

TD preuve d'un algorithme

Term NSI — Octobre 2026

Algorithmes de tri par insertion et par sélection. algorithmes calculant sur une suite par récurrence, preuve de correction, preuve de terminaison, INVARIANT, VARIANT

1 Invariants de boucle et raisonnement par récurrence

Terminale NSI — Fiche d'exercices

Objectifs

- Faire le bilan sur les tris par sélection et par insertion vus en Première.
- Savoir prouver qu'un algorithme itératif est **correct** (invariant de boucle) et qu'il **termine** (variant).
- Écrire des algorithmes itératifs (boucles `for` ou `while`) à partir d'une **relation de récurrence**.

Notation : pour un tableau (ou une chaîne) `t`, on note `t[:k]` la portion `t[0]`, `t[1]`, ..., `t[k-1]`, c'est-à-dire les `k` premiers éléments (portion vide si `k = 0`). C'est aussi la syntaxe Python.

1.1 Partie A — Bilan : les tris par insertion et par sélection (≈ 20 min)

Exercice 1 — Exercice résolu : le tri par insertion

Cet exercice est entièrement rédigé. Lisez-le attentivement : il sert de **modèle** pour l'exercice 2. Le raisonnement en trois étapes utilisé ici sera formalisé dans la partie B.

```

1  def tri_insertion(t):
2      for i in range(1, len(t)):
3          cle = t[i]
4          j = i - 1
5          while j >= 0 and t[j] > cle:
6              t[j + 1] = t[j]
7              j = j - 1
8          t[j + 1] = cle

```

Principe. À chaque tour, on prend l'élément suivant (`cle`) et on l'insère à sa place dans la partie gauche déjà triée, en décalant d'une case vers la droite les éléments plus grands que lui.

Suivi de l'exécution avec `t = [5, 2, 8, 1, 4]` (contenu de `t` à la fin de chaque tour de la boucle externe) :

Moment	<code>cle</code>	Contenu de <code>t</code>
Avant la boucle	—	[5, 2, 8, 1, 4]
Fin du tour <code>i = 1</code>	2	[2, 5, 8, 1, 4]
Fin du tour <code>i = 2</code>	8	[2, 5, 8, 1, 4]
Fin du tour <code>i = 3</code>	1	[1, 2, 5, 8, 4]
Fin du tour <code>i = 4</code>	4	[1, 2, 4, 5, 8]

Invariant de la boucle externe (on ne s'intéresse pas à la boucle interne) :

Après `k` tours de la boucle externe, `t[:k+1]` contient les mêmes éléments qu'au départ dans `t[:k+1]`, mais **triés**.

Preuve de l'invariant.

- **Initialisation** ($k = 0$) : avant le premier tour, $t[:1]$ ne contient qu'un seul élément. Il est donc trié.
- **Conservation** : supposons qu'au début du tour $i = k + 1$, la portion $t[:k+1]$ soit triée. La boucle `while` décale d'une case vers la droite tous les éléments de cette portion strictement plus grands que `cle`, puis `cle` est placée dans la case libérée. À sa gauche ne restent que des éléments inférieurs ou égaux à `cle`, à sa droite que des éléments plus grands, et chaque groupe garde son ordre. La portion $t[:k+2]$ contient donc les mêmes éléments qu'avant, triés.
- **Conclusion** : la boucle externe fait $n - 1$ tours (i va de 1 à $n - 1$). Après $n - 1$ tours, l'invariant dit que $t[:n]$, c'est-à-dire le tableau entier, est trié.

Remarque importante. Après 2 tours, $t[:3] = [2, 5, 8]$, alors que les trois plus petits éléments du tableau sont 1, 2 et 4. Dans le tri par insertion, la partie gauche est **triée mais pas définitive** : un élément plus petit, situé plus loin, peut encore venir s'y insérer.

Exercice 2 – Le tri par sélection

En vous inspirant de l'exercice résolu, étudier le tri par sélection.

```

1  def tri_selection(t):
2      n = len(t)
3      for i in range(n - 1):
4          i_min = i
5          for j in range(i + 1, n):
6              if t[j] < t[i_min]:
7                  i_min = j
8          t[i], t[i_min] = t[i_min], t[i]
```

2.1 Expliquer en une phrase le principe de ce tri.

2.2 On exécute `tri_selection(t)` avec $t = [5, 2, 8, 1, 4]$. Compléter le tableau donnant le contenu de t à la fin de chaque tour de la boucle externe.

