

1^{ère} NSI — Thème 6: Algorithmique

Algorithmique & Mise au Point de Programmes

Lycée Fustel de Coulanges, Massy

Marc Biver, février 2024, *v0.3*

Ce document reprend les notions abordées en cours et pratiquées en TP / TD; il inclut la totalité de ce qui a été diffusé en classe sur ce thème. Ses deux principales sources d'inspiration sont, d'une part, le cours mis en ligne par Charles Poulmaire et Pascal Remy¹ et, d'autre part, le manuel de NSI en 1^{ère} édité chez Ellipses². Si vous avez des questions à son propos n'hésitez pas à me contacter par le biais de la messagerie de l'ENT.

Document de cours incluant les réponses aux exercices.

1. Accessible [ici](#).

2. Numérique et sciences informatiques, Spécialité NSI 1re: 30 leçons avec exercices corrigés, par Thibaut Balabonski, Sylvain Conchon, Jean-Christophe Filliâtre, et Kim Nguyen.

Table des matières

1	Point d'étape – où est-on / où va-t-on ?	3
1.1	Ce qu'on a couvert jusqu'à présent	3
1.2	Ce dont on va parler dans ce nouveau chapitre	3
1.3	Comment on va procéder	4
1.4	Comment réviser / préparer les contrôles ?	4
2	Introduction à la conception d'algorithmes	5
2.1	Écrire un algorithme : spécification	6
2.2	Ecrire un algorithme : pseudo-code	8
2.3	Tester un algorithme	13
3	Preuves d'algorithmes	15
3.1	Terminaison : variant de boucle	15
3.2	Correction partielle : invariant de boucle	17
4	Complexité d'algorithmes	22
4.1	Cadre théorique	22
4.2	Exercices d'application	22
4.3	Principes d'estimation de la complexité	26
4.4	Et dans la vraie vie... ?	28
4.5	Complexité - à retenir	30
5	Algorithmes de tri	31
5.1	Pourquoi trier ?	31
5.2	Le tri par permutation ou "tri à bulles"	31
5.3	Le tri par sélection	33
5.4	Tri par insertion	36
5.5	Comparaisons entre ces algorithmes...	39
5.6	Tris fournis par Python	42
5.7	Algorithmes de tri – à retenir	42
6	Algorithmes de recherche	43
6.1	Cas général : recherche séquentielle	43
6.2	Recherche dichotomique	44
7	Algorithmes gloutons	49

1 Point d'étape – où est-on / où va-t-on ?

1.1 Ce qu'on a couvert jusqu'à présent

- Rudiments de l'architecture physique d'un ordinateur – le modèle de Von Neumann.
- Mise en jambes sur de l'écriture de code : réalisation d'une page Web en HTML.
- Introduction à Python, et plus spécifiquement :
 - Ce qu'on appelle ses "constructions élémentaires" – variables, fonctions, conditions & embranchements, boucles...
 - Les types et valeurs de base : entiers (naturels et relatifs), flottants (réels), chaînes de caractères et booléens (qu'on n'a que brièvement abordés pour l'instant).
 - Un type dit "construit" – les listes.
- Un peu de théorie : la représentation des types et valeurs de base en machine (les entiers naturels et relatifs, les réels, les alphanumériques).
- Un peu plus de théorie : introduction à la logique booléenne (que l'on n'a couverte que très rapidement – on y reviendra en fin d'année).
- Un retour à la pratique : le traitement de données en table, la manipulation de fichiers dans Python, et des types de données plus complexes — les dictionnaires, les listes de dictionnaires...

1.2 Ce dont on va parler dans ce nouveau chapitre

On a donc fait des "sauts" de la théorie vers la pratique, puis vers la théorie de nouveau — pour finalement atterrir ici, dans ce chapitre sur l'algorithmique qui se trouve presque exactement au "milieu" de cet axe théorie–pratique.

Comme je vous l'ai dit à plusieurs reprises dans les parties précédentes – et spécialement dans celle portant sur le traitement de données en table, l'algorithmique, vous en faites déjà : je vous pose des problèmes par le biais de notebooks Jupyter dans Cappytale v2 et vous réfléchissez à comment les résoudre. Dit autrement, *vous concevez des algorithmes*.

Ce que l'on va faire dans ce nouveau chapitre c'est formaliser cette démarche, la structurer, puis étudier quelques exemples d'algorithmes beaucoup plus avancés que ce que l'on a vu jusqu'à présent.

Spécifiquement, on va parcourir le chemin suivant :

- Introduction à l'algorithmique – étapes de conception et de rédaction d'un algorithme ;
- Pratique : tests d'algorithmes / de programmes ;
- Preuve d'algorithmes ;
- Complexité d'algorithmes ;
- Etude de certains algorithmes spécifiques :
 - Algorithmes de tri - par sélection, par insertion ;
 - Algorithme de recherche dichotomique dans un tableau ;
 - Algorithmes gloutons ;
 - Algorithmes des k plus proches voisins – algorithme d'apprentissage, "machine learning".

- Pratique : mise au point de programmes ; programmation défensive.

1.3 Comment on va procéder

Comme dit plus haut, on est ici à la frontière entre la théorie et la pratique – on va donc avoir un fonctionnement hybride en classe :

- **Prise de notes essentielle** — vous commencez à connaître les cours que je vous fournis, ils sont *très* longs. Ils doivent vous servir de référence, vous permettre surtout de bien revoir les corrections d'exercices, mais votre savoir, lui, doit venir de votre prise de notes ;
- Plusieurs exercices sur papier qu'on fera en classe et dont il sera très important que vous gardiez une trace ;
- En parallèle et en complément, quelques applications / exercices sur machine.

1.4 Comment réviser / préparer les contrôles ?

Comme pour les autres chapitres de cours de NSI, les deux choses les plus importantes que vous devez acquérir pour être en mesure de bien réussir les contrôles sont :

- Connaître et comprendre les notions présentées dans le cours (et en particulier les définitions) ;
- Être capable de faire tous les exercices présents dans ce cours et également dans le cahier d'exercices qui l'accompagne.

Plus spécifiquement, chaque section de ce cours se termine par un encart (sur fond vert) intitulé "À Savoir \rightleftharpoons À Réviser" dans lequel je vous explique plus en détails ce qu'il y a à retenir de la section en question.

En tout état de cause mon conseil principal pour les révisions est de prendre la version de ce cours et du cahier d'exercices qui ne contient ***pas*** les solutions, vous exercer à faire les exercices qui sont dedans, puis vérifier les réponses dans la version qui contient les solutions.

Bon courage !

2 Introduction à la conception d'algorithmes

Quelques questions pour commencer...

→ Qu'est-ce qu'un algorithme (indice : c'est constitué de trois parties) ?

Un algorithme est une **suite finie d'instructions** permettant de **résoudre un problème**. Il est structuré en trois parties :

- a. L'entrée des données ;
- b. Le traitement des données ;
- c. La sortie des données

→ Parmi les éléments suivants, qu'est-ce qui est un algorithme, qu'est-ce qui n'en est pas ?

- a. Une recette de gâteau au chocolat ;
- b. La liste des présidents de la V^{ème} République ;
- c. Les règles du jeu d'échecs ;
- d. Les règles à appliquer pour résoudre une équation du 2nd degré ;
- e. Les instructions de montage d'un meuble Ikea.

- a. Oui c'en est un – ingrédients en entrée, étapes de confection, gâteau en sortie. C'est bien la résolution d'un problème (en l'occurrence "comment fabriquer un gâteau au chocolat ?").
- b. Non ce n'en est pas un – c'est une liste d'informations, pas des étapes à suivre.
- c. Ce n'en est pas un non plus – ce n'est qu'une liste de principes. En revanche on pourrait les utiliser pour mettre en œuvre un algorithme qui joue aux échecs (et résoud le problème "comment jouer – et gagner – une partie d'échecs ?).
- d. Oui c'en est un – c'est même écrit dans la description ("résoudre").
- e. Oui c'en est un également.

Donc, pour reprendre :



DÉFINITION: algorithme

Un algorithme est une **suite finie d'instructions** permettant de **résoudre un problème**. Il est structuré en trois parties :

- a. L'entrée des données ;
- b. Le traitement des données ;
- c. La sortie des données



DÉFINITION: l'algorithme

L'algorithme est :

- La conception (et la production) d'algorithmes ;
- Leur étude – leur analyse, la mesure de leur fiabilité (est-ce qu'il répond vraiment au problème posé ?) et de leur efficacité (est-ce qu'il le fait en un temps acceptable ?).

Nous allons dans ce cours nous intéresser à ces deux composantes, en commençant, dans ce chapitre, par la première dont les étapes peuvent se résumer ainsi :

1. Énoncé d'un problème à résoudre ;
2. Spécification de l'algorithme – nom, entrées, sorties ;
3. Explicitation de la démarche en pseudo-code – description du traitement des données qui va permettre de résoudre le problème ;
4. Traduction en langage de programmation.

La 1^{ère} et la 4^{ème} étape sont à la marge de notre propos ici :

- L'énoncé d'un problème à résoudre par le biais d'un programme informatique est une activité à part entière (souvent appelée "expression de besoin" dans le monde professionnel). Vous y avez un petit peu touché dans le cadre de vos projets, mais dans l'ensemble, dans le contexte de la NSI, les problèmes vous sont posés – et votre rôle consiste à savoir les résoudre ;
- La traduction en langage de programmation, que dans le contexte de la NSI nous réalisons en Python, a fait l'objet d'un pan du cours distinct ; nous allons évidemment y revenir en partie ici, mais la syntaxe Python n'est pas l'objet de notre étude ici.

2.1 Écrire un algorithme : spécification

Avant d'écrire un algorithme il faut bien définir ce que l'on veut faire et à partir de quoi ; il s'agit de donner **une spécification au problème**. Pour cela on doit :

- Donner un nom explicite à l'algorithme — *par exemple CuissonGâteauChocolat* ;
- Décrire les conditions d'utilisation de l'algorithme, les données qu'il attend en entrée et les conditions dans lesquelles il va pouvoir être exécuté, sa **précondition** — *par exemple "Beurre et Lait non périmés"* ;
- De même, décrire le résultat attendu, sa **postcondition**, la nature des données renvoyées et à quoi elles correspondent — *par exemple "gâteau rond, moelleux, et succulent"*.

Cette étape de spécification est fondamentale – c'est en quelque sorte la "carte d'identité" de notre algorithme, ce qui va permettre à quelqu'un qui ne le connaît pas de le comprendre sans avoir besoin de lire son code. On va donc la transcrire dans notre code Python en tête de la fonction lui correspondant – et c'est exactement ce que je vous demande de faire dans vos projets.



RÈGLES & MÉTHODES: Docstring

La transcription de la spécification d'un algorithme en tête de la fonction Python lui correspondant s'appelle "**la docstring**" ou "**la documentation**" de la fonction. Il est inscrit entre deux séries de trois apostrophes – par exemple :

```

1  def MaxNombre(n1, n2):
2      '''
3      Fonction dont les paramètres sont entiers ou réels.
4      Elle renvoie la plus grande de ces deux valeurs ou, en cas
5      d'égalité, la première valeur.
6      '''
7      if n1 < n2:
8          return n2
9      else:
10         return n1

```

Exercice 1: Rédaction d'une spécification de fonction

Considérez la fonction suivante :

```

1  def Fonction(n1, n2, n3):
2      if n1 < n2 < n3 or n3 < n2 < n1:
3          return n2
4      elif n1 < n3 < n2 or n2 < n3 < n1:
5          return n3
6      elif n2 < n1 < n3 or n3 < n1 < n2:
7          return n1
8      elif n1 == n2 and n2 == n3:
9          return n1
10     else:
11         return None

```

Est-ce que ce qu'elle fait est clair d'entrée de jeu ?

Rédigez la docstring de cette fonction pour remédier à cela.

On aura noté deux problèmes ici – l'absence de docstring, mais également l'absence d'un nom explicite à la fonction ("Fonction", ce n'est franchement pas génial...). Remédions à tout cela :

```

1  def Mediane(n1, n2, n3):
2      '''
3      Fonction qui prend en entrée trois valeurs numériques (int ou float).
4      Elle renvoie la médiane de ces trois valeurs quand celle-ci existe.
5      Si elle n'existe pas (deux valeurs égales, la troisième différente),
6      elle renvoie None.
7      Par ex: Mediane(0, 2, 1) renverra 1;
8      Mediane(1, 1, 1) renverra 1;
9      Mediane(0, 2, 2) renverra None.
10     '''
11     if n1 < n2 < n3 or n3 < n2 < n1:
12         return n2
13     elif n1 < n3 < n2 or n2 < n3 < n1:
14         return n3
15     elif n2 < n1 < n3 or n3 < n1 < n2:
16         return n1
17     elif n1 == n2 and n2 == n3:
18         return n1
19     else:
20         return None

```

Vous remarquerez l'inclusion d'exemples dans la docstring – il ne faut surtout pas hésiter à y recourir, c'est ce qu'il y a de plus parlant pour quelqu'un qui découvre votre code !



