COMPLEXITÉ

B. Landelle

Table des matières

I Introduction

1 Problématique

Pour un problème informatique donné, il existe souvent plusieurs algorithmes possibles qui répondent au problème. Il est donc pertinent de pouvoir classer ces algorithmes entre eux afin de choisir le plus performant. Il peut aussi être utile de prévoir le temps d'exécution d'un algorithme, en particulier quand l'argument fourni à l'algorithme est de très grande taille et que la réponse n'est pas immédiate (primalité d'un nombre par exemple).

Pour ce faire, on introduit la notion de *complexité* dont les objectifs principaux sont :
- la comparaison d'algorithmes réalisant le même traitement ;
- la prévision du temps d'exécution d'un algorithme ;
- la prévision de l'utilisation de l'espace mémoire.

La classification obtenue par étude de complexité permettra de choisir l'algorithme nécessitant :
- le moins de ressources de calculs ;
- le moins de ressources de stockage.

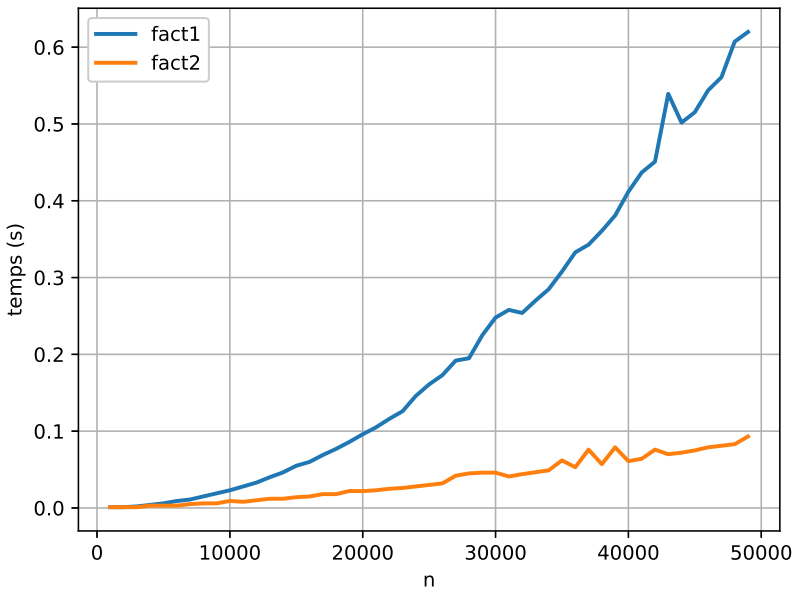

FIGURE 1 – Comparaison de deux implémentations du calcul de $n!$

Exemple : Considérons les fonctions `geom1` et `geom2` qui réalisent le calcul de $\sum_{k=0}^{n-1} q^k$ avec q flottant et n entier sans recours à l'opération d'exponentiation `**`.

```
def geom1(n,q):
    """geom1(n:int,q:int or float)->int or float
    Calcul de 1+q+...+q^(n-1)"""
    res=0
    for k in range(n):
        qk=1
        for i in range(k):
            qk*=q
        res+=qk
    return res
```

Dans la fonction `geom1`, le calcul de q^k est refait intégralement à chaque passage dans la boucle en `k`. Si on compte le nombre total de multiplication, on en effectue

$$\sum_{k=0}^{n-1}\sum_{i=0}^{k-1} 1 = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$$

```
def geom2(n,q):
    """geom2(n:int,q:int or float)->int or float
    Calcul de 1+q+...+q^(n-1)"""
    res=0
    qk=1
    for k in range(n):
        res+=qk
        qk*=q
    return res
```

Dans la fonction `geom2`, on garde en mémoire le terme q^k et on passe de l'étape k à $k+1$ en multipliant par q puisque $q^{k+1} = q^k \times q$. On effectue en tout n additions et multiplications ce qui est largement préférable à la version précédente.

2 Relation de domination O

Pour quantifier les complexités, on utilisera relation de *domination* avec la *notation de Landau* en « grand O ».

Définition 1. *La suite $(u_n)_n$ est dite* dominée *par la suite $(v_n)_n$, relation de* domination *que l'on note $u_n = O(v_n)$ si*

$$\exists \mathrm{N} \in \mathbb{N} \quad \exists \mathrm{M} > 0 \qquad \forall n \geqslant \mathrm{N} \qquad |u_n| \leqslant \mathrm{M}\,|v_n|$$

Remarque : La bonne notion pour l'étude de complexité n'est pas en réalité la relation de domination. Effet, si un algorithme a une complexité temporelle en $\mathrm{O}(n)$, on peut tout aussi bien annoncer qu'elle est en $\mathrm{O}(n^2)$. C'est correct même si c'est nettement moins pertinent. L'usage consiste donc à déterminer la domination la plus fine qui soit ce qui équivaut à utiliser la relation hors-programme *de l'ordre de* notée Θ. On note $u_n = \Theta(v_n)$ ce qui signifie $u_n = \mathrm{O}(v_n)$ et $v_n = \mathrm{O}(u_n)$.

Proposition 1. *La notation $O(1)$ désigne une suite réelle bornée. Étant donnée une suite réelle $(v_n)_n$ qui ne s'annule pas, on a*

$$u_n = O(v_n) \iff u_n = v_n O(1)$$

Remarques : (1) La notation de Landau est générique. Par exemple

$$n = \mathrm{O}(n) \quad \text{et} \quad 2n = \mathrm{O}(n)$$

mais on n'a évidemment pas $n = 2n$ pour n non nul. La notation désigne un certain comportement asymptotique, c'est-à-dire pour $n \to +\infty$.

(2) On omet délibérément la dépendance en n dans la notation $\mathrm{O}(1)$ (on ne note pas $(\mathrm{O}(1)_n)_{n\in\mathbb{N}}$).

