Complexité

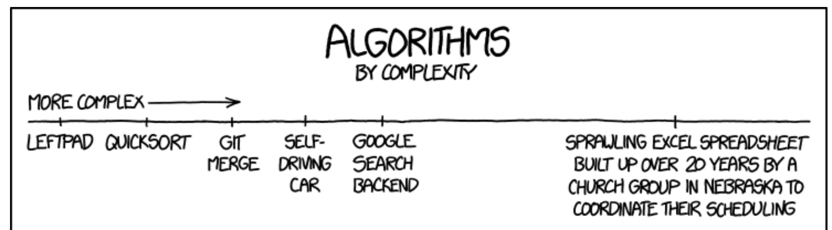


Table des matières

1 Complexité **1**
- 1.1 Notation mathématique . . . 1
- 1.2 Type de ressource . . . 2
- 1.3 Complexité dans le pire des cas . . . 3
- 1.4 Complexité en moyenne . . . 3
- 1.5 Complexité temporelle et temps de calcul . . . 4

2 Calcul de complexité temporelle **5**
- 2.1 Algorithme itératif . . . 5
- 2.2 Algorithme récursif . . . 8

3 Calcul de complexité spatiale **11**
- 3.1 Algorithme itératif . . . 11
- 3.2 Algorithme récursif . . . 12

1 Complexité

1.1 Notation mathématique

Définition 1.1

Soit (u_n) et (v_n) deux suites réelles positives

— On dit que $u_n = \mathrm{O}(v_n)$ lorsqu'il existe $B > 0$ et $N \in \mathbb{N}$ tels que

$$\forall n \geqslant N, \quad u_n \leqslant Bv_n.$$

— On dit que $u_n = \Omega(v_n)$ lorsqu'il existe $A > 0$ et $N \in \mathbb{N}$ tels que

$$\forall n \geqslant N, \quad u_n \geqslant Av_n.$$

— On dit que $u_n = \Theta(v_n)$ lorsqu'il existe $A, B > 0$ et $N \in \mathbb{N}$ tels que

$$\forall n \geqslant N, \quad Av_n \leqslant u_n \leqslant Bv_n.$$

Remarques

- On a $u_n = \Theta(v_n)$ si et seulement si $u_n = \mathrm{O}(v_n)$ et $u_n = \Omega(v_n)$.
- La relation Θ est une relation d'équivalence sur l'ensemble des suites positives. En particulier, elle est symétrique.
- Ces définitions sont asymptotiques : Elles ne dépendent pas des premiers termes de ces suites.
- Lorsque nous ferons des calculs de complexité, nous travaillerons avec des suites (v_n) strictement positives. Dans ce cas
  — $u_n = \mathrm{O}(v_n)$ si et seulement si il existe $B > 0$ tel que : $\forall n \in \mathbb{N}, \quad u_n \leqslant Bv_n$.
  — $u_n = \Omega(v_n)$ si et seulement si il existe $A > 0$ tel que : $\forall n \in \mathbb{N}, \quad u_n \geqslant Av_n$.
  — $u_n = \Theta(v_n)$ si et seulement si il existe $A, B > 0$ tels que : $\forall n \in \mathbb{N}, \quad Av_n \leqslant u_n \leqslant Bv_n$.
  Ces caractérisations n'ont plus besoin du « à partir d'un certain rang ».

⇨ Il faut bien retenir les interprétations intuitives :
— « $u_n = \mathrm{O}(v_n)$ » signifie « u_n est au plus de l'ordre de grandeur de v_n ».
— « $u_n = \Omega(v_n)$ » signifie « u_n est au moins de l'ordre de grandeur de v_n ».
— « $u_n = \Theta(v_n)$ » signifie « u_n et v_n sont du même ordre de grandeur ».

⇨ Si $v_n = \Theta(w_n)$, alors une suite u_n est un O, un Ω ou un Θ de v_n si et seulement si c'est un O, un Ω ou un Θ de w_n. En particulier, puisque quel que soit $b > 1$, $\log_b n = \Theta(\ln n)$, dire que $u_n = \Theta(\log_2 n)$ est équivalent à dire que $u_n = \Theta(\ln n)$. On écrira simplement $u_n = \Theta(\log n)$.

Proposition 1.2

Soit (u_n) une suite positive et (v_n) une suite strictement positive. On suppose que

$$\frac{u_n}{v_n} \xrightarrow[n\to+\infty]{} l \in \mathbb{R}_+ \cup \{+\infty\}.$$

Alors
— $u_n = \mathrm{O}(v_n)$ si et seulement si $l < +\infty$.
— $u_n = \Omega(v_n)$ si et seulement si $l > 0$.
— $u_n = \Theta(v_n)$ si et seulement si $0 < l < +\infty$.

Exercice 1

⇨ Déterminer les relations de comparaison entre les suites de terme général
— $u_n := \lfloor \ln n \rfloor$ et $v_n := \ln n$.
— $u_n := 12n^2 + 3n \log n - n$ et $v_n := n^2$.
— $u_n := 12n^2$ et $v_n := n^{17}$.
— $u_n := 12n^2$ et $v_n := n^2$.

1.2 Type de ressource

Étudier la complexité d'un algorithme, c'est s'intéresser aux ressources qu'il consomme pour effectuer sa tâche, et plus précisément à la manière dont cette consommation évolue lorsque la taille des données augmente. Les principales ressources auxquelles on peut s'intéresser sont :
— le *temps* de calcul.
— l'*espace* mémoire.
— l'*énergie*, qui prend une importance de plus en plus grande à cause de son impact sur
  — l'*autonomie*, principalement dans les téléphones.
  — le *bilan écologique* et le *cout monétaire* des calculs, principalement à l'échelle d'un datacenter.
— les *données échangées* sur le réseau, qui peuvent être un facteur limitant.

L'étude de la complexité se fait le plus souvent de manière *asymptotique*, c'est-à-dire en faisant tendre la taille des données vers l'infini.

Définition 1.3

On dit que la complexité $C(n)$ d'un algorithme est
— *constante* lorsque $C(n) = \Theta(1)$.
— *logarithmique* lorsque $C(n) = \Theta(\log n)$.
— *linéaire* lorsque $C(n) = \Theta(n)$.
— *quasi-linéaire* lorsque $C(n) = \Theta(n \log n)$.
— *quadratique* lorsque $C(n) = \Theta\left(n^2\right)$.
— *polynomiale* lorsqu'il existe $\alpha \in \mathbb{N}$ tel que $C(n) = \Theta\left(n^\alpha\right)$.
— *exponentielle* lorsqu'il existe $\alpha > 1$ tel que $C(n) = \Theta\left(\alpha^n\right)$.
— *factorielle* lorsque $C(n) = \Theta\left(n!\right)$.

