Université de Liège

INFO0946

Introduction à la Programmation

Un exercice dont vous êtes le Héros · l'Héroïne

Programmation Modulaire

Benoit Donnet Simon Liénardy Tasnim Safadi

26 novembre 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 la programmation modulaire.

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.

6.1 Rappels sur les Modules

6.1.1 Généralités

Un *module* est un morceau de code qui est écrit indépendamment du programme principal (i.e., `int main()`) et qui peut être *invoqué* (ou *appelé*) à partir de plusieurs endroits du programme.
Un module peut être une

fonction . Dans ce cas, le module retourne un résultat qui pourra être utilisé par la suite. C'est le cas, par exemple, des modules `fscanf()` ou encore `fopen()` que nous avons vu dans le Chapitre 5.

procédure . Dans ce cas, le module ne retourne aucun résultat. C'est le cas, par exemple, du module `printf()` que nous avons vu dans le Chapitre 1.

6.1.2 Fonctionnement

Le fonctionnement de la programmation modulaire est assez simple :

- Le code (peu importe où) invoque un module. Celui qui invoque est appelé le *code appelant*.
- Dès l'invocation, l'exécution du code appelant est suspendue (i.e., ce n'est plus lui qui occupe le CPU). C'est le module qui prend la main (i.e., c'est le code du module qui occupe le CPU).
- Le code appelant reprend la main lorsque le module invoqué est terminé.

A l'instar d'une fonction mathématique, un module peut disposer d'*arguments*

paramètres formels . Ce sont les données en entrée utilisées par le module. Ces paramètres sont équivalents à des variables locales au module et n'ont donc d'existence que dans le module où ils sont définis.
Pour faire le parallèle avec les fonctions mathématiques, la fonction

$$f(x) = x^2 + 2$$

x est le paramètre formel de la fonction et $x^2 + 2$ représente l'instruction à exécuter quand on exécute la fonction f.

paramètres effectifs . Ce sont les données qui sont passées au module lors de l'invocation et ont pour but d'initaliser les paramètres formels. Le lien entre paramètres formels et paramètres effectifs s'appelle *passage de paramètres* et sera vu au Chapitre 7.
Pour reprendre l'analogie avec les mathématiques, l'invocation de la fonction f peut se faire comme suit :

$$f(5)$$

5 est le paramètre effectif. Lors de l'invocation, le paramètre formel x prend la valeur 5 et la fonction f calcule, effectivement, $5^2 + 2$.

6.1.3 Déclaration

Un module (fonction ou procédure) est composé de deux parties :

le prototype . Il sert à identifier, uniquement, le module. Le prototype reprend l'éventuel type de retour du module, l'identificateur du module [2] et la liste des paramètres formels. On parle aussi de *signature* du module.

le corps . C'est l'ensemble des instructions à exécuter lors de l'invocation du module. Dans le corps du module, on fonctionne comme dans le `main()` : déclaration de variables, instructions, ...

2. L'identificateur d'un module obéit aux mêmes règles que l'identificateur d'une variable – cfr. Chapitre 1.

Pour pouvoir invoquer un module, il faut au préalable le déclarer. Rien d'inhabituel ici, c'est le même principe pour les variables (i.e., une variable ne peut être utilisée avant d'être déclarée). La déclaration d'un module peut être

implicite . Il s'agit d'une déclaration complète où on met ensemble le prototype et le corps du module. La déclaration se fait alors entre les dérives de compilation (e.g., `#include <stdio.h>`) et le programme principal (i.e., `int main()`). Voir Extrait de Code 1 pour une déclaration implicite et invocation d'une fonction.

Extrait de code 1 – Déclaration implicite et invocation d'une fonction

```
#include <stdio.h>

int puissance7(int x){
    int resultat;

    resultat = x*x*x*x*x*x*x;

    return resultat;
}//fin puissance7()

int main(){
  printf("%d\n", puissance7(5));
}//fin programme principal
```

explicite . Il s'agit, ici, de séparer le prototype du corps. Le prototype est alors déclarer dans un fichier séparé (*header*, fichier `.h`) du corps (fichier `.c`). La déclaration implicite permet d'appliquer les principes de la *compilation séparée*. Ce qui mène, généralement, à la création d'une librairie [3]. Voir les Extraits de Code 2, 3 et 4 pour une déclaration explicite et invocation d'une procédure.

Extrait de code 2 – Fichier `tableau.h`

```
void afficher_tableau(int tab[], int n);
```

Extrait de code 3 – Fichier `tableau.c`

```
#include <stdio.h>
#include "tableau.h"

void afficher_tableau(int tab[], int n){
  int i;

  printf("[␣");
  for(i=0; i<n; i++)
    printf("%d␣", tab[i]);
  print("]\n");
}//fin afficher_tableau()
```

Extrait de code 4 – Programme principal (fichier `main-tableau.c`)

```
#include "tableau.h"

int main(){
  const int N = 10;
  int tab[N];

  //remplissage du tableau

  afficher_tableau(tab, N);
}//fin programme
```

3. Cfr. INFO0030

6.2 Énoncé

Simon souhaite disposer d'une énumération des nombres *narcissiques*. Un nombre entier positif est dit narcissique si il est égal à somme de chiffres qui le composent, chacun des chiffres étant élevé à la puissance correspondant à la quantité de chiffres dans le nombre de base.

Par exemple

- 153 est un nombre narcissique car $153 = 1^3 + 5^3 + 3^3$
- 8208 est un nombre narcissique car $8208 = 8^4 + 2^4 + 0^4 + 8^4$.

Simon voudrait afficher sur la sortie standard tous les nombres narcissiques entre 1 et x, donné au clavier.

Dans ce GameCode, nous vous demandons de suivre la méthodologie de développement vue au cours. De plus, ce GameCode impose les contraintes suivantes :

- vous devez construire votre code en suivant les principes de la programmation modulaire (header et fichier `.c`) ;
- vous devez appliquer les principes de la programmation défensive, quand cela s'avère pertinent ;
- vous ne pouvez utiliser aucune librairie extérieure, à l'exception de `stdio.h` et `assert.h` ;
- vous ne pouvez utiliser que des boucles de type `while`.

Pour tester votre solution, vous pourrez partir du programme principal suivant (à compléter bien entendu) :

Extrait de code 5 – Programme principal à compléter

```c
#include <stdio.h>

int main(void){
  unsigned int x;

  printf("Entrez une valeur pour x: ");
  scanf("%u", &x);

  printf("Liste des nombres narcissiques entre 1 et %u: \n", x);
  //ICI: invocation de vos modules

  return 0;
}//fin programme
```

6.2.1 Méthode de Résolution

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

1. Définition du problème (Sec. 6.3) ;
2. Analyse du problème (Sec. 6.4) ;
3. Architecture du code (Sec. 6.5) ;
4. Spécifications (Sec. 6.6) ;
5. Invariants Graphiques (Sec. 6.7) ;
6. Construction du code des modules (Sec. 6.8) ;
7. Code final (Sec. 6.9) ;

6.3 Definition 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 6.3.3
Si la notion de définition d'un problème, voyez la Section 6.3.1
Si vous ne voyez pas comment faire, reportez-vous à l'indice 6.3.2

6.3.1 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

$n \in \mathbb{N}$ [4]

```
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
```

4. 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. 6.3.3.

6.3.2 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. 6.3.3.

6.3.3 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) :

> Simon souhaite disposer d'une énumération des nombres *narcissiques*. Un nombre entier positif est dit narcissique si il est égal à somme de chiffres qui le composent, chacun étant élevé à la puissance correspondant à la quantité de chiffres dans le nombre de base. Simon voudrait afficher sur la sortie standard tous les nombres narcissiques entre 1 et x.

On dispose donc en entrée d'une valeur, `x`, donnant la borne supérieure d'un intervalle de recherche (Input). L'objectif est d'afficher tous les nombres narcissiques dans l'intervalle [1, `x`] (Output).

La définition du problème est donc

Input : `x`, la borne supérieure de l'intervalle de recherche.

Output : affichage de tous les nombres narcissiques dans [1, `x`]

Objet Utilisé :

`x`, la borne supérieure

$x \in \mathbb{N}$

```
unsigned int x;
```

Cette définition est tout à fait cohérente avec le canevas de code déjà donné dans l'énoncé.

Nous pouvons maintenant passer à l'analyse et découpe en Sp du problème. Voir Sec. 6.4.

6.4 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 6.4.3
Si vous ne savez pas ce qu'est l'analyse du problème, allez à la Section 6.4.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 6.4.2

6.4.1 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. *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. 6.4.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 6.4.2

6.4.2 Indice

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

Vous devez aussi vous approprier la définition d'un nombre narcissique. Cette définition donne des indices importants sur la façon de déterminer si un nombre est narcissique, ce qui peut impliquer un ou plusieurs SP intéressant(s).

Vous devez, bien entendu, veiller à ce que votre découpe soit pertinente : ni trop haut niveau (e.g., SP_1 : résoudre le problème), ni trop bas niveau (e.g., SP_1 déclarer des variables). Dans tous les cas, après exécution du dernier SP, le problème général doit être résolu (i.e., vous devez avoir atteint l'Output décrit dans la définition).

Suite de l'Exercice

À vous ! Analysez le problème et passez à la Sec. 6.4.3.