Rappel : Une suite réelle convergente est bornée. En effet, soit $(u_n)_n$ suite convergente de limite ℓ. Il existe un entier N tel que pour $n > \mathrm{N}$, on ait $|u_n - \ell| \leqslant 1$ d'où par inégalité triangulaire

$$\forall n > \mathrm{N} \qquad |u_n| \leqslant |u_n - \ell| + |\ell| \leqslant 1 + |\ell|$$

Prenant $\mathrm{M} = \max(|u_0|, \ldots, |u_{\mathrm{N}}|, 1 + |\ell|)$, on a bien

$$\forall n \in \mathbb{N} \qquad |u_n| \leqslant \mathrm{M}$$

Notation : Dans tout ce qui suit, on notera log le logarithme népérien conformément à l'usage en cours d'informatique.

Proposition 2. *On a les propriétés suivantes :*

1. $\mathrm{C}^{te} \times O(u_n) = O(u_n)$;
2. *Si* $u_n \to +\infty$, *alors* $\mathrm{C}^{te} + O(u_n) = O(u_n)$.
3. *Si* $x_n = O(y_n)$ *et* $y_n = O(u_n)$, *alors* $x_n = O(u_n)$ *(transitivité).*
4. *Si* $x_n = O(u_n)$ *et* $y_n = O(u_n)$, *alors,* $x_n + y_n = O(u_n)$.
5. *Si* $x_n = O(u_n)$ *et* $y_n = O(v_n)$, *alors* $x_n \times y_n = O(u_n \times v_n)$.

Exemples : 1. On a $\log(n) = \mathrm{O}(n)$ puisque d'après le théorème des croissances comparées

$$\frac{\log(n)}{n} \xrightarrow[n\to+\infty]{} 0$$

2. On a $\displaystyle\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \mathrm{O}(n^2)$ car

$$\frac{n(n+1)}{2n^2} = \frac{1}{2}\left[1 + \frac{1}{n}\right] \xrightarrow[n\to+\infty]{} \frac{1}{2}$$

3. On a $\mathrm{O}(1) + \mathrm{O}(\log(n)) = \mathrm{O}(\log(n))$ puisque $\mathrm{O}(1) = \mathrm{O}(\log(n))$ car

$$\frac{\mathrm{O}(1)}{\log(n)} \xrightarrow[n\to\infty]{} 0$$

Le résultat suit par addition de grands O. On ne peut dire mieux (prendre $0 + \log(n)$ par exemple).

4. On a $\mathrm{O}(n) + \mathrm{O}(n^2) = \mathrm{O}(n^2)$ puisque $\mathrm{O}(n) = \mathrm{O}(n^2)$ car

$$\frac{\mathrm{O}(n)}{n^2} = \frac{n\mathrm{O}(1)}{n^2} = \frac{\mathrm{O}(1)}{n} \xrightarrow[n\to\infty]{} 0$$

Le résultat suit par addition de grands O. On ne peut dire mieux (prendre $0 + n^2$ par exemple).

Remarques : (1) L'égalité $\mathrm{O}(1) = \mathrm{O}(\log(n))$ est licite : elle signifie que $\mathrm{O}(1)$ est *dominée par* $\log(n)$ puisque $\dfrac{\mathrm{O}(1)}{\log(n)} \xrightarrow[n\to\infty]{} 0$.

⚠ En revanche, si on inverse les rôles, l'égalité $\mathrm{O}(\log(n)) = \mathrm{O}(1)$ est fausse ($\log(n) \xrightarrow[n\to\infty]{} +\infty$ donc non borné). L'égalité avec un O s'entend donc au sens de la définition 1 et non au sens usuel d'une égalité.

(2) Quand on somme des grand O, le « plus gros » terme « digère » les autres.

(3) Si on regarde un grand O d'une expression, seul le « plus gros » terme de cette expression est pertinent.

Exercice : Que peut-on dire des expressions suivantes :

1. $\mathrm{O}(1) + \mathrm{O}(n)$
2. $\mathrm{O}(\log(n)) + \mathrm{O}(n)$
3. $\mathrm{O}(n) + \mathrm{O}(n\log(n))$
4. $\mathrm{O}\left(\sum_{k=1}^{n-1} k\right)$

Corrigé : 1. On a $\mathrm{O}(1) + \mathrm{O}(n) = \mathrm{O}(n)$ et on ne peut dire mieux (prendre $0 + n$ par exemple).
2. On a $\mathrm{O}(\log(n)) + \mathrm{O}(n) = \mathrm{O}(n)$ et on ne peut dire mieux (prendre $0 + n$ par exemple).
3. On a $\mathrm{O}(n) + \mathrm{O}(n\log(n)) = \mathrm{O}(n\log(n))$ et on ne peut dire mieux (prendre $0 + n\log(n)$ par exemple).
4. On a $\mathrm{O}\left(\sum_{k=1}^{n-1} k\right) = \mathrm{O}\left(\dfrac{n^2-n}{2}\right) = \mathrm{O}(n^2)$ et on ne peut dire mieux.

3 Sommation des O

Théorème 1. *Soit $(u_n)_n$ une suite de réels strictement positifs. On a*

$$\sum_{k=1}^{n} O(u_k) = O\left(\sum_{k=1}^{n} u_k\right)$$

Corollaire 1. $\displaystyle\sum_{k=1}^{n} O(1) = O(n) \qquad \sum_{k=1}^{n} O(k) = O(n^2) \qquad \sum_{k=2}^{n} O(\log(k)) = O(n\log(n))$

Exercice : Parmi les quantités suivantes, déterminer celles qui sont en $\mathrm{O}(1)$, $\mathrm{O}(\log(n))$, $\mathrm{O}(n)$, $\mathrm{O}(n\log(n))$, $\mathrm{O}(n^2)$:

1. $\displaystyle\sum_{i=1}^{p} \mathrm{O}(1)$ avec p fixé ;
2. $\displaystyle\sum_{k=1}^{n-1} \mathrm{O}(k)$;
3. $\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n} \mathrm{O}(1)$;
4. $\displaystyle\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{O}(1)$;

5. $\sum_{k=1}^{n} [\mathrm{O}(k) + \mathrm{O}(\log(k))]$;
6. $\sum_{i=1}^{n}\sum_{j=1}^{i} \mathrm{O}(1)$;
7. $\sum_{k\in\mathbb{N} \mid 2^k \leqslant n} \mathrm{O}(1)$.