Quand on s'intéresse à la complexité temporelle d'un algorithme, on cherche à évaluer comment son temps d'exécution évolue quand on fait tendre la taille des données n vers l'infini. Comme le temps de calcul dépend de nombreux facteurs difficiles à contrôler comme le langage, l'implémentation ou la machine, on se concentre sur le *nombre d'instructions élémentaires* exécutées. Il faut donc :
— Déterminer le nombre de fois où chaque instruction est exécutée.
— Déterminer si chacune de ces instructions est *élémentaire* ou non. Pour les instructions qui ne sont pas élémentaires, estimer leur complexité.
— Sommer toutes ces complexités et tenter éventuellement d'en tirer des conclusions sur le temps de calcul.

Ce qui caractérise une opération élémentaire, c'est qu'elle s'exécute en temps constant. Dans l'idéal, il serait bon de ne considérer comme élémentaire que les instructions assembleur exécutées par le processeur. En Python, les opérations suivantes s'effectuent en temps constant :

— Utiliser les opérateurs `not`, `and`, et `or` sur les booléens.
— Ajouter, soustraire, multiplier, diviser, comparer deux entiers ou deux flottants. Le fait que Python travaille avec des entiers de taille variable rend ces opérations non élémentaires, mais on supposera que les entiers que nous manipulons restent de taille raisonnable (disons qu'ils sont représentables sur 64 bits) ce qui a pour conséquence le fait que les opérations arithmétiques sur ces derniers restent élémentaires.
— Calculer la longueur d'une liste ou d'une chaine de caractères avec `len(t)`, accéder à ou modifier l'élément d'indice i d'une liste par `t[i]`, accéder au caractère d'indice i d'une chaine de caractères par `s[i]`.
— Ajouter ou enlever un élément à la fin d'une liste grâce aux méthodes `append` et `pop`.
— Enfiler ou défiler un élément sur une file du module `collections`. Tester si une clé fait partie d'un dictionnaire, obtenir la valeur associée à une clé, créer, mettre à jour ou supprimer une association dans un dictionnaire.
— Affecter une variable.
— Appeler une fonction.

Cependant, les opérations suivantes ne s'effectuent pas en temps constant :
— Si x est un entier, le calcul de x^n par `x ** n` se fait en $\Theta(\log n)$.
— La concaténation `u + v` de deux chaines de caractères ou de deux listes u et v s'effectue en $\Theta(|u| + |v|)$.
— La création d'une liste de taille n par `[x] * n` s'effectue en $\Theta(n)$.
— La méthode `u.extend(v)` s'effectue en $\Theta(|v|)$.
— Le slicing `t[a:b:p]` s'effectue en un temps proportionnel à la longueur de la liste créée.

Dans de nombreux exemples, il arrive qu'on vous rappelle quelles sont les opérations élémentaires que l'on doit prendre en compte pour le calcul de la complexité. Par exemple, lors de l'étude de tris, il est courant de ne prendre en compte que le nombre de comparaisons. Dans ce cas, les autres opérations doivent être ignorées. Cependant, comme on détermine les complexités à un Θ près, la spécification exacte des opérations élémentaires à prendre en compte n'a en général aucune influence sur le résultat final.

1.3 Complexité dans le pire des cas

On donne toujours la complexité d'un algorithme en fonction de la taille n de l'entrée. Pourtant, la plupart des algorithmes peuvent avoir un temps d'exécution très variable entre deux entrées de même taille. La fonction suivante, qui cherche si un élément x appartient ou non à la liste t de longueur n, s'exécute en temps constant si le premier élément de a vaut x, et en temps proportionnel à n si x n'appartient pas à t.

```
1 def appartient(x, t):
2     """appartient(x: int, t: list[int]) -> bool"""
3     for i in range(len(t)):
4         if t[i] == x:
5             return True
6     return False
```

Par défaut, nous nous intéresserons toujours à la complexité *dans le pire des cas* : autrement dit, le $C(n)$ cherché est le nombre maximum d'opérations élémentaires pour traiter une entrée de taille n. Dans ce cadre, la fonction précédente a une complexité en $\Theta(n)$.

Définition 1.4

Si un algorithme nécessite $C(d)$ opérations élémentaires pour s'exécuter sur une donnée d, on appelle
— *complexité dans le pire des cas* et on note $C_{\max}(n)$, le nombre maximal d'opérations élémentaires nécessaires pour traiter une donnée de taille n.

$$C_{\max}(n) \coloneqq \max_{|d|=n} C(d).$$

— *complexité dans le meilleur des cas* et on note $C_{\min}(n)$, le nombre minimal d'opérations élémentaires nécessaires pour traiter une donnée de taille n.

$$C_{\min}(n) \coloneqq \min_{|d|=n} C(d).$$

Exemple

⇨ Si on compte comme opération élémentaire le nombre de tests `==`, la fonction `appartient` a une complexité dans le pire et dans le meilleur des cas de

$$C_{\max}(n) = n = \Theta(n), \qquad C_{\min}(n) = 1 = \Theta(1).$$

Le pire cas est obtenu lorsque le tableau ne contient pas l'élément x et le meilleur des cas est obtenu lorsque l'élément x est contenu dans la case d'indice 0 du tableau t.

1.4 Complexité en moyenne

Certains algorithmes peuvent être très lents dans une petite proportion de cas « pathologiques » et très efficaces sur les autres. Il peut alors être intéressant de calculer la *complexité en moyenne* de l'algorithme, c'est-à-dire l'espérance du temps de calcul pour une certaine loi de probabilité sur les données.

Ce type de complexité est plus délicat à étudier que la complexité dans le pire des cas, et ce pour deux raisons.

— Il n'est pas toujours évident de définir une loi de probabilité sur les données, et encore moins une loi de probabilité pertinente. Dans l'exemple précédent, quelle peut bien être la probabilité que le premier élément soit égal à x ?

— Une fois que l'on a fixé la loi de probabilité, les calculs sont en général beaucoup plus délicats que pour le pire cas. Il peut même être nécessaire de faire des mathématiques très difficiles.

Définition 1.5

Si un algorithme nécessite $C(d)$ opérations élémentaires pour s'exécuter sur une donnée d, et si $\mathbb{P}$ est une mesure de probabilité sur l'ensemble des données de taille n, on appelle *complexité en moyenne* et on note $C_{\text{moy}}(n)$ le nombre

$$C_{\text{moy}}(n) \coloneqq \sum_{|d|=n} \mathbb{P}(d)C(d)$$

Remarques

⇨ Dans le cas où il existe un nombre fini m de données $d_1, \ldots, d_m$ de taille n, on prend le plus souvent la loi de probabilité uniforme. Dans ce cas

$$C_{\text{moy}}(n) \coloneqq \frac{1}{m} \sum_{k=1}^{m} C(d_k).$$

⇨ On a bien évidemment

$$C_{\text{moy}}(n) = \mathrm{O}(C_{\max}(n)) \quad \text{et} \quad C_{\text{moy}}(n) = \Omega(C_{\min}(n)).$$

⇨ L'exemple le plus connu, et sans doute le plus important, d'algorithme pour lequel il est pertinent de faire une analyse en moyenne est celui du *tri rapide*. En effet, il est en $\Theta\left(n^2\right)$ dans le pire des cas mais en $\Theta(n \log n)$ en moyenne si le tableau d'entrée est dans un ordre aléatoire. De plus, étant donné qu'il se fait en place, il est souvent plus efficace que le tri fusion qui est pourtant en $\Theta(n \log n)$ dans le pire cas.

