La Complexité

October 20, 2023

1 La complexité

1.1 Coût d'un programme (ou d'une fonction)

Coût d'un programme (ou d'une fonction) est le nombre d'opérations élementaires effectuées par le programme (ou la fonction)

Les opérations élémentaires sont :

1. Les opérations de lecture (input) ;
2. Les opérations d'écriture (output à l'aide de la fonction print) ;
3. Les opérations arithmétiques (+, -, *, /, //, %) ;
4. Les opérations de tests ou de comparaison (==, !=, <, >, <=, >=) ;
5. Les affectations.

Calculons les coût des programme suivant :

[1]:
```python
a = 2           # Une affectation, coût = 1
b = 7           # Une affectation, coût = 1
s = a + b       # Une affectation et une opération arithmétique, coût = 2
print(s)        # Une opération d'écriture, coût = 1
# Le coût du programme est 1 + 1 + 2 + 1 = 5
```

```
9
```

[2]:
```python
a = input("Veuillez saisir une lettre : ")     # Une affectation, une opération␣
 ↪de lecture et une opération d'écriture, coût = 3
print("La lettre que vous avez saisi est : ") # Une opération d'écriture, coût␣
 ↪= 1
print(a) # Une opération d'écriture, coût = 1
# Le coût du programme est 3 + 1 + 1 = 5
```

```
Veuillez saisir une lettre :  5
```

```
La lettre que vous avez saisi est :
5
```

[ ]:
```python
a = input("Veuillez saisir une lettre : ")     # Une affectation, une opération␣
 ↪de lecture et une opération d'écriture, coût = 3
```

```python
print("La lettre que vous avez saisi est : ", a) # Deux opérations d'écriture, coût = 2
# Le coût du programme est 5
```

[ ]:
```python
a = 5                    # Une affectation, coût = 1
if a <= 5:               # Un test, coût = 1
  print("Oui")           # Une opération d'écriture, coût = 1
  print("inférieur ou égal à 5")    # Une opération d'écriture, coût = 1
else:
  print("Non")                      # Une opération non éxécutée
print("Fin du programme")           # Une opération d'écriture, coût = 1
# Le coût du programme est 5
```

```
Oui
inférieur ou égal à 5
Fin du programme
```

[ ]:
```python
a = int(input("Veuillez saisir un entier : ")) # Une affectation, une opération de lecture et une opération d'écriture, coût = 3
if a <= 5:               # Un test, coût = 1
  print("Oui")           # Une opération d'écriture, coût = 1
  print("inférieur ou égal à 5")    # Une opération d'écriture, coût = 1
else:
  print("Non")                      # Une opération d'écriture, coût = 1
print("Fin du programme")           # Une opération d'écriture, coût = 1

# La valeur de a est inconnue, si a est inférieure ou égale à 5 le coût sera 7 sinon le coût sera 6
# On dit que le coût dans le pire des cas est 7
# On dit que le coût dans le meilleur des cas est 6
# coût du programme <= 7
```

[ ]:
```python
for i in range(6):
  print('ok')
  print(i)

# Chaque itération de la boucle for execute 2 opérations d'écriture et une affectation de i ce qui vaut 3
# La boucle for execute 6 itérations, le cout du programme est donc 3 * 6 = 18
```

```
ok
0
ok
1
ok
2
ok
```

```
3
ok
4
ok
5
```

[ ]:
```python
for i in range(5):
  print("Ok")
  for j in range(4):
    print("Non")
    print(i * j)

# Chaque itération de la boucle for j execute 4 opérations élémentaires␣
↪(affectation de j, 2 print, *)
# La boucle for j fait 4 itérations, le cout de la boucle est donc 4 * 4 = 16

# Chaque itération de la boucle for i execute 18 opérations (affectation de i,␣
↪print, for j)
# La boucle for i execute 5 itérations, le cout de la boucle est donc 18 * 5 =␣
↪90
```

[ ]:
```python
for i in range(7):
  print("Ok")
  for j in range(i):
    print("Non")
    print(i * j)
```

