Introduction à la complexité des algorithmes

Complexité algorithmique

Le séquencement d'un programme ou d'une partie d'un programme repose sur l'exploitation de ressources matérielles disponibles sur la machine sur laquelle est réalisée l'exécution. La performance de cette exécution est ainsi conditionnée par les caractéristiques de la machine mais elle est aussi étroitement impactée par la manière avec laquelle sont utilisées et mises en œuvre les variables et les instructions du programme. La structure d'un algorithme influence ainsi directement la performance de l'exécution d'un programme au cœur duquel il est implémenté. La complexité algorithmique s'attache alors à évaluer et comparer la performance des algorithmes afin de quantifier leur efficacité au regard du coût de leur exécution. Des bouts de code écrits en langages Python et C sont présentés afin d'illustrer les explications.

Différentes formes de la complexité algorithmique

L'analyse de la complexité d'un algorithme consiste à décrire l'efficacité de son exécution en termes de quantité de mémoire nécessaire pour traiter les données et de temps de traitement. Cette analyse est ainsi déclinée selon deux aspects distincts :

— complexité temporelle,
— complexité spatiale.

La complexité temporelle d'un algorithme est évaluée à partir du nombre d'opérations élémentaires nécessaires à son exécution. Une opération élémentaire correspond à une instruction assurant une affectation, une comparaison ou une opération arithmétique. La complexité temporelle agit donc directement sur la charge de calcul assurée par le processeur de la machine et donc sur le temps d'exécution de l'algorithme.

La complexité spatiale d'un algorithme relève de la quantification de l'espace mémoire utilisé lors de son exécution. Le nombre et le type de variables mises en œuvre constituent ainsi le critère essentiel de cette évaluation. Par conséquent, la complexité spatiale est liée à l'espace mémoire nécessaire à l'exécution de l'algorithme.

D'une manière générale, c'est surtout la complexité temporelle, et donc le coût processeur, qui est prise en compte pour mesurer et comparer la performance des algorithmes.

Utilisation de la notation de Landau

La notation couramment employée pour exprimer la complexité d'un algorithme est celle proposée en 1909 par Edmund Georg Hermann Landau (1877-1938) et qui repose sur des travaux de Paul Bachmann (1837-1920) publiés en 1894. Cette notation est dite *grand O*. D'autres notations ont cependant été proposées et notamment celles de Godfrey Harold

Hardy (1877-1947) en 1910 et de Ivan Matveievitch Vinogradov (1891-1983) en 1930.

L'expression de la complexité algorithmique repose sur le comportement asymptotique de la mesure en fonction de la taille n des données passées à l'algorithme. Des ordres de grandeur sont alors définis afin de classer les algorithmes en fonction de leur temps d'exécution ou de l'espace mémoire consommé au cours de leur exécution selon qu'il s'agit d'exprimer respectivement la complexité temporelle, notée $T(n)$, ou la complexité spatiale, notée $S(n)$.

Enfin et de manière courante, la complexité est considérée dans le cas le plus défavorable de l'exécution de l'algorithme.

Principe de détermination de la complexité

La détermination de la complexité temporelle d'un algorithme revient à comptabiliser le nombre d'opérations élémentaires effectuées durant son exécution. Par ailleurs, l'hypothèse retenue est que toutes les opérations élémentaires sont à égalité de coût, soit une unité de temps.

À titre d'exemple, les bouts de code ci-dessous concernent une fonction `permut()` destinée à permuter les valeurs de deux variables entières `l[0]` et `l[1]` contenues dans une liste `l` qui est un tableau à une dimension passé en argument à la fonction :

```
def permut(l):
    tmp = l[0]
    l[0] = l[1]
    l[1] = tmp
    return l
```

```
int *permut(int *l)
{
  int tmp;

  tmp = l[0];
  l[0] = l[1];
  l[1] = tmp;
  return l;
}
```

Que le langage utilisé soit Python ou C et hormis le renvoi de la liste, les opérations élémentaires sont au nombre de 3 et sont toutes des affectations. La complexité temporelle de la fonction de permutation est donc de 3, soit $T(n) = 3$.