6.4.3 Mise en Commun de l'Analyse du Problème

La Découpe en SP est ici un peu plus compliquée que celle du GameCode du Chapitre 5.

Première Approche (à la grosse louche)

Repartons de l'énoncé pour essayer d'inférer les grandes étapes de notre programme :

> Simon souhaite disposer d'une énumération des nombres *narcissiques*. Un nombre entier positif est dit narcissique si il est égal à somme de chiffres qui le composent, chacun étant élevé à la puissance correspondant à la quantité de chiffres dans le nombre de base. Simon voudrait afficher sur la sortie standard tous les nombres narcissiques entre 1 et x.

Sur base de l'énoncé et des éléments mis en évidence, il est très aisé de venir avec une première découpe, quelque peu inefficace, en SP :

SP$_1$: énumération des nombres dans l'intervalle $[1, x]$ et affichage des nombres narcissiques.

SP$_2$: déterminer si un nombre `i` est narcissique.

On voit bien que l'enchaînement sera le suivant :

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

Dans l'absolu, ce n'est pas une mauvaise découpe mais le SP$_2$ apparaît encore trop large. En effet, déterminer si un nombre donné est narcissique ou pas est assez complexe et nécessite d'effectuer plusieurs étapes. On peut donc raffiner le SP$_2$.

Deuxième Approche (déterminer un nombre narcissique)

Repartons de la définition d'un nombre narcissique :

> Un nombre entier positif est dit narcissique si il est égal à somme de chiffres qui le composent, chacun étant élevé à la puissance correspondant à la quantité de chiffres dans le nombre de base.

Cette définition nous indique clairement comment on peut déterminer si un nombre est narcissique ou pas. En particulier, et comme mis en évidence, il faut

1. pouvoir déterminer la "taille" d'un nombre. Cette taille est exprimée sous la forme de la quantité de chiffres qui le compose. Par exemple, pour le nombre 8208, sa taille sera de 4 car il est composé de 4 chiffres.
2. pouvoir élevé à une certaine puissance des chiffres. Il faut donc pouvoir calculer a^b, de manière générale (pour rappel, les contraintes de l'énoncé indique que nous ne pouvons utiliser aucune librairie autre que `stdio.h`)

Sur base de cela, on peut déterminer deux autres SP

SP$_{2.a}$: déterminer le nombre de chiffres dans un nombre donné `i`

SP$_{2.b}$: calculer a^b

Il n'y a pas de relation directe entre les 2 SP mais ils font partie du SP$_2$

Troisième Approche (raffinement d'un nombre narcissique)

A nouveau, repartons de la définition d'un nombre narcissique et voyons ce qu'il nous manque :

> Un nombre entier positif est dit narcissique si il est égal à somme de chiffres qui le composent, chacun étant élevé à la puissance correspondant à la quantité de chiffres dans le nombre de base.

Non seulement, il faut connaitre la taille du nombre considéré (SP$_{2.a}$) mais il faut en plus sommer ses chiffres, chacun élevé à la puissance correspondant à la taille du nombre (SP$_{2.b}$). Il faut donc décomposer le nombre en ses différents chiffres et les sommer après les avoir portés à la puissance. Ceci nous donne donc le SP suivant :

SP$_{2.c}$: décomposer un nombre en chiffres et calculer la somme des puissances

Découpe Finale (tous ensemble)

On dispose, au final, des Sp suivants :

Sp$_1$: énumération des nombres dans l'intervalle $[1, x]$ et affichage des nombres narcissiques.

Sp$_{2.a}$: déterminer le nombre de chiffres dans un nombre donné i

Sp$_{2.b}$: calculer a^b

Sp$_{2.c}$: décomposer un nombre i en chiffres et calculer la somme des puissances

L'enchaînement est le suivant (sachant que les Sp$_{2.a}$, Sp$_{2.b}$ et Sp$_{2.c}$ sont une décomposition du Sp$_2$)

$$\left[\left(\text{Sp}_{2.a} \to (\text{Sp}_{2.b} \subset \text{Sp}_{2.c})\right) \subset \text{Sp}_1\right]$$

Nous pouvons maintenant passer à l'architecture globale du code et aux relations entre les différents fichiers de code source. Voir Sec. 6.5.

6.5 Architecture du Code

Dans cette section, nous allons nous pencher sur l'architecture générale du code, en particulier dans le cadre d'une compilation séparée.

Si vous voyez de quoi on parle, rendez-vous à la Section 6.5.2
Si vous ne savez pas ce qu'est la compilation séparée, allez à la Section 6.1.3
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 6.5.1

6.5.1 Indice

Le principe de la compilation séparée implique une déclaration explicite des modules. Et donc une découpe du code en header (pour les prototypes) et en fichiers `.c` (pour les prototypes et corps).

L'idée est donc de regrouper sous un même chapeau l'ensemble des fonctionnalités (i.e., modules) permettant de résoudre un même problème. Les prototypes iront dans le header, les prototypes et corps dans le `.c` associé.

De manière générale, header et `.c` portent le même nom.

On peut ensuite schématiser l'ensemble en indiquant les relations entre les différents éléments (cfr. Chapitre 6, Slide 38).

Suite de l'Exercice

À vous ! Proposez une architecture de code et passez à la Sec. 6.5.2.

6.5.2 Mise en Commun de l'Architecture du Code

Le problème que nous devons résoudre porte sur les nombres narcissiques. Il serait donc normal d'avoir une "librairie" dédiée à ces nombres. Appelons là, naturellement, *Narcissique*. On pourrait alors avoir la découpe du code suivante :

`narcissique.h` : c'est le header qui va contenir l'interface des modules (cfr. les spécifications).

`narcissique.c` : c'est le fichier qui va contenir l'implémentation des modules définis dans le header.

On peut ensuite considérer que le code à compléter se trouve dans le fichier source `main-narcissique.c`.

Avec ces trois fichiers, on peut déterminer maintenant l'architecture générale du code et l'interaction entre les différents fichiers. Soit

Nous pouvons maintenant passer à l'interface de chaque Sp (i.e., signature et spécifications des modules). Voir Sec. 6.6.

6.6 Spécifications

Il s'agit, maintenant, de transformer chaque Sp sous la forme d'un module. Et donc de spécifier chacun des modules.

Si vous voyez de quoi on parle, rendez-vous à la Section 6.6.3
Si vous ne savez pas ce qu'est une spécification, allez à la Section 6.6.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 6.6.2

6.6.1 Rappels sur les Spécifications

6.6.1.1 Généralités

La *programmation par contrat* permet de généraliser la notion de définition d'un problème en déterminant les

suppositions sur les arguments en entrée d'un module. Il s'agit donc de préciser les propriétés que doivent respecter les données en entrée (i.e., paramètres effectifs) d'un module. Les suppositions portent sur l'Input.

certifications après exécution du module. Il s'agit de décrire le résultat du module. Elles portent donc sur l'Output.

Il est possible d'implémenter la programmation par contrat à l'aide des *spécifications*, i.e., un contrat logiciel qui lie le programmeur d'un module (i.e., celui qui a écrit le code du module) et l'utilisateur du module (i.e., celui qui l'invoque).

Une spécification se définit en deux temps :

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

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

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

La spécification d'un module se met, en commentaires, avec le prototype du module. L'ensemble (i.e., spécification + prototype) forme l'*interface* du module. Typiquement, dans le cadre de la compilation séparée, l'interface du module se placera dans le header. L'implémentation du module (sans rappeler la spécification) sera dans le fichier `.c`.

L'idée dernière la notion d'interface est de permettre à l'utilisateur du module (i.e., celui qui va l'invoquer) d'avoir toutes les cartes en main. En particulier, comment invoquer le module (i.e., le prototype) et sous quelle(s) contrainte(s) (i.e., la PréCondition). Dans ce cas, le programmeur garantit que le résultat du module sera celui décrit dans la PostCondition.

6.6.1.2 Construction d'une Interface

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

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

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

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

Question 3 : Quels sont les contraintes sur les paramètres ? Répondre à cette question permet d'obtenir la PréCondition.

6.6.1.3 Exemple

Pour illustrer la notion de spécification, travaillons sur un exemple. Il s'agit, ici, de calculer la somme des éléments dans un intervalle d'un tableau contenant des valeurs entières.

La construction de la spécification se fait en répondant aux trois questions. Le moyen le plus simple pour y arriver, c'est de s'appuyer sur un dessin qui, bien entendu, représente le problème. Plus le dessin sera précis, plus il sera facile de dériver un Invariant de Boucle (si le problème, bien entendu, le nécessite).

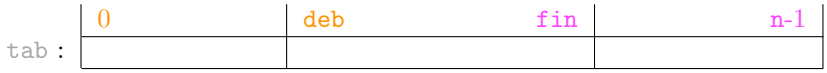
Objets nécessaires (Question 1) Il s'agit, ici, de se demander ce dont on a besoin pour résoudre le problème. Dit autrement, avec quel(s) objet(s) on part. Il faut, bien entendu, repartir de l'énoncé et construire un dessin mettant en exergue les objets. Prenons donc l'énoncé et faisons ressortir les éléments importants :

calculer la somme des éléments dans un intervalle d'un tableau contenant des valeurs entières.

