Université de Liège

INFO0946

Introduction à la Programmation

Un exercice dont vous êtes le Héros · l'Héroïne

Méthodologie de Développement et Construction de Programme

Benoit Donnet Simon Liénardy Tasnim Safadi

5 octobre 2020

Préambule

Exercices

Dans ce « TP dont vous êtes le héros », 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.

3.1 Rappels sur la Méthodologie de Développement

Le schéma méthodologique, que nous allons suivre durant tout le cours, est illustré à la Fig. 1.

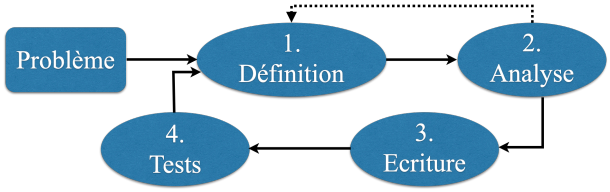


FIGURE 1 – Schéma de la méthodologie de développement.

Partant du problème initial, ce schéma comporte quatre étapes :

1. *Définition du Problème.* Cette étape revient à définir précisement le problème en terme de donnée(s) en entrée (Input), de résultat(s) attendu(s) (Output) et d'Objet(s) Utilisé(s). Cette première étape permet donc de prendre connaissance des informations nécessaires à la résolution du problème et, surtout, de bien comprendre le problème.
2. *Analyse du Problème.* Cette étape revient à penser l'architecture générale du code en découpant le problème initial en *sous-problèmes* (SP), soit des ensembles d'actions à exécuter pour résoudre le problème général. L'objectif de l'analyse est aussi de voir comment les différents SP interagissent entre eux. On parle aussi d'*approche systémique*. A noter, comme illustré dans la Fig. 1, que chaque SP peut être défini.
3. *Ecriture du Code.* Il s'agit d'implémenter les différents SP, peu importe l'ordre. Si l'implémentation d'un SP nécessite la présence d'une boucle, il faudra au préalable définir un *Invariant Graphique* grâce auquel le code pourra être construit. La boucle doit aussi se terminer un jour. On peut le démontrer formellement (i.e., mathématiquement) grâce à la définition d'une *Fonction de Terminaison*.
4. *Tests.* Cette dernière étape permet de vérifier que l'implémentation résout bien le problème. Cela peut, naturellement, nécessiter de revenir à une étape précédente en cas d'erreur. La notion de test ne sera pas abordée dans ce cours mais bien au Q2, dans le cours INFO0030.

3.2 Enoncé

Un *sablier* est un instrument qui permet de mesurer un intervalle de temps correspondant à la durée d'écoulement d'une quantité calibrée de "sable", à l'intérieur d'un récipient transparent. Dans cet exercice, nous allons vous demander de dessiner, sur la sortie standard, un sablier en utilisant le caractère espace (que nous symbolisons dans ce GameCode par `'␣'`) et le caractère `'*'`. La Fig. 2 illustre un sablier. On remarque que le sablier est composé de deux triangles *isocèles* (i.e., deux côtés de même longueur, deux angles de même mesure et un axe de symmétrie). Le triangle supérieur est de sommet A et de base $[BC]$.

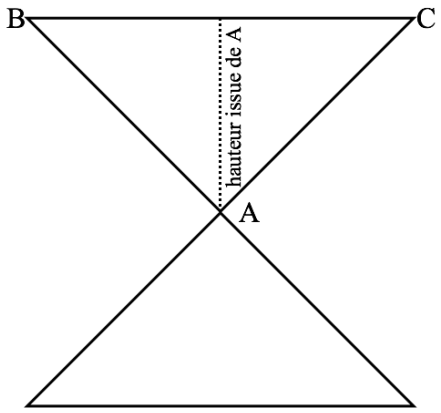


FIGURE 2 – Un sablier.

On vous demande, en suivant la méthodologie vue au cours, d'écrire un programme qui dessine un sablier tel que, pour chaque triangle, la hauteur issue de A soit égale à n. La valeur de n est introduite au clavier par l'utilisateur. Un exemple d'exécution pour $n = 6$ est donné ici :

```
**********
 ********
  ******
   ****
    **
     *
     *
    **
   ****
  ******
 ********
**********
```

3.2.1 Méthode de résolution

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

1. Idée générale de la solution (Sec. 3.3) ;
2. Définition du problème (Sec. 3.4) ;
3. Analyse du problème (Sec. 3.5) ;
4. Invariants Graphiques (Sec. 3.6) ;
5. Construction de la ZONE 1 (Sec. 3.7) ;
6. Construction des Critères d'Arrêt et Fonctions de Terminaison (Sec. 3.8)
7. Construction de la ZONE 2 (Sec. 3.9) ;

8. Construction de la Zone 3 (Sec. 3.10) ;
9. Code final (Sec. 3.11) ;

Si vous ne vous sentez pas à l'aise avec la méthodologie de développement, reportez-vous au rappel.

Alerte : Exercice difficile !

Cet exercice est probablement l'un des plus compliqués et des plus longs que nous avons fait jusqu'à présent. Il demande donc d'être particulièrement concentré pour être mené à son terme.

Nous insistons sur le fait qu'il vaut mieux prendre le temps de le faire seul et de poser des questions sur eCampus plutôt que de lire directement la solution : cela ne vous servirait strictement **à rien**. Rappelez-vous qu'à l'interrogation de mi-quadrimestre et à l'examen, vous ne disposerez d'aucune aide !

3.3 Raisonnement Général

Avant de poursuivre avec la définition du problème, il faut d'abord réfléchir aux concepts importants du problème.

Si vous voyez de quoi on parle, rendez-vous à la Section 3.3.2
Si vous séchez sur les concepts à faire apparaitre, voyez l'indice 3.3.1

3.3.1 Indice

Pour trouver les concepts importants, le plus simple est de repartir de l'énoncé et d'essayer de les inférer en se posant les questions suivantes :

- de quoi est composé un sablier ?
- quels sont les caractères qui composent le sablier ?
- quelles sont les interactions et relations entre les différents caractères composant le sablier et les données du problème ?

Pour répondre à ces questions, il peut être intéressant de s'appuyer sur un dessin (le plus général possible) et de jouer avec des couleurs pour mettre en évidence les éléments clés.

Suite de l'Exercice

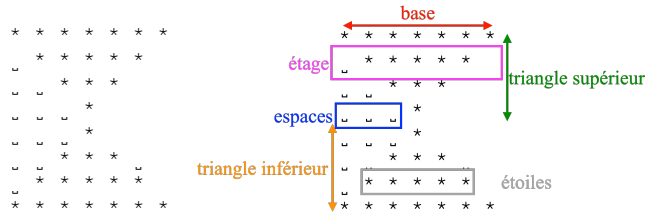
À vous ! Trouvez les concepts clés et leurs relations et rendez-vous à la Sec. 3.3.2.

3.3.2 Mise en Commun – Les Concepts et leurs Relations

Les Concepts

L'objectif du problème est de dessiner un sablier, constitué de deux triangles isocèles, sur la sortie standard à l'aide des caractères `'*'` et `'␣'`. La hauteur de chaque triangle, n, est donnée au clavier.

La première étape est donc de dessiner un sablier. Prenons dès lors un sablier pour lequel la hauteur de chaque triangle est $n = 4$ (Fig. 3a).



(a) Un sablier ($n = 4$)

(b) Le sablier et les concepts clés

FIGURE 3 – Identification des concepts clés.

Partant de la Fig. 3a, on peut identifier les différents concepts (illustrés à la Fig. 3b) :

triangle supérieur : il s'agit du triangle isocèle correspondant à la partie supérieure du sablier. Ce sera le premier triangle à dessiner sur la sortie standard. La hauteur du triangle est donnée, c'est n.

triangle inférieur : il s'agit du triangle isocèle correspondant à la partie inférieure du sablier. C'est exactement le même que le triangle supérieur, avec un effet miroir en plus.

base : il s'agit du côté sur lequel "repose" chacun des triangles. On ne connait pas sa valeur. Il va falloir l'inférer.

étage : il s'agit d'une ligne, composée de `'*'` et `'␣'`. Le nombre d'étages d'un triangle correspond à sa hauteur, n. C'est donc une information connue.

espaces : ce sont les caractères qui permettent de décaler les `'*'` sur un étage donné. On ne connait pas le nombre de `'␣'` par étage. Il va falloir l'inférer.

étoiles : ce sont les caractères qui permettent de dessiner les triangles. On voit sur la figure qu'à chaque étage, le nombre de `'*'` change. On ne connait pas le nombre exact de `'*'` par étage. Il va falloir l'inférer.

