T-PMTH-402 – Math. appliquées à l'info.
Chapitre 7 - La complexité algorithmique

Jean-Sébastien Lerat
Jean-Sebastien.Lerat@heh.be

Haute École en Hainaut

2019-2020

Plan

Introduction

Les objectifs de ce chapitre sont

1. d'évaluer la faisabilité d'exécution d'algorithmes afin de déterminer s'ils s'exécutent dans un temps raisonnable ;
2. comparer l'efficacité de deux algorithmes.

Définition

Complexité

La **complexité** d'un algorithme est une fonction qui exprime le nombre d'opérations fondamentales effectuées par cet algorithme sur un ensemble de données $|D| = n$

Note : la complexité permet d'estimer le temps de calcul d'un algorithme.

Définition

Complexité

La **complexité** d'un algorithme est une fonction qui exprime le nombre d'opérations fondamentales effectuées par cet algorithme sur un ensemble de données $|D| = n$

Note : la complexité permet d'estimer le temps de calcul d'un algorithme.

Exemple de temps de calculs

Nombre d'opérations	1Ghz	3Ghz
$4,1015 \times 10^9$	$\approx 6,7$ semaines	$\approx 2,2$ semaines
$7,7245 \times 10^6$	≈ 2 heures	≈ 43 minutes

Note : la complexité exprime donc la faisabilité d'exécution dans un temps raisonnable indépendamment de la puissance de calcul non parallélisé.

Notation

La complexité d'un algorithme se décline sous trois évaluations :

Meilleur des cas noté $o(\bullet)$ correspond à la complexité minimal, c'est-à-dire quand les données sont favorables à l'algorithme.

Cas moyen noté $\theta(\bullet)$ correspond à la complexité moyenne, c'est-à-dire sur des données quelconques.

Pire des cas noté $O(\bullet)$ correspond à la complexité maximale, c'est-à-dire quand les données sont défavorables à l'algorithme.

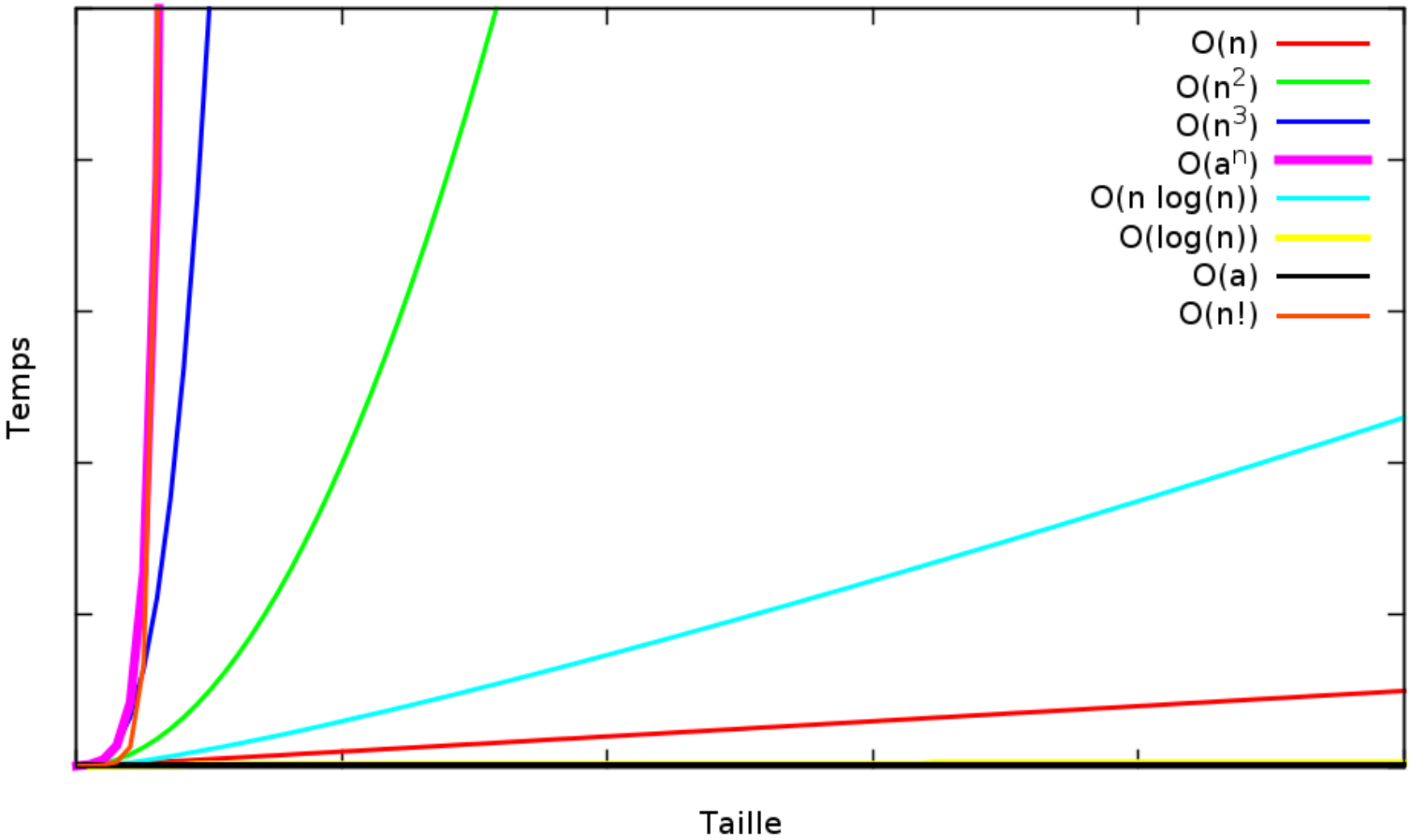
La notation $\Theta(\bullet)$ *peut être utilisée dans le cas où* $o(\bullet) = \theta(\bullet) = O(\bullet)$.