À SAVOIR ⇌ À RÉVISER:

De cette section il faut que vous reteniez :

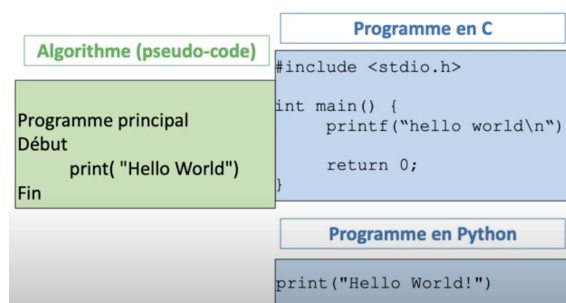
- La démarche qui va de l'énoncé d'un problème à la programmation de sa résolution en passant par l'algorithme ;
- La nécessité de **spécifier** un algorithme – le *nommer*, définir sa *précondition* et sa *postcondition*.
- La nécessité de transcrire systématiquement cette spécification dans la docstring de la fonction qui en résulte.

2.2 Ecrire un algorithme : pseudo-code

Si l'on se réfère aux étapes listées plus haut, on en est maintenant au moment où l'on sait *ce que va réaliser* notre programme, *ce qu'il va prendre en entrée*, et *ce qu'il va retourner en sortie*. Il s'agit à présent d'explicitier le **comment** – quelles sont les étapes qui vont être effectuées pour résoudre le problème ? Quel traitement va-t-on appliquer aux données en entrée pour produire les données en sortie ?

On a déjà utilisé à de multiples reprises le pseudo-code dans ce cours – donc (*en théorie*) vous devriez déjà être convaincus de son intérêt et savoir l'utiliser. Nous allons donc passer directement à quelques exercices d'application – en rappelant tout de même au préalable les principes et règles suivants :

- Le pseudo-code est une façon de décrire un algorithme pour qu'il soit compréhensible "entre humains".
- Le pseudo-code est indépendant du langage de programmation – un algorithme convenablement écrit devrait en théorie pouvoir être implémenté aussi aisément en Python qu'en C ou qu'en JavaScript³. A titre d'exemple, voici un "Hello World" en deux langages de programmation distincts, mais partant du même pseudo-code :



- Conséquence : les règles de syntaxe de pseudo-code sont inspirées des éléments communs à la plupart des langages de programmation.

3. C'est évidemment un peu simpliste d'écrire ceci ainsi, mais en théorie le principe est vrai : lorsque vous rédigez un algorithme en pseudo-code, vous devriez ne pas avoir de langage de programmation spécifique en tête.

- d. Il n'y a pas de pseudo-code universel – le seul principe à respecter, c'est que les règles de syntaxe appliquées soient bien définies, comprises et partagées par toutes celles et ceux qui seront amené·e·s à lire les algorithmes.

Ce dernier point implique qu'il y ait quand même une ossature de règles minimales dans un contexte donné – comme pour ce cours par exemple :



RÈGLES & MÉTHODES: pseudo-code en 1^{ère} NSI

- On spécifie explicitement en début d'algorithme les entrées attendues et les sorties prévues ;
- On utilise une flèche vers la gauche (" \leftarrow " ou " \triangleleft ") pour affecter des variables ;
- On utilise l'indentation pour délimiter les fonctions, les conditions, les boucles...^a
- On explicite la fin de toute structure (fonction, condition, boucle...) débütée ;
- On n'hésite pas à inclure des commentaires pour expliquer les étapes – et dans ce cas, on les préfixe d'une flèche vers la droite (" \rightarrow " ou " \triangleright ")
- ... et c'est tout !

^a. Ici on triche un peu puisque l'indentation est spécifique à Python – mais pas tant que ça puisque cela restera compréhensible même pour une implémentation dans un autre langage.

Pour illustrer ces principes, voici le pseudo-code d'une fonction (qu'on a déjà vue dans un chapitre précédent, d'ailleurs) prenant une liste de réels positifs en entrée et qui en renvoie le maximum :

Entrée: *liste* dont tous les éléments $\in \mathbb{R}^+$

Sortie: *Max* $\in \mathbb{R}^+$

```
1: fonction TROUVEMAX(liste)
2:   Max  $\leftarrow$  0
3:   pour tout Element de liste faire
4:     si Element > Max alors                 $\triangleright$  On a trouvé un nouveau max
5:       Max  $\leftarrow$  Element
6:     fin si
7:   fin pour
8:   retourner Max
9: fin fonction
```

Exercice 2: Rédaction d'algorithmes en pseudo-code

En appliquant les principes énoncés ci-dessus, rédigez les algorithmes suivants :

- Un algorithme qui prend deux nombres en entrée et affiche leur somme.
- Un algorithme qui calcule la somme des N premiers entiers naturels.
- Un algorithme qui génère les N premiers termes de la séquence de Fibonacci. (rappel : c'est une suite de nombres dont les deux premiers sont 0 et 1 et dont chaque élément est la somme des deux précédents – donc : 0, 1, 1, 2, 3, 5, 8, 13, etc...).
- Un algorithme qui vérifie si une chaîne de caractères est un palindrome (se

lit de la même manière dans les deux sens).

Il n'y a jamais (ou rarement) une solution algorithmique unique à un problème – ce qui suit n'est donc que des propositions de solution (qui sont, évidemment, valides – mais pas uniques).

a. **Entrée:** deux nombres a et $b \in \mathbb{R}$

Sortie: $a + b$

```
1: fonction SOMME( $a$ ,  $b$ )  
2:    $Resultat \leftarrow a + b$   
3:   Afficher  $Resultat$   
4: fin fonction
```

b. **Entrée:** $N \in \mathbb{N}$

Sortie: la somme des N premiers entiers naturels

```
1: fonction SOMMEENTIER( $N$ )  
2:    $Resultat \leftarrow 0$  ▷ On initialise le résultat à 0  
3:   pour  $i$  allant de 0 à  $N$  faire  
4:      $Resultat \leftarrow Resultat + i$   
5:   fin pour  
6:   retourner  $Resultat$   
7: fin fonction
```

c. **Entrée:** $N \in \mathbb{N}$ avec $N > 2$ ▷ On précise bien les valeurs acceptables en entrée

Sortie: Liste des N premiers termes de la Suite de Fibonacci

```
1: fonction FIBONACCI( $N$ )  
2:    $Resultat \leftarrow [0,1]$  ▷ On initialise avec les deux premiers termes  
3:   pour  $i$  allant de 2 à  $N-1$  faire ▷  $N$  premiers termes, donc on s'arrête à  $N-1$   
4:      $Resultat[i] \leftarrow Resultat[i-1] + Resultat[i-2]$   
5:   fin pour  
6:   retourner  $Resultat$   
7: fin fonction
```

d. **Entrée:** chn , une chaîne de caractères

Sortie: Vrai ou Faux selon si la chaîne est un palindrome ou non

```
1: fonction ESTPALINDROME( $chn$ )  
2:    $Long \leftarrow longueur(chn)$   
3:   si  $Long$  est pair alors  
4:      $Milieu \leftarrow Long/2$   
5:   sinon  
6:      $Milieu \leftarrow (Long - 1)/2$   
7:   fin si  
8:   pour  $i$  allant de 0 à  $Milieu$  faire  
9:     si  $chn[i] \neq chn[Long - i]$  alors  
10:      retourner Faux  
11:   fin pour  
12:   retourner Vrai  
13: fin fonction
```

Quelques remarques sur cet algorithme :

- Lignes 8 et 9 : le but ici est de bien comprendre l'algorithme – avec cette formule on comprend bien ce qu'on fait : on part du début pour aller jusqu'au milieu. Strictement parlant les formules ne marchent pas (en Python `chn[Long]` donnerait une erreur) – mais on s'en fiche : la démarche est claire ici, et c'est le but.
- Ligne 3 : on vérifie la parité de la longueur puisque dans le cas des longueurs impaires on ignorera le caractère du milieu. Vous noterez qu'on dit ici "si Long est pair" sans expliquer comment on fait – la difficulté de l'algorithme n'étant pas là, c'est tout à fait acceptable (on imagine qu'il y aura un algorithme pour une fonction "EstPair" explicité ailleurs).
- Lignes 10 et 13 : c'est une démarche qu'on a déjà utilisée ailleurs pendant notre cours, qui est très classique, et qu'il faut impérativement maîtriser : on part du principe que quelque chose est vrai — ici "chaîne est un palindrome" — et dès qu'on trouve une preuve du contraire (ici : un caractère est différent de son homologue de l'autre côté) on retourne "Faux", c'est-à-dire qu'on arrête immédiatement la fonction puisqu'on connaît son résultat. Si on arrive au bout de la boucle, c'est qu'on a pas trouvé de preuve que c'est faux – donc c'est vrai et c'est ce qu'on retourne.

Exercice 3: Traduction de pseudo-code en Python

- Traduisez en une fonction Python l'algorithme de la suite de Fibonacci ;
- Traduisez en une fonction Python l'algorithme de vérification qu'un mot est un palindrome ;
- Expliquez comment vous allez tester votre fonction. Comment choisissez-vous les cas que vous allez tester ?**

Pensez bien à remplir correctement la documentation (ou docstring) de votre fonction. Que renvoie la fonction `help(nom_de_votre_fonction)` ?

Pour la suite de Fibonacci, on pourra utiliser le code suivant :

```
1  def Fibonacci(N):
2      '''
3      Fonction qui prend en entrée un entier N > 2.
4      Elle renvoie les N premiers termes de la séquence de Fibonacci.
5      Par ex: Fibonacci(5) renverra [0, 1, 1, 2, 3].
6      '''
7      Resultat = [0, 1]
8      for i in range(2, N):
9          Resultat.append(Resultat[i - 1] + Resultat[i - 2])
10     return Resultat
```

Pour la tester, il suffira de la lancer avec différentes valeurs de N et de vérifier que le résultat est correct. Assez rapidement on constatera que si l'on passe une valeur de N qui n'est pas dans le champ des possibles (1, par exemple, ou -2) alors on obtient un résultat faux – nous reviendrons plus tard dans ce cours sur

comment gérer cela.

Pour le palindrome, on pourra utiliser le code suivant :

```
1  def EstPalindrome(chn):
2      '''
3      Fonction qui prend en entrée une chaîne de caractères.
4      Elle renvoie Vrai si la chaîne est un palindrome, Faux sinon.
5      Par ex: EstPalindrome("abba") renverra Vrai;
6      EstPalindrome("abbac") renverra Faux.
7      '''
8      Long = len(chn)
9      if Long % 2 == 0:
10         Milieu = Long // 2
11         # Utilisation de la division entière ("/" et non "/" pour la
12         ↪ division simple) de manière à avoir des entiers et non des float
13     else:
14         Milieu = (Long - 1) // 2
15
16     for i in range(Milieu):
17         if chn[i] != chn[Long - 1 - i]: # Sans le -1 on aura une erreur
18             return False
19
20     return True
```

Pour la tester, il faudra réfléchir un peu plus à tous les cas de figures possibles :

- Chaîne de longueur paire qui est un palindrome ;
- Chaîne de longueur paire qui n'en est pas un ;
- Chaîne de longueur impaire qui est un palindrome ;
- Chaîne de longueur impaire qui n'en est pas un.

Le choix des données à tester (les "jeux de tests") fait l'objet de la section suivante de ce cours.

Vous aurez remarqué que la fonction `help` affiche à l'écran la `docstring` de la fonction que vous avez programmée – utile, vous ne pensez pas ?



À SAVOIR



À RÉVISER:

De cette section il faut que vous reteniez :

- Les éléments qu'il faut systématiquement inclure dans la rédaction du pseudo-code d'une fonction :
 - Ses entrées – et conditions qui s'y appliquent ;
 - Ses sorties ;
 - Le traitement des données.
- Les *principes* de rédaction de pseudo-code – et par-dessus tout **le fait que le pseudo-code ne peut pas souffrir d'ambiguïté**. Le plus simple pour atteindre cet objectif est de commencer par se conformer aux règles énoncées ici ;
- Ne pas perdre de vue que la traduction d'un algorithme en code Python est un exercice en soi : la *logique* de la démarche appartient à l'algorithme, mais son *implémentation* relève du bon usage de la syntaxe du langage.

2.3 Tester un algorithme

Cette étape est cruciale dans le développement d'un programme informatique car les erreurs dans les phases de rédaction de l'algorithme et de traduction en langage de programmation sont plus que fréquentes – elles sont systématiques dès lors qu'un programme atteint un certain niveau de complexité.

Un test permet de vérifier que l'algorithme fonctionne sur une donnée précise. Pour programmer efficacement il faut concevoir des *jeux de tests* permettant de vérifier que l'algorithme renvoie, dans des cas particuliers bien choisis, ce que l'on attend de lui. Il est impossible d'écrire un ensemble de tests permettant d'exclure toutes les erreurs possibles, mais on peut cependant essayer d'en construire un en respectant déjà les règles suivantes :



RÈGLES & MÉTHODES: tester une fonction

Les **jeux de tests** (ensembles de données que l'on va tester) à préparer pour tester une fonction donnée doivent :

- Si la spécification de l'algorithme mentionne plusieurs cas possibles, les tester tous (ex : chaînes de caractères paires et impaires pour le palindrome) ;
- Si l'algorithme doit renvoyer une valeur booléenne, construire des tests permettant d'obtenir les deux valeurs de vérité (toujours l'exemple du test de palindrome) ;
- Si l'algorithme s'applique à une liste / un tableau, effectuer un test avec un tableau vide ;
- Si l'algorithme s'applique à un nombre, effectuer des tests avec des valeurs positives, négatives, et avec zéro.

Attention : ces règles ne sont pas exhaustives – vous devez plus les voir comme des principes à appliquer et à adapter à chaque fonction que vous développez.



À SAVOIR \Leftrightarrow À RÉVISER:

Apprendre par cœur les règles énoncées ici n'aurait aucun sens. Ce qu'il faut comprendre ce sont les principes qui les sous-tendent et être capable, pour les fonctions que vous allez développer (en exercice, projet, ou contrôle), de proposer des jeux de tests qui les respectent.

