

UNIVERSITÉ DE LIÈGE

INFO0946

INTRODUCTION À LA PROGRAMMATION

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

Structures de Données

Benoit DONNET

Simon LIÉNARDY

Géraldine BRIEVEN

Tasnim SAFADI Lev MALCEV

3 octobre 2021



Préambule

Exercices

Dans ce GAMECODE, nous vous proposons de suivre pas à pas la résolution d'un exercice portant sur le tri d'un tableau en fonction d'une valeur pivot.

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.

5.1 Rappels sur les Tableaux

Généralités

Un *tableau* (ou *vecteur*) est une structure de données homogène qui permet de stocker, sous un identificateur unique, une collection finie (la structure est donc *bornée*) de valeurs du même type. L'accès aux données se fait de manière aléatoire à l'aide d'un *indice* qui évolue dans les limites des bornes du tableau. Les différents éléments composant le tableau sont stockés de manière contigüe en mémoire.

	0	1	2	3	4	5	6	7	8	9	10	11
t :	4	-9	10	12	21	45	45	67	3	4	0	2

FIGURE 1 – Représentation graphique d'un tableau à 12 valeurs entières.

La représentation graphique d'un tableau est donnée à la Fig. 1. Le tableau d'identificateur `t` peut contenir, au maximum, 12 valeurs. 12 est la *taille* du tableau. Cette taille est fixée à la déclaration et doit être une valeur entière strictement positive. Les valeurs stockées dans `t` (considérées comme des valeurs à gauche ou à droite) appartiennent à l'intervalle $[0, 11]$. C'est dans cet intervalle qu'on va spécifier l'*indice* permettant d'accéder à chacune des cases du tableau (on ne peut accéder qu'à une seule case à la fois). De manière générale, les indices d'un tableau évolueront dans l'intervalle $[0, N - 1]$, où N est la taille du tableau.

Alerte : Débordement

L'erreur classique, pour le programmeur débutant, est d'essayer d'accéder au tableau en dehors des bornes du tableau (i.e., l'indice $\notin [0, N - 1]$, avec N la taille du tableau). Une telle erreur est appelée *débordement* ou *buffer overflow*.

Il n'y a aucun mécanisme automatique qui vous permet d'éviter cela (du moins en C). Il est de la responsabilité du programmeur de s'assurer que le tableau est toujours accédé à l'intérieur de ses bornes.

Si vous faites un débordement de tableau et que vous avez "de la chance", l'accès mémoire que vous tentez de faire sera interdit par votre système d'exploitation et vous générerez une erreur de type SEGMENTATION FAULT. Si vous "n'avez pas de chance", le programme continuera de tourner comme si de rien n'était mais les résultats seront incohérents. A noter que dans ce dernier cas, le débordement peut permettre à un attaquant de prendre possession, à distance, de votre machine (attaque par buffer overflow – voir [INFO0045](#)).

Dans tous les cas, un débordement de tableau amènera toujours une note nulle lors d'une évaluation (Challenge, Interro, Examen).

Dans un tableau, l'ordre des valeurs n'a pas spécialement d'importance. Aucune propriété particulière (e.g., une seule occurrence de chaque valeur, tri quelconque) n'est imposée aux valeurs d'un tableau.

Déclaration

A ce stade-ci du cours², les tableaux doivent être déclarés de manière statique, i.e., la taille du tableau doit être connue à la compilation. Le tableau doit être déclaré dans votre programme et peut stocker n'importe quel type de données (`int`, `char`³, `float`, `double`). Ainsi, par exemple, pour le tableau illustré à la Fig. 1, on a

2. Nous généraliserons l'utilisation des structures de données, notamment en leur apportant un aspect dynamique, dans le Chapitre 8.

3. Un tableau de caractères est un cas un peu particulier. Nous vous renvoyons au cours, Chapitre 5, pour voir comment manipuler efficacement un tableau de caractères.

Extrait de code 1 – Déclaration d’un tableau

```
1 int main(){
2     const unsigned int N = 12;
3
4     int t[N];
5 }
```

Un tel tableau `t` permet de stocker en mémoire, de manière contiguë, 12 valeurs entières. Attention, dans cette déclaration, le tableau `t` n’est pas initialisé. Cela signifie que les 12 positions du tableau contiennent des valeurs aléatoires (i.e., ce qui traîne en mémoire à cet endroit là, à ce moment là).

Vous avez toujours la possibilité d’initialiser, à la déclaration, un tableau. Dans ce cas, vous devez fournir une valeur pour chaque case du tableau. Ainsi, pour le tableau illustré à la Fig. 1, on a

Extrait de code 2 – Déclaration et initialisation d’un tableau

```
1 int main(){
2     const unsigned int N = 12;
3
4     int t[N] = {4, -9, 10, 12, 21, 45, 45, 67, 3, 4, 0, 2};
5 }
```

Manipulation

L’accès à un tableau se fait case par case (on ne peut pas accéder, simultanément, à plusieurs cases du tableau ni spécifier un intervalle d’indices renvoyant un “sous-tableau”) à l’aide d’un *indice* placé entre `[]` juste après l’identificateur du tableau⁴. L’indice est une expression donnant un résultat entier⁵.

Une case du tableau peut être utilisée comme *valeur à gauche* et donc servir de réceptacle à une affectation. Par exemple, si on reprend le tableau tel que déclaré dans l’Extrait de Code 2, le code suivant

Extrait de code 3 – valeur à gauche et tableau

```
1 int main(){
2     //Déclaration et initialisation du tableau
3
4     t[5] = 10;
5 }
```

donne le tableau illustré à la Fig. 2.

	0	1	2	3	4	5	6	7	8	9	10	11
t :	4	-9	10	12	21	10	45	67	3	4	0	2

FIGURE 2 – Représentation graphique d’un tableau à 12 valeurs entières.

Une case du tableau peut aussi être utilisée comme *valeur à droite*, intervenant ainsi dans l’évaluation d’une expression. Par exemple, si on prend, à nouveau, le tableau tel que déclaré dans l’Extrait de Code 2, on a les instructions suivantes :

Extrait de code 4 – valeur à droite et tableau

```
1 #include <stdio.h>
2
3 int main(){
4     //Déclaration et initialisation du tableau
5
6     int x = t[3] * 2;
```

4. L’opérateur `[]` est très prioritaire. Nous vous renvoyons au Chapitre 1, Slide 36, pour plus de détails

5. Attention, il faut toujours être certain que le résultat de l’expression se trouve dans les bornes du tableau.

```
7  
8 printf("%d_%d\n", x, t[3]);  
9 }
```

qui affichent à l'écran les valeurs 36 (x) et 12 (t[3]).

5.2 Énoncé

On considère des tableaux, de taille N , contenant uniquement des valeurs entières (le tableau est déjà rempli). Par exemple, pour $N = 5$

t :

0				4
7	9	13	18	10

On désire écrire un programme permettant de trier ce tableau **t** en fonction d'une valeur pivot **x**, lue au clavier. Ainsi, toute valeur dans **t** strictement inférieure à **x** se retrouvera, après application du programme, dans la moitié inférieure (ou, si vous préférez, la moitié gauche) de **t**. A l'inverse, toute valeur dans **t** supérieure ou égale à **x** se retrouvera dans la moitié supérieure (ou, si vous préférez, la moitié droite). Attention, l'ordre des valeurs dans la partie inférieure (ou supérieure) du tableau n'a aucune importance. Seule la séparation par rapport au pivot **x** compte.

Par exemple, si on applique ce tri au tableau **t** indiqué supra et si la valeur pivot **x** vaut 11, alors on obtient le tableau **t** suivant :

t :

0				4
7	9	10	18	13

Pour effectuer ce tri, on désire un code ayant une complexité proportionnelle à la taille du tableau (i.e., $O(N)$) et composée d'une et une seule boucle de type while. Vous ne pouvez pas utiliser de tableau intermédiaire.

On vous demande, en suivant la méthodologie vue au cours, de compléter un programme qui va trier le tableau comme indiqué ci-dessus. Le squelette de code à compléter est le suivant :

Extrait de code 5 – Squelette de code à compléter

```
1 #include <stdio.h>
2
3 int main(){
4     const unsigned int N = ...;
5     int t[N];
6
7     /*
8      * Code de remplissage du tableau t
9      * Ce code ne vous est pas donné.
10     */
11
12     // Votre code viendra ici (variables + instructions)
13 }
```

5.2.1 Méthode de résolution

Pour résoudre ce problème, nous allons appliquer, pas à pas, la méthodologie de développement vue au cours. Voici le programme :

1. Définition du problème (Sec. 5.3) ;
2. Analyse du problème (Sec. 5.4) ;
3. Invariant Graphique (Sec. 5.5) ;
4. Construction de la ZONE 1 (Sec. 5.6) ;
5. Construction du Critère d'Arrêt et de la Fonction de Terminaison (Sec. 5.7)

6. Construction de la ZONE 2 (Sec. 5.8);
7. Construction de la ZONE 3 (Sec. 5.9);
8. Code final (Sec. 5.10);

5.3 Définition du Problème

La première étape de la méthodologie de développement (cfr. Chapitre 3, Slide 5) est la *définition du problème*.