⇨ Voici les complexités des différents algorithmes de tri que nous avons vu :

Méthode	$C_{\min}(n)$	$C_{\text{moy}}(n)$	$C_{\max}(n)$
Tri sélection	$\Theta\left(n^2\right)$	$\Theta\left(n^2\right)$	$\Theta\left(n^2\right)$
Tri insertion	$\Theta(n)$	$\Theta\left(n^2\right)$	$\Theta\left(n^2\right)$
Tri fusion	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Tri rapide	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta\left(n^2\right)$

1.5 Complexité temporelle et temps de calcul

Après avoir analysé la complexité temporelle d'un algorithme, on dispose d'une information du type $T(n) = \Theta(n \log n)$. En théorie, cela ne nous dit absolument rien du temps de calcul réel pour une valeur donnée de n, mais en pratique on peut en tirer quelques informations.

Valeur de la constante cachée : Pour un algorithme raisonnablement simple, on peut supposer que la constante multiplicative cachée dans le Θ est de l'ordre de 10, voire de 100. Pour certains algorithmes très sophistiqués, elle peut cependant être très grande et rendre l'algorithme beaucoup moins efficace qu'on ne le pense, voire inutilisable en pratique.

Traduction d'une opération élémentaire : Certaines des opérations élémentaires vues plus haut (ajouter deux flottants, par exemple) correspondent directement à une instruction processeur. D'autres sont plus complexes et correspondront à plusieurs instructions (une bonne centaine pour faire un `append` en Python).

Cycle processeur : Vu de l'extérieur, l'état d'un processeur évolue de manière discrète. Il est dans un certain état à l'instant t_n, dans un certain état à l'instant $t_{n+1} = t_n + h$ et dans un état non défini entre ces deux instants. Ce h est la durée d'un *cycle*, c'est-à-dire l'inverse de la *fréquence d'horloge* que les constructeurs communiquent. Comme vous le savez peut-être, la fréquence d'un processeur actuel varie entre 1 GHz et 5 GHz, et un cycle prend donc de 0.2ns à 1ns.

Temps pour exécuter une instruction : Exécuter une instruction prend au moins un cycle (un processeur peut cependant exécuter plusieurs instructions en parallèle sur un même cœur), mais peut prendre beaucoup plus longtemps. Quelques exemples :

— Ajouter deux entiers, comparer deux flottants : 1 cycle.
— Une division entière : Une vingtaine de cycles.
— Le calcul du cosinus d'un nombre flottant : Une centaine de cycles.
— Accès mémoire : entre 1 et 1000 cycles.

Une recette de cuisine : En étant optimiste
— La constante cachée vaut 2 (très optimiste).
— Chaque opération élémentaire donne 5 instructions (optimiste).
— On a un IPC (nombre d'instructions exécutées par cycle d'horloge) de 2 (très raisonnable).
— Un cycle prend 0.2ns (optimiste).

On arrive alors à un temps de calcul de $f(n)$ nanosecondes si la complexité est en $\Theta(f(n))$. C'est possible si par exemple on programme bien une multiplication matricielle en assembleur, voire en C. Si au contraire on travaille dans un langage « lent » comme Python et si l'algorithme se prête moins à un traitement efficace en machine, on peut facilement être mille fois plus lent. On pourra donc retenir la recette suivante : *Si la complexité temporelle est en $\Theta(f(n))$, pour des valeurs de n « pas trop petites », le temps de calcul sera le plus souvent compris entre $f(n)$ nanosecondes et $f(n)$ microsecondes.*

	10	100	1 000	10 000	1 000 000	10^9
$\Theta(\log n)$	1ns	10ns	10ns	10ns	10ns	100ns
$\Theta(\sqrt{n})$	1ns	10ns	100ns	100ns	1mics	1ms
$\Theta(n)$	10ns	100ns	1mics	10mics	1ms	1s
$\Theta(n \log n)$	10ns	100ns	10mics	100mics	10ms	10s
$\Theta(n^2)$	100ns	10mics	1ms	100ms	10 min	10 ans
$\Theta(n^3)$	1mics	1ms	1s	10 min	10 ans	∞
$\Theta(2^n)$	1mics	∞	∞	∞	∞	∞

Ordre de grandeur du temps de calcul pour quelques valeurs de n et complexités usuelles.

On est ici très optimiste : il faut par exemple comprendre que pour une complexité en $\Theta(n \log n)$ avec $n = 10^6$, on prendra *au mieux quelques dizaines de millisecondes*. Pour une version pessimiste, il faut essentiellement tout multiplier par mille.

2 Calcul de complexité temporelle

Dans cette partie, on suppose que l'on connait la complexité de toutes les fonctions prédéfinies qu'on utilise (on sait donc que `len` est en $\Theta(1)$ par exemple), et que l'on souhaite calculer la complexité temporelle dans le pire cas d'une fonction. Autrement dit, on cherche une estimation asymptotique du nombre $C(n)$ d'opérations effectuées dans le pire cas, idéalement sous la forme d'un Θ, ou d'un « bon » O. Si nous parlons d'un bon « O », c'est que si vous prouvez rigoureusement que la complexité du tri par insertion est en $O(n!)$, vous n'avez certes pas commis d'erreur, mais vous n'avez pas non plus obtenu de points.

2.1 Algorithme itératif

2.1.1 Boucles for

Pour une boucle `for`, il faut simplement sommer le nombre d'opérations pour chacune des itérations.

Commençons par l'exemple du tri par sélection, dont le code nous est rappelé ci-dessous :

```
def swap(t, i, j):
    """swap(t: list[int], i: int, j: int) -> NoneType"""
    t[i], t[j] = t[j], t[i]

def indice_minimum(t, i):
    """indice_minimum(t: list[int], i: int) -> int"""
    j_min = i
    for j in range(i + 1, len(t)):
        if t[j] < t[j_min]:
            j_min = j
    return j_min

```

```python
def tri_selection(t):
    """tri_selection(t: list[int]) -> NoneType"""
    for i in range(len(t) - 1):
        j = indice_minimum(t, i)
        swap(t, i, j)
```

La fonction `swap` s'effectue en $\Theta(1)$ car elle n'effectue que deux accès mémoire et deux affectations. Si on note n la longueur du tableau, la fonction `indice_minimum` effectue quant à elle $n-1-i$ passages dans la boucle. Comme les opérations à l'intérieur de la boucle sont élémentaires, sa complexité est donc en $\Theta(n-1-i)$. La complexité de la fonction `tri_selection` est donc en

$$
\begin{aligned}
C(n) = \sum_{i=0}^{n-2} \left[\Theta(n-1-i) + \Theta(1)\right] &= \sum_{i=0}^{n-2} \Theta(n-1-i) = \Theta\left(\sum_{i=0}^{n-2} (n-1-i)\right) \\
&= \Theta((n-1) + \cdots + 2 + 1) = \Theta\left(\frac{n(n-1)}{2}\right) \\
&= \Theta(n^2).
\end{aligned}
$$

L'algorithme de tri sélection est donc un algorithme de complexité quadratique. Remarquons que nous aurions trouvé le même résultat si nous avions seulement compté le nombre de comparaisons.