3 Preuves d'algorithmes

On vient de voir comment tester un algorithme – démarche qui va nous permettre de nous rendre compte, sur des cas concrets, s'il se comporte de la manière que l'on souhaite. On ne peut en revanche jamais tester *l'intégralité* des situations possibles, et il est parfois vital d'être certain, malgré cela, que l'algorithme se termine et produit le résultat attendu à tous les coups – on peut penser par exemple à l'informatique embarquée dans le pilotage automatique de voitures ou de trains...

Pour atteindre ce but on va (comme on le ferait en mathématiques) **démontrer** qu'un algorithme est correct, et ce en deux étapes :

- La **terminaison** : on montre que l'algorithme se termine.
- La **correction partielle** : on montre que l'algorithme produit bien le résultat attendu.

3.1 Terminaison : variant de boucle

Dans ce cours on va limiter le champ de cette étude à la vérification que toute boucle **tant que** (ou boucle conditionnelle, ou boucle **while**) se termine bien et n'est pas infinie. En effet, dans le contexte du programme de 1^{ère}, c'est le seul cas où la terminaison ne serait pas garantie puisque, d'une part, les seules structures se répétant que nous envisageons sont les boucles⁴, et que, d'autre part, les boucles **pour** (ou boucles itératives, ou boucles **for**) sont conçues pour se terminer au bout d'un nombre d'itérations donné, fixé, et connu à l'avance⁵.

Pour prouver qu'un algorithme s'arrête, il faut donc démontrer que pour chaque boucle **tant que**, la condition d'entrée **sera invalidée dans un temps fini** – ou, dit autrement, après un nombre fini de passages dans cette boucle. La plupart du temps, cela revient à montrer qu'il existe un **variant** pour chacune de ces boucles.



DÉFINITION: variant de boucle

Un **variant de boucle** est une quantité entière positive qui décroît strictement à chaque itération de la boucle (une suite d'entiers naturels strictement décroissante est nécessairement finie).

Exemple, pour la boucle suivante, on peut aisément se convaincre que la quantité $5 - i$ est un variant de boucle selon la définition ci-dessus : elle commence à 5 puis décroît jusqu'à atteindre 0 – on est donc bien certains que la boucle se terminera.

```
1:  $i \leftarrow 0$ 
2: tant que  $i < 5$  faire
3:    $i \leftarrow i + 1$ 
4: fin tant que
```

Attention ! Pour qu'une quantité soit un variant de boucle il faut bien qu'elle décroisse, *mais aussi* qu'elle soit *toujours* positive – en d'autres termes que la

4. En terminale on s'intéressera à la récursivité qui induisent des répétitions potentiellement infinies sans l'utilisation de boucles.

5. même si on peut en théorie imaginer une boucle allant "de 1 à n" dans laquelle on augmente n... Mais c'est en dehors du périmètre de ce cours.

condition d'arrêt de la boucle ne soit pas étroite au point de permettre au variant de "dépasser" 0. Considérez la boucle suivante par exemple :

```
1:  $i \leftarrow 0$ 
2:
3: tant que  $i \neq 5$  faire
4:    $i \leftarrow i + 2$ 
5: fin tant que
```

On voit bien que dans ce cas $5 - i$ décroît bien... mais devient assez rapidement négatif!

Exercice 4: Variant de boucle

Considérez les deux algorithmes suivants :

1: Afficher "entrez un entier positif"	1: Afficher "entrez un entier positif"
2: lire Nb	2: lire Nb
3: $i \leftarrow 0$	3: $k \leftarrow 1$
4: tant que $Nb > 0$ faire	4: $i \leftarrow 0$
5: $Nb \leftarrow Nb // 10$	5: tant que $k < (Nb + 1)$ faire
6: $i \leftarrow i + 1$	6: $k \leftarrow k \times 10$
7: fin tant que	7: $i \leftarrow i + 1$
8: Afficher i	8: fin tant que
	9: Afficher i

- Que sont censés faire ces algorithmes ?
- Utilisez la technique du variant pour justifier que le premier se termine.
- Que se passerait-il si on remplaçait la condition " $Nb > 0$ " par " $Nb \neq 0$ " ?
- Utilisez la technique du variant pour justifier que le second se termine.
- Que se passerait-il si on remplaçait " $k < (Nb + 1)$ " par " $k \neq (Nb + 1)$ " ?

- Les deux sont censés afficher le nombre de chiffres que comporte le nombre entré par l'utilisateur :
 - Le premier effectue des divisions entières successives du nombre jusqu'à atteindre 0 ;
 - Le second procède "dans l'autre sens" en partant de 1 et en multipliant par 10 jusqu'à dépasser le nombre.
- Le variant ici est le nombre Nb : en effet, pour tout nombre strictement positif, sa division entière par 10 lui est strictement inférieure – $Nb // 10 < Nb$; donc Nb est strictement décroissant.
- Si l'on remplace " $Nb > 0$ " par " $Nb \neq 0$ " le variant demeure valide. En effet, à terme, les divisions entières successives par 10 atteignent exactement 0 (et ne peuvent en aucun cas devenir négatives).
- Le variant ici est la quantité $Nb + 1 - k$ – il est facile de se convaincre qu'à chaque itération de la boucle, k étant multiplié par 10, la quantité décroît strictement.
- Si on remplaçait " $k < (Nb + 1)$ " par " $k \neq (Nb + 1)$ " en revanche, l'arrêt n'aurait plus lieu que pour les valeurs de Nb égales à 9, 99, 999, etc... –

pour toutes les autres, le variant passera en négatif sans "s'arrêter" à la valeur 0.



À SAVOIR \rightleftharpoons À RÉVISER:

De cette section il faut évidemment avoir compris ce qu'est un variant de boucle, mais il faut surtout être capable de résoudre des exercices tel que celui ci-dessus : **identifier** le variant de boucle, **prouver** qu'il décroît strictement, en **déduire** en appliquant la condition de la boucle qu'elle se termine.

3.2 Correction partielle : invariant de boucle

On va s'intéresser ici (dans le cadre du programme de 1^{ère}) à démontrer la correction des boucles dont on conçoit les algorithmes et, pour ce faire, on va se poser des questions du type :

- Les variables sont-elles bien initialisées *avant* le début de la boucle ?
- Le nombre de tours de la boucle est-il correct ?
- S'il y en a un, est-ce que l'indice est bien choisi ?
- Et, *in fine*, les valeurs obtenues en sortie de boucle sont-elles les bonnes ?

Toutes ces questions vont être abordées au moyen de la notion d'*invariant de boucle*.



DÉFINITION: invariant

Un **invariant** est une propriété d'un algorithme qui reste vraie tout au long de son exécution.

Un invariant de boucle est une proposition toujours vraie à chaque fois que l'on entre dans la boucle. La démarche que nous allons adopter se déroule en quatre étapes :

1. On choisit l'invariant :
 - Comprendre clairement le but de la boucle – qu'est-elle censée accomplir ? Quel est le résultat attendu ?
 - Partir "de la fin", c'est-à-dire du résultat attendu et identifier quelle quantité est "construite" au fur et à mesure des itérations de la boucle pour constituer ce résultat.
 - Ceci devrait vous mettre sur la voie de votre invariant – une propriété (somme d'éléments déjà traités, ordre d'éléments dans une liste...) qui ne change pas malgré les itérations de la boucle.
2. On montre que l'invariant est vérifié avant la boucle (initialisation) ;
3. On montre que si l'invariant est vérifié *avant* un passage dans la boucle, alors il est préservé *après* le passage dans la boucle ;
4. On peut conclure sur la valeur finale à la sortie de la boucle.

Et ainsi, *par récurrence*, on démontre la correction partielle.

▷ Ca vous semble très abstrait ? Vous avez notamment l'impression que le choix de l'invariant est *très, très, très* flou ? C'est normal ! Le seul moyen d'expliquer ça clairement est de s'appuyer sur des exemples...

Considérons la fonction suivante :

Entrée: $a, b \in \mathbb{N}$ avec

Sortie: Le produit $a \times b$

```
1: fonction PRODUIT(a, b)
2:   m ← 0
3:   p ← 0
4:   tant que m < a faire
5:     m ← m + 1
6:     p ← p + b
7:   fin tant que
8:   retourner p
9: fin fonction
```

On commence par noter qu'on a bien un variant de boucle... Lequel ?⁶ La terminaison est donc prouvée.

Étapes de la démarche :

1. Choix de l'invariant : le but est de renvoyer le produit p qui, à la fin, vaudra $a \times b$; il est construit dans cette boucle par ajouts successifs de b , m fois. On peut donc avoir l'intuition que l'invariant est $p = m \times b$. Vérifions cela avec les deux étapes suivantes.
2. Avant la boucle on a p et m tous les deux à 0 – donc l'invariant est vérifié.
3. Supposons qu'au début d'une itération de la boucle l'invariant est vérifié, avec m et p ; à la fin de cette itération, les valeurs respectives de m et p seront de $m' = m + 1$ et $p' = p + b$. On aura alors :

$$p' = p + b = m \times b + b = (m + 1) \times b = m' \times b$$

Et donc l'invariant est bien vérifié à la fin de la boucle.

4. En fin de boucle on a $m = a$ et donc à la sortie de la boucle on a bien $p = a \times b$.

On a donc bien démontré la correction de la boucle.

Exercice 5: Détermination d'un invariant de boucle

Donner un invariant de boucle pour la fonction suivante qui calcule x à la puissance n :

Entrée: $x, n \in \mathbb{N}$

Sortie: x^n

```
1: fonction PUISSANCE(x, n)
2:   r ← 1
3:   pour i allant de 0 à n - 1 faire
4:     r ← r × x
5:   fin pour
6:   retourner r
7: fin fonction
```

La démarche est extrêmement proche de celle qu'on a adoptée pour le produit dans l'exemple précédent :

1. On peut choisir comme invariant " $r = x^i$ " ;
2. Au début de la boucle on a $r = 1$ et $i = 0$, donc l'invariant $r = x^i$ est bien vérifié, quel que soit x .
3. Si au début d'une itération de la boucle on a l'invariant vérifié, notons

6. $a - m$ bien sûr !

$r' = r \times x$ et $i' = i + 1$ les valeurs respectives de r et i à la fin de cette itération. On a :

$$r' = r \times x = x^i \times x = x^{i+1} = x^{i'}$$

4. En fin de boucle on est passé n fois dans la boucle (de 0 à $(n-1)$) donc on a bien $r = x^n$.

Un invariant de boucle peut être une formule mathématique (une égalité, une inégalité) mais pas nécessairement – il peut également être une *propriété* qui reste vraie tout au long de la boucle.

Exercice 6: Un autre invariant d'un type un peu différent

Considérez le code suivant :

```

1 def tous_xxx(liste):
2     '''
3     Fonction qui .....
4     '''
5     i = 0
6     while i < len(liste) and liste[i] % 2 == 0:
7         i += 1
8     return i == len(liste)

```

- La docstring semble effacée et le nom de la fonction incomplet – que fait cette fonction selon vous ?
- Quel est le variant de boucle ?
- Quel est la propriété (portant sur les nombres de la liste et faisant intervenir l'indice i) qui constitue l'invariant de boucle de cette fonction ?

- On se convainc assez facilement que la fonction cherche à vérifier si tous les nombres de la liste passée en entrée sont pairs. Mais si on n'en est pas convaincu, on peut le prouver – c'est tout l'intérêt des invariants de boucle !
- Le variant de boucle est à l'évidence $\text{len}(\text{liste}) - i$ – je vous laisse le démontrer. A noter que la condition $\text{liste}[i] \% 2 == 0$ ne change rien à la preuve de terminaison puisque tout ce qu'elle pourra changer sera que la boucle se termine avant que le variant n'atteigne 0.
- Démarche pour l'invariant de boucle :

(a) Choix de l'invariant : le but est de une évaluation de si tous les nombres de la liste sont pairs. L'invariant va donc nécessairement avoir quelque chose à voir avec une liste de nombre pairs (sinon il ne permettra pas de prouver la correction partielle de la boucle et donc ne servira à rien). En creusant un peu on arrive à constater qu'à tout moment, en théorie, tous les nombres qui ont été passés en revue par la boucle sont pairs (car sinon on sort de la boucle). On peut donc exprimer l'invariant comme étant la propriété $P(i)$ suivante : "Pour tout indice k de 0 à $i-1$, $\text{liste}[k] \% 2 == 0$ ".

(b) Il est vrai avant la première itération de la boucle – puisqu'on n'a

vérifié aucun nombre à cette étape, la propriété est vraie.

(c) *S'il est vrai avant d'incrémenter i , la condition de la boucle donne deux résultats possibles :*

- Si cette condition est vraie, alors $P(i)$ reste vrai pour la prochaine valeur de i .*
- Si la condition n'est pas vraie, la boucle s'arrête et on ne touche plus à i .*

(d) *De la même manière, lorsque la boucle se termine, c'est pour une de deux raisons :*

- C'est soit parce que i est égal à $\text{len}(\text{liste})$, et dans ce cas tous les éléments sont pairs – puisqu'on a bien considéré tous les éléments de la liste, de 0 à $\text{len}(\text{liste}) - 1$ – et $P(i)$ est vrai, ce qui signifie que la fonction retourne *True*,*
- Ou parce qu'on a trouvé un élément impair et dans ce cas la boucle s'arrête et $P(i)$ indique que tous les éléments avant cet indice sont pairs, mais pas l'élément à l'indice i , donc la fonction retourne *False*.*

Exercice 7: Et un dernier pour la route...