[3]:
```python
i = 6
while i <= 10:
  print(i)
  i = i + 1

# Chaque itération de la boucle while effectue 4 opérations élémentaires (test,␣
↪print, =, +)
# La boucle while execute 5 itérations et un test qui résulte en l'arret de la␣
↪boucle
# Le cout de la boucle while est 4 * 5 + 1 = 21
# Le cout du programmes est donc 22 (En tenant compte de i = 6)
```

```
6
7
8
9
10
```

[ ]:
```python
# calculons le cout de la fonction suivante
def somme_liste(L):
  n = len(L)
  s = 0
  for i in range(n):
    s = s + L[i]
  return s

# Supposons que le cout de la focntion len est 1, le cout des instructions hors␣
↪la boucle est 4
# Chaque itération de la boucle for execute 3 opérations (affectation de i,␣
↪affectation de s et +)
# La boucle for execute n itérations donc son cout est 3n
# Le cout de la fonction est donc 3n + 4
# On remarque que le cout de la fonction varie en fonction de n linéairement.
# On dit que la complexité de la fonction somme_liste est linéaire.
# On appelle n le paramètre de la complexité.
```

1.2 La complexité d'un programme (ou d'une fonction) - Notion de O

Pour trouver la complexité d'un programme ou d'une fonction : 1. Calculer le coût du programme ou de la fonction (Nombre d'opérations élémentaires) ; 2. Identification des paramètres de la complexité ; 3. Conclure en ce qui concerne l'évolution du cout en fonction des paramètres de la complexité.

[ ]:
```python
# Trouvons la complexité de la fonction suivante
def fonction1(a, b):
  for i in range(10):
    print(a + b)

# 1 - Calculons le coût :
# Chaque itération de la boucle for execute 3 opérations élémentaires␣
↪(affectation de i, print, la somme +).
# Chaque itération donc a un coût égal à 3.
# La boucle for execute 10 itérations, sont cout est donc 30 opérations.

# 2 - Identification des paramètres de la complexité :
# Le cout est 30, il ne dépend de rien.
# Dans ce cas, pas de paramètre de la comlplexité.

# 3 - Conclusion :
# La complexité est dite constante - O(1)
```

[ ]:
```python
# Trouvons la complexité de la fonction suivante
def fonction2(k):
  for i in range(k):
    print(i)
```

```
# 1 - Calculons le coût :
# Chaque itération de la boucle for execute 2 opérations élémentaires␣
↪(affectation de i, print).
# Chaque itération donc a un coût égal à 2.
# La boucle for execute k itérations, sont cout est donc 2k opérations.

# 2 - Identification des paramètres de la complexité :
# Le coût de la fonction ne dépend que de la valeur k. k est donc le paramètre␣
↪de la complexité.

# 3 - Conclusion :
# Le cout augmente linéairement en fonction de k
# La complexité est dite linéaire - O(k)
```

[ ]:
```
# Trouvons la complexité de la fonction suivante
def fonction3(k):
  for i in range(k):
    for j in range(k):
      print(i * j)

# 1 - Calculons le coût :
# Chaque itération de la boucle for j execute 3 opérations élémentaires␣
↪(affectation de j, print, *)
# La boucle for j execute k itérations. Le cout de la boucle est 3k.
# Chaque itération de la boucle for i execute une affectation de i et la boucle␣
↪for j d'un cout egal à 3k.
# La boucle for i execute k itérations, son coût est donc (3k + 1) * k = 3k² + k

# 2 - Identification des paramètres de la complexité :
# Le coût de la fonction ne dépend que de la valeur k. k est donc le paramètre␣
↪de la complexité.

# 3 - Conclusion :
# Le cout augmente quadratiquement en fonction de k
# La complexité est dite quadratique - O(k²)
```

[ ]:
```
# Trouvons la complexité de la fonction suivante
def fonction4(k):
  for i in range(k):
    for j in range(k):
      fonction3(k)

# 1 - Calculons le coût :
# Chaque itération de la boucle for j execute une affectation de j et un appel␣
↪de la fonction3
```

```
# Le cout de chaque itération de la boucle for j est donc 3k² + k + 1
# La boucle for j execute k itérations, son cout est donc 3k³ + k² + k
# Chaque itération de la boucle for i execute une affectation de i et la boucle for j d'un cout 3k³ + k² + k
# La boucle for i execute k itérations, son cout est donc 3k  + k³ + k² + k

# 2 - Identification des paramètres de la complexité :
# Le coût de la fonction ne dépend que de la valeur k. k est donc le paramètre de la complexité.


# 3 - Conclusion :
# Le cout augmente d'une façon polynomiale en fonction de k
# La complexité est dite polynomiale - O(k )
```