Si vous voyez de quoi on parle, rendez-vous à la Section [5.3.3](#)

Si la notion de définition d'un problème, voyez la Section [5.3.1](#)

Si vous ne voyez pas comment faire, reportez-vous à l'indice [5.3.2](#)

5.3.1 Rappels sur la Définition d'un Problème

Définir un problème revient à le réexprimer en fonction de la définition (large) d'un programme. Un *programme* est un traitement exécuté sur des données en entrée (*input*) et qui fournit un résultat (*output*). La définition d'un problème revient donc à exprimer les données en entrée, le résultat attendu (et éventuellement la "forme" du résultat) et, enfin, les (grands) objets utilisés pour atteindre ce résultat. On ne dit jamais comment on va atteindre ce résultat. Cette étape de définition doit se faire dès le début (sans avoir écrit la moindre ligne de code) puisque l'objectif caché est de bien comprendre le problème à résoudre.

Voici la forme que doit prendre la définition d'un problème :

Input correspond aux données en entrée du problème à résoudre.

Output correspond aux données en sortie (et, si applicable, une description succincte du format attendu).

Objet(s) Utilisé(s) reprend les données/variables déjà identifiées (grâce à l'Input). Il s'agit de décrire les objets et de leur associer une déclaration en C. Il est évident que les identifiants donnés à ces objets sont les plus pertinents et proches possible du problème à résoudre. Les types C des variables doivent être précis et en accord avec la sémantique de l'objet.

Attention, un programme peut ne pas avoir d'input ni d'objet(s) utilisé(s). Par contre, il doit toujours fournir un résultat (un programme qui ne fait rien n'a aucun intérêt).

Une fois cela fait, on peut très facilement poser un premier canevas du code, i.e., inclure les dérives de compilation nécessaires, rédiger le bloc du `main()` et y ajouter les déclarations des premières variables telles que décrites dans les Objets Utilisés. Assurez-vous d'être cohérent entre ce que vous indiquez dans les Objets Utilisés et le canevas de votre code. Une incohérence engendrera nécessairement la colère de l'équipe pédagogique !

Par exemple, si le problème consiste à écrire n fois un caractère à l'écran, la définition pourrait être la suivante :

Input : le caractère à afficher et n , le nombre d'occurrences du caractère. Les deux informations sont lues au clavier.

Output : une ligne, sur la sortie standard, comprenant n occurrences du caractère donné.

Objets Utilisés :

n , le nombre d'occurrences du caractères c

$n \in \mathbb{N}$ ⁶

`unsigned int n;`

c , le caractère à écrire sur la sortie standard.

`char c;`

On en tire, naturellement, le canevas suivant :

```
1 #include <stdio.h>
2
3 int main(){
4     unsigned int n;
5     char c;
6
7     //déclaration des variables supplémentaires
8
9     //résolution des différents sous-problèmes
10 } //fin programme
```

6. Une valeur négative n'a pas de sens ici, on ne peut pas écrire -5 fois un caractère sur la sortie standard.

Alerte : Piège

Il est fréquent que des étudiants qui indiquent, dans les Objets Utilisés, l'entièreté des variables de leur code, comme, par exemple la variable utilisée pour l'itération. Cela n'a pas de sens car cette variable, vous l'inférez de l'**Invariant Graphique** en construisant la **ZONE 1** de votre code. Ne tombez donc pas dans le piège !

Suite de l'Exercice

À vous ! Définissez le problème et passez à la Sec. **5.3.3**.

5.3.2 Indice

La meilleure façon de bien définir un problème, c'est de s'imprégner au mieux de l'énoncé. Il s'agit donc, ici, de relire plusieurs fois l'énoncé et de veiller à bien identifier ce que le programme prend en entrée (Input) et ce qu'il fournit comme résultat (Output). De l'Input, on peut facilement inférer les Objets Utilisés.

Suite de l'Exercice

À vous ! Définissez le problème et passez à la Sec. 5.3.3.

5.3.3 Mise en Commun de la Définition du Problème

L'énoncé nous indique les éléments pertinents nécessaires à la définition du problème (mis en évidence en fluo) :

On désire écrire un programme permettant de trier ce tableau t à N valeurs entières en fonction d'une valeur pivot x , lue au clavier. Ainsi, toute valeur dans t strictement inférieure à x se retrouvera, après application du programme, dans la moitié inférieure (ou, si vous préférez, la moitié gauche) de t . A l'inverse, toute valeur dans t supérieure ou égale à x se retrouvera dans la moitié supérieure (ou, si vous préférez, la moitié droite). Attention, l'ordre des valeurs dans la partie inférieure (ou supérieure) du tableau n'a aucune importance. Seule la séparation par rapport au pivot x compte.

On dispose donc en entrée d'un tableau t et d'une valeur x (Input). L'objectif est de trier t en fonction de la valeur pivot (Output).

La définition du problème est donc

Input : t , un tableau de N entiers (déjà rempli) et la valeur pivot, x , lue au clavier.

Output : Le tableau t est trié par rapport à la valeur pivot.

Objets Utilisés :

N , la taille du tableau.

$N \in \mathbb{N}$ et ne peut pas être modifié

`const unsigned int N = ...;`

t , un tableau de N entiers (déjà rempli).

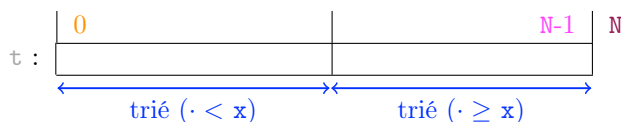
`int t[N];`

x , la valeur pivot.

$x \in \mathbb{Z}$

`int x;`

On peut représenter graphiquement l'Output comme suit :



Les codes couleur utilisés pour la représentation graphique de l'Output correspondent à ceux de l'**Invariant Graphique**. Ceci nous sera utile lorsqu'il s'agira de construire l'Invariant Graphique.x

On peut donc compléter le canevas du code (donné dans l'énoncé) :

```
1 #include <stdio.h>
2
3 int main(){
4     const unsigned int N = ...;
5     int t[N];
6
7     /*
8      * Code de remplissage du tableau brut
9      * Ce code ne vous est pas donné.
10     */
11
12     unsigned int x;
13 } //fin programme
```

Notez bien la cohérence entre les Objets Utilisés dans la définition et les variables déclarées dans le canevas. Dans les deux cas, les identificateurs et les types sont identiques !

Suite de l'Exercice

Nous pouvons maintenant passer à l'analyse et découpe en SP du problème.

5.4 Analyse du Problème

La deuxième étape de la méthodologie de développement (cfr. Chapitre 3, Slide 5) est l'*analyse du problème*.

Si vous voyez de quoi on parle, rendez-vous à la Section [5.4.3](#)

Si vous ne savez pas ce qu'est l'analyse du problème, allez à la Section [5.4.1](#)

Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice [5.4.2](#)

5.4.1 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. Il est toujours possible de **définir** chaque SP. 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* (i.e., `scanf()`, `printf()` – plus de détails lors du Chapitre 6).

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, l'**Output** du problème doit être atteint. De même, le premier SP doit s'appuyer sur les informations fournies par l'**Input**. On envisage deux formes d'agencement des SP :

1. *Linéaire* : $SP_i \rightarrow SP_j$. Dans ce cas, le SP_j est exécuté après le SP_i . Typiquement, l'**Output** du SP_i sert d'**Input** au SP_j .
2. *Inclusion* : $SP_j \subset SP_i$. 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.

Pour chaque SP, il peut s'avérer utile de le **définir** proprement. En particulier, trouver une représentation graphique de l'Output d'un SP vous facilitera la vie pour trouver l'Invariant Graphique associé.

Suite de l'Exercice

À vous ! Analysez le problème et passez à la Sec. 5.4.3.

Si vous séchez, reportez-vous à l'indice à la Sec. 5.4.2

5.4.2 Indice

Pour découper le problème principal, il convient de repartir de l'énoncé et de la définition du problème.

Attention, le remplissage du tableau **t** ne doit pas faire partie de vos SP. Ce problème est déjà résolu pour vous et vous devez considérer que votre intervention commence après ce remplissage.

Vous devez, bien entendu, veiller à ce que votre découpe soit pertinente : ni trop haut niveau (e.g., SP₁ : résoudre le problème), ni trop bas niveau (e.g., SP₁ déclarer des variables). Dans tous les cas, après exécution du dernier SP, le problème général doit être résolu (i.e., vous devez avoir atteint l'Output décrit dans la définition).

Suite de l'Exercice

À vous ! Analysez le problème et passez à la Sec. 5.4.3.

5.4.3 Mise en Commun de l'Analyse du Problème

Comparé au problème du Chapitre 3, l'analyse ici est assez simple. On envisage deux SP :

SP₁ : lecture de **x** au clavier.

SP₂ : tri du tableau **t** en fonction de **x**.

L'enchaînement est le suivant (dérivé naturellement du jeu de couleurs utilisé) :

$$\text{SP}_1 \rightarrow \text{SP}_2$$

Alerte : Studentite aigüe !

Si vous n'avez pas une découpe en SP aussi simple, c'est que vous avez contracté une *studentite aigüe*, la maladie caractérisée par une inflammation de l'étudiant qui cherche midi à quatorze heures, probablement en pratiquant de manière intensive le **microscopisme**.