Sur base de ces concepts, nous devons encore trouver deux relations :

1. Combien de `'*'` par étage ?
2. Combient de `'␣'` par étage ?

Regardons cela tout de suite.

Les Caractères `'*'`

Commençons par nous interroger sur le nombre de `'*'` par étage. Comme illustré dans la Fig. 4 (triangle supérieur mais le raisonnement est le même pour le triangle inférieur), chaque étage permet, en fait, de construire un nouveau triangle, plus petit que le précédent. Dès lors, chaque étage correspond à une base pour un nouveau triangle (plus petit que celui de l'étage supérieur). Ici, nous avons choisi de numéroter les étages "à l'envers", i.e., la pointe du triangle est l'étage 1, la base du du plus grand triangle est l'étage 5 dans cet exemple. La raison derrière ce choix est assez simple : l'étage 5 correspond, en fait, à un triangle d'une hauteur $n = 5$. Par conséquent, l'étage 4 correspond à un triangle d'une hauteur $n = 4$. Etc.

En partant de la Fig. 4, essayons de trouver le nombre de `'*'` par étage.

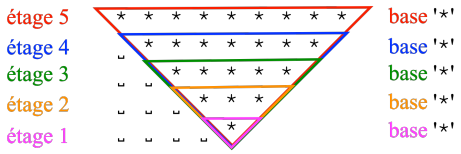


FIGURE 4 – Raisonnement pour déterminer le nombre de `'*'` par étage ($n = 5$ – triangle supérieur).

$n = 5$: Un simple calcul nous indique que, sur l'étage 5, il y a 9 `'*'`. Il s'agit de faire, maintenant, le lien entre l'étage (5) et le nombre de `'*'`. On peut trouver la relation suivante : $9'*' = (2 \times 5) - 1$.

$n = 4$: A nouveau, un simple calcul montre que, sur l'étage 4, il y a 7 `'*'`. Si on fait le lien avec l'étage, on obtient la relation : $7'*' = (2 \times 4) - 1$.

$n = 3$: On applique le même raisonnement et on trouve que sur l'étage 3, il y a 5 `'*'`. Soit la relation : $5'*' = (2 \times 3) - 1$.

$n = 2$: En appliquant toujours ce raisonnement, on trouve que sur l'étage 2, il y a 3 `'*'`. Soit la relation : $3'*' = (2 \times 2) - 1$.

$n = 1$: Enfin, pour l'étage 1, nous avons 1 `'*'`. Soit la relation : $1'*' = (2 \times 1) - 1$.

Il s'agit, maintenant, d'essayer de généraliser ces relations en faisant intervenir des variables (qui deviendront, forcément, des variables de notre code).

Sur base du raisonnement ci-dessus, on voit clairement que le nombre d'`'*'`, sur un étage donné, c'est deux fois cet étage moins 1. Dit autrement :

$$\texttt{nb\_etoiles} = (2 \times \texttt{etage}) - 1$$

`nb_etoiles` et `etage` seront des variables du code. L'équation ci-dessus indique, simplement, comment initaliser `nb_etoiles` en fonction de l'`etage` sur lequel on se situe.

Les Caractères `'␣'`

Passons maintenant au nombre de `'␣'` par étage et appuyons nous sur un raisonnement similaire que pour les `'*'`. Les étages sont numérotés de la même façon. Il suffit donc de compter, pour chaque étage, le nombre de `'␣'`, comme illustré à la Fig. 5.

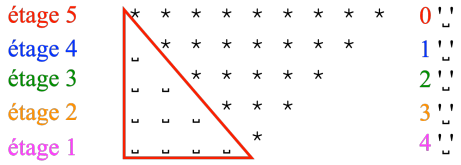


FIGURE 5 – Raisonnement pour déterminer le nombre de `'␣'` par étage ($n = 5$ – triangle supérieur).

Par exemple, pour l'étage 5, il n'y a aucun `'␣'` (0). Par contre, pour l'étage 4, il y a 1 espace. On peut donc envisager que le nombre de `'␣'` est donné par la hauteur du triangle de notre sablier (i.e., n) à laquelle on retranche l'étage sur lequel on se trouve. Soit :

$$\texttt{nb\_espaces} = \texttt{n} - \texttt{etage}$$

`nb_espaces`, `n` et `etage` seront des variables de notre code. L'équation ci-dessus indique, simplement, comment initaliser `nb_espace` en fonction de la hauteur désirée de chaque triangle constituant le sablier (`n`) et l'`etage` sur lequel on se situe.

On peut maintenant passer à la définition du problème. Voir Sec. 3.4.

3.4 Definition du Problème

La première étape de la méthodologie de développement est la *définition du problème*.
Si vous voyez de quoi on parle, rendez-vous à la Section 3.4.3
Si la notion de définition d'un problème est floue pour vous, voyez la Section 3.4.1
Si vous ne voyez pas comment faire, reportez-vous à l'indice 3.4.2

3.4.1 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

$n \in \mathbb{N}$ [2]

```
unsigned int n;
```

c, le caractère à écrire sur la sortie standard.

```
char c;
```

On en tire, naturellement, le canevas suivant :

```c
#include <stdio.h>

int main(){
  unsigned int n;
  char c;

  //déclaration des variables supplémentaires

  //résolution des différents sous-problèmes
}//fin programme
```

2. 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. 3.4.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 3.4.2

3.4.2 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. 3.4.3.

3.4.3 Mise en Commun de la Définition du Problème

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

> On vous demande, en suivant la méthodologie vue au cours, d'écrire un programme qui dessine un sablier composé de `'*'` et `'␣'` tel que, pour chaque triangle, la hauteur issue A soit égale à n. La valeur de n est introduite au clavier par l'utilisateur.

On dispose donc bien en entrée d'une certaine valeur n (Input) avec, pour objectif, de dessiner un sablier formé de deux triangles composés des caractères `'*'` et `'␣'` (Output).

La définition du problème est donc

Input : n, la hauteur de chacun des deux triangles composants le sablier (lue au clavier)

Output : un sablier, fait de deux triangles, est dessiné sur la sortie standard à l'aide des caractères `'*'` et `'␣'`.