Montrer que $r = a - b \times q$ est bien un invariant de boucle de la fonction suivante, qui réalise une division euclidienne :

Entrée: $a, b \in \mathbb{N}; b \neq 0$

Sortie: q, r le quotient et le reste de la division euclidienne de a par b

```
1: fonction DIVEUCLID(a, b)
2:   r ← a
3:   q ← 0
4:   tant que r ≥ b faire
5:     r ← r - b
6:     q ← q + 1
7:   fin tant que
8:   retourner q, r
9: fin fonction
```

La démarche est extrêmement proche de celle qu'on a adoptée pour le produit dans l'exemple précédent :

- 1. Le choix de la quantité identifiée comme invariant est donné par l'énoncé ;*
- 2. Au début de la boucle on a $r = a$ et $q = 0$, donc l'invariant $r = a - b \times q$ est bien vérifié, quel que soit b .*
- 3. Si au début d'une itération de la boucle on a l'invariant vérifié, notons $r' = r - b$ et $q' = q + 1$ les valeurs respectives de r et q à la fin de cette itération. On a :*

$$r' = r - b = a - b \times q - b = a - b \times (q + 1) = a - b \times q'$$

- 4. En fin de boucle l'invariant est nécessairement encore vrai, par récurrence (le point 2 ci-dessous étant l'initialisation et le point 3 la preuve de la récurrence).*



À SAVOIR \Leftrightarrow À RÉVISER:

Ce qu'il faut retenir de cette section est l'équivalent de ce qui était nécessaire pour la précédente :

- Comprendre la notion d'invariant de boucle ;
- Être capable de résoudre des exercices comme ceux ci-dessus : **identifier** un invariant de boucle, **montrer** qu'il est vrai **avant la boucle**, **prouver par récurrence** qu'il l'est à toutes les itérations de la boucle, et en **déduire la correction partielle** de la boucle.

Note : je vous fournirai prochainement un cahier d'exercices corrigés pour vous entraîner sur les invariants, les variants — et plus généralement sur tout le contenu de ce chapitre.

4 Complexité d'algorithmes

4.1 Cadre théorique

Lorsque l'on commence à traiter d'importants volumes de données, que l'on commence à considérer des traitements complexes, ou tout simplement longs en nombre d'instructions exécutées se pose la question de la *performance* du traitement. On évalue cette performance suivant deux axes :

- **Performance temporelle** (que nous allons aborder ici) – il s'agit du temps d'exécution du programme.
- **Performance spatiale** (qui n'est pas au programme) – il s'agit de la mémoire nécessaire à l'exécution du programme (pour stocker notamment l'intégralité des variables qu'il manipule).

On ne s'intéresse pas ici à la durée exacte d'exécution d'un programme – celle-ci est trop dépendante de la machine sur laquelle on l'exécute, des autres traitements en cours le cas échéant, du langage de programmation... Ce à quoi on va s'intéresser c'est à la mesure du **nombre d'opérations élémentaires** que va effectuer un programme en fonction des données qu'il va recevoir en entrée. Par "opération élémentaire" on entend "étape" du programme (ligne de code, le plus souvent) dont, pour simplifier, on va considérer qu'elles prennent toutes la même temps à exécuter. Ainsi, en estimant grossièrement le nombre d'opérations élémentaires en fonction du volume de données en entrée on pourra commencer à se faire une idée de la performance d'un algorithme donné : c'est ce qu'on appelle un **calcul de complexité**.

Le but principal d'un calcul de complexité est de pouvoir comparer l'efficacité entre différents algorithmes répondant à un même problème. En d'autres termes, de répondre à la question : "*Quelle que soit la machine et le langage de programmation utilisé, l'algorithme A est-il plus performant que l'algorithme B pour de grands volumes de données ?*"



DÉFINITION: complexité d'un algorithme

La **complexité d'un algorithme** est une mesure intrinsèque à l'algorithme qui est indépendante de toute implémentation. Elle est calculée en fonction d'un *paramètre représentatif des entrées* (la taille) et à l'aide d'une *mesure élémentaire* (nombre de comparaisons, nombre d'opérations arithmétiques, etc.).

4.2 Exercices d'application

Le plus simple pour comprendre ces notions est de démarrer par quelques exemples concrets – on en déduira à la section suivantes des règles et une méthode d'évaluation de la complexité.

Exercice 8: Comptage d'opérations élémentaires

Pour chacun des deux algorithmes suivants, calculer le nombre d'opérations élémentaires effectué. Dépend-il des données en entrée ?

```

1: fonction PRODUIT1(n, b)
2:    $p \leftarrow n \times b$ 
3:   retourner p
4: fin fonction

```

```

1: fonction PRODUIT2(n, b)
2:    $m \leftarrow 0$ 
3:    $p \leftarrow 0$ 
4:   tant que  $m < n$  faire
5:      $m \leftarrow m + 1$ 
6:      $p \leftarrow p + b$ 
7:   fin tant que
8:   retourner p
9: fin fonction

```

Pour Produit1, c'est vite vu : 2 opérations (le produit et le "retourner") quelles que soient les données en entrée.

Pour Produit2 c'est un peu plus compliqué :

- On a trois opérations effectuées quoi qu'il arrive (les deux affectations avant la boucle, et le "retourner").
- Dans la boucle, on a deux opérations également : le calcul du nouveau m et le calcul du nouveau p .
- On a également une opération "cachée" dans la boucle : la comparaison " $m < n$ " qui doit être effectuée à chaque itération – ce qui nous fait un total de trois opérations pour la boucle.
- La boucle est effectuée n fois — de $m = 0$ à $m = n - 1$.
- Le nombre total d'opérations élémentaires pour Produit2 est donc de : $3 \times n + 2$ (il dépend donc de la valeur de n uniquement – pas de celle de b).

Je vous laisse deviner lequel de ces deux algorithmes est le meilleur pour déterminer le produit de deux entiers n et b (imaginez le calcul de 10 milliards \times 2 par exemple).

Exercice 9: Complexité – somme des premiers entiers

Reprenons un algorithme qu'on avait écrit dans un exercice précédent et qui calculait la somme des n premiers entiers :

Entrée: $n \in \mathbb{N}$

Sortie: la somme des N premiers entiers naturels

```

fonction SOMMEENTIER(N)
   $Resultat \leftarrow 0$ 
  pour  $i$  allant de 1 à  $N$  faire
     $Resultat \leftarrow Resultat + i$ 
  fin pour
  retourner  $Resultat$ 
fin fonction

```

Quelle est sa complexité? ^a

^a. Derrière cette question s'en cachent en fait deux : 1/ combien d'opérations élémentaires effectue l'algorithme ? 2/ de quelle quantité ("paramètre représentatif des entrées") dépend ce nombre et selon quelle formule mathématique ?

- On a deux opérations en-dehors de la boucle (que l'on va assez rapidement apprendre à ignorer – il est évident qu'elles n'ont aucune influence sur la performance globale de l'algorithme).
- On a une opération évidente dans la boucle : l'ajout de l'entier en cours au résultat.
- On a en fait deux opérations cachées en plus : l'incréméntation de i , d'une part, et sa comparaison avec n d'autre part.

On arrive donc à un total de $3 \times n + 2$ opérations élémentaires – on peut donc dire que la complexité de l'algorithme **dépend de n** .^a

a. Pour prouver que les maths sont parfois utiles : on sait que $1 + 2 + \dots + n = \frac{n(n+1)}{2}$. Il est donc possible de faire le même calcul en exactement quatre opérations contre plus de 3 millions par exemple si on utilise l'algorithme de l'énoncé sur les un million premiers entiers...

Exercice 10: Complexité – boucles imbriquées

Envisageons à présent la fonction suivante, qui contient deux boucles imbriquées, et pour laquelle la complexité sera un peu plus compliquée à estimer :

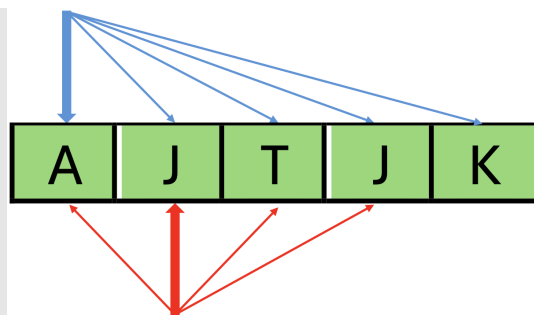
```

1  def Verif_Doublons(liste):
2      '''
3      Fonction qui prend en entrée une liste quelconque et qui vérifie qu'elle
4      ↪ ne contient pas de doublon.
5      Elle renvoie True si elle en trouve au moins un, False sinon.
6      '''
7      # On commence par parcourir la totalité de la liste
8      for i in range(len(liste)):
9          # Pour chaque élément liste[i] on le compare avec tous les autres
10         ↪ éléments de la liste
11         for j in range(len(liste)):
12             if i != j and liste[i] == liste[j]:
13                 # Si on a trouvé un doublon, on s'arrête
14                 return True
15     # Si on a atteint ce point c'est qu'on a fait toutes les comparaisons et
16     ↪ qu'on n'a trouvé aucun doublon.
17     return False

```

- Quelle est sa complexité ?
- (question bonus) Ne pensez-vous pas que cet algorithme effectue des opérations inutiles ? Lesquelles ? Comment l'améliorer ?

a. Commençons par nous assurer que nous avons bien compris le fonctionnement de l'algorithme derrière cette fonction, en imaginant qu'on lui passe en entrée une liste de 5 lettres ['A', 'J', 'T', 'J', 'K'] :



On a ici **deux boucles imbriquées** :

- La boucle extérieure (`for i in range(len(liste))`) est représentée sur le schéma par les flèches épaisses et verticales. Pour chaque valeur à l'intérieur de cette boucle, on exécute...
- ... la boucle intérieure (`for j in range(len(liste))`) qui est représentée sur le schéma par les flèches fines et diagonales.

On se convainc facilement que dans ce cas les comparaisons représentées en bleu (A-J, A-T, A-J, A-K) seront effectuées en premier sans trouver de doublon, puis ce seront les comparaisons en rouge, (J-A, J-T, J-J) et la fonction s'arrêtera sur la dernière puisqu'on aura trouvé un doublon, et elle renverra `True`.

Le nombre d'opérations élémentaires dépend donc à l'évidence de deux choses :

- La longueur (qu'on va appeler "*n*") de la liste en entrée ;
- La présence et la localisation du premier doublon – si la liste avait été [`'J'`, `'J'`, `'A'`, `'T'`, `'K'`] le traitement aurait immédiatement trouvé un doublon et donc n'aurait effectué qu'une comparaison, tandis que si la liste n'avait pas contenu de doublons l'intégralité des deux boucles aurait été parcourue.

Comme on va l'expliquer ci-dessous on va considérer la situation "au pire", c'est à dire celle où la complexité est la plus élevée, donc celle où la liste en entrée ne contient pas de doublon.

Comme c'est décrit ci-dessus, la boucle intérieure est exécutée intégralement pour chaque valeur de la boucle extérieure – pour déterminer le nombre total d'opérations élémentaires il nous suffit donc de calculer les nombre d'opérations de chacune de ces boucles et de les multiplier entre eux.

- La boucle extérieure est simple : elle effectue "*n*" incrémentations de *j* ;
- La boucle intérieure effectue une incrémentation de *i* ainsi que deux comparaisons ("`i != j`" et "`liste[i] == liste[j]`"), et ce *n* fois.

On en conclut donc que, "au pire", la complexité de la fonction sera de $n \times (3 \times n) = 3n^2$.

- b. Dans l'exemple, on peut constater que la comparaison entre A et le premier J est effectuée deux fois – une fois quand la boucle extérieure est positionnée sur le A, et une autre fois quand elle l'est sur le J. C'est

évidemment inutile, il suffirait de faire des comparaisons, dans la boucle intérieure, uniquement sur les éléments de la liste situés après celui de la boucle extérieure, donc d'indice démarrant à $(i+1)$:

```
1  def Verif_Doublons_Optim(liste):
2      '''
3      Fonction qui prend en entrée une liste quelconque et qui vérifie
4      ↪ qu'elle ne contient pas de doublon.
5      Elle renvoie True si elle en trouve au moins un, False sinon.
6      '''
7      # On commence par parcourir la totalité de la liste
8      for i in range(len(liste)):
9          # Pour chaque élément liste[i] on le compare avec tous ceux
10         ↪ qui suivent
11         for j in range(i + 1, len(liste)):
12             if liste[i] == liste[j]:
13                 # Si on a trouvé un doublon, on s'arrête
14                 return True
15     # Si on a atteint ce point c'est qu'on a fait toutes les
16     ↪ comparaisons et qu'on n'a trouvé aucun doublon.
17     return False
```

(et on notera au passage que grâce au démarrage de j à l'indice $i+1$ on s'évite également à chaque passage la comparaison $i \neq j$ que l'on avait dans la version précédente de l'algorithme)

4.3 Principes d'estimation de la complexité

Sur la base de ce que l'on vient de voir dans les exercices, on peut poser les principes de calcul de la complexité suivants :



RÈGLES & MÉTHODES: Calcul de complexité

- On commence par "compter" le nombre d'opérations élémentaires – et en déduire le paramètre représentatif des entrées dont la complexité dépend (le plus souvent ce sera la longueur d'une liste ou d'une chaîne, la valeur d'un nombre...).
- On se place dans un contexte "au pire" : lorsque plusieurs cas sont possibles on simplifie en considérant que le maximum possible d'opérations va être effectué.
- On se limite à l'ordre de grandeur de la complexité – en d'autres termes on ignore tous les termes du calcul que l'on vient de faire à l'exception de sa plus importante dépendance au paramètre des entrées : on ignore les valeurs constantes quand il y a une dépendance à n , les puissances de n inférieures à la plus grandes, les facteurs multiplicateurs constants...
Exemples :
 - Si notre calcul a donné $3n + 1$ on dira que l'ordre de grandeur de la complexité dépend de n , et on parlera d'une "complexité linéaire".
 - Si notre calcul a donné $7n^2 + 3n + 1$ on dira que l'ordre de grandeur de la complexité dépend de n^2 , et on parlera d'une "complexité quadratique".