Attention à bien voir la différence entre des boucles *successives* qui donnent une complexité en $\Theta(n)$ si les corps de boucle sont des instructions élémentaires

```python
for i in range(n):
    corps de................  # On passe n fois ici
    ..................boucle
for i in range(n):
    corps de................  # On passe n fois ici
    ..................boucle
```

et des boucles *imbriquées* qui donnent une complexité en $\Theta(n^2)$ dans la même situation.

```python
for i in range(n):
    for j in range(n):
        corps de................  # On passe n^2 fois ici
        ..................boucle
```

Prenons désormais un exemple un peu plus général. On suppose maintenant que f est une fonction de signature `f(t: list[int], i: int) -> int`, et on considère la fonction :

```python
def g(t):
    """g(t: list[int]) -> NoneType"""
    n = len(t)
    for i in range(n):
        j = f(t, i)
        t[i] = j
```

La ligne 3 n'est exécutée qu'une seule fois, et prend un temps unitaire. Les lignes 5 et 6, le *corps* de la boucle, sont exécutées n fois chacune. La ligne 6 est une opération élémentaire, mais la ligne 5 contient un appel à une fonction inconnue f avec le paramètre i. Le nombre total d'instructions exécutées dans cette boucle est donc

$$C(n) = \sum_{i=0}^{n-1} \left[1 + T(i)\right].$$

où $T(i)$ correspond au nombre d'instructions pour le calcul de `f(t, i)`.

— Si f s'exécute en temps constant, par exemple si

```python
def f(t, i):
    """f(t: list[int], i: int) -> int"""
    return t[i]
```

alors chaque itération est en $\Theta(1)$ et on obtient au total

$$C(n) = \sum_{i=0}^{n-1} \left[1 + \Theta(1)\right] = \sum_{i=0}^{n-1} \Theta(1) = \Theta(n).$$

— Le temps d'exécution de `f(t, i)` pourrait aussi dépendre de `n`, tout en restant indépendant de `i`. Par exemple

```python
def f(t, i):
    """f(t: list[int], i: int) -> int"""
    k = 0
    n = len(t)
    for j in range(n):
        if t[j] < t[i]:
            k += 1
    return k
```

Dans ce cas, chacun des termes de la somme est un $\Theta(n)$, et l'on obtient donc

$$C(n) = \sum_{i=0}^{n-1} [1 + \Theta(n)] = \sum_{i=0}^{n-1} \Theta(n) = \Theta\left(\sum_{i=0}^{n-1} n\right) = \Theta\left(n^2\right).$$

— Considérons maintenant la fonction `f` suivante, qui s'exécute en temps $\Theta(i)$.

```python
def f(t, i):
    """f(t: list[int], i: int) -> int"""
    k = 0
    for j in range(i):
        if t[j] < t[i]:
            k += 1
    return k
```

— Le calcul de la complexité totale ne pose pas de problème

$$C(n) = \sum_{i=0}^{n-1} [1 + \Theta(i)] = \sum_{i=0}^{n-1} \Theta(i) = \Theta\left(\sum_{i=0}^{n-1} i\right) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2).$$

— Comme on a toujours $i < n$, la fonction f s'exécute en temps $\mathrm{O}(n)$. Si l'on souhaite seulement une *majoration* de la complexité totale, on peut donc écrire

$$C(n) = \sum_{i=0}^{n-1} [1 + \mathrm{O}(n)] = \sum_{i=0}^{n-1} \mathrm{O}(n) = \mathrm{O}\left(\sum_{i=0}^{n-1} n\right) = \mathrm{O}(n^2).$$

Au point précédent, on avait obtenu une estimation plus précise : notre « grand-O » est en fait un Θ. Ce ne sera pas toujours le cas. Par exemple, si la fonction f s'exécute en temps $\Theta(2^i)$, alors une majoration brutale donnerait

$$C(n) = \sum_{i=0}^{n-1} \left[1 + \Theta\left(2^i\right)\right] = \sum_{i=0}^{n-1} \Theta\left(2^i\right) = \sum_{i=0}^{n-1} \mathrm{O}\left(2^n\right) = \mathrm{O}\left(\sum_{i=0}^{n-1} 2^n\right) = \mathrm{O}(n2^n).$$

C'est correct, car c'est bien une *majoration*, mais elle est grossière. En réalité, on a

$$C(n) = \sum_{i=0}^{n-1} \Theta\left(2^i\right) = \Theta\left(\sum_{i=0}^{n-1} 2^i\right) = \Theta(2^n - 1) = \Theta(2^n).$$

Dans les exemples précédents, nous avons à plusieurs reprises dû trouver un ordre de grandeur de la somme $\sum_{k=0}^{n-1} k$. Comme nous sommes amenés à manipuler des sommes de ce type, on pourra utiliser directement les résultats suivants :

Proposition 2.1

Soit $\alpha, \beta \geqslant 0$ et $\gamma > 1$. Alors

$$\sum_{k=0}^{n} k^\alpha = \Theta(n^{\alpha+1}), \qquad \sum_{k=1}^{n} k^\alpha \log^\beta k = \Theta(n^{\alpha+1} \ln^\beta n) \qquad \text{et} \qquad \sum_{k=0}^{n} \gamma^k = \Theta(\gamma^n).$$

2.1.2 Boucle while

La différence avec une boucle `for` est qu'on ne connait pas *à priori* le nombre d'itérations. Le plus souvent, on sera amené à le majorer, mais il faudra faire attention à ne pas être trop grossier.

Revenons à l'exemple de la recherche d'un élément d'une liste triée par dichotomie, dont nous avons donné le code dans le chapitre sur les listes.

```python
def dichotomie(x, t):
    """dichotomie(x: int, t: list[int]) -> bool"""
    g = 0
    d = len(t)
    while g < d:
        m = (g + d) // 2
        if x == t[m]:
            return True
        elif x < t[m]:
            d = m
        else:                  # Cas où x > t[m]
            g = m + 1
    return False
```

On note g_k et d_k les valeurs respectives de `g` et `d` lors du passage d'indice k dans la boucle. On a $g_0 := 0$ et $d_0 := n$ et, dans les deux cas où x n'est pas trouvé à l'itération k, on montre que

$$d_{k+1} - g_{k+1} \leqslant \frac{d_k - g_k}{2}$$

Autrement dit, la largeur $l_k := d_k - g_k$ de la tranche de recherche est au moins divisée par 2 à chaque itération. On en déduit que

$$l_k \leqslant \frac{n}{2^k}.$$

Dès que $l_k \leqslant 1$, on sort de la boucle au plus tard à l'itération suivante. Or

$$\frac{n}{2^k} \leqslant 1 \iff 2^k \geqslant n \iff \log_2(2^k) \geqslant \log_2 n \iff k \geqslant \log_2 n \iff k \geqslant \lceil \log_2 n \rceil.$$

Cet algorithme effectue donc au plus $1 + \lceil \log_2 n \rceil$ itérations. La complexité dans le pire des cas de la recherche dichotomique est donc en $\mathrm{O}(\log n)$. Comme ces itérations sont toutes effectuées si la liste ne contient pas l'élément recherché, on se convaincra que cette complexité est en fait en $\Theta(\log n)$. D'autre part, la complexité dans le meilleur des cas est en $\Theta(1)$, ce cas étant atteint lorsque l'élément x est exactement au milieu de notre liste.