Heureusement, ce n'est pas dangereux tant que vous conservez une certaine distanciation sociale ! Faites une petite pause dans cette résolution d'exercice. Prenez le temps de vous oxygéner, en faisant un peu d'exercice physique par exemple. Trois répétitions de dix pompes devraient faire l'affaire.

Suite de l'Exercice

On peut maintenant passer à l'**Invariant de Boucle**.

5.5 Invariant Graphique

La troisième étape de la méthodologie de développement (cfr. Chapitre 3, Slide 5) est l'*écriture du code*. Si jamais l'écriture implique un traitement itératif, vous devez, au préalable, établir un Invariant Graphique qui vous permettra, ensuite, de construire votre code.

Si vous voyez de quoi on parle, rendez-vous à la Section [5.5.3](#)

Si vous ne savez pas ce qu'est un Invariant Graphique, allez la Section [5.5.1](#)

Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice [5.5.2](#)

5.5.1 Rappels sur l'Invariant Graphique

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

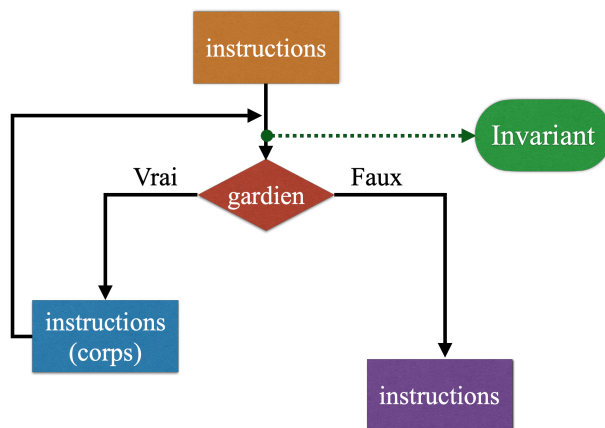


FIGURE 3 – Sémantique de l'Invariant Graphique.

La Fig. 3 présente la sémantique du Invariant Graphique. Dans cette figure, l'**expression** représente le Gardien de Boucle et le **bloc d'instructions** représente le Corps de la Boucle. Le **bloc orange** correspond aux instructions avant la boucle. Le **bloc mauve** correspond aux instructions après la boucle. Les flèches indiquent le sens du flux du programme, i.e., l'ordre d'exécution des différentes instructions et d'évaluation des expressions. Il s'agit ici d'une boucle **while** (mais on peut facilement envisager une boucle **for**). Dans la Fig. 3, on voit bien que l'Invariant de Boucle est en dehors du flux d'exécution du programme et qu'il s'agit d'une propriété qui est évaluée avant chaque évaluation du Gardien de Boucle.

Représentation de l'Invariant de Boucle

On peut représenter un Invariant de Boucle de multiples façons. Dans le cadre du cours INFO0946, nous choisissons de faire un Invariant Graphique.

Au second quadrimestre, dans le cours INFO0947, nous verrons comment traduire l'Invariant Graphique en un prédicat mathématique.

Exemple

Voici un exemple d'Invariant Graphique. Le problème consiste à calculer le produit de tous les entiers entre des bornes fournies, i.e., a et b (avec $b > a$ – les deux valeurs sont fournies au clavier), et à afficher le produit à l'écran. La **définition** du problème est la suivante :

Input : a et b (lus au clavier), les bornes entre lesquelles les entiers doivent être multipliés (avec $b > a$).

Output : le produit cumulé de tous les entiers dans $[a, b]$ est affiché à l'écran.

Objets Utilisés :

a , la borne inférieure

$a \in \mathbb{Z}$

`int a;`

b , la borne supérieure

$b \in \mathbb{Z}$

`int b;`

La Fig. 4 montre l'Invariant Graphique correspondant.

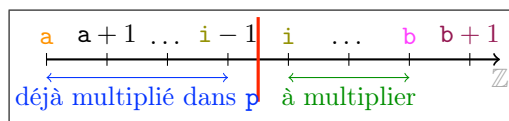


FIGURE 4 – Exemple d'Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

On représente d'abord les entiers entre les limites du problème (a et b) grâce à une ligne numérique (chaque marque représente un entier). Cette ligne numérique correspond, bien entendu, à la droite des entiers. Ensuite, puisque tous les entiers entre a et b devront être considérés par le programme, nous représentons la situation (du programme) après un certain nombre d'itérations. Une barre verticale (rouge) est dessinée au milieu de la droite des entiers, la divisant ainsi en deux zones (une telle barre est appelée *ligne de démarcation*). La partie gauche, en vert, représente les entiers qui ont déjà été multipliés dans une variable p (p est donc l'accumulateur stockant les résultats intermédiaires, itération après itération). La partie gauche, en bleu, représente les entiers qui doivent encore être multipliés. Nous décidons de nommer l'entier le plus proche, sur la droite, de la ligne de démarcation avec la variable i . Évidemment, les variables i et p que nous venons d'introduire devront se retrouver dans le code (cfr. la **construction** basée sur l'Invariant Graphique).

Règles

Pour construire un Invariant Graphique satisfaisant, il est impératif de suivre sept règles :

- Règle 1 : réaliser un dessin pertinent par rapport au problème (la droite des entiers dans la Fig. 4) et le nommer (\mathbb{Z} dans la Fig. 4) ;
- Règle 2 : placer sur le dessin les bornes de début et de fin (a , b , et $b + 1$ dans la Fig. 4) ;
- 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. 4). 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. 4) ;

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. 4) ;

Règle 6 : identifier ce qu'il reste à faire dans les itérations suivantes ("à multiplier" dans la Fig. 4) ;

Règle 7 : Toutes les structures identifiées et variables sont présentes dans le code (a, b, i, et p sur la Fig. 4)⁷.

N'hésitez pas à créer votre Invariant Graphique à l'aide du **GLIDE**. Il contient un mode "Débutant" qui vous guide, règle après règle, dans la construction de votre Invariant Graphique. Le mode "Expert" est plus libre mais permet, néanmoins, de confronter votre Invariant Graphique avec les règles décrites ci-dessus.

Alerte : Règle d'or

Une boucle == un Invariant Graphique.

Si votre code nécessite 2450 boucles, alors vous devez définir 2450 Invariants Graphiques.

On n'est pas dans le Seigneur des Anneaux^a ! Il ne peut y avoir un Invariant Graphique qui représente toutes les boucles de votre code.

a. J. R. R. Tolkien. *The Lord of the Rings*. Ed. Allen & Unwin. 1954/1955.

Code Couleur

Les Invariants Graphiques dans un GAMECODE, dans le cours théorique et dans le GLIDE suivent le code couleur suivant :










Éléments du dessin	Couleur	Règle #
Nom de la structure		Règle 1
Borne minimale		Règle 2
Borne maximale		Règle 2
Taille de la structure		Règle 2
Ligne(s) de démarcation		Règle 3
Étiquettes ligne démarcation		Règle 4
Ce qui a été réalisé jusqu'à maintenant		Règle 5
Zones "à faire"		Règle 6
Propriétés conservées		Règle 5 + Règle 6

TABLE 1 – Code couleur pour les éléments dans la représentation graphique d'un Invariant de Boucle.

Suite de l'Exercice

À vous ! Rédigez l'Invariant Graphique et passez à la Sec. 5.5.3.

Si vous séchez, reportez-vous à l'indice à la Sec. 5.5.2

⁷. Plus de détails sur le lien entre le Invariant Graphique et la construction du code dans le **rappel** associé

5.5.2 Indice

L'Invariant Graphique est là pour vous aider à construire votre boucle mais aussi le code autour de la boucle. Un Invariant Graphique n'a donc de sens qu'avec un traitement itératif.

Le premier indice est donc le suivant : suite à l'**analyse**, identifiez les SP qui ont besoin d'un traitement itératif. Ce sont les SP pour lesquels un Invariant Graphique doit être proposé. Gardez en mémoire la règle d'or : une boucle == un Invariant Graphique

Quand vous essayez de trouver un Invariant Graphique, procédez de la sorte :

- suivez scrupuleusement les règles 1 → 7 pour la construction de votre Invariant Graphique. Les règles 1 → 4 vous permettent de syntaxiquement construire l'Invariant Graphique (le dessin et ce qui va autour). Les règles 5 et 6 vous permettent de donner du sens au dessin (sémantique de l'Invariant Graphique).
- faites un dessin qui soit en relation avec le SP sur lequel vous travaillez (**Règle 1** de la construction d'un Invariant Graphique). Le dessin doit représenter, ici, la structure de données à manipuler.
- soyez bien conscient des Inputs et Outputs du SP sur lequel vous travaillez. Ceci vous donne des informations sur certaines variables disponibles (typiquement les bornes – **Règle 2** de la construction d'un Invariant Graphique) et sur l'objectif à atteindre (ce qui vous donne une information sur ce que chaque itération devra caculer – **Règle 5** de la construction d'un Invariant Graphique).
- placez une (ou plusieurs) lignes de démarcation sur le dessin (**Règle 3** de la construction d'un Invariant Graphique) et positionnez proprement une variable d'itération à gauche ou à droite de la ligne et non juste sur la ligne (**Règle 4** de la construction d'un Invariant Graphique).
- indiquez ce que les itérations précédentes ont déjà calculé (**Règle 5** de la construction d'un Invariant Graphique) en vous appuyant sur l'Output de votre SP. Soyez précis. Une formulation du style “déjà calculé” ne veut absolument rien dire.
- Enfin, indiquez ce qu'il vous reste à faire dans les itérations suivantes (**Règle 6** de la construction d'un Invariant Graphique).

