Cours n° 6

Complexité des algorithmes

Nous avons vu qu'après avoir traduit un algorithme en un programme syntaxiquement correct, il fallait s'assurer que ce programme terminait et retournait un résultat correct lorsque les préconditions portant sur les arguments sont respectées.
Le problème suivant est celui des **performances** du programme produit : un programme correct ne sert à rien en pratique s'il prend un temps déraisonnable pour retourner son résultat.

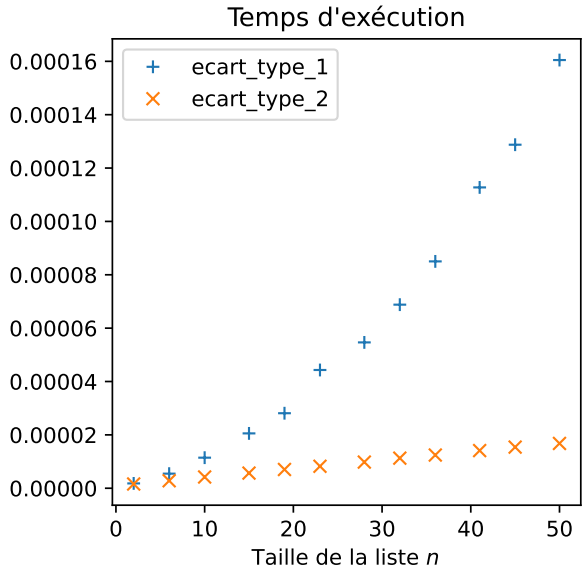
Exemple On programme deux fonctions, subtilement différentes, pour calculer l'écart-type d'une liste de nombres flottants. On rappelle la définition mathématique de l'écart-type d'une série $(x_1, \dots, x_n)$ de nombres réels :

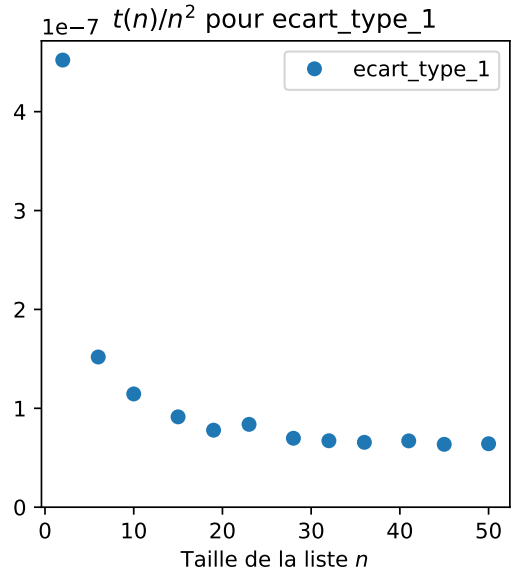
$$\sigma_x = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2} \quad \text{où} \quad \overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i.$$

```python
from math import sqrt

def moyenne(lst: list[float]) -> float:
    S = 0.0
    for x in lst:
        S = S + x
    return S / len(lst)
```

```python
def ecart_type_1(lst: list[float]) -> float:
    S = 0.0
    for x in lst:
        S = S + (x - moyenne(lst))**2
    return sqrt(S / len(lst))

def ecart_type_2(lst: list[float]) -> float:
    S = 0.0
    moy = moyenne(lst)
    for x in lst:
        S = S + (x - moy)**2
    return sqrt(S / len(lst))
```

On peut prouver la correction totale des deux fonctions `ecart_type_1` et `ecart_type_2` sous la précondition que `lst` est une liste non vide de flottants.
On mesure le temps d'exécution, en secondes, d'un appel à chacune des deux fonctions `ecart_type_1` et `ecart_type_2` en faisant varier la taille de la liste fournie. On obtient les résultats suivants :