Une autre manière de procéder consiste à effectuer la permutation de la valeur des variables `l[0]` et `l[1]` sans faire appel à une variable de stockage temporaire. Les exemples ci-après présentent une autre possibilité d'implémentation de la fonction `permut()` :

```
def permut(l):
    l[0] = l[0] + l[1]
    l[1] = l[0] - l[1]
    l[0] = l[0] - l[1]
    return l
```

```
int *permut(int *l)
{
  l[0] = l[0] + l[1];
  l[1] = l[0] - l[1];
  l[0] = l[0] - l[1];
  return l;
}
```

Dans ce second cas, le nombre d'opérations élémentaires est plus conséquent puisqu'il prend en compte trois affectations mais aussi une addition et deux soustractions. Ce qui aboutit à une complexité temporelle de 6, soit $T(n) = 6$.

La première version de l'implémentation de la fonction `permut()` est ainsi plus performante du point de vue de la complexité temporelle. En revanche, elle met en œuvre trois variables, soit $S(n) = 3$, alors que la seconde version se limite à deux variables et présente de fait une meilleure complexité spatiale avec $S(n) = 2$.

Cas des structures conditionnelles

Le séquencement d'un programme incluant des structures conditionnelles est déterminé par l'évaluation logique de différentes conditions. En effet, certains blocs de code ne sont exécutés qu'en fonction du résultat d'une évaluation logique. Les bouts de code simplifiés ci-dessous montrent un exemple d'application pour une structure conditionnelle de type `if...else` :

```
if i > 10:
    # Exécution du bloc A
else:
    # Exécution du bloc B
```

```
if (i > 0)
  /* Exécution du bloc A */
else
  /* Exécution du bloc B */
```

Le principe qui s'applique est celui de considérer le pire cas entre les différents blocs de code, soit dans l'exemple ci-dessus $max(tA, tB)$ où tA et tB correspondent à la complexité temporelle de chacun des deux blocs de code.

Le code des fonctions présentées ci-dessous s'applique à un test sur une variable entière passée en argument d'une fonction `test()` dont le résultat conditionne le nombre d'opérations élémentaires exécutées et donc la complexité temporelle :

```
def test(n):
    if n > 5:
        return n *= n
    else:
        return n
```

```
int test(int n)
{
  if (n > 5)
    return n *= n;
  else
    return n;
}
```

Dans cet exemple, il convient donc de considérer que la complexité algorithmique temporelle de la fonction `test()` est fixée par le bloc de code exécuté lorsque la condition `n > 5` est vérifiée puisque son traîtement nécessite deux opérations élémentaires de plus que pour l'autre bloc de code.

Analyse des structures itératives simples

Les structures itératives reposent sur une ou plusieurs boucles dont le temps d'exécution est impacté par le nombre d'éléments à parcourir et le nombre d'opérations élémentaires à exécuter.

Les exemples de code ci-dessous concernant le cas simple d'une boucle utilisée pour additionner une succession de valeurs entières croissantes permettent d'illustrer la progression de la complexité temporelle en fonction du nombre d'éléments à considérer :

```
def addi(n):
    s = 0

    for i in range(1, n + 1):
        s += i

    return s
```

```
int addi(int n)
{
  int i;
  int s = 0;

  for (i = 1; i < n + 1; i++)
    s += i;

  return s;
}
```

Dans le cas présenté ci-dessus, le nombre d'opérations élémentaires effectuées à chaque itération de la boucle est de 4. Il s'agit de l'incrémentation de la variable d'index `i`, de la comparaison de la valeur de cette variable avec la valeur de la variable `n` passée en argument, de l'addition des valeurs des variables `i` et `s` ainsi que de l'affectation de la variable `s`. L'évaluation de la complexité de la fonction `addi()` doit également prendre en compte l'initialisation préalable des variables `s` et `i` correspondant à des opérations élémentaires effectuées une seule fois.

