Chapitre 3 :
Analyse d'algorithme

1) Pourquoi ?

2) Analyse de la correction d'un algorithme

3) Analyse de la terminaison d'un algorithme

1) Pourquoi analyser un algo ?

```
Fun int somme1(int n):
    int s:=1
    int i
    Pour i allant de 1 à n:
        s:=s+i
    Renvoyer s
```

- Cet algorithme renvoie la **somme des entiers de 1 à n :** à chaque étape, on additionne le i (correspondant à l'étape) à la somme qu'on a calculée précédemment

1) Pourquoi analyser un algo ?

```
Fun int somme1(int n):
    int s:=1
    int i
    Pour i allant de 1 à n:
        s:=s+i
    Renvoyer s
```

somme1(1) vaut 2 (au lieu de 1)
somme1(2) vaut 4 (au lieu de 1+2)

FAUX

- Cet algorithme renvoie la **somme des entiers de 1 à n :** à chaque étape, on additionne le i (correspondant à l'étape) à la somme qu'on a calculée précédemment

1) Pourquoi analyser un algo. ?

$\Rightarrow$ Pour vérifier (et convaincre les autres) que l'algorithme répond véritablement à la question posée.

1) Vérifier **la correction d'un algo**. = vérifier qu'il renvoie la réponse correcte
2) Vérifier **la terminaison d'un algo.** = vérifier que l'algorithme termine

   (éventuellement en précisant au bout de combien de "temps"/itérations/opérations)

2) Analyse de la correction d'un algo.

En général, on se concentre sur **l'analyse des boucles** (Pour et Tant Que). On cherche **à écrire une phrase qui reste vraie pendant tout le déroulement de la boucle**.

Vocabulaire: on appelle cette "phrase" une propriété ou encore un **invariant de boucle**.

2) Analyse de la correction d'un algo.

```
Fun int somme2(int n): /* version juste */
    int s:=0
    int i
    Pour i allant de 1 à n:
        s:=s+i
        /* Invariant de boucle:
        à cette étape de l'algorithme, s contient la
         somme des entiers de 1 à i (inclus) */
    Renvoyer s
```

2) Analyse de la correction d'un algo.

Invariant de boucle:

1) Il faut vérifier qu'il est **vrai à la première itération** (= premier passage dans la boucle)
2) Il faut vérifier qu'il **reste vrai d'une itération à l'autre** *(≈ récurrence)*
3) Il faut ensuite vérifier que l'algorithme fait ce qu'on attend de lui grâce à **l'invariant en sortie de boucle** (combiner aux éventuelles instructions qui suivent la boucle avant de quitter la fonction)

2) Analyse de la correction d'un algo.

```
Fun int somme2(int n): /* version juste */
    int s:=0
    int i
    Pour i allant de 1 à n:
        s:=s+i
        /* Invariant de boucle:
        à cette étape de l'algorithme, s contient la
         somme des entiers de 1 à i (inclus) */
    Renvoyer s
```

1e itération:
à cet endroit, i vaut 1 et s vaut 1
→ OK

2) Analyse de la correction d'un algo.

```
Fun int somme2(int n):
    int s:=0
    int i
    Pour i allant de 1 à n:
        s:=s+i
        /* Invariant de boucle:
        à cette étape de l'algorithme, s contient la
         somme des entiers de 1 à i (inclus) */
    Renvoyer s
```

D'une itération à l'autre:
retraduisons l'invariant de boucle aux autres endroits utiles pour notre analyse

Ici, s vaut la somme des entiers de 1 à i-1 (invariant de l'itération précédente)

2) Analyse de la correction d'un algo.

```
Fun int somme2(int n):
    int s:=0
    int i
    Pour i allant de 1 à n:
        s:=s+i
        /* Invariant de boucle:
        à cette étape de l'algorithme, s contient la
         somme des entiers de 1 à i (inclus) */
    Renvoyer s
```

On vérifie la cohérence de nos propriétés à tous les endroits où on les a explicités
→ **OK**

Ici, s vaut la somme des entiers de 1 à i-1 (invariant de l'itération précédente)

2) Analyse de la correction d'un algo.

```
Fun int somme2(int n):
    int s:=0
    int i
    Pour i allant de 1 à n:
        s:=s+i
        /* Invariant de boucle:
        à cette étape de l'algorithme, s contient la
         somme des entiers de 1 à i (inclus) */
    Renvoyer s
```

Ici, juste avant de sortir de la dernière itération de boucle: i vaut n
Donc, grâce à l'invariant, s vaut 1+ … + n
Correction de mon algo → OK

2) Analyse de la correction d'un algo.

