Prouver un algorithme, c'est démontrer deux choses :

- prouver que l'algorithme se termine bien dans tous les cas ; c'est ce que l'on appelle la **terminaison** de l'algorithme.
- prouver que le résultat de l'algorithme est bien la réponse souhaitée ; c'est ce que l'on appelle la **correction** de l'algorithme.

I Terminaison de l'algorithme

Pour établir la terminaison d'un algorithme, on utilise un **variant de boucle** : c'est une suite d'entiers naturels (donc positifs) qui décroît **strictement** après chaque itération.
Pour illustrer les notions de ce chapitre, nous utiliserons les trois algorithmes suivants qui recherchent un élément `a` dans une liste `L` (supposée triée dans l'ordre croissant pour le troisième algorithme) :

Recherche en parcourant le tableau complet

```
def recherche1(L,a):
    trouve = False
    n = len(L)
    for k in range(n):
        if a == L[k]:
            trouve = True
    return trouve
```

Recherche en parcourant le tableau jusqu'à ce que l'on ait trouvé l'élément

```
def recherche2(L,a):
    k = 0
    trouve = (a == L[0])
    n = len (L)
    while not (trouve) and k <= n-1 :
        if a == L[k] :
            trouve = True
        k += 1
return trouve
```

Recherche dichotomique dans un tableau trié

```
def rechercheDicho(L,a):
    if a < L[0] or a > L[-1] :
        return False
    n = len(L)
    gauche = 0
    droite = n-1
    milieu = (gauche + droite)//2
    trouve = False
    while gauche <= droite and not (trouve
        ):
        elt = L[milieu]
        if a == elt :
            trouve = True
        elif a > elt :
            gauche = milieu + 1
        else :
            droite = milieu - 1
        milieu = (gauche + droite)//2
    return trouve
```

— Dans la fonction `recherche1` (comme dans toute boucle `for`), si k est le nombre de fois où la boucle a été effectuée, l'entier $n-k$ est un variant de boucle. On est donc sûr que ce genre de boucle se termine bien.

— Dans la fonction `recherche2`, la détermination d'un variant de boucle est nécessaire de façon à vérifier que la boucle conditionnelle `while` se termine bien : on peut vérifier, grâce à l'incrémentation de la variable k dans la boucle, que $n-k$ est un variant de boucle.

— Dans la fonction `rechercheDicho`, si on note d_k et g_k les valeurs des entiers `gauche` et `droite` lorsque la boucle conditionnelle `while` a été effectuée k fois, et $u_k = d_k - g_k \in \mathbb{N}$, on vérifie que u_k est un variant de boucle : le `milieu` dans la boucle k est $m_k = \dfrac{g_k + d_k}{2} - \varepsilon_k$ avec $\varepsilon_k \in \left\{0, \dfrac{1}{2}\right\}$ (selon la parité de $g_k + d_k$). On a alors, selon les cas, $u_{k+1} = \begin{cases} d_k - m_k - 1 = \dfrac{u_k}{2} - \varepsilon_k - 1 \\ \text{ou} \\ m_k - 1 - g_k = \dfrac{u_k}{2} + \varepsilon_k - 1 \end{cases}$ donc $u_{k+1} \in \left\{\dfrac{u_k}{2} - \dfrac{3}{2}, \dfrac{u_k}{2} - 1, \dfrac{u_k}{2} - \dfrac{1}{2}\right\}$.
On a donc $u_{k+1} \leqslant \dfrac{u_k}{2} - \dfrac{1}{2}$ et $u_{k+1} - u_k \leqslant -\dfrac{u_k}{2} - \dfrac{1}{2}$.
Enfin, comme la boucle conditionnelle `while` n'est effectuée que si $u_k \geqslant 0$ (et si l'élément n'a pas encore été trouvé), on a bien dans tous les cas, $u_{k+1} < u_k$ donc l'algorithme se termine.

II Correction d'un algorithme

Pour prouver la correction d'un algorithme, on utilise un **invariant de boucle** : c'est une propriété, dépendant des différentes variables de la boucle, qui est satisfaite à chaque boucle et dont l'expression en sortie de la dernière boucle effectuée implique le résultat attendu.

Pour trouver un invariant de boucle on part donc du résultat que doit obtenir l'algorithme et on procède par récurrence.