[ ]:
```python
# Trouvons la complexité de la fonction suivante
def fonction5(m,n):
  for i in range(m):
    for j in range(n):
      print(i * j)

# 1 - Calculons le coût :
# Chaque itération de la boucle for j execute 3 opérations élémentaires (affectation de j, print, *)
# La boucle for j execute n itérations. Le cout de la boucle est 3n.
# Chaque itération de la boucle for i execute une affectation de i et la boucle for j d'un cout egal à 3n.
# La boucle for i execute m itérations, son coût est donc (3n + 1) * m = 3nm + m

# 2 - Identification des paramètres de la complexité :
# Le coût de la fonction dépend de m et de n. m et n sont les paramètres de la complexité.

# 3 - Conclusion :
# Le cout augmente en fonction de m et n, le terme dominant est 3nm
# La complexité est donc O(nm)
```

[ ]:
```python
# Trouvons la complexité de la fonction suivante
def bin(n):
  m = 1
  s = 0
  while n != 0:
    s = s + (n % 2) * m
    m = m * 10
    n = n // 2
  return s
```

```
# 1 - Calculons le coût :
# Chaque itération de la boucle while effectue 9 opérations, tenant compte du
 ↪dernier test le cout de
# la boucle while est 9k + 1 avec k le nombre d'itérations de la boucle while
# En ajoutant l'affectation de m, celle de s et return, le cout de la fonction
 ↪sera 9k + 4
# k = log2(n) + 1 (voir l'image ci dessous)
# Le cout de la fonction est donc 9 * log2(n) + 13

# 2 - Identification des paramètres de la complexité :
# Le coût de la fonction ne dépend que de n. C'est alors le paramètre de la
 ↪complexité.

# 3 - Conclusion :
# Le cout de la fonction augmente logarithmiquement en fonction de n.
# La complexité est dite logarithmique - O(log(n))
```

[ ]:
```
# Trouvons la complexité de la fonction suivante

# La fonction suivante prend en paramètre une liste triée dans l'ordre
 ↪croissant L, et un objet x
# La fonction retourne l'indice de x dans L si x existe, sinon la fonction
 ↪retourne -1.
def recherche_dichotomique(L, x):
  d = 0
  f = len(L) - 1
  while d <= f:
    m = (d + f) // 2
    if L[m] == x:
      return m
    if x > L[m]:
      d = m + 1
    else:
      f = m - 1
  return -1


# 1 - a - Calculons le coût dans le pire des cas : Le cout au pire des cas
 ↪correspond à l'absence de x dans L
# Chaque itération de la boucle while effectue 8 opérations, tenant compte du
 ↪dernier test le cout de
# la boucle while dans le pire des cas est 8k + 1 avec k le nombre d'itérations
 ↪de la boucle while
```

```
# En ajoutant l'affectation de d et celle de f et return, le cout de la␣
 ↪fonction sera 8k + 5
# Posons n la longueur de l'intervalle [d, f] (n = f - d), la boucle while␣
 ↪divise le'intervalle sur 2 jusqu'à ce quelle soit vide (f < d)
# Donc le nombre d'itérations k peut être approximé par log2(n)
# Le cout de la fonction dans le pire des cas est 8 * log2(n) + 5

# 1 - b - Calculons le coût dans le meilleur des cas : Le cout au meilleur des␣
 ↪cas correspond à ll'existence de x dans la position médiane de L
# La boucle while execute une seule itération (jusqu'à return m)
# Le cout de la fonction dans ce cas sera 9

# 2 - Identification des paramètres de la complexité :
# Le coût de la fonction dans le pire des cas ne dépend que de n. C'est alors␣
 ↪le paramètre de la complexité.
# Dans le meilleur des cas le cout de la fonction ne depend de rien

# 3 - Conclusion :
# Le cout de la fonction dans le pire des cas augmente logarithmiquement en␣
 ↪fonction de n.
# La complexité dans le pire des cas est dite logarithmique - O(log(n))
# La complexité dans le meilleur des cas est constante - O(1)
```

1.2.1 Exercice 1

1. Ecrire une fonction egaux(L, M) qui prend en paramètre deux listes de meme tailles L et M et qui retourne True si les deux listes sont égaux et False sinon.
2. Calculer la complexité de la fonction egaux() dans le meilleur et le pire des cas.