Vous pouvez aussi procéder par “éclatement de la PostCondition”.

Dans tous les cas, n'hésitez pas à vous appuyer sur le **GLIDE** pour le dessin. Le **GLIDE** vous permettra en outre de valider votre Invariant Graphique par rapport aux six premières règles.

Suite de l'Exercice

À vous! Rédigez l'Invariant Graphique et passez à la Sec. **5.5.3**.

5.5.3 Mise en Commun de l'Invariant Graphique

L'**analyse** du problème nous indique qu'il y a deux SP. Le **SP₁** ne demande aucune boucle car c'est une simple lecture au clavier. Le **SP₂**, lui, exige un traitement itératif. Conformément à l'**énoncé**, nous ne pourrons utiliser qu'une et une seule boucle (**while**) pour résoudre le SP₂.

On peut envisager deux techniques pour produire l'Invariant Graphique du SP₂ :

- une **application stricte des règles** ;
- une application de la technique dite de l'**éclatement de la Postcondition**.

5.5.3.1 Construction de l'Invariant Graphique par Application Stricte des Règles

Reprenons la description du SP_2 et tâchons d'identifier l'Input et l'Output.

tri du tableau t en fonction de x .

Le SP_2 nécessite de manipuler un tableau. Le dessin de notre Invariant Graphique devra donc représenter ce tableau, de manière générale (i.e., pas celui qui sert à illustrer l'énoncé ou le **rappel général sur les tableaux**). En outre, ce tableau correspond à la variable t . Il faut donc nommer le tableau adéquatement. Le résultat est donné à la Fig. 5a. Le tableau est bien généralisé car il ne contient aucune valeur particulière.

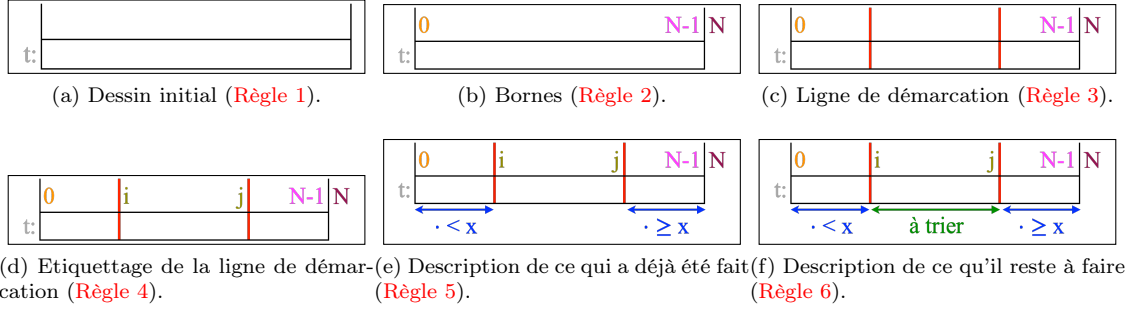


FIGURE 5 – Construction de l'Invariant Graphique pour le SP_2 .

La description du SP_2 ne nous apprend rien, directement, sur les bornes du problème. Par contre, puisque nous devons manipuler un tableau, les bornes sont assez évidentes : ce sont les bornes du tableau t (qui est de taille N). Le placement des bornes est assez simple car nous **savons** que les indices d'un tableau évolue entre 0 et la taille du tableau moins un. Soit, pour le SP_2 , dans l'intervalle $[0, N-1]$. Ceci est illustré à la Fig. 5b.

Maintenant que le dessin est borné, nous pouvons placer une ou plusieurs ligne(s) de démarcation afin de distinguer le travail déjà effectué par les itérations précédentes de ce qu'il nous reste à faire. Pour savoir comment placer la/les ligne(s) de démarcation, repartons de l'énoncé :

Toute valeur dans t strictement inférieure à x se retrouvera, après application du programme, dans la **moitié inférieure** (ou, si vous préférez, la moitié gauche) de t . A l'inverse, **toute valeur dans t supérieure ou égale à x** se retrouvera dans la **moitié supérieure** (ou, si vous préférez, la moitié droite). Attention, l'ordre des valeurs dans la partie inférieure (ou supérieure) du tableau n'a aucune importance. Seule la séparation par rapport au pivot x compte.

L'objectif est donc de diviser le tableau en deux zones d'intérêt : à gauche les valeurs inférieures à x , à droite les valeurs supérieures ou égales à x . Dit autrement, la situation finale devra être celle illustrée à la Fig. 6. Partant de cette situation finale, il suffit de réduire la taille de la zone $< x$ vers la gauche et la taille de la zone $\geq x$ vers la droite du tableau t . Nous avons donc deux lignes de démarcation. Une qui permet de délimiter la zone $< x$ et l'autre la zone $\geq x$. Cette situation est illustrée à la Fig. 5c.

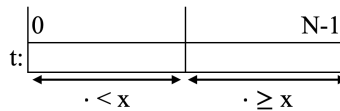


FIGURE 6 – Illustration de la situation finale.

Il faut ensuite étiqueter les lignes de démarcation avec des variables (deux lignes de démarcation == deux variables). La grande question à laquelle nous devons répondre est la suivante : de quelle côté d'une

ligne de démarcation mettre la variable ? Soit à l'intérieur des zones d'intérêt (i.e., $[0, i]$ pour la partie gauche et $[j, N - 1]$ pour la partie droite), soit à l'extérieur des zones d'intérêt (i.e., $[0, i - 1]$ pour la partie gauche et $[j + 1, N - 1]$ pour la partie droite). Arbitrairement, nous choisissons à l'extérieur des zones d'intérêt. Cette situation est illustrée à la Fig. 5d.

Maintenant que l'architecture générale de l'Invariant Graphique est placée, on peut lui donner un sens en indiquant ce qui a déjà été effectué lors des itérations précédentes. La situation, illustrée à la Fig. 6, nous indique comment étiqueter les zones d'intérêt. La Fig. 6 n'est jamais qu'une illustration graphique de l'Output du SP_2 . La Fig. 5e adapte l'Output du SP_2 au traitement partiel après un certain nombre d'itérations.

Pour terminer l'Invariant Graphique, il ne reste plus qu'à indiquer ce qu'il reste à faire, à nouveau en s'appuyant sur le vocabulaire du problème. C'est illustré à la Fig. 5f.

Le schéma ainsi obtenu (Fig. 5f) est l'Invariant Graphique pour le SP_2 .

Suite de l'Exercice

Maintenant, vous pouvez

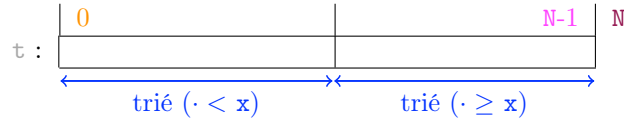
- voir la construction de l'Invariant Graphique par **éclatement de la Postcondition** ;
- rédiger la **ZONE 1** du code.

5.5.3.2 Construction de l'Invariant Graphique par Éclatement de la Postcondition

La technique de l'éclatement de la Postcondition fonctionne en deux étapes :

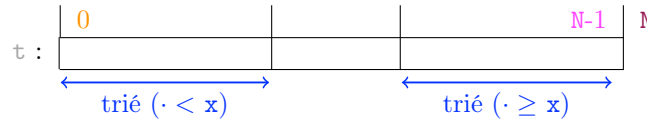
1. faire une représentation graphique de l'Output ;
2. appliquer les règles (restantes) sur le dessin.

La **définition** du problème nous indique clairement que l'Output du problème général est identique à l'Output du SP_2 . À savoir un tableau trié en fonction de la valeur pivot. La représentation graphique de l'Output obtenue dans la **définition** du problème est la suivante :

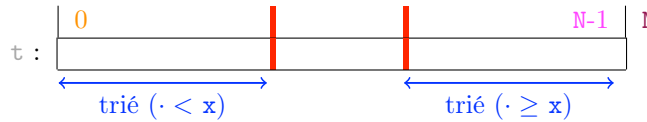


Un tel dessin de l'Output fait déjà apparaître naturellement l'application des Règles **Règle 1** et **Règle 2**. La partie en **bleu** fait penser à l'application de la Règle **Règle 5** mais dans le cas particulier où le problème est résolu. Il s'agit donc d'une situation particulière et non d'une situation générale, en cours d'itération.

