Université de Liège

INFO0946

Introduction à la Programmation

GameCode : Un exercice dont vous êtes le Héros · l'Héroïne

Approche Constructive – Tri par Insertion

Benoit Donnet Simon Liénardy

23 février 2021

Préambule

Exercices

Dans ce GameCode, nous vous proposons de suivre pas à pas la résolution d'un exercice portant sur l'entièreté de la méthodologie de développement.

Il est dangereux d'y aller seul [1] !

Partagez vos commentaires, questions, solutions alternatives sur le forum eCampus. N'hésitez jamais !

1. Référence vidéoludique bien connue des Héros.

2.1 Enoncé

Il s'agit de construire un programme implémentant le *tri par insertion* pour un tableau d'entiers. Ce tri fonctionne comme suit : à chaque étape le tableau est divisé en deux parties ; la seconde partie est inchangée depuis le début tandis que la première est triée. Il s'agit alors « tout simplement » d'insérer le premier élément de la seconde partie « à sa place » dans la première partie. C'est par exemple, comme cela qu'un joueur de manille peut trier ses cartes.

Par exemple, considérons le tableau (de taille 5) suivant :

0				4
3	2	1	1	0

On obtiendra successivement les résultats suivants :

- 1re itération

0				4
3	2	1	1	0

- 2e itération

0				4
2	3	1	1	0

- 3e itération

0				4
1	2	3	1	0

- 4e itération

0				4
1	1	2	3	0

- 5e itération

0				4
0	1	1	2	3

2.1.1 Méthode de Résolution

On va procéder à la résolution de ce problème en suivant pas à pas l'approche constructive :

1. Formalisation du Problème en proposant des notations (voir Sec. 2.2) ;
2. Analyse du problème et découpe en Sp (Sec. 2.3) ;
3. Spécifications de chaque Sp (Sec. 2.4) ;
4. Invariant de Boucle pour chaque Sp (Sec. 2.5) ;
5. Construction de chaque Sp (Sec. 2.6) ;
6. Mise en commun du code (Sec. 2.7) ;

2.2 Formalisation du Problème

La première étape dans la résolution du problème nécessite sa formalisation, ce qui passe par l'introduction de nouvelles notations.

Si vous voyez de quoi on parle, rendez-vous à la Section 2.2.3
Si vous ne savez comment introduire une notation, voyez la Section 2.2.1
Si vous ne voyez pas comment faire, reportez-vous à l'indice 2.2.2

2.2.1 Rappels sur la Formalisation d'un Problème

2.2.1.1 Généralités

Voici la forme que doit prendre une notation :

NomPredicat(Liste de parametres) ≡ Détail du predicat

NomPredicat est le nom du prédicat [a]. Il faut souvent trouver un nom en rapport avec ce que l'on fait. Mathématiquement parlant, cela ne changera rien de proposer un nom farfelu. Par exemple, on pourrait appeler le prédicat « Gims ». Cependant, pour des raisons de lisibilité, il est toujours préférable detrouver un nom en rapport avec la définition et le problème qu'on cherche à formaliser.

≡ est le symbole mathématique signifiant « identique à ». Il sert à introduire le détail du prédicat.

Liste de paramètres C'est une liste de symboles qui pourront être utilisés dans la définition du prédicat. Comment cela fonctionne-t-il ? Le prédicat peut être utilisé en écrivant son nom ainsi qu'une valeur particulière de paramètre. On remplace cette valeur dans la définition du prédicat et on regarde si c'est vrai ou faux.

Détail du prédicat C'est une combinaison de symboles logiques qui fait intervenir, le plus souvent, les paramètres. Si nécessaire, il est aussi possible d'utiliser des quantificateurs.

a. Pour rappel, un prédicat est une fonction mathématique produisant une valeur booléenne.

Exemple :

$$\text{Pair}(X) \equiv X \ \% \ 2 = 0$$

Le nom du prédicat est « Pair » et prend un argument. $Pair(4)$ remplace $4 \ \% \ 2 = 0$ qui est vrai et $Pair(5)$ signifie $5 \ \% \ 2 = 0$ qui est faux. Le choix du nom du prédicat est en référence à sa signification : il est vrai si X est pair.

▷ **Exercice** Si vous avez compris, vous pouvez définir facilement le prédicat Impair(X).

2.2.1.2 Opérateurs Logiques

Les opérateurs logiques usuels sont les suivants :

$True$ correspond la propriété "vrai" ;

$False$ correspond à la propriété "faux" ;

$\neg$ est l'opérateur de négation. Dès lors, $\neg p$ signifie "non p" ;

$\wedge$ est l'opérateur de conjonction. Dès lors, $p \wedge q$ signifie "p et q" ;

$\vee$ est l'opérateur de disjonction. Dès lors, $p \vee q$ signifie "p ou q" ;

$\Longrightarrow$ est l'opérateur d'implication. Dès lors, $p \implies q$ signifie que si p est vraie, alors q aussi. A noter aussi que si p est fausse, alors $p \implies q$ est vraie, quelque soit q (i.e., du faux on peut conclure ce qu'on veut).

Le Tableau 1 reprend la table de vérité pour les différents opérateurs logiques.
A noter que $p \iff q$ est équivalent à $(p \implies q) \wedge (q \implies p)$.
Enfin, la priorité des opérateurs logiques est la suivante ($<$ signifie "plus prioritaire") :

$$\neg \ < \ \wedge, \vee \ < \ \Longrightarrow, \iff$$

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \implies q$	$p \iff q$
Vrai	Vrai	Faux	Vrai	Vrai	Vrai	Vrai
Vrai	Faux	Faux	Faux	Vrai	Faux	Faux
Faux	Vrai	Vrai	Faux	Vrai	Vrai	Faux
Faux	Faux	Vrai	Faux	Faux	Vrai	Vrai

TABLE 1 – Table de vérité pour les opérateurs logiques usuels.

2.2.1.3 Quantificateurs Logiques

Quantificateur Universel

Soient P et Q, deux prédicats, et (l) une liste (non vide) d'identificateurs.

$$\forall\ (l) \cdot (P \implies Q)$$

est un prédicat représentant la *quantification universelle*. La signification (en français) du prédicat est "pour tout i tel que P alors Q (est vrai)".

Dans le cadre du cours, nous allons utiliser la notation (relaxée) suivante :

$$\forall\ (l), P, Q$$

La quantification universelle peut être vue comme un *sucre syntaxique* simplifiant une suite de conjonctions. Par exemple, le prédicat :

$$\forall\ i, 0 \leq i \leq N-1, T[i] > 0$$

peut se comprendre comme suit :

$$T[0] > 0 \wedge T[1] > 0 \wedge\ \ldots\ \wedge T[N-1] > 0$$

Enfin, lorsque la variable prend ses valeurs sur l'ensemble vide, la quantification est vraie, quel que soit le terme P. Par exemple :

$$\forall\ i, 0 \leq i \leq -1, T[i] = 0 == True$$

Quantificateur Existentiel

Soient P et Q, deux prédicats, et (l) une liste (non vide) d'identificateurs.

$$\exists\ (l) \cdot (P \wedge Q)$$

est un prédicat représentant la *quantification existentielle*. La signification (en français) du prédicat est "il existe (au moins) un i tel que P pour lequel Q (est vrai)".

Dans le cadre du cours, nous allons utiliser la notation (relaxée) suivante :

$$\exists\ (l), P, Q$$

La quantification existentielle peut être vue comme un *sucre syntaxique* simplifiant une suite de disjonctions. Par exemple, le prédicat :

$$\exists\ i, 0 \leq i \leq N-1, T[i] > 0$$

peut se comprendre comme suit :

$$T[0] > 0 \vee T[1] > 0 \vee\ \ldots\ \vee T[N-1] > 0$$

2.2.1.4 Quantificateurs Numériques

Somme

La somme d'une suite de termes est représentée à l'aide de la notation Σ.
Ainsi, par exemple :

$$0 + 1 + 2 + \ \ldots \ + N - 1$$

sera représenté de manière beaucoup plus condensée par

$$\sum_{i=0}^{N-1} i$$

Par définition, la somme sur un intervalle vide vaudra toujours le neutre de l'addition, soit 0 (i.e., somme à zéro terme). Soit :

$$\sum_{i=0}^{-1} i = 0$$

Produit

Le produit d'une suite de termes est représentée à l'aide de la notation Π.
Ainsi, par exemple :

$$T[0] \times T[1] \times T[2] \times \ \ldots \ \times T[N-1]$$

sera représenté de manière beaucoup plus condensée par

$$\prod_{i=0}^{N-1} T[i]$$

Par définition, le produit sur un intervalle vide vaudra toujours le neutre de la multiplication, soit 1 (i.e., produit à zéro terme)

$$\prod_{i=0}^{-1} T[i] = 1$$

Minimum/Maximum

Le minimum (respectivement maximum) d'une suite de termes est représenté à l'aide la notation min (respectivement max).

$$min_P Q$$

où P et Q sont des prédicats

Par exemple :

$$minimum\{T[0], T[1], \ldots, T[N-1]\}$$

sera adéquatement représenté par :

$$min_{0 \leq i \leq N-1} \ (T[i])$$

Dénombrement

Soient P et Q, deux prédicats, et i une variable libre [a]

$$\#i \cdot (P|Q)$$

est un prédicat représentant le *quantificateur de dénombrement*. La signification (en français) du prédicat est "le nombre de fois où le prédicat Q est vrai lorsque P (est vrai)".

a. Pour rappel, une variable est dite *libre* quand le nom que l'on utilise pour définir un objet a de l'importance (il s'agit typiquement d'une variable non introduite par un quantificateur). A l'inverse, une variable est dite *liée* si le nom que l'on utilise pour définir un objet n'a pas d'importance (il s'agit typiquement d'une variable introduite par une quantificateur).