— Dans la fonction `recherche1`, un invariant de boucle est la propriété $\mathcal{P}(k)$: « il existe $p \leqslant k$ tel que `a=L[p]` ou `trouve=False` ».
En effet, la propriété $\mathcal{P}(0)$ est bien vérifiée puisqu'à l'issue de la première boucle ($k = 0$) soit `L[0]=a` soit `trouve=False`.
Si on suppose la propriété $\mathcal{P}(k)$ vérifiée à un rang k alors deux possibilités se présentent : si `L[k+1]=a` alors $\mathcal{P}(k+1)$ sera vérifiée, sinon la propriété $\mathcal{P}(k)$ assurera bien que $\mathcal{P}(k+1)$ est vérifiée aussi.
La propriété $\mathcal{P}(n-1)$ (à la fin de la dernière boucle) est conforme au résultat attendu : si `a` est dans `L`, `trouve=True` et sinon `trouve=False`.

— Dans la fonction `recherche2`, un invariant de boucle est $\mathcal{P}(k)$: « $\forall p \in [\![0, k]\!]$, `L[p]` $\neq$ `a` ou `trouve=True` ».
En effet, $\mathcal{P}(0)$ est vraie et si on suppose $\mathcal{P}(k)$ satisfaite et si la boucle d'indice $k+1$ est effectuée, alors `trouve=False` donc $\forall p \in [\![0, k]\!]$, `L[p]` $\neq$ `a` et $P(k+1)$ sera vraie, selon que `L[k+1]=a` ou non.
En sortie de boucle, on a soit `trouve=True`, soit $k = n-1$ et `trouve=False` et dans ce cas, la propriété $\mathcal{P}(n-1)$ assure bien que l'élément `a` n'est pas dans la liste `L`.

— Dans `rechercheDicho`, un invariant de boucle est « soit `a` n'a pas été trouvé et `a`$\in [$`L`$[g_k]$, `L`$[d_k]]$, soit `trouve=True` », dans le cas où le test initial est passé.
Cette propriété reste vraie si on exécute la boucle suivante. A la sortie de la boucle, soit `a` a été trouvé et `trouve=True`, soit $m_k = d_k = g_k$ et `a` n'est pas dans `L` et `trouve=False`.

III Un petit exercice

On considère la fonction dont le code Python est le suivant :

```
def fct(chaine) :
    n = len(chaine)
    k,fini = 0,False
    while k < n-1 and not fini :
        car = chaine[k]
        p = n-1
        while k < p and not fini :
            if car == chaine[p] :
                fini = True
            p -= 1
        k += 1
    return fini
```

1. Déterminer le rôle de cette fonction après l'avoir testée sur les deux chaînes `'psi1et2'` et `'psiunetdeux'` (on précisera l'état des différentes variables au cours de l'exécution.
2. Prouver la terminaison et la correction de cet algorithme.

IV Complexité : définitions

La complexité d'un algorithme est un moyen de mesurer son efficacité : il s'agit d'estimer le coût de son exécution, soit en terme de temps de calcul (on parle de complexité temporelle), soit en terme d'utilisation de la mémoire (on parle de complexité spatiale).
Pour mesurer la complexité d'un algorithme, il faut d'abord déterminer les opérations fondamentales que l'on souhaite dénombrer : les opérations effectuées (addition, multiplication, ...), les tests (==, !=, ...), les accès à des données (lecture des données d'une liste, d'un tableau, ...), la création d'espaces en mémoire, ...
On peut remarquer que **la complexité peut aussi être définie comme le nombre de boucles effectuées**, ce qui ne change pas son ordre de grandeur le plus souvent, le nombre d'opérations effectuées à l'intérieur de chaque boucle est constant (ou au moins majoré).
Notons n le nombre de données fournies à l'algorithme (le nombre d'éléments d'un tableau par exemple). La complexité,

calculée en comptant les boucles, est du même ordre de grandeur que celle calculée en comptant le nombre d'accès au tableau.
Pour comparer la complexité des algorithmes, on utilise le vocabulaire suivant :

Soient (a_n) et (b_n) deux suites réelles positives

1. On dit que (a_n) **majore** (b_n), ou que (b_n) est **un grand** O de (a_n), noté $b_n = O(a_n)$, si

$$\exists A > 0, \exists N \in \mathbb{N}, \forall n \geqslant N, b_n \leqslant A a_n$$

2. On dit que (a_n) **minore** (b_n), ou que (b_n) est **un grand** Ω de (a_n), noté $b_n = \Omega(a_n)$ si

$$\exists B > 0, \exists N \in \mathbb{N}, \forall n \geqslant N, B a_n \leqslant b_n$$

3. On dit que (a_n) et (b_n) sont **du même ordre**, ou que (b_n) est **un grand** Θ de (a_n), noté $b_n = \Theta(a_n)$ si

$$\exists A > 0, \exists B > 0, \exists N \in \mathbb{N}, \forall n \geqslant N, B a_n \leqslant b_n \leqslant A a_n$$

La définition ne fait intervenir que le comportement pour des valeurs de n suffisamment grandes (comportement asymptotique). Une complexité en $\Theta(n)$ signifie que pour des « grandes »valeurs de n, le nombre d'opérations double quand le nombre de données double, alors qu'il est multiplié par 4 pour une complexité en $\Theta(n^2)$. Pour illustrer les notions de complexité, nous réutiliserons les trois algorithmes précédents de recherche d'un élément dans une liste.