Remarque

⇨ Attention, il ne faut pas oublier de prendre en compte le temps nécessaire à évaluer la condition de la boucle. Pour prendre un exemple un peu idiot, la fonction suivante s'exécute en temps $\Theta(|u|^2)$.

```python
def somme(t):
    """somme(t: list[int]) -> int"""
    s = 0
    for x in t:
        s = s + x
    return s

def f(u, borne):
    """f(u: list[int], borne: int) -> int"""
    i = 0
    while i < len(u) and somme(u[:i]) < borne: # Evaluation en O(i) !!!
        i += 1                                  # Corps de la boucle en O(1)
    return i
```

2.2 Algorithme récursif

Pour la complexité temporelle, les algorithmes récursifs font naturellement apparaitre une formule de récurrence. Diverses techniques que nous allons voir nous permettent d'en déduire le comportement asymptotique.

Dans les cas les plus simples, on commencera par établir une forme close pour $C(n)$. Prenons l'exemple de la fonction suivante qui calcule la factorielle d'un entier positif :

```python
def factorielle(n):
    """factorielle(n: int) -> int"""
    if n == 0:
        return 1
    else:
        return n * factorielle(n - 1)
```

On souhaite estimer la complexité de cette fonction en calculant le nombre $C(n)$ de multiplications effectuées. La lecture du code nous donne

$$C(0) = 0, \qquad \text{et} \qquad \forall n \in \mathbb{N}^*, \quad C(n) = C(n-1) + 1.$$

On en déduit que $C(n) = n$ et donc que $C(n) = \Theta(n)$.

Mais rapidement, on se rend compte que les formes closes pour $C(n)$ deviennent complexes. Considérons par exemple l'algorithme d'exponentiation rapide dans sa version récursive :

```
def expo_rapide(x, n):
    """expo_rapide(x: int, n: int) -> int"""
    if n == 0:
        return 1
    else:
        p = n // 2
        y = expo_rapide(x, p)
        if n % 2 == 0:
            return y * y
        else:
            return x * y * y
```

Si on note $C(n)$ le nombre de multiplications effectuées pour le calcul de x^n, on a

$$C(0) = 0, \quad \text{et} \quad \forall n \in \mathbb{N}^*, \quad C(n) = \begin{cases} C(\lfloor n/2 \rfloor) + 1 & \text{Si } n \text{ est pair,} \\ C(\lfloor n/2 \rfloor) + 2 & \text{Si } n \text{ est impair.} \end{cases}$$

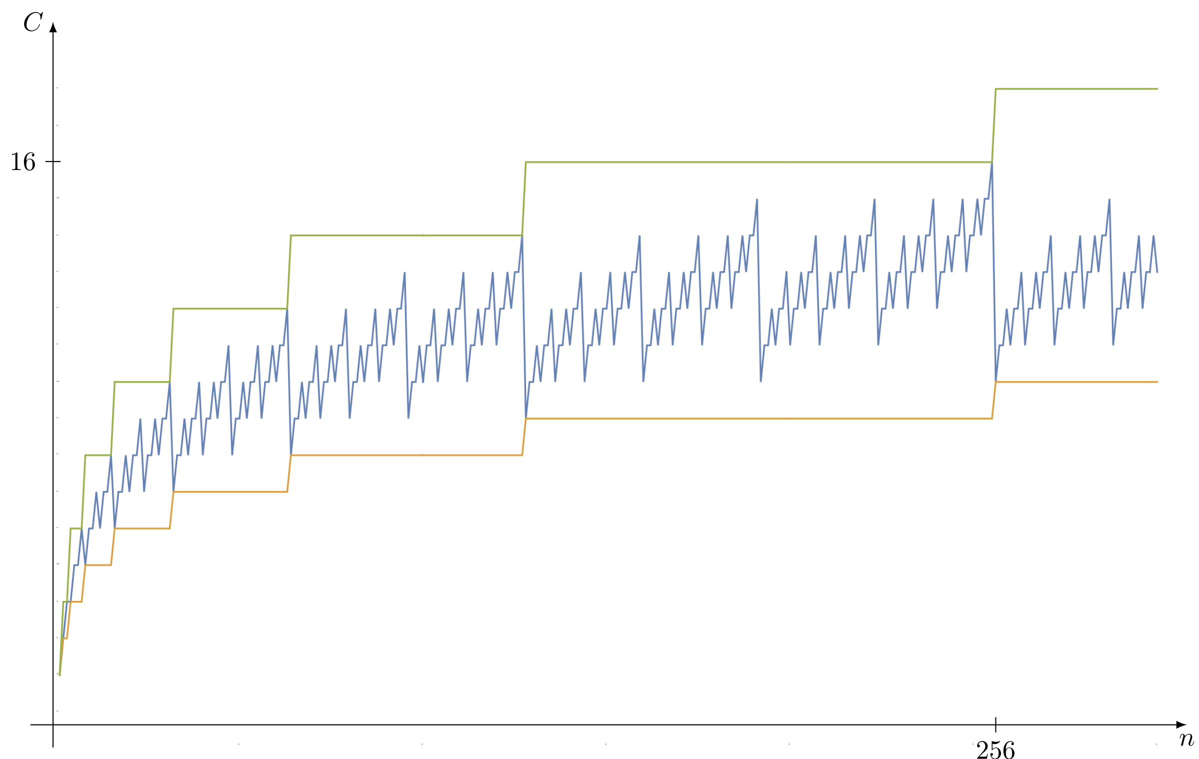
Cette récurrence devient plus facile à appréhender si l'on décompose n en base 2. Si $n = \underline{d_{w-1} \dots d_1 d_0}_2$, on a

$$C(\underline{d_{w-1} \dots d_1 d_0}_2) = \begin{cases} C(\underline{d_{w-1} \dots d_1}_2) + 1 & \text{Si } d_0 = 0, \\ C(\underline{d_{w-1} \dots d_1}_2) + 2 & \text{Si } d_0 = 1. \end{cases}$$

En notant $z(n)$ le nombre de 0 dans la décomposition de n en base 2 et $u(n)$ le nombre de 1 dans cette même décomposition, on en déduit que $C(n) = z(n) + 2u(n)$. Autrement dit, si on définit $b(n) \coloneqq z(n) + u(n)$ comme le nombre de chiffres dans la décomposition en base 2 de n, on a $C(n) = b(n) + u(n)$. Puisque $b(n) = 1 + \lfloor \log_2 n \rfloor$ et $1 \leqslant u(n) \leqslant b(n)$, on en déduit que

$$2 + \lfloor \log_2 n \rfloor \leqslant C(n) \leqslant 2 + 2\lfloor \log_2 n \rfloor,$$

ce qui nous permet de conclure que $C(n) = \Theta(\log n)$. Le graphe ci-dessous représente au centre, en bleu, la courbe d'évolution de $C(n)$, encadrée par les deux fonctions obtenues dans l'inégalité précédente.