ATTENTION

1) Il faut bien préciser à **quel endroit** vous placez votre invariant de boucle

2) Analyse de la correction d'un algo.

```
Fun int somme2(int n):
    int s:=0
    int i
    Pour i allant de 1 à n:
        /* Invariant de boucle:
        à cette étape de l'algorithme, s contient la
         somme des entiers de 1 à i (inclus) */
        s:=s+i
    Renvoyer s
```

FAUX

Par ex. à la 1e itération:
i vaut 1 et s vaut 0

2) Analyse de la correction d'un algo.

ATTENTION

1) Il faut bien préciser à **quel endroit** vous placez votre invariant de boucle
2) Il faut bien vérifier qu'il est **vrai à la 1^{e} itération** (comme le cas de base d'une récurrence)

2) Analyse de la correction d'un algo.

```
Fun int somme1(int n):
    int s:=1
    int i
    Pour i allant de 1 à n:
        s:=s+i
        /* Invariant de boucle:
        à cette étape de l'algorithme, s contient la
         somme des entiers de 1 à i (inclus) */
    Renvoyer s
```

FAUX

A la 1e itération:

i vaut 1 et s vaut 2

2) Analyse de la correction d'un algo.

ATTENTION

1) Il faut bien préciser à **quel endroit** vous placez votre invariant de boucle
2) Il faut bien vérifier qu'il est **vrai à la 1e itération** (comme le cas de base d'une récurrence)
3) Il faut vérifier qu'il est vrai d'une **itération à l'autre**

2) Analyse de la correction d'un algo.

ATTENTION

1) Il faut bien préciser à **quel endroit** vous placez votre invariant de boucle
2) Il faut bien vérifier qu'il est **vrai à la 1^{e} itération** (comme le cas de base d'une récurrence)
3) Il faut vérifier qu'il est vrai d'une **itération à l'autre**
4) Il faut généralement porter un soin particulier à la **dernière itération** → on y reviendra dans un instant

2) Analyse de la correction d'un algo.

ATTENTION

1) Il faut bien préciser à **quel endroit** vous placez votre invariant de boucle
2) Il faut bien vérifier qu'il est **vrai à la 1e itération** (comme le cas de base d'une récurrence)
3) Il faut vérifier qu'il est vrai d'une **itération à l'autre**
4) Il faut généralement porter un soin particulier à la **dernière itération** → on y reviendra dans un instant
5) Une fois votre invariant prouvé, il faut vérifier que votre **algorithme fait ce qui est demandé dans l'énoncé grâce à votre invariant.**

2) Analyse de la correction d'un algo.

```
Fun nombre sommeTab(nombre tab[ ]):
    int i
    nombre sommeJusquIci:=0
    int n:=longu(tab)
    Pour i allant de 0 à n:
        sommeJusquIci := sommeJusquIci + tab[i]
        /* Invariant de boucle :
        sommeJusquIci contient la valeur tab[0]+ … + tab[i] */
    Renvoyer sommeJusquIci
```

1e itération:
i vaut 0
sommeJusquIci vaut tab[0]
→ OK

2) Analyse de la correction d'un algo.

```
Fun nombre sommeTab(nombre tab[ ]):
    int i
    nombre sommeJusquIci:=0
    int n:=longu(tab)
    Pour i allant de 0 à n:
        ici: sommeJusquIci contient la valeur tab[0] + … + tab[i-1]
        sommeJusquIci := sommeJusquIci + tab[i]
        /* Invariant de boucle :
        sommeJusquIci contient la valeur tab[0]+ … + tab[i] */
    Renvoyer sommeJusquIci
```

1e itération:
i vaut 0
sommeJusquIci vaut tab[0]
→ OK

D'une itération à l'autre:
sommeJusquIci passe de
tab[0] + … + tab[i-1]
à sommeJusquIci + tab[i]
donc à tab[0] + …. + tab[i]
→ OK

2) Analyse de la correction d'un algo.

```
Fun nombre sommeTab(nombre tab[ ]):
    int i
    nombre sommeJusquIci:=0
    int n:=longu(tab)
    Pour i allant de 0 à n:
        ici: sommeJusquIci contient la valeur tab[0] + … + tab[i-1]
        sommeJusquIci := sommeJusquIci + tab[i]
        /* Invariant de boucle :
        sommeJusquIci contient la valeur tab[0]+ … + tab[i] */
    Renvoyer sommeJusquIci
```

1e itération:
i vaut 0
sommeJusquIci vaut tab[0]
→ OK

D'une itération à l'autre:
sommeJusquIci passe de
tab[0] + … + tab[i-1]
à sommeJusquIci + tab[i]
donc à tab[0] + …. + tab[i]
→ OK

Fin de dernière itération:
sommeJusquIci vaut
tab[0] + … + tab[n]
→ pb car tab[n] n'existe pas!

