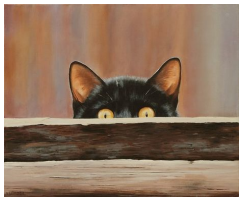
CHAPITRE 1

Algorithmique



I. What - How - Why - How Fast

Pour qu'un algorithmique soit pleinement fonctionnel on doit se poser les questions « What - How - Why - How Fast » autrement dit :

a) What ? ou les spécifications

- On définit les **spécifications** :
  Rôle : quel est le but de l'algorithme
  Entrée(s) : quelles sont les données nécessaires pour l'exécution de l'algorithme
  Précondition(s) : quelles sont les conditions sur les données en entrée
  Sortie(s) : quelles sont les données qui sont seront retournées par l'algorithme
  Postcondition(s) : quelles sont les conditions sur les données en sortie

b) How ? ou la correction partielle

- On procède à la **correction partielle** :
  L'algorithme fait-il ce pour quoi il a été créé ? Autrement dit, si les entrées respectent les préconditions, est-ce que les sorties vont respecter les postconditions ?

c) Why ? ou la terminaison

- On étudie la **terminaison** de l'algorithme :
  autrement dit, on vérifie que l'algorithme s'arrête après un certain nombre d'étapes.

d) How fast ? ou la complexité

- On étudie la **complexité** de l'algorithme :
  Complexité temporelle : quel est l'efficacité (on pourrait presque dire la « rapidité ») d'exécution de l'algorithme. En première approximation, on peut l'étudier en comptant le nombre d'opérations élémentaires exécutées en fonction de la taille des données en entrée.

Remarque
En terme de complexité, on peut aussi étudier la complexité spatiale et l'énergie dépensée pour l'exécution d'un programme. Ces questions ne sont pas étudiée au lycée.

- **Complexité spatiale :** quel est la taille mémoire nécessaire à l'exécution de l'algorithme en fonction de la taille des données en entrée.
- **L'énergie :** pour arriver au résultat (écrire l'algorithme puis l'exécuter).
  Cette question n'est pas théorique, on pense que le numérique consomme déjà 10% de l'électricité consommée mondialement.
  `https://lejournal.cnrs.fr/articles/numerique-le-grand-gachis-energetique`.

II. Algorithmes vus en première et à connaître par cœur

a) Déterminer le minimum ou le maximum d'un tableau

Les spécifications

Entrée(s) : .....
.....

Précondition(s) : .....
.....

Sortie(s) : .....
.....

Postcondition(s) : .....
.....

L'algorithme

En pseudo-code :

```
1 mini ← T[0];
2 pour i allant de 1 à taille du tableau T - 1 faire
3 |   si T[i] < mini alors
4 |   |   mini ← T[i];
5 |   fin si
6 fin pour
```

En Python :

```
1 mini = T[0]
2 for i in range(1,len(T)):
3     if T[i] < mini :
4         mini = T[i]
```

La correction partielle

Il faut avouer que le raisonnement ci-dessous n'est pas simple : la **correction partielle** est le point le plus délicat de ce cours. On doit pouvoir vérifier que l'algorithme effectue bien ce pour quoi il a été conçu.
Les séquences d'instructions simples (lignes 1 ou 4) ne posent pas de problèmes.
Les conditionnelles (**Si ... Alors ... Sinon ...**) conduisent à un raisonnement par cas **branche alors** (ligne 4).
On peut donc facilement vérifier ce que fait l'algorithme.

Le problème se situe donc dans les répétitives, autrement dit dans les boucles. Ici, on a une boucle **POUR** (lignes 2 à 4).

Pour y parvenir, on met en place un **raisonnement inductif** autour de ce qu'on appelle un **invariant**, qui :
- est un prédicat sur les variables impliquées dans la répétitive,
- est vrai juste avant la répétitive,
  (induite par la précondition et les instructions en début d'algorithme)
- est vrai après chaque itération de la répétitive
  (induite par l'invariant et condition+instructions de l'itération courante)
- sera vrai après la fin de la répétitive

La première étape consiste donc à déterminer cet invariant. Pour ce faire, on doit s'inspirer fortement de la postcondition et imaginer où en serons-nous quand la boucle `POUR` ne sera pas terminé et donc raisonner par rapport à la variable `i` ?
L'invariant est donc ici tout trouvé :

mini est une valeur du tableau T
mini est la valeur minimale des cases allant de $T[0]$ à $T[i]$.

Procédons maintenant à la correction partielle. Il *suffit* de vérifier que l'**invariant** reste vrai à chaque étape de l'algorithme. Une des meilleure façon de procéder est peut-être d'utiliser un tableau.

Avant la première boucle

	Valeur de i	Valeur de *mini*	Invariant
Ligne 1	i n'existe pas	*mini* vaut $T[0]$	.....
			.....
			Invariant .....

Première boucle ($i = 1$) :

	Valeur de i	Valeur de *mini*	Invariant
Ligne 3	$i = 1$	*mini* vaut $T[0]$	
Ligne 4	$i = 1$	si $T[1] < mini$,	.....
		$mini = T[1]$	.....
			Invariant .....
Ligne 4	$i = 1$	sinon rien	.....
		$mini = T[0]$	.....
			Invariant .....