1. Des deux fonctions `ecart_type_1` et `ecart_type_2`, laquelle semble la plus efficace pour calculer un écart-type ? Expliquer succinctement pourquoi.
2. À l'aide d'une lecture attentive des deux graphiques, proposer une formule donnant approximativement le temps $t(n)$ de calcul de l'écart-type d'une liste de n flottants, et ce, pour chacune des deux fonctions.
3. Estimer le temps de calcul de l'écart-type d'une liste de 1 000 éléments par chacune des deux fonctions. Même question pour une liste de 1 000 000 d'éléments.
4. Pour une liste de n éléments, calculer pour les deux fonctions le nombre d'opérations arithmétiques (`+`, `-`, `*`, `/`, `**`) effectuées pour obtenir l'écart-type de la liste.

1 Concepts de base

1.1 Notion de complexité

L'exécution d'un programme a un coût, qui a deux aspects :

- le **temps** que prend l'exécution du programme ;
- la **quantité de mémoire** utilisée par le programme lors de son exécution.

Ces deux coûts ont tendance à augmenter lorsque la **taille des entrées** du programme augmente.

Définition 1
La **complexité** d'un algorithme est l'expression de son coût en fonction de la taille (ou des tailles) de ses entrées. Plus précisément :

- La **complexité temporelle** évalue le temps nécessaire pour exécuter la fonction.
- La **complexité spatiale** évalue l'espace mémoire nécessaire pour l'exécuter.

La taille des entrées est toujours un entier naturel. Il s'agira la plupart du temps :

- la valeur d'un **nombre entier** n qui est paramètre d'une fonction,
- Parfois, on considérera que la taille d'un entier est le **nombre de chiffres** nécessaires pour l'écrire en binaire. Mathématiquement, il s'agit du nombre entier $p = \lceil \log_2(n) \rceil$;
- de la **longueur** d'une liste, d'une chaîne de caractères paramètre d'une fonction ; du **nombre de clés** d'un dictionnaire.

Nous nous concentrerons surtout cette année sur l'évaluation de la complexité temporelle.

1.2 Complexité dans le pire des cas

Prenons l'exemple de la fonction de recherche « naïve » d'un élément dans une liste :

```
def recherche(lst: list, obj) -> bool:
    pos, trouvé = 0, False
    while not trouvé and pos < len(lst):
        # trouvé dit si obj est dans lst[0:pos]
        if lst[pos] == obj:
            trouvé = True
        else:
            pos = pos + 1
    return trouvé
```

La taille du problème est clairement la longueur n de la liste `lst`.
Remarquons que le nombre d'itérations de la boucle `while` ne va pas seulement dépendre de n, mais aussi de la position de l'élément recherché dans la liste :

- une seule itération si `obj` se trouve au rang 0 dans la liste `lst` ;
- $k+1$ itérations s'il se trouve au rang $k-1$ dans la liste ;
- n itérations s'il ne se trouve pas dans la liste.

Ici, la complexité de l'algorithme ne dépend pas seulement de la taille des entrées, mais aussi de leur valeur exacte : comment évaluer la complexité de cette fonction ?

Méthode 2

On évalue en général la complexité d'un algorithme **dans le pire des cas.** Pour une/des taille(s) d'entrées donnée(s), on fait systématiquement les hypothèses les plus pessimistes en terme de coût d'exécution, et on cherche alors à trouver un majorant du résultat exact.

Remarques

1. Dans l'exemple ci-dessus, les cas les pires sont les situations où l'élément ne se trouve pas dans la liste. La boucle effectuera alors le nombre maximal d'itérations, n, et le temps pris par l'algorithme sera à peu près proportionnel à n. Pour cet exemple, on dit que la *complexité temporelle dans le pire des cas est linéaire.*
2. Il existe d'autres notions de complexité, que nous n'étudierons pas cette année : la complexité **dans le meilleur des cas** (que l'on va plutôt minorer), la **complexité en moyenne** (plus difficile à formaliser et à évaluer).