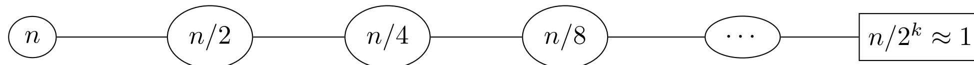
Le fait qu'un programme aussi simple ait une complexité au comportement aussi erratique nous permet de réaliser qu'il est illusoire d'obtenir des formes closes de $C(n)$ pour la plupart des programmes que nous rencontrerons. C'est pourquoi, on se contentera d'une estimation asymptotique. Malheureusement, l'obtention rigoureuse de ces estimations est extrêment technique ; on se permettra donc de sacrifier une trop grande rigueur mathématique. Les deux techniques suivantes nous seront très utiles.

— *Technique de la sommation télescopique* : Elle consiste à « résoudre » la relation de récurrence en se permettant quelques approximations. Dans notre cas, on confondra $\lfloor n/2 \rfloor$ et $n/2$ et on écrira $C(n) = C(n/2) + \Theta(1)$, puis

$$\begin{array}{rcl} C(n) - C(n/2) & = & \Theta(1) \\ C(n/2) - C(n/4) & = & \Theta(1) \\ \vdots & = & \vdots \\ C(n/2^{k-1}) - C(n/2^k) & = & \Theta(1) \end{array}$$

où k est tel que $n/2^k \approx 1$, c'est-à-dire $k \approx \log_2 n$. En sommant ces égalités, on obtient $C(n) - C(1) = \Theta(1) \log_2 n$, donc $C(n) = \Theta(\log n)$.

— *Technique de l'arbre d'appels* : Cette technique commence par faire une esquisse de l'arbre d'appels de notre fonction récursive :



Puisque $n/2^k \approx 1$, on en déduit que $k \approx \log_2 n$. La profondeur de notre arbre d'appels est donc de l'ordre de $\log_2 n$. Pour chaque appel, on calcule ensuite sa contribution propre à la complexité totale. Autrement dit, on estime les opérations élémentaires effectuées par cet appel en ignorant celles effectuées par ses enfants. Dans notre cas, chaque appel a une complexité propre en $\Theta(1)$. On somme ensuite ces données sur l'ensemble des noeuds de l'arbre. Dans notre cas, on obtient une complexité totale en $C(n) = \Theta(1) \log_2 n = \Theta(\log n)$.

Ces deux techniques sont sur le fond assez semblables, mais la technique de l'arbre d'appels montrera rapidement sa supériorité lorsque les arbres seront plus complexes, comme nous allons le voir dans l'exemple suivant.

Nous allons revenir sur l'algorithme de tri fusion dont nous rappelons le code ici. On commence par remarquer que la fonction `fusion(t1, t2)` a une complexité en $\Theta(|t_1| + |t_2|)$. En effet, chaque passage dans la boucle `while` ajoute un élément parmi les listes t_1 et t_2 à notre liste t. Les appels à `extend` ajoutent les éléments restants une fois qu'une des deux listes est épuisée.

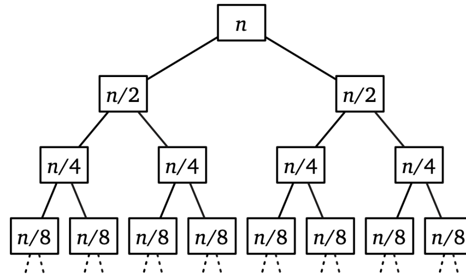
```
1  def fusion(t1, t2):
2      """fusion(t1: list[int], t2: list[int]) -> list[int]"""
3      t = []
4      i1 = 0
5      i2 = 0
6      while i1 < len(t1) and i2 < len(t2):
7          if t1[i1] < t2[i2]:
8              t.append(t1[i1])
9              i1 = i1 + 1
10         else:
11             t.append(t2[i2])
12             i2 = i2 + 1
13     t.extend(t1[i1:])
14     t.extend(t2[i2:])
15     return t
```

```
1  def tri_fusion(t):
2      """tri_fusion(t: list[int]) -> list[int]"""
3      n = len(t)
4      if n <= 1:
5          return t[:]
6      n1 = n // 2
7      t1 = tri_fusion(t[0:n1])
8      t2 = tri_fusion(t[n1:n])
9      t = fusion(t1, t2)
10     return t
```

Si on note $C(n)$ la complexité de la fonction `tri_fusion`, on a donc

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

Pour obtenir une estimation asymptotique de $C(n)$, nous allons utiliser la technique de l'arbre d'appels. La racine représente l'appel initial à `tri_fusion` avec une liste de taille n, qui appelle récursivement notre fonction avec des listes de taille $n/2$, qui elles-mêmes appellent récursivement notre fonction avec des listes de taille $n/4$, etc.



On tombe sur un cas de base lorsque la taille des listes est inférieure ou égale à 1. La hauteur h de cet arbre vérifie donc $n/2^h \approx 1$, ce qui nous donne $h \approx \log_2 n$. Pour estimer la complexité de l'appel initial, il suffit désormais de prendre en compte le cout propre de chaque appel, c'est-à-dire le cout de la fusion. Pour simplifier le calcul, on va sommer ligne par ligne.

- Sur la première ligne, on compte une fusion pour une liste de taille n ; ce cout est de $\Theta(n)$.
- Sur la seconde ligne, on compte 2 fusions pour des listes de taille $n/2$; ce cout est de $2\Theta(n/2) = \Theta(n)$.
- Sur la troisième ligne, on compte 4 fusions pour des listes de taille $n/4$; ce cout est de $4\Theta(n/4) = \Theta(n)$.

On constate que sur chaque ligne, le cout est de $\Theta(n)$. Comme l'arbre est de hauteur $\log_2 n$, on en déduit que la complexité du tri fusion est de $C(n) = \Theta(n) \log_2 n = \Theta(n \log n)$.

3 Calcul de complexité spatiale

3.1 Algorithme itératif

La complexité en espace mesure la quantité de mémoire de travail utilisée par l'algorithme. On ne compte pas la taille des données.

La fonction suivante calcule le n-ième nombre de Fibonacci pour $n \geqslant 1$:

```
def fibo(n):
    """fibo(n: int) -> int"""
    t = [None] * (n + 1)
    t[0] = 0
    t[1] = 1
    for i in range(2, n + 1):
        t[i] = t[i - 1] + t[i - 2]
    return t[n]
```

Elle a une complexité en espace en $\Theta(n)$ puisqu'elle alloue un tableau de taille $n + 1$ pour réaliser ses calculs. Sa complexité en temps est également linéaire.