Deuxième boucle ($i = 2$) :

	Valeur de i	Valeur de *mini*	Invariant
Ligne 3	$i = 2$	*mini* est le minimum de $T[0]; T[1]$	
Ligne 4	$i = 2$	si $T[2] < mini$,	.....
		$mini = T[2]$	.....
			Invariant .....
Ligne 4	$i = 2$	sinon rien	.....
		mini est inchangé	.....
			Invariant .....

On pourrait continuer ainsi à la main, étape après étape. Le problème c'est que nous ne savons pas combien il y a d'étapes puisque cela dépend de la taille du tableau qui nous est inconnue. Il faut faire ce type de raisonnement quelque soit cette taille. Pour cela il faut imaginer ce qui va se passer pour une boucle quelconque et donc pour une valeur quelconque de la variable i entre 1 et la taille du tableau -1.
Oui, mais juste avant, toutes les boucles pour de 1 à $i - 1$ ont été exécuté et si tout va bien elles ont respectées l'invariant. Ainsi on devrait avoir avant d'attaquer cette boucle i :

mini est une valeur du tableau T
mini est la valeur minimale des cases allant de $T[0]$ à $T[i-1]$.

boucle (i) :

	Valeur de i	Valeur de *mini*	Invariant
Ligne 3	i	*mini* est le minimum de $T[0]; T[1]; ...; T[i-1]$	
Ligne 4	i	si $T[i] < mini$,	.....
		$mini = T[i]$	.....
			Invariant .....
Ligne 4	i	sinon rien	.....
		mini est inchangé	.....
			Invariant .....

Il faut bien évidemment finir la répétitive en regardant la dernière boucle. Juste avant, i avait pour valeur $n - 2$ où n représente la taille du tableau.

mini est une valeur du tableau T
mini est la valeur minimale des cases allant de $T[0]$ à $T[n-2]$.

dernière boucle ($i = n - 1$) :

	Valeur de i	Valeur de *mini*	Invariant
Ligne 3	$n-1$	*mini* est le minimum de $T[0]; ...; T[n-2]$	
Ligne 4	$n-1$	si $T[i] < mini$,	.....
		$mini = T[i]$	.....
			Invariant .....
Ligne 4	$n-1$	sinon rien	.....
		mini est inchangé	.....
			Invariant .....

Nous obtenons bien le résultat escompté, notre algorithme fera bien ce que nous lui avons demandé. Autrement dit, en respectant les préconditions, nous obtenons des sorties qui respectent les postconditions.

La terminaison

Il faut vérifier que l'algorithme se termine bien.

- Tout algorithme sans appel de fonction ni répétitive se termine.
- Toute répétitive **POUR** se termine en un nombre fini d'étapes.

Le problème peut se poser uniquement pour les répétitives **TANT ... QUE**.
Vous vous rappelez du programme ci-contre ?

```
1 i = -1
2 while i <= 0  :
3     i = i - 1
```

Donc ici pas de souci, c'est une boucle **POUR**, l'algorithme se termine.

La complexité

Pour mesurer l'efficacité d'un algorithme, on parle de **complexité**. Dans notre cas, une chose est certaine, cette complexité va dépendre de la taille n du tableau.
On va, en première approximation compter le nombre d'opérations élémentaires effectuées par notre algorithme.

En pseudo-code :

```
1 mini ← T[0];
2 pour i allant de 1 à taille du tableau T - 1 faire
3 |   si T[i] < mini alors
4 |   |   mini ← T[i];
5 |   fin si
6 fin pour
```

Nombre d'opérations :
ligne 1 : .....
ligne 2 : .....
ligne 3 : .....
ligne 4 : .....

Nous obtenons donc ..... opérations.
Comme ce nombre d'opérations est une fonction affine, on dit qu'il s'agit d'une complexité linéaire.

Lorsque la complexité s'exprime sous la forme d'une fonction affine, on dit que

la complexité est linéaire

et on la notera

$$O(n)$$

On appelle cette notation, la notation de Landau.
Si vous souhaitez en savoir plus sur l'étude de la complexité d'un algorithme et si vous aimez (énormément) les maths, vous pouvez aller visiter la page wikipédia suivante :
`https://fr.wikipedia.org/wiki/Comparaison_asymptotique`.

Exercice 1

On suppose que l'on dispose devant soi de cartes tirées uniquement parmi les atouts [1] d'un jeu de Tarot.
On souhaite rechercher la valeur maximale et sa place.

- Donner les spécifications de votre algorithme.
- Proposer votre algorithme.
- Étudier la correction partielle de votre algorithme.
- Étudier la terminaison de votre algorithme.
- Étudier la complexité de votre algorithme.

1. Les valeurs vont uniquement de 1 à 21.

Retour sur la terminaison

Pour résoudre cet exercice 1 en tenant compte des préconditions, vous avez très certainement utilisé une boucle **TANT ... QUE**.
Pour établir sa terminaison, on doit identifier un **variant** [2] de la répétitive.