1.3 Complexité en ordre de grandeur

Évaluer précisément le temps d'exécution et la quantité de mémoire utilisée par un algorithme **est illusoire :**

- les différentes opérations élémentaires ne prennent pas toutes le même temps (par exemple, calculer un produit prend plus de temps à la machine que calculer une somme) et ce temps varie d'une machine à l'autre ;
- les machines actuelles exécutent plusieurs programmes en parallèle, et le temps d'exécution d'une tâche dépend de la disponibilité de la machine ;
- la quantité de mémoire utilisée dépend de la façon précise dont les données sont représentées en binaire dans la mémoire ; elle est rarement connue avec précision et peut également varier d'une machine à l'autre.

Pour la complexité temporelle, on contourne ces problèmes de la façon suivante :

- On considère que sur une machine donnée, les opérations « simples » prendront un temps qui restera entre deux bornes fixées, quelles que soient la valeur de leurs entrées. **On considérera que les opérations élémentaires ont toutes un coût unitaire,** et on se contentera de les **dénombrer** plutôt que de mesurer plus précisément le temps qu'elles prennent.
- Comme différentes opérations élémentaires prennent en réalité des temps différents, il n'est pas nécessaire de retenir leur nombre exact, mais **seulement leur ordre de grandeur** en fonction de la/des taille(s) des entrées.

Exercice 3

Rappeler comment effectuer chacune des opérations ci-dessous en Python, puis dire s'il est raisonnable de considérer ces opérations comme élémentaires :

1. sur les entiers : addition, soustraction, produit, division, exponentiation ;
2. sur les flottants : addition, soustraction, produit, division, exponentiation ;
3. sur les listes : lire la longueur, ajouter un élément en fin de liste, extraire une sous-liste, supprimer un élément de la liste ;
4. sur les dictionnaires : lire le nombre de clés, déterminer si une clé en fait partie, lire la valeur associée à une clé, ajouter une association clé/valeur au dictionnaire, supprimer une clé du dictionnaire.

Remarque Dans les exercices proposés au concours, les opérations élémentaires à décompter seront en général précisées par l'énoncé.

On dit qu'une complexité $C(n)$ est de l'ordre de grandeur de a_n (où (a_n) est une suite « simple ») lorsqu'on peut majorer $C(n)$ par un multiple de a_n. En outre, cela n'aura pas d'importance si cette majoration n'est valable qu'à partir d'un certain rang. Ce concept est formalisé par la notion mathématique de suites dominées :

Définition 4

Soit deux suites réelles $(C(n))_{n\in\mathbb{N}}$ et $(a_n)_{n\in\mathbb{N}}$.
On dit que la suite $(C(n))_{n\in\mathbb{N}}$ est dominée par $(a_n)_{n\in\mathbb{N}}$, et on écrit $C(n) = \mathrm{O}(a_n)$, quand :

$$\exists K \geqslant 0,\ \exists n_0 \in \mathbb{N} \ /\ \forall n \geqslant n_0, \quad |C(n)| \leqslant K\,.\,|a_n|.$$

Remarques

1. Dans cette écriture, $(a_n)_{n\in\mathbb{N}}$ sera toujours une suite de référence, la plus simple possible (voir la liste ci-dessous).
2. Lorsqu'on étudie la complexité, $C(n)$ et a_n sont des réels positifs donc les valeurs absolues sont inutiles.
3. Cette notion de domination a un défaut : si une suite est $\mathrm{O}(n)$, elle est aussi $\mathrm{O}(n^2)$, et aussi $\mathrm{O}(2^n)$... Pour rendre compte de la complexité, on essaie donc de dominer la complexité par la suite **la plus petite possible.**

Vocabulaire 5

Les complexités les plus souvent rencontrées sont les suivantes, des moins coûteuses aux plus coûteuses :

