TP D'INFORMATIQUE 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 (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 (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 $0 \leq d - f \leq \frac{n}{2^{k-1}}$. 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 :
   (a) $d' = \lfloor \frac{g+d}{2} \rfloor - 1$ et $g' = g$ et alors $d' - g' \leq \frac{d-g}{2} \leq \frac{n}{2^k}$;
   (b) 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 en déduit une complexité en $(\ln(n))$.
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 (1).

2 Tri fusion

1.

```
def fusion(L1, L2):
  n1 = len(L1)
    n2 = len(L2)
    i1 = 0
    i2 = 0
    R = []
    while i1 < n1 and i2 < n2:
    if L1[i1] <= L2[i2]:
      R.append(L1[i1])
      i1 = i1 + 1
    else:
      R.append(L2[i2])
      i2 = i2 + 1
  for i in range(i1, n1):
    R.append(L1[i1])
  for i in range(i2, n2):
    R.append(L2[i2])
  return R
```

2. Chaque passage dans une boucle est en $O(1)$, et il y a en tout $n1 + n2$ passages dans ces trois boucles. La complexité est donc en $O(n1 + n2)$.

3.

```
def tri_fusion(L):
    if len(L) <= 1:
        return L
    m = n//2
    L1 = L[0:m]
    L2 = L[m:n]
    return fusion(tri_fusion(L1), tri_fusion(L2))
```

4. La complexité $T(n)$ d'un appel sur une liste de longueur n vérifie la relation de récurrence :

$$T(n) = 2T(n/2) + O(n)$$

où $T(n/2)$ est le coût de chaque appel récursif, et $O(n)$ est le coût de création de $L1$ et $L2$ et de l'appel à la fonction fusion.
D'après le cours, on a alors $T(n) = O(n \log n)$.

3 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 (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 maximalement 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 :

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

On en déduit que $T(n) = (n^2)$.

5. Si maintenant les listes sont maximalement équilibrées alors `L1` et `L3` sont de même taille (à 1 près) $\frac{n}{2}$. On a donc :

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

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

6. D'après les deux questions précédentes, dans le pire des cas on a une complexité en $(n\ln(n))$ et dans le pire des cas en (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 (n) (un appel à la fonction `repartition`).

4 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 (mn).

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 `L1` et n_2 la longueur de `L2`.
Dans le pire des cas, aucun des éléments de `L1` n'est dans `L2`. Alors, pour chaque valeur de `x` dans `L1` on fait n_2 comparaisons. On a donc une complexité en (n_1n_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 $(n\ln(n))$ d'après le cours. Ensuite on fait au maximum `n1+n2` passages dans la boucle `while`. On a donc une complexité en $(n\ln(n)) + (2n) = (n\ln(n))$.