La complexité d'un algorithme est un moyen de mesurer son efficacité : il s'agit d'estimer le coût de son exécution, soit en terme de temps de calcul (on parle de complexité temporelle), soit en terme d'utilisation de la mémoire (on parle de complexité spatiale).
Pour mesurer la complexité d'un algorithme, il faut d'abord déterminer les opérations fondamentales que l'on souhaite dénombrer : les opérations effectuées (addition, multiplication, ...), les tests (==, !=, ...), les accès à des données (lecture des données d'une liste, d'un tableau, ...), la création d'espaces en mémoire, le nombre d'utilisation d'une certaine fonction, ...
Cette complexité est en général évaluée lorsque l'on applique la fonction à un objet (l'ensemble des paramètres) de « taille » fixée : la longueur d'une liste par exemple. Il faut donc introduire une quantité « mesurable » pour évaluer cette taille.
La plupart du temps, on ne cherche pas à déterminer une valeur exacte pour la complexité (qui n'aurait pas vraiment de sens car toutes les opérations n'ont pas le même coût) mais une majoration de la complexité. On a ainsi une idée du temps nécessaire à l'exécution du code. Pour cette raison, les complexités sont souvent données sous la forme $O(\,)$ et il est souvent suffisant de dénombrer le nombre de boucles effectuées.
On considère différentes complexités :

- La complexité dans le meilleur des cas (pour l'exécution de la fonction sur un objet de taille n fixée) : c'est la valeur minimale de la complexité pour une taille d'objet donnée.
- La complexité dans le pire des cas : c'est la valeur maximale.
- La complexité en moyenne : si on introduit une probabilité sur l'ensemble des objets de taille donnée (uniforme par exemple), la complexité peut être vue comme une variable aléatoire ; la complexité en moyenne est alors son espérance.

Vocabulaire :

$O(1)$	constante
$O(\log(n))$	logarithmique
$O(n)$	linéaire
$O(n^2)$	quadratique
$O(q^n)$ avec $q > 1$	exponentielle

I Complexité de certaines opérations de bases

1. Listes

Le type `list` possède un certain nombre de fonctions ou méthodes dont les coûts ne sont pas équivalents.
Dans le tableau suivant n désigne la longueur d'une liste `L`

`len(L)`	$O(1)$	longueur de la liste
`L.append(elt)`	$O(1)$	ajout en fin de liste
`L.pop()`	$O(1)$	suppression en fin de liste
`L[i]`	$O(1)$	accès à un élément (ou modification)
`L[p:q]`	$O(q-p)$	slicing
`elt in L`	$O(n)$	test d'appartenance

Remarque(s) :

(*I.1*) Les lignes suivantes ont le même effet mais pas le même coût
- `L.append(elt)` et `L=L+[elt]`
- `L.pop()` et `L=L[:-1]`

2. Dictionnaires

`len(d)`	$O(1)$	longueur du dictionnaire
`d[cle]`	$O(1)$	ajout, lecture ou modification d'une entrée
`del(d[cle])`	$O(1)$	suppression d'une entrée
`cle in d`	$O(1)$	test d'appartenance d'une clef
`val in d.values()`	$O(n)$	test d'appartenance d'une valeur

II Exemple du calcul des puissances

1. Calcul itératif

On peut calculer x^n à partir de l'égalité $x^n = x \times x^{n-1}$:

```
def p(x,n) :
    r = 1
    for _ in range(n) :
        r *= x
    return r
```

La complexité de `p(x,n)` est alors $O(n)$

2. Calcul dichotomique

En remarquant $x^n = \begin{cases} \left(x^2\right)^{n/2} & \text{si } n \text{ est pair} \\ x \times \left(x^2\right)^{(n-1)/2} & \text{si } n \text{ est impair} \end{cases}$

```
def P(x,n) :
    r = 1
    while n>0 :
        if n%2 == 1 :
            r *= x
        x = x**2
        n = n//2
    return r
```

La complexité dans le pire des cas vérifie

$$C(n) = 1 + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

donc

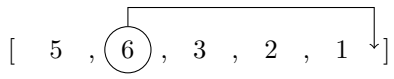
$$C(n) = O(\log(n))$$

III Algorithmes de tri

1. Tris quadratiques

1..1 Tri par sélection

Le principe du tri par sélection est de rechercher la maximum de la liste et de la placer à la fin puis de recommencer sur la liste restante :

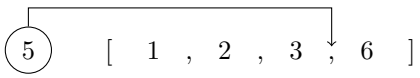


La recherche du maximum d'une liste de longueur n est $O(n)$; il faut le faire successivement pour des listes de longueurs n, $n-1$, …, 2 donc la complexité est

$$O\left(\sum_{k=2}^{n} k\right) = O\left(n^2\right)$$

1..2 Tri par insertion

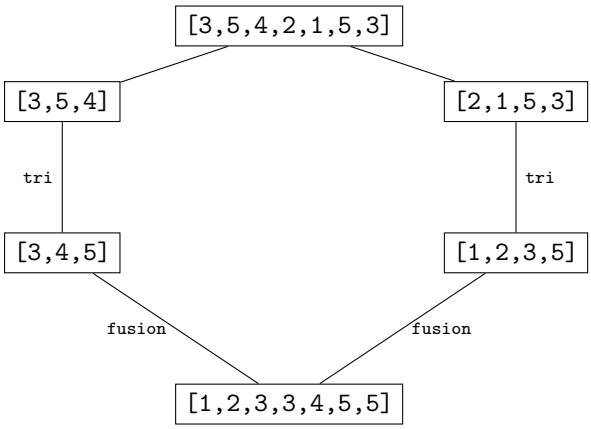
Le principe du tri par insertion est de trier la liste au fur et à mesure en cherchant à insérer les nouveaux éléments à leur bonne place :



La recherche de la position d'insertion dans une liste de longueur n est $O(n)$; il faut le faire pour des listes de longueurs 2, 3, …, n donc la complexité du tri est à nouveau $O\left(n^2\right)$

2. Tri fusion

Le principe du tri fusion est de diviser la liste en deux listes de longueurs « égales », de les trier (récursivement) avant de les fusionner en conservant l'ordre croissant :

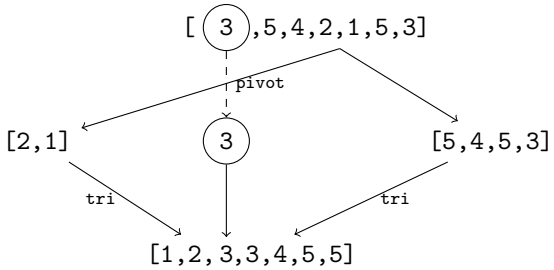


La complexité de la fusion de deux listes de longueurs p et q est $O(p+q)$ donc la complexité du tri fusion vérifie

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n) = O(n\log(n))$$

3. Tri rapide

Le principe du tri rapide est de choisir un pivot, de séparer la liste en deux sous-listes, celle des éléments inférieurs au pivot et celle des éléments supérieurs au pivot ; reste ensuite à trier, récursivement, ces deux sous-listes :



Dans le pire des cas, le pivot est, par exemple, le plus petit élément de la liste à chaque fois (donc la liste était déjà triée) et la complexité sera alors $O\left(n^2\right)$. L'intérêt du tri rapide est que sa complexité en moyenne est $O(n\log(n))$, contrairement aux tris par sélection et par insertion pour lesquels la complexité moyenne reste quadratique.