- **complexité bornée** quand $C(n) = \mathrm{O}(1)$
  Exemples : en Python, longueur d'une liste ou d'un dictionnaire [*1], ajout d'un élément à la fin d'une liste, ajout d'une association clé/valeur à un dictionnaire.
- **complexité logarithmique** quand $C(n) = \mathrm{O}(\log n)$ [*2]
  Exemples : recherche dichotomique dans une liste triée, exponentiation rapide.
- **complexité linéaire** quand $C(n) = \mathrm{O}(n)$
  Exemples : somme des éléments d'une liste, évaluation d'un polynôme par la méthode de Hörner, recherche naïve d'un élément dans une liste, exponentiation naïve.
- **complexité quasi-linéaire** quand $C(n) = \mathrm{O}(n \log n)$ ou plus généralement $C(n) = \mathrm{O}(n\,(\log n)^\alpha)$
  Exemple : tri fusion.
- **complexité quadratique** quand $C(n) = \mathrm{O}(n^2)$
  Exemples : tri bulle, tri par insertion ou sélection, évaluation naïve de la valeur d'un polynôme.
- **complexité polynomiale** quand $C(n) = \mathrm{O}(n^k)$ avec $k > 1$
  Exemples : pivot de Gauss / produit de matrices sur des matrices carrées de taille n : en $\mathrm{O}(n^3)$.
- **complexité exponentielle** quand $C(n) = \mathrm{O}(a^n)$ avec $a > 1$
  Exemple : factorisation d'un entier (si n est le nombre de chiffres de l'entier).

∗1. Ces longueurs sont stockées en mémoire dans l'objet, Python ne les recalcule pas à chaque demande.

∗2. Dans cette écriture, la base du logarithme n'a pas d'importance car tous les logarithmes sont proportionnels entre eux : par définition, $\log_b(n) = {}^1/_{\ln b}\,.\ln(n)$. La notation log peut désigner indifféremment ln, $\log_{10}$ ou $\log_2$. Le logarithme binaire $\log_2$ est souvent bien adapté aux problèmes informatiques.

En considérant qu'une opération élémentaire prend un temps moyen de 10 nanosecondes, que la constante K cachée dans le O vaut 1, et en travaillant avec le logarithme binaire, on obtient les temps d'exécutions approximatifs suivants :

	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
$n = 5$	10 ns	23 ns	50 ns	120 ns	250 ns	1,25 µs	320 ns
$n = 10$	10 ns	33 ns	100 ns	330 ns	1 µs	10 µs	10 µs
$n = 20$	10 ns	43 ns	200 ns	860 ns	4 µs	80 ms	10,5 ms
$n = 30$	10 ns	49 ns	300 ns	1,4 µs	9 µs	270 µs	11 s
$n = 40$	10 ns	53 ns	400 ns	2,1 µs	16 µs	640 µs	3,1 h
$n = 50$	10 ns	56 ns	500 ns	2,8 µs	25 µs	1,25 ms	4,3 mois
$n = 100$	10 ns	66 ns	1 µs	6,6 µs	100 µs	10 ms	4.10^{14} ans
$n = 250$	10 ns	80 ns	2,5 µs	20 µs	625 µs	156 ms	—
$n = 1\,000$	10 ns	100 ns	10 µs	100 µs	10 ms	10 s	—
$n = 10^4$	10 ns	130 ns	100 µs	1,3 ms	1 s	2,7 h	—
$n = 10^5$	10 ns	170 ns	1 ms	17 ms	1,7 min	3,8 mois	—
$n = 10^6$	10 ns	200 ns	10 ms	200 ms	2,8 h	316 ans	—

On en tire les constats fondamentaux suivants :

- les algorithmes de complexité logarithmique sont très performants : ils s'exécutent de façon quasi-instantanée même sur des entrées de très grande taille.
- jusqu'à la complexité quasi-linéaire incluse, les algorithmes s'exécutent en un temps raisonnable, même pour des entrées de très grande taille.
- les algorithmes polynomiaux (quadratiques inclus) deviennent inutilisables pour des entrées de grande taille.
- les algorithmes exponentiels sont inutilisables dès que n atteint quelques dizaines.

Exercice 6

Aux élections présidentielles de 2022, on dénombrait 66 571 électeurs inscrits sur les listes électorales à Besançon.

