

Nombre de boucles

1. $C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} n = n^2$
2. $C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$
3. Si $n = 2^p$ alors $C(n) = \sum_{k=1}^p 1 = p = \log_2(n)$ et dans le cas général, $C(n) = 1 + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ donc C est croissante (récurrence forte) et si $2^p \leq n \leq 2^{p+1} - 1$, on a $p \leq C(n) < p+1$ donc $C(n) = \lfloor \log_2(n) \rfloor$.
4. Si $n = 2^p$, $C(n) = \sum_{k=0}^{p-1} \sum_{j=0}^{2^{p-k}-1} 1 = \sum_{k=0}^{p-1} 2^{p-k} = 2^p \frac{1-2^{-p}}{1-2^{-1}} = 2(2^p - 1) = 2(n-1)$. Dans le cas général, $C(n) = n + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ ce qui donne à nouveau une suite croissante et donc, si $2^p \leq n < 2^{p+1}$, on a $2(2^p - 1) \leq C(n) \leq 2(2^{p+1} - 1)$ donc $C(n) = O(n)$.

Codes à complexité imposée

1.

```
def max1(liste) :
    M = liste[0]
    for i in liste[1:] :
        if i > M :
            M = i
    return M
```
2.

```
def maxi2(liste) :
    n = len(liste)
    a = liste[0]
    # on balaye la liste jusqu'à trouver un autre élément
    i = 1
    while a == liste[i] :
        i += 1
    b = liste[i]
    # on range ces deux premiers éléments et on initialise les variables
    if a < b :
        M1, M2 = b, a
        L1, L2 = [i], list(range(i))
    else :
        M1, M2 = a, b
        L1, L2 = list(range(i)), [i]
    # on balaye le reste et on actualise les variables selon les cas
    for j in range(i+1, n) :
        elt = liste[j]
        if elt > M1 :
            M1, M2, L1, L2 = elt, M1, [j], L1[:]
        elif elt == M1 :
            L1.append(j)
        elif elt > M2 :
            M2, L2 = elt, [j]
        elif elt == M2 :
            L2.append(j)
    return M2, L2
```

Liste ou dictionnaire

1. Le test `x in L2` possède une complexité linéaire, on l'effectue `len(L1)` fois donc la complexité est $O(n^2)$.
2. La création du dictionnaire `d` est en $O(n)$ et comme le test `x in d` possède une complexité en $O(1)$, la complexité de `Inter` devient $O(n)$.
3. On utilise un dictionnaire pour mémoriser les éléments déjà ajoutés à la copie de `L` :

```
def SansDoublon(L) :  
    d = {}  
    L1 = []  
    for x in L :  
        if not x in d :  
            d[x] = True  
            L1.append(x)  
    return L1
```

Améliorer la complexité

1. La fonction `CC` renvoie la longueur de la plus grande séquence croissante qui commence à l'indice `x`.
2. Dans le pire des cas, la liste `L` est croissante et `x=0`, on parcourt alors toute la liste. La complexité est donc en $O(n)$.
3. La fonction `PGCC` fait appel à la fonction `CC` à chaque boucle `for` (2 fois en plus!) donc la complexité de `PGCC` est quadratique dans le pire des cas (une liste croissante).
4. Il suffit de ne pas tester tous les éléments de `L` mais de ne recalculer la longueur d'une composante croissante que à partir de la fin de la précédente :

```
def PGCC2(L) :  
    M = CC(L, 0)  
    i = M  
    while i < len(L) :  
        m = CC(L, i)  
        if m > M :  
            M = m  
        i += m  
    return M
```

On parcourt la liste `L` une seule fois, en entier, donc la complexité est linéaire.