Le **variant** est une expression :
- impliquée dans la répétitive ;
- à valeur entière ;
- qui évolue de façon strictement décroissante au cours des itérations ;
- qui est positive ou nulle du fait des initialisations et de la condition de la répétitive.

Pour votre algorithme, le candidat variant est ..... :
- .....est impliquée dans la boucle **TANT ... QUE**, lignes ..... ;
- .....est un entier ;
- à chaque boucle, ...
  Donc .....est strictement décroisant ;
- .....
  .....
  Donc .....est positive ou nulle.

Donc votre algorithme se termine bien.

Retour sur la complexité

Comme déjà dit, vous avez très certainement utilisé une boucle **TANT ... QUE**.
Cela parait alors difficile d'étudier la complexité de l'algorithme puisqu'on ne sait pas combien de boucles on va avoir besoin. Par exemple si la première carte est de valeur 21, il n'y aura aucune boucle. Par contre si toutes les cartes sont de valeurs inférieurs et la dernière est le 21, il y aura autant $n-1$ boucles, n étant le nombre de cartes.
Le choix qui est fait le plus souvent est d'imaginer le pire cas [3] puis d'effectuer l'étude de la complexité dans ce cas.

Dans notre exercice 1, le pire cas est .....

b) Déterminer la somme des valeurs d'un tableau

Les spécifications

Entrée(s) : .....
.....

Précondition(s) : .....
.....

Sortie(s) : .....
.....

Postcondition(s) : .....
.....

2. Ne pas confondre avec l'**in**variant !
3. On peut aussi étudier le meilleur cas, ou le cas moyen. On pourra discuter de l'intérêt du pire, du meilleur ou du cas moyen.

L'algorithme

En pseudo-code :

```
somme ← T[0];
pour i allant de 1 à taille du tableau T - 1 faire
    somme ← somme + T[i]
fin pour
```

En Python :

```
somme = T[0]
for i in range(1,len(T)):
    somme = somme + T[i]
```

Exercice 2

On se donne l'algorithme de calcul de la somme des valeurs d'un tableau vu ci-dessus.

- Étudier la correction partielle de votre algorithme.
- Étudier la terminaison de votre algorithme.
- Étudier la complexité de votre algorithme.

Exercice 3

Modifier légèrement l'algorithme précédent pour obtenir un algorithme de calcul de la moyenne des valeurs d'un tableau.

Exercice 4

Voici deux algorithmes qui prétendent effectuer la multiplication de a par b :

Algorithme 1 :

```
Entrées : deux nombres entiers a et b
Sorties : le résultat de a × b
m ← 0 ;
tant que b > 0 faire
    m ← m + a ;
    b ← b − 1 ;
fin tq
retourner m ;
```

Algorithme 2 :

```
Entrées : deux nombres entiers a et b
Sorties : le résultat de a × b
i ← 0 ;
m ← 0 ;
tant que i < b faire
    m ← m + a ;
    i ← i + 1 ;
fin tq
retourner m ;
```

1. Donner le variant de chacun de ces deux algorithmes.
2. Ces deux algorithmes terminent-ils ?

c) Trier les valeurs d'un tableau par la méthode du tri par insertion

Le principe

C'est un des tris les plus simples.
On parcourt le tableau de la gauche vers la droite et en maintenant une partie déjà triée sur la gauche :

déjà trié									pas encore trié			
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[j]$	...	$T[i-2]$	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[n-1]$

On va alors prendre la valeur non encore triée la plus à gauche, $T[i]$. On va la comparer à la première valeur triée juste en-dessous, $T[i-1]$.

déjà trié									pas encore trié			
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[j]$	...	$T[i-2]$	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[n-1]$

Deux solutions :

- Soit $T[i-1] \leq T[i]$, alors $T[i]$ est à sa bonne place et on ne fait rien :

déjà trié										pas encore trié		
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[j]$	...	$T[i-2]$	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[n-1]$

- Soit $T[i-1] > T[i]$, alors, on redescend la valeur de $T[i]$ d'une case :

déjà trié										pas encore trié		
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[j]$	...	$T[i-2]$	$T[i]$	$T[i-1]$	$T[i+1]$	...	$T[n-1]$

puis on recommence en comparant les valeurs $T[i-2]$ et $T[i]$:

déjà trié										pas encore trié		
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[j]$	...	$T[i-2]$	$T[i]$	$T[i-1]$	$T[i+1]$	...	$T[n-1]$

...

en poursuivant ainsi, de proche en proche, on trouvera une valeur $T[j-1] \leq T[i]$ et $T[i]$ obtiendra sa place dans la partie du tableau triée.

déjà trié										pas encore trié		
$T[0]$	$T[1]$	$T[2]$	...	$T[j-1]$	$T[i]$	$T[j]$	...	$T[i-2]$	$T[i-1]$	$T[i+1]$	...	$T[n-1]$

Un exemple

Revoyons la méthode, pas à pas, avec un exemple.
Nous voulons trier le tableau T de longueur 6 suivant :

85	1	20	33	70	57

On va de la gauche, vers la droite. On commence donc par la première case. Le nombre est bien rangé (normal, il est tout seul).