1. Sous les mêmes hypothèses que pour la construction de la table ci-dessus, évaluer le temps nécessaire, dans le pire des cas, pour déterminer si Bertrand Micaux est inscrit sur cette liste :
   - par recherche naïve (algorithme linéaire) ;
   - par recherche dichotomique dans une liste triée (algorithme logarithmique).
   - Calculer le rapport entre ces deux temps.
2. On considère cette fois que la constante K dissimulée dans le O vaut 20.
   Évaluer le temps nécessaire pour trier la liste électorale par date de naissance croissante :
   - par l'algorithme du tri par insertion ;
   - par l'algorithme du tri fusion ;
   - Calculer le rapport entre ces deux temps.

Exercice 7

Si un algorithme de tri « naïf » a pour complexité $C_1(n) = 6\,n^2$ et qu'un algorithme de tri plus « sophistiqué » a pour complexité $C_2(n) = 48\,n \log_2(n)$, quel algorithme choisir en fonction de la taille n de la liste à trier ?
On pourra répondre à la question en s'aidant d'un petit programme en Python.

2 Exemples de calculs de complexité

Calculer la complexité temporelle, dans le pire des cas, d'un algorithme, peut être un problème de difficulté variée. On donne ici quelques repères pour aborder ces problèmes.

Méthode 8

Pour déterminer la complexité temporelle d'un algorithme dans le pire des cas, on procède généralement de la façon suivante :

1. On précise **la/les taille(s) des entrées.**
2. On choisit les **opérations élémentaires** que l'on va dénombrer (en précisant éventuellement quelles sont les opérations que l'on va négliger).
3. On **dénombre** les opérations élémentaires.
   Le but n'est pas nécessairement d'obtenir le nombre exact d'opérations, mais une majoration raisonnable.
   On prêtera particulièrement attention aux points qui suivent :

   Dans le cas d'un algorithme itératif :

   - Les appels de fonctions ou de méthodes n'ont pas toujours un coût borné.
   - Le coût d'un test `if` est majoré par la maximum des coûts de toutes ses branches.
   - Le coût d'une boucle `for` ou `while` est égal à la somme des coûts de ses itérations :
     - si toutes les itérations ont le même coût, ce coût est multiplié par le nombre d'itérations : facile à déterminer pour les boucles `for`, plus délicat pour les boucles `while` ;
     - si ce n'est pas le cas, le coût total s'exprimera à l'aide du signe $\sum$.

   Dans le cas d'un algorithme récursif :
   les mêmes principes s'appliquent, mais le raisonnement conduira à une relation de récurrence.
4. Obtenir une majoration explicite de la complexité : éliminer les éventuels signes $\sum$ et résoudre les relations de récurrence.
5. Trier les termes par ordre décroissant d'importance et ne conserver que le terme prépondérant dans le O.

Exemple fondamental 9

Déterminer la complexité de l'algorithme de calcul de somme :

```
1 def somme(lst: list[float]) -> float:
2     res = 0.0
3     for x in lst:
4         res = res + x
5     return res
```

Solution :

- **Taille du problème :** n, la longueur de la liste `lst`.
- **Opérations élémentaires :** affectation =, addition +
  Opérations non élémentaires : *aucune.*
- **Décompte des opérations élémentaires :**
  - Coût de la boucle (**L3**–**L4**) :
    Chaque itération coûte 2 opérations (**L4**) ;
    il y a n itérations (autant qu'il y a d'éléments dans `lst`, **L3**),
    donc la boucle coûte $2n$ opérations.
  - Avant la boucle, **L2** coûte 1 opération.
  - Au total, la complexité temporelle est :

$$C(n) = 2n + 1 = \mathrm{O}(n).$$

La fonction `somme` a donc une **complexité linéaire.**

Exemple fondamental 10

Déterminer la complexité de l'algorithme de tri par sélection :

```
1  def tri_selection(lst: list) -> None:
2      n = len(lst)
3      for i in range(0, n):
4          # inv: lst[0:i] contient les i plus petits éléments de lst
5          #      triés par ordre croissant
6
7          # chercher le minimum de lst[i:n]
8          pos_min = i
9          for j in range(i+1, n):
10             if lst[j] < lst[pos_min]:
11                 pos_min = j
12
13         # échanger lst[i] et ce minimum
14         lst[i], lst[pos_min] = lst[pos_min], lst[i]
```