Remarque

Le calcul de complexité peut également être utilisé afin d'estimer l'espace mémoire nécessaire au lieu du temps d'exécution.

Plan

Temps



Noms

Constante	$O(a)$
Logarithmique	$O(\log(n))$
Quasi-linéaire	$O(n\ \log(n))$
Linéaire	$O(n)$
Quadratique	$O(n^2)$
Polynomiale	$O(n^p)$
Exponentielle	$O(a^n)$
Factorielle	$O(n!)$

où a est une constante.

Noms

Constante	$O(a) \Leftrightarrow O(1)$
Logarithmique	$O(\log(n))$
Quasi-linéaire	$O(n\ \log(n))$
Linéaire	$O(n)$
Quadratique	$O(n^2)$
Polynomiale	$O(n^p)$
Exponentielle	$O(a^n) \Leftrightarrow O(2^n)$
Factorielle	$O(n!)$

où a est une constante.

La complexité n'est exprimée qu'avec le terme englobant, le plus significatif.
Par exemple : $2n! + 4^2 + 3$

Plan

Cas simples – $O(1)$

Allocation

L'allocation correspond à la demande d'un espace mémoire dans la pile d'exécution.

Exemple d'allocation en C

```
1 int number;
```

Cas simples – $O(1)$

Affectation

L'affectation correspond à l'accès d'une adresse mémoire et de l'écriture de cette valeur

Exemple d'affectation en C

```
1        number = 5;
```

Cas simples – $O(1)$

Initialisation

L'initialisation correspond à une déclaration et à une affectation.

Exemple d'initialisation en C

```
1        int number = 5;
```

Cas simples – $O(1)$

Accès

L'accès à une variable nécessite un accès à la pile. L'accès à un tableau nécessite un calcul d'adresse.

Exemple d'accès en C

```
1 array[42];
```

$$address(array) + 42$$

Cas simples – $O(1)$

Modification

La modification est une affectation. La modification d'une cellule d'un tableau nécessite au préalable un calcul d'adresse.

Exemple de modification en C

```
1        array[42] = 5;
```

$$address(array) + 42 \Leftrightarrow 42$$

Cas simples – $O(1)$

Opérations

Chaque opération nécessite un traitement via le CPU. Les opérations sont plus ou moins rapide en fonction du nombre de portes logiques à opérer (généralement basé sur des portes NAND).