déjà trié	pas encore trié				
85	1	20	33	70	57

On prend le nombre non trié le plus à gauche : 1. On le compare au nombre rangé le plus grand, autrement dit 85.

déjà trié	pas encore trié				
85	1	20	33	70	57

85>1, on va donc descendre le nombre 1.

déjà trié		pas encore trié			
1	85	20	33	70	57

Il n'y a plus rien à comparer, on a terminé pour le nombre 1.

déjà trié		pas encore trié			
1	85	20	33	70	57

On prend le nombre non trié le plus à gauche : 20. On le compare au nombre rangé le plus grand, autrement dit 85.

déjà trié		pas encore trié			
1	85	20	33	70	57

85>20, on va donc descendre le nombre 20.

déjà trié			pas encore trié		
1	20	85	33	70	57

On compare ensuite 20 avec 1. $1 \leq 20$. On a terminé pour le nombre 20.

déjà trié			pas encore trié		
1	20	85	33	70	57

...
et si vous poursuiviez le tri à la main pour cette méthode ? ? ? ? ?

Les spécifications

Entrée(s) : .....
.....

Précondition(s) : .....
.....

Sortie(s) : .....
.....

Postcondition(s) : .....
.....

L'algorithme

L'algorithme de tri par insertion :

```
1 pour i allant de 1 à n − 1 faire
2 |  j ← i ;
3 |  tant que j > 0 et T[j − 1] > T[j] faire
4 |  |  temp ← T[j − 1] ;
5 |  |  T[j − 1] ← T[j] ;
6 |  |  T[j] ← temp ;
7 |  |  j ← j − 1 ;
8 |  fin tq
9 fin pour
```

La version en langage Python :

```
1 for i in range(1,n) :
2     #on prend un nouvel indice
3     j = i
4     while (j > 0) and (T[j-1] > T[j]) :
5         #tant qu'à gauche, il y a un élé
      ment plus grand
6         #on les échange
7         T[j], T[j-1] = T[j-1], T[j]
8         j = j - 1
```

On peut tester ce programme en ligne :
https://github.com/NaturelEtChaud/NSI-Terminale/blob/main/1Algorithmique/.

Correction partielle

Pour déterminer l'invariant on va s'inspirer fortement de la postcondition et imaginer les différentes étapes d'exécution de l'algorithme.
Pour mémoire, la postcondition est :

> Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
> ET
> Si $0 \leq p \leq k \leq n-1$ alors $T[p] \leq T[k]$.

On peut s'en inspirer pour déterminer l'invariant à la i-ème étape. Comme alors nous n'avons trié que les cases du tableau de $T[0]$ à $T[i]$, on a :

> Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
> ET
> Si $0 \leq p \leq k \leq i$ alors $T[p] \leq T[k]$.

1. Initialisation de l'invariant : pour $i = 0$.
   On n'effectue pas la boucle **POUR**. On ne regarde que la première valeur du tableau et pour l'invariant :

   > Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
   > ET
   > Si $0 \leq p \leq k \leq 0$ alors $T[p] \leq T[k]$.

   donc $p = k = 0$ autrement dit $T[p] = T[k] = T[0]$.
   L'invariant est vrai.

2. Regardons le passage de la $(i-1)$-ième étape à la i-ième étape, afin de vérifier l'invariant. Supposons donc que l'invariant est vrai à la $(i-1)$-ième étape. Autrement dit, on a :

   > Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
   > ET
   > Si $0 \leq p \leq k \leq i-1$ alors $T[p] \leq T[k]$.

   Déroulons l'algorithme à la i-ème étape.
   Ligne 2, $j \leftarrow i$, donc j prend pour valeur i.
   Ligne 3, pour la boucle **TANT ... QUE**, on a deux possibilités :
   - Si $T[j-1] \leq T[j]$, la condition d'entrée dans la boucle n'est pas respectée. Donc on ne fait rien.
     Comme $i = j$, on a $T[j-1] \leq T[j] = T[i]$. L'invariant devient :

     > Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
     > ET
     > Si $0 \leq p \leq k \leq i$ alors $T[p] \leq T[k]$.

     L'invariant est vrai.
   - Si $T[j-1] > T[j]$, on va échanger dans la boucle **TANT ... QUE** les valeurs $T[j-1]$ et $T[j]$. A chaque échange, les valeurs avant et après $T[j]$ (jusqu'à $T[i]$) sont triées.
     Comme on échange jusqu'à ce qu'enfin $T[j-1] \leq T[j]$, quand on sort de la boucle **TANT ... QUE**, l'invariant obtenu est :

     > Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
     > ET
     > Si $0 \leq p \leq k \leq i$ alors $T[p] \leq T[k]$.

     L'invariant est vrai.

La correction partielle est terminée. L'algorithme trie bien les valeurs du tableau.

Terminaison

On l'a déjà vu le problème réside uniquement dans la boucle **TANT ... QUE**, lignes 3 à 7 du pseudo-code (liges 4 à 8 du programme en Python).
Pour établir sa terminaison, on doit identifier un **variant**.