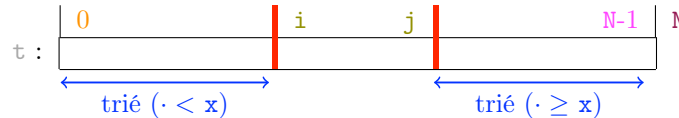
À ce stade, il suffit donc de généraliser le schéma de l'Output en faisant apparaître un résultat partiel (i.e., la Règle **Règle 5** ne couvre pas toute la zone du tableau). Puisqu'il y a deux zones relatives à la Règle **Règle 5** dans le dessin de l'Output, il faut donc faire apparaître deux zones partielles. Il vient donc :



L'avantage d'avoir procédé de la sorte est qu'on voit naturellement deux zones correspondant à la Règle **Règle 5**. Si on applique, sur ce schéma, la Règle **Règle 3**, on voit donc apparaître deux lignes de démarcation. Soit :



Il faut ensuite étiqueter les lignes de démarcation avec des variables (Règle **Règle 4**)⁸. La grande question à laquelle nous devons répondre est la suivante : de quelle côté d'une ligne de démarcation mettre la variable ? Soit à l'intérieur des zones d'intérêt (i.e., $[0, i]$ pour la partie gauche et $[j, N - 1]$ pour la partie droite), soit à l'extérieur des zones d'intérêt (i.e., $[0, i - 1]$ pour la partie gauche et $[j + 1, N - 1]$ pour la partie droite). Arbitrairement, nous choisissons à l'extérieur des zones d'intérêt. Il vient :

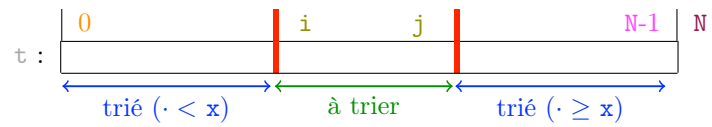


La Règle **Règle 5** ayant déjà été naturellement appliquée sur le schéma (par principe de la technique appliquée ici), il ne reste plus que la Règle **Règle 6**, soit identifier la zone devant encore être traitée dans les itérations suivantes. On obtient, dès lors, l'Invariant Graphique suivant :

Suite de l'Exercice

Maintenant, vous pouvez

8. Attention, deux lignes de démarcation == deux variables



- voir la construction de l'Invariant Graphique par **application stricte des règles** ;
- rédiger la **ZONE 1** du code.

5.6 Construction du Code : ZONE 1

Une fois l'Invariant Graphique établi, il faut s'appuyer dessus pour la construction du code, en particulier pour la ZONE 1, le Critère d'Arrêt et la Fonction de Terminaison, la ZONE 2 et, enfin, la ZONE 3. Cette section s'intéresse à la ZONE 1

Si vous voyez de quoi on parle, rendez-vous à la Section

5.6.3

Si vous ne voyez pas le lien entre Invariant Graphique et construction du code, reportez-vous à la Section

5.6.1

Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice

5.6.2

5.6.1 Rappel sur l'Invariant Graphique et la Construction du Code

L'Invariant de Boucle (de manière générale) et l'Invariant Graphique (de manière particulière) sont au coeur de la construction des programmes. Construire un programme en s'appuyant sur l'Invariant de Boucle correspond à ce qu'on appelle l'*approche constructive*.

En partant de la **sémantique de l'Invariant de Boucle**, on peut définir une stratégie⁹ de résolution du problème. Cette stratégie permet d'identifier quatre points particuliers : trois ZONES (voir la Fig. 7) et le Critère d'Arrêt.

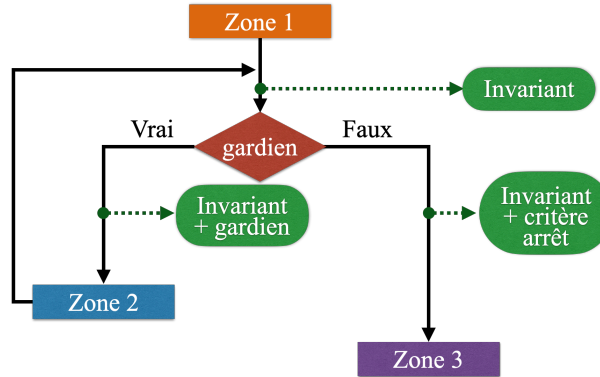


FIGURE 7 – Relation entre le Invariant Graphique et les différentes zones du programme.

Ces différents éléments permettent de construire le code :

ZONE 1 : correspond aux instructions juste avant la boucle. Par définition de l'Invariant de Boucle, lors de la toute première évaluation du Gardien de Boucle (avant d'entrer la toute première fois dans le Corps de la Boucle), l'Invariant de Boucle doit être vrai. L'Invariant de Boucle indique donc les variables dont on a besoin mais aussi leurs valeurs d'initialisation. Cette *situation initiale* est obtenue en déplaçant la ligne de démarcation de l'Invariant Graphique à sa valeur de base.

ZONE 2 : correspond au Corps de la Boucle. Cela signifie donc qu'on vient d'évaluer le Gardien de Boucle et que celui-ci est vrai. Dès lors, on se retrouve dans une situation où l'Invariant de Boucle est vrai mais aussi le Gardien de Boucle. C'est donc une situation particulière qui doit permettre de déduire une suite d'instructions permettant de faire avancer le problème (i.e., on se rapproche de la solution). Après la dernière instruction du Corps de la Boucle, le Gardien de Boucle va de nouveau être évalué. Dès lors, par définition, l'Invariant de Boucle doit être vrai à cet endroit. L'objectif de la ZONE 2 est donc, partant de l'Invariant de Boucle et du fait que le Gardien de Boucle est vrai, trouver un ensemble d'instructions qui restaurent l'Invariant de Boucle juste avant la prochaine évaluation du Gardien de Boucle. Ces instructions sont naturellement déduites de l'Invariant Graphique.

ZONE 3 : correspond aux instructions après la boucle. Le Gardien de Boucle vient d'être évalué à faux (on a donc atteint le Critère d'Arrêt) et, par définition, l'Invariant de Boucle est vrai. Sur base de ces deux propriétés, il faut trouver l'ensemble des instructions qui permet de clôturer le problème, i.e., atteindre l'**Output** du problème ou du SP. Cette situation particulière est obtenue en étirant la ligne de démarcation à sa valeur finale.

Critère d'Arrêt : en plaçant la ligne de démarcation à sa valeur finale, on peut observer la valeur particulière que va prendre la variable utilisée pour étiqueter la ligne de démarcation (voir **Règle 4** de la construction de l'Invariant Graphique). Cette valeur particulière correspond au Critère d'Arrêt. Pour obtenir le Gardien de Boucle, il suffit donc de nier le Critère d'Arrêt¹⁰.

9. Il faut comprendre le terme "stratégie" comme étant "l'élaboration d'un plan". On n'est pas à Kho Lanta ou aux autres télérealités dans lesquelles les candidats vicieux ou manipulateurs sont qualifiés de "stratèges".

10. Pour rappel, le Critère d'Arrêt est **la négation** du Gardien de Boucle

N'hésitez pas à vous appuyer sur le **GLIDE**. Il permet, explicitement, de manipuler graphiquement l'Invariant de Boucle afin de faire ressortir les différentes zones.

Exemple

Pour illustrer le fonctionnement de l'approche constructive basée sur l'Invariant Graphique, nous allons nous appuyer sur un exemple. Le problème consiste à calculer le produit de tous les entiers entre des bornes fournies, i.e., **a** et **b** (avec $b > a$ – les deux valeurs sont fournies au clavier), et à afficher le produit à l'écran. La **définition** du problème est la suivante :

Input : **a** et **b** (lus au clavier), les bornes entre lesquelles les entiers doivent être multipliés (avec $b > a$).

Output : le produit cumulatif de tous les entiers dans $[a, b]$ est affiché à l'écran.

Objets Utilisés :

a, la borne inférieure

$a \in \mathbb{Z}$

`int a;`

b, la borne supérieure

$b \in \mathbb{Z}$

`int b;`

La Fig. 8a montre l'Invariant Graphique correspondant (les détails sur la construction de l'Invariant Graphique sont donnés dans le **rappel**).

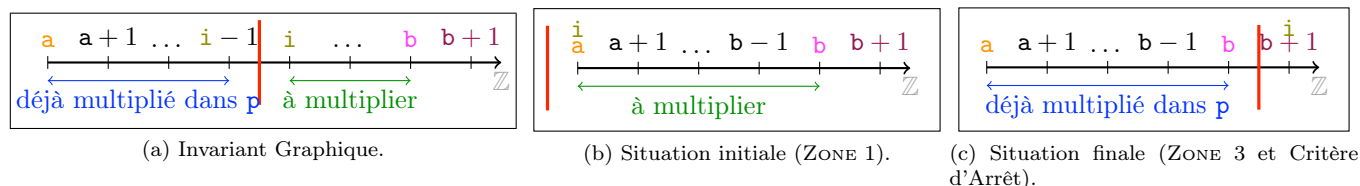


FIGURE 8 – Invariant Graphique et situations particulières pour le calcul du produit d'entiers entre **a** et **b**.

Une fois défini, l'Invariant Graphique doit être utilisé pour déduire les instructions du code, zone après zone. Allons-y...