2) Analyse de la correction d'un algo.

```
Fun nombre sommeTab(nombre tab[ ]):
    int i
    nombre sommeJusquIci:=0
    int n:=longu(tab)
    Pour i allant de 0 à n-1:
        ici: sommeJusquIci contient la valeur tab[0] + … + tab[i-1]
        sommeJusquIci := sommeJusquIci + tab[i]
        /* Invariant de boucle :
        sommeJusquIci contient la valeur tab[0]+ … + tab[i] */
    Renvoyer sommeJusquIci
```

1e itération:
i vaut 0
sommeJusquIci vaut tab[0]
→ OK

D'une itération à l'autre:
sommeJusquIci passe de
tab[0] + … + tab[i-1]
à sommeJusquIci + tab[i]
donc à tab[0] + …. + tab[i]
→ OK

Fin de dernière itération:
sommeJusquIci vaut
tab[0] + … + tab[n-1]
→ OK
→ l'algo est correct

Invariant de boucle : autre utilité

Un **invariant de boucle** peut parfois être **décidé AVANT d'écrire notre boucle:**

Dans ce cas, le choix de l'invariant de boucle exprime la manière dont on veut procéder et c'est lui qui **guide** pour l'écriture de notre algorithme.

Souvent, **la stratégie est "entre les deux":**

1) J'écris une 1e version de l'algorithme (pas forcément juste)
2) Je cherche l'invariant de boucle
3) Je modifie mon algorithme pour qu'il colle à l'invariant

Exemple

Ecrire une fonction qui prend en argument un tableau de nombres et qui renvoie le plus grand nombre contenu dans le tableau.

Idée: parcourir le tableau, et garder dans une variable locale le plus grand élément du tableau parmi les cases déjà parcourues
→ cela me donne le brouillon suivant:

```
Fun nombre maxTab(nombre tab[ ]):
    nombre maxJusquIci := …. (à compléter plus tard)
    Pour i de 0 à longu(tab)-1:
        … (Mettre à jour maxJusquIci si necessaire)
        /* Invariant: maxJusquIci contient le plus grand
           nombre parmi tab[0], …, tab[i] */
```

Exemple

Ecrire une fonction qui prend en argument un tableau de nombres et qui renvoie le plus grand nombre contenu dans le tableau.

```
Fun nombre maxTab(nombre tab[ ]):
    nombre maxJusquIci := …. (à compléter plus tard)
    int i
    Pour i de 0 à longu(tab)-1:
        Si tab[i]>maxJusquIci:
            maxJusquIci := tab[i]
        /* Invariant: maxJusquIci contient le plus grand
           nombre parmi tab[0], …, tab[i] */
    Renvoyer maxJusquIci
```

Exemple

Ecrire une fonction qui prend en argument un tableau de nombres et qui renvoie le plus grand nombre contenu dans le tableau.

```
Fun nombre maxTab(nombre tab[ ]):
    nombre maxJusquIci := ....        <-- ??
    int i
    Pour i de 0 à longu(tab)-1:
        Si tab[i]>maxJusquIci:
            maxJusquIci := tab[i]
        /* Invariant: maxJusquIci contient le plus grand
           nombre parmi tab[0], ..., tab[i] */
    Renvoyer maxJusquIci
```

Exemple

Ecrire une fonction qui prend en argument un tableau de nombres et qui renvoie le plus grand nombre contenu dans le tableau.

```
Fun nombre maxTab(nombre tab[ ]):
    nombre maxJusquIci := ....        <-- 0 ?
    int i
    Pour i de 0 à longu(tab)-1:
        Si tab[i]>maxJusquIci:
            maxJusquIci := tab[i]
        /* Invariant: maxJusquIci contient le plus grand
           nombre parmi tab[0], ..., tab[i] */
    Renvoyer maxJusquIci
```

1e itération: i vaut 0
et maxJusquIci vaut encore 0 si tab[0] n'est pas strictement plus grand que 0
→ **pas bon!!**

Exemple

Ecrire une fonction qui prend en argument un tableau de nombres et qui renvoie le plus grand nombre contenu dans le tableau.

```
Fun nombre maxTab(nombre tab[ ]):
    nombre maxJusquIci := tab[0]
    int i
    Pour i de 0 à longu(tab)-1:
        Si tab[i]>maxJusquIci:
            maxJusquIci := tab[i]
        /* Invariant: maxJusquIci contient le plus grand
           nombre parmi tab[0], …, tab[i] */
    Renvoyer maxJusquIci
```

maxJusquIci contient le plus grand nombre parmi tab[0], …, tab[i-1]

1e itération → OK
D'une itération à l'autre → OK
Dernière itération → OK
Valeur de retour de ma fonction → OK

Exemple

Ecrire une fonction qui prend en argument un tableau de nombres et qui renvoie le plus grand nombre contenu dans le tableau.

```
Fun nombre maxTab(nombre tab[ ]):
    nombre maxJusquIci := tab[0]
    int i
    Pour i de 0 à longu(tab)-1:
        Si tab[i]>maxJusquIci:
            maxJusquIci := tab[i]
        /* Invariant: maxJusquIci contient le plus grand
           nombre parmi tab[0], …, tab[i] */
    Renvoyer maxJusquIci
```

Invariant de boucle
→ vous aide à faire des algos justes!