Par exemple, un prédicat indiquant le nombre de valeurs à 0 dans un tableau `T` à `N` valeurs entières se présente comme suit :

$$\#i \cdot (0 \leq i \leq N-1|T[i] = 0)$$

Suite de l'Exercice

À vous ! Formalisez le problème et passez à la Sec. 2.2.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 2.2.2

2.2.2 Indice

Pour formaliser un problème, il faut répondre aux deux questions suivantes :

- De quoi parle le Problème ?
- Quel est le Sujet ?

La réponse devrait mettre en évidence les éléments essentiels à la résolution du problème.
Ensuite, une bonne idée est d'illustrer le fonctionnement de la notation sur un dessin qui sera ensuite traduit en prédicat.

Suite de l'Exercice

À vous ! Définissez le problème et passez à la Sec. 2.2.3.

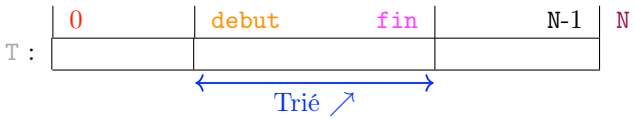
2.2.3 Mise en Commun de la Formalisation du Problème

Repartons de l'énoncé et mettons en évidence la/les notion(s) importantes permettant de répondre aux questions "de quoi parle le Problème ?" et "Quel est le sujet ?" :

> Il s'agit de construire un programme implémentant tri par insertion pour un tableau d'entiers. Ce tri fonctionne comme suit : à chaque étape le tableau est divisé en deux parties ; la seconde partie est inchangée depuis le début tandis que la première est triée. Il s'agit alors « tout simplement » d'insérer le premier élément de la seconde partie « à sa place » dans la première partie. C'est par exemple, comme cela qu'un joueur de manille peu trier ses cartes.

On voit ici que la notion principale du problème est le *tri*. Le reste du texte est une explication de comment va fonctionner le code. Il faudrait donc une notation pour le tri.

Nous introduisons un prédicat qui représente le tri (non strict) d'une portion de tableau. La portion est triée si tous les entiers se suivent selon la relation d'ordre $\leq$ (i.e., tri par ordre croissant). Nous pouvons illustrer ce prédicat de la façon suivante :



soit le sous-tableau `T[debut ...fin]` est trié par ordre croissant, avec `debut` et `fin` dans les bornes du tableau.

Le prédicat va se présenter en quatre parties :

1. il faut donner un nom au prédicat, si possible qui illustre son objectif. Puisque nous voulons définir qu'une portion du tableau est triée, le prédicat s'appelera Tri.
2. il faut définir les paramètres du prédicat. Le dessin les indique clairement : nous avons besoin du tableau (T), de la taille du tableau (N), de la borne inférieure de l'intervalle (debut) et la borne supérieure (fin).
3. il faut vérifier que l'intervalle `[debut ...fin]` se trouve bien dans les bornes du tableau. Cela se traduit par :

   $$0 \leq \mathrm{debut} \leq \mathrm{fin} < N$$

4. il faut indiquer que la portion `[debut ...fin]` est triée par ordre croissant (non strict). Cela peut se traduire en utilisant une quantification universelle sur la portion du tableau. Soit :

   $$\forall i, \mathrm{debut} \leq i < \mathrm{fin}, T[i] \leq T[i+1]$$

On peut maintenant tout mettre ensemble et obtenir le prédicat suivant :

$$\begin{aligned} Tri(\mathrm{T}, N, \mathrm{debut}, \mathrm{fin}) \quad \equiv \quad & 0 \leq \mathrm{debut} \leq \mathrm{fin} < N \\ & \wedge \forall i, \mathrm{debut} \leq i < \mathrm{fin}, T[i] \leq T[i+1] \end{aligned}$$

On voit, dans le prédicat, que les points 3. et 4. sont reliés par une conjonction ($\wedge$). C'est normal car pour que la définition soit correcte, il faut que l'intervalle d'intérêt soit dans les bornes du tableau (3.) et que cet intervalle soit trié (4.).

Remarque L'élément $T[\mathrm{fin}]$ est-il compris dans le tri ? Oui parce que la valeur maximale i_{max} est $\mathrm{fin} - 1$ et si on le remplace dans $T[i] \leq T[i+1]$, cela donne $T[\mathrm{fin} - 1] \leq T[\mathrm{fin}]$.

Suite de l'Exercice

Nous pouvons maintenant passer à l'analyse du problème et la découpe en Sp. Voir Sec. 2.3.

2.3 Découpe en Sp

La deuxième étape de la résolution du problème est l'*analyse du problème*, nous amenant à une découpe en Sp.

Si vous voyez de quoi on parle, rendez-vous à la Section 2.3.3
Si vous ne savez pas ce qu'est l'analyse du problème, allez à la Section 2.3.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 2.3.2

2.3.1 Rappels sur l'Analyse

L'étape d'*analyse* permet de réfléchir à la structuration du code en appliquant une *approche systémique*, i.e., la découpe du problème principal en différents sous-problèmes (SP) et la façon dont les SP interagissent les uns avec les autres. C'est cette interaction entre les SP qui permet la structure du code.

Un SP correspond à une tâche particulière que le programme devra effectuer dans l'optique d'une résolution complète du problème principal. Si la tâche correspond à un module, il faudra le spécifier. Il est impératif que chaque SP dispose d'un nom (pertinent) ou d'une courte description de la tâche qu'il effectue.

Alerte : Microscopisme

Inutile de tomber, ici, dans une découpe en SP trop fine (ou *microscopique*). Le bon niveau de granularité, pour un SP, est la boucle ou un *module*.
Une erreur classique est de considérer la déclaration des variables comme un SP à part entière.

L'agencement des différents SP doit permettre de résoudre le problème principal, i.e., à la fin du dernier SP, la PostCondition du problème doit être atteint. De même, le premier SP doit s'appuyer sur les informations fournies par la PréCondition. On envisage deux formes d'agencement des SP :

1. *Linéraire* : $\text{SP}_i \rightarrow \text{SP}_j$. Dans ce cas, le SP_j est exécuté après le SP_i. Typiquement, la PostCondition du SP_i sert de PréCondition au SP_j.
2. *Inclusion* : $\text{SP}_j \subset \text{SP}_i$. L'exécution du SP_j est incluse dans celle du SP_i. Cela signifie que le SP_j est invoqué plusieurs fois dans le SP_i. Typiquement, le SP_j est un traitement exécuté lors de chaque itération du SP_i. Par exemple, le SP_i est une boucle et, son corps, comprend une exécution du SP_j (qui lui aussi peut être une boucle).

Attention, les deux agencements ne sont pas exclusifs. On peut voir apparaître les différents agencements au sein d'un même problème.

Suite de l'Exercice

À vous ! Analysez le problème et passez à la Sec. 2.3.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 2.3.2

2.3.2 Indice

Pour découper le problème principal, il convient de repartir de l'énoncé

Vous devez, bien entendu, veiller à ce que votre découpe soit pertinente : ni trop haut niveau (e.g., SP_1 : résoudre le problème), ni trop bas niveau (e.g., SP_1 déclarer des variables). Dans tous les cas, après exécution du dernier SP, le problème général doit être résolu (i.e., le tableau est trié par ordre croissant).

Suite de l'Exercice

À vous ! Définissez le problème et passez à la Sec. 2.2.3.

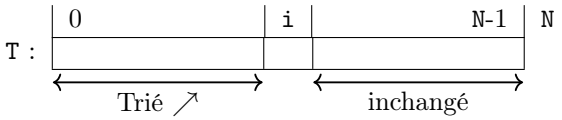
2.3.3 Mise en Commun de l'Analyse du Problème

Raisonnement Général

De l'énoncé, on note :

> « Il s'agit de construire un programme implémentant tri par insertion pour un tableau d'entiers. Ce tri fonctionne comme suit : à chaque étape le tableau est divisé en deux parties ; la seconde partie est inchangée depuis le début tandis que la première est triée. Il s'agit alors « tout simplement » d'insérer le premier élément de la seconde partie « à sa place » dans la première partie. »

Sur base de cela, on peut illustrer l'algorithme de la façon suivante :



Il est donc possible de résoudre le problème en parcourant le tableau de la gauche vers la droite à l'aide de la variable `i`. À chaque itération, il faut insérer l'élément `T[i]` dans `T[0 ... i-1]`. Ceci peut se faire en trois étapes :

1. Trouver la position où insérer `T[i]`. Notons cette position `k`.
2. Décaler `T[k ... i-1]` de 1 position vers la droite.
3. Placer l'élément $\texttt{T}_0[\texttt{i}]$ dans `T[k]`.

Ces étapes forment un premier Sp qui consiste à insérer `T[i]` dans `T[0 ... i-1]`.

Première Découpe

On pourrait partir sur la découpe suivante :

Sp$_1$ Insérer une valeur dans un sous-tableau déjà trié.

Sp$_0$ Le problème principal (i.e., tri du tableau).

Raffinement du Sp$_1$

Pour le Sp$_1$, on peut encore être plus précis : en effet, le décalage est une opération générique et peut être effectué dans un second Sp. On obtient alros la découpe suivante :

Sp$_1$ Insérer une valeur dans un sous-tableau déjà trié.

Sp$_2$ Décaler un sous-tableau d'une case vers la droite.

Sp$_0$ Le problème principal (i.e., tri du tableau).

L'emboitement est donc : Sp$_2$ $\subset$ Sp$_1$ $\subset$ Sp$_0$.