ZONE 1 La définition du problème nous indique déjà deux variables : **a** et **b**. En outre, l'Invariant Graphique introduit deux autres variables : **p** (pour le produit cumulatif) et **i** (pour lister les entiers entre **a** et **b**). Les valeurs initiales de **a** et **b** sont données par la définition : lecture au clavier. Cependant, pour obtenir les valeurs initiales des variables **i** et **p**, il faut déplacer la ligne de démarcation vers la gauche afin d'obtenir la situation décrite par la Fig. 8b. Il s'agit de la situation initiale, avant la toute première évaluation du Gardien de Boucle. La variable **i** est toujours l'entier le plus proche à droite de la ligne de démarcation. Sa valeur initiale est donc, d'après la Fig. 8b, **a**. On remarque aussi sur la Fig. 8b que la zone verte est vide (c'est normal, aucun tour de boucle n'a encore eu lieu) : **p** représente donc le produit vide (i.e., le produit sans opérandes) et doit être initialisé à 1 (1 est le neutre de la multiplication). On obtient dès lors le code suivant :

```

1 int a, b, p, i;
2
3 scanf ("%d", &a);
4 scanf ("%d", &b);
5
6 i = a;
7 p = 1;

```

Critère d'Arrêt On peut déduire le Critère d'Arrêt de la situation finale illustrée à la Fig. 8c. Cette situation particulière est obtenue de l'Invariant Graphique en déplaçant la ligne de démarcation vers la droite, jusqu'à ce que la zone verte s'étende entièrement entre a et b (i.e., l'objectif du programme est atteint). On peut voir que les itérations doivent s'arrêter quand $i = b + 1$. Dès lors, le Gardien de Boucle correspondant est $i \neq b + 1$ mais on préférera la version $i \leq b$ qui vient avec l'avantage de naturellement représenter la position relative de i et b sur la droite des entiers. On obtient dès lors le code suivant :

```
1 while(i <= b){
2     //à compléter (ZONE 2)
3 }//fin boucle
```

ZONE 2 L'Invariant Graphique permet, bien entendu, de déduire les instructions du Corps de la Boucle. Pour cela, il faut s'appuyer sur la Fig. 8a et identifier les instructions qui permettraient à la zone verte de progresser vers la droite. La valeur entière qui doit être considérée dans la situation courante est l'entier i (i.e., première valeur entière de la zone bleue, juste après la ligne de démarcation). Afin de faire grandir la zone verte, i doit être multiplié par p (i.e., $p *= i$), car p contient le produit cumulatif calculé sur la zone verte. Lors de la prochaine itération, l'entier qui devra être considéré est $i + 1$. Dès lors, l'entier i doit être incrémenté d'une unité (i.e., $i++$). Après cette dernière instruction, on constate que l'Invariant Graphique est restauré et que le Gardien de Boucle doit être de nouveau évalué. Le code du Corps de la Boucle est donc

```
1 p *= i;
2 i++;
```

ZONE 3 On peut déduire, de la situation finale illustrée à la Fig. 8c, l'ensemble des instructions à exécuter après la boucle. Cette situation particulière est obtenue de l'Invariant Graphique en déplaçant la ligne de démarcation vers la droite, jusqu'à ce que la zone verte s'étende entièrement entre a et b (i.e., l'objectif du programme est atteint). Le Critère d'Arrêt est atteint et, comme on vient d'évaluer le Gardien de Boucle, l'Invariant de Boucle est vrai aussi. Mais nous sommes dans une situation particulière où la zone verte recouvre tout l'intervalle entre a et b , la zone bleue étant inexistante. Par conséquent, d'après l'Invariant Graphique, p contient le produit de tous les entiers entre a et b . La définition du problème nous indique que le résultat doit être affiché à l'écran. À la sortie de la boucle, il faut donc afficher à l'écran le contenu de la variable p . Soit

```
1 printf("%d", p);
```

Au final, le code complet (quand on met tous les bouts ensemble) est le suivant :

```
1 #include <stdio.h>
2
3 int main(){
4     //ZONE 1
5     int a, b, p, i;
6
7     scanf("%d", &a);
8     scanf("%d", &b);
9
10    i = a;
11    p = 1;
12
13    while(i <= b){
14        //ZONE 2
15        p *= i;
16        i++;
17    }//fin boucle
18
19    //ZONE 3
20    printf("%d", p);
21 }//fin programme
```

Alerte : Alerte

L'Invariant Graphique et la construction du code basée sur l'Invariant de Boucle (i.e., l'approche constructive) sont au centre de la philosophie des cours INFO046 et INFO0947.

Être à l'aise avec l'approche et bien l'appliquer est primordial. En cas de doute, d'incompréhension, de problème(s), n'hésitez pas à interagir avec l'équipe pédagogique sur [eCampus](#). Ne laissez surtout pas traîner la moindre incompréhension. Cela risquerait de faire un effet boule de neige et les difficultés deviendraient, alors, difficilement surmontables.

Suite de l'Exercice

À vous de jouer ! Construisez :

- la **ZONE 1**
- la **ZONE 2**
- la **ZONE 3**
- le **Critère d'Arrêt**

Si vous séchez, reportez-vous à l'indice pour

- la **ZONE 1**
- la **ZONE 2**
- la **ZONE 3**
- le **Critère d'Arrêt**

5.6.2 Indice

La construction de la ZONE 1 se fait exclusivement en manipulant, graphiquement, l'Invariant Graphique. Procédez donc de la sorte :

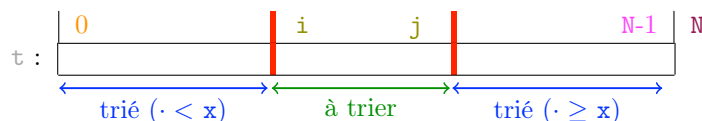
- repartez de l'Invariant Graphique du SP_2 ;
- identifiez les variables présentes dans votre Invariant Graphique ;
- aidez-vous du **GLIDE** pour faciliter la manipulation graphique (vous pouvez explicitement déplacer les lignes de démarcation, l'Invariant Graphique s'adapte alors automatiquement) ; Cette manipulation vous donnera certaines valeurs d'initialisation ;

Suite de l'Exercice

À vous ! Construisez la ZONE 1 pour les différents SP et passez à la Sec. **5.6.3**.

5.6.3 Mise en Commun de la ZONE 1

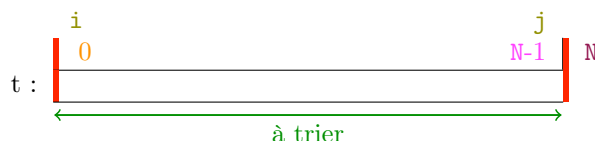
Pour rappel, le SP₂ s'intéresse au tri du tableau **t** en fonction de **x**. L'Invariant Graphique obtenu est le suivant :



En regardant l'Invariant Graphique, nous pouvons déjà identifier les variables :

- t** : il s'agit du tableau à trier. Cette variable a déjà été identifiée lors de la **définition du problème**. Les valeurs contenues dans le tableau **t** sont de type **int**. Le tableau est déjà rempli (cfr. **énoncé**).
- N** : il s'agit de la taille du tableau **t**. Cette constante a déjà été identifiée lors de la **définition du problème**. Elle est de type **unsigned int** et sa valeur est fixée pour nous (cfr. **énoncé**).
- x** : il s'agit de la valeur pivot par rapport à laquelle nous allons devoir trier le tableau. Cette variable a déjà été identifiée lors de la **définition du problème** et est de type **int**.
- i** : c'est une des variables d'itération qui permet de délimiter la zone $\cdot < x$ dans le tableau **t**. C'est donc un indice pour notre tableau. Les valeurs des indices appartenant à l'intervalle $[0, N - 1]$ et **N** étant de type **unsigned int**, il est donc souhaitable que **i** soit aussi de type **unsigned int**.
- j** : c'est une des variables d'itération qui permet de délimiter la zone $\cdot \geq x$ dans le tableau **t**. C'est donc un indice pour notre tableau. Les valeurs des indices appartenant à l'intervalle $[0, N - 1]$ et **N** étant de type **unsigned int**, il est donc souhaitable que **j** soit aussi de type **unsigned int**.

Il faut maintenant voir comment initialiser les deux variables d'itération (i.e., **i** et **j**). Pour cela, il suffit de manipuler graphiquement l'Invariant Graphique en faisant glisser les deux lignes de démarcation vers les bornes du tableau (i.e., vers la gauche pour **i** et la droite pour **j**). Lors de ce déplacement, on accompagne la ligne de démarcation avec la variable d'itération. Cette manipulation graphique nous donne le schéma suivant :



On voit alors clairement que les parties bleues ont disparu et que la partie verte recouvre tout le tableau. Cela signifie qu'à la première évaluation du Gardien de Boucle, aucun tri n'a encore été effectué. On voit clairement aussi que la valeur de la variable **i** est 0 et N-1 pour **j**.

La variable **x** est elle initialisée grâce au SP₁ (i.e., lecture au clavier).

On a donc le code suivant (les déclarations et initialisations de **t** et **N** se trouvent dans l'**énoncé**) :

Extrait de code 6 – ZONE 1

```

1 unsigned int i, j;
2
3 scanf("%u", &x);
4
5 i = 0;
6 j = N-1;
```

Suite de l'Exercice

On peut maintenant continuer l'écriture du code, en particulier le **Critère d'Arrêt** et la **Fonction de Terminaison**.

5.7 Critères d'Arrêt et Fonctions de Terminaison

Une fois l'Invariant Graphique établi, il faut s'appuyer dessus pour la construction du code, en particulier pour la ZONE 1, le Critère d'Arrêt et la Fonction de Terminaison, la ZONE 2 et, enfin, la ZONE 3. Cette section s'intéresse à la ZONE 1

Si vous voyez de quoi on parle, rendez-vous à la Section

5.7.3

Si vous ne voyez pas le lien entre Invariant Graphique et construction du code, reportez-vous à la Section

5.6.1

Si le concept de Fonction de Terminaison ne vous parle pas, rendez-vous à la Section

5.7.1

Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice

5.7.2

5.7.1 Rappel sur l'Invariant Graphique et la Fonction de Terminaison

Une Fonction de Terminaison permet de déterminer, formellement¹¹, qu'une boucle se termine (i.e., le Critère d'Arrêt est atteint).