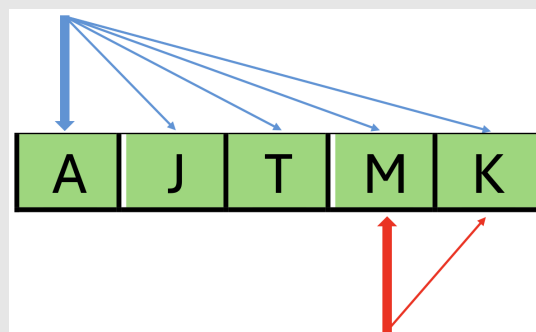
- Cas particulier : si notre calcul a donné 5 opérations, on dira que la complexité est constante (ne dépend pas des entrées).
- Cette complexité est décrite en utilisant la notation " \mathcal{O} " (appelée "grand \mathcal{O} ") – ainsi, pour les trois cas précédents on notera les complexités, respectivement, $\mathcal{O}(n)$ (complexité linéaire), $\mathcal{O}(n^2)$ (complexité quadratique), et $\mathcal{O}(1)$ (complexité constante).

Exercice 11: Nommons les complexités précédentes

En appliquant ces règles :

- Comment peut-on nommer et noter les complexités des fonctions des exercices précédents – `Produit1`, `Produit2`, `SommeEntiers`, et `Verif_Doublons` ?
- Est-ce que la complexité de la version optimisée de `Verif_Doublons` (qu'on a appelée `Verif_Doublons_Optim`) serait différente de celle de `Verif_Doublons` ?

- *`Produit1` a une complexité constante ($\mathcal{O}(1)$) ;*
 - *`Produit2` a une complexité linéaire ($\mathcal{O}(n)$) ;*
 - *`SommeEntiers` a une complexité linéaire ($\mathcal{O}(n)$) ;*
 - *`Verif_Doublons` a une complexité quadratique ($\mathcal{O}(n^2)$) .*
- Encore une fois le principe ici est d'identifier une dépendance à n , pas un calcul exact. La boucle extérieure ne pose pas de problème – on va passer n fois dedans. Ce qui va changer c'est le nombre de passages dans la boucle intérieure : le premier coup on y passera n fois, puis $(n-1)$, puis $(n-2)$ puis (...) puis, pour $i = n-2$, on n'y passera qu'une fois (comparaison entre l'avant-dernier et le dernier élément de la liste). Le schéma suivant représente en bleu les comparaisons effectuées lors du premier passage dans la boucle extérieure, et en rouge celles de l'avant-dernier passage :*



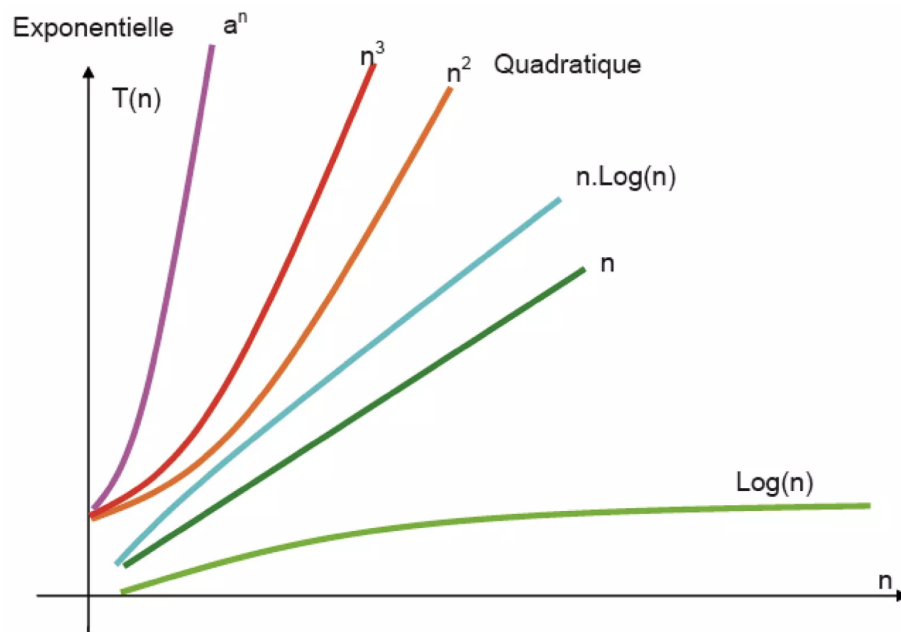
Le nombre d'opérations effectuées par cette fonction au pire sera donc (en ne regardant que celles qui dépendent de n) la somme suivante : $n + (n-1) + (...) + 2 + 1$. Ça vous rappelle quelque chose... ? $1 + 2 + ... + n = \frac{n(n+1)}{2} = \frac{1}{2} \times n^2 + \frac{1}{2} \times n$. Or on a dit que pour l'estimation de la complexité on ignore les facteurs multiplicateurs constants – donc la complexité est la même que dans la version non-optimisée, quadratique : $\mathcal{O}(n^2)$.

Ca peut paraître étrange mais c'est en fait logique : la complexité n'est pas un calcul précis d'une durée de traitement, mais une estimation de la

quantité dont dépend l'évolution du temps de traitement – et on comprend bien qu'une dépendance à n^2 ou à $\frac{1}{2} \times n^2$ revient au même : si n double, le temps de traitement quadruplera dans les deux cas.

4.4 Et dans la vraie vie... ?

On vient de voir les complexités constantes, linéaires, et quadratiques — il en existe en fait de nombreuses autres, mais qui ne sont pas au programme de 1^{ère}. Leur importance vaut le coup d'être mentionnée – et elle est évidente si l'on regarde ce graphique de croissance des principales fonctions utilisées dans les calculs de complexité.



Le lien entre la complexité et les performances temporelles d'un algorithme devraient sembler évidents à la lecture de ce graphique, mais pour s'en convaincre davantage encore, il suffit de considérer ces tables (*source : cours NSI Charles Poulmaire*). L'unité utilisée ("FLOPS") signifie "Floating-point operations per second" ou opération sur nombre flottant par seconde – c'est donc bien une unité de mesure de la performance d'un ordinateur qui se rapporte directement à la complexité qui, elle, est mesurée comme une estimation des opérations élémentaires.

		Complexité						
		1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
Taille des données	$n = 10^2$	1 ps	6,64 ps	0,1 ns	0,66 ns	0,01 μs	1 μs	4×10^{10} a
	$n = 10^3$	1 ps	9,96 ps	1 ns	9,96 ns	1 μs	1 ms	∞
	$n = 10^4$	1 ps	13,28 ps	10 ns	132,8 ns	10 ms	1 s	∞
	$n = 10^5$	1 ps	16,6 ps	0,1 μs	1,6 μs	0,01 s	> 16 min	∞
	$n = 10^6$	1 ps	19,93 ps	1 μs	19,93 μs	1 s	> 11 j	∞

(a) Puissance de l'unité de calcul : 1 téraFLOPS (ordinateurs actuels)

		Complexité						
		1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
Taille des données	$n = 10^2$	1 fs	6,64 fs	0,1 ps	0,66 ps	0,01 ns	1 ns	4×10^8 a
	$n = 10^3$	1 fs	9,96 fs	1 ps	9,96 ps	1 ns	1 μs	∞
	$n = 10^4$	1 fs	13,28 fs	10 ps	132,8 ps	10 μs	1 ms	∞
	$n = 10^5$	1 fs	16,6 fs	0,1 ns	1,6 ns	0,01 ms	1 s	∞
	$n = 10^6$	1 fs	19,93 fs	1 ns	19,93 ns	1 ms	> 16 min	∞

(b) Puissance de l'unité de calcul : 1 pétaFLOPS (ordinateurs les plus puissants du monde)

		Complexité						
		1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
Taille des données	$n = 10^2$	1 as	6,64 as	0,1 fs	0,66 fs	0,01 ps	1 ps	4×10^5 a
	$n = 10^3$	1 as	9,96 as	1 fs	9,96 fs	1 ps	1 ns	∞
	$n = 10^4$	1 as	13,28 as	10 fs	132,8 fs	10 ns	1 μs	∞
	$n = 10^5$	1 as	16,6 as	0,1 ps	1,6 ps	0,01 μs	1 ms	∞
	$n = 10^6$	1 as	19,93 as	1 ps	19,93 ps	1 μs	1 s	∞

(c) Puissance de l'unité de calcul : 1 exaFLOPS (certains réseaux actuels ; ordinateurs attendus d'ici fin des années 2020)

Les unités de temps utilisées ici – qui permettent également de mieux se rendre compte de la vitesse à laquelle évoluent les ordinateurs – sont :

∞ Infini – temps que par convention on considère comme infini car irréalisable dans la réalité.

a Année

j Jour

min Minute

s Seconde

ms Milliseconde – 10^{-3} secondes – un millièème de seconde

μs Microseconde – 10^{-6} secondes – un millièème de seconde

ns Nanoseconde – 10^{-9} secondes – un milliardième de seconde

ps Picoseconde – 10^{-12} secondes – un mille milliardième de seconde

fs Femtoseconde – 10^{-15} secondes – un million de milliardième de seconde

as Attoseconde – 10^{-18} secondes – un milliard de milliardième de seconde

Et puisque l'on parle d'unités, je précise également les unités listées dans les libellés de ces trois tableaux qui décrivent la puissance de calcul des ordinateurs considérés :

téraFLOPS 10^{12} opérations par seconde, soit mille milliards.

pétaFLOPS 10^{15} opérations par seconde, soit un million de milliards (mille fois plus puissant que le premier).

exaFLOPS 10^{18} opérations par seconde, soit un milliard milliards (mille fois plus puissant que le précédent, un million de fois plus que le premier).

4.5 Complexité - à retenir



À SAVOIR



À RÉVISER:

Nous reviendrons dans les sections qui suivent sur les notions développées ici en les appliquant à certains algorithmes connus, mais ce qu'il faut retenir *a minima* est :

- Notions de complexité spatiale et temporelle ;
- Compréhension de l'importance du type dépendance entre le volume de données en entrée et le nombre d'opérations élémentaires à effectuer ;
- Comptage des opérations élémentaires et estimation de la complexité – être capable de résoudre des exercices comme ceux qui précèdent ;
- En particulier être capable d'appliquer ces notions aux complexités constante ($\mathcal{O}(1)$), linéaire ($\mathcal{O}(n)$), et quadratique ($\mathcal{O}(n^2)$).

5 Algorithmes de tri

5.1 Pourquoi trier ?

Imaginez que vous êtes dans une bibliothèque qui contient tous les livres qui étaient disponibles en France en 2019 – ça fait quand même 810.130 livres différents⁷.



Ca ne paraît peut-être pas énorme, mais si on imagine qu'ils sont tous des livres de poche plus ou moins standard d'avec une tranche d'une épaisseur d'environ 1,9 cm, si on les range bout à bout sur une étagère, l'étagère devra mesurer plus de 15 kilomètres de long.... Alors si après ça je vous demande d'aller me chercher le livre Le Petit Prince par Antoine de Saint-Exupéry, vous allez être *très* mécontents si je précise que les livres sont rangés dans le désordre, et regretter qu'ils ne soient pas rangés par auteur puis par titre, par exemple....

En informatique, c'est la même chose – et on en a déjà un petit peu parlé dans le chapitre précédent, sur le traitement des données en table : si j'ai un tableau qui contient des milliers ou des millions d'informations (par exemple une liste de tous les clients d'Amazon, ou de tous les articles jamais publiés dans le journal Le Monde), il pourra être très lent d'en retrouver un si ils ne sont pas triés, mais à l'inverse ça pourra être relativement rapide s'ils le sont.

Le problème se pose donc de *comment* trier de grandes quantités de données – et nous allons voir qu'il y a plusieurs solutions différentes. En 1^{ère}, nous allons aborder 3 algorithmes de tri différents :

- a. Le tri par permutation ;
- b. Le tri par sélection ;
- c. Le tri par insertion.

5.2 Le tri par permutation ou "tri à bulles"

Considérez l'algorithme suivant :

Entrée: *liste* d'éléments ordonnables – des nombres par exemple

Sortie: ????

```
1: fonction TRI1(liste)
2:   N ← longueur(liste)
3:   pour i allant de 1 à N-1 faire
4:     si liste[i] > liste[i + 1] alors
5:       Échanger Liste[i] et liste[i+1]
6:     fin si
7:   fin pour
8:   retourner liste
9: fin fonction
```

7. Source : [Document "Chiffres-clés du secteur du livre 2018-2019"](#), Ministère de la Culture.

→ Que ferait cet algorithme sur la liste $[2, 5, 3, 1]$?