Ici le candidat variant est ..... :

- ..... est impliquée dans la boucle **TANT ... QUE**, lignes ..... ;
- ..... est un entier ;
- à chaque boucle, ...
  Donc ..... est strictement décroisant ;
- .....
  .....
  Donc ..... est positive ou nulle.

Donc votre algorithme se termine bien.

Complexité

L'algorithme de tri par insertion :

```
1 pour i allant de 1 à n − 1 faire
2 |  j ← i ;
3 |  tant que j > 0 et T[j − 1] > T[j] faire
4 |  |  temp ← T[j − 1] ;
5 |  |  T[j − 1] ← T[j] ;
6 |  |  T[j] ← temp ;
7 |  |  j ← j − 1 ;
8 |  fin tq
9 fin pour
```

On compte toutes les opérations :

Ligne 1 à 9 : on répète ..... fois
- **ligne 1 :** 1 opération
  (on incrémente i de 1 jusqu'à $n-1$)
- **ligne 2 :** 1 opération
  (on affecte une valeur à j, $j \leftarrow i$)

Lignes 3 à 8 : on répète ..... fois
- **ligne 3 :** 2 opérations
  (on effectue deux tests pour savoir si on boucle)
- **ligne 4 :** 1 opération
  (on affecte une valeur à $temp$)
- **ligne 5 :** 1 opération
  (on affecte une valeur à $T[j-1]$)
- **ligne 6 :** 1 opération
  (on affecte une valeur à $T[j]$)
- **ligne 7 :** 2 opérations
  (on soustrait puis on affecte une valeur à j)

La question qui demeure est : « combien de fois répète-t-on les lignes 3 à 8 ? ». La réponse est malheureusement « ça dépend ! » mais cette réponse ne nous convient pas. Heureusement, ce que l'on recherche est plutôt la réponse à cette question :

« Combien de fois répète-t-on les lignes 3 à 8 dans le pire cas ? »

Ce qui nous amène à une autre question :

« Quel est le pire cas ? »

Réponse :

« Lorsque le tableau est rangé ..... ! »

En effet à ce moment là, la valeur non trié la plus à gauche doit être décalée, étape après étape, pour se retrouver sur la toute première case du tableau $T[0]$ [4].

4. C'est vraiment dommage, si on recherche la valeur maximale du tableau et si on sait que le tableau est rangé dans l'ordre décroissant, ce n'est pas la peine de ranger le tableau dans l'ordre croissant, n'est-ce pas ?

Comptons alors le nombre de répétitions des lignes 3 à 8.

- Si $i = 1$, on effectue une seule permutation entre $T[0]$ et $T[1]$.
  1 seule répétition.
- Si $i = 2$, on doit faire passer la valeur $T[2]$ en $T[0]$ avec deux permutations.
  2 répétitions.
- Si $i = 3$, on doit faire passer la valeur $T[3]$ en $T[0]$ avec trois permutations.
  3 répétitions.
  ...
- Pour i, on doit faire passer la valeur $T[i]$ en $T[0]$ avec i permutations.
  i répétitions.
  ...
- Si $i = (n-1)$, on doit faire passer la valeur $T[n-1]$ en $T[0]$ avec $(n-1)$ permutations.
  $(n-1)$ répétitions.

Donc nous avons au total :

$$S = 1 + 2 + 3 + ... + i + ... + (n-1)$$

Nos amis mathématiciens ont appris une formule [5] qui nous permet de dire que :

$$S = \frac{(n-1)n}{2}$$

Nous avons donc le nombre de répétitions dans le pire des cas. On peut maintenant compter le nombre total d'opérations :

$$Nb = \underbrace{(n-1) \times 1}_{\text{les répétitions de la ligne 1}} + \underbrace{\frac{(n-1)n}{2} \times 7}_{\text{les répétitions de toutes les boucles } \textbf{TANT ... QUE}}$$

$$Nb = n - 1 + \frac{7}{2}n^2 - \frac{7}{2}n$$

$$Nb = \frac{7}{2}n^2 - \frac{5}{2}n - 1$$

Le nombre d'opérations obtenu n'est pas une fonction affine. C'est une fonction polynôme du second degré.

Lorsque le nombre d'opérations d'un algorithme est une fonction polynôme du second degré, le nombre d'opérations est dans la famille des n^2.
On dit alors que la **complexité est quadratique** et on la note $O(n^2)$.

Autrement dit :
Si le tableau est de taille $n = 10$, comme $n^2 = 100$, on aura des centaines d'opérations pour ranger le tableau.
Si le tableau est de taille $n = 100$, comme $n^2 = 10\ 000$, on aura plusieurs dix milliers d'opérations pour ranger le tableau.
Si le tableau est de taille $n = 1\ 000$, comme $n^2 = 1\ 000\ 000$, on aura des millions d'opérations pour ranger le tableau.