Corrigé : 1. Une somme finie de suites bornées est une suite bornée d'où $\sum_{i=1}^{p} \mathrm{O}(1) = \mathrm{O}(1)$.

2. On a $\sum_{k=1}^{n-1} \mathrm{O}(k) = \mathrm{O}\left(\sum_{k=1}^{n-1} k\right) = \mathrm{O}(n^2)$.
3. On a $\sum_{i=1}^{n}\sum_{j=1}^{n} \mathrm{O}(1) = \mathrm{O}\left(\sum_{i=1}^{n}\sum_{j=1}^{n} 1\right) = \mathrm{O}(n^2)$.
4. On a $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{O}(1) = \mathrm{O}\left(\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1\right) = \mathrm{O}\left(\sum_{i=1}^{n-1}(n-i)\right) = \mathrm{O}\left(\sum_{k=1}^{n-1} k\right) = \mathrm{O}(n^2)$.
5. On a $\sum_{k=1}^{n} [\mathrm{O}(k) + \mathrm{O}(\log k)] = \sum_{k=1}^{n} \mathrm{O}(k) = \mathrm{O}\left(\sum_{k=1}^{n} k\right) = \mathrm{O}(n^2)$.
6. On a $\sum_{i=1}^{n}\sum_{j=1}^{i} \mathrm{O}(1) = \mathrm{O}\left(\sum_{i=1}^{n}\sum_{j=1}^{i} 1\right) = \mathrm{O}\left(\sum_{i=1}^{n} i\right) = \mathrm{O}(n^2)$.
7. On a $\sum_{k\in\mathbb{N} \mid 2^k \leqslant n} \mathrm{O}(1) = \sum_{0\leqslant k\leqslant \log_2(n)} \mathrm{O}(1) = \mathrm{O}\left(\sum_{k=0}^{\lfloor \log_2(n)\rfloor} 1\right) = \mathrm{O}(\log(n))$.

II Complexité

1 Types de complexité

Définition 2. *On appelle* opération élémentaire *une opération que l'ordinateur peut effectuer en temps constant.*

Par exemple, les opérations implémentées bas-niveau sur les processeurs comme les opérations logiques ou arithmétiques sur des registres sont des opérations élémentaires.

Définition 3. *La* complexité temporelle *d'un algorithme désigne le nombre d'opérations élémentaires réalisées par l'algorithme.*

En pratique, on ne s'intéressera pas au coût exact d'un algorithme mais plutôt à une classe de complexité décrite avec la notation de Landau.

Définition 4. *La* complexité spatiale *d'un algorithme désigne l'espace mémoire occupé lors de l'exécution de l'algorithme.*

Remarque : Ce critère est moins considéré du fait des très importantes capacités mémoires des machines actuelles. Une des situations où ce critère reste pertinent est celui des fonctions récursives puisque les multiples appels de la fonction par elle-même nécessitent une utilisation importante de mémoire.

⚠ Dans un énoncé, quand il est fait mention d'un calcul de « complexité » sans précision additionnelle, il faut comprendre complexité temporelle.

Proposition 3. *La complexité spatiale d'un algorithme est majorée par sa complexité temporelle puisque chaque occupation additionnelle en mémoire implique une instruction d'écriture en mémoire.*

Définition 5. *La complexité temporelle* dans le pire des cas *est le temps d'exécution maximum de l'algorithme, à savoir le temps d'exécution dans le cas le plus défavorable. La complexité spatiale* dans le pire des cas *désigne l'espace mémoire maximum occupé par l'algorithme, à savoir l'espace occupé dans le cas le plus défavorable.*

Définition 6. *La complexité temporelle* dans le meilleur des cas *est le temps d'exécution minimum de l'algorithme, à savoir le temps d'exécution dans le cas le plus favorable. La complexité spatiale* dans le meilleur des cas *est l'espace mémoire minimum occupé par l'algorithme, à savoir l'espace occupé dans le cas le plus favorable.*

Remarques : (1) Ces complexités dans le pire et dans le meilleur des cas correspondent respectivement à des bornes supérieures et inférieures des temps d'exécution ou d'occupation en mémoire de l'algorithme.
(2) On évoque les notions de pire cas et meilleur cas quand cela est pertinent : il peut tout à fait ne pas exister de pire et meilleur cas mais si cette distinction existe, il faut la faire.

2 Classes de complexité

Plutôt que de compter exactement le nombre d'opérations élémentaires d'un algorithme, on cherche à classer sa complexité parmi des ordres de grandeur de référence. Cette classification est simplificatrice tout en gardant la pertinence du coût d'un algorithme.

L'entier n désigne en général la taille de l'argument. Dans le calcul d'un algorithme portant sur un calcul arithmétique, il peut aussi désigner l'argument lui-même.

Définition 7. *On distingue les principales classes de complexité suivantes :*
- $O(1)$ *: complexité constante ;*
- $O(\log(n))$ *: complexité logarithmique ;*
- $O(\sqrt{n})$ *: complexité racinaire ;*
- $O(n)$ *: complexité linéaire ;*
- $O(n\log(n))$ *: complexité quasi-linéaire ;*
- $O(n^2)$ *: complexité quadratique ;*
- $O(n^p)$ *: complexité polynomiale ;*
- $O(a^n)$ *: complexité exponentielle* $(a > 1)$.

Les complexités au plus quasi-linéaires sont raisonnables. Les complexités polynomiales, au moins quadratiques, sont acceptables mais les complexités exponentielles sont à proscrire.