[ ]:
```
# 1
def egaux(L, M):
  assert len(M) == len(L), "Les listes doivent avoir la meme dimension"
  for i in range(len(M)):
    if L[i] != M[i]:
      return False
  return True

# 2

# Soit n = len(M)

# Le meilleur des cas : Correspond au cas où la différence entre les deux␣
 ↪listes se trouve dés le premier élément
# Dans ce cas La fonction n'executera qu'une affectation de i et un test de␣
 ↪comparaion. Tenant compte du test assert et de return
# Le cout sera 4
```

```
# La complexité est donc constante - O(1).

# La pire des cas : Correspond au cas où les deu listes sont égaux ou si la␣
↪différence se trouve au niveau du dernier élément
# Dans ce cas :
# Chaque itération de la boucle for n'execute que des opérations constantes,␣
↪chaque itération donc est d'une complexité constante O(1).
# La boucle for effectuera n itérations, sa complexité est donc n * O(1) = O(n).
# En dehors de la boucle, on ne trouve que des opérations élémentaires à␣
↪complexité constante.
# La complexité de la fonction est donc linéaire - O(n)
```

1.2.2 Exercice 2

Calculer la complexité de la focntion application.

Nous avons déja montré que la complexité de la fonction bin(n) est O(log(n)).

[ ]:
```
def bin(n):
  m = 1
  s = 0
  while n != 0:
    s = s + (n % 2) * m
    m = m * 10
    n = n // 2
  return s

def application(n):
  for i in range(n):
    print(bin(n))

# Chaque itération de la boucle for execute le fonction bin de complexité␣
↪O(log(n)) et d'autres opérations de complexité constante (Affectation de i,␣
↪print)
# Chaque itération a une complexité logarithmique O(log(n))
# La boucle for execute n itérations
# Sa complexité est donc n * O(log(n)) = O(nlog(n))
# Il s'agit de la complexité quasilinéaire
```

1.2.3 Exercice 3

Calculer la complexité de la focntion application.

Nous avons déja montré que la complexité de la fonction bin(n) est O(log(n)).

[ ]:
```
def bin(n):
  m = 1
  s = 0
```

```python
  while n != 0:
    s = s + (n % 2) * m
    m = m * 10
    n = n // 2
  return s

def application(m):
  for i in range(1, m + 1):
    print(bin(i))

# Chaque itération de la boucle for execute le fonction bin de complexité
 ↪O(log(i)) et d'autres opérations de complexité constante (Affectation de i,
 ↪print)
# Chaque itération a une complexité  O(log(i))
# La boucle for execute ces opérations pour i allant de 1 à m
# Sa complexité est donc O(log(m!))
```

1.3 La complexité d'une fonction récursive

[ ]:
```python
# Calculons la complexité de la fonction suivante
def fact(n):
  if n == 0:
    return 1
  return n * fact(n - 1)

# Un appel de la fonction fact(n) execute un test, une opération d'écriture
 ↪(return), la multilication, la soustraction et un appel de la fonction
 ↪fact(n - 1)
# C(n) = 4 + C(n - 1)
#      = 4 + 4 + C(n - 2)
#      = 4 + 4 + 4 + C(n - 3)
#      = 4 * (n - 1) + C(1)
#      = 4n + C(0)

# Or C(0) = 2, donc C(n) = 4n + 2

# La complexité est donc linéaire O(n).
```

1.3.1 Règle de la complexité pour les fonctions récursives

- C(n) = C(n-1) + a —> O(n)
- C(n) = a*C(n-1) + b —> O(a**n)
- C(n) = C(n-1) + an + b —> O(n^2)
- C(n) = C(n/2) + b —> O(log(n))
- C(n) = a*C(n/2) + b —> O(n**log(a))
- C(n) = C(n/2) + an + b —> O(n)
- C(n) = 2*C(n/2) + an + b —> O(nlog(n))