puis effectue une recherche dichotomique de la valeur v
        Input :
            - l, une liste d'entiers
            - v, une valeur
        Output : True si v est dans le tableau, False sinon
        """
        def rechercheDichoPresque(l, v):
            # écrire le code
```

```
In [ ]: assert(rechercheDichoPresque([5,2,4,2,6,4,6], 4))
        assert(rechercheDichoPresque([5,2,4,2,6,4,6], 5))
        assert(rechercheDichoPresque([5,2,4,2,6,4,6], 2))
        assert(rechercheDichoPresque([5,2,4,2,6,4,6], 6))
        assert(not rechercheDichoPresque([5,2,4,2,6,4,6], 10))
```

Comparez les performances de *recherche* et *rechercheDichoPresque*

```
In [ ]:

In [ ]:

In [ ]:

In [ ]:
```

Python a lui aussi une fonction de recherche implantée nativement

```
In [ ]: l = [1,2,3]
        3 in l
```

Comparez sur tableau trié / non trié les performances de vos différentes fonctions et de la fonction native de python.

```
In [ ]:
```