Suite de l'Exercice

Nous pouvons maintenant passer à l'interface de chaque Sp (i.e., prototype + spécifications). Voir Sec. 2.4.

2.4 Spécifications

Il s'agit, maintenant, de proposer une interface pour chaque Sp.

Si vous voyez de quoi on parle, rendez-vous à la Section 2.4.3
Si vous ne savez pas ce qu'est une spécification, allez à la Section 2.4.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 2.4.2

2.4.1 Rappels sur les Spécifications

2.4.1.1 Généralités

Une spécification se définit en deux temps :

La PréCondition implémente les suppositions. Elle caractérise donc les conditions initiales du module, i.e., les propriétés que doivent respecter les valeurs en entrée (i.e., paramètres effectifs) du module. Elle se définit donc sur les paramètres formels (qui seront initialisés avec les paramètres effectifs). La PréCondition doit être satisfaite avant l'exécution du module.

La PostCondition implémente les certifications. Elle caractérise les conditions finales du résultat du module. Dit autrement, la PostCondition décrit le résultat du module sans dire comment il a été obtenu. La PostCondition sera satisfaite après l'invocation.

A l'instar de la définition d'un problème, un module doit toujours avoir une PostCondition (un module qui ne fait rien ne sert à rien) mais il est possible que le module n'ait pas de PréCondition (e.g., le module ne prend pas de paramètres formels).

La spécification d'un module se met, en commentaires, avec le prototype du module. L'ensemble (i.e., spécification + prototype) forme l'*interface* du module.

Contrairement au cours INFO0946, les spécifications d'un module seront, dorénavant, exprimées de manière formelle (i.e., à l'aide de prédicats et de notations).

2.4.1.2 Construction d'une Interface

Pour construire l'interface d'un module, il est souhaitable de commencer par faire un dessin. Le dessin, qui naturellement sera lié à Invariant Graphique, doit représenter des situations particulières du problème à résoudre. La situation initiale (i.e., avant la première instruction de la Zone 1/Init) doit aider pour trouver le prototype du module, ainsi que la PréCondition. La situation finale (i.e., après exécution de la Zone 3/End) doit vous aider à trouver la PostCondition.

En s'appuyant sur les dessins, il suffit ensuite de répondre à trois questions pour construire l'interface d'un module :

Question 1 : Quels sont les objets dont j'ai besoin pour atteindre mon objectif ? Répondre à cette question permet de construire le prototype du module (éventuel type de retour, identificateur du module, paramètres formels).

Question 2 : Quel est l'objectif du module ? Répondre à cette question permet d'obtenir la PostCondition, représentée à l'aide d'un prédicat.

Question 3 : Quelles sont les contraintes sur les paramètres ? Répondre à cette question permet d'obtenir la PréCondition, représentée à l'aide d'un prédicat.

L'avantage de faire un dessin est, bien entendu, de dresser un pont avec l'Invariant Graphique, mais aussi de faciliter la production d'un prédicat (il "suffit" de traduire le dessin en un prédicat).

2.4.1.3 Exemple

A titre d'exemple, essayons de spécifier une fonction qui permet de calculer le produit de tous les entiers entre des bornes fournies, i.e., `a` et `b` (avec $b > a$).

Le problème peut s'illustrer comme indiqué à la Fig. 1. On dessine une droite (représentant les nombres qu'on va manipuler) avec les bornes de l'intervalle d'intérêt. La flèche bleue indique ce qu'on doit calculer.

Question 1
Pour atteindre l'objectif, nous avons simplement besoin des bornes de l'intervalle. Sur la Fig. 1, on peut clairement voir que le calcul du produit s'étale entre `a` et `b`. Ce sont les seuls éléments "extérieurs" dont on a besoin pour résoudre le problème. `a` et `b` seront donc les paramètres formels du module.

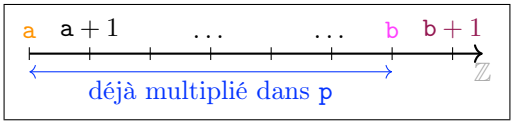


FIGURE 1 – Exemple de représentation graphique du problème à résoudre.

La Fig. 1 illustre aussi le fait qu'on manipule des entiers ($\mathbb{Z}$). a et b seront donc de type `int`.

Enfin, on peut adéquatement nommer le module `produit()`, ce qui permet de naturellement décrire ce qu'il fait.

Tout ceci nous permet de dériver le prototype du module :

```
1 produit(int a, int b);
```

Question 2

L'objectif du module est de calculer (et donc retourner) le produit de tous les entiers dans l'intervalle $[a, b]$. Soit :

$$a \times (a+1) \times \cdots \times (b-1) \times b$$

On peut représenter ce produit de manière plus condensée en utilisant le quantificateur numérique du produit : $\prod$. Soit :

$$\prod_{i=a}^{b} i$$

Puisque a et b sont de type `int`, le produit résultat sera aussi de type `int`. Le module `produit()` est donc une fonction retournant un `int`. On représente cela de la façon suivante :

$$produit = \prod_{i=a}^{b} i$$

Enfin, `a` et `b` ne sont pas modifiés par le module.

A ce stade, on obtient donc l'interface incomplète suivante :

```
1 /*
2  * PostCondition : produit = ∏_{i=a}^{b} i ∧ a = a_0 ∧ b = b_0
3  */
4 int produit(int a, int b);
```

Question 3

Enfin, nous pouvons déterminer si il existe des conditions sur les paramètres. C'est bien le cas ici. L'énoncé du problème et la Fig. 1 indiquent clairement que b est strictement plus grand que a. On peut le représenter de la façon suivante :

$$b > a$$

Interface
Au final, l'interface du module est la suivante :

```
/*
 * PréCondition : b > a
 * PostCondition : produit = ∏_{i=a}^{b} i ∧ a = a_0 ∧ b = b_0
 */
int produit(int a, int b);
```

Suite de l'Exercice

À vous ! Proposez les interfaces pour les SP et passez à la Sec. 2.4.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 2.4.2

2.4.2 Indice

Pour chaque Sp, répondez aux trois questions en vous appuyant sur un schéma. Le schéma permettra d'aisément traduire la situation particulière (PréCondition ou PostCondition) sous forme formelle.

Ne soyez pas ambigu dans la PréCondition, ni dans la PostCondition. Au contraire, soyez le plus précis possible (rappelez-vous, un prédicat est une formulation mathématique et, dès lors, ne peut souffrir d'une quelconque ambiguïté). On n'oubliera pas de s'appuyer sur les notations introduites.

Une fois que c'est fait, rédigez l'interface de chaque module.

Suite de l'Exercice

À vous ! Spécifiez les Sp et passez à la Sec. 2.4.3.

2.4.3 Mise en Commun des Spécifications

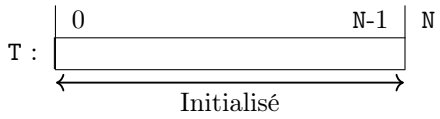
Il faut maintenant donner l'interface (i.e., spécifications + prototype) de chacun des SP.

- spécification du SP_0 ;
- spécification du SP_1 ;
- spécification du SP_2.

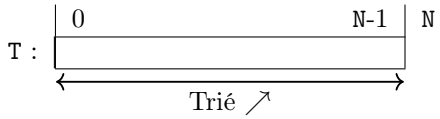
2.4.3.1 Spécification du SP_0

Pour rappel, le SP_0 fait référence au problème principal, i.e., tri du tableau.

En PréCondition, il faut s'assurer que le tableau est initialisé sur toute sa longueur, sinon c'est inutile de le trier. On peut représenter, graphiquement, cette situation particulière de la façon suivante :



En PostCondition, il suffit d'indiquer que le tableau doit être trié sur toute sa longueur. On peut représenter, graphiquement, cette situation particulière de la façon suivante :



On voit clairement sur le schéma qu'on peut utiliser la notation $Tri()$ mais, non plus sur une portion du tableau, mais bien sur tout le tableau ($debut$=0 et fin=N-1).

De plus, ce doit être une *permutation* du tableau de départ (pas d'ajout ni de retrait d'élément).

On obtient donc :

```c
/*
 * PréCondition : T[0.. N-1] init
 * PostCondition : Tri(T, N, 0, N-1) ∧ T perm T0
 */
void tri_insertion(int *T, const unsigned int N);
```

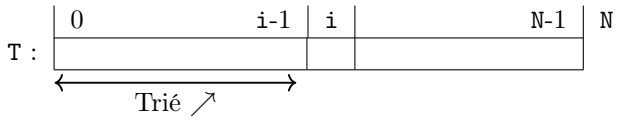
Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- spécification du SP_1 ;
- spécification du SP_2 ;
- Invariants de Boucle des modules.

2.4.3.2 Spécification du SP_1

Pour rappel, l'objectif du SP_1 est d'insérer une valeur dans un tableau trié. On peut représenter la situation par le schéma suivant :



En PréCondition, il faut indiquer que le tableau dans lequel on souhaite insérer l'élément est trié entre 0 et `i-1`. En PostCondition, on demande d'avoir insérer l'élément en `i` et de ne pas avoir ajouté d'éléments (le tableau final est une permutation du tableau initial). Ce qui donne :

```
/*
 * PréCondition : Tri(T, N, 0, i-1) ∧ T[0 .. i] init ∧ 0 ≤ i < N
 * PostCondition : Tri(T, N, 0, i) ∧ T perm T_0
 */
void insertion(int *T, unsigned int i, unsigned int N);
```

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

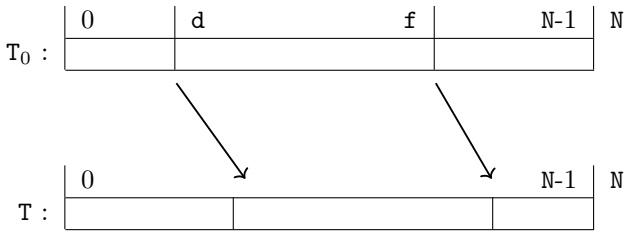
- spécification du SP_0 ;
- spécification du SP_2 ;
- Invariants de Boucle des modules.