Les quatre opérations élémentaires traitées à chaque itération de la boucle relèvent d'une complexité algorithmique temporelle en $O(1)$. Néanmoins, dès lors qu'elles sont exécutées n fois, la complexité résultante est plus grande. Le calcul de la complexité de la fonction `addi()` se décompose ainsi sous la forme $T(n) = 1+1+4n$, soit $T(n) = 2+4n$. La complexité algorithmique obtenue est alors considérée en $O(n)$. Elle est appelée *complexité linéaire*.

L'expression de la complexité algorithmique temporelle de la fonction `addi()` est ainsi la suivante :

$$T(n) = \sum_{i=1}^{n} O(1) = O(n)$$

La limite supérieure n de l'expression de sommation ci-dessus est bien en conformité avec la condition d'exécution `i < n + 1` de la boucle. En effet, lorsque la variable `i` atteint la valeur $n+1$, les traitements associés à la boucle ne sont pas effectués. Dans le cas du langage C, la condition d'exécution `i < n + 1` peut aussi être indifféremment codée sous la forme `i <= n`.

Le cas des boucles imbriquées implique une dégradation de la complexité algorithmique temporelle par rapport à celle engendrée par une boucle unique. En effet et dans le cas de deux boucles imbriquées, il aboutit à une multiplication des itérations de la boucle interne par celles de la boucle externe.

Les bouts de code ci-dessous montrent un exemple de mise en œuvre de deux boucles imbriquées dans une fonction destinées à effectuer l'addition de plusieurs séries de valeurs entières successives stockées dans un tableau à deux dimensions égales, c'est-à-dire une matrice carrée d'ordre n :

```python
def matrice(n):
    s = 0

    for i in range(1, n + 1):
        for j in range(1, n + 1):
            s += j

    return s
```

```c
int matrice(int n)
{
  int i;
  int j;
  int s = 0;

  for (i = 1; i < n + 1; i++)
    for (j = 1; j < n + 1; j++)
      s += j;

  return s;
}
```

Le nombre d'opérations élémentaires effectuées à chaque itération de la boucle externe est de 2. Ces opérations élémentaires concernent l'incrémentation de la variable d'index `i` et la comparaison de la valeur de cette variable avec la valeur correspondant à `n + 1`. La boucle interne implique des traîtements plus conséquents et le nombre d'opérations élémentaires effectuées est de 4 comme dans le cas d'une boucle simple présenté plus haut. Aux opérations élémentaires associées aux itérations des deux boucles, il convient d'ajouter les opérations préalables d'initialisation des variables `i`, `j` et `s`.

Le calcul de la complexité algorithmique temporelle de la fonction `matrice()` correspond alors à $T(n) = 1 + 1 + 1 + (2n * 4n)$, soit $T(n) = 3 + 8n^2$. Dans ce cas, la complexité algorithmique obtenue est polynomiale de degré 2 et se note $O(n^2)$. Elle est appelée *complexité quadratique*.

L'expression de la complexité algorithmique temporelle de la fonction `matrice()` relève d'une somme de somme. Elle est la suivante :

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} O(1) = O(n^2)$$

Dans le cas où la matrice n'est pas de type carré, les dimensions notées n et m amènent à une expression de la complexité algorithmique temporelle de la forme suivante :

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{m} O(1) = O(n * m)$$

Cependant et puisque le principe est d'évaluer le pire cas, la complexité doit être également considérée en $O(n^2)$.

De manière générale et lorsque les variables de boucle font l'objet d'une incrémentation ou d'une décrémentation ou qu'elles évoluent à chaque itération au moyen d'une opération d'addition ou de soustraction, le nombre p de boucles imbriquées fixe le degré de la compléxité algorithmique de la fonction ou de la portion de code concernée, soit une complexité en $O(n^p)$. Il est alors possible de généraliser ces cas de figure sous l'appellation de *complexité polynomiale*.

Ainsi, pour une imbrication de trois boucles, la complexité polynomiale est de degré 3 et se note $O(n^3)$. Elle est spécifiquement appelée *complexité cubique*.