Solution :

- **Taille du problème :** n, la longueur de la liste `lst`.
- **Opérations élémentaires :** affectation `=`, comparaison `<`, lecture d'un élément de la liste `lst[...]`
  Opérations non élémentaires : *aucune.*
- **Décompte des opérations élémentaires :**
  - *Coût de la boucle interne* (**L9**–**L11**) :
    Chaque itération coûte **au plus** 4 opérations (**L10**-**L11** : 2 lectures, 1 comparaison, 1 affectation) ;
    il y a $n-(i+1)$ itérations (**L9**)
    donc la boucle interne coûte au plus $4(n-1-i)$ opérations.
  - *Coût de la boucle externe* (**L3**–**L14**) :
    Chaque itération coûte $4(n-1-i)+5$ opérations : la boucle interne, 2 lectures et 3 affectations (**L8** et **L14** ; **attention ! ce coût dépend de la valeur de** i**.**)
    D'après **L3**, i varie de 0 à $n-1$. Le coût total de la boucle est donc :

$$\begin{aligned} C_1(n) &\leqslant \sum_{i=0}^{n-1} \Big[\, 4(n-1-i)+5 \,\Big] \\ &= 4\sum_{i=0}^{n-1}(n-1-i) + 5\sum_{i=0}^{n-1} 1 \end{aligned}$$

Dans la deuxième somme, on somme un terme constant.
Dans la première somme, on effectue le changement d'indice $j = n-1-i$: quand $i=0$, $j=n-1$, et quand $i=n-1$, $j=0$.

$$\begin{aligned} C_1(n) &\leqslant 4\sum_{j=0}^{n-1} j + 5\big((n-1)-0+1\big) \\ &= 4\big((n-1)-0+1\big)\frac{0+(n-1)}{2} + 5n \\ &= 2n(n-1)+5n \\ &= 2n^2+3n. \end{aligned}$$

- *Coût total :* il reste à ajouter l'affectation **L2** :

$$C(n) \leqslant 1 + C_1(n) = 2n^2+3n+1 = \mathrm{O}(n^2).$$

La fonction `tri_selection` a donc une **complexité quadratique.**

Exemple fondamental 11

On rappelle une mise en œuvre possible de l'algorithme d'exponentiation rapide :

```
1  def expo_rapide(a: int, n: int) -> int:
2      # précond: n >= 0
3      b, p, res = a, n, 1
4      while p > 0:
5          # inv: b**p * res == a**n
6          if p % 2 == 1:
7              res = res * b
8          b, p = b**2, p//2
9
10     # postcond: res == a**n
11     return res
```

On suppose pour simplifier que toutes les opérations arithmétiques ont un coût unitaire.

1. Nous avons démontré précédemment que cette fonction terminait, et qu'elle était correcte. À l'aide de quelle notion prouve-t-on la terminaison ? la correction ?

On note q_n le nombre d'itérations de la boucle pour la valeur n,
et p_k la valeur de la variable p à la fin de la k^{e} itération de la boucle ($k \in [\![0, q_n]\!]$).

2. Donner un majorant du nombre d'opérations élémentaires effectuées dans le pire des cas par la boucle (**L4**–**L8**).
3. Montrer que : $p_k \leqslant \dfrac{n}{2^k}$ pour tout $k \in [\![0, q_n]\!]$.
4. Quand la boucle effectue au moins une itération, que vaut p_{q_n-1}, la valeur de p à la fin de l'avant-dernière itération ?
5. En déduire un majorant de q_n en fonction de n, puis la complexité de l'algorithme.