Le premier point important, dans l'énoncé, c'est la notion de tableau. Il s'agit d'un tableau contenant des valeurs entières. Naturellement, le tableau vient avec sa taille (appelons là `n`).

La notion d'intervalle veut dire, ici, qu'on s'intéresse à une portion (contigüe) du tableau. Cette portion est définie, bien entendu, entre les bornes du tableau. Appelons la borne inférieure de l'intervalle `deb` (pour "début") et la borne supérieure `fin`.

On arrive donc au dessin suivant :



La taille du tableau (`n`) est par définition une valeur entière non négative. Son type sera donc `unsigned int`. Les bornes de l'intervalle d'intérêt (`deb` et `fin`) seront du même type.

Toutes les variables que nous venons d'identifier vont servir de paramètres formels au module. Il reste encore à trouver un identificateur pour le module. On peut simplement choisir de l'appeler `somme_tableau`.

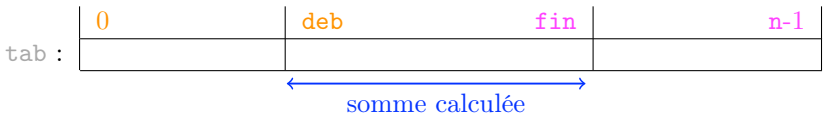
Il vient donc le prototype (partiel[5]) suivant :

```
1 somme_tableau(int tab[], unsigned int n, unsigned int deb, unsigned int fin);
```

Objectif du module (Question 2) Il s'agit, ici, de se demander ce que doit faire, comme travail, le module. Il faut, bien entendu, repartir de l'énoncé et adapter le dessin entamé à la Question 1. Prenons donc l'énoncé et faisons ressortir les éléments importants :

calculer la somme des éléments dans un intervalle d'un tableau contenant des valeurs entières.

Il s'agit donc d'additionner toutes les valeurs dans l'intervalle d'intérêt (i.e., [deb, fin]) du tableau. On peut donc compléter le schéma de la façon suivante :



Comme les éléments du tableau sont de type `int`, le résultat du calcul (et donc le type de retour du module) sera aussi de type `int`.

On peut compléter le prototype en indiquant le type de retour et la PostCondition :

5. La réponse à la Question 2 permettra de déterminer l'éventuel type de retour

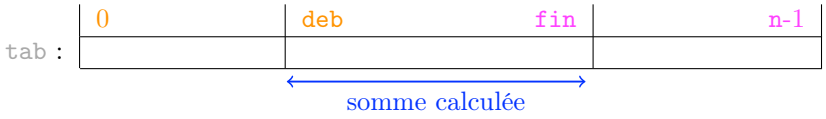
```c
/*
 * PostCondition : somme_tableau vaut la somme des entiers dans l'intervalle [deb, fin].
 */
int somme_tableau(int tab[], unsigned int n, unsigned int deb, unsigned int fin);
```

Alerte : Complétude !

Pour rappel, dans une PostCondition, si le module est une fonction, il faut indiquer que l'identificateur de la fonction vaut le résultat. C'est tout à fait normal puisque, après l'invocation, la fonction est remplacée par son évaluation, soit la valeur retournée.
C'est bien ce que nous faisons ici en indiquant `somme_tableau vaut ...`

Contraintes (Question 3)

Pour répondre à cette dernière question, il faut repartir du dessin obtenu à la Question 2 et observer les relations entre les variables identifées comme étant des paramètres formels. Soit le dessin :



Les variables et leurs contraintes sont les suivantes :

`tab` : le tableau doit exister et contenir des valeurs. Cela se traduit, dans une spécification, par `tab` $\neq$ `NULL`.

`n` : puisque le tableau existe et contient des valeurs, la taille du tableau ne peut pas valoir 0. On obtient donc `n` > 0.

`deb` et `fin` : le dessin est assez clair pour ces variables. L'intervalle [`deb`, `fin`] se trouve dans les bornes du tableau, [0, `n`-1]. On obtient donc naturellement la contrainte suivante : $0 \leq$ `deb` $\leq$ `fin` $\leq$ `n` $- 1$.

Interface Finale

Extrait de code 6 – Interface (fichier `.h`)

```c
/*
 * \precondH: tab ≠ NULL, 0 ≤ deb ≤ fin ≤ n − 1
 * \postcondH: somme_tableau vaut la somme des entiers dans l'intervalle [deb, fin]
 */
int somme_tableau(int tab[], unsigned int n, unsigned int deb, unsigned int fin);
```

Suite de l'Exercice

À vous ! Proposez les interfaces pour les Sp et passez à la Sec. 6.6.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 6.6.2

6.6.2 Indice

Sur base des différents Sp, identifiez ceux que vous pouvez raisonnablement externaliser dans un module.

Pour chaque module, répondez aux trois questions en vous appuyant sur un schéma (schéma qui vous sera utile pour les Invariants Graphiques).

Ne soyez pas ambigu dans la PréCondition, ni dans la PostCondition. Au contraire, soyez le plus précis possible (rappelez-vous, c'est un contrat).

Une fois que c'est fait, rédigez l'interface de chaque module dans le fichier `narcissique.h` (cfr. Architecture du code).

Suite de l'Exercice

À vous ! Analysez le problème et passez à la Sec. 6.6.3.

6.6.3 Mise en Commun des Spécifications

Pour rappel, les SP sont les suivants :

SP$_1$: énumération des nombres dans l'intervalle $[1, x]$ et affichage des nombres narcissiques.

SP$_{2.a}$: déterminer le nombre de chiffres dans un nombre donné i

SP$_{2.b}$: calculer a^b

SP$_{2.c}$: décomposer un nombre i en chiffres et calculer la somme des puissances

et l'enchaînement est le suivant :

$$\left[\left(\text{SP}_{2.a} \rightarrow (\text{SP}_{2.b} \subset \text{SP}_{2.c})\right) \subset \text{SP}_1\right]$$

Au vu de cela, il semble évident que chaque SP pourra être externaliser dans un module. Il faut, cependant, faire attention au SP$_{2.c}$ qui, comme l'indique l'enchaînement, utilisera les SP$_{2.a}$ et SP$_{2.b}$.

- Spécifications du SP$_1$;
- Spécifications du SP$_{2a}$;
- Spécifications du SP$_{2b}$;
- Spécifications du SP$_{2c}$.

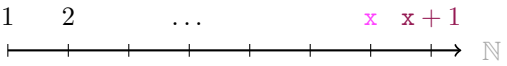
6.6.3.1 SP_1

Le SP_1 s'intéresse à l'énumération des nombres dans l'intervalle $[1, x]$ et à l'affichage des nombres narcissiques. Pour construire l'interface du module implémentant le SP_1, appliquons la méthode :

Objets Nécessaires (Question 1) : Repartons de la description du SP :

énumération des nombres dans l'intervalle $[1, x]$ et affichage des nombres narcissiques.

Puisqu'il s'agit d'une énumération de valeurs dans l'intervalle $[1, x]$, on peut facilement proposer le dessin suivant :



La droite de valeurs est celle des naturels car les nombres narcissiques ne concernent que les nombres entiers positifs (cfr. énoncé). La variable `x` est la borne supérieure de l'intervalle. Elle sera donc de type `unsigned int`.

La variable `x` va servir de paramètre formel au module. Il nous reste à trouver un identificateur pour le module. On peut simplement choisir de l'appeler `affiche_narcissique`.

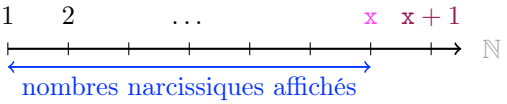
Il vient donc le prototype (partiel) suivant :

```
1    affiche_narcissique(unsigned int x);
```

Objectif du Module (Question 2) : il faut, bien entendu, repartir de la description du SP_1 pour en faire ressortir le(s) élément(s) important(s) et adapter le dessin entamé à la Question 1. Soit :

énumération des nombres dans l'intervalle $[1, x]$ et affichage des nombres narcissiques.

L'objectif du module est donc bien l'affichage des nombres narcissiques dans l'intervalle d'intérêt (c'est donc la PostCondition). On peut donc compléter le schéma de la façon suivante :

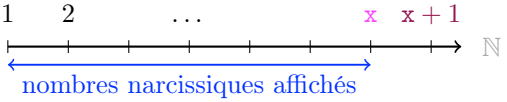


Le module ne va donc retourner aucun résultat. C'est donc une procédure.

On dès lors peut compléter le prototype en indiquant le type de retour et la PostCondition :

```
1  /*
2   * PostCondition : Affiche tous les nombres narcissiques dans [1, x].
3   */
4  void affiche_narcissique(unsigned int x);
```

Contraintes (Question 3) : Pour répondre à cette dernière question, il faut repartir du dessin obtenu à la question précédente et observer les relations entre les variables identifées comme étant des paramètres formels. Soit le dessin :



On dipose d'une seule variable, `x`, qui représente la borne supérieure de l'intervalle. On a ici deux possibilités :

1. approche conservatrice. `x` est la borne supérieure de l'intervalle. On exige donc, dans la PréCondition, qu'elle soit plus grande ou égale à la borne inférieure ($\texttt{x} \geq 1$ ou $\texttt{x} > 0$).

2. approche laxiste. On considère que le code que nous allons écrire sera suffisamment robuste pour traiter le cas où `x` == 0.[6] Il n'y a alors pas de PréCondition.