3) Analyse de la terminaison d'un algo.

Il existe des programmes qui ne terminent jamais :

```
int i:=0
Tant que i<10:
    print(i)
```

→ Boucle infinie car i n'est jamais modifié

Il est donc important de **vérifier que votre algorithme termine.**

3) Analyse de la terminaison d'un algo.

Parmi les instructions que l'on a vue jusqu'à présent : seule une instruction **Tant que** peut mener à un algorithme qui ne termine pas.

On limitera donc notre **analyse de terminaisons** aux boucles **Tant que.**

Attention: lorsque l'on verra la notion de récursivité, on aura un risque important d'algorithmes qui ne terminent pas!

3) Analyse de la terminaison d'un algo.

Pour prouver qu'une boucle Tant que termine : il faut prouver qu'à chaque étape, **on se rapproche "d'au moins un cran" d'un cas d'arrêt.**

```
int i:=0
Tant que i<10:
    print(i)
    i++
```

Cas d'arrêt: i supérieur ou égal à 10

i commence à 0 puis est incrémenté de 1 à chaque étape:
0 → 1 → 2 → …… → 10 → **OK**

3) Analyse de la terminaison d'un algo.

Pour prouver qu'une boucle Tant que termine : il faut prouver qu'à chaque étape, **on se rapproche "d'au moins un cran" d'un cas d'arrêt.**

```
int i:=0
Tant que i<10:
    print(i)
    i - -
```

Cas d'arrêt: i supérieur ou égal à 10

i commence à 0 puis est décrémenté de 1 à chaque étape:
0 → - 1 → - 2 → …… **→ ne sera jamais supérieur à 10 !**

3) Analyse de la terminaison d'un algo.

Pour prouver qu'une boucle Tant que termine : il faut prouver qu'à chaque étape, on se **rapproche "d'au moins un cran" d'un cas d'arrêt.**

```
int i:=0
Tant que i !=15:
    print(i)
    i := i+2
```

Cas d'arrêt: i égal à 15

i commence à 0 puis est incrémenté de 2 à chaque étape:
0 → 2 → 4 → …… → 14 → 16 → 18 → …
→ i ne sera jamais égal à 15 !

3) Analyse de la terminaison d'un algo.

Ecrivez un algorithme qui prend en entrée un tableau de nombres, et un nombre cible, et qui renvoie Vrai si la cible est dans le tableau, Faux sinon.

Tentative:

```
Fun bool trouve(nombre tab[ ], nombre cible)
    int i:=0
    bool aTrouve:=Faux
    Tant que non aTrouve:
        Si tab[i]==cible:
            aTrouve :=Vrai
        i++
    Renvoyer aTrouve
```

3) Analyse de la terminaison d'un algo.

Ecrivez un algorithme qui prend en entrée un tableau de nombres, et un nombre cible, et qui renvoie Vrai si la cible est dans le tableau, Faux sinon.

```
Fun bool trouve(nombre tab[ ], nombre cible)
    int i:=0
    bool aTrouve:=Faux
    Tant que non aTrouve :
        Si tab[i]==cible:
            aTrouve :=Vrai
        i++
    Renvoyer aTrouve
```

Cas d'arrêt: aTrouve est Vrai

i commence à 0 puis est incrémenté de 1 à chaque étape: On ne se rapproche pas du cas d'arrêt!

3) Analyse de la terminaison d'un algo.

Ecrivez un algorithme qui prend en entrée un tableau de nombres, et un nombre cible, et qui renvoie Vrai si la cible est dans le tableau, Faux sinon.

```
Fun bool trouve(nombre tab[ ], nombre cible)
    int i:=0
    bool aTrouve:=Faux
    Tant que non aTrouve et i<longu(tab) :
        Si tab[i]==cible:
            aTrouve :=Vrai
        i++
    Renvoyer aTrouve
```

Cas d'arrêt: aTrouve est Vrai
ou i ≥ longu(tab)

i commence à 0 puis est incrémenté de 1 à chaque étape:
On se rapproche du cas d'arrêt n°2

Fin chapitre 3: analyse d'algo

Récapitulatif des notions qui ont été abordées:

- *Utilité des analyses d'algorithmes*
  - *Nous aide à construire des algo justes*
  - *Sert à convaincre autrui que notre algo est juste*
- *Preuve de correction avec invariant de boucle*
  - *Bien définir l'endroit où l'on place l'invariant*
  - *Attention à la 1^{e} et dernière itération*
- *Preuve de terminaisons (surtout pour les boucles Tant Que)*