3 Coût des instructions

Définition 8. *Les instructions suivantes sont considérées comme* opérations élémentaires *:*
- *affectation de types simples ;*
- *comparaison de types simples ;*
- *opérations arithmétiques et logiques* `+,-,*,/, //, %, and, or, not`

Les opérations élémentaires sont les unités de mesure du coût d'un algorithme.

Remarque : Les affectations et comparaisons concernent les types simples.

Cette classification d'opérations élémentaires est simpliste bien que nécessaire pour une première approche. Par exemple, pour des flottants ou de « petits » entiers ($\leqslant 2^{30}$), la multiplication et l'addition peuvent effectivement être vues comme des opérations à coûts constants.

En revanche, si on s'intéresse à de grands entiers, la multiplication est plus coûteuse que l'addition. L'algorithme naïf appris dans les petites classes confirme intuitivement cette idée. La réalité des implémentations est plus subtile.

Depuis les années 60, de nouvelles générations d'algorithmes de multiplication rapide ont supplanté la méthode naïve :

- algorithme de Karatsuba (1960) ;
- algorithme de Toom-Cook (1963) ;
- algorithme de Schönhage-Strassen (1971) ;
- algorithme de Fürer (2007) ;
- algorithme de Harvey-van der Hoeven (2019).

Pour des opérations sur des types composés (liste, chaîne de caractère, tuple), le coût peut être fonction de la taille de l'argument. Considérons par exemple la situation suivante :

```
def f(n):
    res=[0]*n       # construit une liste de n zéros
    ...
```

On initialise la variable `res` en construisant une liste de n zéros où n est l'argument de la fonction. Cette affectation équivaut, en coût temporel et spatial, à n affectations de types simples d'où une complexité temporelle et spatiale en $\mathrm{O}(n)$.

Proposition 4. *Les temps d'accès à un élément d'une liste en lecture/écriture sont en $O(1)$.*

Cette caractéristique combinée à une structure dynamique est un des atouts majeurs des listes en python.

Complexité temporelle des instructions ou méthodes sur des listes de taille n :

Opération	Complexité
`append`	$\mathrm{O}(1)^*$
`pop`	$\mathrm{O}(1)^*$
`==, !=`	$\mathrm{O}(n)$
`in`	$\mathrm{O}(n)$
`remove`	$\mathrm{O}(n)$
`delete`	$\mathrm{O}(n)$
`count`	$\mathrm{O}(n)$
`max, min`	$\mathrm{O}(n)$
`reverse`	$\mathrm{O}(n)$
`sort`	$\mathrm{O}(n\log(n))$

(*) : il s'agit de complexité amortie (coût moyen des opérations en utilisation).

La taille de l'écriture binaire d'un entier n non nul est en $O(\log(n))$ ce qu'on peut observer expérimentalement avec l'instruction `getsizeof` du module `sys`.

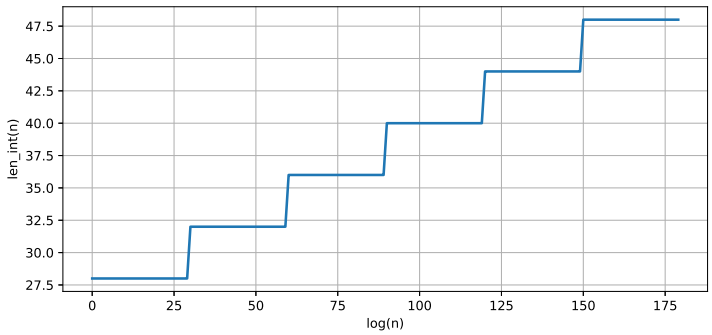

FIGURE 2 – Mémoire allouée à la représentation d'un entier

Jusqu'à 2^{30}, le coût d'écriture d'un entier est constant. Dans cette configuration, on considère que la complexité spatiale liée à la représentation des entiers est en $O(1)$. Sauf mention particulière, c'est une des hypothèses de travail qui est faite habituellement.

Avec cette hypothèse simplificatrice, la complexité spatiale de l'instruction `range(n)` est en $O(1)$ tandis que celle de `list(range(n))` est en $O(n)$.

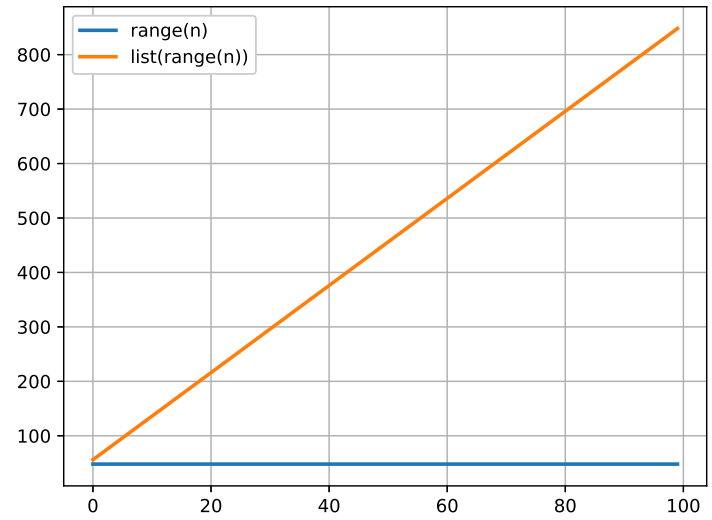

FIGURE 3 – Complexité spatiale : `range(n)` *versus* `list(range(n))`

Proposition 5. *Pour les dictionnaires, les tests d'appartenance et les temps d'accès en lecture/écriture à un couple (clé,valeur) sont en $O(1)$.*

Les dictionnaires en python sont une implémentation d'une structure abstraite de données appelée *table de hachage*, structure qui permet une association performante clé-valeur.

III Calculs de complexité

1 Bases de calculs

On rappelle que l'entier n désigne la taille de l'argument ou l'argument lui-même.