La fonction suivante effectue le même calcul :

```
def fibo(n):
    """fibo(n: int) -> int"""
    u = 0
    v = 1
    for _ in range(n):
        u, v = v, u + v
    return u
```

Elle a également une complexité temporelle linéaire, mais sa complexité spatiale est constante, puisqu'elle utilise uniquement deux variables entières.

De nombreux algorithmes « échangent de l'espace contre du temps » : pour obtenir une meilleure complexité temporelle, on accepte une moins bonne complexité spatiale. C'est par exemple le cas de tous les algorithmes de programmation dynamique, que nous étudierons ultérieurement. C'est souvent efficace en pratique, mais il y a un certain nombre de seuils qu'il est couteux de dépasser : les données ne rentrent plus dans le cache, les données ne rentrent plus en mémoire vive, les données ne rentrent plus dans le stockage de masse local, etc.

Taille des données et capacité des mémoires : Il faut avoir en tête quelques ordres de grandeur :

— Un entier ou un flottant prennent en général 8 octets en mémoire.
— Un ordinateur typique a quelques gigaoctets de mémoire vive.
— Ce même ordinateur peut disposer de quelques téraoctets de mémoire de stockage, mais si l'on doit utiliser cette mémoire pour les calculs, les performances peuvent facilement être divisées par mille, voire un million.

3.2 Algorithme récursif

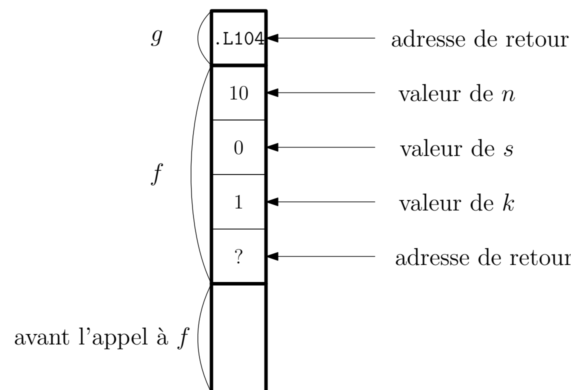
3.2.1 Pile d'appels

Considérons les deux fonctions suivantes

```
def g(n):
    """g(n: int) -> int"""
    return n * n

def f(n):
    """f(n: int) -> int"""
    s = 0
    for k in range(1, n + 1):
        s = s + g(k)     # .L104
    return s
```

Voici l'état de la pile d'appels lors du premier passage ligne 9 dans le calcul de $f(10)$.



Que se passe-t-il « en vrai » quand on calcule $f(2)$? f crée des variables locales, leur donne une valeur, puis appelle g avec l'argument 1. Pour cela, elle cède le contrôle d'exécution à g et l'exécution « saute » au début du code de g. Quand g aura fini son calcul, il faudra qu'elle rende la main à f : mais l'exécution de f doit reprendre là où elle s'est arrêtée, et dans l'environnement qui était valable à ce moment : `s` doit valoir 0, `k` doit valoir 1, etc. Il faut donc que f sauvegarde un certain nombre d'informations avant d'appeler g.

Cette sauvegarde d'informations se fait sur la *pile d'appels*. Vu de loin, un élément (on parle de *stackframe* ou *bloc d'activation*) de la pile contient toutes les informations relatives à un appel donné d'une fonction donnée :
— Les arguments de la fonction
— Les variables locales de la fonction.
— Le point auquel l'exécution du programme devra reprendre quand l'appel sera terminé.

Quand une fonction est appelée, elle crée une *stackframe* au sommet de la pile, qu'elle supprimera juste avant de renvoyer son résultat et de passer le contrôle au point qui était sauvegardé. À tout moment de l'exécution du programme, la hauteur de la pile (en nombre de *frames*) est donc égale au nombre de fonctions actives, c'est-à-dire ayant été appelées et n'ayant pas encore retourné leur résultat.

Une remarque sur la terminologie : le nom de « pile » n'est bien sûr pas anodin, il y a bien une très forte analogie avec la structure de données que nous avons vue. On empile au moment des appels, on dépile au moment des retours.

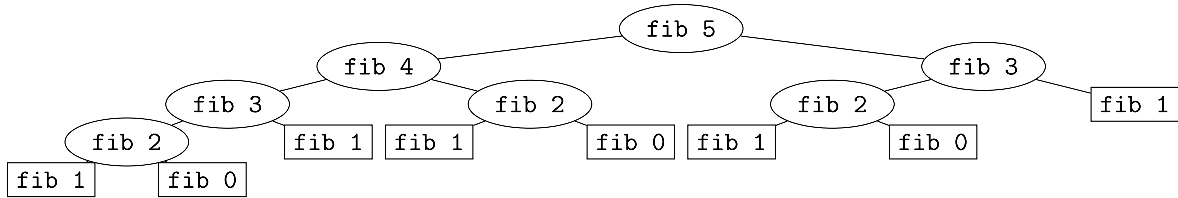
3.2.2 Pile et fonction récursive

Pour l'instant, on a considéré le cas d'une fonction f qui appelle une fonction g. Mais rien n'empêche bien sûr f de s'appeler elle-même : ce sera le cas si f est récursive. Dans ce cas, il y aura *un bloc d'activation par appel actif de f*. C'est logique, puisque chacun de ses appels possède ses propres variables locales, et son propre compteur de programme étant donné que chacun des appels en est à un point différent de son exécution.