On a dans ce cas $N = 4$ et on peut décrire les états successifs de i et de liste en parcourant la boucle :

i	$liste[j]$	$liste[j+1]$	liste
0 (avant la boucle)	N/A	N/A	$[2, 5, 3, 1]$
1	2	5	$[2, 5, 3, 1]$
2	5	3	$[2, 3, 5, 1]$
3	5	1	$[2, 3, 1, 5]$

→ Du coup que fait l'algorithme ci-dessus ? Par quoi faudrait-il remplacer " ? ? ? ? " pour décrire ce qu'il renvoie ?

On voit que cet algorithme, par permutations successives, fait systématiquement "remonter" l'élément le plus grand de la liste jusqu'à la dernière position du tableau. On pourrait donc décrire la sortie comme étant "liste, avec son plus grand élément placé en dernière position".

→ Est-ce que cela donne des idées pour enrichir cet algorithme pour qu'il fasse un tri complet de la liste qu'il prend en entrée ?

On constate qu'une exécution de cet algorithme permet de ramener le plus grand élément de la liste, où qu'il se trouve, en dernière position. Il n'est pas compliqué de se convaincre qu'en exécutant cet algorithme une fois encore sur le même tableau permettrait de ramener l'élément le deuxième plus grand de la liste en avant dernière position — et par extension, N exécutions de l'algorithme permettrait d'aboutir à un tableau complètement trié !

Si on applique ce principe au tableau précédent, on aurait, à l'issue de chaque exécution successive de l'algorithme :

1. $[2, 5, 3, 1]$
2. $[2, 3, 1, 5]$
3. $[2, 1, 3, 5]$
4. $[1, 2, 3, 5]$

Appliquons cette conclusion à notre algorithme, traduisons-le en Python et considérons le résultat :

```
1 def TriBulles(liste):
2     '''
3     Fonction qui effectue le tri par permutations de la liste passée en entrée.
4     '''
5     n = len(liste)
6     for i in range(n):
7         for j in range(n-1):
8             if liste[j] > liste[j+1]:
9                 # Cette syntaxe permet d'affecter deux variables simultanément
10                 ↪ et donc de ne pas passer par une variable intermédiaire
11                 liste[j], liste[j+1] = liste[j+1], liste[j]
12     return liste
```




Explication video:

[Youtube short](#) présentant le tri à bulles. Les variables de cette vidéo et celles du code ci-dessus sont les mêmes :

— $i \rightleftharpoons i$

— $j \rightleftharpoons j$

→ *Quel est la complexité de cet algorithme ?*

La réponse en l'occurrence est assez immédiate : on a deux boucles imbriquées, chacune s'exécutant un nombre de fois directement dépendant de n — donc on a affaire à une complexité en $\mathcal{O}(n^2)$, autrement dit quadratique.

→ *Allez – une dernière question avant de conclure le chapitre : pourquoi à votre avis appelle-t-on ce tri par permutations "tri à bulles" ?*



C'est tout simplement l'image des bulles dans une boisson gazeuse comme du champagne qui, l'une après l'autre, remontent à la surface, à l'instar des valeurs les plus grandes de la liste qui "remontent" vers la fin.

5.3 Le tri par sélection

Présentation générale

Le tri par sélection s'inspire de la manière dont on trie des copies par ordre alphabétique des élèves :

- Au début, on tient toutes les copies dans la main gauche ;
- On choisit la copie dont le nom est le premier dans l'ordre alphabétique, et on la place face retournée sur la table devant soi ;
- On choisit ensuite parmi les copies restantes celle qui a le nom qui est le premier dans l'ordre alphabétique et on la place face retournée sur la précédente ;
- On réitère ce procédé jusqu'à ce qu'on n'ait plus de copies dans la main gauche.

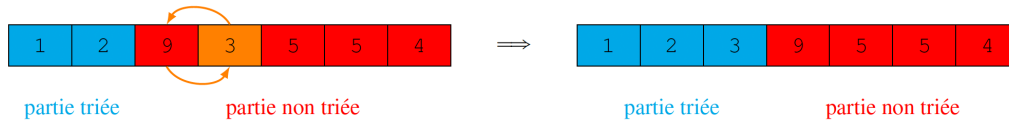
On se convainc donc facilement du fait que les copies présentes sur la table à tout instant sont triées.

Le tri par sélection fonctionne de manière similaire au détail près qu'il n'utilise pas deux "lieux" (la main et la table) – que l'on ne fait qu'échanger les éléments à l'intérieur de la table. A chaque étape du tri, on a :

- Le tableau est séparé en deux parties une partie triée "à gauche" (indices les plus petits) et une partie non triée "à droite".

1	2	9	3	5	5	4
partie triée		partie non triée				

- On choisit le plus petit élément de la partie non triée et on le place au début de la partie non triée de telle sorte que la partie triée soit augmentée d'un élément et la partie non triée soit diminuée d'un élément.

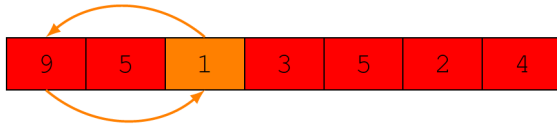


- On poursuit ainsi jusqu'à avoir parcouru toute la table.

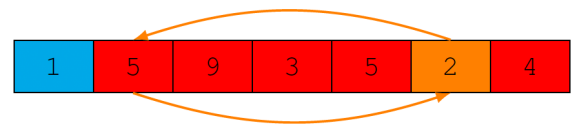
Voyons l'application de cet algorithme de bout en bout à un exemple concret, la liste [9, 5, 1, 3, 5, 2, 4] :



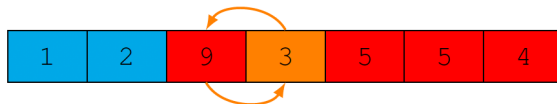
A : table dans son état initial



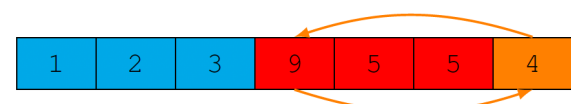
B : On déplace le minimum (1) en première position.



C : On déplace le minimum suivant (2) en deuxième position.



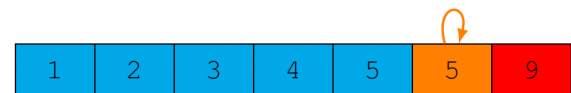
D : On déplace le minimum suivant (3) en troisième position.



E : On déplace le minimum suivant (4) en quatrième position.



F : On échange le minimum suivant (5) avec lui-même puisqu'il est déjà en place.



G : On échange le minimum suivant (5) avec lui-même puisqu'il est déjà en place.



H : table dans son état final

Algorithme

→ Sur la base de la description ci-dessus comment s'écrit cet algorithme en pseudo-code ?

Exprimé en pseudo-code cet algorithme s'écrit ainsi :

Entrée: *tab*, une liste

Sortie: *tab*, triée

1: **fonction** TRISELECTION(*tab*)

2: $n \leftarrow \text{longueur}(\text{tab})$

3: **pour** p allant de 1 à $(n-1)$ **faire**

4: $p_{\min} \leftarrow p$

5: **pour** j allant de $(p+1)$ à n **faire**

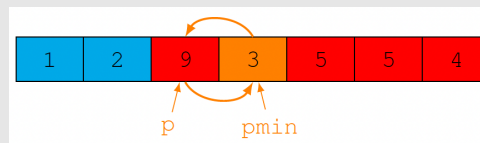
6: **si** $\text{tab}[j] < \text{tab}[p_{\min}]$ **alors** ▷ On a trouvé un nouveau min

```

7:          $pmin \leftarrow j$ 
8:     fin si
9: fin pour
10:    Échanger  $tab[pmin]$  et  $tab[p]$ 
11: fin pour
12: retourner  $tab$ 
13: fin fonction

```

Le compteur p correspond à l'indice du premier élément de la partie non triée (position à laquelle il faut déplacer le plus petit élément de la partie non triée) et la variable $pmin$ correspond à l'indice du plus petit élément de la partie non triée. On a ainsi le schéma suivant :



Explication video:

[Youtube short](#) présentant le tri par sélection. Les correspondances entre les variables de cette vidéo et celles du pseudo-code ci-dessus sont :

- $i \rightleftharpoons p$
- $j \rightleftharpoons j$
- $min \rightleftharpoons pmin$

Preuve & Complexité

Terminaison : l'algorithme est composé de boucles pour dont la fin dépend, dans les deux cas, de n , longueur du tableau en entrée, qui ne varie pas : on connaît donc le nombre de répétitions, et donc l'algorithme se termine.

Correction partielle : l'algorithme préserve l'invariant de boucle suivant : "si $p \geq 1$, tab est trié entre les indices 1 et $p - 1$, et tous les éléments restants sont supérieurs ou égaux à $tab[p-1]$ ".

Complexité : Dans la boucle interne (pour j allant de $(p+1)$ à n), on effectue $n - p$ comparaisons. p variant de 1 à $(n-1)$ on en effectue donc successivement : $n - 1$ (pour $p = 1$), puis $n - 2$ (pour $p = 2$), puis (...), puis 2 (pour $p = n - 2$), et enfin une seule (pour $p = n - 1$). La complexité de l'algorithme s'exprime donc :

$$1 + 2 + (...) + (n - 2) + (n - 1) = \frac{n(n - 1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

On peut donc en conclure que la complexité de cet algorithme est quadratique, en $\mathcal{O}(n^2)$.

Exercice 12: Codage du tri par sélection

Vous fondant sur le pseudo-code ci-dessus, coder une fonction `TriSel(tab)` qui effectue le tri par sélection d'une liste passée en argument.

```

1  def TriSel(tab):
2      '''
3      Fonction qui effectue le tri par sélection de la table passée en entrée.
4      '''
5      n = len(tab)
6      for p in range(n-1):
7          pmin = p
8          for j in range(p + 1, n):
9              if tab[j] < tab[pmin]:
10                 pmin = j
11             tab[pmin], tab[p] = tab[p], tab[pmin]
12     return tab

```

5.4 Le tri par insertion

Présentation générale

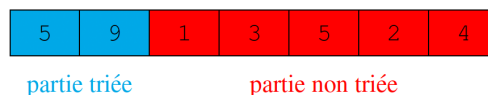
Le tri par insertion s'inspire de la manière dont la plupart des gens trient les cartes à jouer qu'ils ont en main (pour rendre l'explication lisible on va imaginer ici que toutes les cartes sont de la même couleur – il s'agit donc ici juste de les trier par ordre croissant) :

- Au début, on tient toutes ses cartes dans le désordre ;
- On regarde la deuxième en partant de la gauche : et si elle est de valeur supérieure ou égale à la première on la laisse en place ; dans le cas contraire on la ramène en première position ;
- On poursuit ainsi : on regarde à chaque fois la carte suivante, on la laisse en place si elle est supérieure à celle qui est juste à sa gauche, on la place au bon endroit dans la partie gauche de la main dans le cas contraire ;
- On réitère ce procédé jusqu'à avoir placé la carte la plus à droite.

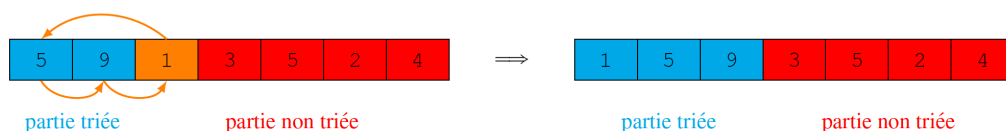
On se convainc donc facilement du fait que les cartes présentes à gauche de la main à tout instant sont triées.

Le tri par insertion fonctionne de manière similaire. A chaque étape du tri, on a :

- Le tableau est séparé en deux parties : une partie triée que l'on suppose à gauche et une partie non triée que l'on suppose à droite ;



- On insère le premier élément de la partie non triée à sa place dans la partie triée en décalant vers la droite tous les éléments de la partie triée qui lui sont supérieurs afin que la partie triée soit augmentée d'un élément et la partie non triée soit diminuée d'un élément.

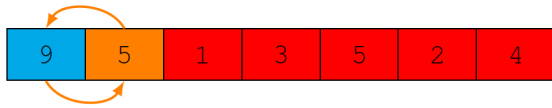


— On poursuit ainsi jusqu'à avoir parcouru toute la table.

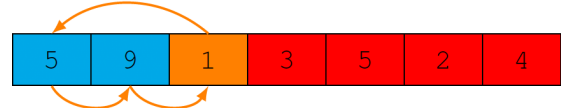
Voyons l'application de cet algorithme de bout en bout à un exemple concret, la même liste que celle que l'on a utilisée pour le tri par sélection, $[9, 5, 1, 3, 5, 2, 4]$:



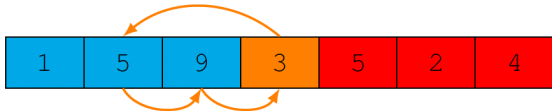
A : table dans son état initial



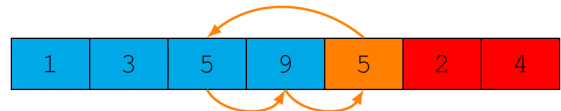
B : On regarde le deuxième élément (5), on "l'insère" à sa place – en l'occurrence au début de la liste ; on effectue donc un échange.



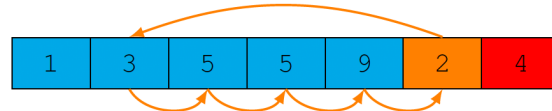
C : On insère l'élément suivant (1), toujours en début de liste – il faut donc décaler les deux éléments déjà triés "vers la droite" (donc augmenter la valeur de leurs indices).