• Séquence simple

Dans tout ce qui suit, les instructions repérées par `Instruction x`, `Test y` sont de complexité temporelle en O(1). On considère que seules les instructions en fin de bloc peuvent provoquer des sorties (`break` ou `return`). Les entiers `p`, `r`, `p1`, ...`pr` désignent des entiers fixés indépendants de l'entier n.

La séquence d'instructions

```
Instruction 1
Instruction 2
...
Instruction p
```

avec p constant, indépendant de n, est de complexité temporelle en

$$\sum_{i=1}^{p} \mathrm{O}(1) = \mathrm{O}(1)$$

La séquence d'instructions

```
if Test:
    Instruction 1
    ...
    Instruction p
```

avec p constant, indépendant de n, est de complexité temporelle en

$$\mathrm{O}(1) \quad \text{ou} \quad \mathrm{O}(1) + \sum_{i=1}^{p} \mathrm{O}(1) = \mathrm{O}(1)$$

La séquence d'instructions

```
if Test:
    Instruction 1
    ...
    Instruction p1
else:
    Instruction p1+1
    ...
    Instruction p1+p2
```

avec les p_i constants, indépendants de n, est de complexité temporelle en

$$\mathrm{O}(1) + \sum_{i=1}^{p_1} \mathrm{O}(1) = \mathrm{O}(1) \quad \text{ou} \quad \mathrm{O}(1) + \sum_{i=1}^{p_2} \mathrm{O}(1) = \mathrm{O}(1)$$

La séquence d'instructions

```
if Test 1:
    Instruction 1
    ...
    Instruction p1
elif Test 2:
    Instruction p1+1
    ...
    Instruction p1+p2
...
elif Test r:
    ...
    Instruction p1+...+pr
```

avec les p_i constants, indépendants de n, est de complexité temporelle en

$$\mathrm{O}(1)+\sum_{i=1}^{p_1}\mathrm{O}(1)=\mathrm{O}(1) \quad \text{ou} \quad \mathrm{O}(1)+\mathrm{O}(1)+\sum_{i=1}^{p_2}\mathrm{O}(1)=\mathrm{O}(1)$$

$$\text{ou} \quad \dots \quad \text{ou} \quad \sum_{i=1}^{r}\mathrm{O}(1)+\sum_{i=1}^{p_r}\mathrm{O}(1)=\mathrm{O}(1)$$

• **Séquence avec boucle**

On considère dans un premier temps que les instruction repérées par `Instruction x` ne provoquent pas de sortie (ni `break`, ni `return`).

La séquence d'instructions

```
for k in range(n):
    Instruction 1
    ...
    Instruction p
```

est de complexité temporelle en

$$\sum_{k=0}^{n-1}\mathrm{O}(1)=\mathrm{O}\left(\sum_{k=0}^{n-1}1\right)=\mathrm{O}(n)$$

La séquence d'instructions

```
for x in L:
    Instruction 1
    ...
    Instruction p
```

avec `L` une liste de taille n est de complexité temporelle en

$$\sum_{i=1}^{n}\mathrm{O}(1)=\mathrm{O}\left(\sum_{i=1}^{n}1\right)=\mathrm{O}(n)$$

On peut provoquer une sortie de boucle prématurée avec un `return` ou un `break` dans une boucle `for`. Une sortie de boucle doit être conditionnelle sans quoi la boucle n'a plus de raison d'être. Dans ce cas, on peut être amené à distinguer un meilleur cas et un pire cas.

```
def fonction(L):
    for x in L:
        if Test:
            return ...
    return ...
```

Si `Test` est `True` au premier passage dans la boucle, on a une complexité temporelle dans le meilleur des cas en $\mathrm{O}(1)$. En revanche, si `Test` est `False` tout au long de la boucle, on a une complexité temporelle dans le pire des cas en $\mathrm{O}(n)$.

On considère que les instruction repérées par `Instruction x` ne provoquent pas de sortie (ni `break`, ni `return`).

Notant n le nombre de fois où la valeur de `Test` est `True`, la séquence d'instructions

```
while Test:
    Instruction 1
    ...
    Instruction p
```

est de complexité temporelle en

$$\sum_{i=1}^{n}\mathrm{O}(1) = \mathrm{O}\left(\sum_{i=1}^{n}1\right) = \mathrm{O}(n)$$

Remarque : Il n'y a pas lieu de considérer le cas d'une sortie prématurée d'une boucle `while` car c'est l'étude de la condition `Test` qui amène à distinguer d'éventuels pire cas/meilleur cas.

• Séquence avec boucles imbriquées

Une boucle peut contenir une boucle ou une instruction non élémentaire. Dans ce cas les complexités s'additionnent.

La séquence d'instructions

```
for i in range(n):
    for j in range(n):
        Instruction 1
        ...
        Instruction p
```

est de complexité temporelle en

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\mathrm{O}(1) = \mathrm{O}\left(\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}1\right) = \mathrm{O}(n^2)$$

De même, la séquence d'instructions

```
for i in range(n):
    for j in range(i):
```

```
        Instruction 1
        ...
        Instruction p
```

est de complexité temporelle en

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}\mathrm{O}(1) = \mathrm{O}\left(\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}1\right) = \mathrm{O}(n^2)$$

Remarquons que la somme triangulaire qui apparaît vaut

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}1 = \sum_{i=0}^{n-1} i \underset{n\to+\infty}{\sim} \frac{n^2}{2}$$

mais le facteur $\dfrac{1}{2}$ est « digéré » par le symbole O.

Exceptée la situation de la programmation récursive, le nombre de variables utilisées est fixe. Un programme qui utilise un nombre fixe de variables de tailles fixées à une complexité spatiale en O(1). On fait en général l'hypothèse simplificatrice que le stockage d'un entier est à coût constant. C'est vrai dans une certaine mesure, pour des entiers d'un ordre $\leqslant 2^{30}$ (au delà, le coût en mémoire est en $\mathrm{O}(\log(n))$).