Objet Utilisé n, la hauteur de chaque triangle

$n \in \mathbb{N}$ [3]

```
unsigned int n;
```

On en tire naturellement le canevas suivant :

```
#include <stdio.h>

int main(){
  unsigned int n;

  //déclaration de variables supplémentaires

  //résolution des SPs

}//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 !

Nous pouvons maintenant passer à l'analyse et découpe en SP du problème. Voir Sec. 3.5.

3. Une valeur négative n'a pas de sens ici, on ne peut pas avoir une hauteur de -23. En outre, une valeur réelle (e.g., 2.4) n'a pas de sens dans le cadre de ce qui est demandé.

3.5 Analyse du Problème

La deuxième étape de la méthodologie de développement est l'*analyse du problème*.
Si vous voyez de quoi on parle, rendez-vous à la Section 3.5.3
Si vous ne savez pas ce qu'est l'analyse du problème, allez à la Section 3.5.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 3.5.2

3.5.1 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. *Linéraire* : $\text{SP}_i \rightarrow \text{SP}_j$. 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* : $\text{SP}_j \subset \text{SP}_i$. 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. 3.5.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 3.5.2

3.5.2 Indice

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

Ensuite, on peut s'appuyer sur un dessin qui met en exergue les éléments clés identifiés précédement. En particulier, certains concepts clés identifiés peuvent s'apparenter à un Sp.

Enfin, en donnant un nom pertinent à chaque Sp et en jouant avec des couleurs, l'enchainement des différents Sp devrait apparaître plus naturellement.

Attention, vous devez veiller à ce que votre découpe soit pertinente : ni trop haut niveau (e.g., Sp_1 : dessiner un sablier), 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., vous devez avoir atteint l'Output décrit dans la définition).

Suite de l'Exercice

À vous ! Analysez le problème et passez à la Sec. 3.5.3.

3.5.3 Mise en Commun de l'Analyse du Problème

Première Approche (à la grosse louche)

Dans un premier temps, il est très aisé de venir avec une découpe inefficace. Par exemple :

SP$_1$: lecture de `n` au clavier.

SP$_2$: le sablier est dessiné à l'écran.

Avec l'enchaînement suivant :

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

Dans l'absolu, ce n'est pas mauvais mais le SP$_2$ est beaucoup trop large (il représente, finalement, le problème principal car il correspond à l'output). On doit pouvoir encore le décomposer de manière à mieux appréhender le problème.

Deuxième Approche (décomposition du sablier)

Puisque le sablier est composé de deux triangles isocèles (le triangle supérieur et le triangle inférieur), on pourrait décomposer le SP$_2$ de la façon suivante (le SP$_1$ ne change pas) :

SP$_2$: dessin du triangle supérieur.

SP$_3$: dessin du triangle inférieur.

Avec l'enchaînement suivant :

$$\text{SP}_1 \rightarrow \text{SP}_2 \rightarrow \text{SP}_3$$

À nouveau, ce n'est pas mauvais. Mais, comme précédemment, il y a moyen de mieux subdiviser les SP$_2$ et SP$_3$ afin de bien appréhender le problème.

Troisième Approche (découpe d'un triangle)

Intéressons nous au SP$_2$. Après tout, le SP$_3$ concerne le triangle inférieur et les deux triangles sont extrêmement similaires.

Si on s'appuie sur le raisonnement général, on sait qu'un triangle est composé de différents étages. Il suffirait donc d'énumérer les différents étages et afficher chacun d'entre eux à l'écran. Soit :

SP$_2$: énumération des valeurs dans $[1\,;\,\texttt{n}]$ de façon à afficher les différents étages du triangle supérieur.

SP$_3$: affichage de l'étage `etage`.

On voit bien que le SP$_3$ est inclus dans le SP$_2$. Ce qui donne l'enchaînement suivant :

$$\text{SP}_1 \rightarrow (\text{SP}_3 \subset \text{SP}_2)$$

À nouveau, ce n'est pas mal du tout mais la notion d'étage reprend à la fois la gestion des `'␣'` et la gestion des `'*'`. Or, avec le raisonnement général, on est capable d'exprimer le nombre de `'␣'` en fonction de l'étage et de la hauteur du triangle (`n`). Il en va de même pour le nombre de `'*'`, en fonction de l'étage.

Quatrième Approche (découpe d'un étage)

Grâce au raisonnement général, on sait qu'un étage est composé de `'␣'` et de `'*'`. On peut donc voir les choses comme suit :

SP$_3$: affichage des caractères `'␣'` sur l'étage `etage`.

SP$_4$: affichage des caractères `'*'` sur l'étage `etage`.

Ce qui donne, comme enchaînement :

$$\text{SP}_1 \rightarrow [(\text{SP}_3 \rightarrow \text{SP}_4) \subset \text{SP}_2]$$

Ceci ne concerne que le triangle supérieur. Il faut intégrer, dans l'histoire, le triangle inférieur.

Découpe Finale (tous ensemble)

La Fig. 6 illustre les SP. On envisage, au total, cinq SP. La Fig. 6 en montre 4, auxquels il faut ajouter la lecture de `n` au clavier.

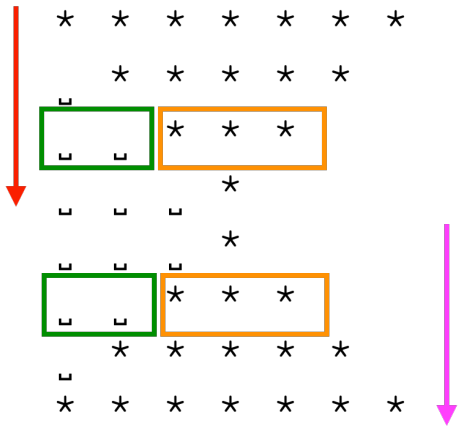


FIGURE 6 – Application des sous-problèmes à l'output demandé (exemple pour $n = 4$).

SP$_1$: lecture de `n` au clavier.

SP$_2$: énumération des valeurs dans [1 ; `n`] de façon à afficher les différents étages du triangle supérieur.

SP$_3$: affichage des caractères `'␣'` sur l'étage `etage`.

SP$_4$: affichage des caractères `'*'` sur l'étage `etage`.

SP$_5$: énumération des valeurs dans [1 ; `n`] de façon à afficher les différents étages du triangle inférieur.

Il est inutile de donner des SP pour l'affichage des `'␣'` et `'*'` dans le triangle inférieur. Les SP$_3$ et SP$_4$ peuvent s'en charger. Ils auront donc une interaction avec le SP$_2$ (triangle supérieur) et le SP$_5$ (triangle inférieur).

Le numéro de chaque SP indique aussi l'ordre d'apparition. On ne peut pas exécuter le SP$_2$ sans avoir lu, au préalable, la valeur de `n`.

L'enchaînement est le suivant :

$$\text{SP}_1 \rightarrow [(\text{SP}_3 \rightarrow \text{SP}_4) \subset \text{SP}_2] \rightarrow [(\text{SP}_3 \rightarrow \text{SP}_4) \subset \text{SP}_5]$$

On peut maintenant passer aux Invariants de Boucle. Voir Sec. 3.6.

3.6 Invariants Graphiques

La troisième étape de la méthodologie de développement. 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 3.6.3
Si vous ne savez pas ce qu'est un Invariant Graphique, allez la Section 3.6.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 3.6.2

3.6.1 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

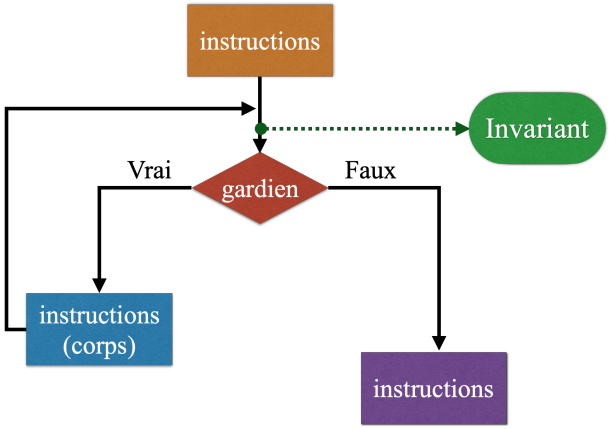


FIGURE 7 – Sémantique de l'Invariant Graphique.

La Fig. 7 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. 7, 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 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. 8 montre l'Invariant Graphique correspondant.

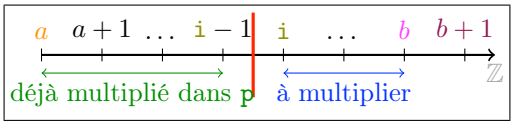


FIGURE 8 – Exemple d'Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

On représente d'abord les entiers entre les limities 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 multiplié. 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. 8) et le nommer ($\mathbb{Z}$ dans la Fig. 8) ;

Règle 2 : placer sur le dessin les bornes de début et de fin (`a`, `b`, et `b`+1 dans la Fig. 8) ;

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. 8). 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. 8) ;

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. 8) ;

Règle 6 : identifier ce qu'il reste à faire dans les itérations suivantes ("à multiplier" dans la Fig. 8) ;

Règle 7 : Toutes les structures identifiées et variables sont présentes dans le code (`a`, `b`, `i`, et `p` sur la Fig. 8) [4].

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.

Suite de l'Exercice

À vous ! Rédigez vos Invariants Graphiques et passez à la Sec. 3.6.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 3.6.2

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

3.6.2 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 :

- 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). Pensez aussi à utiliser le raisonnement général afin d'utiliser les variables et relations identifiées.
- 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, de manière générale, l'objectif à atteindre. N'hésitez pas à vous appuyer sur le GLI pour le dessin.
- 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'objectif 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).

À nouveau, n'hésitez pas à vous appuyer sur le GLI. Il vous permettra, en outre, de valider votre Invariant Graphique sur ses aspects "syntaxiques". On entend par là les Règle 1 → Règle 4 et la présence (mais pas la pertinence ni la signification) des Règle 4 et Règle 5

