

CORRIGÉ - TP 10 : PROGRAMMATION ET COMPLEXITÉ

1 Recherche clé

1.

```
def valeur_associee(L, x):
    for (c, v) in L:
        if c == x:
            return v
    return None
```

2. Notons n la longueur de L .

Dans le meilleur des cas, si la clé cherchée est en première position dans la liste, alors la complexité est en $O(1)$.

Dans le pire des cas, si aucune valeur n'est associée à x alors on fait un parcours de toute la liste en exécutant à chaque fois le même nombre d'opérations; on a donc une complexité en $O(n)$.

3.

```
def valeur_associee_dicho(L, x):
    g = 0
    d = len(L) - 1
    while d - g >= 0:
        m = (g + d) // 2
        c, v = L[m]
        if c == x:
            return v
        elif x < c:
            d = m - 1
        else:
            g = m + 1
    return None
```

4. On montre par récurrence que $d - g \leq \frac{n}{2^{k-1}}$.

Initialisation. Si il y a une première itération, alors $d - g = n - 1 \leq \frac{n}{2^{1-1}}$. Donc la propriété est vraie pour $k = 1$.

Hérédité. On suppose que $0 \leq d - g \leq \frac{n}{2^{k-1}}$. Si on a trouvé la clé dans la liste, l'algorithme s'arrête et il n'y a pas de nouvelle itération. Sinon 2 cas possibles :

- $d' = \lfloor \frac{g+d}{2} \rfloor - 1$ et $g' = g$ et alors $d' - g' \leq \frac{d-g}{2} \leq \frac{n}{2^k}$;
- ou $d' = d$ et $g' = \lfloor \frac{g+d}{2} \rfloor + 1$ et encore $d' - g' \leq \frac{d-g}{2} \leq \frac{n}{2^k}$.

Dans tous les cas la propriété est héréditaire.

5. On est sûr d'être sur la dernière itération si $c = d$, car soit l'élément cherché est à l'indice d et on a fini, soit il n'y est pas et alors à l'issue du passage dans la boucle, $d - c < 0$ donc on tombe dans le critère d'arrêt de la boucle **while**.

Ainsi, d'après la question précédente, il suffit que $\frac{m}{2^{k-1}} < 1$, soit $\log_2(n) + 1 < k$. C'est vérifié pour $k = \lfloor \log_2(n) \rfloor + 2$.

6. Puisque pour chaque passage dans la boucle il y a un nombre fixe d'opérations, et que dans le pire des cas on fait $\lfloor \log_2(n) \rfloor + 2$ passages dans la boucle, on en déduit une complexité en $O(\ln(n))$.

Dans le meilleur des cas, la clé est au milieu et est trouvée lors du premier passage dans la boucle; on a une complexité en $O(1)$.

2 Tri rapide

1.

```
def repartition(L):
    p = L[0]
    L1, L2, L3 = [], [], []
    for x in L:
        if x < p:
            L1.append(x)
        elif x == p:
            L2.append(x)
        else:
            L3.append(x)
    return L1, L2, L3
```

2. On note n la longueur de L . Pour chaque passage dans la boucle on a un nombre fixe d'opérations (2 tests et un ajout à une liste). On fait n passages dans la boucle, on a donc une complexité en $O(n)$.

3.

```
def tri_rapide(L):
    if len(L) <= 1:
        return L
    L1, L2, L3 = repartition(L)
    return tri_rapide(L1) + L2 + tri_rapide(L3)
```

4. Si tous les éléments sont distincts, cela signifie qu'à chaque appel de la fonction `repartition` la liste $L2$ est de longueur 1. Sous l'hypothèse que les appels récursifs se font toujours sur des listes maximales déséquilibrées, alors à chaque appel à la fonction `repartition`, $L1$ est de longueur $n - 1$ et $L3$ vide ou le contraire. Si on note $T(n)$ la complexité du tri rapide pour une liste de longueur n , on a :

$$T(n) = T(n - 1) + O(n).$$

On en déduit que $T(n) = O(n^2)$.

5. Si maintenant les listes sont maximalelement équilibrées alors L_1 et L_3 sont de même taille (à 1 près) $\frac{n}{2}$. On a donc :

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

On retrouve une relation du type de celle vue en cours pour le tri fusion. On a donc une complexité en $O(n \ln(n))$.

6. D'après les deux questions précédentes, dans le pire des cas on a une complexité en $O(n \ln(n))$ et dans le pire des cas en $O(n^2)$.
7. Si on retire l'hypothèse que tous les éléments sont distincts, alors le meilleur des cas est lorsque tous les éléments de la liste sont égaux. Un appel suffit alors, on a une complexité en $O(n)$ (un appel à la fonction `repartition`).

3 Recherche de mot dans un texte

- 1.

```
def recherche_mot(texte, mot):
    for i in range(len(texte)):
        if mot == texte[i:i+len(mot)]:
            return i
    return None
```

2. Notons n la longueur du texte et m la longueur du mot. Pour chaque passage dans la boucle `for` on fait m comparaisons. On a donc une complexité en $O(mn)$.

4 Suite récurrente et complexité

1. Si on note $T(n)$ la complexité du calcul de $u(n)$, on a alors pour $n \geq 1$, $T(n) = 2T(n-1)$. La suite $(T(n))_n$ est donc géométrique et on a une complexité en $O(2^n)$.
2. On propose :

```
def u(n):
    if n==0:
        return 1
    p = u(n-1)
    return (2*p+3) / (p-5)
```

On a alors $T(n) = T(n-1) + c$ avec c une constante, on en déduit qu'on a désormais une complexité en $O(n)$.

5 Inclusion ensembliste

- 1.

```
def est_incluse(L1, L2):
    for x in L1:
        if x not in L2:
            return False
    return True
```

2. On note n_1 la longueur de L_1 et n_2 la longueur de L_2 .
Dans le pire des cas, aucun des éléments de L_1 n'est dans L_2 . Alors, pour chaque valeur de x dans L_1 on fait n_2 comparaisons. On a donc une complexité en $O(n_1 n_2)$.
3. On propose :

```
def est_incluse2(L1, L2):
    n1, n2 = len(L1), len(L2)
    R1 = tri_fusion(L1)
    R2 = tri_fusion(L2)
    i1, i2 = 0, 0
    while i1 < n1 and i2 < n2:
        if R1[i1] < R2[i2]:
            return False
        elif R1[i1] == R2[i2]:
            i1 += 1
        else:
            i2 += 1
    return i1 == n1
```

On suppose les tailles des deux listes comparables à peu près égales à n .
Dans ce cas on fait 2 tris fusion en complexité $O(n \ln(n))$ d'après le cours. Ensuite on fait au maximum $n_1 + n_2$ passages dans la boucle `while`. On a donc une complexité en $O(n \ln(n)) + O(2n) = O(n \ln(n))$.