Pour la complexité spatiale, on s'intéresse plus précisément à la taille des variables de type chaîne ou liste créées. Un programme qui utilise une ou plusieurs variables dont la taille croît a pour complexité spatiale un grand O de la somme de ses tailles. Enfin, on ne compte pas l'argument dans la complexité spatiale d'une fonction : celui-ci est un apport extérieur à la fonction, il n'est pas considéré comme une ressource de mémoire requise par la fonction. En revanche, si on duplique l'argument dans une variable locale, alors celui-ci doit être pris en compte dans le calcul.

Exemples : 1. Calcul de $\binom{n}{k}$ avec k, n entiers et $k \in [\![0\,;\, n]\!]$.

```
def binom(n,k):
    """binom(n:int,k:int)->int
    Renvoie le nombre de combinaisons de k parmi n"""
    res=1
    for i in range(k):
        res=res*(n-i)//(i+1)
    return res
```

La complexité temporelle est en $\mathrm{O}(k)$. La fonction utilise un nombre fixe de variables de tailles fixées d'où une complexité spatiale en O(1).

2. Retournement d'une liste sans slicing.

```
def renv_list(L):
    """renv_list(L:list)->list
    Renvoie une nouvelle liste qui est le retournement de la liste L"""
    n=len(L)
    res=[]
```

```
    for k in range(n-1,-1,-1):
        res.append(L[k])
    return res
```

La complexité temporelle de `renv_list` est en

$$\sum_{k=0}^{n-1} \mathrm{O}(1) = \mathrm{O}(n)$$

La variable `res` contient la liste obtenue par retournement de la liste `L`. Les autres variables `n` et `k` sont de taille fixée. Par conséquent, la complexité spatiale de `renv_list` est en $\mathrm{O}(n)$.

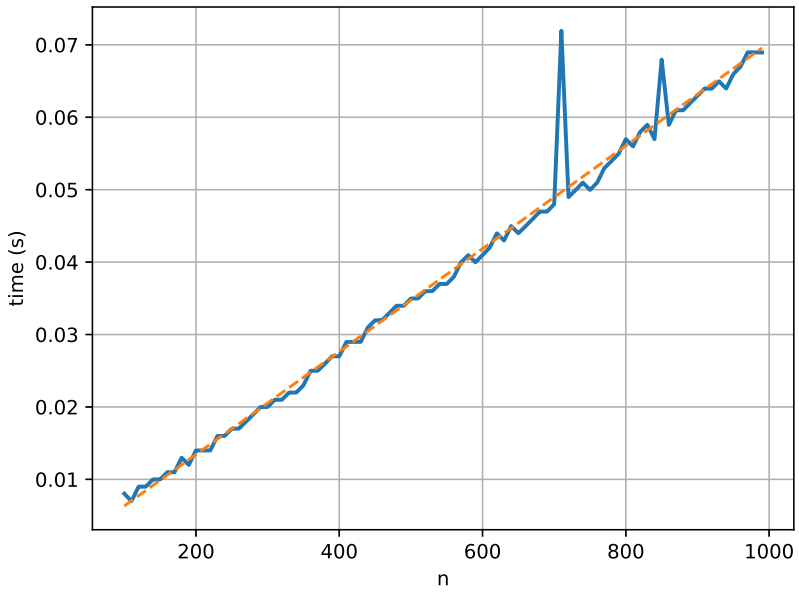

FIGURE 4 – Complexité linéaire de `renv_list`

On observe bien la tendance linéaire de la complexité temporelle de `renv_list`. Une partie des pics dans le tracé est vraisemblablement imputable au comportement de `append` : celle-ci est en moyenne en $\mathrm{O}(1)$ mais de temps à autre, il y a une allocation mémoire additionnelle à effectuer qui coûte plus en temps de traitement d'où un pic lors de cet événement.

3. Retournement d'une chaîne sans slicing.

```
def renv_string(S):
    """renv_string(S:string)->string
    Renvoie une chaîne qui est le retournement de la chaîne S"""
    n=len(S)
    res=""
    for k in range(n):
        res=S[k]+res
    return res
```

Lors du i-ième passage, dans la boucle `for` (avec $i = k+1$, le premier passage correspondant à $k = 0$), on a concaténé $i-1$ caractères dans `res` ce qui signifie que la chaîne est de taille $i-1$

et la concaténation avec un nouveau caractère induit la création d'une nouvelle chaîne de taille i d'où un coût en $\mathrm{O}(i)$. Ainsi, la complexité temporelle est en

$$\sum_{i=1}^{n}\mathrm{O}(i) = \mathrm{O}\left(\sum_{i=1}^{n} i\right) = \mathrm{O}(n^2)$$

La variable `res` contient une chaîne obtenue par retournement de la chaîne `S`. Les autres variables `n` et `k` sont de taille fixée. Par conséquent, la complexité spatiale de `renv_string` est en $\mathrm{O}(n)$.

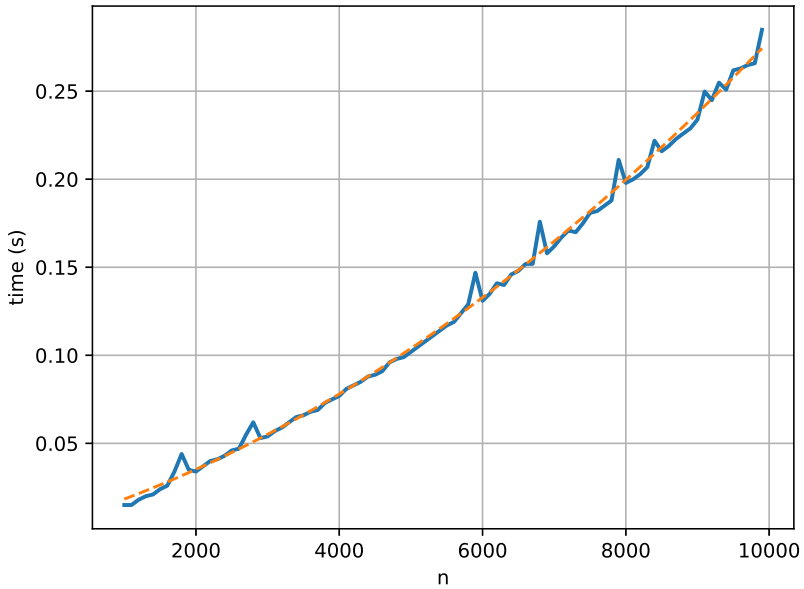

FIGURE 5 – Complexité quadratique de `renv_string`

4. Écriture binaire d'un entier.

```
def binaire(n):
    """binaire(n:int)->list
    Renvoie la liste de l'écriture binaire de n"""
    res=[]
    a=n
    while a>0:
        res.append(a%2)
        a//=2
    return res
```