Suite de l'Exercice

À vous ! Rédigez les Invariants Graphiques et passez à la Sec. 3.6.3.

3.6.3 Mise en Commun des Invariants Graphiques

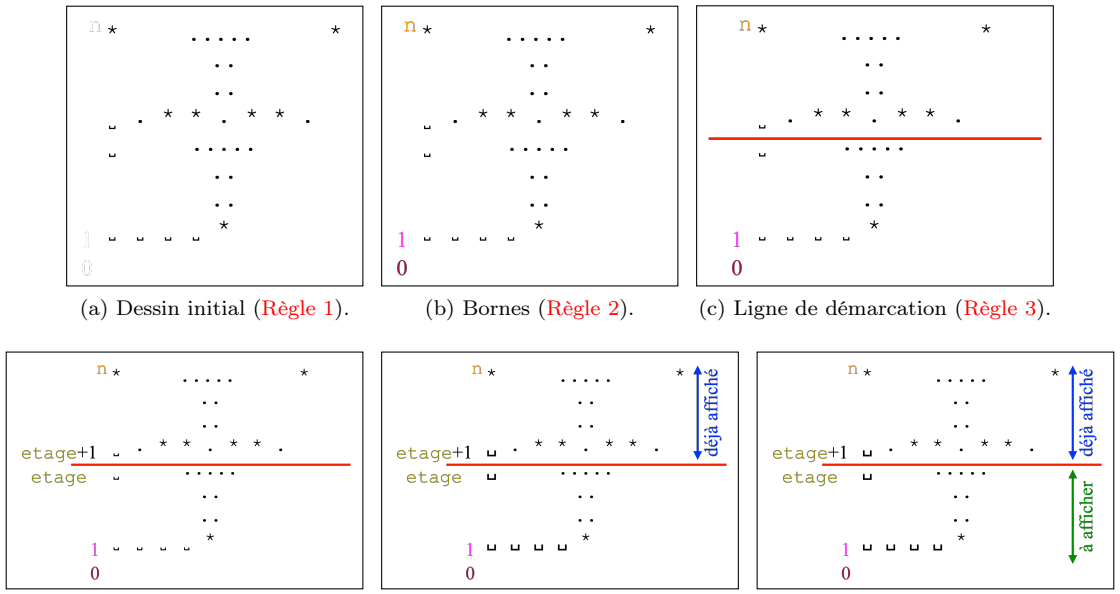
L'analyse du problème nous indique qu'il y a cinq Sp. Le Sp_1 ne demande aucune boucle car c'est une simple lecture au clavier. Les autres Sp, eux, exigent un traitement itératif. On aura donc quatre Invariants Graphiques. Passons-les en revue.

Sp_2

Reprenons la description du Sp_2 et tâchons d'identifier les inputs et outputs.

énumération des valeurs dans [1 ; n] de façon à afficher les différents étages du triangle supérieur.

Le Sp_2 se concentre sur le triangle supérieur du sablier. Le dessin de notre Invariant Graphique devra donc représenter ce triangle, de manière générale (i.e., pour une hauteur `n` quelconque). Le résultat est donné à la Fig. 9a. Le triangle est bien généralisé car il ne dépend pas d'une hauteur `n` particulière. Les ... généralisent le comportement de chacun des étages du triangle.



(a) Dessin initial (Règle 1). (b) Bornes (Règle 2). (c) Ligne de démarcation (Règle 3).

(d) Etiquettage de la ligne de démarcation (Règle 4). (e) Description de ce qui a déjà été fait (Règle 5). (f) Description de ce qu'il reste à faire (Règle 6).

Figure 9 – Construction de l'Invariant Graphique pour le Sp_2.

La description du Sp_2 nous apprend aussi que notre triangle a `n` étages (`n` est hérité du Sp_1). Ceci nous renseigne sur les bornes du dessin. Pour placer les bornes sur le dessin, nous allons suivre le raisonnement général, i.e., numérotation des étages "à l'envers" (la pointe du triangle est l'étage 1, la base est l'étage `n`). Ceci est illustré à la Fig. 9b.

Maintenant que le dessin est borné, nous pouvons placer une ligne de démarcation qui permettra de distinguer le travail déjà effectué par les itérations précédentes de ce qu'il nous reste à faire. La ligne doit être placée dans une situation générale (i.e., ni au niveau des étages inférieurs, ni supérieurs), comme illustré à la Fig. 9c.

Il faut ensuite étiqueter la ligne de démarcation avec une variable, qui nous servira à énumérer toutes les valeurs dans [1 ; `n`] (cfr. description du Sp_2). Nous allons adéquatement nommer cette variable `etage` car c'est plus proche de la sémantique du problème et c'est en relation avec le raisonnement général et la découpe en Sp. Nous choisissons de placer la variable `etage` en dessous de la ligne de démarcation, comme illustré à la Fig. 9d. Ceci relève plus du choix de religion qu'autre chose. Le résultat final (à l'écran) sera le

même si on avait choisi de placer `etage` au-dessus de la ligne de démarcation. Par contre, le code ne sera pas le même [5].

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 description du SP_2 (cfr. ci-dessus) nous indique que l'objectif est d'afficher le triangle supérieur à l'écran. Chaque itéation devra donc afficher une partie du triangle (i.e., un étage). Puisque l'affichage doit se faire sur la sortie standard (i.e., écran), nous sommes obligés de commencer par l'étage `n` jusque l'étage 1 en utilisant le vocabulaire du problème (i.e., affichage). La Fig. 9e illustre cela.

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. 9f.

Le schéma ainsi obtenu (Fig. 9f) est l'Invariant Graphique pour le SP_2.

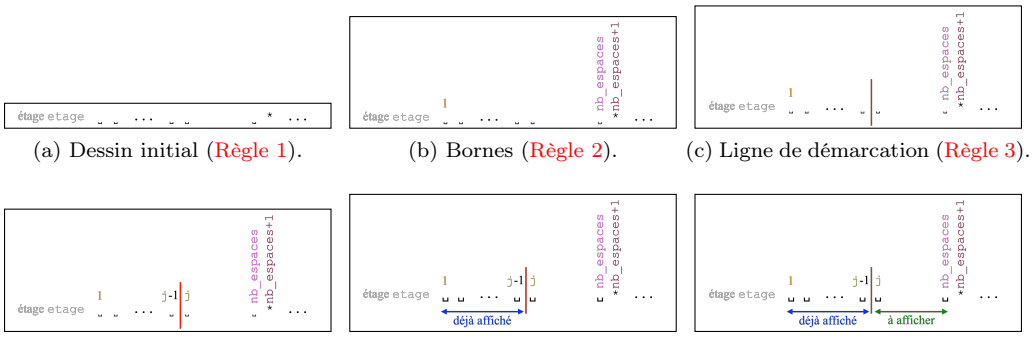
SP_3

La création de l'Invariant Graphique pour le SP_3 suit exactement le même principe que pour le SP_2.

Reprenons la description du SP_3 et tâchons d'identifier les inputs et outputs.

> affichage des caractères `'␣'` sur l'étage `etage`.

Le SP_3 se concentre sur un étage particulier du triangle. Le numéro de l'étage (la variable `etage`) est donné par le SP_2. Ca tombe bien, l'Invariant Graphique du SP_2 indique que la variable servant pour l'énumération des étages s'appelle `etage`. Le dessin de notre Invariant Graphique devra donc représenter un étage, de manière générale (i.e., pour un nombre de `'␣'` quelconque) et nommer cet étage particulier à l'aide de la variable `etage`. Le résultat est donné à la Fig. 10a. L'étage est bien généralisé car il ne dépend pas d'un nombre de `'␣'` particulier. Les ... généralisent le comportement des `'␣'` sur l'étage.



(a) Dessin initial (Règle 1). (b) Bornes (Règle 2). (c) Ligne de démarcation (Règle 3).

(d) Etiquettage de la ligne de démarcation (Règle 3). (e) Description de ce qui a déjà été fait (Règle 5). (f) Description de ce qu'il reste à faire (Règle 6).

FIGURE 10 – Construction de l'Invariant Graphique pour le SP_3.

La découverte des bornes est moins évidente que pour le SP_2. Ici, il faut s'appuyer sur le raisonnement général et en particulier le calcul du nombre de `'␣'`. Celui-ci dépend de la hauteur et de l'étage sur lequel on se trouve. Cependant, dans le raisonnement général, nous avons une équation qui les relie. Ce qui signifie que le nombre de `'␣'` peut être calculé à l'avance, et donc stocké dans une variable. Cette variable, d'après le raisonnement général, c'est `nb_espaces`. Les bornes peuvent donc être placées, comme sur la Fig. 10b.

Maintenant que le dessin est borné, nous pouvons placer une ligne de démarcation qui permettra de distinguer le travail déjà effectué par les itérations précédentes de ce qu'il nous reste à faire. La ligne doit être placée dans une situation générale (i.e., ni au début de l'étage, ni à la fin), comme illustré à la Fig. 10c.

5. C'est un bon exercice à faire par vous-même. Produisez le Invariant Graphique avec la variable `etage` au dessus de la ligne de démarcation et faites la construction du code en fonction de cet Invariant de Boucle. Venez en discuter sur eCampus

En outre, nous pouvons étiqueter la ligne de démarcation avec une variable (`j`), que nous plaçons à droite de la ligne (cfr. Fig. 10d).

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 (Fig. 10e) et ce qu'il reste à faire (Fig. 10f).

Le schéma ainsi obtenu (Fig. 10f) est l'Invariant Graphique pour le SP$_3$.

SP$_4$

Le SP$_4$ est quasiment identique au SP$_3$ à la différence qu'on doit afficher un certain nombre de `'*'` sur un `etage` donné. Le dessin général de l'Invariant Graphique est donc légèrement différent ainsi que les bornes. En particulier, la borne supérieure est donné par le raisonnement général et en particulier le calcul du nombre de `'*'`. Celui-ci dépend seulement de l'étage sur lequel on se retrouve. Cependant, dans le raisonnement général, nous avons une équation qui les relie. Ce qui signifie que le nombre de `'*'` peut être calculé à l'avance, et donc stocké dans une variable. Cette variable, d'après le raisonnement général, c'est `nb_etoiles`. La Fig. 11 donne l'Invariant Graphique pour le SP$_4$. Le raisonnement pour le construire est le même que celui du SP$_3$.

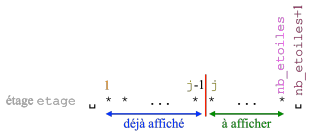


FIGURE 11 – Invariant Graphique pour le SP$_4$.

SP$_5$

L'Invariant Graphique pour le SP$_5$ est assez proche du SP$_2$, à deux différences près :

1. le triangle est orienté dans l'autre sens (i.e., la pointe vers le haut) ;
2. les étages sont numérotés dans l'autre sens (i.e., de 1 à `n`).

L'Invariant Graphique est donné à la Fig. 12.

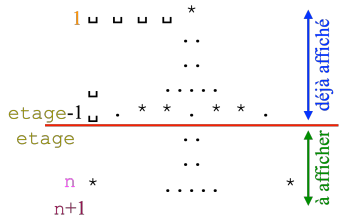


FIGURE 12 – Invariant Graphique pour le SP$_5$.

On peut maintenant passer à l'écriture du code, en particulier la ZONE 1. Voir Sec. 3.7.

3.7 Construction du Code : Zone 1

Une fois les Invariants Graphiques établis, 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 3.7.3
Si vous ne voyez pas le lien entre Invariant Graphique et construction du code, reportez-vous à la Section 3.7.1
Enfin, si vous ne savez pas quoi faire, reportez-vous à l'indice 3.7.2

3.7.1 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 cours 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 stratégie [6] de résolution du problème. Cette stratégie permet d'identifier quatre points particuliers : trois Zones (voir la Fig. 13) et le Critère d'Arrêt.

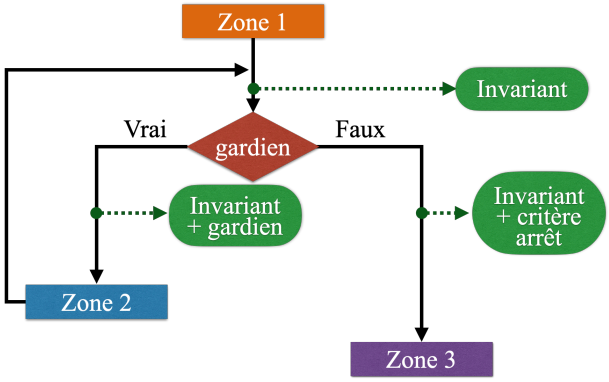


Figure 13 – Relation entre le Invariant Graphique et les différentes zones du programme.

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'Arrêt [7].

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

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

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

$a \in \mathbb{Z}$

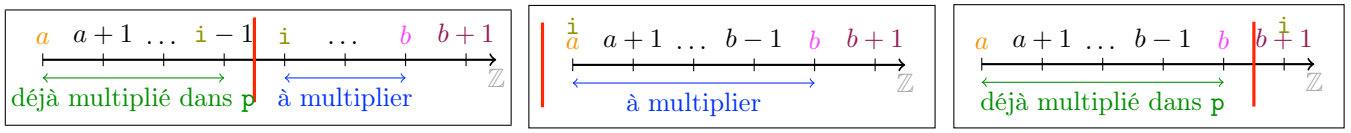
```
int a;
```

b, la borne supérieure

$b \in \mathbb{Z}$