5. $S = 1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}$.

d) Tri par sélection

Le principe

C'est a aussi un des tris les plus simples.
Le tri par sélection consiste à sélectionner la valeur que l'on souhaite ranger au lieu de prendre la première venue.
On maintient toujours une partie déjà triée sur la gauche et, parmi les valeurs non encore triées du tableau, on va sélectionner celle qui est la plus petite. On placera alors cette valeur comme la plus grande des valeurs triées :

déjà trié					pas encore trié							
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[j-1]$	$T[j]$	$T[j+1]$	...	$T[n-1]$

On regarde toutes les valeurs non encore triées et on cherche la plus petite.

déjà trié					pas encore trié							
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[j-1]$	$T[j]$	$T[j+1]$	...	$T[n-1]$

Deux solutions :

- Soit $T[i]$ est la valeur la plus petite des valeurs non triées, alors $T[i]$ est à sa bonne place et on ne fait rien :

déjà trié						pas encore trié						
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[j-1]$	$T[j]$	$T[j+1]$	...	$T[n-1]$

- Soit $T[j]$ est la valeur la plus petite :

déjà trié					pas encore trié							
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[i]$	$T[i+1]$	...	$T[j-1]$	$T[j]$	$T[j+1]$	...	$T[n-1]$

On échange $T[j]$ avec la première valeur non triée $T[i]$:

déjà trié					pas encore trié							
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[j]$	$T[i+1]$	...	$T[j-1]$	$T[i]$	$T[j+1]$	...	$T[n-1]$

La valeur $T[j]$ est alors à sa place :

déjà trié						pas encore trié						
$T[0]$	$T[1]$	$T[2]$	...	$T[i-1]$	$T[j]$	$T[i+1]$	...	$T[j-1]$	$T[i]$	$T[j+1]$	...	$T[n-1]$

On pousuit ensuite avec les valeurs non encore triées.

Un exemple

Revoyons la méthode, pas à pas, avec notre exemple :

85	1	20	33	70	57

On regarde tous les nombres et on sélectionne le plus petit ce sera 1 :

85	1	20	33	70	57

On le met alors en première position :

déjà trié	pas encore trié				
1	85	20	33	70	57

Parmi tous les nombres non encore triées, on sélectionne le plus petit, 20 :

déjà trié	pas encore trié				
1	85	20	33	70	57

On l'échange avec le premier nombre non trié, 85 :

déjà trié		pas encore trié			
1	20	85	33	70	57

...

et si vous poursuiviez le tri à la main pour cette méthode ? ? ? ? ?

Les spécifications

Entrée(s) : .....

.....

Précondition(s) : .....

.....

Sortie(s) : .....

.....

Postcondition(s) : .....

.....

L'algorithme

L'algorithme doit faire 2 choses. Chercher le minimum parmi les valeurs non triées puis l'échanger avec la première valeur non encore triée.
L'algorithme de tri par sélection :

```
1  pour i allant de 0 à n − 1 faire
2  |  indice_min ← i ;
3  |  pour j allant de i + 1 à n − 1 faire
4  |  |  si T[indice_min] > T[j] alors
5  |  |  |  indice_min ← j ;
6  |  |  fin si
7  |  fin pour
8  |  temp ← T[i] ;
9  |  T[i] ← T[indice_min] ;
10 |  T[indice_min] ← temp ;
11 fin pour
```

La version en langage Python :

```
1 for i in range(0,n) :
2     indice_min = i
3         #on cherche le minimum à partir de T[i]
4     for j in range(i+1,n) :
5                 if T[indice_min] > T[j] :
6                         indice_min = j
7         #on échange
8     T[indice_min], T[i] = T[i], T[indice_min]
```

On peut tester ce programme en ligne :
`https://github.com/NaturelEtChaud/NSI-Terminale/blob/main/1Algorithmique/`.

Dans l'algorithme, aux lignes 8 à 10, on échange les valeurs de $T[indice\_min]$ et $T[i]$. Il est nécessaire d'avoir une variable temporaire.
Pour l'écriture en Python, il n'y a pas de problème. On peut faire directement l'échange à la ligne 8 car on a utilisé des

.....

Correction partielle

Comme pour le tri par insertion, on garde à gauche une partie triée et à droite une partie non encore triée. On propose donc le même **invariant** tout en rajoutant une idée supplémentaire : la partie triée ne contient que des nombres plus petit que la partie non triée.

> Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
> ET
> Si $0 \leq p \leq k \leq i$ alors $T[p] \leq T[k]$
> ET
> Pour tout $0 \leq k \leq i$ et tout $i+1 \leq j \leq n-1$, $T[k] \leq T[j]$.

1. Initialisation de l'invariant : pour $i = 0$.
   On recherche le minimum de toutes les valeurs et on le met à la position 0 :

   > Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
   > ET
   > Si $0 \leq p \leq k \leq 0$ alors $T[0] = T[p] \leq T[k] = T[0]$ OK
   > ET
   > Pour tout $0 \leq k \leq 0$ et tout $1 \leq j \leq n-1$, $T[0] \leq T[j]$ OK puisque c'est le minimum

   L'invariant est vrai.

2. Regardons le passage de la $(i-1)$-ième étape à la i-ième étape, afin de vérifier l'invariant. Supposons donc que l'invariant est vrai à la $(i-1)$-ième étape. Autrement dit, on a :

   > Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
   > ET
   > Si $0 \leq p \leq k \leq i-1$ alors $T[p] \leq T[k]$
   > ET
   > Pour tout $0 \leq k \leq i-1$ et tout $i \leq j \leq n-1$, $T[k] \leq T[j]$.