Soit n entier non nul d'écriture binaire $n = \langle d_{p-1}, \ldots, d_0\rangle$. La variable `a` reçoit n comme valeur initiale puis, à chaque passage dans la boucle `while`, est quotientée par deux. Il s'ensuit que la taille de l'écriture binaire de `a` décroît de un à chaque passage dans la boucle. Comme la taille initiale de `a` est $\lfloor \log_2(n) + 1 \rfloor$, on en déduit une complexité temporelle en $\mathrm{O}(\log(n))$. La variable `res` reçoit, sous forme de liste, l'écriture binaire de taille $\lfloor \log_2(n) + 1 \rfloor$ et la variable `a` reçoit `n` en valeur initiale dont le stockage en mémoire est celui de son écriture binaire (on ne le néglige pas ici puisqu'il est du même ordre que celui de la variable `res`). On en déduit une complexité spatiale en $\mathrm{O}(\log(n))$.

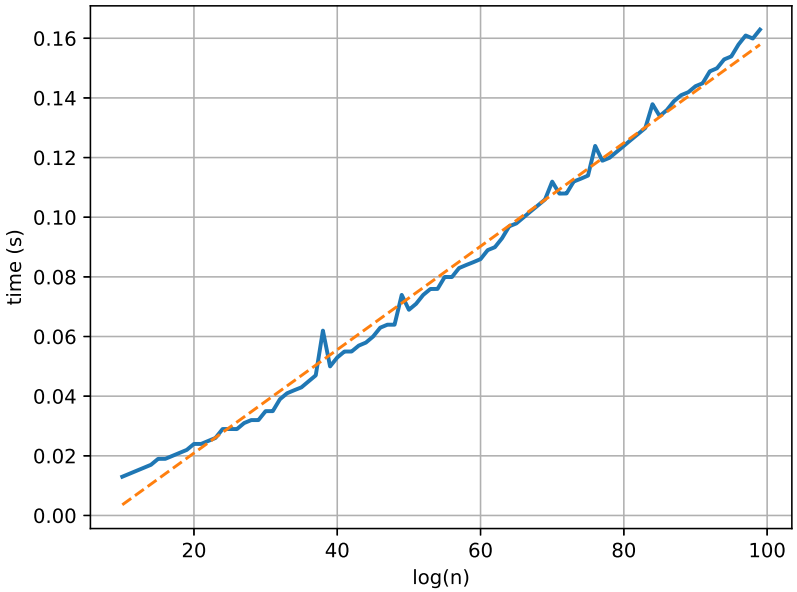

FIGURE 6 – Complexité de `binaire`

L'expérimentation ne permet d'observer le comportement annoncé. Le tracé logarithmique du temps d'exécution de `binaire` pour n en fonction de $\log(n)$ peut sembler linéaire au début mais pas vraiment pour la courbe dans son intégralité. Comme il s'agit d'observer un comportement asymptotique pour $n \to +\infty$, on expérimente avec de grandes valeurs de n ce qui met en défaut les hypothèses faites sur les opérations arithmétiques : on ne pas considérer raisonnablement que celles-ci soient en $O(1)$ sur de grandes valeurs.

Exercice : Soit x réel et $P = \sum_{k=0}^{n-1} a_k X^k \in \mathbb{R}[X]$. L'algorithme de Horner consiste à calculer efficacement $P(x)$ en observant

$$P(x) = \sum_{k=0}^{n-1} a_k x^k = (\ldots(((0 \times x + a_{n-1})\, x + a_{n-2})\, x + a_{n-3})\, x + \ldots)\, x + a_0$$

d'où l'implémentation :

```
def poly(x,P):
    """Calcul de P(x) suivant l'algorithme de Horner
    x : flottant
    P : [a_0, ..., a_{n-1}] liste de flottants"""
    res=0
    n=len(P)
    for k in range(n-1,-1,-1):
        res=x*res+P[k]
    return res
```

Déterminer la complexité temporelle et spatiale de la fonction `poly`.

Corrigé : On effectue n passages dans la boucle avec, lors de chaque passage une multiplication et une addition qu'on suppose à coût constant d'où une complexité temporelle en

$$\sum_{k=0}^{n-1} \mathrm{O}(1) = \mathrm{O}(n)$$

La fonction `poly` utilise un nombre fixe de variables de taille fixée d'où une complexité spatiale en O(1).

Exercice : Pour n entier non nul, il existe un unique couple d'entiers (α, b) tel que

$$n = 2^\alpha(2b+1)$$

La quantité α s'appelle la 2-valuation de n et désigne la plus grande puissance de 2 factorisable dans n.

```
def val2(n):
    """val2(x:int)->int
    Renvoie la 2-valuation de x"""
    a=n
    k=0
    while a%2==0:
       k+=1
       a//=2
    return k
```

Déterminer la complexité temporelle et spatiale de `val2`.

Corrigé : Si l'entier n est impair, on ne rentre pas dans la boucle. Au contraire, si n est une puissance 2 *i.e.* $n = 2^\alpha$, on rentre $\alpha = \log_2(n)$ fois dans la boucle. On en déduit que la complexité temporelle est en O(1) dans le meilleur des cas et en $\mathrm{O}(\log(n))$ dans le pire des cas. La fonction `val2` utilise un nombre fixe de variables de taille fixée d'où une complexité spatiale en O(1).