```
int b;
```

La Fig. 14a montre l'Invariant Graphique correspondant (les détails sur la construction de l'Invariant Graphique sont donnés dans le rappel).



(a) Invariant Graphique.

(b) Situation initiale (Zone 1).

(c) Situation finale (Zone 3 et Critère d'Arrêt).

Figure 14 – 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. 14b. 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. 14b, `a`. On remarque aussi sur la Fig. 14b 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. 14c. 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 $\texttt{i} = \texttt{b} + 1$. Dès lors, le Gardien de Boucle correspondant est `i != b + 1` mais on préfèrera la version `i <= 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 :

```
while(i <= b){
  //à compléter (Zone 2)
}//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. 14a 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

```
p *= i;
i++;
```

Zone 3 On peut déduire, de la situation finale illustrée à la Fig. 14c, 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

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

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

```
#include <stdio.h>

int main(){
  //Zone 1
  int a, b, p, i;

  scanf("%d", &a);
  scanf("%d", &b);

  i = a;
  p = 1;

  while(i <= b){
    //Zone 2
    p *= i;
    i++;
  }//fin boucle

  //Zone 3
  printf("%d", p);
}//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 trainer 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

3.7.2 Indice

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

- reprenez les Invariants Graphiques des différents Sp ;
- identifiez les variables présentes dans votre Invariant Graphique ;
- aidez-vous du Gli pour faciliter la manipulation graphique (vous pouvez explicitement déplacer la ligne de démarcation, l'Invariant Graphique s'adapte alors automatiquement) ; Cette manipulation vous donnera certaines valeurs d'initialisation ;
- pensez à vous appuyez sur le raisonnement général. Nous y avons établi des relations entre certains éléments clés du problème.

Suite de l'Exercice

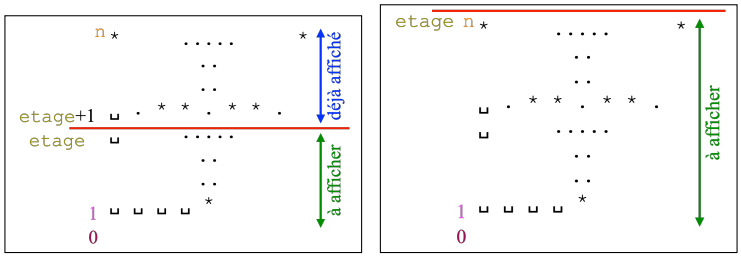
À vous ! Construisez la Zone 1 pour les différents Sp et passez à la Sec. 3.7.3.

3.7.3 Mise en Commun de la Zone 1

Il faut dériver les instructions pour la Zone 1 pour chaque Sp faisant intervenir un Invariant de Boucle. Passons en revue chacun des Sp.

Sp_2

La Fig. 15a donne l'Invariant Graphique pour le Sp_2. Pour rappel, le Sp_2 s'intéresse à l'énumération des valeurs dans [1 ; n] de façon à afficher les différents étages du triangle supérieur.



(a) Invariant Graphique pour le Sp_2. (b) Zone 1

Figure 15 – Manipulation graphique de l'Invariant Graphique du Sp_2 pour la Zone 1.

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

`n` : il s'agit de la hauteur du triangle. Cette variable a déjà été identifiée lors de la définition du problème. Elle est de type `unsigned int`.

`etage` : c'est la variable d'itération qui va nous permettre d'énumérer les valeurs dans [1 ; n]. Pour le choix du type, il faut rester cohérent. Si `n` est de type `unsigned int`, alors `etage` doit être du même type (on ne compare pas des pommes et des poires).

Il faut maintenant voir comment initialiser ces deux variables. Pour cela, il suffit de manipuler graphiquement l'Invariant Graphique en faisant glisser la ligne de démarcation vers sa valeur maximale (vers le haut, dans notre cas), puisqu'on travaille de l'étage `n` à l'étage 1. Lors de ce déplacement, on accompagne la ligne de démarcation avec la variable d'itération (`etage`). On voir alors clairement (cfr. Fig. 15b) que la partie verte a disparu et que la partie verte recouvre tout le triangle. Cela signifie qu'à la première évaluation du Gardien de Boucle, aucun étage du triangle n'a déjà été affiché. On voit clairement aussi que la valeur de la variable `etage` est `n`.

La variable `n` est elle initialisée grâce au Sp_1 (i.e., lecture au clavier).

On a donc le code suivant :

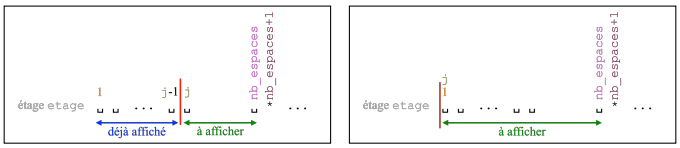
```
1 unsigned int n, etage;
2
3 scanf("%u", &n);
4
5 etage = n;
```

Sp_3

La Fig. 16a donne l'Invariant Graphique pour le Sp_3. Pour rappel, le Sp_3 s'intéresse à l'affichage des caractères '␣' sur l'étage `etage`.

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

`etage` : c'est l'étage sur lequel on travaille actuellement. Cette variable est donnée par le Sp_2.



(a) Invariant Graphique pour le SP_3. (b) Zone 1

FIGURE 16 – Manipulation graphique de l'Invariant Graphique du SP_3 pour la ZONE 1.

`nb_espaces` : c'est la variable qui indique le nombre total de `'␣'` à afficher pour l'étage courant. Ce nombre ne peut pas être un réel ni négatif. Dès lors, naturellement, son type sera `unsigned int`.

`j` : c'est la variable d'itération qui nous permet de compter le nombre de `'␣'` affiché à l'écran. Pour le choix du type, il faut rester cohérent. Si `nb_espaces` est de type `unsigned int`, alors `j` doit être du même type (on ne compare pas des pommes et des poires).

Il faut maintenant voir comment initialiser ces deux variables. `etage` est obtenu grâce au SP_2. Il n'y a donc pas besoin de l'initialiser. Pour `nb_espaces`, il faut s'appuyer sur le raisonnement général où nous avons indiqué la relation existante entre le nombre de `'␣'`, la hauteur du triangle et l'étage sur lequel on se trouve (i.e., $\texttt{nb\_espaces} = \texttt{n} - \texttt{etage}$). C'est cette relation qui permettra l'initialisation de `nb_espaces`.

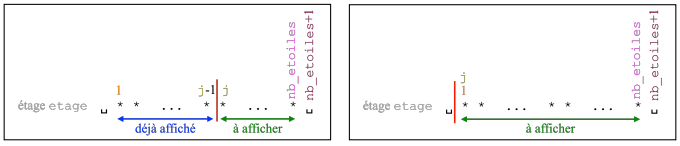
Enfin, pour `j`, il suffit de faire glisser la ligne de démarcation vers sa valeur minimale (vers la gauche, dans notre cas). Lors de ce déplacement, on accompagne la ligne de démarcation avec la variable d'itération (`j`). On voir alors clairement (cfr. Fig. 16b) que la partie verte a disparu et que la partie verte recouvre toute la ligne. Cela signifie qu'à la première évaluation du Gardien de Boucle, aucun `'␣'` n'a déjà été affiché. On voit clairement aussi que la valeur de la variable `j` est `1`.

On a donc le code suivant :