   Déroulons l'algorithme à la i-ème étape.
   On cherche $indice\_min$, l'indice de la valeur minimale de T entre $i+1$ et $n-1$.
   Comme toutes les valeurs de $T[i]$ à $T[n-1]$ sont toutes plus grandes que toutes les valeurs de $T[0]$ à $T[i-1]$, on en déduit que $T[indice\_min]$ est bien plus grande que toutes les valeurs de $T[0]$ à $T[i-1]$.
   Ensuite on échange $T[indice\_min]$ et $T[i]$, donc $T[i]$ est bien plus grande que toutes les valeurs de $T[0]$ à $T[i-1]$.
   Comme de plus toutes les valeurs de $T[0]$ à $T[i-1]$ sont triées, on en déduit que toutes les valeurs de $T[0]$ à $T[i]$ sont triées.
   Soit :

   > Les valeurs de T sont les mêmes en entrée et en sortie (mais dans un ordre différent)
   > ET
   > Si $0 \leq p \leq k \leq i$ alors $T[p] \leq T[k]$
   > ET
   > Pour tout $0 \leq k \leq i$ et tout $i+1 \leq j \leq n-1$, $T[k] \leq T[j]$.

   L'invariant est vrai.

La correction partielle est terminée. L'algorithme trie bien les valeurs du tableau.

Terminaison

Une boucle **POUR** termine toujours. Ici, il y en a deux. Donc cet algorithme termine.

Complexité

L'algorithme de tri par sélection :

```
1  pour i allant de 0 à n − 1 faire
2  |  indice_min ← i ;
3  |  pour j allant de i + 1 à n − 1 faire
4  |  |  si T[indice_min] > T[j] alors
5  |  |  |  indice_min ← j ;
6  |  |  fin si
7  |  fin pour
8  |  temp ← T[i] ;
9  |  T[i] ← T[indice_min] ;
10 |  T[indice_min] ← temp ;
11 fin pour
```

On compte toutes les opérations :

Ligne 1 à 11 : on répète ..... fois
- **ligne 1 :** 1 opération
  (on incrémente i de 0 jusqu'à $n-1$)
- **ligne 2 :** 1 opération
  (on affecte une valeur à $indice\_min$)
- **Lignes 3 à 6** : on répète ..... fois
  - **ligne 3 :** 1 opération
    (on incrémente j de $i+1$ jusqu'à $n-1$)
  - **ligne 4 :** 1 opération
    (on effectue un test)
    - **ligne 5 :** 1 opération
      (on affecte une valeur à $indice\_min$)
- **ligne 8 :** 1 opération
  (on affecte une valeur à $temp$)
- **ligne 9 :** 1 opération
  (on affecte une valeur à $T[i]$)
- **ligne 10 :** 1 opération
  (on affecte une valeur à $T[indice\_min]$)

La question qui demeure est : « combien de fois répète-t-on la ligne 5 ? ». Comme il est impossible d'y répondre, à nouveau, la question réelle est :

« Combien de fois répète-t-on la ligne 5 dans le pire cas ? »

Ce qui nous ramène toujours à cette autre question :

« Quel est le pire cas ? »

Réponse :

« Lorsque le tableau est rangé dans l'ordre ..... ! »

En effet à ce moment là, la valeur minimale parmi les valeurs non triés est toujours celle qui est la plus à droite.

Comptons alors le nombre de répétitions des lignes 3 à 6.
- Si $i = 0$, on effectue $n-1$ tests pour savoir quelle la valeur minimale.
  $n-1$ répétitions.
- Si $i = 1$, on effectue $n-2$ tests pour savoir quelle la valeur minimale entre $T[1]$ et $T[n-1]$.
  $n-2$ répétitions. ...
- Pour i, on effectue $n-i-1$ tests pour savoir quelle la valeur minimale entre $T[i]$ et $T[n-1]$.
  $n-i-1$ répétitions. ...
- Si $i = (n-2)$, on effectue $n-(n-2)-1 = 1$ test pour savoir quelle la valeur minimale entre $T[n-2]$ et $T[n-1]$.
  1 répétition.

Donc nous avons au total :

$$S = (n-1) + (n-2) + ... + (n-i-1) + ...1$$

A nouveau grâce à nos amis mathématiciens, nous avons donc le nombre de répétitions dans le pire des cas :

$$S = \frac{(n-1)n}{2}$$

On peut maintenant compter le nombre total d'opérations :

$$Nb = \underbrace{n \times 5}_{\text{les répétitions des lignes 1, 2, 8, 9 et 10}} + \underbrace{\frac{(n-1)n}{2} \times 3}_{\text{les répétitions de toutes les lignes 3, 4 et 5}}$$

$$Nb = 5n + \frac{3}{2}n^2 - \frac{3}{2}n$$

$$Nb = \frac{3}{2}n^2 - \frac{7}{2}n$$

Le nombre d'opérations obtenu est une fonction polynôme du second degré.