2.4.3.3 Spécification du SP_2

Pour rappel, le SP_2 permet de décaler une portion de tableau d'une case vers la droite.
Ici, on va modifier le tableau `T`, il faudra bien distinguer :

- T_0 qui est le tableau initial (avant l'invocation du sous-problème)
- T qui est le tableau final

On peut schématiser de la façon suivante (`T`$_0$`[d ... f]` est la portion à décaler d'une position vers la droite) :



Il est évident que la portion `T`$_0$`[d ... f]` [2] doit se trouver dans les bornes du tableau. De même, la position `T[f+1]` doit exister de façon à pouvoir effectuer le décalage.

Ce qui nous donne :

```
/*
 * PréCondition : T[d ... f+1] init ∧ 0 ≤ d ≤ f + 1 < N
 * PostCondition : ∀i, d < i ≤ f + 1, T[i] = T0[i − 1]
void shift(int *T, unsigned int d, unsigned int f, unsigned int N);
```

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- spécification du SP_0 ;
- spécification du SP_1 ;
- Invariants de Boucle des modules.

2. `d` pour "début", `f` pour "fin".

2.5 Invariant de Boucle

L'écriture de code impliquant un traitement itératif (i.e., boucle) nécessite au préalable la proposition d'un Invariant de Boucle.

Si vous voyez de quoi on parle, rendez-vous à la Section 2.5.3
Si vous ne savez pas ce qu'est un Invariant de Boucle, allez la Section 2.5.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 2.5.2

2.5.1 Rappels sur l'Invariant de Boucle

2.5.1.1 Généralités

Un Invariant de Boucle est une **propriété** vérifiée à chaque évaluation du Gardien de Boucle.
L'Invariant de Boucle présente un **résumé** de tout ce qui a déjà été calculé, jusqu'à maintenant (i.e., jusqu'à l'évaluation courante du Gardien de Boucle), par la boucle.

Le fait que l'Invariant de Boucle soit une propriété est important : ce n'est pas du code, ce n'est pas exclusivement destiné au langage C. Tout programme qui inclut une boucle doit s'appuyer sur un Invariant de Boucle.

Alerte : Ce que l'Invariant de Boucle n'est pas

De manière générale, un Invariant de Boucle

- n'est pas une instruction exécutée par l'ordinateur ;
- n'est pas une directive comprise par le compilateur ;
- n'est pas une preuve de correction du programme ;
- est indépendant du Gardien de Boucle ;
- ne garantit pas que la boucle se termine [a] ;
- n'est pas une assurance de l'efficacité du programme.

a. Seule la Fonction de Terminaison vous permet de garantir la terminaison

2.5.1.2 Sémantique Formelle

La *sémantique* formelle de l'Invariant de Boucle est présentée ici :

```
1  // {PréCondition}
2  Init
3  // {Inv}
4  while(B){
5       // {Inv ∧ B}
6       Iter
7       // {Inv}
8  }
9  // {Inv ∧ ¬B}
10 End
11 // {PostCondition}
```

Init correspond à la Zone 1. Ce sont donc les instructions qui, partant de la PréCondition, doivent amener la boucle et, donc, l'Invariant de Boucle.

B correspond au Gardien de Boucle. B doit être vrai pour pouvoir entrer dans la boucle.

Iter correspond à la Zone 2. Il s'agit donc du Corps de la Boucle. Ce sont donc les instructions qui doivent être répétées pour faire avancer le problème. Dans Iter, le point de départ est la situation particulière $\{\text{Inv} \wedge B\}$ et l'objectif, après la dernière intstruction du Corps de la Boucle, est de restaurer l'Invariant de Boucle.

¬ B correspond au Critère d'Arrêt. Quand ¬ B est vérifié, alors on sort de la boucle.

End correspond à la Zone 3. Il s'agit des instructions qui, une fois la boucle terminée (i.e., $\{\text{Inv} \wedge \neg B\}$), amène la PostCondition.

2.5.1.3 Règles pour l'Élaboration d'un Invariant Graphique

Bien que le cours INFO0947 ait pour objectif de présenter la programmation sous l'angle de ses fondements mathématiques (et, donc, par conséquence un Invariant de Boucle est présenté sous la forme d'un prédicat), il est toujours souhaitable de passer par l'étape dessin (et donc, l'Invariant Graphique).

Nous rappelons, ici, les 7 règles pour l'élaboration d'un Invariant Graphique satisfaisant (en les illustrants avec la Fig. 2) :

Règle 1 : réaliser un dessin pertinent par rapport au problème (la droite des entiers dans la Fig. 2) et le nommer ($\mathbb{Z}$ dans la Fig. 2) ;

Règle 2 : placer sur le dessin les bornes de début et de fin (a, b, et b +1 dans la Fig. 2) ;

Règle 3 : placer une ligne de démarcation qui sépare ce qu'on a déjà calculé dans les itérations précédentes de ce qu'il reste encore à faire (la ligne rouge sur la Fig. 2). Si nécessaire, le dessin peut inclure plusieurs lignes de démarcation ;

Règle 4 : étiqueter proprement chaque ligne de démarcation avec, e.g., une variable (i sur la Fig. 2) ;

Règle 5 : décrire ce que les itérations précédentes ont déjà calculé. Ceci implique souvent d'introduire de nouvelles variables ("déjà multiplié dans p" dans la Fig. 2) ;

Règle 6 : identifier ce qu'il reste à faire dans les itérations suivantes ("à multiplier" dans la Fig. 2) ;

Règle 7 : Toutes les structures identifiées et variables sont présentes dans le code (a, b, i, et p sur la Fig. 2) [3].

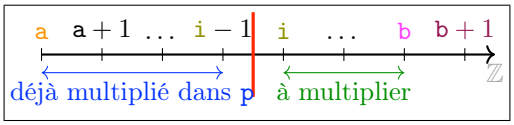


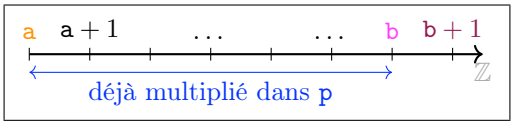
FIGURE 2 – Exemple d'Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

2.5.1.4 Comment Trouver un Invariant Graphique ?

C'est la grande question. On peut envisager plusieurs méthodes mais, dans le cadre d'un GAMECODE et du cours, il nous semble plus pertinent d'insister sur la méthode de *l'éclatement de la* POSTCONDITION. La technique est, finalement, assez simple :

1. repartir de la représentation graphique du problème et, en particulier, de la POSTCONDITION ;
2. appliquer les règles de conception d'un Invariant Graphique sur le dessin représentant la POSTCONDITION ;

Par exemple, l'Invariant Graphique du problème de calcul du produit des entiers entre deux bornes peut facilement être inféré de la représentation graphique du problème. Reprenons là ici :



On voit bien qu'on dispose, ainsi, d'une situation particulière (en général, la situation à la sortie de la boucle). Naturellement, le dessin s'accomode déjà de la Règle 1 et Règle 2. Placer une ligne de démarcation (Règle 3) naturellement rétrécit la zone bleue à une situation générale (Règle 5). Il suffit d'étiqueter correctement cette ligne de démarcation avec une variable (Règle 4) et, ensuite, de rajouter l'information sur la zone encore à traiter dans les itérations suivantes (Règle 6). L'Invariant Graphique ainsi produit est alors complet et cohérent avec les règles de conception. Il ne reste plus qu'à le transformer en Invariant Formel.

3. Plus de détails sur le lien entre l'Invariant de Boucle et la construction du code dans le rappel associé

2.5.1.5 De l'Invariant Graphique à l'Invariant Formel

A des fins d'exemple, traduisons l'Invariant Graphique donné à la Fig. 2 en un Invariant Formel. Un Invariant Graphique bien formé (i.e., qui respecte les 7 règles) contient toutes les informations nécessaires pour un Invariant Formel. Passer de l'un à l'autre est donc une simple quesiton de traduction d'un langage à l'autre (i.e., du dessin aux mathématiques).

$$\textsc{Inv} \equiv \mathtt{a} \leq \mathtt{i} \leq \mathtt{b} + 1 \tag{1}$$

$$\wedge\, \mathtt{p} = \prod_{j=a}^{\mathtt{i}-1} j \tag{2}$$

$$\wedge\, \mathtt{a} = \mathtt{a}_0 \tag{3}$$

$$\wedge\, \mathtt{b} = \mathtt{b}_0 \tag{4}$$

Ce qui signifie :

1. la variable de parcours, `i`, est dans l'intervalle d'intérêt ([`a`, `b`]). Cela correspond à la Règle 2 et la Règle 4 de l'Invariant Graphique ;
2. `p` contient le produit jusqu'à maintenant. Cela correspond à Règle 5. A noter que le produit se fait entre `a` et `i-1`, ce qui implicitement positionnne la variable de parcours, `i`, par rapport à la ligne de démarcation (Règle 3 et Règle 4) ;
3. `a` n'est pas modifié par le traitement itératif ;
4. `b` n'est pas modifié par le traitement itératif ;
5. les différentes parties de l'Invariant Formel sont connectées à l'aide de l'opérateur de conjonction ($\wedge$) ;
6. la partie encore à traiter (Règle 6) est implicite dans l'Invariant Formel ;
7. toutes les variables et structures identifiées dans l'Invariant Graphique sont bien présentes dans l'Invariant Formel (Règle 7).