Solution :

1. La **terminaison** de la boucle se prouve à l'aide du **variant** p : cette expression est à valeurs entières, strictement décroissantes et minorées par 0.
   La **correction** de la fonction se prouve à l'aide de l'**invariant de boucle** qui est rappelé **L5** : $b^p \times res = a^n$. Cet invariant se prouve par récurrence.
2. Chaque itération de la boucle (**L4**–**L8**) coûte :
   - 2 comparaisons (**L4** et **L6**),
   - au plus 4 opérations arithmétiques (**L6**, **L7**, **L8**),
   - au plus 3 affectations (**L7**, **L8**).

   Chaque itération coûte au plus 9 opérations.
   Par définition, la boucle effectue q_n opérations.
   Le coût de la boucle vérifie donc $C_1(n) \leqslant 9\,q_n$**.**
3. On procède par **récurrence finie** sur $k \in [\![0, q_n]\!]$:
   - **Initialisation :** pour $k = 0$, $p_0 = n$ d'après **L3** et $n/2^0 = n$ donc on a bien $p_0 \leqslant n/2^0$.
   - **Hérédité :** soit $k \in [\![0, q_n[\![$ fixé.
     Supposons que $p_k \leqslant n/2^k$,
     et montrons que $p_{k+1} \leqslant n/2^{k+1}$:

$$\begin{aligned} p_{k+1} &= \left\lfloor \frac{p_k}{2} \right\rfloor && \text{(d'après \textbf{L8})} \\ &\leqslant \frac{p_k}{2} && \text{(propriété de } \lfloor\cdot\rfloor) \\ &\leqslant \frac{n/2^k}{2} && \text{(hyp. de récurrence)} \\ &= \frac{n}{2^{k+1}}. \end{aligned}$$

   La propriété $p_k \leqslant \dfrac{n}{2^k}$ est initialisée et héréditaire :
   elle est donc vraie pour tout rang $k \in [\![0, q_n]\!]$.
4. Quand la boucle s'arrête, c'est que p contient la valeur 0 : $p_{q_n} = 0$. Ce 0 est le quotient de la division de la valeur précédente de p par 2 (**L8**). Cette valeur précédente vaut donc 0 ou 1 ; mais cela ne peut pas être 0 car sinon la boucle se serait arrêtée avant !
   En conclusion : si $q_n \geqslant 1$, alors $p_{q_n-1} = 1$.
5. La question 3 appliquée à $q_n - 1$ donne :

$$p_{q_n-1} \leqslant \frac{n}{2^{q_n-1}} \quad \text{donc, d'après Q4 :} \quad 1 \leqslant \frac{n}{2^{q_n-1}}.$$

   On en déduit :

$$\begin{aligned} 2^{q_n-1} &\leqslant n && (\times 2^{q_n-1} \geqslant 0) \\ q_n - 1 &\leqslant \log_2(n) && (\log_2 \text{ croissante sur } \mathbb{R}_+^*) \\ q_n &\leqslant \log_2(n) + 1. \end{aligned}$$

   En reprenant Q2, on obtient :

$$C_1(n) \leqslant 9\big(\log_2(n) + 1\big),$$

   et en prenant en compte le coût de **L3**, on peut majorer le coût total :

$$C(n) \leqslant 9\big(\log_2(n) + 1\big) + 3.$$

   Les constantes étant négligeables devant le logarithme, on conclut que $C(n) = \mathrm{O}(\log_2 n)$ **: l'algorithme d'exponentiation rapide est de complexité logarithmique.**

