Complexité et Dichotomie

I. Introduction à la complexité

En informatique, l'exécution de tout programme, fonction, portion de code, etc. consomme des ressources : cela prends du temps, utilise de la mémoire, consomme de l'énergie.

Conséquence numéro 1 On évite de recalculer plusieurs fois la même chose.

Exercice 1 Supposons que l'on a le code suivant, qui utilise la fonction `premier(n)` qui calcule le `n`-ème nombre premier, et la fonction `bizarre2` dont le principal rôle est de vous empêcher de connaître à l'avance le nombre d'itération de la boucle **`while`**.

```
def bizarre(n):
    s = 0
    i = 1
    while i < premier(n):
        s = s + 1
        i = bizarre2(i)
    return s
```

On s'aperçoit que `premier(n)` est exécuté *à chaque itération* alors que sa valeur est toujours la même, il n'y a donc pas besoin de la recalculer.

1. Exécutez `bizarre(100)`, `bizarre(200)` et `bizarre(1000)`[1].
2. Modifier la fonction `bizarre_rapide` (qui au départ a exactement le même code que `bizarre`) pour éviter de recalculer `premier(n)` à chaque fois.
3. Comparez les vitesses d'exécution de `bizarre` et `bizarre_rapide` pour les arguments `100`, `200` et `1000`.

1. Complexité temporelle

Pour étudier la vitesse d'un fonction, nous allons étudier le temps mis par cette fonction non pas dans l'absolu (cela n'a pas beaucoup de sens et varie beaucoup, pour la même fonction, d'un ordinateur à l'autre). À la place, nous allons utiliser la notion suivante :

[1] Si vous trouvez que l'exécution est trop longue, vous pouvez faire `ctrl c`

Définition

- ⋆ La **complexité temporelle** d'une fonction est le temps mis pour son exécution en fonction de la taille de l'entrée.
- ⋆ La **taille** de l'entrée est une mesure de la taille des arguments. Les exemples les plus typiques sont la valeur d'un entier et la taille d'un tableau.
- ⋆ On parle de complexité **dans le pire des cas**, c'est à dire que l'on considère le maximum des temps pour une taille d'entrée donnée[a].
- ⋆ On ne considère en général que l'ordre de grandeur de la variation, en n'en considérant qu'un équivalent, voire un « grand O ».

[a] Éventuellement, on considère le maximum des temps pour des entrées de taille *au plus* la taille indiquée

2. Complexités constante, linéaire et quadratique

⋆ Une fonction est de complexité **constante** lorsque son temps d'exécution ne dépend pas de la taille de l'entrée.

Exemple La fonction `len` qui détermine la taille d'un tableau est de complexité constante.

⋆ Une fonction est de complexité **linéaire** si son temps d'exécution croît comme la taille de l'entrée.

Exemple Un parcours simple de tableau avec le corps de la boucle en temps constant est linéaire.

⋆ Une fonction est de complexité **quadratique** si son temps d'exécution croît comme le carré de la taille de l'entrée.

Exemple Une fonction avec deux boucles imbriquées dont les bornes sont majorées par un entier n, et donc le corps est de complexité constante, est quadratique en n.

Exercice 2 Déterminer la complexité des fonctions suivantes.

```
def somme(t):
    s = 0
    for i in range(len(t)):
        s = s + t[i]
    return s

def plus_proches(t):
    d = abs(t[1] - t[0])
```

```
    for i in range(len(t)):
        for j in range(i):
            d = min(d, abs(t[i] - t[j]))
    return d
```

3. Quelques expérimentations

Nous allons vérifier expérimentalement les complexités temporelles des fonctions précédentes.

Dans le fichier que vous utilisez, il y a une fonction `chrono` qui prends deux arguments `f` et `x` et retourne le temps en secondes mis pour exécuter `f(x)`.

Par exemple, pour engendrer une liste de 10 000 de nombres aléatoires de $[0,1]$ et calculer sa somme, on peut faire la chose suivante :

```
>>> l = [random.random() for i in range(10000)]
>>> chrono(somme, l)
0.0008182250003301306
```

Question 3 Calculer plusieurs fois le temps mis pour calculer la somme de `l` (que vous aurez définie au préalable).

Vous pouvez remarquer qu'il y a des fluctuations dans les durées obtenues. Nous y reviendrons après avoir collecté plus de données.

Question 4 Écrire une fonction `chrono_somme(n, f)` qui calcule et renvoie le temps cumulé pour calculer `f` fois la somme d'une liste aléatoire à `n` éléments.

Vous pouvez maintenant exécuter `trace_somme()` pour voir affichés plusieurs exécution de `chrono_somme` pour différentes longueurs.