Exemple d'opération en C

```
1       4 + 2;
        number * array[42];
3
```

Condition

Exemple de condition en C

```
if(a < b){
    // bloc d'instructions 1
}else if(a > b){
    // bloc d'instructions 2
}else {
    // bloc d'instructions 3
}
```

$$o(\bullet) = o(\ min(\ o(\text{bloc d'instructions 1}), o(\text{bloc d'instructions 2}), o(\text{bloc d'instructions 3})\ )\ )$$

$$O(\bullet) = O(\ max(\ O(\text{bloc d'instructions 1}), O(\text{bloc d'instructions 2}), O(\text{bloc d'instructions 3})\ )\ )$$

$$\theta(\bullet) = \theta(\frac{1}{|A|\times|B|} \sum_{a\in A, b\in B} \text{nombres d'opérations avec } a, b)$$

Itération (boucle)

Exemple de boucles

```
1  int i,j;
   for(i=0;i<10;i++){
3      j = 0;
       // bloc d'instruction 1
5      while(j < 5){
           // bloc d'instruction 2
7          j += 1;
       }
9  }
```

$$\theta(10 \times (\text{bloc d'instruction 1} + 5 \times \text{bloc d'instruction 2}))$$

Plan

Complexité

$$array = [1, 2, 3, 4]$$

```
1 def insertionSort(array):
      sortedArray = [ None ] * len(array) # O(1)
3     for (pos,element) in enumerate(array):
          sortedArray[position] = element  # O(1)
5         while(pos > 0 and element < sortedArray[position -1]):
              sortedArray[pos],sortedArray[pos-1] = sortedArray[
      pos - 1] , sortedArray[pos]  # O(1)
7             pos = pos - 1 # O(1)
      return sortedArray # O(1)
```

Meilleur des cas ($o(\bullet)$)

Quel est le meilleur des cas ?

Quel est sa complexité ?

```
def insertionSort(array):
    sortedArray = [ None ] * len(array) # O(1)
    for (pos,element) in enumerate(array):
        sortedArray[position] = element  # O(1)
        while(pos > 0 and element < sortedArray[position -1]):
            sortedArray[pos],sortedArray[pos-1] = sortedArray[
    pos - 1] , sortedArray[pos]  # O(1)
            pos = pos - 1 # O(1)
    return sortedArray # O(1)
```

Meilleur des cas ($o(\bullet)$)

La liste est déjà triée :

$$array = [1, 2, 3, 4]$$

```
def insertionSort(array):
    sortedArray = [ None ] * len(array) # O(1)
    for (pos,element) in enumerate(array):
        sortedArray[position] = element  # O(1)
        while(pos > 0 and element < sortedArray[position-1]):
            sortedArray[pos],sortedArray[pos-1] = sortedArray[
    pos - 1] , sortedArray[pos]  # O(1)
            pos = pos - 1 # O(1)
    return sortedArray # O(1)
```

Pas besoin de la seconde boucle ...

Meilleur des cas ($o(\bullet)$)

La liste est déjà triée :

$$array = [1, 2, 3, 4]$$

```
def insertionSort(array):
    sortedArray = [ None ] * len(array) # O(1)
    for (pos,element) in enumerate(array):
        sortedArray[position] = element  # O(1)
    return sortedArray # O(1)
```

$$|array| = n \Rightarrow o(n)$$

Pire des cas ($O(\bullet)$)

Quel est le pire des cas ?

Quel est sa complexité ?

```
1 def insertionSort(array):
      sortedArray = [ None ] * len(array) # O(1)
3     for (pos,element) in enumerate(array):
          sortedArray[position] = element  # O(1)
5         while(pos > 0 and element < sortedArray[position -1]):
              sortedArray[pos],sortedArray[pos-1] = sortedArray[
      pos - 1] , sortedArray[pos]  # O(1)
7             pos = pos - 1 # O(1)
      return sortedArray # O(1)
```

Pire des cas ($O(\bullet)$)

La liste est déjà triée dans le sens inverse : $array = [4, 3, 2, 1]$

```
def insertionSort(array):
    sortedArray = [ None ] * len(array) # O(1)
    for (pos,element) in enumerate(array):
        sortedArray[position] = element  # O(1)
        while(pos > 0 and element < sortedArray[position-1]):
            sortedArray[pos],sortedArray[pos-1] = sortedArray[
    pos - 1] , sortedArray[pos]  # O(1)
            pos = pos - 1 # O(1)
    return sortedArray # O(1)
```

La seconde boucle est utilisée à chaque fois.
Pour chaque élément i que l'on va insérer parmi les n éléments, il faudra effectuer $n - i$ échange dans la seconde boucle :

$$O(0 + 1 + 2 + 3 \ldots) = O\left(\frac{n(n-1)}{2}\right)$$

Le terme englobant est n^2 car $\frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n)$

Cas moyen ($\Theta(\bullet)$)