Suite de l'Exercice

À vous ! Proposez les Invariants de Boucle pour les Sp et passez à la Sec. 2.5.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 2.5.2

2.5.2 Indice

Contrairement au cours INFO0946, les Invariants de Boucle sont ici proposés sous la forme de prédicats mathématiques. Il est néanmoins compliqué de proposer, directement, le prédicat.

L'approche la plus raisonnable reste donc de partir sur un Invariant Graphique et de le traduire, ensuite, sous la forme d'un Invariant Formel. Pour l'Invariant Graphique, il suffit de repartir des règles définies dans le cours INFO0946. Vous pouvez, aussi, vous appuyer sur le Gli pour le dessin. Le Gli vous permettra en outre de valider votre Invariant Graphique par rapport aux six premières règles.

Suite de l'Exercice

À vous ! Proposez des Invariants de Boucle pour les Sp et passez à la Sec. 2.5.3.

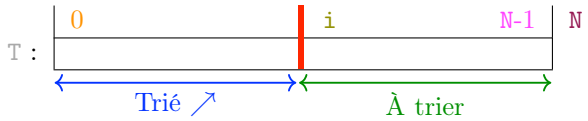
2.5.3 Mise en Commun des Invariants de Boucle

A ce stade de votre cursus, produire un Invariant de Boucle (assez simple) pour un tableau doit se faire naturelle. Dès lors, cette mise en commun n'expliquera pas comment on trouve les Invariants de Boucle. Nous allons, plutôt, nous concentrer sur la traduction Invariant Graphique → Invariant Formel.

- Invariant Graphique et Invariant Formel du SP_0 ;
- Invariant Graphique et Invariant Formel du SP_1 ;
- Invariant Graphique et Invariant Formel du SP_2.

2.5.3.1 Module `tri_insertion()` (SP_0)

Pour rappel, le SP_0 fait référence au problème principal, i.e., tri du tableau. On peut repartir des dessins réalisés pour la spécification afin de créer l'Invariant Graphique. Ceci nous donne :



Traduisons, maintenant, de manière formelle cet Invariant Graphique :

$$\begin{aligned} \text{Inv} \equiv\ & 0 \leq i \leq N && (1)\\ & \wedge\ Trie(T, N, 0, i-1) && (2)\\ & \wedge\ T\ perm\ T_0 && (3)\\ & \wedge\ N = N_0 && (4) \end{aligned}$$

Cet Invariant Formel est composé de quatre parties :

1. Description de la position relative de `i` par rapport aux bornes du tableau ([0, N-1]).
2. Portion du tableau déjà triée.
3. Pas d'ajout éléments dans T. Donc T est une permutation du tableau initial.
4. La taille du tableau n'est pas modifiée.

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

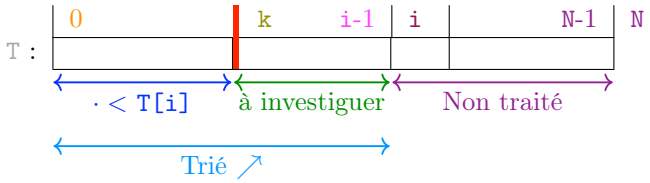
- Invariant Graphique et Invariant Formel du SP_1 ;
- Invariant Graphique et Invariant Formel du SP_2 ;
- construction des modules.

2.5.3.2 Module `insertion()` (SP_1)

Dans l'étape d'analyse, nous avons préalablement estimé qu'il fallait, pour le SP_1, :

1. Trouver la position où insérer `T[i]`. Notons cette position `k`.
2. Décaler `T[k ... i-1]` de une position vers la droite.
3. Placer l'élément $\texttt{T}_0[\texttt{i}]$ dans `T[k]`.

Le point 1 nécessite une boucle, donc, un Invariant de Boucle. Il peut se représenter graphiquement de la façon suivante :



Traduisons maintenant de manière formelle cet Invariant Graphique :

$$
\begin{aligned}
\text{Inv} \equiv\ & 0 \leq k \leq i < N && (1)\\
& \wedge T = T_0 && (2)\\
& \wedge N = N_0 && (3)\\
& \wedge i = i_0 && (4)\\
& \wedge Tri(T, N, 0, i-1) && (5)\\
& \wedge \forall j, 0 \leq j < k, T[j] < T[i] && (6)
\end{aligned}
$$

Cet Invariant Formel est composé de 6 parties : On écrit cela parce que :

1. On représente la position de `k` par rapports aux bornes du tableau.
2. On exprime que le tableau en sera pas modifié.
3. `N` n'est pas modifié.
4. `i` n'est pas modifié.
5. Le tableau est trié sur la portion qui nous intéresse. En fait, cela découle de la PréCondition et de $T = T_0$ donc ce n'est pas grave si vous l'avez omis.
6. On doit représenter formellement une propriété sur une **portion** de tableau, on utilise donc une quantification universelle pour dire que *tous* les éléments de la zone sont inférieurs à `T[i]`.

Suite de l'exercice

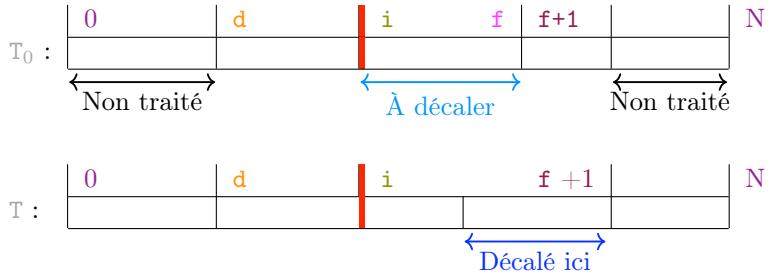
Vous pouvez maintenant passer à la suite de l'exercice :

- Invariant Graphique et Invariant Formel du SP_0 ;
- Invariant Graphique et Invariant Formel du SP_2 ;
- construction des modules.

2.5.3.3 Module `shift()` (SP_2)

Pour rappel, le SP_2 permet de décaler une portion du tableau d'une position vers la droite.

Pour l'Invariant Graphique, vu qu'il faut dessiner le tableau de départ et le tableau d'arrivée, on peut représenter 2 tableaux : `T` (le tableau d'arrivée) et $\texttt{T}_0$ (le tableau de départ) :



On représente le décalage des éléments depuis $\texttt{T}_0$ jusque `T`. Le sous-problème `shift` ne prend que les paramètres `T`, `d`, `f` et `N`.

Pourquoi est-ce que le `i` est à droite de la ligne de démarcation ? **Par expérience !** Ici, le `i` final devra coïncider avec la variable `d` qui est elle-même à droite de sa barre de démarcation. C'est pour cela que `i` est mis de ce côté.

Essayer de le mettre de l'autre côté pour voir comment se traduisent les notations formelles et vous reconsidérerez vite votre choix !

Traduisons maintenant de manière formelle cet Invariant Graphique :

$$\begin{aligned} \text{Inv} \equiv\ & 0 \leq \texttt{d} \leq \texttt{i} \leq \texttt{f} + 1 < \texttt{N} && (1)\\ \wedge\ & \texttt{d} = \texttt{d}_0 \wedge \texttt{f} = \texttt{f}_0 && (2)\\ \wedge\ & \forall j, \texttt{i} < j \leq \texttt{f} + 1, \texttt{T}[j] = \texttt{T}_0[j-1] && (3) \end{aligned}$$

On écrit cela parce que :

1. On représente la position de `d`, `i` et `f` par rapport aux bornes du tableau ;
2. On rappelle qu'on conserve la position de l'intervalle ;
3. On décrit le décalage vers la droite. Ici, on a représenté la quantification dans le tableau `T`, par rapport à $\texttt{T}_0$. Donc les j vont de $\texttt{i}+1$ jusque $\texttt{f}+1$. On aurait pu faire l'inverse et écrire :

   $$\forall j, \texttt{i} \leq j \leq \texttt{f}, \texttt{T}_0[j] = \texttt{T}[j+1],$$

   ce qui revient exactement au même !

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- Invariant Graphique et Invariant Formel du SP_0 ;
- Invariant Graphique et Invariant Formel du SP_1 ;
- construction des modules.

2.6 Construction du Code

Une fois l'Invariant Formel proposé, on peut construire le code en adoptant l'approche constructive et on précisant, à chaque étape, les assertions intermédiaires.

Si vous voyez de quoi on parle, rendez-vous à la Section 2.6.3
Si vous ne savez pas comment fonctionne l'approche constructive, allez la Section 2.6.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 2.6.2

2.6.1 Rappels sur l'Approche Constructive

2.6.1.1 Généralités

Voici les différentes étapes de l'Approche Constructive.

INIT De la PréCondition, il faut arriver dans une situation où l'Invariant de Boucle est vrai.

B Il faut trouver d'abord le Critère d'Arrêt $\neg B$ et en déduire le Gardien de Boucle, B.

ITER Il faut faire progresser le Corps de la Boucle puis restaurer l'Invariant de Boucle.

END Vérifier si la PostCondition est atteinte si $\{Inv \wedge \neg B\}$, sinon, amener la PostCondition.

Fonction de Terminaison Il faut donner une Fonction de Terminaison pour montrer que la boucle se termine (cfr. INFO0946, Chapitre 3, Slides 97 → 107).

Tout ceci doit être justifié sur base de l'Invariant de Boucle, à l'aide d'assertions intermédiaire. Notez que l'ordre dans lequel vous faites ces étapes n'a pas d'importance !

