Python – Dichotomie

Recherche dichotomique

Pour cette feuille, nous allons avoir besoin des modules suivants :

```
import random
import time
import matplotlib.pyplot as plt
```

Exercice 1 : Compléter la fonction `recherche_naive(val, tab)` qui cherche `val` dans `tab` en regardant élément par élément. La fonction renoie soit l'index auquel se trouve `val` s'il est dans le tableau et sinon **None**.

```
def recherche_naive(val, tab):
    for i in range(len(tab)):
        if ...:
            return ...
    return ...
```

```
>>> recherche_naive(4, [1, 4, 7, 9, 90])
1
>>> recherche_naive(90, [1, 4, 7, 9, 90])
4
>>> recherche_naive(6, [1, 4, 7, 9, 90])  # None ne s'affiche pas
>>>
```

Exercice 2 : Compléter la fonction `recherche_dichotomique(val, tab)` qui fait la recherche en utilisant la dichotomie. Le tableau `tab` doit être trié dans l'ordre croissant. La fonction renvoie un indice où se trouve `val` si elle est dans `tab` et **None** sinon.

```
def recherche_dichotomique(val, tab):
    g = ... # indice du premier élément
    d = ... # indice du dernier élément
    while ...: # il reste des positions à tester
        m = ... # On prend l'indice au milieu de g et de d
        if ...: # la valeur se trouve a gauche
            d = ...
        elif ...: # la valeur se trouve apres
            g = ...
        else: # tab[m] == val
            return ...
    return ... # On n'a pas trouvé
```

```
>>> recherche_dichotomique(4, [1, 4, 7, 9, 90])
1
>>> recherche_dichotomique(90, [1, 4, 7, 9, 90])
4
>>> recherche_dichotomique(6, [1, 4, 7, 9, 90])
>>>
```

Exercice 3 : Compléter la fonction `genere_tableau(n, saut_max)` qui génère un tableau trié dont les valeurs sont tirées au hasard. L'écart entre deux valeurs successibles est entre 0 et `saut_max`. La première valeur est également entre 0 et `saut_max`. On rappelle que la fonction `random.randint(a, b)` renvoie un entier tiré au hasard entre `a` et `b` inclus.

```
def genere_tableau(n=1000, saut_max=5):
    tab = []
    val = 0
    for i in range(...):
        val = ...
        tab.append(...)
    return tab
```

```
>>> genere_tableau(10, 3)
[3, 3, 5, 5, 8, 8, 10, 12, 14, 17]
>>> genere_tableau(10, 3)
[2, 2, 2, 4, 7, 10, 10, 11, 13, 15]
>>> genere_tableau(10, 15)
[4, 5, 16, 18, 29, 38, 44, 49, 54, 62]
```

Comparaison de l'efficacité

La fonction suivante permet de mesurer le temps de recherche moyen d'un élément dans `tab`, en répétant `k` recherches avec l'algoritme `recherche`. Si `pire_cas` est vrai, on cherche une valeur plus grande que la dernière valeur du tableau. Sinon on prend une valeur au hasard entre `tab[0]-1` et `tab[0]+1`.

```
def mesure_temps_recherche(tab, recherche, k, pire_cas=False):
    debut = tab[0] - 1
    fin = tab[-1] + 1
    temps_total = 0
    for _ in range(k):
        if pire_cas:
            val = fin
        else:
            val = random.randint(debut, fin)
        temps_debut = time.time()
        recherche(val, tab)
        temps_total += time.time() - temps_debut
    return temps_total / k
```

On utilise cette fonction pour comparer les temps de recherches moyen sur un tableau de taille donné avec les deux algorithmes :

```
def comparaison_temps(taille, k=100, pire_cas=False, saut_max=5):
    tab = genere_tableau(taille, saut_max)
    temps_naif = mesure_temps_recherche(tab, recherche_naive, k, pire_cas)
    temps_dichotomie = mesure_temps_recherche(tab,\
                                        recherche_dichotomique, k, pire_cas)
    print(f"Pour chercher dans un tableau de taille {taille} :")
    print(f"Recherche naive : {temps_naif}")
    print(f"Recherche dicho : {temps_dichotomie}")
```

Exercice 4 : Comparer les temps moyens pour des tableaux de longueur 100, 10 000 et 1 000 000, que ce soit dans le pire des cas ou pas. Vous pouvez éventuellement diminuer la valeur de `k`.