Normalement, le bas des courbes forme une droite. C'est ce qui compte et illustre le fait que la complexité de `somme` est linéaire. Les valeurs hautes ne sont pas à prendre en compte, comme expliqué dans cet extrait recopié depuis la documentation du module `timeit` qui permet de chronométrer des temps d'exécution.

> Il est tentant de vouloir calculer la moyenne et l'écart-type des résultats et notifier ces valeurs. Ce n'est cependant pas très utile. En pratique, la valeur la plus basse donne une estimation basse de la vitesse maximale à laquelle votre machine peut exécuter le fragment de code spécifié ; les valeurs hautes de la liste sont typiquement provoquées non pas par une variabilité de la vitesse d'exécution de Python, mais par d'autres processus interférant avec la précision du chronométrage. Le `min()` du résultat est probablement la seule valeur à laquelle vous devriez vous intéresser. Pour aller plus loin,

vous devriez regarder l'intégralité des résultats et utiliser le bon sens plutôt que les statistiques.

II. Dichotomie et complexité logarithmique

Nous avons vu lors de l'introduction à la récursivité, nous avons beaucoup vu d'exemples où l'avancée vers un cas de base se fait à l'aide d'un entier naturel qui décroit vers 0 de 1 en 1.

Nous allons ici étudier des cas où on divise un entier naturel par 2 plutôt que de lui soustraire 1, ce qui conduit à moins d'étapes d'appels récursifs ou d'itérations d'une boucle. Nous allons étudier deux premiers algorithmes mettant en œuvre une telle accélération.

1. Exponentiation rapide

Pour calculer a^n, on peut utiliser la fonction itérative suivante :

```
def expo_iter(a, n):
    r = 1
    for i in range(1, n + 1):
        r = r * a
    return r
```

On peut aussi se baser sur les relations :

$$a^0 = 1 \qquad \forall\, n \geqslant 1,\ a^n = a^{n-1} \times a$$

Exercice 5 Écrire une fonction `expo_rec(a, n)` qui traduit ces relations.

Il existe cependant un méthode bien plus efficace pour calculer une exponentielle. On peut tout d'abord remarquer que

$$a^{2n} = (a^n)^2$$

Ainsi, pour passer de l'exposant n à $2n$, il suffit d'un produit (lequel ?) et non n.

Question 6 Comment obtenir a^{2n+1} à partir de a et de a^n en seulement deux produits ?

Question 7 En déduire une fonction récursive calculant *efficacement* une exponentielle. On fera en particulier attention à l'écriture du ou des cas de bases, et on s'assurera que les appels récursifs s'approchent bien d'un cas de base, en conformité avec les impératifs de la récursivité.

Question 8

1. Montrer par récurrence sur n que pour tout $n \in \mathbf{N}$ et tout $k \in [\![2^n, 2^{n+1}-1]\!]$, le calcul de a^k avec l'algorithme d'exponentiation rapide se fait en au plus $2n$ multiplications.
2. En déduire que le nombre de multiplications nécessaires pour calculer a^n avec cet algorithme est majoré par $2\lfloor \log_2 n \rfloor$.

2. Recherche dichotomique

Nous allons reprendre la fonction de recherche d'un élément dans un tableau.

Exercice 9 Écrire une fonction `recherche(t, e)` qui détermine si `e` est une valeur présente dans `t` et renvoie :

- ⋆ l'indice d'*une* occurence de `e` dans `t` ;
- ⋆ `-1` si `e` n'est pas présent dans `t`.

On se basera sur un parcours partiel de tableau.

Question 10 Justifier que la fonction précédente s'exécute en temps linéaire (dans le pire des cas) en fonction de la taille du tableau.

Dans la suite, nous allons supposer que les tableaux dans lesquels on fait une recherche **est ordonné de façon croissante.**

Question 11 On suppose que l'on a un tableau `t` de longueur n, dans lequel on cherche un élément `e`. En notant `n` la longueur de `t`, on compare `e` à `t[n // 2]`. Trois cas sont possibles :

- ⋆ Si `e == t[n // 2]`, que faire ?
- ⋆ Si `e < t[n // 2]`, que peut-on dire des positions éventuelles des occurences de `e` dans `t` ?
- ⋆ Quel est le troisième cas, et que peut-on dire alors ?

Supposons maintenant que l'on a deux indices `deb` et `fin` tels que si la valeur `e` est présente dans `t`, alors c'est nécessairement entre les indices `deb` et `fin` (inclus). Autrement dit, `e` ne peut pas être présent dans `t` à l'extérieur de ces indices et, comme le tableau est ordonné, cela revient à avoir la propriété (P) qui est vérifiée :

$$\forall\, k \in [\![0, \texttt{len\{t\}}[\![, \left(k < \texttt{deb} \implies \texttt{t}[k] < \texttt{e} \ \textbf{et}\ k > \texttt{fin} \implies \texttt{t}[k] > \texttt{e}\right) \qquad \text{(P)}$$

Question 12 À quelle condition sur `deb` et `fin` est-on sûr que `e` n'apparaît pas dans `t` ?