Moment	Contenu de t
Avant la boucle	$[5, 2, 8, 1, 4]$
Fin du tour $i = 0$	
Fin du tour $i = 1$	
Fin du tour $i = 2$	
Fin du tour $i = 3$	

2.3 Observer la partie gauche du tableau dans chaque ligne. Compléter l'invariant de la boucle externe :

Après k tours de la boucle externe, $t[:k]$ est et contient les éléments du tableau.

2.4 Expliquer en deux ou trois phrases pourquoi, si l'invariant est vrai au début d'un tour, il est encore vrai à la fin de ce tour.

2.5 La boucle externe ne fait que $n - 1$ tours. En utilisant l'invariant, expliquer pourquoi le tableau est quand même entièrement trié à la fin.

2.6 Relire la remarque finale de l'exercice résolu. Dans le tri par sélection, la partie gauche du tableau est-elle définitive ? Quelle est la différence essentielle entre les invariants des deux tris ?

1.2 Partie B – Méthode : prouver un algorithme itératif (≈ 20 min)

À retenir

Un **invariant de boucle** est une propriété $P(k)$, portant sur les variables du programme, qui est vraie **après k tours de boucle**, quel que soit k .

On le démontre en trois étapes :

1. **Initialisation** : $P(0)$ est vraie, c'est-à-dire avant le premier tour.
2. **Conservation** : si $P(k)$ est vraie au début d'un tour, alors $P(k+1)$ est vraie à la fin de ce tour.
3. **Conclusion** : à la sortie de la boucle, l'invariant (et éventuellement la condition d'arrêt) permet d'obtenir le résultat attendu.

Avec une boucle `for`, le nombre de tours est connu d'avance : la boucle se termine forcément. Avec une boucle `while`, il faut aussi prouver la **terminaison** à l'aide d'un **variant** : une quantité **entière, positive**, qui **décroît strictement** à chaque tour. Elle ne peut pas décroître indéfiniment, donc la boucle s'arrête.

Exercice 3 – Somme des premiers entiers

```
1 def somme(n):
2     s = 0
3     for i in range(1, n + 1):
4         s = s + i
5     return s
```

3.1 Compléter le tableau de suivi pour l'appel `somme(5)`.

Nombre de tours k	Valeur de i pendant le tour	Valeur de s en fin de tour
0	—	0
1		
2		
3		
4		
5		

3.2 Compléter l'invariant : $P(k)$: « après k tours, $s = \dots\dots\dots$ ».

3.3 Rédiger la preuve de l'invariant en trois étapes (initialisation, conservation, conclusion). En déduire que `somme(n)` renvoie bien $1 + 2 + \dots + n$.

3.4 En mathématiques, comment s'appelle ce type de raisonnement ? Faire correspondre les trois étapes de la preuve d'un invariant avec celles de ce raisonnement.

1.3 Partie C – Des relations de récurrence aux boucles (≈ 60 min)

Chaque exercice suit le même schéma : **relation de récurrence** → **programme** → **tableau de suivi** → **invariant** → **preuve**.

Exercice 4 – Multiplier sans utiliser `*` (boucle `for`)

On veut calculer $a \times b$ pour un entier a quelconque et un entier $b \geq 0$, **en n'utilisant que l'addition**.

4.1 Compléter la relation de récurrence :

- $a \times 0 = \dots\dots$

- pour $k \geq 1$, $a \times k = a \times (k - 1) + \dots$

4.2 Écrire une fonction `multiplie(a, b)` qui calcule $a \times b$ à l'aide d'une boucle `for`, sans utiliser l'opérateur `*`.

4.3 Compléter le tableau de suivi pour l'appel `multiplie(7, 4)`, en notant `p` la variable qui contient le résultat.

Nombre de tours <code>k</code>	Valeur de <code>p</code>
0	
1	
2	
3	
4	

4.4 Compléter l'invariant : « après `k` tours, `p` = ... ».

4.5 Prouver que `multiplie(a, b)` renvoie bien $a \times b$ (initialisation, conservation, conclusion).

Exercice 5 – Factorielle (boucle `for`)

Pour un entier $n \geq 1$, $n! = 1 \times 2 \times \dots \times n$, et par convention $0! = 1$.