2 Algorithmes classiques

• Test d'appartenance

La fonction `detect(elt,L)` d'argument `elt` un objet et `L` une liste de taille n renvoie `True` si `elt` est présent dans `L` et `False` sinon.

```
def detect(elt,L):
    """detect(elt:any,L:list)->bool
    Renvoie test de présence de elt dans L"""
    for x in L:
       if elt==x:
           return True
    return False
```

Si `elt` est présent en première position de la liste `L`, le `return` provoque une sortie de boucle prématurée après le premier passage. Si `elt` est absent de `L`, la boucle est effectuée intégralement. On en déduit une complexité temporelle en O(1) dans le meilleur des cas et en $\mathrm{O}(n)$ dans le pire des cas. La fonction utilise un nombre fixe de variables de taille fixée d'où une

complexité spatiale en O(1).

• Liste d'occurrences

La fonction `pos(elt,L)` d'arguments `elt` un objet et `L` une liste de taille n renvoie une liste des indices de `elt` dans `L`. Si `elt` est absent de `L`, la fonction renvoie la liste vide.

```
def pos(elt,L):
    """pos(elt:any,L:list)->list
    Renvoie la liste des indices des occurrences de elt dans L"""
    res=[]
    n=len(L)
    for k in range(n):
        if L[k]==elt:
            res.append(k)
    return res
```

La boucle est parcourue intégralement et quel que soit le résultat du test, les instructions dans la boucle sont en O(1) puisque la méthode `append` est à coût constant. On en déduit une complexité temporelle en O(n). Les variables `n` et `k` sont de taille fixées (variables à valeurs entières). Si `elt` est absent de la liste `L`, la variable `res` demeure une liste vide. En revanche, si la liste `L` est une répétition de `elt`, alors `res` reçoit la liste de toutes les positions de 0 à $n-1$. On en déduit une complexité spatiale en O(1) dans le meilleur des cas et en O(n) dans le pire des cas.

• Recherche dichotomique

On rappelle le principe de la recherche dichotomique d'un objet `elt` dans une liste triée `L` de taille n :

- on considère l'élément au milieu de `L` ;
- si c'est `elt`, on s'arrête ;
- si `elt` est plus petit que l'élément du milieu, on se place sur la moitié de gauche, sinon on se place sur la moitié de droite ;
- on poursuit ce processus tant qu'on n'a pas rencontré `elt` et que la zone de recherche n'est pas vide.

```
def rech_dicho(elt,L):
    """rech_dicho(elt:int,L:list)->(bool,int)
    Renvoie le résultat de la recherche dichotomique
    de elt dans L liste triée :
    * si L[k]==elt       -> (True,k)
    * si elt absent de L -> (False,k)"""
    deb=0
    fin=len(L)-1
    trouve=False
    while not trouve and deb<=fin:
        milieu=(deb+fin)//2
        if L[milieu]==elt:
            trouve=True
        elif L[milieu]>elt:
```

```
            fin=milieu-1
        else:
            deb=milieu+1
    return trouve,milieu
```

Si l'élément est présent au milieu de la liste, on ne rentre qu'une fois dans la boucle. Sinon, lors de chaque passage dans la boucle, la zone de recherche est au moins divisée par deux. L'écriture binaire de la taille de la zone de recherche est donc au moins décrémentée de un à chaque itération. Comme la taille initiale de la zone de recherche est n dont l'écriture binaire est de taille $\lfloor \log_2(n)+1 \rfloor$, on en déduit une complexité temporelle en $O(1)$ dans le meilleur des cas et en $O(\log(n))$ dans le pire des cas. La fonction utilise un nombre fixe de variables de taille fixée d'où une complexité spatiale en $O(1)$.

• Exponentiation rapide

Soit x un réel et n un entier non nul d'écriture binaire $n = \langle d_{p-1}, \dots, d_0 \rangle$. On a

$$x^n = x^{\left(\sum_{i=0}^{p-1} d_i 2^i\right)} = \prod_{i=0}^{p-1} \left(x^{d_i 2^i}\right) = \prod_{i=0}^{p-1} \left(x^{2^i}\right)^{d_i}$$

Dans le produit, à i fixé dans $[\![0\,;\, p-2]\!]$, on passe du terme x^{2^i} au suivant $x^{2^{i+1}}$ en élevant au carré :

$$x^{2^{i+1}} = x^{2\times 2^i} = \left(x^{2^i}\right)^2$$

La contribution de x^{2^i} dans le produit est déterminée par la valeur de d_i : si $d_i = 0$, le terme n'apparaît pas dans le produit et sinon il apparaît. Cette écriture permet d'envisager un algorithme performant pour le calcul de x^n, algorithme dit d'*exponentiation rapide* :

```
def expo(x,n):
    """expo(x:int or float,n:int)->int or float
    Calcul de x**n par exponentiation rapide"""
    a,r,e=x,1,n
    while e>0:
        if e%2==1:
            r*=a
        a*=a
        e//=2
    return r
```

La variable `e` reçoit `n` comme valeur initiale dont l'écriture binaire est de taille $\lfloor \log_2(n)+1 \rfloor$ et est quotientée par deux à chaque passage dans la boucle. La taille de son écriture binaire décroît donc de un à chaque passage. Comme les opérations arithmétiques réalisées à l'intérieur de la boucle sont à coût constant, on en déduit une complexité temporelle en $O(\log(n))$. La fonction utilise un nombre fixe de variables de taille fixée d'où une complexité spatiale en $O(1)$.

Remarque : Ce calcul est simpliste. L'hypothèse d'un coût constant est valide pour des nombres flottants ou dans le cadre d'une exponentiation modulaire mais pour des entiers avec n potentiellement très grand, cette hypothèse ne tient plus vraiment la route...