Question 13 On suppose maintenant que `deb` $\leqslant$ `fin` et on pose

```
mil = (deb + fin) // 2.
```

et on compare `e` avec `t[mil]`.

1. En cas d'égalité, que peut-on dire ?
2. Si `e < t[mil]`, où est-on sûr de ne pas pouvoir trouver `e` ? Comment modifier `deb` ou `fin` en conséquence, de façon à ce que (P) soit toujours vérifée ?
3. Et si `e > t[mil]` ?

Question 14 Compléter le code de la fonction `recherche_dichotomique`.

```
def recherche_dichotomique(t, e):
    deb = 0
    fin = len(t) - 1
    while ...:
        mil = (deb + fin) // 2
        if t[mil] == e:
            return ...
        elif t[mil] > e:
            ...
        else:
            ...
    # on n'a pas trouvé e
    return -1
```

Question 15 Testez votre fonction en l'essayant avec divers arguments. Vous essayerez de tester le plus de situations possibles (argument présent, argument plus petit que le plus petit élément du tableau, tableau vide, etc.) en vous assurant que le résultat obtenu est correct et en corrigeant votre fonction le cas échéant.

On suppose maintenant qu'au début de la boucle **`while`**, les variables `deb` et `fin` sont respectivement égales à d_0 et f_0, et l'on ne sort pas de la boucle à l'aide du **`return`** et qu'à la fin, leurs valeurs sont respectivement d_1 et f_1.

Question 16 Montrer que $f_1 - d_1 + 1 \leqslant \frac{1}{2}(f_0 - d_0 + 1)$.

Question 17 Montrer que si la taille initiale du tableau est n, alors au début de la k-ème itération, on a :

$$\texttt{fin} - \texttt{deb} + 1 \leqslant \frac{n}{2^k}.$$

Question 18 En déduire que la fonction précédente fera au plus $\lceil \log_2 n \rceil$ itérations.

III. S'il vous reste du temps

1. Retour sur les expérimentations

Question 19 Comme expliqué, il convient de ne retenir que le minimum des temps mesurés. Vous pouvez donc essayer de modifier `chrono_somme` pour retourner le temps minimum parmi les `f` temps mesurés.

Question 20 Inspirez-vous des fonctions `chrono_somme` et `test_somme` pour, de même, représenter la vitesse d'exécution de la fonction `plus_proches`. De par son caractère quadratique, on limitera la taille maximale des longueur des tableaux.

2. Recherche dichotomique, le retour

Question 21 De même, vous pouvez tenter de tracer la courbe des temps mis pour une recherche dichotomique pour bien illustrer le caractère logarithmique.

Question 22 La propriété (P) est vérifiée avant d'entrer dans la boucle (lorsque l'on a `deb = 0` et `fin = len(t) - 1`), et à chaque tour de boucle, elle reste vraie entre le début et la fin du corps de la boucle. C'est un *invariant de boucle* qui permet de prouver que l'algorithme est bien correct. On peut en déduire que lorsque l'on sort de la boucle, elle est encore vérifiée. Montrer qu'alors, si l'on renvoie `-1`, cela signifie bien que `e` n'est pas présent dans `t`.

Question 23 **(Plus compliqué)** Nous allons considére une variante de l'invariant précédent :

$$\forall k \in [\![0, \texttt{len(t)}[\![, \left(k < \texttt{deb} \implies \texttt{t}[k] < \texttt{e} \textbf{ et } k \geqslant \texttt{fin} \implies \texttt{t}[k] \geqslant \texttt{e}\right) \qquad \text{(P')}$$

ainsi que le programme à compléter suivant :

```
def recherche_dichotomiquebis(t, e):
    deb = 0
    fin = len(t)
    while deb < fin:
        mil = (deb + fin) // 2
        if t[mil] < e:
            deb = mil + 1
        else: # t[mil] >= e
            ...
    return ...
```

1. Justifier que (P') est vérifié avant d'entrer dans la boucle.

2. On suppose maintenant que (P') est vérifié au début de l'exécution du corps de la boucle, et que l'on a `t[mil] < e`. Montrer que (P') est encore vérifié à la fin du corps de la boucle.

3. Si `t[mil] >= e`, comme modifier `fin` pour que (P') reste aussi vérifiée.

4. Si (P') est vérifiée en sortie de boucle, et qu'alors `deb = fin` (ce que l'on admettra), que peut-on dire ? En déduire une deuxième version de la recherche dichotomique.

5. Comparer sa vitesse à la première.