Ici, un peu aléatoirement, nous choisissons l'approche conservatrice.

Au final, on obtient donc l'interface suivante :

```
/*
 * PréCondition : x > 0.
 * PostCondition : Affiche tous les nombres narcissiques dans [1, x].
 */
void affiche_narcissique(unsigned int x);
```

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

- Spécifications du SP_{2a} ;
- Spécifications du SP_{2b} ;
- Spécifications du SP_{2c} ;
- Invariants Graphiques des modules.

6. Ce qui devrait de toute façon être le cas, quelque soit l'approche choisie.

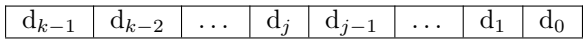
6.6.3.2 SP_{2a}

Le $\text{SP}_{2.a}$ permet de déterminer le nombre de chiffres dans un nombre donné. Pour construire l'interface du module implémentant le $\text{SP}_{2.a}$, appliquons la méthode :

Objets Nécessaires (Question 1) : Repartons de la description du SP :

> déterminer le nombre de chiffres dans un nombre donné.

On s'appuie donc sur un nombre (nécessairement un entier positif, vu le contexte dans lequel on travaille). Appelons ce nombre `x`. On peut alors facilement proposer le dessin suivant :



Dans ce schéma, chaque valeur d_i (avec $0 \leq i \leq k-1$) représente un symbole dans $\{0, \ldots, 9\}$. Le nombre `x` est donc représenté comme suit : $(d_{k-1}d_{k-2} \ldots d_j \ldots d_1 d_0)_{10}$ [7].

La variable `x` va servir de paramètre formel au module. Il nous reste à trouver un identificateur pour le module. On peut simplement choisir de l'appeler `nb_chiffres`.

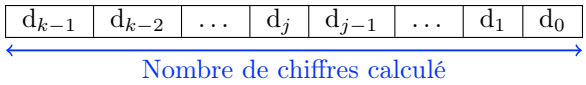
Il vient donc le prototype (partiel) suivant :

```
1  nb_chiffres(unsigned int x);
```

Objectif du Module (Question 2) : Il faut, bien entendu, repartir de la description du SP_1 pour en faire ressortir le(s) élément(s) important(s) et adapter le dessin entamé à la Question 1. Soit :

> déterminer le nombre de chiffres dans un nombre donné.

L'objectif du module est de retourner (c'est donc une fonction) le nombre de chiffres qui composent le nombre en entrée (c'est donc la PostCondition). On peut donc compléter le schéma de la façon suivante :

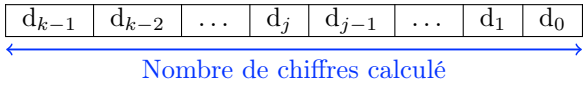


Le module retourne donc un résultat correspondant au nombre de chiffres dans `x`. C'est donc une fonction. Ce résultat ne peut être négatif et, dès lors, le type de retour sera `unsigned int`.

On peut dès lors compléter le prototype en indiquant le type de retour et la PostCondition :

```
1  /*
2   * PostCondition : nb_chiffres vaut le nombre de chiffres dans x.
3   */
4  unsigned int nb_chiffres(unsigned int x);
```

Contraintes (Question 3) : Pour répondre à cette dernière question, il faut repartir du dessin obtenu à la question précédente et observer les relations entre les variables identifées comme étant des paramètres formels. Soit le dessin :



Il y a ici une seule variable, `x`, sur laquelle nous ne devons mettre aucune contrainte particulière. Notre code doit fonctionner sur n'importe quel nombre naturel. Il n'y a donc pas de PréCondition.

7. Cette représentation a été introduite dans l'Introduction, Slides 35 → 36.

Au final, on obtient donc l'interface suivante :

```
/*
 * PréCondition : /
 * PostCondition : nb_chiffres vaut le nombre de chiffres dans x.
 */
unsigned int nb_chiffres(unsigned int x);
```

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

- Spécifications du SP_1 ;
- Spécifications du SP_{2b} ;
- Spécifications du SP_{2c} ;
- Invariants Graphiques des modules.

6.6.3.3 SP$_{2b}$

Le SP$_{2.b}$ permet de calculer a^b. Pour construire l'interface du module implémentant le SP$_{2.b}$, appliquons la méthode :

Objets Nécessaires (Question 1) : Repartons de la description du SP :

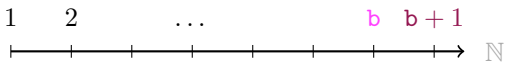
> calculer a^b.

On s'appuie donc sur deux nombres, la base et l'exposant. Etant donné le contexte dans lequel on travaille, la base et l'exposant seront nécessairement des entiers positifs. Appelons les `a` (la base) et `b` (l'exposant).

Pour le dessin, il faut déjà avoir une idée de la façon dont on va effectuer le calcul. Comme ne nous pouvons pas utiliser de librairies externes (autre que `stdio.h` ou `assert.h`), nous devons effectuer le calcul nous-même. Pour ce faire, on s'appuie sur la proporiété suivante :

$$a^b = \underbrace{a \times a \times \ldots \times a}_{\text{b times}}.$$

De là, on tire le dessin suivant :



La variable `a` n'est pour le moment pas représentée sur le dessin car elle ne donne aucune information relative au dimensionnement du problème. Elle n'intervient que dans le calcul itératif.

Les variables `a` et `b` vont servir de paramètre formel au module. Il nous reste à trouver un identificateur pour le module. On peut simplement choisir de l'appeler `puissance`.

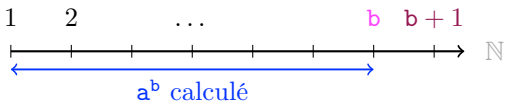
Il vient donc le prototype (partiel) suivant :

```
1  puissance(unsigned int a, unsigned int b);
```

Objectif du Module (Question 2) : il faut, bien entendu, repartir de la description du SP$_{2.b}$ pour en faire ressortir le(s) élément(s) important(s) et adapter le dessin entamé à la Question 1. Soit :

> calculer a^b.

L'objectif du module est donc bien d'effectuer un calcul et retourner le résultat du calcul au code appelant. Ce calcul donne le résultat de a^b (c'est donc la POSTCONDITION). On peut donc compléter le schéma de la façon suivante :

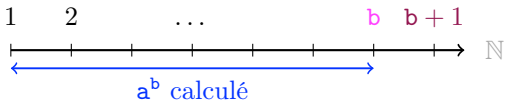


Le module retourne un résultat. Ce sera donc une fonction. Le type de retour sera naturellement `unsigned int` puisque `a` et `b` sont eux-mêmes de type `unsigned int`.

On dès lors peut compléter le prototype en indiquant le type de retour et la POSTCONDITION :

```
1  /*
2   * PostCondition : puissance vaut a^b.
3   */
4  unsigned int puissance(unsigned int a, unsigned int b);
```

Contraintes (Question 3) : Pour répondre à cette dernière question, il faut repartir du dessin obtenu à la question précédente et observer les relations entre les variables identifées comme étant des paramètres formels. Soit le dessin :



Le dessin montre clairement que $\texttt{b} \geq 1$. Un peu à l'instar du SP_1, on pourrait au choix adopter une approche conservatrice (i.e., la PréCondition indique que $\texttt{b} \geq 1$) ou bien une approche laxiste (i.e., pas de PréCondition car le code devrait prendre en compte le calcul de a^0). Cependant, étant donné le contexte dans lequel on travaille, le cas $\texttt{b} = 0$ n'a aucun sens (un nombre est composé d'au moins un chiffre). Nous optons donc pour l'approche conservatrice.

Au final, on obtient donc l'interface suivante :

```
/*
 * PréCondition : b ≥ 1.
 * PostCondition : puissance vaut a^b.
 */
unsigned int puissance(unsigned int a, unsigned int b);
```

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

- Spécifications du SP_1 ;
- Spécifications du SP_{2a} ;
- Spécifications du SP_{2c} ;
- Invariants Graphiques des modules.

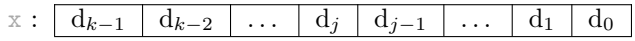
6.6.3.4 SP_{2c}

Le $\text{SP}_{2.c}$ permet de décomposer un nombre en chiffres et de calculer la somme des puissances. Si la somme vaut le nombre de départ, alors ce nombre est dit narcissique. Pour construire l'interface du module implémentant le $\text{SP}_{2.c}$, appliquons la méthode :

Objets Nécessaires (Question 1) : Repartons de la description du SP :

> décomposer un nombre en chiffres et de calculer la somme des puissances. Si la somme vaut le nombre de départ, alors ce nombre est dit narcissique.

Il faut être prudent dans le traitement du $\text{SP}_{2.c}$ car il englobe/utilise les $\text{SP}_{2.a}$ et $\text{SP}_{2.b}$. Ici, le seul objet utilisé est le nombre de départ pour lequel on va calculer sa taille ($\text{SP}_{2.a}$) et calculer une somme à l'aide de puissances ($\text{SP}_{2.b}$). Appelons ce nombre `x`. Nous avons alors le dessin suivant :



Dans ce schéma, chaque valeur d_i (avec $0 \leq i \leq k-1$) représente un symbole dans $\{0, \ldots, 9\}$. Le nombre `x` est donc représenté comme suit : $(d_{k-1}d_{k-2} \ldots d_j \ldots d_1d_0)_{10}$ [8].