On se retrouvera donc dans la situation suivante :

```
1  // {PréCondition}
2  Init
3  // {Inv}
4  while(B){
5       // {Inv ∧ B}
6       Iter
7       // {Inv}
8  }
9  // {Inv ∧ ¬B}
10 End
11 // {PostCondition}
```

Pour chaque étape, on démarre d'un point de départ et doit dériver les instructions (ainsi que les assertions intermédiaires) permettant d'arriver à l'objectif. Ainsi :

{PréCondition} Init {Inv} : Init est donc un bloc d'instructions pour lequel {PréCondition} est la précondition (i.e., point de départ) et {Inv} la postcondition (i.e., point d'arrivée). À nous de trouver comment arriver au but ({Inv}) en partant du point de départ ({PréCondition}). A noter que Init peut être éventuellement vide. Dans ce cas, il faut démontrer que {PréCondition} $\implies$ {Inv}.

{Inv $\wedge$ B} Iter {Inv} : Iter est donc un bloc d'instructions pour lequel $\{Inv \wedge B\}$ est la précondition (i.e., le point de départ) et {Inv} la postcondition (i.e., le point d'arrivée). À nous de trouver comment arriver au but ({Inv}) en partant du point de départ ({Inv $\wedge$ B}) tout en faisant avancer le problème. Il faut donc garantir qu'on a conservé notre Invariant de Boucle à la fin du Corps de la Boucle.

{Inv $\wedge \neg$B} End {PostCondition} : End est donc un bloc d'instructions pour lequel $\{Inv \wedge \neg B\}$ est la précondition (i.e., point de départ) et {PostCondition} la postcondition (i.e., le point d'arrivée). À nous de trouver comment arriver au but ({PostCondition}) en partant du point de départ ({Inv $\wedge$ $\neg B$}). A noter que End peut être éventuellement vide. Dans ce cas, il faut démontrer que $\{Inv \wedge \neg B\}$ $\implies$ {PostCondition}.

On voit donc bien que l'Invariant de Boucle est l'étape clé autour de laquelle s'articule non seulement la construction de la boucle mais aussi ce qui se passe avant et après la boucle. L'Invariant de Boucle est donc la colle entre les différentes parties du code.

Alerte : Oubli

Attention, pour chaque partie, on n'oublie pas d'indiquer les assertions intermédiaires. C'est obligatoire afin de s'assurer que les différentes parties s'enchaînent correctement.

2.6.1.2 Exemple

Afin d'illustrer l'approche constructive, travaillons sur un exemple. Il s'agit de rédiger une fonction qui permet de calculer le produit de tous les entiers entre des bornes fournies, i.e., `a` et `b` (avec $b > a$).

Pour rappel, l'interface de la fonction est la suivante :

```
/*
 * PréCondition : b > a
 * PostCondition : produit = ∏_{i=a}^{b} i ∧ a = a_0 ∧ b = b_0
 */
int produit(int a, int b);
```

L'Invariant Graphique est le suivant :

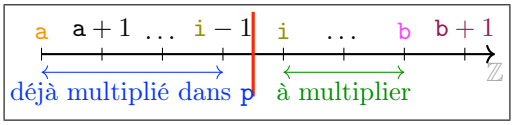


FIGURE 3 – Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

L'Invariant Formel est le suivant :

$$\begin{aligned} \text{INV} \equiv\ & \mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1 \\ & \wedge\, \mathtt{p} = \prod_{j=a}^{\mathtt{i}-1} j \\ & \wedge\, \mathtt{a} = \mathtt{a}_0 \\ & \wedge\, \mathtt{b} = \mathtt{b}_0 \end{aligned}$$

INIT

Sur base de l'Invariant Formel :

$$\text{INV} : \mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1 \wedge \mathtt{p} = \prod_{j=a}^{\mathtt{i}-1} j \wedge \mathtt{a} = \mathtt{a}_0 \wedge \mathtt{b} = \mathtt{b}_0$$

il faut rendre ceci vrai en partant de la PRÉCONDITION (i.e., $\mathtt{b} > \mathtt{a}$). Cela consiste donc à :

- déclarer les variables nécessaires. Soit `i` (la variable d'itération) et `p` (l'accumulateur pour le produit cumulatif) ;
- initialiser les variables. La valeur de `i` est donnée par $\mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1$. La valeur de `p` en découle naturellement.

On obtient donc la séquence suivante :

```
// {b > a}
int i, p;
// {i =?? ∧ p =?? ∧ a = a_0 ∧ b = b_0 ∧ b > a}
i = a;
// {i = a ∧ p =?? ∧ a = a_0 ∧ b = b_0 ∧ b > a}
// {a ≤ i ≤ b + 1 ∧ p =?? ∧ a = a_0 ∧ b = b_0}
p = 1;
// {a ≤ i ≤ b + 1 ∧ p = 1 ∧ a = a_0 ∧ b = b_0}
// {⇒ INV}
```

On peut écrire la ligne 6 car, par la PréCondition, $\texttt{b} > \texttt{a}$. Soit $\texttt{a} \leq \texttt{b} + 1$ (les deux expressions sont identiques).

La transition ligne 8 → ligne 9 est permise car

$\texttt{p} = 1$

$\texttt{p} = \prod_{j=\texttt{a}}^{\texttt{i}-1}$

i est initialisé à a (ligne 4), ce qui donne $\texttt{p} = \prod_{j=\texttt{a}}^{\texttt{a}-1}$. Soit un produit à 0 termes. Par définition, p vaut le neutre de la multiplication, soit 1.

B et Fonction de Terminaison

Le Critère d'Arrêt (¬ B) est donné par le schéma suivant :

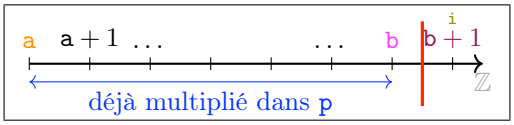


Figure 4 – Exemple d'Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

C'est confirmé par la partie suivante de l'Invariant Formel :

$\texttt{a} \leq \texttt{i} \leq \texttt{b} + 1$

On a donc :

$\neg B \equiv \texttt{i} = \texttt{b} + 1$

On en dérive naturellement le Gardien de Boucle (B) :

$B \equiv \texttt{i} \leq b$

Pour la Fonction de Terminaison, il suffit d'estimer la taille de la zone "encore à traiter" dans l'Invariant Graphique. Soit : $(b - a + 1) - (i - a)$. Ce qui donne :

$f \equiv b - i + 1$

Iter

```
1 while(i <= b){
2   // {Inv ∧ B ≡ a ≤ i ≤ b + 1 ∧ p = ∏_{j=a}^{i-1} j ∧ a = a0 ∧ b = b0 ∧ i <= b}
3   // {a ≤ i ≤ b ∧ p = ∏_{j=a}^{i-1} j ∧ a = a0 ∧ b = b0}
4   p *= i;
5   // {a ≤ i ≤ b ∧ p = [(∏_{j=a}^{i-1} j) × i] ∧ a = a0 ∧ b = b0}
6   // {a ≤ i ≤ b ∧ p = ∏_{j=a}^{i} j ∧ a = a0 ∧ b = b0}
```

```
7     i++;
8     // {Inv}
9   }
```

La transition ligne 2 → ligne 3 est naturelle dans le sens où il s'agit de réécrire le prédicat (ligne 3) de sorte qu'il intègre complètement $\{\text{Inv} \wedge B\}$ (ligne 2).

Cette situation de départ nous indique qu'il faut faire progresser le problème, soit accumuler encore une valeur dans le produit cumulatif (ligne 4). Cette instruction nous conduit à la situation décrite à la ligne 6 (après réécriture du prédicat à la ligne 5).

Pour restaurer l'Invariant Formel (qui est l'objectif à atteindre pour Iter), il suffit d'incrémeter `i` d'un unité, ce qui permettra de restaurer $\mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1$ et $\mathtt{p} = \prod_{j=a}^{\mathtt{i}-1} j$.

End

À la sortie de la boucle, on est dans la situation $\{\text{Inv} \wedge \neg B\}$. Soit :

$$\mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1 \wedge \mathtt{p} = \prod_{j=a}^{\mathtt{i}-1} j \wedge \mathtt{a} = \mathtt{a}_0 \wedge \mathtt{b} = \mathtt{b}_0 \wedge \mathtt{i} = \mathtt{b}+1$$

On voit que la variable d'itération (`i`) prend une valeur particulière dans l'intervalle $\mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1 : \mathtt{b}+1$. On peut donc réécrire le prédicat en remplaçant la variable `i` par sa valeur particulière `b` $+1$. Soit :

$$\mathtt{p} = \prod_{j=a}^{\mathtt{b}+1-1} j \wedge \mathtt{a} = \mathtt{a}_0 \wedge \mathtt{b} = \mathtt{b}_0$$

Et encore :

$$\mathtt{p} = \prod_{j=a}^{\mathtt{b}} j \wedge \mathtt{a} = \mathtt{a}_0 \wedge \mathtt{b} = \mathtt{b}_0$$

Cette situation correspond à la PostCondition, à la différence près que `p` contient le produit cumulatif au lieu de la fonction `produit()` ! Dès lors, à la sortie de la boucle, il nous suffit de retourner la valeur de `p`.