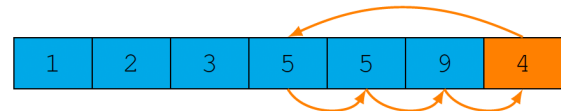
D : On insère l'élément suivant (3) en deuxième position – et on décale les éléments triés qui suivent.



E : Même chose avec le 5...



F : On insère à présent le 2, et ce sont quatre éléments qu'il faut décaler vers la droite.



G : On termine avec le 4.



H : table dans son état final

Algorithme

→ Sur la base de la description ci-dessus comment s'écrit cet algorithme en pseudo-code ? (Attention ! Pour ne pas "perdre" de valeur, vous allez devoir utiliser une variable temporaire pendant que vous faites le décalage d'indices qui précède l'insertion)

Exprimé en pseudo-code cet algorithme s'écrit ainsi :

Entrée: *tab*, une liste

Sortie: *tab*, triée

1: **fonction** TRIINSERTION(*tab*)

2: $n \leftarrow \text{longueur}(\text{tab})$

3: **pour** i allant de 2 à n **faire**

4: $\text{temp} \leftarrow \text{tab}[i]$

5: $p \leftarrow i$

6: **tant que** $p > 1$ et $\text{tab}[p-1] > \text{temp}$ **faire**

 ▷ On remonte le tableau jusqu'à trouver la place de $\text{tab}[i]$

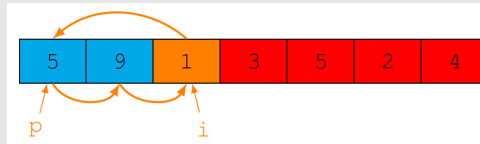
7: $\text{tab}[p] \leftarrow \text{tab}[p-1]$

```

8:          $p \leftarrow p-1$ 
9:     fin tant que
10:     $\text{tab}[p] \leftarrow \text{temp}$  ▷ On a trouvé la place de  $\text{tab}[i]$ 
11: fin pour
12: retourner  $\text{tab}$ 
13: fin fonction

```

Le compteur i correspond à l'indice du premier élément de la partie non triée et la variable p correspond à l'indice de la position où il faut insérer $\text{tab}[i]$ dans la partie triée. On a ainsi le schéma suivant :



Explication video:

[Youtube short](#) présentant le tri par insertion. La vidéo utilise des indices décalés par rapport à ce qui est présenté ici donc ne vous attardez pas trop sur les variables en tant que telles (vous risqueriez de vous y perdre), mais regardez le procédé dépeint dans la vidéo qui, lui, est le bon. Pour être complet cependant, les correspondances entre les variables de cette vidéo et celles du pseudo-code ci-dessus sont :

- $\text{key} \rightleftharpoons \text{temp (ou } \text{tab}[i])$
- $j \rightleftharpoons p-1$

Preuve & Complexité

Terminaison :

- La boucle **pour** sera de toute manière réalisée $(n - 1)$ fois, et n ne varie pas — donc elle se terminera.
- La boucle **tant que** doit être vérifiée au moyen d'un variant de boucle — p en l'occurrence : p est initialisé à i — donc est strictement positif — et décroît strictement. Donc la boucle se termine également.

Correction partielle : l'algorithme préserve l'invariant de boucle suivant : "si $i \geq 2$, tab est trié entre les indices 1 et $i - 1$, et tous les éléments restants restent à trier".

Complexité : Dans le pire des cas (si la liste est triée au départ par ordre décroissant), la boucle interne (**tant que**) effectue $2i$ comparaisons (donc 4, puis 6, puis (...), puis $(2n - 2)$, puis enfin $2n$). La complexité de l'algorithme s'exprime donc :

$$\begin{aligned}
 4 + 6 + (...) + (2n - 2) + 2n &= 2 \times (2 + 3 + (...) + (n - 1) + n) \\
 &= 2 \times \frac{(n - 1) \times (n + 2)}{2} \\
 &= n^2 + n - 2
 \end{aligned}$$

On peut donc en conclure que la complexité de cet algorithme est quadratique, en $\mathcal{O}(n^2)$.

Exercice 13: Codage du tri par insertion

Vous fondant sur le pseudo-code ci-dessus, coder une fonction `TriIns(tab)` qui effectue le tri par insertion d'une liste passée en argument.

```
1  def TriIns(tab):
2      '''
3      Fonction qui effectue le tri par insertion de la liste passée en entrée.
4      '''
5      n = len(tab)
6      for i in range(1, n):
7          p = i
8          temp = tab[i]
9          while p > 0 and tab[p-1] > temp:
10             tab[p] = tab[p-1]
11             p -= 1
12             tab[p] = temp
13     return tab
```

5.5 Comparaisons entre ces algorithmes...

Complexité "au mieux"

On a dit plus haut dans ce cours que les calculs de complexité se font dans des configurations "au pire" – donc dans les cas où, à taille de données en entrée équivalente, les données sont telles que l'algorithme effectuera un maximum d'opérations. Mais si, comme il semble que ce soit le cas ici, on a des algorithmes apparemment équivalents, est-ce qu'on ne pourrait pas regarder ce qu'il se passe "au mieux" pour les départager ?

Considérons de nouveau le tri par insertion et imaginons que les données en entrée sont déjà triées. En ce cas, on voit bien que la condition `tab[p-1] > temp` n'est *jamaïs* satisfaite, et que donc seule la boucle externe, la boucle **pour** influence la complexité. Or celle-ci est effectuée $(n - 1)$ fois, donc on se retrouvera avec une complexité linéaire, en $\mathcal{O}(n)$.

Cette considération a-t-elle un intérêt ? Imaginez que vous mettiez bout à bout deux listes, toutes deux déjà triées, une (appelée "t") très longue, et une (appelée "s") très courte, comme dans le schéma ci-dessous :

t[1]	t[2]	t[...]	t[999999]	t[1000000]	s[1]	s[2]	s[3]	s[4]	s[5]
------	------	--------	-----------	------------	------	------	------	------	------

On se convainc aisément que dans ce cas la vaste majorité des itérations se fera sans passage dans la boucle interne et que seules les quelques dernières donneront lieu à un réel tri, ce qui réduira considérablement le temps de traitement par rapport à un tri par sélection où toutes les comparaisons seront effectuées quoi qu'il arrive.

Deux remarques au sujet de ce qui précède :

1. Vous noterez que dans cet exemple le fait de mettre la liste courte **après** la liste longue est fondamental – c'est typiquement le genre de question d'optimisation

qu'il faut régulièrement se poser ;

2. *En pratique cette situation ne se poserait pas en ces termes – je vous la présente ici simplement pour illustrer une différence de performances entre ces deux algorithmes de tri.*

Stabilité d'un tri

Un autre critère de comparaison entre différents algorithmes de tri est leur stabilité – on entend par là dans quel ordre se retrouveront les éléments égaux entre eux une fois la table triée. Il y a deux possibilités :

- Tri stable : les éléments conservent quoiqu'il arrive le même ordre qu'ils avaient dans la table initiale ;
- Tri instable : les éléments peuvent changer d'ordre par rapport à la table initiale.

L'importance de ce critère peut se comprendre lorsque l'on considère des tris suivant plusieurs axes. Imaginons qu'on a une liste d'albums de musique qui est déjà triée par noms d'album et qu'on veut ensuite lui appliquer un tri par nom d'artiste : un tri stable fera que les albums d'un même artiste resteront en ordre alphabétique, tandis qu'un tri instable ne le garantira pas.

Imaginons à présent qu'on considère une liste de bouteilles triée par marque, représentée ainsi :



On souhaite la trier par contenance de bouteilles. Un tri instable donnera par exemple :



Tandis qu'un tri stable donnera (unique solution possible) :



(observez les bouteilles de contenance 1 et 4 dans cet exemple)

Pourquoi est-ce important ? Imaginons que vous avez la liste de tous les élèves de 1^{ère} générale de tous les lycées de France pour la rentrée 2023 — un peu plus de 550.000 élèves d'après le fichier qu'on avait manipulé en TD au chapitre précédent. Cette liste a été triée deux fois : une première fois selon l'ordre alphabétique des noms des élèves, puis une seconde selon l'ordre alphabétique des noms de lycées. Si l'on vous demande de dans quels lycées il y a des élèves dont le nom de famille est "Biver", votre recherche sera *très* différente selon la stabilité du second tri :

- Si le second tri est stable, alors vous savez que les noms des élèves à l'intérieur des établissements sont restés en ordre alphabétique (puisque la liste était initialement triée par noms d'élèves) : il vous suffit donc de passer d'établissement en établissement, et de regarder à "BIV" si vous trouvez des élèves.
- Dans le cas contraire en revanche, vous n'avez aucun moyen de savoir comment sont triés les élèves à l'intérieur des établissements \implies vous n'avez d'autre choix que de parcourir la totalité des 550.000 lignes du fichier à la recherche du nom Biver...

Tri initial		Tri stable		Tri instable	
Lycée	Nom	Lycée	Nom	Lycée	Nom
Lycee2	Abebe	Lycee1	Abebe	Lycee1	Biver
Lycee2	Abebe	Lycee1	Ali	Lycee1	Bello
Lycee1	Abebe	Lycee1	Ali	Lycee1	Ali
Lycee3	Ahmed	Lycee1	Bello	Lycee1	Bose
Lycee2	Ahmed	Lycee1	Biver	Lycee1	Abebe
Lycee1	Ali	Lycee1	Bose	Lycee1	Burtin
Lycee1	Ali	Lycee1	Burtin	Lycee1	Chen
Lycee2	Allere	Lycee1	Chen	Lycee1	Ali
Lycee3	Alliot	Lycee2	Abebe	Lycee2	Abebe
Lycee2	Bello	Lycee2	Abebe	Lycee2	Allere
Lycee1	Bello	Lycee2	Ahmed	Lycee2	Bello
Lycee1	Biver	Lycee2	Allere	Lycee2	Biver
Lycee2	Biver	Lycee2	Bello	Lycee2	Chen
Lycee3	Bose	Lycee2	Biver	Lycee2	Ahmed
Lycee1	Bose	Lycee2	Burtin	Lycee2	Burtin
Lycee3	Burtin	Lycee2	Chen	Lycee2	Abebe
Lycee1	Burtin	Lycee3	Ahmed	Lycee3	Ahmed
Lycee2	Burtin	Lycee3	Alliot	Lycee3	Chen
Lycee3	Chen	Lycee3	Bose	Lycee3	Burtin
Lycee2	Chen	Lycee3	Burtin	Lycee3	Alliot
Lycee1	Chen	Lycee3	Chen	Lycee3	Bose

Ceci était un début de lien avec la section suivante de ce chapitre – la *recherche dans les tableaux* ; nous y reviendrons donc.

Avantages et inconvénients

	Tri à bulles	Tri par sélection	Tri par insertion
Avantages	* Simple à comprendre & implémenter * Stable	* Performant sur de petites listes	* Stable * Performant sur des listes déjà triées en partie

5.6 Les tris fournis par Python

Bon, soyons honnêtes... Les tris que l'on a étudiés ici ont pour but principal de vous initier à l'étude de l'algorithmique – en pratique ils ne sont pas très performants et il en existe d'autres de bien mieux (notamment certains que l'on étudiera en Terminale pour ceux d'entre vous qui conserveront la spécialité NSI).

Parmi les fonctions de tri beaucoup plus performantes il en est deux qui vous sont directement fournies par Python : la fonction `sorted()` qui fournit une copie triée de votre tableau ; et la méthode `t.sort()` qui elle effectue une tri *en place* d'un tableau `t` (c'est-à-dire qu'elle modifie le tableau lui même pour le trier). Ces deux fonctions effectuent un tri stable.

Illustration de leur fonctionnement :

```
>>> tab = [55, 2, 1, 34, 3, 99, 20, 12, 3, 0]
>>> sorted(tab)
[0, 1, 2, 3, 3, 12, 20, 34, 55, 99]
>>> tab
[55, 2, 1, 34, 3, 99, 20, 12, 3, 0]
>>> tab.sort()
>>> tab
[0, 1, 2, 3, 3, 12, 20, 34, 55, 99]
```

5.7 Algorithmes de tri – à retenir



À SAVOIR ⇌ À RÉVISER:

De ces algorithmes de tri il faut que vous reteniez :

- Les principes de base – vous devez être capables de les expliquer si on vous en donne le pseudo-code ou le code ;
- La complexité et le calcul de son ordre de grandeur ($\mathcal{O}()$) ;
- Le fonctionnement – tel qu'on l'a décrit ici, qui doit vous permettre de résoudre des exercices comme ceux qui sont inclus dans le cahier d'exercices d'entraînement ;
- Sommairement, les avantages et inconvénients de chacun d'entre eux.

6 Algorithmes de recherche

Il est fréquent en informatique de devoir effectuer une recherche dans un grand nombre de données – pensez connexion à une plate-forme sur internet, production d’une facture (de téléphone, d’internet), affichage d’éléments répondant à certains critères sur des sites marchands (tous les livres d’un certain auteur sur un site comme la librairie en ligne *decitre* par exemple), etc... Dans tous ces cas il s’agit d’isoler un petit nombre d’éléments (voire un élément unique comme dans le cas d’une connexion par exemple) qui se trouvent parmi un nombre extrêmement grand d’éléments similaires – il va de soi que la vitesse à laquelle ces recherches vont se faire va être cruciale.