La variable `x` va servir de paramètre formel au module. Comme les nombres narcissiques sont forcément des nombres entiers positifs, son type sera `unsigned int`. Il nous reste à trouver un identificateur pour le module. On peut simplement choisir de l'appeler `est_narcissique`.

Il vient donc le prototype (partiel) suivant :

```
1  est_narcissique(unsigned int x);
```

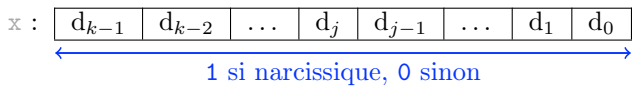
Objectif du Module (Question 2) : Il faut, bien entendu, repartir de la description du SP_1 pour en faire ressortir le(s) élément(s) important(s) et adapter le dessin entamé à la Question 1. Soit :

> décomposer un nombre en chiffres et de calculer la somme des puissances. Si la somme vaut le nombre de départ, alors ce nombre est dit narcissique.

L'objectif du module est donc de déterminer si un nombre donné respecte une certaine propriété (i.e., SP de vérification), à savoir si le nombre est narcissique ou pas. Il ne nous importe pas ici de savoir comment détermine le respect de cette propriété La POSTCONDITION doit juste décrire le résutlat, pas comment on l'obtient.

Dès lors, notre module devra retourner un résultat (c'est donc une fonction) à sémantique booléenne. Pour rappel, le type booléen n'existe pas en tant que tel mais dépend d'une interprétation différente d'un nombre entier : la valeur `0` est interpétée comme *faux*, tout autre valeur entière comme *vrai*. Pour rester cohérent avec la représentation booléenne en algèbre et en électronique, nous décidons que la fonction va retourner `1` si le nombre en entrée est un nombre narcissique, `0` sinon (c'est donc la POSTCONDITION). Cependant, pour rester cohérent avec la notion de booléen en C, le type de retour de la fonction sera `int`.

On peut donc compléter le schéma de la façon suivante :



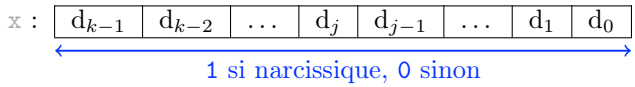
Le module retourne donc un résultat correspondant à `1` si `x` est un nombre narcissique, `0` sinon. C'est donc une fonction. Ce résultat peut être `0` ou `1`, dès lors, le type de retour sera `int`.

Il ne nous reste plus qu'à compléter le prototype en indiquant le type de retour et la POSTCONDITION :

8. Cette représentation a été introduite dans l'Introduction, Slides 35 → 36.

```c
/*
 * PostCondition : est_narcissique vaut 1 si x est un nombre narcissique.
 *                 0 sinon.
 */
int est_narcissique(unsigned int x);
```

Contraintes (Question 3) : Pour répondre à cette dernière question, il faut repartir du dessin obtenu à la question précédente et observer les relations entre les variables identifées comme étant des paramètres formels. Soit le dessin :



Il y a ici une seule variable, `x`, sur laquelle nous ne devons mettre aucune contrainte particulière. Notre code doit fonctionner sur n'importe quel nombre naturel. Il n'y a donc pas de PréCondition.

Au final, on obtient donc l'interface suivante :

```c
/*
 *
 * PréCondition : /
 * PostCondition : est_narcissique vaut 1 si x est un nombre narcissique.
 *                 0 sinon.
 */
int est_narcissique(unsigned int x);
```

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

- Spécifications du SP_1 ;
- Spécifications du SP_{2a} ;
- Spécifications du SP_{2b} ;
- Invariants Graphiques des modules.

6.6.3.5 Mise en Commun des Interfaces

On peut maintenant mettre en commun toutes les interfaces dans le header. Il vient :

Extrait de code 7 – Fichier `narcissique.h`

```
/*
 * PréCondition : x > 0.
 * PostCondition : Affiche tous les nombres narcissiques dans [1, x].
 */
void affiche_narcissique(unsigned int x);

/*
 * PréCondition : /
 * PostCondition : nb_chiffres vaut le nombre de chiffres dans x.
 */
unsigned int nb_chiffres(unsigned int x);

/*
 * PréCondition : b > 1.
 * PostCondition : puissance vaut a^b.
 */
unsigned int puissance(unsigned int a, unsigned int b);

/*
 * PréCondition : /
 * PostCondition : est_narcissique vaut 1 si x est un nombre narcissique.
 *                 0 sinon.
 */
int est_narcissique(unsigned int x);
```

Nous pouvons maintenant passer aux Invariants Graphiques. Voir Sec. 6.7.

6.7 Invariants Graphiques

Si jamais l'écriture implique un traitement itératif, vous devez, au préalable, établir un Invariant Graphique qui vous permettra, ensuite, de construire votre code. Cette section discute donc des différents Invariants Graphiques.

Si vous voyez de quoi on parle, rendez-vous à la Section 6.7.3
Si vous ne savez pas ce qu'est un Invariant Graphique, allez la Section 6.7.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 6.7.2

6.7.1 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

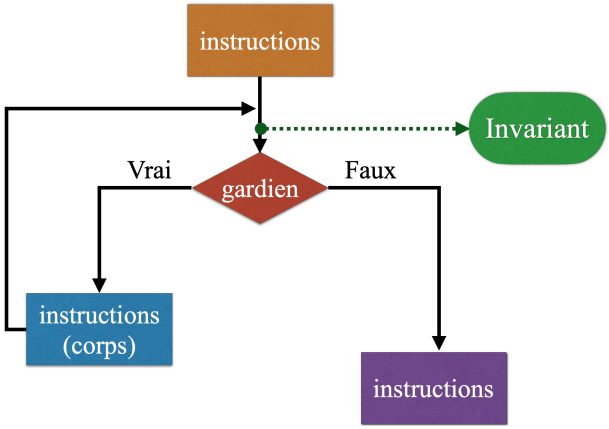


FIGURE 1 – Sémantique de l'Invariant Graphique.

La Fig. 1 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. 1, 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. 2 montre l'Invariant Graphique correspondant.

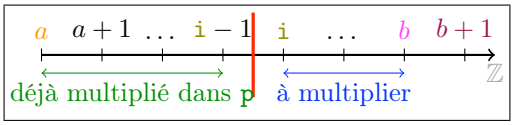


FIGURE 2 – Exemple d'Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

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. 2) et le nommer ($\mathbb{Z}$ dans la Fig. 2) ;

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

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

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

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

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

Règle 7 : Toutes les structures identifiées et variables sont présentes dans le code (`a`, `b`, `i`, et `p` sur la Fig. 2) [9].

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 les Invariants Graphiques des SP :

- SP_1 ;
- SP_{2a} ;
- SP_{2b} ;
- SP_{2c}.

Si vous séchez, reportez-vous à l'indice à la Sec. 6.7.2

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

6.7.2 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 :

- repartez du dessin que vous avez créé pour élaborer l'interface du module. Ce dessin représente en fait la situation finale (i.e., situation grâce à laquelle vous dérivez la Zone 3).
- il suffit donc de voir ce qui manque au dessin pour suivre scrupuleusement les règles 1 $\rightarrow$ 7 pour la construction de votre Invariant Graphique. Normalement, les règles 1 et 2 sont déjà présentes. Il vous reste donc à ajouter la ligne de démarcation (Règle 3) et la variable d'itération (Règle 4 de la construction d'un Invariant Graphique).
- repartez de la formulation de la PostCondition afin de donner du sens à votre dessin (i.e., sémantique – Règle 5 et Règle 6 de la construction d'un Invariant Graphique).

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

Suite de l'Exercice

À vous ! Rédigez les Invariants Graphiques des Sp :

- Sp_1 ;
- Sp_{2a} ;
- Sp_{2b} ;
- Sp_{2c}.

6.7.3 Mise en Commun de l'Invariant Graphique

Pour rappel, les Sp sont les suivants :

Sp$_1$: énumération des nombres dans l'intervalle $[1, x]$ et affichage des nombres narcissiques.

Sp$_{2.a}$: déterminer le nombre de chiffres dans un nombre donné i

Sp$_{2.b}$: calculer a^b

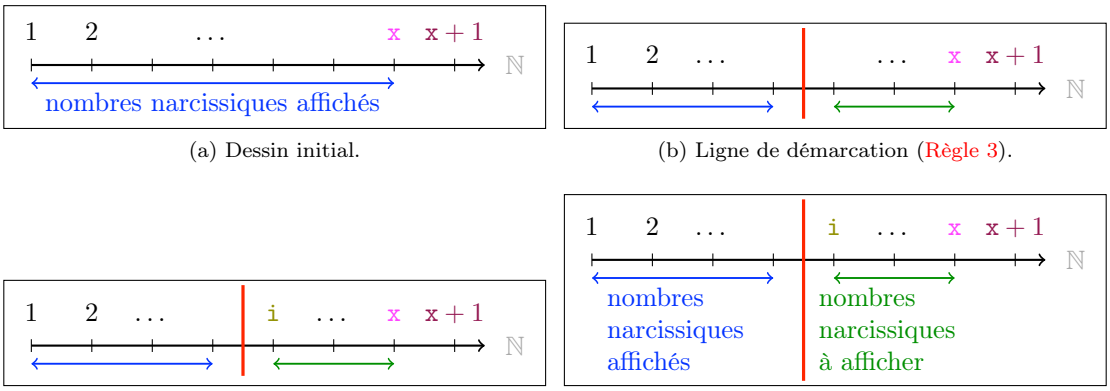
Sp$_{2.c}$: décomposer un nombre i en chiffres et calculer la somme des puissances

Chacun de ces Sp nécessitent un traitement itératif, et donc un Invariant Graphique.