Les exemples ci-après concernent le cas de figure de la complexité cubique dans le contexte du parcours d'un tableau à trois dimensions homogènes :

```python
def cube(n):
    s = 0

    for i in range(1, n + 1):
        for j in range(1, n + 1):
            for k in range(1, n + 1):
                s += j

    return s
```

```c
int cube(int n)
{
  int i;
  int j;
  int k;
  int s = 0;

  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
      for (k = 1; k <= n; k++)
        s += j;

  return s;
}
```

La complexité algorithmique temporelle est $T(n) = 1 + 1 + 1 + 1 + (2n * 2n * 4n)$, ce qui donne $T(n) = 4 + 16n^3$.

Bien évidemment et tout comme dans le cas des matrices, les dimensions du tableau parcouru ne sont pas nécessairement homogènes.

Évaluation de quelques structures itératives particulières

L'exemple de code ci-après concerne une fonction assurant un traitement itératif implémenté dans une boucle `while` mais dont l'exécution est contrôlée par le doublement de la valeur de la variable de boucle `i` à chaque itération :

```
def fonction_1(n):
    i = 1

    while i <= n:
        i *= 2

    return i
```

```
int fonction_1(int n)
{
  int i = 1;

  while (i <= n)
    i *= 2;

  return i;
}
```

Dans ce cas particulier, le nombre d'itérations effectuées par la boucle `while` dépend directement de la vitesse d'évolution et de convergence de la valeur i de la variable `i` vers la valeur n de la variable `n` passée en argument de la fonction. Cette évolution n'est pas linéaire dès lors qu'elle est conditionnée par des doublements successifs effectués à chaque itération de la boucle.

Le tableau 1 montre l'évolution de la valeur i de la variable `i` retournée par la fonction `fonction_1()`. La relation entre le nombre k d'itérations et la valeur i de la variable `i` est mise en évidence. Elle correspond à $i = 2^k$.

Le nombre maximal d'itérations est alors fixé en fonction de la valeur $log_2(n)$. Comme ce nombre peut être un réel, il convient de procéder à un abaissement à l'entier inférieur afin de borner le nombre maximal

Nombre k d'itérations	Valeur i	
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4
5	32	2^5
6	64	2^6
7	128	2^7
8	256	2^8
9	512	2^9
10	1024	2^{10}
11	2048	2^{11}
12	4096	2^{12}
...	...	...

Tab. 1 – *Évolution de la valeur retournée*

d'itérations par un nombre entier. Ainsi et à titre d'exemple, pour une borne fixée à 127 par la valeur n, 6 itérations seront exécutées. La valeur i de la variable `i` n'atteint donc pas nécessairement la borne correspondant à la valeur $\lfloor log_2(n) \rfloor$ et elle peut en aucun cas la dépasser.

Il est ainsi possible d'exprimer mathématiquement la complexité algorithmique temporelle de la fonction `fonction_1()` de la façon suivante :

$$T(n) = \sum_{log_2(i)=1}^{\lfloor log_2(n) \rfloor} O(1) = O(log_2(n))$$

Dans ce cas, la classe de complexité algorithmique est appelée *complexité logarithmique*. Elle est généralisée par la notation $O(log(n))$.

De façon générale, la complexité algorithmique temporelle d'une boucle est considérée comme étant en $O(log(n))$ si la variable de boucle est divisée ou multipliée par une valeur constante qui fixe la base logarithmique.

L'exemple ci-dessous présente le contexte d'un bout de code basé sur une imbrication de boucles dans lequel la boucle externe est de type `for` et la boucle interne est de type `while` :

```
def fonction_2(n):
    for i in range(n):
        j = 1

        while j <= n:
            j *= 2

    return i * j
```

```c
int fonction_2(int n)
{
  int i;
  int j;

  for (i = 1; i <= n; i++)
  {
    while (j <= n)
      j *= 2;
  }

  return i;
}
```

L'exécution de la seule boucle interne relève d'une complexité logarithmique et donc en $O(log(n))$ comme dans le cas de l'exemple précédent. La boucle externe, quant à elle, correspond à une complexité algorithmique linéaire, soit en $O(n)$.