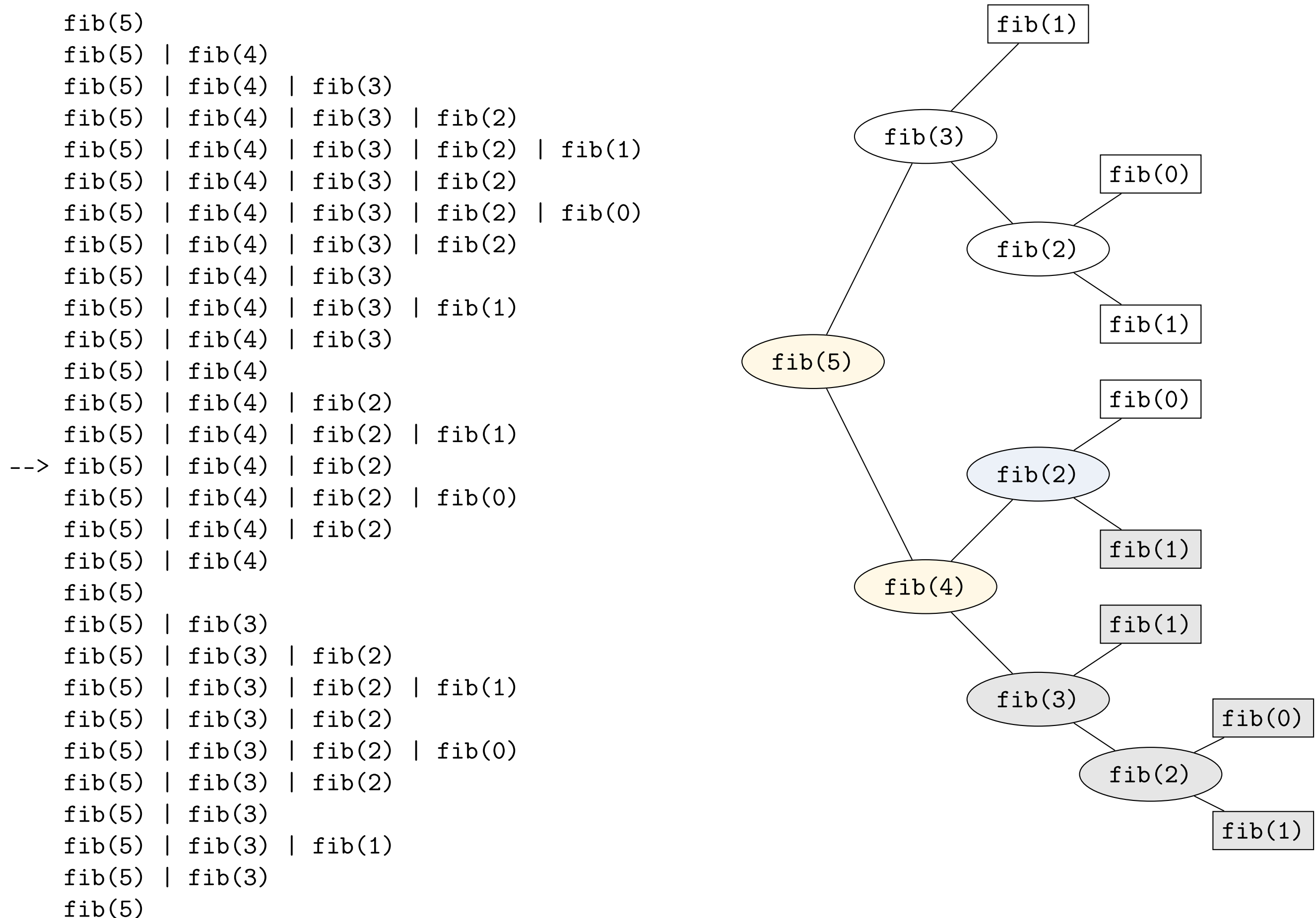
Pour visualiser ce type de situation, il est plus intéressant de réfléchir en termes d'*arbres d'appels*. Considérons une fonction calculant les termes de la suite de Fibonacci de manière récursive et naïve. Ici, « naïve » est à comprendre au sens de *scandaleusement mauvais et contraire aux bonnes mœurs*.

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

Un appel à `fib(5)` produit un appel à `fib(4)` et un à `fib(3)`, qui produisent eux-mêmes d'autres appels, etc. La situation est très bien résumée par l'arbre d'appels suivant pour `fib(5)`.



Voici l'évolution de la pile d'appels lors du calcul de `fib(5)`.



```
    fib(5)
    fib(5) | fib(4)
    fib(5) | fib(4) | fib(3)
    fib(5) | fib(4) | fib(3) | fib(2)
    fib(5) | fib(4) | fib(3) | fib(2) | fib(1)
    fib(5) | fib(4) | fib(3) | fib(2)
    fib(5) | fib(4) | fib(3) | fib(2) | fib(0)
    fib(5) | fib(4) | fib(3) | fib(2)
    fib(5) | fib(4) | fib(3)
    fib(5) | fib(4) | fib(3) | fib(1)
    fib(5) | fib(4) | fib(3)
    fib(5) | fib(4)
    fib(5) | fib(4) | fib(2)
    fib(5) | fib(4) | fib(2) | fib(1)
--> fib(5) | fib(4) | fib(2)
    fib(5) | fib(4) | fib(2) | fib(0)
    fib(5) | fib(4) | fib(2)
    fib(5) | fib(4)
    fib(5)
    fib(5) | fib(3)
    fib(5) | fib(3) | fib(2)
    fib(5) | fib(3) | fib(2) | fib(1)
    fib(5) | fib(3) | fib(2)
    fib(5) | fib(3) | fib(2) | fib(0)
    fib(5) | fib(3) | fib(2)
    fib(5) | fib(3)
    fib(5) | fib(3) | fib(1)
    fib(5) | fib(3)
    fib(5)
```

Il faut avoir de cet arbre une vision dynamique : on en effectue un parcours en profondeur. Quand on est en train de visiter un nœud, les appels actifs sont ceux correspondant au nœud actuel ainsi qu'à tous ses ancêtres, c'est-à-dire tous les nœuds situés sur le chemin qui le relie à la racine. La hauteur maximale de la pile est égale à la hauteur de l'arbre. Ici, cette hauteur est de l'ordre de n alors que le temps d'exécution est exponentiel.

Exercice 2

⇨ Montrer que la complexité temporelle de la fonction `fib` est en

$$\Theta(\varphi^n)$$

où $\varphi \coloneqq (1+\sqrt{5})/2$ est le nombre d'or.

Cependant, si l'on s'intéresse à une fonction ayant une structure d'appels plus simple, les choses peuvent être différentes.

```python
def somme(n):
    """somme(n: int) -> int"""
    if n == 0:
        return 0
    else:
        return n + somme(n - 1)
```

Ici, l'arbre d'appels obtenu est en fait linéaire. Voici par exemple l'arbre d'appels pour `somme(6)` :

Le problème, qui n'est pas forcément évident quand on regarde le code, est qu'un appel à `somme(n)` demande un espace mémoire proportionnel à n : c'est la hauteur maximale de la pile d'appels. De plus, l'espace disponible sur la pile est bien plus limité que l'espace mémoire total. Le résultat :

```
In [1]: somme(3000)
RecursionError: maximum recursion depth exceeded in comparison
```

Notons que le temps de calcul n'est pas le problème : le *stackoverflow* se produit presque instantanément.

Proposition 3.1

La complexité en espace d'une fonction récursive est au moins égale à sa profondeur maximale de récursion, c'est-à-dire à la profondeur de son arbre d'appels.

Remarques

- Elle peut bien sûr être supérieure à cette profondeur, typiquement si la fonction crée des listes auxiliaires.
- Par défaut, l'espace disponible sur la pile est assez faible, de l'ordre de quelques mégaoctets. On peut donc faire un *stackoverflow* bien avant d'épuiser la mémoire disponible.

Ici, cette complexité linéaire en espace est problématique : une fonction itérative aura clairement une utilisation mémoire constante, sauf si l'on s'amuse à stocker tous les résultats intermédiaires.

```python
def somme(n):
    """somme(n: int) -> int"""
    s = 0
    for k in range(1, n + 1):
        s += k
    return s
```

Et le problème disparait :

```
In [2]: somme(3000)
Out[2]: 4501500
```