- Invariant Graphique du Sp$_1$;
- Invariant Graphique du Sp$_{2a}$;
- Invariant Graphique du Sp$_{2b}$;
- Invariant Graphique du Sp$_{2c}$.

6.7.3.1 Module `affiche_narcissique()` (SP_1)

Repartons du dessin effectué lors de la confection de l'interface du SP_1 (voir Fig. 3a)



(a) Dessin initial.

(b) Ligne de démarcation (Règle 3).

(c) Variable d'itération (Règle 4).

(d) Sémantique (Règle 5 et Règle 6).

FIGURE 3 – Construction de l'Invariant Graphique pour le SP_1.

La première étape est d'ajouter la ligne de démarcation. On voit, sur la Fig. 3b que cette action permet déjà de délimiter ce qu'on a déjà fait (flèche bleue) de ce qu'il reste à faire (flèche verte).

Une fois la ligne de démarcation placée, on peut ajouter la variable d'itération. Appelons la `i`. Comme indiqué sur la Fig. 3c, il est préférable de la placer à droite de la ligne de démarcation (et donc de l'exclure de la zone déjà traitée).

A ce stade, les aspects syntaxiques de l'Invariant Graphique sont remplis. Il nous reste donc à veiller à la sémantique. Pour cela, il suffit de repartir de la POSTCONDITION et de l'appliquer à la zone déjà traitée (flèche bleue) et encore à traiter (flèche verte). C'est illustré à la Fig. 3d.

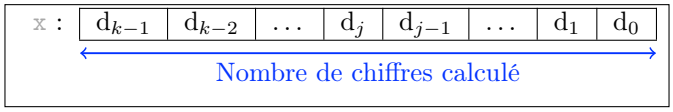
Suite de l'exercice

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

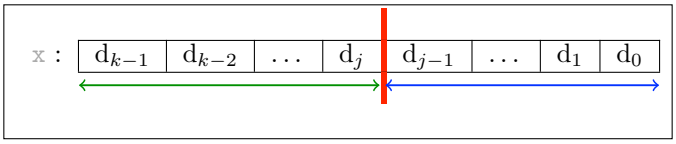
- Invariant Graphique SP_{2a} ;
- Invariant Graphique SP_{2b} ;
- Invariant Graphique SP_{2c} ;
- construction du code des modules .

6.7.3.2 Module `nb_chiffres()` (SP$_{2.a}$)

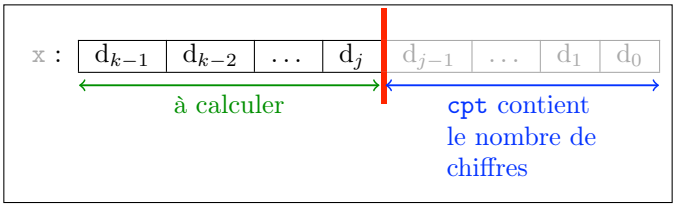
Repartons du dessin effectué lors de la confection de l'interface du SP$_{2.a}$ (voir Fig. 4a).



(a) Dessin initial.



(b) Ligne de démaracation (Règle 3).



(c) Sémantique (Règle 5 et Règle 6).

FIGURE 4 – Construction de l'Invariant Graphique pour le SP$_{2a}$.

La première étape est d'ajouter la ligne de démarcation. On voit, sur la Fig. 4b que cette action permet déjà de délimiter ce qu'on a déjà fait (flèche bleue) de ce qu'il reste à faire (flèche verte).

Dans le cas de cet Invariant de Boucle, il n'est pas nécessaire d'avoir explicitement une variable d'itération puisqu'on va réduire `x` chiffre après chiffre (c'est donc `x` qui joue ce rôle). Dans le dessin, la variable indicée j n'est là que pour généraliser le dessin à des nombres comportant n'importe quelle quantité (k) de chiffres.

A ce stade, les aspects syntaxiques de l'Invariant Graphique sont remplis. Il nous reste donc à veiller à la sémantique. Pour cela, il suffit de repartir de la POSTCONDITION et de l'appliquer à la zone déjà traitée (flèche bleue) et encore à traiter (flèche verte). C'est illustré à la Fig. 4c. Vous remarquerez que la partie du nombre correspond à la zone déjà traitée (flèche bleue) est grisée. C'est normal. Cette partie du nombre `x` n'existe plus. A chaque étape, le nombre `x` est amputé d'un chiffre. La zone grisée représente la partie de `x` déjà amputée.

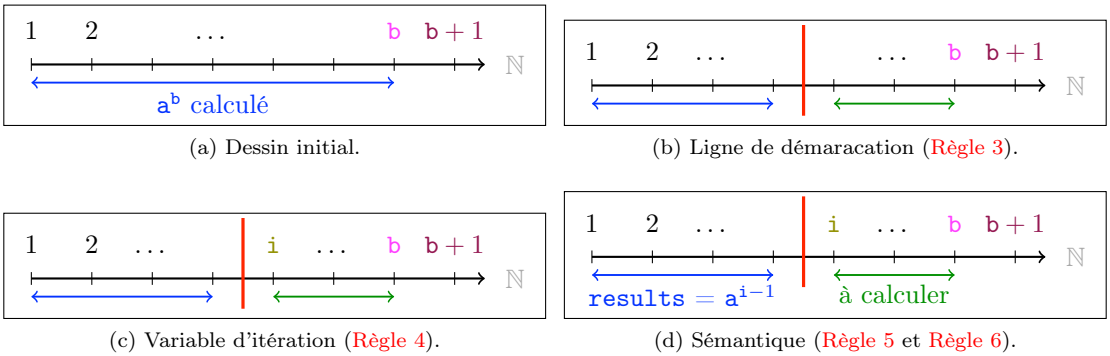
Suite de l'exercice

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

- Invariant Graphique SP$_1$;
- Invariant Graphique SP$_{2b}$;
- Invariant Graphique SP$_{2c}$;
- construction du code des modules .

6.7.3.3 Module `puissance()` ($\text{SP}_{2.b}$)

Repartons du dessin effectué lors de la confection de l'interface du $\text{SP}_{2.b}$ (voir Fig. 5a)



(a) Dessin initial.

(b) Ligne de démaracation (Règle 3).

(c) Variable d'itération (Règle 4).

(d) Sémantique (Règle 5 et Règle 6).

FIGURE 5 – Construction de l'Invariant Graphique pour le SP_{2b}.

La première étape est d'ajouter la ligne de démarcation. On voit, sur la Fig. 5b que cette action permet déjà de délimiter ce qu'on a déjà fait (flèche bleue) de ce qu'il reste à faire (flèche verte).

Une fois la ligne de démarcation placée, on peut ajouter la variable d'itération. Appelons la `i`. Comme indiqué sur la Fig. 5c, il est préférable de la placer à droite de la ligne de démarcation (et donc de l'exclure de la zone déjà traitée).

A ce stade, les aspects syntaxiques de l'Invariant Graphique sont remplis. Il nous reste donc à veiller à la sémantique. Pour cela, il suffit de repartir de la POSTCONDITION et de l'appliquer à la zone déjà traitée (flèche bleue) et encore à traiter (flèche verte). C'est illustré à la Fig. 5d.

Suite de l'exercice

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

- Invariant Graphique SP_1 ;
- Invariant Graphique SP_{2a} ;
- Invariant Graphique SP_{2c} ;
- construction du code des modules .

6.7.3.4 Module `est_narcissique()` (SP$_{2.c}$)

L'élaboration de l'Invariant Graphique pour le SP$_{2.c}$ est un peu plus complexe. Pour bien le construire, il faut repartir de la définition d'un nombre narcissique :

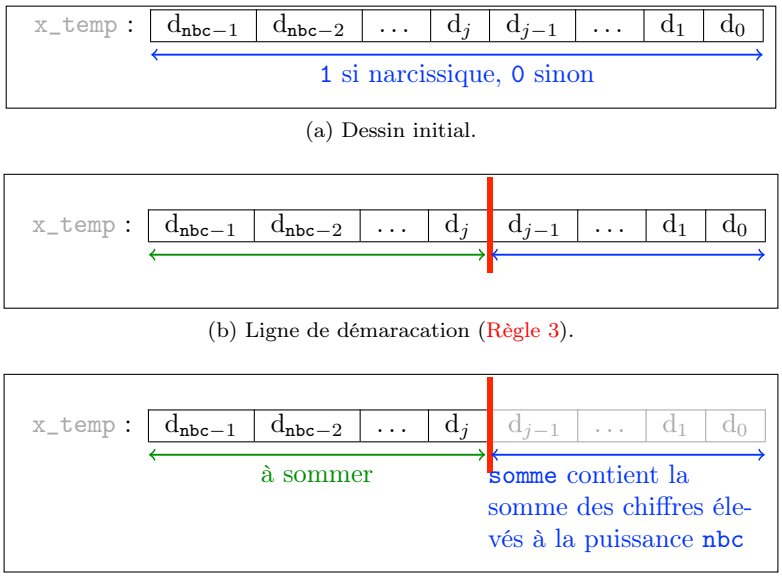
> Un nombre entier positif est dit narcissique si il est égal à somme de chiffres qui le composent, chacun des chiffres étant élevé à la puissance correspondant à la quantité de chiffres dans le nombre de base.
>
> Ainsi, par exemple, 8208 est un nombre narcissique car $8208 = 8^4 + 2^4 + 0^4 + 8^4$.