La complexité algorithmique temporelle de la fonction `fonction_2()` correspond ainsi à l'expression mathématique suivante :

$$T(n) = \sum_{i=1}^{n} \sum_{log_2(i)=1}^{\lfloor log_2(n) \rfloor} O(1) = O(n\, log_2(n))$$

De façon génrale, la considération des deux boucles imbriquées avec leur complexité algorithmique temporelle respective, comme dans le cas de la fonction `fonction_2()`, aboutit à un résultat de complexité en $O(n\, log(n))$. Cette classe de complexité est appelée *complexité quasi-linéaire*.

Autres classes de complexité

Au-delà des différentes classes de complexité présentées ci-dessus, il existe des cas pour lesquels le coût de traitement est très élevé voire irréaliste du point de vue de la mise en application et de la sollicitation de la ressource processeur.

En effet et compte tenu de la dégradation engendrée et du temps d'exécution nécessaire, ces classes de complexité sont à proscrire lorsque le nombre d'éléments à considérer est important.

Les principales classes concernées sont, dans l'ordre croissant de leur coût temporel, la *complexité quasi-polynomiale* correspondant à la notation $O(n^{log(n)})$, la *complexité exponentielle* notée $O(a^n)$ et la *complexité factorielle* exprimée en $O(n!)$.

Cas des fonctions récursives

La complexité algorithmique d'une fonction récursive peut être décrite comme une relation de récurrence mathématique. Pour la déterminer, il faut alors résoudre les récurrences. Le cas du calcul de la factorielle d'un nom entier est présenté en exemple ci-après :

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

```c
int fact(int n)
{
  if (n == 0)
    return 1;
  else
    return (n * fact(n - 1));
}
```

Dans l'exemple présenté, le traitement récursif s'arrête lorsque la valeur n de la variable `n` devient nulle. Lorsque cette valeur est non nulle, le programme exécute deux opérations élementaires que sont la multiplication de la valeur n par la valeur retournée par la fonction de rang $n-1$ et le test de la valeur n. Il faut ajouter à ces deux opérations élémentaires le nombre d'opérations élémentaires relevant de l'appel de la fonction de rang $n-1$.

Il est alors possible de déterminer la complexité algorithmique de la fonction par $T(n) = T(n-1)+2$, soit une complexité linéaire.

La complexité algorithmique correspond donc à une suite arithmético-géométrique. Ce qui est toujours le cas pour les fonctions récursives.

De façon générale et en considérant le cas d'une fonction faisant a fois appel à cette même fonction et dans laquelle b opérations élémentaires sont exécutées, il est possible d'exprimer le contexte de complexité algorithmique temporelle sous la forme $T(n) = aT(n-1) + b$. Ce cas de figure correspond à une complexité algorithmique exponentielle.

Un second exemple de fonction récursive est celui du calcul de la suite de Fibonacci :

```python
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