```
1 unsigned int nb_espaces, j;
2
3 nb_espaces = n - etage;
4
5 j = 1;
```

SP_4

La Fig. 17a donne l'Invariant Graphique pour le SP_4. Pour rappel, le SP_4 s'intéresse à affichage des caractères `'*'` sur l'étage `etage`. C'est donc un problème assez proche du SP_3.



(a) Invariant Graphique pour le SP_4. (b) Zone 1

FIGURE 17 – Manipulation graphique de l'Invariant Graphique du SP_4 pour la ZONE 1.

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

`etage` : c'est l'étage sur lequel on travaille actuellement. Cette variable est donnée par le SP_2.

`nb_etoiles` : c'est la variable qui indique le nombre total de `'*'` à afficher pour l'étage courant. Ce nombre ne peut pas être un réel ni négatif. Dès lors, naturellement, son type sera `unsigned int`.

`j` : c'est la variable d'itération qui nous permet de compter le nombre de `'*'` affiché à l'écran. C'est la même variable que pour le SP_3. Ca ne pose pas de problème puisqu'elle intervient dans deux boucles totalement différentes.

Il faut maintenant voir comment initialiser ces deux variables. `etage` est obtenu grâce au SP_2. Il n'y a donc pas besoin de l'initialiser. Pour `nb_etoiles`, il faut s'appuyer sur le raisonnement général où nous avons indiqué la relation existante entre le nombre de `'*'` et l'étage sur lequel on se trouve (i.e., `nb_etoiles` = (2 × `etage`) − 1). C'est cette relation qui permettra l'initialisation de `nb_etoiles`.

Enfin, pour `j`, il suffit de faire glisser la ligne de démarcation vers sa valeur minimale (vers la gauche, dans notre cas). Lors de ce déplacement, on accompagne la ligne de démarcation avec la variable d'itération (`j`). On voir alors clairement (cfr. Fig. 17b) que la partie verte a disparu et que la partie verte recouvre toute la ligne . Cela signifie qu'à la première évaluation du Gardien de Boucle, aucun `'*'` n'a déjà été affiché. On voit clairement aussi que la valeur de la variable `j` est `1`.

On a donc le code suivant :

```
unsigned int nb_etoiles;

nb_etoiles = (2 * etage) - 1;

j = 1;
```

SP_5

La découverte de la ZONE 1 pour le SP_5 est quasiment identique au SP_2. Le raisonnement est exactement le même, mais `n` a déjà été initialisé.

On a donc le code suivant :

```
etage = 1;
```

On peut maintenant continuer l'écriture du code, en particulier le Critère d'Arrêt et la Fonction de Terminaison. Voir Sec. 3.8.

3.8 Critères d'Arrêt et Fonctions de Terminaison

Une fois les Invariants Graphiques établis, 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 au Critère d'Arrêt et la Fonction de Terminaison.

Si vous voyez de quoi on parle, rendez-vous à la Section 3.8.3
Si vous ne voyez pas le lien entre Invariant Graphique et construction du code, reportez-vous à la Section 3.7.1
Si le concept de Fonction de Terminaison ne vous parle pas, rendez-vous à la Section 3.8.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 3.8.2

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

Une Fonction de Terminaison permet de déterminer, formellement [8], qu'une boucle se termine (i.e., le Critère d'Arrêt est atteint).

Une *Fonction de Terminaison* est une fonction entière (i.e., les résulats $\in \mathbb{Z}$) 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, $f : \{\text{valeurs des variables}\} \to \mathbb{Z}$, 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 ne décroit pas strictement à chaque itération. Dans ce cas, il est 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. 18 [9].

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

8. Formellement == mathématiquement
9. Les détails sur la création de cet Invariant Graphique sont donnés dans le rappel.

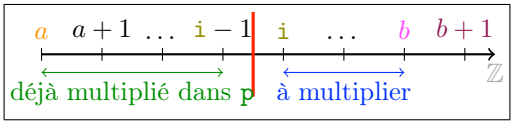


FIGURE 18 – Exemple d'Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

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 cette 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 : $i <= b$ [10].

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 $0 <= b - i$.
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. 3.8.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 3.8.2

10. Pour les détails, voir le rappel sur la construction basée sur l'Invariant Graphique

3.8.2 Indice

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

- reprenez les Invariants Graphiques des différents Sp ;
- assurez-vous que les Invariants Graphiques contiennent bien une ligne de démarcation et une variable pour étiqueter cette ligne ;
- aidez-vous du Gli pour faciliter la manipulation graphique (vous pouvez explicitement déplacer la 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 le Critère d'Arrêt ;

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

Suite de l'Exercice

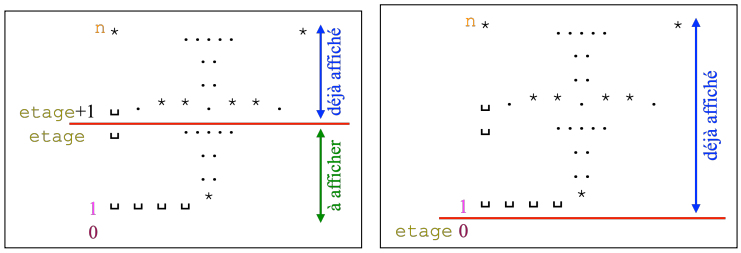
À vous ! Rédigez les Critères d'Arrêt et Fonctions de Terminaison pour les différents Sp et passez à la Sec. 3.8.3.

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

Il faut dériver le Critère d'Arrêt et la Fonction de Terminaison pour chaque SP faisant intervenir un Invariant de Boucle. Passons en revue chacun des SP.

SP_2

La Fig. 19a donne l'Invariant Graphique pour le SP_2. Pour rappel, le SP_2 s'intéresse à l'énumération des valeurs dans [1 ; n] de façon à afficher les différents étages du triangle supérieur.



(a) Invariant Graphique pour le SP_2. (b) Situation finale.

FIGURE 19 – Manipulation graphique de l'Invariant Graphique du SP_2 pour le Critère d'Arrêt.

Commençons par le Critère d'Arrêt. Pour le trouver, il faut faire glisser la ligne de démarcation de sorte que la zone verte ("à afficher") ait totalement disparu et que la zone bleue ("déjà affiché") recouvre tout le dessin. Nous nous trouvons alors dans la situation finale (voir Fig. 19b) car la boucle a atteint son objectif. Nous observons que la variable d'itération, `etage`, prend la valeur particulière 0. Nous déduisons que le Critère d'Arrêt est : `etage` == 0. On peut maintenant déduire le Gardien de Boucle : `etage != 0`. Nous préférons la formulation (équivalente) `etage >= 1` qui a l'avantage de mieux exprimer la position relative de `etage` sur le triangle.

On obtient alors le code suivant :