6.1 Cas général : recherche séquentielle

Dans le cas où le tableau dans lequel on cherche l’information n’est pas trié il n’y a pas de raccourci possible et on n’a d’autre choix que de faire une recherche systématique ou *recherche séquentielle* (en anglais : *linear search*), qui consiste simplement à parcourir la liste jusqu’à trouver l’élément que l’on cherche.



Explication video:

Non que ce soit franchement nécessaire vu la simplicité de l’algorithme, mais voici un [Youtube short](#) présentant la recherche séquentielle.

→ On a déjà utilisé de tels algorithmes en cours – comment s’écrit cet algorithme en pseudo-code ?

Entrée: *liste, elt_rech*

Sortie: *position de l’élément elt_rech dans la liste, -1 si non trouvé*

```
1: fonction CHERCHEELT(liste, elt_rech)
2:    $N \leftarrow \text{longueur}(\text{liste})$ 
3:   pour  $i$  allant de 1 à  $N$  faire
4:     si  $\text{elt\_rech} = \text{liste}[i]$  alors           ▷ On a trouvé l’élément recherché
5:       retourner  $i$ 
6:     fin si
7:   fin pour
8:   retourner  $-1$                                ▷ On ne l’a pas trouvé
9: fin fonction
```

Exercice 14: Recherche séquentielle

- Quelle est la complexité de cet algorithme ?
- Vous fondant sur le pseudo-code ci-dessus, coder une fonction `RechSeq(tab, elt)` qui effectue une recherche séquentielle de `elt` dans `tab`.

- À l’évidence la complexité est linéaire, en $\mathcal{O}(n)$, puisque l’on parcourt dans le cas le pire la totalité du tableau, une fois (note : dans le cas au mieux, donc celui où l’élément recherché est en tout début de tableau, on bascule évidemment sur une complexité constante, en $\mathcal{O}(1)$) – mais il va de soi que l’on ne peut pas compter dessus (sinon la fonction recherche serait inutile,

non ?)

b. Code Python :

```

1  def RechSeq(tab, elt):
2      '''
3      Fonction qui effectue une recherche séquentielle de l'élément elt
      ↪ dans la liste tab.
4      '''
5      n = len(tab)
6      for i in range(n):
7          if tab[i] == elt:
8              return i
9      return -1

```

6.2 Recherche dichotomique

Présentation générale

→ A et B jouent au jeu suivant : A choisit un nombre entre 0 et 100, et ne le communique pas à B qui doit le deviner en posant des questions. Les réponses de A ne peuvent être que "oui" ou "non".

Quelle stratégie doit adopter B pour poser le moins de questions possibles ?

B doit chercher à éliminer le plus de possibilités à chaque question qu'il pose.

Il pourrait commencer par demander "est-ce que ton nombre est supérieur à 90 ?" : si A répondait "oui", il aurait éliminé d'un coup 90% des possibilités – mais le problème, évidemment, est qu'il y a 90% de chances que A réponde "non" auquel cas il n'en aura éliminé que 10%...

Comme il ne sait rien à propos du nombre qu'il cherche sa stratégie optimale consiste donc à diviser par deux le champ des possibilités à chaque question qu'il pose : commencer par "est-ce que ton nombre est supérieur à 50 ?" puis, selon la réponse de A, "est-ce que ton nombre est supérieur à 25 ?" ou "est-ce que ton nombre est supérieur à 75 ?", puis de nouveau avec 12, ou 37, ou 62, ou 87, etc...

La démarche que nous venons de décrire en est une de *recherche dichotomique* (*binary search* en anglais).



Explication video:

[Youtube short](#) présentant la recherche dichotomique telle qu'on vient de la décrire.

→ Si c'est vraiment plus efficace que la recherche séquentielle, pourquoi ne se sert-on pas toujours d'une recherche dichotomique ?

*La réponse est évidente – la recherche dichotomique n'est possible **que** si les éléments parmi lesquels on recherche sont triés (et dans le cas du jeu entre A et B, puisqu'on considère tous les nombres entre 0 et 100, ils sont implicitement*

triés).

Algorithme

Le principe est donc le suivant : on considère un tableau T trié de n éléments et une valeur x . L'algorithme prend en entrée T et x . Il doit renvoyer la position de la valeur x dans le tableau T quand elle est présente et -1 sinon. Puisque le tableau est trié, comparer x avec la valeur médiane permet de savoir si x est situé dans la première moitié du tableau ("à gauche"), positionné au milieu, ou bien appartient à la seconde moitié du tableau ("à droite"). Le principe est donc le suivant : comparer x avec la valeur de la case au milieu du tableau T :

- Si les valeurs sont égales alors la position du milieu est la position cherchée ;
- Sinon on recommence en cherchant x dans la moitié du tableau qui convient.

→ Sur la base de ces descriptions, parvenez-vous à rédiger cet algorithme en pseudo-code ?

Entrée: T, x

Sortie: position de l'élément x dans T , -1 si non trouvé

```
1: fonction RECHDICHOTOMIQUE( $T, x$ )
2:    $Debut \leftarrow 1$ 
3:    $Fin \leftarrow longueur(T)$ 
4:   tant que  $Debut \leq Fin$  faire
5:      $Milieu \leftarrow (Debut + Fin) // 2$ 
6:     si  $x = T[Milieu]$  alors                                ▷ On a trouvé l'élément recherché
7:       retourner  $Milieu$ 
8:     sinon si  $x > T[Milieu]$  alors
9:        $Debut \leftarrow Milieu + 1$                             ▷ L'élément n'est pas "à gauche"
10:    sinon
11:       $Fin \leftarrow Milieu - 1$                                 ▷ L'élément n'est pas "à droite"
12:    fin si
13:  fin tant que
14:  retourner  $-1$                                               ▷ On ne l'a pas trouvé
15: fin fonction
```

Exercice 15: Application concrète

Effectuez une recherche dichotomique de l'élément 5 dans la liste $[1, 2, 5, 8, 9, 12, 14, 15]$ et complétez le tableau suivant :

Debut	Fin	Milieu	liste[Milieu]
...

On est en pseudo-code donc je pars du principe que les indices du tableau vont de 1 à n (8 en l'occurrence). On cherche la valeur 5.

<i>Debut</i>	<i>Fin</i>	<i>Milieu</i>	<i>liste[Milieu]</i>
1	8	4	8
1	3	2	2
3	3	3	5

Et l'algorithme retourne donc l'indice 3.

Exercice 16: Application concrète – 2

Même exercice que le précédent, avec l'élément 7, toujours dans la liste [1, 2, 5, 8, 9, 12, 14, 15].

Debut	Fin	Milieu	liste[Milieu]
...

On est en pseudo-code donc je pars du principe que les indices du tableau vont de 1 à n (8 en l'occurrence). On cherche la valeur 7.

<i>Debut</i>	<i>Fin</i>	<i>Milieu</i>	<i>liste[Milieu]</i>
1	8	4	8
1	3	2	2
3	3	3	5
4	3	—	—

A cette dernière étape on a $Fin < Debut$, donc on sort de la boucle et l'algorithme retourne la valeur -1.

Preuve & Complexité

Terminaison : à chaque tour de boucle on divise par 2 la longueur de la partie du tableau à examiner. Cette longueur, qui est une valeur entière, est un variant de la boucle tant que (car elle est positive et strictement décroissante).

Correction partielle : on a deux situations distinctes à envisager :

- Si x n'est pas dans T alors on se convainc aisément qu'il n'appartient à aucun des sous-tableaux de T donc puisque l'on sait que la boucle se termine on renverra bien -1.
- Si x appartient au tableau T , alors il s'agit de montrer qu'à chaque tour de boucle x appartient bien à la partie du tableau T comprenant les éléments de **Debut** à **Fin** ou bien que l'algorithme renvoie bien sa position dans le tableau. Le tableau est découpé en trois parties : $Intervalle_1 = [Debut, Milieu-1]$, $Intervalle_2 = [Milieu, Milieu]$, et $Intervalle_3 = [Milieu+1, Fin]$.
 - Si x est à $T[Milieu]$ (donc dans $Intervalle_2$) alors le résultat est renvoyé et l'algorithme est terminé;
 - Si x est dans $Intervalle_1$ alors le test $x > T[Milieu]$ est faux (car le tableau est trié) et **Fin** prend la valeur **Milieu** - 1;
 - Si x est dans $Intervalle_3$ alors le test $x > T[Milieu]$ est vrai (car le tableau est trié) et **Debut** prend bien la valeur **Milieu** + 1;

L'algorithme préserve donc bien l'invariant de boucle " x appartient à la partie du tableau T comprenant les éléments de **Debut** à **Fin**" et on a donc bien prouvé la correction partielle de cet algorithme.

Complexité : nous n'allons pas réaliser ici le calcul exact de cette complexité car il dépasse le cadre du programme – nous allons en revanche l'étudier brièvement empiriquement et en apprendre le résultat.

Exercice 17: Recherche dichotomique dans différents tableaux

On se place dans le cas le pire, donc celui où l'élément recherché n'est pas dans le tableau : imaginons donc que l'on cherche l'élément 1 dans des tableaux de différentes tailles ne contenant que des 0 (qui sont donc, à l'évidence, triés). Combien de tours de boucles sont-ils nécessaires pour arriver à cette conclusion (donc renvoyer -1) si le tableau contient :

- 1 élément ?
- 2 éléments ?
- 4 éléments ?
- 32 éléments ?
- Que pourriez-vous deviner sur la complexité sur la base de ces quelques exemples ?

- a. Dès le départ on a $Debut = Milieu = Fin$, donc on fera une vérification, puis on aura $Debut$ qui passera à 2 et on sortira donc de la boucle :

<i>Debut</i>	<i>Fin</i>	<i>Milieu</i>
1	1	1
2	1	—

- b. Un tour supplémentaire :

<i>Debut</i>	<i>Fin</i>	<i>Milieu</i>
1	2	1
2	2	2
3	2	—

- c. Encore un tour de plus !

<i>Debut</i>	<i>Fin</i>	<i>Milieu</i>
1	4	2
3	4	3
4	4	4
5	4	—

- d. Cette fois, on ajoute 3 tours !

<i>Debut</i>	<i>Fin</i>	<i>Milieu</i>
1	32	16
17	33	25
26	33	29
30	33	31
32	33	32
33	33	33
34	33	—

e. On a 1 tour pour 1 élément ; 2 pour 2 ; 3 pour 4 ; 6 pour 32. Soustrayons 1 de toutes ces valeurs et on tombe sur : $0 \Leftrightarrow 1$; $1 \Leftrightarrow 2$; $2 \Leftrightarrow 4$; $5 \Leftrightarrow 32$. Écrit autrement, cela donne : $0 \Leftrightarrow 2^0$; $1 \Leftrightarrow 2^1$; $2 \Leftrightarrow 2^2$; $5 \Leftrightarrow 2^5$ — et je vous laisse faire les calculs mais vous trouveriez également que $3 \Leftrightarrow 8 = 2^3$, $4 \Leftrightarrow 16 = 2^4$, $10 \Leftrightarrow 1024 = 2^{10}$, et même $100 \Leftrightarrow 1.267.650.600.000.000.000.000.000.000 \approx 2^{100}$!

En d'autres termes, l'algorithme mettra, au pire, 100 fois plus de temps pour chercher un élément dans un tableau en contenant plus de mille milliards de milliards de milliards (10^{30}) que dans un tableau en contenant deux — pas mal, non ? En tous cas **beaucoup** plus performant qu'une recherche séquentielle à la complexité linéaire (et qui donc mettrait mille milliards de milliards de milliards de fois plus de temps) !

Cette valeur, la "puissance de 2 à laquelle correspond un nombre" a un nom : c'est le logarithme binaire (dont la fonction est notée \log_2) qui a la propriété suivante pour tout entier naturel n : $\log_2(2^n) = n$.

C'est une fonction mathématique que vous étudierez ultérieurement — et c'est pour cela que nous n'irons pas plus loin aujourd'hui, si ce n'est pour affirmer :



A retenir:

La complexité de la recherche dichotomique dépend de la valeur de la première puissance de 2 supérieure à la taille du tableau. Elle est donc logarithmique, c'est-à-dire qu'elle est de l'ordre de $\log_2(n)$. On dit qu'elle est en $\mathcal{O}(\log_2(n))$.

Exercice 18: Codage de la recherche dichotomique

Vous fondant sur le pseudo-code ci-dessus, coder une fonction `RechDicho(T, x)` qui effectue une recherche dichotomique de x dans un tableau T passés en argument.

```

1  def RechDich(T, x):
2      '''
3      Fonction qui effectue une recherche dichotomique de x dans T.
4      '''
5      debut = 0
6      fin = len(T) - 1
7      while fin >= debut:
8          milieu = (fin + debut) // 2
9          if T[milieu] == x:
10             return milieu
11          elif x > T[milieu]:
12             debut = milieu + 1
13          else:
14             fin = milieu - 1
15     return -1

```




À SAVOIR \Leftrightarrow À RÉVISER:

A l'instar des sections précédentes, ils vous sera demandé ici de retenir et de comprendre :

- Les principes de base de fonctionnement de la recherche séquentielle et, surtout, de la recherche dichotomique – vous devez être capables de les expliquer si on vous en donne le pseudo-code ou le code ;
- La complexité des deux - mais pas le calcul logarithmique ;
- Le fonctionnement – tel qu'on l'a décrit ici, qui doit vous permettre de résoudre des exercices comme ceux qui sont inclus dans le cahier d'exercices d'entraînement.

7 Algorithmes gloutons