On voit clairement qu'il faut disposer, dès le début du SP de la quantité de chiffres qui compose le nombre d'intérêt, `x`. Supposons que la variable `nbc` (pour "nombre de chiffres") contienne cette valeur [10].

Il faut ensuite décomposer `x` en ses chiffres et faire une somme cumulative de chacun des chiffres à la puissance `nbc`. Enfin, nous déciderons si `x` est narcissique en le comparant à la somme obtenue. L'objectif de la boucle sera donc de calculer cette somme.

La valeur initiale de `x` doit donc être conservée tout au long du processus. Dès lors, nous allons utiliser une variable temporaire, `x_temp`, qui sera modifiée en cours d'itération.

Repartons du dessin effectué lors de la confection de l'interface du SP$_{2.b}$ et adaptons le avec la variable `nbc` et `x_temp` (voir Fig. 5a).



(a) Dessin initial.

(b) Ligne de démaracation (Règle 3).

(c) Sémantique (Règle 5 et Règle 6).

FIGURE 6 – Construction de l'Invariant Graphique pour le SP$_{2c}$.

Il s'agit maintenant d'ajouter la ligne de démarcation. On voit, sur la Fig. 6b que cette action permet déjà de délimiter ce qu'on a déjà fait (flèche bleue) de ce qu'il reste à faire (flèche verte).

Dans le cas de cet Invariant de Boucle, il n'est pas nécessaire d'avoir explicitement une variable d'itération puisqu'on va réduire `x_temp` chiffre après chiffre (c'est donc `x_temp` qui joue ce rôle). Dans le dessin, la variable indicée j n'est là que pour généraliser le dessin à des nombres comportant n'importe quelle quantité de chiffres (`nbc` peut prendre n'importe quelle valeur strictement positive).

A ce stade, les aspects syntaxiques de l'Invariant Graphique sont remplis. Il nous reste donc à veiller à la sémantique. Pour cela, il suffit de repartir de la POSTCONDITION et de l'appliquer à la zone déjà traitée (flèche bleue) et encore à traiter (flèche verte). C'est illustré à la Fig. 6c. Vous remarquerez que la partie du nombre correspond à la zone déjà traitée (flèche bleue) est grisée. C'est normal. Cette partie du nombre

10. En fait, `nbc` contient le résultat du SP$_{2.a}$.

`x_temp` n'existe plus. A chaque étape, le nombre `x_temp` est amputé d'un chiffre. La zone grisée représente la partie de `x_temp` déjà amputée.

Nous voyons ici que l'Invariant Graphique obtenu est assez éloigné, finalement, de ce qu'on avait dessiné pour la spécification du $\text{SP}_{2.c}$. C'est tout à fait normal et c'est lié à la façon dont on détermine qu'un nombre donné est narcissique ou pas. Cela implique un calcul intermédiaire (i.e., le calcul de la somme) qu'on devra, au final, comparer avec le nombre de départ.

Suite de l'exercice

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

- Invariant Graphique SP_1 ;
- Invariant Graphique SP_{2a} ;
- Invariant Graphique SP_{2b} ;
- construction du code des modules .

Nous pouvons maintenant passer à la construction du code de chaque module. Voir Sec. 6.8.

6.8 Construction des Modules

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 construction des différents modules.

Si vous voyez de quoi on parle, rendez-vous à la Section 6.8.4

Si vous ne voyez pas le lien entre Invariant Graphique et construction du code, reportez-vous à la Section 6.8.1

Si vous ne voyez pas le lien entre spécifications et construction par Invariant de Boucle, allez à la Section 6.8.2

Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 6.8.3

6.8.1 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 stratégie [11] de résolution du problème. Cette stratégie permet d'identifier quatre points particuliers : trois Zones (voir la Fig. 7) et le Critère d'Arrêt.

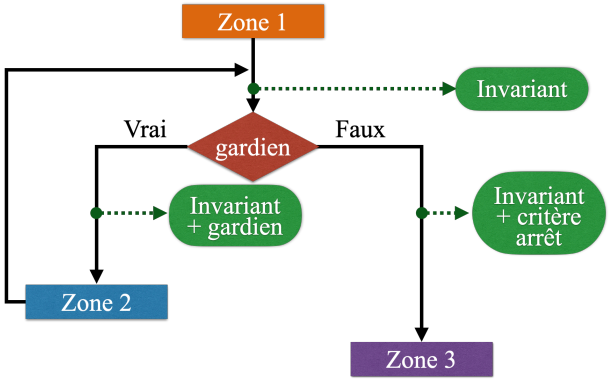


Figure 7 – Relation entre le Invariant Graphique et les différentes zones du programme.

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'Arrêt [12].

11. 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".
12. 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

$a \in \mathbb{Z}$

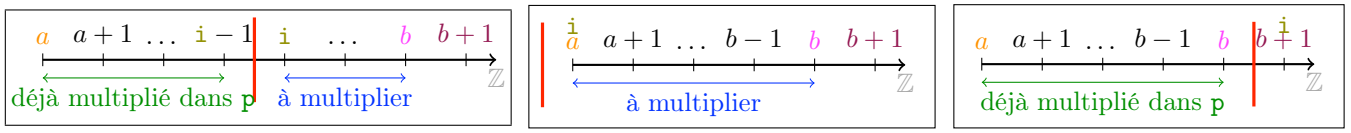
```
int a;
```

b, la borne supérieure

$b \in \mathbb{Z}$

```
int b;
```

La Fig. 8a 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 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 != 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. 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

```
p *= i;
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

```
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

A vous de jouer ! Passer à la construction de la Zone 1, Zone 2, Zone 3, du Critère d'Arrêt et de la Fonction de Terminaison des Sp :

- Sp_1 ;
- Sp_{2a} ;
- Sp_{2b} ;
- Sp_{2c} ;

Si vous séchez, reportez-vous à l'indice, à la Sec. 6.8.3.

6.8.2 Liens Spécification et Construction par Invariant de Boucle

La Fig. 9 établit le lien entre spécifications et construction par Invariant de Boucle.

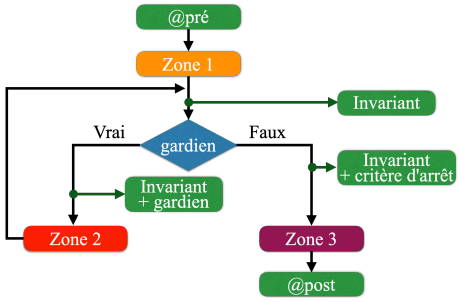


FIGURE 9 – Lien entre spécification et Invariant de Boucle.

En particulier, la PRÉCONDITION permet de faciliter la construction de la ZONE 1. En effet, on dispose d'un point de départ (la PRÉCONDITION) qui permet de s'assurer de la validité de certaines instructions et de l'Invariant Graphique pour identifier les variables nécessaires et leurs valeurs d'initialisation (potentiellement confirmées par la PRÉCONDITION)

Enfin, la POSTCONDITION facilite l'écriture de la ZONE 3. On dispose d'un point de départ (Invariant Graphique + Critère d'Arrêt) et le dessin utilisé pour la contruction de la spécification donne la situation finale. Il suffit dès lors de dériver les instructions en partant du point de départ pour arriver à la situation finale donnée par la POSTCONDITION.

Suite de l'Exercice

A vous de jouer ! Passer à la construction de la ZONE 1, ZONE 2, ZONE 3, du Critère d'Arrêt et de la Fonction de Terminaison des SP :

- SP_1 ;
- SP_{2a} ;
- SP_{2b} ;
- SP_{2c}.

Si vous séchez, reportez-vous à l'indice, à la Sec. 6.8.3.

6.8.3 Indice

La construction du code d'un module se fait en s'appuyant sur l'Invariant Graphique et la spécification. Pour

- la Zone 1, repartez de l'Invariant Graphique et identifiez les variables présentes dans l'Invariant Graphique. Manipulez la ligne de démarcation pour découvrir les valeurs d'initialisation. Pensez aussi à vous appuyer sur la PréCondition ;
- le Critère d'Arrêt, repartez de l'Invariant Graphique et déplacez la ligne de démarcation jusqu'à sa valeur maximale. La variable d'itération prendra une valeur particulière, ce qui vous donnera le Critère d'Arrêt. La négation du Critère d'Arrêt vous permet d'obtenir le Gardien de Boucle ;
- la Fonction de Terminaison, il s'agit simplement d'estimer la taille de la zone verte sur l'Invariant Graphique ;
- la Zone 2, repartez de l'Invariant Graphique et 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. N'oubliez pas qu'après la dernière instruction de la Zone 2, le Gardien de Boucle sera réévalué. La situation générale décrite par votre Invariant Graphique doit donc être restaurée ;
- la Zone 3 repartez de l'Invariant Graphique et déplacez la ligne de démarcation jusqu'à sa valeur maximale. Cette situation particulière est votre point de départ, votre point d'arrivée est décrit par la PostCondition. Il suffit donc de dériver les instructions nécessaires (généralement assez simples).