On dit alors que la **complexité est quadratique**. La complexité est en $O(n^2)$.

e) Un petit bilan sur les 2 algorithmes de tri étudiés

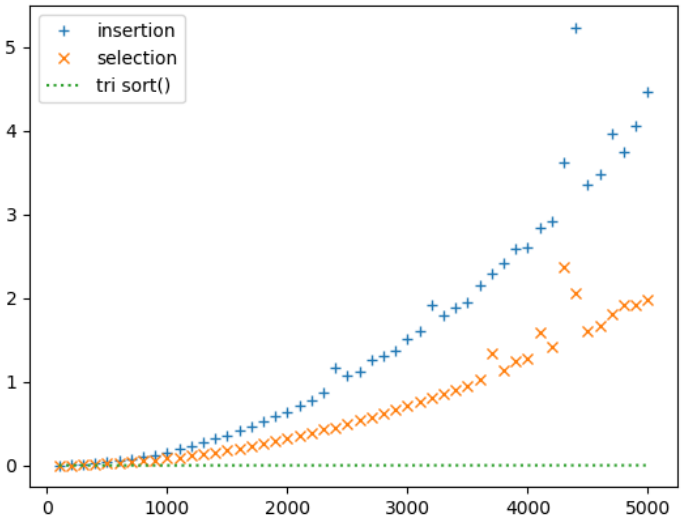
Les deux algorithmes ont la même complexité, ce qui signifie qu'ils sont dans la même famille d'efficacité.
Par contre, on a constaté que le pire cas n'est pas le même pour l'un ou l'autre des algorithmes.
De même, le meilleur cas n'est pas le même non plus.
Ainsi,

- le meilleur cas pour l'algorithme par insertion est : .....
- le meilleur cas pour l'algorithme par sélection est : .....

Ainsi, si le tableau est presque trié par ordre ....., on choisira l'algorithme .....
et si le tableau est presque trié par ordre ....., on choisira l'algorithme .....

De la même manière, mais à un niveau beaucoup plus complexe, lorsqu'on utilise la méthode `.sort()` en Python, cette méthode va d'abord faire une analyse rapide du tableau pour déterminer quel est l'algorithme qui sera le plus efficace en fonction de la configuration du tableau [6].

Même si ce n'est pas une preuve, on peut installer une compteur/horloge au sein d'un programme en Python, pour tester les différents algorithmes et leur efficacité. Attention, ce programme met un certain temps avant de terminer tous les tests :



6. Pour ceux qui en veulent toujours plus, la méthode `.sort()` à une complexité en $O(n\log(n))$ et on ne peut actuellement pas faire mieux.

Exercice 5 (*)

Soient T_1 et T_2 deux tableaux contenant des nombres entiers. Voici deux algorithmes qui calculent la somme de tous les produits de les tous nombres de T_1 par tous les nombres de T_2 :

Algorithme 1 :

Entrées : deux tableaux d'entiers T_1 et T_2
Sorties : la somme du produit des nombres de T_1 par les nombres de T_2

```
somme ← 0 ;
pour i allant de 0 à taille de T1 faire
    pour j allant de 0 à taille de T2 faire
        somme ← somme + T1[i] × T2[j] ;
    fin pour
fin pour
retourner somme ;
```

Algorithme 2 :

Entrées : deux tableaux d'entiers T_1 et T_2
Sorties : la somme du produit des nombres de T_1 par les nombres de T_2

```
somme1 ← 0 ;
somme2 ← 0 ;
pour i allant de 0 à taille de T1 faire
    somme1 ← somme1 + T1[i] ;
fin pour
pour j allant de 0 à taille de T2 faire
    somme2 ← somme2 + T2[j] ;
fin pour
retourner somme1 × somme2 ;
```

1. Expliquer pourquoi ces deux algorithmes produisent le même résultat.
2. Étudier la complexité de ces deux algorithmes en supposant que les deux tableaux soient de même taille n. Faire de même avec tableau T_1 est de taille n et que le tableau T_2 est de table m.

Exercice 6 (**)
Si nous avons deux tableaux d'entiers T_1 et T_2 classés dans l'ordre croissant, l'algorithme suivant permet d'afficher toutes les valeurs de T_1 et de T_2 dans le même ordre :

Entrées : deux nombres tableaux d'entiers T_1 et T_2 classés dans l'ordre croissant
Sorties : toutes les valeurs de T_1 et de T_2 dans l'ordre croissant

```
1  i1 ← 0 ;
2  i2 ← 0 ;
3  tant que i1 < taille du tableau T1 ET i2 < taille du tableau T2 faire
4      si T1[i1] < T2[i2] alors
5          afficher T1[i1] ;
6          i1 ← i1 + 1 ;
7      sinon
8          afficher T2[i2] ;
9          i2 ← i2 + 1 ;
10     fin si
11 fin tq
```

1. Justifier que cet algorithme termine à l'aide de la technique du variant.
2. En supposant que les deux tableaux soient de même taille n, calculer la complexité de cet algorithme.
3. En supposant que le tableau T_1 est de taille n et que le tableau T_2 est de table m recalculer la complexité de cet algorithme.