Une *Fonction de Terminaison* est une fonction entière (i.e., les résultats $\in \mathbb{Z}$) portant sur les variables du programmes. La fonction doit avoir une valeur strictement positive avant toute exécution du Corps de la Boucle. La valeur de la fonction décroît strictement à chaque exécution du Corps de la Boucle.

En général, la Fonction de Terminaison décrit de manière formelle (i.e., mathématique) le nombre de tours de boucle (c'est donc forcément une valeur entière – on ne peut pas faire un demi tour de boucle) restant avant d'atteindre le Critère d'Arrêt. Ce nombre doit forcément diminuer après chaque itération (pas d'itération gratuite ou inutile).

Alerte : Ce que la Fonction de Terminaison n'est pas

De manière générale, une Fonction de Terminaison

- n'est pas le Gardien de Boucle ;
- n'est pas le Critère d'Arrêt ;
- n'est pas l'Invariant de Boucle ;
- n'a pas forcément une valeur négative ou nulle quand la boucle se termine (ce n'est pas demandé par les propriétés que doit respecter une Fonction de Terminaison). Une fois que la boucle se termine, la valeur de la Fonction de Terminaison n'a plus aucune importance (puisque l'objectif est atteint : la boucle est terminée).

Règles

Une fois que vous avez déterminé la Fonction de Terminaison, vous pouvez vérifier qu'elle suit les deux règles suivantes :

- Règle 1 : la Fonction de Terminaison est bien une fonction entière. Cela signifie que le résultat de la fonction se trouve dans l'ensemble des entiers. Plus formellement, $f : \{\text{valeurs des variables}\} \rightarrow \mathbb{Z}$, où f est votre fonction de terminaison. Votre Fonction de Terminaison ne peut donc pas avoir la forme $f = x > 0$ puisque, dans ce cas, le résultat est booléen ;
- Règle 2 : la Fonction de Terminaison décroît strictement à chaque itération. Dans le cas contraire, il serait impossible de déterminer avec certitude que la boucle se terminera un jour (i.e., qu'elle atteindra le Critère d'Arrêt).

Déterminer une Fonction de Terminaison

Il existe deux techniques qui permettent de déterminer la Fonction de Terminaison d'une boucle.

Technique 1 : Invariant Graphique Il s'agit ici de dériver graphiquement, à partir de l'Invariant Graphique, la Fonction de Terminaison. Il suffit, généralement, de déterminer la taille de la zone "encore à ..." de l'Invariant Graphique (cfr. la Règle 6 de la construction d'un Invariant Graphique).

Prenons l'exemple du problème consistant à calculer le produit de tous les entiers entre des bornes fournies, i.e., **a** et **b** (avec $b > a$ – les deux valeurs sont fournies au clavier), et à afficher le produit à l'écran. L'Invariant Graphique correspondant est donné par la Fig. 9¹².

Ici, déterminer la Fonction de Terminaison revient à déterminer la taille de la zone bleue. Pour ce faire :

11. Formellement == mathématiquement

12. Les détails sur la création de cet Invariant Graphique sont donnés dans le **rappel**.

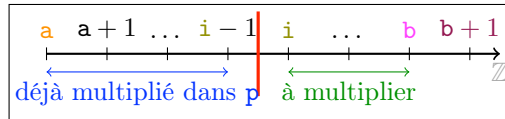


FIGURE 9 – Exemple d’Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

1. il faut déterminer la taille complète de l’intervalle sur lequel on travaille, soit la taille de la zone verte + la taille de la zone bleue. Dans notre cas, cela donne $b - a + 1$.
 2. il faut ensuite déterminer la taille de la zone verte (ce qu’on a déjà accompli lors des itérations précédentes). Dans notre cas : $i - a$.
 3. au final, pour obtenir la Fonction de Terminaison, il suffit de soustraire la taille de la zone verte à la taille totale. Soit $b - a + 1 - (i - a)$. On peut simplifier cette expression et on obtient alors $b - i + 1$.
- Dans cet exemple, la Fonction de Terminaison est : $f : b - i + 1$.

Technique 2 : Gardien de Boucle Il s’agit, ici, de déterminer la Fonction de Terminaison en manipulant le Gardien de Boucle. Cette technique est moins pratique que celle basée sur l’Invariant Graphique, en particulier dans les situations où le Gardien de Boucle correspond à une expression complexe (i.e., des expressions connectées par des opérateurs booléens).

Si on repart sur le même exemple que pour la première technique, le Gardien de Boucle est le suivant : $i \leq b$ ¹³.

Il s’agit de procéder de la façon suivante :

1. identifier dans le Gardien de Boucle la variable d’itération (ici, i) et la borne supérieure (ici, b).
2. faire passer la variable d’itération du côté de la borne supérieure dans l’expression. Soit $0 \leq b - i$.
3. si l’opérateur de comparaison n’est pas $<$, il faut le transformer (de façon à rester strictement décroissant). Dans notre cas, il suffit de rajouter 1 dans l’opérande de droite. Soit : $0 < b - i + 1$.

Dans cet exemple, la Fonction de Terminaison est donc : $f : b - i + 1$.

Suite de l’Exercice

À vous! Rédigez vos Critères d’Arrêt et Fonction de Terminaison et passez à la Sec. 5.7.3.

Si vous séchez, reportez-vous à l’indice à la Sec. 5.7.2

13. Pour les détails, voir le **rappel** sur la construction basée sur l’Invariant Graphique

5.7.2 Indice

La découverte du Critère d'Arrêt se fait exclusivement en manipulant, graphiquement, l'Invariant Graphique. Procédez donc de la sorte :

- reprenez le Invariant Graphique du SP_2 ;
- assurez-vous que le Invariant Graphique contient bien une ou plusieurs ligne(s) de démarcation et une variable pour étiqueter chacune des lignes ;
- aidez-vous du **GLIDE** pour faciliter la manipulation graphique (vous pouvez explicitement déplacer la/les lignes de démarcation, l'Invariant Graphique s'adapte alors automatiquement) ; Cette manipulation de la/les lignes de démarcation vers leurs positions finales vous donnera des informations sur le Critère d'Arrêt ;

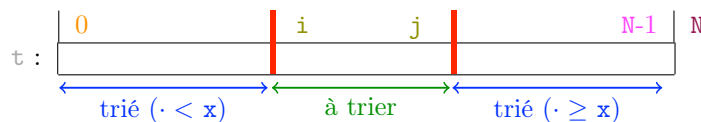
Pour la Fonction de Terminaison, il s'agit (simplement) d'évaluer la taille de la zone “verte” d'un Invariant Graphique (cfr. Règle 6).

Suite de l'Exercice

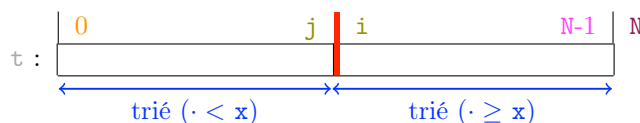
À vous ! Rédigez les Critères d'Arrêt et Fonctions de Terminaison pour le SP_2 et passez à la Sec. 5.7.3.

5.7.3 Mise en Commun du Critère d'Arrêt et de la Fonction de Terminaison

La Fig. ?? donne l'Invariant Graphique pour le SP_2 . Pour rappel, le SP_2 s'intéresse au tri du tableau t en fonction de x . L'Invariant Graphique est le suivant :



Commençons par le Critère d'Arrêt. Pour le trouver, il faut faire glisser les lignes de démarcation de sorte que la zone verte ("à trier") ait disparu et que la zone bleue (" $\cdot < x$ " et " $\cdot \geq x$ ") recouvre tout le dessin. L'objectif est donc de retrouver la représentation graphique de l'Output. On obtient :



Nous voyons donc que le tableau sera trié en fonction de x quand $i > j$. Notre Critère d'Arrêt est donc : $i > j$. Puisqu'on sait qu'il existe un lien entre Critère d'Arrêt et Gardien de Boucle (i.e., le Critère d'Arrêt est la négation du Gardien de Boucle), on obtient le Gardien de Boucle suivant : $!(i > j)$. Soit, de manière plus naturelle : $i \leq j$.

On dérive alors le code suivant :

```
1 while(i <= j){
2   //à compléter (ZONE 2)
3 }//fin while - etage
```

Passons à la Fonction de Terminaison. Il suffit, ici, d'estimer la taille de la zone verte ("à trier"). La zone bleue de gauche (" $\cdot < x$ ") s'étend de 0 à $i - 1$. Elle a donc une taille de i . La zone entre 0 et j (soit la zone bleue de gauche + la zone verte) a une taille de $j + 1$. Il suffit alors de faire la différence entre ces deux tailles pour obtenir la Fonction de Terminaison. Soit $j - i + 1$.

La Fonction de Terminaison est : $f : j - i + 1$.

Suite de l'Exercice

On peut maintenant continuer l'écriture du code, en particulier la **ZONE 2**.