La fonction suivante permet d'afficher un graphique montrant l'évolution du temps de calcul pour les deux algorithmes.

```
def graphique_temps(n_max, k, points=10, pire_cas=False, saut_max=5):
    t_taille = [] # pour les abscisses
    # Tableaux des ordonnées
    t_naif = []
    t_dicho = []
    pas = n_max//points # distance entre les abscisses des points
    n = pas
    for _ in range(points):
        t_taille.append(n)
        # temps de calcul pour un tableau de taille n
        tab = genere_tableau(n, saut_max)
        temps_naif = mesure_temps_recherche(tab,\
                            recherche_naive, k, pire_cas)
        temps_dichotomie = mesure_temps_recherche(tab,\
                          recherche_dichotomique, k, pire_cas)
        # On rajoute les temps moyens aux listes des ordonnées
        t_naif.append(temps_naif)
        t_dicho.append(temps_dichotomie)
        n += pas
    # affichage des courbes
    plt.plot(t_taille, t_naif, label="naïf")
    plt.plot(t_taille, t_dicho, label="dichotomie")
    plt.legend()
    plt.show()
```

La taille maximale des tableaux est donné par `n_max`, le nombre de répétition pour calculer la moyenne de temps par `k` et `points` indique le nombre de valeurs de `n` qui seront utilisées pour tracer la courbe. Plus on augmente les paramètres, plus le temps de calcul augmente.

Exercice 5 : Tracer les courbes pour différentes valeurs en comparant les résultats en mettant `pire_cas` à vrai ou à faux.

Deviner le nombre

Dans cette partie, nous allons programmer le jeu où un jouer fait deviner un nombre à l'autre. D'abord avec le joueur qui devine le nombre de l'ordinateur, puis le contraire.

Exercice 6 : Compléter la fonction `deviner_nombre(mini=1, maxi=63)` dans lequel l'ordinateur tire un nombre au hasard entre `mini` et `maxi`. Le joueur donne alors des nombres et l'ordinateur lui dit si le secret est plus grand ou plus petit, jusqu'à ce qu'il trouve. À la fin, on lui indique le nombre d'essais qu'il a fait.

```
def deviner_nombre(mini=1, maxi=63):
    nb = -1
    secret = random.randint(mini, maxi)
    tentatives = 0
    while ...:
        nb = int(input(f"Donne un nombre entre {mini} et {maxi} : "))
        if ...:
            print("C'est plus")
        elif ...:
            print("C'est moins")
        else:
            print("Bravo")
        tentatives += 1
    print(f"Tu as trouvé en {tentatives} coups")
```

```
>>> deviner_nombre(1, 63)
Donne un nombre entre 1 et 63 : 32
C'est moins
Donne un nombre entre 1 et 63 : 16
Bravo
Tu as trouvé en 2 coups
```

Exercice 7 : Écrire une fonction `deviner_nombre_aide(mini=1, maxi=63, secret=-1)` qui se comporte comme la fonction précédente, sauf qu'à chaque étape, l'ordinateur indique le meilleur nombre à proposer. Si `secret` est égal à -1, une nouvelle valeur est choisie au hasard, sinon, c'est la valeur donnée qui est cherchée. Cela permet de tester plus facilement la fonction.

```
>>> deviner_nombre_aide(1, 20)
Donne un nombre entre 1 et 20 (10 serait pas mal) : 10
C'est plus
Donne un nombre entre 11 et 20 (15 serait pas mal) : 15
C'est moins
Donne un nombre entre 11 et 14 (12 serait pas mal) : 12
C'est moins
Donne un nombre entre 11 et 11 (11 serait pas mal) : 11
Bravo
Tu as trouvé en 4 coups
>>> deviner_nombre_aide(1, 1000, 397)
Donne un nombre entre 1 et 1000 (500 serait pas mal) : 397
Bravo
Tu as trouvé en 1 coups
```

Exercice 8 : Compléter la fonction `ordinateur(g=1, d=63)` qui invite l'utilisateur à penser à un nombre entre `g` et `d` et lui fait des propositions. L'utilisateur peut répondre avec `"+"`, `"-"` et `"="`. À la fin, l'ordinateur indique le nombre d'essais. Attention, il faut s'assurer que l'utilisateur donne des réponses cohérentes. Par exemple, le nombre cherché ne peut pas être plus petit que le minimum.

```
def ordinateur(g=1, d=63):
    print(f"Pensez à un nombre entre {g} et {d}")
    rep = ""
    nb_coups = 0
    while ...:
        m = (g+d)//2
        rep = ""
        while rep not in ["+", "-", "="]:
            rep = input(f"C'est {m} ? (+/-/=) ")
        if rep == "+":
            ...
        elif rep == "-":
            ...
        else:
            g = ...
            d = ...
        nb_coups += 1
    if ...:
        print(f"Dis donc, tu triches, non ?")
    else:
        print(f"J'ai trouvé en {nb_coups} coups")
```

```
>>> ordinateur(1, 5)
Pensez à un nombre entre 1 et 5
C'est 3 ? (+/-/=) -
C'est 1 ? (+/-/=) +
C'est 2 ? (+/-/=) =
J'ai trouvé en 3 coups
>>> ordinateur(1, 5)
Pensez à un nombre entre 1 et 5
C'est 3 ? (+/-/=) -
C'est 1 ? (+/-/=) -
Dis donc, tu triches, non ?
```

Exercice 9 : Écrire une fonction `ordinateur2(secret, g=1, d=63)` qui renvoie le nombre d'essais nécessaire pour trouver le secret par recherche dichotomique entre `g` et `d`. Si le secret n'est pas entre `g` et `d`, la fonction renvoie 0.

```
>>> ordinateur2(190, 1, 3000)
11
>>> ordinateur2(0, 1, 3000)
0
>>> {i: ordinateur2(i, 1, 7) for i in range(1, 8)}
{1: 3, 2: 2, 3: 3, 4: 1, 5: 3, 6: 2, 7: 3}
```

La dernière commande permet de créer un dictionnaire qui associe à chaque nombre le temps qu'il faut pour le trouver.