```
int fibo(int n)
{
  if (n < 2)
    return n;
  else
    return (fib(n - 1) + fib(n - 2));
}
```

Dans cet exemple, l'appel de la fonction `fib()` est réalisé à deux reprises au sein même de la fonction. Sa complexité est ainsi exponentielle.

Synthèse des principales classes de complexité

Il est important de connaître la classe de complexité d'un algorithme, d'un programme ou d'une fonction afin d'estimer son degré de performance. En effet, plus sa classe se rapproche de la complexité constante, soit $O(1)$, et plus son exécution sera performante. Cependant, certains problèmes ne permettent pas toujours d'atteindre une méthode de résolution selon une complexité algorithmique avantageuse.

Le tableau 2 consigne et synthétise par ordre croissant des coûts d'exécution les principales classes de complexité algorithmique et les notations associées.

Notation	Classe
$O(1)$	Constante
$O(log(n))$	Logarithmique
$O(n)$	Linéaire
$O(n\,log(n))$	Quasi-linéaire
$O(n^2)$	Quadratique
$O(n^3)$	Cubique
$O(n^p)$	Polynomiale
$O(n^{log(n)})$	Quasi-polynomiale
$O(a^n)$	Exponentielle
$O(n!)$	Factorielle

Tab. 2 – *Principales complexités algorithmiques*

La figure 1 présente une comparaison graphique des classes de complexité algorithmique courantes en termes d'évolution du nombre d'opérations élémentaires au regard du nombre d'éléments à prendre en compte pour l'exécution de l'algorithme.

Afin d'obtenir une représentation cohérente des différentes courbes, des échelles logarithmiques sont appliquées pour l'abcisse et l'ordonnée.

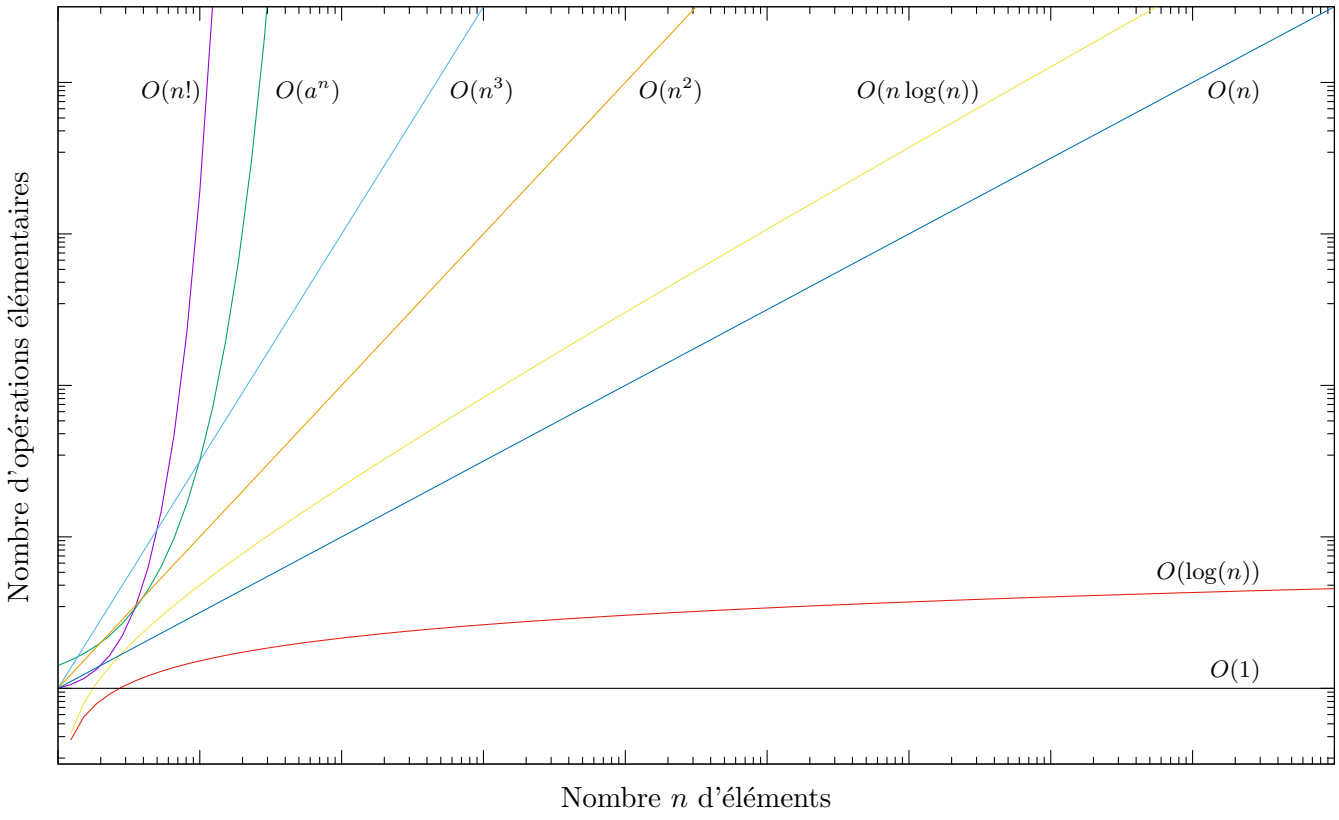

Fig. 1 – *Représentation graphique des complexités algorithmiques temporelles*