V Complexité dans le meilleur et dans le pire des cas

Supposons qu'un tableau `T` soit rempli avec des entiers de $[\![1, p]\!]$ et que l'on cherche, à l'aide d'un algorithme `recherche`, si un élément `a` appartient au tableau ou non. On considérera comme opération fondamentale tout accès au tableau `T` (toute utilisation de `T[k]`).
Si note $E_{n,p}$ l'ensemble des tableaux à n éléments dans $[\![1, p]\!]$ et $C(\texttt{T})$ le nombre d'opérations fondamentales faites lors de l'appel de l'algorithme `recherche` sur le tableau `T`, on définit la complexité dans le meilleur des cas et la complexité dans le pire des cas par les formules :

$$C_{\text{pire}}(n) = \max_{\texttt{T} \in E_{n,p}} C(\texttt{T}) \quad \text{et} \quad C_{\text{meilleur}}(n) = \min_{\texttt{T} \in E_{n,p}} C(\texttt{T})$$

— Cas de la fonction `recherche1` : comme dans cette fonction le tableau est toujours parcouru dans son intégralité, on a $C_{\text{pire}}(n) = C_{\text{meilleur}}(n) = n$.

— Cas de la fonction `recherche2` : le meilleur des cas se produit lorsque l'élément `a` se trouve dans le tableau à l'indice 0 (la boucle conditionnelle n'est alors jamais exécutée) alors que le pire des cas correspond au cas où l'élément `a` n'est pas dans le tableau (ou s'il est à la dernière position), le tableau étant alors parcouru en entier. On a donc $C_{\text{pire}}(n) = n - 1$ et $C_{\text{meilleur}}(n) = 0$.

— Cas de la recherche dichotomique (on suppose donc que le tableau `T` est trié dans l'ordre croissant et que l'on recherche un élément `a` compris entre les deux valeurs extrêmes du tableau, on pourra donc supprimer le premier test de la fonction) : dans le meilleur des cas, l'élément `a` est trouvé dès le premier test et dans ce cas la boucle conditionnelle n'est effectuée qu'une fois. On a donc $C_{\text{meilleur}}(n) = 1$.
Dans le pire des cas, l'élément n'appartient pas au tableau, la boucle est alors exécutée jusqu'à ce que les paramètres vérifient `gauche=droite`. À une étape donnée, un tableau de longueur n est divisé (en ne tenant pas compte du milieu du tableau qui est en fait écarté de la recherche par la suite) en deux sous-tableaux de longueurs $\left\lfloor \frac{n}{2} \right\rfloor$ et $\left\lceil \frac{n}{2} \right\rceil$ et le choix du tableau à examiner nécessite un accès au tableau, on a donc

$$C_{\text{pire}}(n) = 1 + \max\left\{ C_{\text{pire}}\left(\left\lceil \frac{n}{2} \right\rceil\right), C_{\text{pire}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right\}.$$

Si on définit la suite $(u_n)_{n\in\mathbb{N}}$ par $u_1 = 1$ et, pour $n \geqslant 1$, $u_{n+1} = 1 + \max\left\{u_{\lceil \frac{n}{2} \rceil}, u_{\lfloor \frac{n}{2} \rfloor}\right\}$.
On aura alors, dans le cas particulier où n est une puissance de 2, $u_{2^{k+1}} = 1 + u_{2^k} = 2 + u_{2^{k-1}} = \cdots = k + 2$.
Dans le cas général, il existe un entier k tel que $2^k \leqslant n < 2^{k+1}$. Or on peut vérifier que la suite (u_n) est croissante : on prouve par récurrence sur n que $u_{n+1} \geqslant u_n$ à partir de la relation entre u_n, $u_{\lceil \frac{n}{2} \rceil}$ et $u_{\lfloor \frac{n}{2} \rfloor}$).
On a donc $k + 1 = u_{2^k} \leqslant u_n \leqslant u_{2^{k+1}} = k + 2$. Comme $k = \lfloor \log_2(n) \rfloor$, on a $\log_2(n) + 1 \leqslant u_n \leqslant \log_2(n) + 2$ donc $u_n = \Theta(\log_2(n))$ et $C_{\text{pire}}(n) = O(\log_2(n))$
Cette méthode est donc nettement plus rapide que les deux autres.