Qu'en est-il de la complexité du cas moyen ?

```
def insertionSort(array):
    sortedArray = [ None ] * len(array) # O(1)
    for (pos,element) in enumerate(array):
        sortedArray[position] = element  # O(1)
        while(pos > 0 and element < sortedArray[position-1]):
            sortedArray[pos],sortedArray[pos-1] = sortedArray[
    pos - 1] , sortedArray[pos]  # O(1)
            pos = pos - 1 # O(1)
    return sortedArray # O(1)
```

Cas moyen ($\Theta(\bullet)$)

Moyenne des complexités sur toutes les combinaisons possibles de la liste $[4, 3, 2, 1]$

$\Leftrightarrow$ liste à moité triée, à moitié non-trié

```
def insertionSort(array):
    sortedArray = [ None ] * len(array) # O(1)
    for (pos,element) in enumerate(array):
        sortedArray[position] = element  # O(1)
        while(pos > 0 and element < sortedArray[position -1]):
            sortedArray[pos],sortedArray[pos-1] = sortedArray[
    pos - 1] , sortedArray[pos]  # O(1)
            pos = pos - 1 # O(1)
    return sortedArray # O(1)
```

$\Leftrightarrow$ Comme le pire des cas divisé par deux (à moitié triée)

$\Rightarrow O\left(\frac{1}{2}\frac{n(n-1)}{2}\right) \Rightarrow O(n^2)$

Plan

Exercices – Tri en bulles

```
def bubbleSort(array):
    for repeat in range(len(array) - 1):
        for pos in range(1, len(array)):
            if (array[pos] < array[pos - 1]):
                array[pos], array[pos - 1] = array[pos - 1],
    array[pos]
    return array
```

Exercices – Tri rapide

```
def partition(array,left,right):
    pivot     = array[right]
    sortedPos = left
    for j in range(left,right):
        if(array[j] <= pivot):
            array[sortedPos],array[j] = array[j],array[
    sortedPos]
            sortedPos += 1
    array[sortedPos],array[right] = array[right],array[
    sortedPos]
    return sortedPos

def quickSort(array,left=0,right=None):
    if(right is None): right = len(array)-1
    if(left < right):
        pivotPos = partition(array,left,right)
        quickSort(array,left,pivotPos-1)
        quickSort(array,pivotPos+1,right)
    return array
```

Exercices – Tri par fusion

```
def mergeSort(array):
    n = len(array)
    if (n <= 1): return array
    division = n >> 1
    part1 = mergeSort(array[:division])
    part2 = mergeSort(array[division:])
    posPart1, posPart2 = len(part1) - 1, len(part2) - 1
    while (posPart1 >= 0 and posPart2 >= 0):
        n -= 1
        if (part1[posPart1] > part2[posPart2]):
            array[n] = part1[posPart1]
            posPart1 -= 1
        else:
            array[n] = part2[posPart2]
            posPart2 -= 1
    while (posPart1 >= 0):
        n        -= 1
        array[n] = part1[posPart1]
        posPart1 -= 1
    while (posPart2 >= 0):
        n        -= 1
        array[n] = part2[posPart2]
        posPart2 -= 1
    return array
```

Exercices – Tri par tas

```
def heapify(array,subTree,size):
    left = (subTree << 1) + 1
    right = left + 1
    minNode = subTree
    if(left < size and array[left] < array[minNode]): minNode = left
    if(right < size and array[right] < array[minNode]): minNode = right
    if(minNode != subTree):
        array[subTree],array[minNode] = array[minNode],array[subTree]
        heapify(array,minNode,size)
        if(not subTree == 0):
            parent = subTree >> 1
            if(subTree % 2 == 0): parent -= 1
            heapify(array,parent,size)

def heapSort(array):
    for i in range(len(array)>>1 , -1, -1):
        heapify(array,i,len(array))
    size = len(array)
    while(size > 1):
        size -= 1
        array[0],array[size] = array[size],array[0]
        heapify(array,0,size)
    return array
```

Solutions

Algorithme de tri	$o(\bullet)$	$\theta(\bullet)$	$O(\bullet)$
en bulles	$o(n^2)$	$\theta(n^2)$	$O(n^2)$
par insertion	$o(n)$	$\theta(n^2)$	$O(n^2)$
par rapide	$o(n\log(n))$	$\theta(n\log(n))$	$O(n^2)$
par fusion	$o(n\log(n))$	$\theta(n\log(n))$	$O(n\log(n))$
par tas	$o(n\log(n))$	$\theta(n\log(n))$	$O(n\log(n))$