5.8 Construction du Code : ZONE 2

Une fois l'Invariant Graphique établi, il faut s'appuyer dessus pour la construction du code, en particulier pour la ZONE 1, le Critère d'Arrêt et la Fonction de Terminaison, la ZONE 2 et, enfin, la ZONE 3. Cette section s'intéresse à la ZONE 2

Si vous voyez de quoi on parle, rendez-vous à la Section

[5.8.2](#)

Si vous ne voyez pas le lien entre Invariant Graphique et construction du code, reportez-vous à la Section

[5.6.1](#)

Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice

[5.8.1](#)

5.8.1 Indice

La construction de la ZONE 2 se fait exclusivement en manipulant, graphiquement, l'Invariant Graphique. Procédez donc de la sorte :

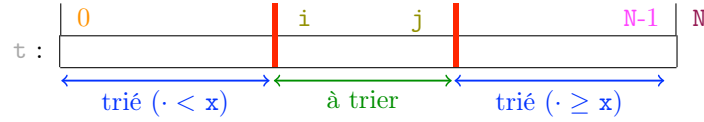
- repartez de l'Invariant Graphique du SP_2 ;
- repérez bien les zones bleues (déjà caculées lors des itérations précédentes) et les zones vertes (encore à calculer dans les itérations suivantes) ;
- l'idée principale est de faire avancer le problème, i.e., faire grandir la/les zone(s) bleue(s) et faire diminuer d'autant la zone verte ;
- ne pas oublier qu'après la dernière instruction du Corps de la Boucle, le Gardien de Boucle sera évalué. Par conséquent, la forme générale de l'Invariant Graphique doit être restaurée.

Suite de l'Exercice

À vous ! Construisez la ZONE 2 pour les différents SP et passez à la Sec. 5.8.2.

5.8.2 Mise en Commun de la ZONE 2

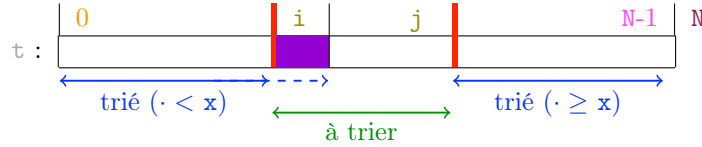
Pour rappel, le SP_2 s'intéresse au tri du tableau t en fonction de x . L'Invariant Graphique est le suivant :



Comme nous avons deux lignes de démarcation, il faut décider dans quel sens nous allons parcourir le tableau t . Soit du début vers la fin, en utilisant i comme variable de parcours. Soit de la fin vers le début, en utilisant j comme variable de parcours. Il n'y a pas, ici, une solution qui est meilleure que l'autre. Nous choisissons, arbitrairement, d'utiliser i comme variable de parcours.

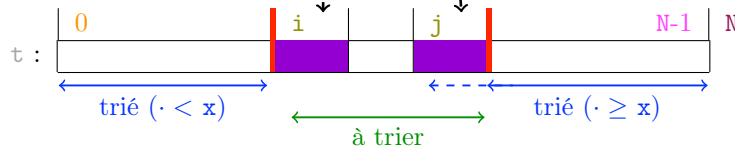
La ZONE 2 correspond au Corps de la Boucle. Par conséquent, l'Invariant Graphique est vrai et le Gardien de Boucle vient d'être évalué à vrai. Cela signifie qu'il reste encore au moins une case du tableau ($t[i]$) à trier en fonction de x . L'objectif de la ZONE 2 est de faire avancer le problème (i.e., trier t en fonction de x) en augmentant la taille de la zone bleue et en réduisant d'autant la zone verte. Or, l'Invariant Graphique du SP_2 contient deux zones bleues. Il y aura donc deux cas à considérer :

1. $t[i] < x$. Graphiquement, la situation est la suivante :



Dans ce cas, $t[i]$ est déjà bien placé (pour rappel, i est la variable d'itération et nous parcourons le tableau t du début à la fin) et il suffit de faire grandir la zone bleue " $\cdot < x$ " d'une position vers la droite (i.e., $i++$). Cela correspond à la ligne 5 dans l'Extrait de Code 7.

2. $t[i] \geq x$. Graphiquement, la situation est la suivante :



Dans ce cas, le contenu de $t[i]$ doit être transféré dans la zone bleue " $\cdot \geq x$ ". Il faut donc permuter le contenu de la case $t[i]$ et la case $t[j]$ ¹⁴ (ceci implique de faire intervenir une variable temporaire, tmp , pour permettre la permutation) et faire croître la zone bleue " $\cdot \geq x$ " d'une position vers la gauche (i.e., $j--$). On ne touche pas à l'indice i car, suite à la permutation, on ne sait pas ce que contient $t[i]$ (cette valeur sera traitée dans l'itération suivante). Cela correspond aux lignes 6 → 13 dans l'Extrait de Code 7.

La discrimination de ces deux cas, dans le code, se fait via une structure de contrôle conditionnelle (ligne 3 dans le code 7).

On obtient alors le code suivant :

Extrait de code 7 – ZONE 2

```

1 int tmp;
2 while(i <= j){
3     if(t[i] < x)
4         //Cas 1
5         i++;

```

14. La permutation est ici obligatoire. Nous ne savons pas ce que contient $t[j]$ et ceci ne nous importe pas. Puisqu'il y a permutation, l'ancienne valeur de $t[j]$ se retrouve dans $t[i]$ et sera traitée lors de la prochaine itération.

```
6  else{
7      //Cas 2
8      tmp = t[j];
9      t[j] = t[i];
10     t[i] = tmp;
11
12     j--;
13 }
14 }
```

Suite de l'Exercice

On peut maintenant continuer l'écriture du code, en particulier la **ZONE 3**.

5.9 Construction du Code : ZONE 3

Une fois l'Invariant Graphique établi, il faut s'appuyer dessus pour la construction du code, en particulier pour la ZONE 1, le Critère d'Arrêt et la Fonction de Terminaison, la ZONE 2 et, enfin, la ZONE 3. Cette section s'intéresse à la ZONE 3

Si vous voyez de quoi on parle, rendez-vous à la Section

5.9.2

Si vous ne voyez pas le lien entre Invariant Graphique et construction du code, reportez-vous à la Section

5.6.1

Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice

5.9.1

5.9.1 Indice

La construction de la ZONE 3 se fait exclusivement en manipulant, graphiquement, l'Invariant Graphique. Procédez donc de la sorte :

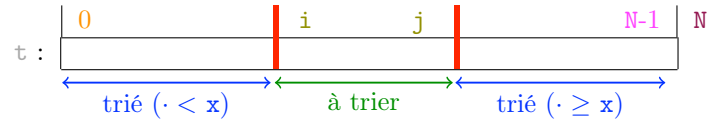
- repartez de l'Invariant Graphique du SP_2 ;
- assurez-vous que le Invariant Graphique contient bien une ou plusieurs lignes de démarcation et une ou plusieurs variables pour étiqueter cette/ces lignes ;
- aidez-vous du **GLIDE** pour faciliter la manipulation graphique (vous pouvez explicitement déplacer chaque ligne de démarcation, l'Invariant Graphique s'adapte alors automatiquement) ; Cette manipulation de la ligne de démarcation vers sa position finale vous donnera des informations sur la ZONE 3.

Suite de l'Exercice

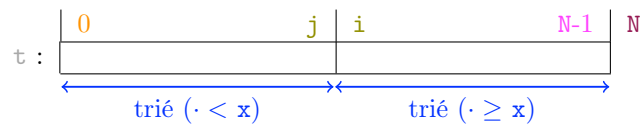
À vous ! Construisez la ZONE 3 pour les différents SP et passez à la Sec. 5.9.2.

5.9.2 Mise en Commun de la ZONE 3

La ZONE 3 correspond aux instructions après la boucle. Par conséquent, l'Invariant Graphique est vrai et le Gardien de Boucle vient d'être évalué à faux (le Critère d'Arrêt a été rencontré). Pour rappel, l'Invariant Graphique est le suivant :



Dans le cadre du SP_2 , cela signifie que le tableau t a été entièrement trié en fonction de x . Soit, après manipulation de l'Invariant Graphique pour faire apparaître la situation correspondant à la ZONE 3 :



Le problème général est donc terminé (nous avons atteint l'**Output** du problème). Le programme est dès lors terminé.

Suite de l'Exercice

On peut maintenant continuer l'**écriture du code**, en particulier il s'agit de mettre les différentes parties ensemble.

5.10 Code Final

Il s'agit de remettre le code des différents SP ensemble en suivant leur **enchaînement**.

```
1 #include <stdio.h>
2
3 int main(){
4     const unsigned int N = ...;
5     int t[N];
6
7     /*
8      * Code de remplissage du tableau t
9      * Ce code ne vous est pas donné.
10     */
11
12     int x;
13     unsigned int i = 0, j = N-1;
14     int tmp;
15
16     //début SP1
17     scanf("%d", &x);
18     //fin SP1
19
20     //début SP2
21     while(i <= j){
22         if(t[i]<x)
23             i++;
24         else{
25             tmp = t[j];
26             t[j] = t[i];
27             t[i] = tmp;
28
29             j--;
30         }
31     } //fin boucle
32     //fin SP2
33 } //fin programme
```