Exercice 10 : Écrire une fonction `nb_moyen(g=1, d=63)` qui détermine le nombre d'essais moyens qu'il faut pour trouver n'importe quel nombre entre `g` et `d`.

```
>>> nb_moyen()
5.095238095238095
>>> nb_moyen(1, 1000000)
18.951445
```

Pour aller plus loin

Exercice 11 : Compléter la fonction `cherche_nombre(g=1, d=63)` qui propose de trouver un nombre entre `g` et `d`, sauf que le nombre n'est pas choisi à l'avance et qu'à chaque fois l'ordinateur essaie de donner la pire des réponses possible à l'utilisateur, jusqu'à ce qu'il ne reste qu'un seul choix possible. Il faut que les réponses données soient cohérentes.

```
def cherche_nombre(g=1, d=63):
    nb_essais = 0
    print(f"On joue entre {g} et {d}")
    while ...:
        rep = int(input("Donnez un nombre : "))
        nb_essais += 1
        if ...: # Un seul choix et il l'a trouvé
            break # Pour sortir de la boucle
        elif rep < g: # Valeur vraiment trop petite
            print("C'est plus")
        elif ...: # Valeur vraiment trop grande
            print("C'est moins")
        elif ...: # Il y a plus de valeur au dessus
            g = ...
            print("C'est plus")
        else: # Il y a plus de valeurs en dessous
            d = ...
            print("C'est moins")
    if g == d: # C'est qu'il a trouvé
        print(f"Bravo vous avez trouvé {g} en {nb_essais} coups")
    else: # Cela ne devrait jamais arriver
        print("étrange", g, d)
```

```
>>> cherche_nombre(1, 10)
On joue entre 1 et 10
Donnez un nombre : 5
C'est plus
Donnez un nombre : 7
C'est plus
Donnez un nombre : 9
C'est moins
Donnez un nombre : 8
Bravo vous avez trouvé 8 en 4 coups
>>> cherche_nombre(1, 10)
On joue entre 1 et 10
Donnez un nombre : 6
C'est moins
```

```
Donnez un nombre : 3
C'est moins
Donnez un nombre : 2
C'est moins
Donnez un nombre : 1
Bravo vous avez trouvé 1 en 4 coups
```

La dichotomie est aussi utilisée pour chercher des valeurs approchées de solutions d'équations que l'on ne sait pas ou ne peut pas calculer en un temps raisonnable. Dans ce cas, on va s'arrêter lorsque la taille de l'intervalle est inférieur à un seuil déterminé à l'avance. Par exemple, la fonction suivante permet de déterminer une valeur approchée de la racine carrée de `n`, avec une précision de moins de `seuil` :

```python
def recherche_racine(n, seuil=0.000001, g=0, d=-1):
    if d < g or not (g**2 <= n <= d**2):
        d = n
    while d-g > seuil:
        m = (g+d)/2
        carre = m**2
        if carre > n:
            d = m
        elif carre < n:
            g = m
        else:
            g = m
            d = m
    if g == d:
        print(f"La racine de {n} est {g}")
    else:
        print(f"La racine de {n} est entre {g} et {d}")
```

Il est possible de donner des valeurs de `g` ou de `d`, mais si elles ne sont pas valides, on prend des valeurs qui permettent de trouver la réponse. C'est surtout utile lorsqu'il peut y avoir plusieurs solutions à une équations.

```
>>> recherche_racine(7, 0.0001)
La racine de 7 est entre 2.645721435546875 et 2.6457748413085938
>>> recherche_racine(7, 0.0001, 0, 1)
La racine de 7 est entre 2.645721435546875 et 2.6457748413085938
>>> recherche_racine(7, 0.0001, 2, 3)
La racine de 7 est entre 2.64569091796875 et 2.645751953125
>>> recherche_racine(7, 0.0000000001)
La racine de 7 est entre 2.6457513110144646 et 2.6457513110653963
>>> recherche_racine(100001023, 0.000000001)
La racine de 100001023 est entre 10000.051149868865 et 10000.05114986956
>>> recherche_racine(4, 0.0000000001)
La racine de 4 est 2.0
>>> recherche_racine(25, 0.0000000001)
La racine de 25 est entre 4.99999999992724 et 5.00000000001819
```

On peut remarquer que la solution exacte, lorsqu'elle est calculable, n'est pas forcément trouvée.