Ce qui nous donne :

```
1  // {Inv ∧ ¬B}
2  return p;
3  // {PostCondition}
```

Code Complet

```
1  int produit(int a, int b){
2    // {PréCondition}
3    int i, p;
4    i = a;
5    p = 1;
6    // {Inv}
7    while(i <= b){
8      p *= i;
9      i++;
10   }//fin while - i
11
12   return p;
13   // {PostCondition}
14 }//fin produit()
```

Suite de l'Exercice

À vous ! Construisez le code pour les Sp et passez à la Sec. 2.6.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 2.6.2

2.6.2 Indice

L'ordre de résolution des Sp n'a pas d'importance. Pour chaque Sp, il faut cependant construire le code en suivant :

Init De la PréCondition, il faut arriver dans une situation où l'Invariant de Boucle est vrai.

B Il faut trouver d'abord le Critère d'Arrêt $\neg B$ et en déduire le Gardien de Boucle.

Iter Il faut faire progresser le programme puis restaurer l'Invariant de Boucle.

End Vérifier si la PostCondition est atteinte si $\{\text{Inv} \wedge \neg B\}$, sinon, amener la PostCondition.

Fonction de Terminaison Il faut donner une Fonction de Terminaison pour montrer que la boucle se termine.

Tout ceci doit être justifié sur base de l'Invariant de Boucle, à l'aide d'assertions intermédiaires. Notez que l'ordre dans lequel vous faites ces étapes n'a pas d'importance !

Suite de l'Exercice

À vous ! Construisez le code pour chaque Sp et passez à la Sec. 2.6.3.

2.6.3 Mise en Commun de l'Approche Constructive

On peut construire le code de chaque Sp.

- Construction du Sp_0 ;
- Construction du Sp_1 ;
- Construction du Sp_2.

2.6.3.1 Construction de `tri_insertion()` (SP_0)

Pour rappel, l'interface de la procédure est la suivante :

```
1 /*
2  * PréCondition : T[0 ... N-1] init
3  * PostCondition : Tri(T, N, 0, N-1)  ∧  T perm T_0
4  */
5 void tri_insertion(int *T, const unsigned int N);
```

L'Invariant Formel est le suivant :

$$\begin{aligned}\text{Inv} \equiv\ & 0 \leq i \leq N \\ & \wedge Trie(T, N, 0, i-1) \\ & \wedge T\ perm\ T_0 \\ & \wedge N = N_0\end{aligned}$$

Init
Sur base de l'Invariant Formel :

$$\text{Inv} \equiv 0 \leq i \leq N \wedge Trie(T, N, 0, i-1) \wedge T\ perm\ T_0 \wedge N = N_0$$

il faut rendre ceci vrai en partant de la PréCondition. Cela consiste donc à :

- déclarer les variables nécessaires. Soit `i` (i.e., la variable d'itération).
- initialiser la variable. Cette valeur de `i` doit vérifier la propriété $Trie(T, N, 0, i-1)$. On peut rendre cette propriété trivialement vraie en considérant un sous-tableau vide (qui, par définition, est trié). Seul $\texttt{i} = 0$ convient.

On obtient donc :

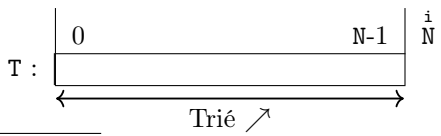
```
1 // {PréCondition}
2 int i;
3 // {T[0...N − 1]init ∧ T = T_0 ∧ N = N_0 ∧ i =??}
4 i = 0;
5 // {T[0...N − 1]init ∧ T = T_0 ∧ N = N_0 ∧ i = 0}
6 // {0 ≤ i ≤ N ∧ T[0...N − 1]init ∧ T = T_0 ∧ N = N_0}
7 // {0 ≤ i ≤ N ∧ Trie(T, N, 0, i − 1) ∧ T  perm  T_0 ∧ N = N_0}
8 // {⇒ Inv}
```

Le passage de la ligne 6 à la ligne 7 est valide car :

- un tableau inchangé est une permutation de lui-même sans le moindre changement (voir la définition de l'opérateur *perm* dans le cours théorique) ;
- un tableau vide est par définition trié [4].

B et Fonction de Terminaison

A la sortie de la boucle, on aura atteint la situation particulière suivante :



4. Remplacez le tableau par une bibliothèque. Si la bibliothèque ne contient aucun livre, elle est naturellement triée. C'est le même raisonnement pour le tableau.

On voit clairement que le Critère d'Arrêt est le suivant :

$$\neg B \equiv i = N$$

On en dérive naturellement le Gardien de Boucle :

$$B \equiv i < N$$

qui représente bien la position relative de la variable d'itération (`i`) par rapport aux bornes du tableau. Ce Gardien de Boucle donne aussi des indications quant au sens du parcours du tableau (i.e., du début vers la fin).

La Fonction de Terminaison est assez facilement obtenue :

$$f \equiv N - i$$

ITER

L'objectif du SP est de trier le tableau par ordre croissant. Nous partons de la situation particulière suivante :

$$\text{INV} \wedge B \equiv 0 \leq i \leq N \wedge Trie(T, N, 0, i-1) \wedge T \ perm \ T_0 \wedge N = N_0 \wedge i < N$$

Ce qui peut se réécrire :

$$\text{INV} \wedge B \equiv 0 \leq i < N \wedge Trie(T, N, 0, i-1) \wedge T \ perm \ T_0 \wedge N = N_0$$

Il reste donc au moins une case à trier. Cela se fait en insérant la valeur de la case courante (i.e., `T[i]`) dans un sous-tableau trié, soit le SP_1. C'est tout à fait possible car le prédicat ci-dessus correspond à la PRÉCONDITION du SP_1. Après l'invocation du SP_1, on se retrouve dans une situation où le sous-tableau trié a augmenté d'une case (la POSTCONDITION du SP_1 est le prédicat qui suit l'invocation du SP_1). Il ne reste plus alors qu'à restaurer l'Invariant de Boucle en incrémentant la variable d'itération (i.e., `i`).

Soit :

```
while(i < N){
  // {INV ∧ B}
  // {PRÉCONDITION_insertion()}
  insertion(T, i, N);
  // {POSTCONDITION_insertion()∧0≤i<N}
  // {0 ≤ i < N ∧ Tri(T, N, 0, i) ∧ TpermT0 ∧ ∧ N = N0}
  i++;
  // {INV}
}
```

END

À la sortie de la boucle, nous sommes dans la situation suivante :

$$\text{INV} \wedge \neg B \equiv 0 \leq i \leq N \wedge Trie(T, N, 0, i-1) \wedge T \ perm \ T_0 \wedge N = N_0 \wedge i = N$$

La variable d'itération (`i`) prend une valeur particulière (`N`). Elle peut donc être remplacée par cette valeur dans le prédicat. Soit :

$$Trie(T, N, 0, N-1) \wedge T \ perm \ T_0 \wedge N = N_0$$

Ce dernier prédicat correspond à la POSTCONDITION. A la sortie de la boucle, il n'y a donc rien à faire.

Code Complet du SP_0

```
void tri_insertion(int *T, const unsigned int N){
  // {PréCondition}
  int i = 0;

  // {Inv}
  while(i < N){
    insertion(T, i, N);
    i++;
  }//fin while - i

  // {PostCondition}
}//fin tri_insertion()
```

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- construction du SP_1 ;
- construction du SP_2 ;
- mise en commun du code.

2.6.3.2 Construction de `insertion()` (SP_1)

Pour rappel, l'interface de la procédure est la suivante :

```
1 /*
2  * PréCondition : Tri(T, N, 0, i-1) ∧ T[0 .. i] init ∧ 0 ≤ i < N
3  * PostCondition : Tri(T, N, 0, i) ∧ T perm T0
4  */
5 void insertion(int *T, unsigned int i, unsigned int N);
```

L'Invariant Formel est le suivant :

$$\begin{aligned}\text{Inv} \equiv\ & 0 \leq k \leq i < N \\ & \wedge T = T_0 \\ & \wedge N = N_0 \\ & \wedge i = i_0 \\ & \wedge Tri(T, N, 0, i-1) \\ & \wedge \forall j, 0 \leq j < k, T[j] < T[i]\end{aligned}$$

Init
Sur base de l'Invariant Formel :

$$\text{Inv} \equiv 0 \leq k \leq i < N \wedge T = T_0 \wedge N = N_0 \wedge i = i_0 \wedge Tri(T, N, 0, i-1) \wedge \forall j, 0 \leq j < k, T[j] < T[i]$$

il faut rendre ceci vrai en partant de la PréCondition. Cela consiste donc à :

- déclarer les variables nécessaires. Soit `k` (i.e., la variable d'itération).
- initialiser la variable. Cette valeur de `k` doit vérifier la dernière partie de l'Invariant Formel : $\forall j, 0 \leq j < k, T[j] < T[i]$. Rendons trivialement vrai cette expression en choisissant un `k` qui fasse que l'ensemble des j possibles soit vide (transition ligne 6 $\rightarrow$ ligne 7). Seul `k` $= 0$ convient.