```
1 while(etage >= 1){
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 afficher"). On sait que le triangle a une hauteur de n. La zone bleue ("déjà affiché") a une taille de $n - etage$. Il suffit alors de faire la différence entre la hauteur du triangle et la taille de la zone bleue. Soit $n - (n - etage)$.

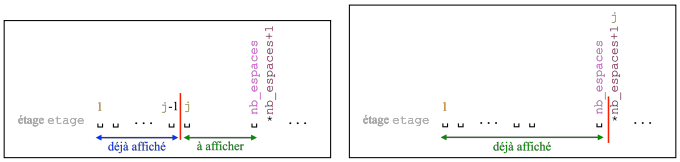
La Fonction de Terminaison est : $f : etage$.

SP_3

La Fig. 20a donne l'Invariant Graphique pour le SP_3. Pour rappel, le SP_3 s'intéresse à l'affichage des caractères '␣' sur l'étage `etage`.

Commençons par le Critère d'Arrêt. Pour le trouver, il faut faire glisser la ligne de démarcation de sorte que la zone verte ("à afficher") ait totalement disparu et que la zone bleue ("déjà affiché") recouvre tout le dessin. Nous nous trouvons alors dans la situation finale (voir Fig. 20b) car la boucle a atteint son objectif. Nous observons que la variable d'itération, `j`, prend la valeur particulière $nb\_espaces + 1$. Nous déduisons que le Critère d'Arrêt est : $j == nb\_espaces + 1$. On peut maintenant déduire le Gardien de Boucle : `j != nb_espaces`. Nous préférons la formulation (équivalente) `j <= nb_espaces` qui a l'avantage de mieux exprimer la position relative de `j` sur l'étage courant.

On obtient alors le code suivant :



(a) Invariant Graphique pour le SP3. (b) Situation finale.

FIGURE 20 – Manipulation graphique de l'Invariant Graphique du SP3 pour le Critère d'Arrêt.

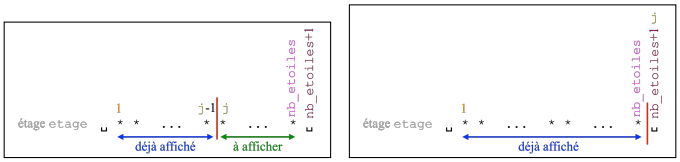
```
while(j <= nb_espaces){
  //à compléter (ZONE 2)
}//fin while - j
```

Passons à la Fonction de Terminaison. Il suffit, ici, d'estimer la taille de la zone verte ("à afficher"). On sait que le nombre de '␣' sur un étage donné est de $nb\_espaces$. La zone bleue ("déjà affiché") a une taille de $j-1$. Il suffit alors de faire la différence entre le nombre de '␣' et la taille de la zone bleue. Soit $nb\_espaces-(j-1)$.

La Fonction de Terminaison est : $f : nb\_espaces - j + 1$.

SP4

La Fig. 21a donne l'Invariant Graphique pour le SP4. Pour rappel, le SP4 s'intéresse à affichage des caractères '*' sur l'étage `etage`. C'est donc un problème assez proche du SP3.



(a) Invariant Graphique pour le SP4. (b) Situation finale.

FIGURE 21 – Manipulation graphique de l'Invariant Graphique du SP4 pour le Critère d'Arrêt.

Commençons par le Critère d'Arrêt. En faisant glisser la ligne de démarcation (Fig. 21b), on se rend compte que le problème est quasi identique au SP3. Seule la borne supérieure (`nb_etoiles`) change. En procédant comme pour le SP3, on trouve le Critère d'Arrêt $j == nb\_etoiles + 1$. Ce qui donne le Gardien de Boucle suivant `j <= nb_etoiles` et le code suivant :

```
while(j <= nb_etoiles){
  //à compléter (ZONE 2)
}//fin while - j
```

Pour la Fonction de Terminaison, le raisonnement est identique au SP3. On obtient alors : $f : nb\_etoiles - j + 1$.

SP5

Le raisonnement pour le SP_5 est sensiblement identique à celui du SP_2, en l'adaptant au fait que le triangle n'est pas dans le même sens.

On obtient le Critère d'Arrêt $etage == n+1$, ce qui mène au Gardien de Boucle `etage <= n` et le code :

```
while(etage <= n){
  //à compléter (Zone 2)
}//fin while - etage
```

La Fonction de Terminaison est $f : n - etage + 1$.

On peut maintenant continuer l'écriture du code, en particulier la Zone 2. Voir Sec. 3.9.

3.9 Construction du Code : Zone 2

Une fois les Invariants Graphiques établis, 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 3.9.2
Si vous ne voyez pas le lien entre Invariant Graphique et construction du code, reportez-vous à la Section 3.7.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 3.9.1

3.9.1 Indice

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

- reprenez les Invariants Graphiques des différents Sp ;
- 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 zone bleue 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 conséquence, la forme générale de l'Invariant Graphique doit être restaurée.

Suite de l'Exercice

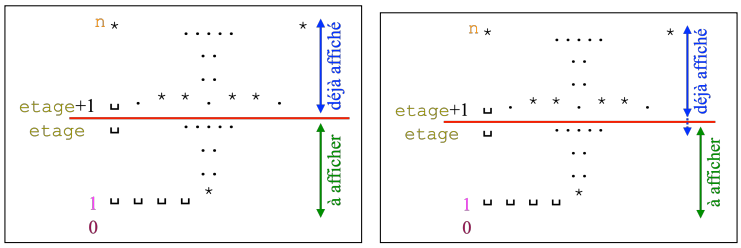
À vous ! Construisez la Zone 2 pour les différents Sp et passez à la Sec. 3.9.2.

3.9.2 Mise en Commun de la Zone 2

Il faut dériver les instructions pour la Zone 2 pour chaque Sp faisant intervenir un Invariant de Boucle. Passons en revue chacun des Sp.

Sp_2

La Fig. 22a donne l'Invariant Graphique pour le Sp_2. Pour rappel, le Sp_2 s'intéresse à l'énumération des valeurs dans [1 ; n] de façon à afficher les différents étages du triangle supérieur.



(a) Invariant Graphique pour le Sp_2. (b) Situation intermédiaire.

Figure 22 – Manipulation graphique de l'Invariant Graphique du Sp_2 pour la Zone 2.

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. Dans le cadre du Sp_2, cela signifie qu'il reste, au moins, un étage du triangle supérieur à afficher à l'écran. L'objectif de la Zone 2 est de faire avancer le problème en augmentant la taille de la zone bleue et en réduisant d'autant la zone verte. Cette situation particulière est illustrée à la Fig. 22b. On voit bien, en étirant la flèche bleue d'une position, qu'il faut afficher à l'écran l'étage `etage`. C'est très facile : un étage est composé d'un certain nombre de `'␣'` (Sp_3) suivi par un certain nombre de `'*'` (Sp_4). Ces deux étapes sont totalement raccord avec l'enchaînement des Sp : le Sp_3 et le Sp_4 se suivent et sont inclus dans le Sp_2 [11]. Une fois que c'est fait, pour restaurer l'Invariant Graphique, il faut passer à la ligne (`printf("\n")`) et décrémenter `etage` d'une unité de façon à faire descendre la ligne de démarcation d'un étage.