Exemple fondamental 12

On donne une implémentation du tri fusion :

```python
def fusion_listes_triees(lst1: list, lst2: list) -> list:
    """ Fusionne deux listes triées en une seule liste triée.
        Les listes initiales ne sont pas modifiées.
    """
    n1, n2 = len(lst1), len(lst2)
    pos1, pos2 = 0, 0  # rang du prochain élément à placer dans chaque liste
    res = []

    # tant qu'il reste au moins un élément à placer ...
    while pos1 < n1 or pos2 < n2:
        # si `lst1` est épuisée,
        # ou que l'élément de `lst1` est inférieur ou égal à celui de `lst2` ...
        if pos2 == n2 or (pos1 < n1 and lst1[pos1] <= lst2[pos2]):
            # l'élément de `lst1` part dans le résultat
            res.append(lst1[pos1])
            # on passe à l'élément suivant dans `lst1`
            pos1 = pos1 + 1
        else:  # dans tous les autres cas ...
            # l'élément de `lst2` part dans le résultat
            res.append(lst2[pos2])
            # on passe à l'élément suivant dans `lst2`
            pos2 = pos2 + 1

    # retour du résultat
    return res

def tri_fusion(lst: list) -> list:
   """ Trie une liste par ordre croissant par l'algorithme du tri fusion.
       La liste initiale n'est pas modifiée, et la liste triée est retournée.
   """
   n = len(lst)
   if n <= 1:  # cas de base: liste vide ou singleton
       return lst.copy()  # simple copie de la liste fournie
   else:  # cas récursif
      # découpage de `lst` en deux fragments à peu près de même longueur
      lst_g, lst_d = lst[0 : n//2], lst[n//2 : n]
      # tri récursif de chaque fragment
      lst_g = tri_fusion(lst_g)
      lst_d = tri_fusion(lst_d)
      # fusion des deux fragments triés
      res = fusion_listes_triees(lst_g, lst_d)
      # retour du résultat
      return res
```

On admet que la fonction `fusion_listes_triees` termine, qu'elle est correcte et que sa complexité est donnée par :

$$C_{\text{fus}}(n_1, n_2) \leqslant a\,(n_1 + n_2),$$

où $a \geqslant 1$ est une constante, n_1 est la longueur de `lst1`, et n_2 la longueur de `lst2`.

Le but de cet exercice est de prouver la terminaison de la fonction `tri_fusion` et d'obtenir sa complexité C_{tri} en fonction de n, la longueur de `lst`.

1. Les fonctions `fusion_listes_triees` et `tri_fusion` sont-elles des fonctions ? des procédures ? Pourquoi ?

2. *Exemple.* Dessiner l'arbre des appels récursifs déclenchés par l'appel de :

```
tri_fusion([54, 21, 43, 10, 32]).
```

   Dessiner ensuite l'arbre des réponses remontées par `tri_fusion`.
   Cet arbre a la même structure que l'arbre des appels. Il se complète en commençant par ses feuilles, qui correspondent aux cas de base. On applique ensuite la fonction `fusion_listes_triees` *pour remonter progressivement vers la racine de l'arbre, qui donne la réponse finale.*

Dans l'arbre des appels récursifs, on décide que :

- l'appel initial est de niveau 0 ;
- les appels récursifs effectués depuis un appel de niveau k sont de niveau $k+1$.

3. *Sur l'exemple,* repérer les appels de niveau 0, 1, 2, etc. Quel est le niveau maximal observé sur cet exemple ? Observer les tailles des listes à chaque niveau.
4. *Dans le cas général,* pour chaque niveau k, majorer en fonction de n et de k :
   - le nombre d'appels ν_k de niveau k ;
   - la taille t_k des listes observées au niveau k.
5. Déduire de la question précédente que la fonction `tri_fusion` termine, puis majorer le niveau maximal p_n en fonction de n.
   Indications : si le niveau k est présent dans l'arbre, il contient au moins une liste de taille $\geqslant 1$. Vous trouverez que p_n est majorée par un $O(\log n)$.
6. Expliquer pourquoi, dans l'arbre des résultats, le coût pour remonter du niveau k au niveau $(k-1)$ est majoré par $a\,n$. En déduire un majorant du coût total de toutes les fusions.
7. De même, majorer le coût des découpages pour passer du niveau $(k-1)$ au niveau k par un multiple de n, puis le coût total de tous les découpages.
8. Majorer enfin le coût total de la fonction `tri_fusion`.