On obtient donc :

```
1 // {PréCondition}
2 int k;
3 // {Tri(T, N, 0, i − 1) ∧ 0 ≤ i < N ∧ k =??}
4 k = 0;
5 // {Tri(T, N, 0, i − 1) ∧ 0 ≤ i < N ∧ T = T0 ∧ N = N0 ∧ i = i0 ∧ k = 0}
6 // {0 ≤ k ≤ i < N ∧ ∧ T = T0 ∧ N = N0 ∧ i = i0 ∧ Tri(T, N, 0, i − 1)}
7 // {⇒ Inv}
```

B et Fonction de Terminaison

En fait, il faut s'arrêter de faire grandir la zone d'éléments inférieurs à `T[i]` si l'élément suivant est plus grand ou égal. Ce qui donne pour le Critère d'Arrêt :

$$\neg B \equiv k = i \vee tab[k] \geq T[i]$$

Une fois le Critère d'Arrêt atteint, nous avons la position où insérer la valeur dans le tableau.
On en dérive naturellement le Gardien de Boucle (B) :

$$B \equiv k < i \wedge T[k] < T[i]$$

La Fonction de Terminaison est la suivante :

$$f \equiv i - k$$

Iter

L'objectif du Sp est d'insérer une valeur dans un sous-tableau déjà trié. En particulier, la boucle est destinée à déterminer l'endroit du sous-tableau où insérer la valeur. Nous partons de la situation particulière suivante :

$$\text{Inv} \wedge B \equiv 0 \leq k \leq i < N \wedge T = T_0 \wedge N = N_0 \wedge i = i_0 \wedge Tri(T, N, 0, i-1) \wedge \forall j, 0 \leq j < k, T[j] < T[i] \wedge k < i \wedge T[k] < T[i]$$

Ce qui peut être simplifié de la façon suivante :

$$\text{Inv} \wedge B \equiv 0 \leq k < i < N \wedge T = T_0 \wedge N = N_0 \wedge i = i_0 \wedge Tri(T, N, 0, i-1) \wedge \forall j, 0 \leq j \leq k, T[j] < T[i]$$

Il nous suffit donc de faire grandir la variable d'itération pour restaurer l'Invariant Formel. Ce qui donne :

```
while(k < i && T[k] < T[i])
  // {Inv ∧ B}
  k++;
  // {Inv}
```

End

Nous partons de la situation suivante :

$$\text{Inv} \wedge \neg B \equiv 0 \leq k \leq i < N \wedge T = T_0 \wedge N = N_0 \wedge i = i_0 \wedge Tri(T, N, 0, i-1) \wedge \forall j, 0 \leq j < k, T[j] < T[i] \wedge k = i \vee tab[k] \geq T[i]$$

Avec cette situation, on est assuré que tous les éléments avant `T[k]` sont inférieurs à `T[i]` et tous ceux après sont supérieurs. Il reste encore à décaler le tableau et à insérer l'élément au bon endroit. Ce qui donne :

```
// {Inv ∧ ¬B}
int save = T[i];
// {Inv ∧ ¬B ∧ save = T0[i]}
shift(T, k, i-1, N);
// {PostCondition_shift : ∀j, k < j ≤ i − 1, T[j] = T0[j − 1] ⇒ ∀j, k + 1 ≤ j < i + 1, T[j] ≥ save}
// De plus on a toujours {∀j, 0 ≤ j < k, T[j] < save}
T[k] = save;
// {T perm T0 ∧ Tri(T, N, 0, i)}
// {⇒ PostCondition}
```

En gros, après avoir vu que les éléments de `T[k ... i-1]` sont plus grands que `T[i]`, on sauve cette valeur dans `save`. On décale alors `T[k ... i-1]` d'une case vers la droite avec `shift()`, ce qui donne le tableau `T[k+1 ... i]`. La case `T[i]` est écrasée (mais sauvée grâce à `save`). On sait donc que les éléments de `T[0 ... k-1]` sont inférieurs à `save` et les éléments de `T[k+1 ... i]` lui sont supérieurs. Mettre `T[k]` à la valeur de `save` trie donc le tableau.

Alerte : Remarque

Avant d'appeler `shift()`, il faut que sa PréCondition soit vraie. Après l'appel, la PostCondition de `shift()` est vraie. C'est une **propriété fondamentale** de l'approche constructive.

Code Complet du Sp_1

```
void insertion(int *T, unsigned int i, unsigned int N){
  // {PréCondition}
  int k = 0;

  // {Inv}
  while(k < i && T[k] < T[i])
```

```
7       k++;
8
9     int save = T[i];
10    shift(T, k, i-1, N);
11    T[k] = save;
12    // {PostCondition}
13  }//fin insertion()
```

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- construction du SP_0 ;
- construction du SP_2 ;
- mise en commun du code.

2.6.3.3 Construction de `shift()` (SP_1)

Pour rappel, l'interface de la procédure est la suivante :

```
/*
 * PréCondition : T[d ... f+1] init  ∧  0 ≤ d ≤ f + 1 < N
 * PostCondition : ∀i, d < i ≤ f + 1, T[i] = T0[i − 1]
 */
void shift(int *T, unsigned int d, unsigned int f, unsigned int N);
```

L'Invariant Formel est le suivant :

$$\begin{aligned}\text{Inv} \equiv\ & 0 \leq d \leq i \leq f + 1 < N \\ & \wedge d = d_0 \\ & \wedge f = f_0 \\ & \wedge \forall j, i < j \leq f + 1, T[j] = T_0[j-1]\end{aligned}$$

Init
Sur base de l'Invariant Formel :

$$\text{Inv} \equiv 0 \leq d \leq i \leq f + 1 < N \wedge d = d_0 \wedge f = f_0 \wedge \forall j, i < j \leq f + 1, T[j] = T_0[j-1]$$

il faut rendre ceci vrai en partant de la PréCondition. Cela consiste donc à :

- déclarer les variables nécessaires. Soit `i` (i.e., la variable d'itération).
- Initialiser la variable. Cette valeur de `i` doit vérifier la dernière partie de l'Invariant Formel : $\forall j, i < j \leq f + 1, T[j] = T_0[j-1]$. Rendons trivialement vrai cette expression en faisant en sorte que l'intervalle sur les j soit vide. C'est possible en initialisant `i` à `f` + 1.

On obtient donc :

```
// {PréCondition}
int i;
// {T[d...f + 1]init ∧ 0 ≤ d ≤ f + 1 < N ∧ i =??}
i = f + 1;
// {d = d0 ∧ f = f0 ∧ i = f + 1}
// {0 ≤ d ≤ i ≤ f + 1 < N ∧ d = d0 ∧ f = f0}
// {0 ≤ d ≤ i ≤ f + 1 < N ∧ d = d0 ∧ f = f0 ∧ ∀j, i < j ≤ f + 1, T[j] = T0[j − 1]}
// {⇒ Inv}
```

B et Fonction de Terminaison

Vu ce que l'on cherche à faire, il faut s'arrêter lorsque `i` = `d`. Le Critère d'Arrêt est donc :

$$\neg B \equiv i = d$$

On en dérive naturellement le Gardien de Boucle :

$$B \equiv i > d$$

Qui représente bien la position relative de `i` et `d`.
On peut facilement dériver du Gardien de Boucle la Fonction de Terminaison :

$$f \equiv i - d$$

Iter

L'objectif du Sp est de décaler un sous-tableau d'une position vers la droite.

On part de la situation $\{\text{Inv} \wedge B\}$. Soit :

$$\text{Inv} \wedge B \equiv 0 \leq d \leq i \leq f+1 < N \wedge d = d_0 \wedge f = f_0 \wedge \forall j, i < j \leq f+1, T[j] = T_0[j-1] \wedge i > d$$

Ce qui peut se réécrire de la façon suivante :

$$\text{Inv} \wedge B \equiv 0 \leq d < i \leq f+1 < N \wedge d = d_0 \wedge f = f_0 \wedge \forall j, i < j \leq f+1, T[j] = T_0[j-1]$$

L'Invariant Formel indique clairement que nous travaillons "à reculons" (i.e., de `f` vers `d`). Dès lors, partant cette situation (i.e., $\{\text{Inv} \wedge B\}$) il suffit, à chaque itération, de décaler la case `T[i-1]` dans la case `T[i]` et de restaurer l'Invariant de Boucle (faire décroitre `i` d'une unité).

Ce qui donne :

```
while(i > d){
  // {Inv ∧ B}
  T[i] = T[i - 1];
  // {0 ≤ d < i ≤ f + 1 < N ∧  ∧ d = d0 ∧ f = f0∀j, i ≤ j ≤ f + 1, T[j] = T0[j − 1]}
  i--;
  // {Inv}
}
```

End

À la sortie de la boucle, nous sommes dans la situation suivante :

$$\text{Inv} \wedge \neg B \equiv 0 \leq d \leq i \leq f+1 < N \wedge d = d_0 \wedge f = f_0 \wedge \forall j, i < j \leq f+1, T[j] = T_0[j-1] \wedge i = d$$

La variable d'itération (`i`) prend une valeur particulière (`d`). Elle peut donc être remplacée par cette valeur dans le prédicat. Soit :

$$f = f_0 \wedge \forall j, d < j \leq f+1, T[j] = T_0[j-1]$$

Ce dernier prédicat correspond à la PostCondition. A la sortie de la boucle, il n'y a donc rien à faire.

Code Complet du Sp_2

```
void shift(int *T, unsigned int d, unsigned int f, unsigned int N){
  // {PréCondition}
  int i = f+1;

  // {Inv}
  while(i > d){
    T[i] = T[i - 1];
    i--;
  }//fin while - i
  // {PostCondition}
}//fin shift()
```

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- construction du Sp_0 ;
- construction du Sp_1 ;
- mise en commun du code.

2.7 Code Complet

Le code complet est le suivant :

```
void shift(int *T, unsigned int d, unsigned int f, unsigned int N){
  int i = f+1;

  while(i > d){
    T[i] = T[i - 1];
    i--;
  }//fin while - i
}//fin shift()

void insertion(int *T, unsigned int i, unsigned int N){
  int k = 0;

  while(k < i && T[k] < T[i])
    k++;

  int save = T[i];
  shift(T, k, i-1, N);
  T[k] = save;
}//fin insertion()

void tri_insertion(int *T, const unsigned int N){
  int i = 0;

  while(i < N){
    insertion(T, i, N);
    i++;
  }//fin while - i
}//fin tri_insertion()
```