On obtient alors le code suivant :

```
while(etage >= 1){
  //Code Sp3

  //Code Sp4

  printf("\n");
  etage--;
}//fin while - etage
```

Sp_3

La Fig. 23a donne l'Invariant Graphique pour le Sp_3. Pour rappel, le Sp_3 s'intéresse à l'affichage des caractères `'␣'` sur l'étage `etage`.

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. Dans le cadre du Sp_2, cela signifie qu'il reste, au moins, un `'␣'` de l'étage courant à afficher à l'écran. L'objectif de la Zone 2 est de faire avancer le problème en augmentant la taille de la zone bleue et en réduisant d'autant la zone verte. Cette situation particulière est illustrée à la Fig. 23b.

11. Ou dans le Sp_5

(a) Invariant Graphique pour le SP_3.

(b) Situation intermédiaire.

FIGURE 23 – Manipulation graphique de l'Invariant Graphique du SP_3 pour la ZONE 2.

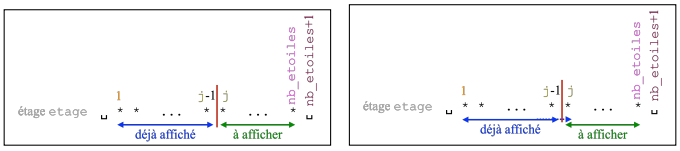
On voit bien, en étirant la flèche bleue d'une position, qu'il faut afficher à l'écran un `'␣'`. Une fois que c'est fait, pour restaurer l'Invariant Graphique, il faut incrémenter la variable d'itération `j` d'une unité de façon à faire glisser la ligne de démarcation vers la droite.

On obtient alors le code suivant :

```
while(j <= nb_espaces){
  printf("␣");
  j++;
}//fin while - j
```

SP_4

La Fig. 24a donne l'Invariant Graphique pour le SP_4. Pour rappel, le SP_4 s'intéresse à l'affichage des caractères `'*'` sur l'étage `etage`. C'est donc un problème assez proche du SP_3.



(a) Invariant Graphique pour le SP_4.

(b) Situation intermédiaire.

FIGURE 24 – Manipulation graphique de l'Invariant Graphique du SP_4 pour la ZONE 2.

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. Dans le cadre du SP_2, cela signifie qu'il reste, au moins, un `'*'` de l'étage courant à afficher à l'écran. L'objectif de la ZONE 2 est de faire avancer le problème en augmentant la taille de la zone bleue et en réduisant d'autant la zone verte. Cette situation particulière est illustrée à la Fig. 24b. On voit bien, en étirant la flèche bleue d'une position, qu'il faut afficher à l'écran un `'*'`. Une fois que c'est fait, pour restaurer l'Invariant Graphique, il faut incrémenter la variable d'itération `j` d'une unité de façon à faire glisser la ligne de démarcation vers la droite.

On obtient alors le code suivant :

```
while(j <= nb_etoiles){
  printf("*");
  j++;
}//fin while -- j
```

SP$_5$

Le raisonnement pour le SP$_5$ est sensiblement identique à celui du SP$_2$, en l'adaptant au fait que le triangle n'est pas dans le même sens. Ce qui nous amène au code suivant :

```
while(etage <= n){
  //Code SP3

  //Code SP4

  printf("\n");
  etage++;
}//fin while -- etage
```

On peut maintenant continuer l'écriture du code, en particulier la ZONE 3. Voir Sec. 3.10.

3.10 Construction du Code : Zone 3

Une fois les Invariants Graphiques établis, 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 3.10.2
Si vous ne voyez pas le lien entre Invariant Graphique et construction du code, reportez-vous à la Section 3.7.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 3.10.1

3.10.1 Indice

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

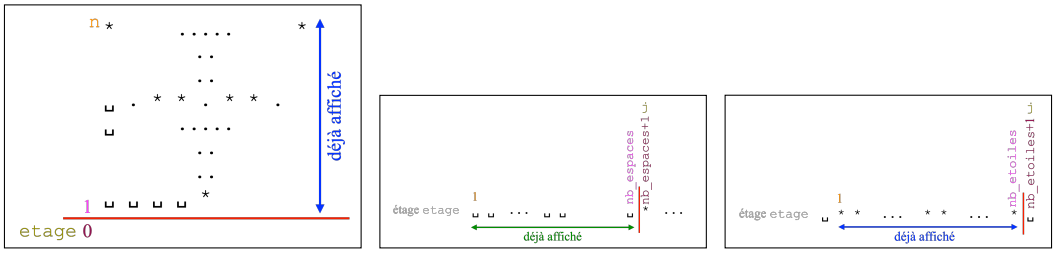
- reprenez les Invariants Graphiques des différents Sp ;
- assurez-vous que les Invariants Graphiques contiennent bien une ligne de démarcation et une variable pour étiqueter cette ligne ;
- aidez-vous du Gli pour faciliter la manipulation graphique (vous pouvez explicitement déplacer la ligne de démarcation, 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. 3.10.2.

3.10.2 Mise en Commun de la Zone 3

Il faut dériver les instructions pour la Zone 3 pour chaque Sp faisant intervenir un Invariant de Boucle. Passons en revue chacun des Sp.



(a) Sp_2. (b) Sp_3. (c) Sp_4.

Figure 25 – Manipulation graphique de l'Invariant Graphique pour 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).

Dans le cadre du Sp_2, cela signifie que le triangle supérieur a été entièrement affiché (voir Fig. 25a). Il faut donc, ensuite, afficher le triangle inférieur. C'est l'objectif du Sp_5. C'est totalement raccord avec l'enchaînement des Sp : le Sp_2 et le Sp_5 se suivent. Après le Sp_2, il suffit donc de placer le code du Sp_5.

Dans le cadre du Sp_3, cela signifie que tous les `'␣'` de l'étage courant ont été affichés (voir Fig. 25b). Il faut donc, ensuite, afficher les `'*'`. Soit le Sp_4. À nouveau, c'est totalement raccord avec l'enchaînement des Sp : le Sp_3 et le Sp_4 se suivent. Dès lors, après le Sp_3, il suffit de placer le code du Sp_4 [12].

Dans le cadre du Sp_4, cela signifie que tous les `'*'` de l'étage courant ont été affichés (voir Fig. 25c). Il faut donc, ensuite, afficher l'étage suivant du triangle. À nouveau, c'est totalement raccord avec l'enchaînement des Sp : le Sp_3 et le Sp_4 se suivent et sont inclus dans le Sp_2 [13]. Il n'y a donc rien de particulier à faire, hormis continuer le Sp_2 [14].

Dans le cadre du Sp_5, cela signifie que le triangle inférieur a été entièrement affiché. Le problème général est donc terminé (nous avons atteint l'output du problème). Le programme est dès lors terminé.

On peut maintenant continuer l'écriture du code, en particulier mettre les différentes parties ensemble. Voir Sec. 3.11.

12. Ceci était, de toute façon, déjà suggéré par la Zone 2 du Sp_2 (voir Sec. 3.9.2)
13. ou dans le Sp_5.
14. ou le Sp_5

3.11 Code Final

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

La construction des boucles (voir 3.8.3 et 3.9.2) s'est faite sur base de boucles `while` afin de bien illustrer le mécanisme de construction du code et des différentes zones. Afin de réduire la quantité de lignes de code, nous proposons, ici, le code final à l'aide de boucles `for`.

On obtient alors le code final suivant :

```
#include <stdio.h>
int main(){
  unsigned int n=0, etage=0, j=0, nb_espaces=0, nb_etoiles=0;

  //début Sp1
  printf("Entrez␣une␣valeur␣pour␣n:␣");
  scanf("%u", &n);
  //fin Sp1

  //début Sp2
  for(etage=n; etage>=1; etage--){
    //début Sp3
    nb_espaces = n - etage;
    for(j=1; j<=nb_espaces; j++)
      printf("␣");
    //fin Sp3

    //début Sp4
    nb_etoiles = (2 * etage) - 1;
    for(j=1; j<=nb_etoiles; j++)
      printf("*");
    //fin Sp4

    printf("\n");
  }//fin for - etage
  //fin Sp2

  //début Sp5
  for(etage=1; etage<=n; etage++){
    //début Sp3
    nb_espaces = n - etage;
    for(j=1; j<=nb_espaces; j++)
      printf("␣");
    //fin Sp3

    //début Sp4
    nb_etoiles = (2 * etage) - 1;
    for(j=1; j<=nb_etoiles; j++)
      printf("*");
    //fin Sp4

    printf("\n");
  }//fin for - etage
  //fin Sp5
}//fin programme
```