5.1 Compléter la relation de récurrence :

- $0! = \dots$
- pour $k \geq 1$, $k! = \dots \times (k - 1)!$

5.2 Écrire une fonction `factorielle(n)` à l'aide d'une boucle `for`.

5.3 Pourquoi la variable qui accumule le résultat doit-elle être initialisée à `1` et non à `0` ? Vérifier que votre fonction renvoie bien `1` pour `factorielle(0)`.

5.4 Réaliser le tableau de suivi pour l'appel `factorielle(5)`.

5.5 Énoncer un invariant de boucle (cette fois sans aide), puis prouver que la fonction renvoie bien $n!$.

Exercice 6 – Division euclidienne par soustractions (boucle `while`)

Pour deux entiers $a \geq 0$ et $b > 0$, la division euclidienne de a par b consiste à trouver le **quotient** `q` et le **reste** `r` tels que :

$$a = b \times q + r \text{ et } 0 \leq r < b$$

On s'interdit les opérateurs `//`, `%` et `/` : seules l'addition et la soustraction sont autorisées.

6.1 Compléter la relation de récurrence :

- si $a < b$: le quotient vaut et le reste vaut
- si $a \geq b$: `quotient(a, b) = ..... + quotient(a - b, b)` et `reste(a, b) = reste(....., b)`.

6.2 Écrire une fonction `division(a, b)` qui renvoie le couple `(q, r)` à l'aide d'une boucle `while`. On pourra partir de `q = 0` et `r = a`.

6.3 Compléter le tableau de suivi pour l'appel `division(17, 5)`.

Nombre de tours	<code>q</code>	<code>r</code>	$b \times q + r$
0			
1			
2			
3			

Que remarque-t-on dans la dernière colonne ?

6.4 Montrer que la propriété « $a = b \times q + r$ et $r \geq 0$ » est un invariant de la boucle (initialisation et conservation).

6.5 À la sortie de la boucle, que sait-on de r grâce à la condition du `while` ? En déduire, avec l'invariant, que q et r sont bien le quotient et le reste de la division euclidienne.

6.6 Terminaison : montrer que r est un variant de la boucle. Que se passe-t-il si on appelle `division(17, 0)` ? Quelle partie de la preuve n'est plus valable ?

Exercice 7 ★ *Approfondissement — Du binaire au décimal : la méthode de Horner*

On représente un nombre binaire par une chaîne de caractères `'0'` et `'1'`, bit de poids fort en premier. Par exemple, `"1011"` représente le nombre 11.

On remarque que :

$$1011 \text{ (base 2)} = 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 = ((1 \times 2 + 0) \times 2 + 1) \times 2 + 1$$

7.1 Donner la valeur décimale de `"1"`, `"10"`, `"101"` et `"1011"`. Quelle relation lie la valeur de `"101"` à celle de `"1011"` ? Généraliser en complétant, pour une chaîne `s` et un bit `c` :

- valeur de la chaîne vide `" "` :
- valeur de `s + c` = ... $\times$ valeur de `s` + ...

7.2 Écrire une fonction `binaire_vers_decimal(s)` qui utilise une boucle `for c in s`. On rappelle que `int('1')` vaut 1 et `int('0')` vaut 0.

7.3 Réaliser le tableau de suivi pour l'appel `binaire_vers_decimal("1011")`.

7.4 Énoncer un invariant de boucle portant sur `s[:k]` (les k premiers caractères), puis le prouver.

7.5 Pour un nombre de n bits, combien de multiplications fait la méthode de Horner ? Comparer avec une méthode qui calculerait chaque puissance de 2 séparément par multiplications successives.

1.4 Bilan

Compléter le tableau récapitulatif.

Algorithme	Relation de récurrence	Cas de départ	Invariant	Variant (si <code>while</code>)
Multiplication				
Factorielle				
Division euclidienne				
Horner				

Pour la prochaine séance. Dans chaque ligne, la relation de récurrence exprime le résultat au rang `k` à partir du résultat au rang `k - 1`. Imaginons qu'une fonction Python puisse **s'appeler elle-même** pour obtenir ce résultat au rang `k - 1`. Essayer d'écrire `factorielle(n)` de cette façon, sans boucle.