Les algorithmes de recherche dichotomique ont une complexité est en $O(\log_2(n))$.

En revanche, le tableau doit être trié au préalable, ce qui n'est pas négligeable en terme de complexité d'algorithme.

VI Complexité en moyenne

Le nombre de tableaux à n éléments dans $[\![1,p]\!]$ est p^n. On suppose que la probabilité d'obtenir un tableau `T` dans $E_{n,p}$ est uniforme, ie $P(\texttt{T}) = \dfrac{1}{p^n}$. La complexité en moyenne est définie par

$$C_{\text{moy}}(n) = \sum_{\texttt{T}\in E_{n,p}} C(\texttt{T})P(\texttt{T})$$

Ainsi, la fonction C étant une variable aléatoire sur $E_{n,p}$, la complexité en moyenne est l'espérance de la variable C :

$$C_{\text{moy}}(n) = \sum_{i\in C(E_{n,p})} i \times P(C=i)$$

On a bien sûr $C_{\text{moy}}(n) \subset [\![C_{\text{meilleur}}(n), C_{\text{pire}}(n)]\!]$.

— Cas de la fonction `recherche1` : on a $C_{\text{moy}}(n) = C_{\text{meilleur}}(n) = C_{\text{pire}}(n) = n$.

— Cas de la fonction `recherche2` :
On étudie les différentes valeurs possibles de $C(\texttt{T})$: on a $C(\texttt{T}) = 1$ si l'élément `a` est le premier élément du tableau. Le nombre de tableaux vérifiant cette propriété est p^{n-1} donc $P(C=1) = \dfrac{p^{n-1}}{p^n} = \dfrac{1}{p}$.
Pour $i \in [\![2, n-1]\!]$, on a $C(\texttt{T}) = i$ si $\texttt{T}[i-1]$=`a` (donc $\texttt{a} \in [\![1,p]\!]$) et si $\texttt{T}[k] \in [\![1,p]\!] \setminus \{\texttt{a}\}$ pour $k \in [\![0, i-2]\!]$; on a alors $p-1$ choix possible pour les éléments d'indices $\leqslant i-2$, les éléments d'indices $\geqslant i$ (il y en a $n-i$) sont quelconques. On en déduit $P(C=i) = \dfrac{(p-1)^{i-1} \times p^{n-i}}{p^n} = \dfrac{(p-1)^{i-1}}{p^i}$ (ce qui est valable pour $i=1$).
Enfin, on aura $C = n$ soit si `a` n'est pas un élément de `T`, qui est alors un tableau à n éléments dans $[\![1,p]\!] \setminus \{\texttt{a}\}$, soit si `a` est le dernier élément de `T` (et seulement celui ci).
On a alors $P(C=n) = \dfrac{(p-1)^n + (p-1)^{n-1}}{p^n} = \dfrac{(p-1)^{n-1}}{p^n} + \left(\dfrac{p-1}{p}\right)^n$. On en déduit :

$$C_{\text{moy}}(n) = n\left(\frac{p-1}{p}\right)^n + \sum_{i=1}^{n} \frac{i}{p}\left(1-\frac{1}{p}\right)^{i-1}$$

Comme, pour $x \neq 1$, on a $\displaystyle\sum_{i=1}^{n} x^i = \left(\sum_{i=0}^{n} x^i\right) - 1 = \frac{1-x^{n+1}}{1-x} - 1$, on obtient en dérivant par rapport à x :
$\displaystyle\sum_{i=1}^{n} i x^{i-1} = \frac{1 + x^n(nx-n-1)}{(1-x)^2}$ puis en prenant $x = 1 - \dfrac{1}{p}$, tous calculs faits, on trouve :

$$C_{\text{moy}}(n) = p\left[1 - \left(1-\frac{1}{p}\right)^n\right] \xrightarrow[p\to+\infty]{} n$$

La complexité en moyenne tend donc vers la complexité dans le pire des cas lorsque la taille des éléments du tableau devient grande (pour un tableau de longueur n fixée).

VII La suite du petit exercice

3. On définit la complexité comme le nombre de boucles effectuées lors de l'appel de la fonction sur une chaîne de longueur n. Déterminer les complexités de cette fonction dans le meilleur et dans le pire des cas (on supposera que la chaîne de caractères ne contient que des lettres non accentuées).