Dans tous les cas, aidez-vous du Gli pour faciliter la manipulation graphique (vous pouvez explicitement déplacer les lignes de démarcation, l'Invariant Graphique s'adapte alors automatiquement).

Suite de l'Exercice

A vous de jouer ! Passer à la construction de la Zone 1, Zone 2, Zone 3, du Critère d'Arrêt et de la Fonction de Terminaison des Sp :

- Sp_1 ;
- Sp_{2a} ;
- Sp_{2b} ;
- Sp_{2c}.

6.8.4 Mise en Commun de la Construction

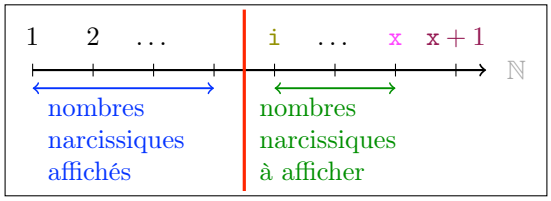
- Construction du SP_1 ;
- construction du SP_{2a} ;
- construction du SP_{2b} ;
- construction du SP_{2c}.

6.8.4.1 Module `affiche_narcissique()` (SP_1)

Pour rappel, l'interface du module est la suivante :

```
/*
 * PréCondition : x > 0.
 * PostCondition : Affiche tous les nombres narcissiques dans [1, x].
 */
void affiche_narcissique(unsigned int x);
```

L'Invariant Graphique est le suivant :



Zone 1

Commençons par identifier les variables nécessaires. L'Invariant Graphique utilise deux variables : `x` et `i`. `x` est déjà donné (c'est le paramètre formel du module) et est de type `unsigned int`. `i`, d'après le dessin, oscille entre 1 et `x` +1. Il sera donc de type `unsigned int`.

On peut maintenant déplacer la ligne de démarcation vers sa valeur minimale. On a, en outre, la garantie que `x` est strictement positif (au pire, il est donc égal à 1) grâce à la PréCondition. On obtient donc la situation particulière suivante :

On voit clairement que la valeur d'initialisation de `i` est 1. `x` sera initialisé lors de l'invocation (via le paramètre effectif) mais il est garanti que sa valeur sera > 0 (cfr. PréCondition). N'oublions pas d'appliquer les principes de la programmation défensive. Avant de déclarer et initialiser `i`, il convient donc de vérifier la PréCondition grâce à la procédure `assert()`.

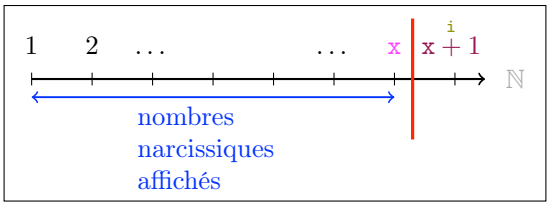
On obtient donc le simple bout de code suivant :

```
void affiche_narcissique(unsigned int x){
  assert(x > 0);
  unsigned int i = 1;
}//fin affiche_narcissique()
```

Critère d'Arrêt et Fonction de Terminaison

Commençons par le Critère d'Arrêt. Pour le trouver, il faut faire glisser la ligne de démarcation de sorte que la zone verte ("nombres narcissiques à afficher") ait disparu et que la zone bleue ("nombres narcissiques affichés") recouvre tout le dessin. Soit :

Nous voyons donc que les nombres narcissiques dans l'intervalle [1, `x`] seront affichés `i == x+1`. Notre Critère d'Arrêt est donc : `i == x+1`. 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 != x+1`. Soit, de manière plus naturelle : `i <= x` (ce qui a pour mérite de mieux mettre en valeur la position relative de `i` par rapport à `x`, telle qu'illustrée par l'Invariant Graphique).

On dérive alors le code suivant :

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

Passons à la Fonction de Terminaison. Il suffit, ici, d'estimer la taille de la zone verte ("nombres narcissiques à afficher"). Pour ce faire, estimons d'abord la taille de l'intervalle complet (i.e., $[1, x]$). Soit x. La zone bleue ("nombres narcissiques affichés") est comprise dans l'intervalle $[1, i-1]$. Elle a donc une taille de $i-1$. Il suffit alors de faire la différence entre ces deux tailles pour obtenir la Fonction de Terminaison. Soit $x-(i-1)$.

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

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. Cela signifie qu'il reste encore au moins un nombre de l'intervalle (`i`) à traiter. Par traiter, on entend déterminer si c'est un nombre narcissique et, si oui, l'afficher à l'écran. Déterminer si un nombre est narcissique, c'est le job du $\text{Sp}_{2.c}$. On a donc deux cas à discriminer :

1. le nombre `i` est narcissique. Dans ce cas, il faut l'afficher à l'écran.
2. le nombre `i` n'est pas narcissique. Dans ce cas, il ne faut rien faire.

La discrimination de ces deux cas, dans le code, se fait via une structure de contrôle conditionnelle (voir ligne 2 dans le segment de code ci-dessous).

Après, il ne faut pas oublier de faire avancer la zone bleue d'une position dans l'intervalle (et donc faire diminuer d'autant la zone verte). Voir la ligne 6 dans le segment de code ci-dessous.

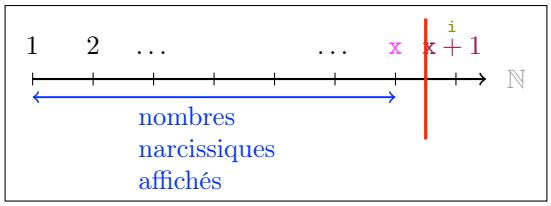
On obtient alors le code suivant :

```
1 while(i <= x){
2   if(est_narcissique(i))
3     //Cas 1
4     printf("%u␣", i);
5
6     i++;
7 }//fin while - i
```

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). On est dès lors dans la situation suivante :

Cette situation sert de point de départ pour atteindre la PostCondition (i.e., "Afficher tous les nombres narcissiques dans [1, x]."). Les deux situations sont, en fait, identiques. Dès lors, il n'y a rien à faire dans la Zone 3. On peut, tout au plus, rajouter un passage à la ligne.



Code Complet du Module

```
void affiche_narcissique(unsigned int x){
  unsigned int i = 1;

  while(i <= x){
    if(est_narcissique(i))
      printf("%u ", i);

    i++;
  }//fin while - i

  printf("\n");
}//fin affiche_narcissique()
```

Suite de l'exercice

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

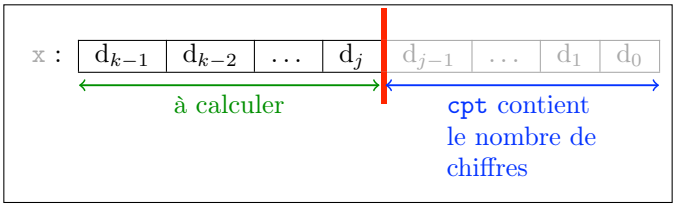
- construction du SP_{2a} ;
- construction du SP_{2b} ;
- construction du SP_{2c} ;
- code complet pour tout le programme.

6.8.4.2 Module `nb_chiffres()` (SP$_{2.a}$)

Pour rappel, l'interface du module est la suivante :

```
/*
 * PréCondition : /
 * PostCondition : nb_chiffres vaut le nombre de chiffres dans x.
 */
unsigned int nb_chiffres(unsigned int x);
```

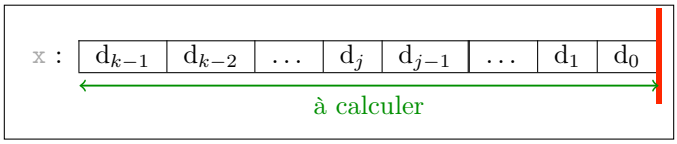
L'Invariant Graphique est le suivant :



Zone 1

Commençons par identifier les variables nécessaires. L'Invariant Graphique utilise deux variables : `x` et `cpt`. `x` est déjà donné (c'est le paramètre formel du module) et est de type `unsigned int`. `cpt` est la variable d'accumulation qui va compter le nombre de chiffres dans `x`. Par défaut, `cpt` ne peut pas être négatif et doit forcément être entier. Il sera donc, aussi, de type `unsigned int`.

On peut maintenant déplacer la ligne de démarcation vers sa valeur minimale. Il n'y a aucune PréCondition sur `x`. On obtient donc la situation particulière suivante :



On voit clairement que, dans la situation initiale, on n'a encore compté aucun chiffre dans `x` (`x` n'a été amputé d'aucun chiffre). Dès lors, la valeur d'initialisation de `cpt` est une somme à zéro terme, soit 0. `x` sera initialisé lors de l'invocation (via le paramètre effectif). Puisque la fonction n'a aucune PréCondition, il n'y a rien à faire en terme de programmation défensive.

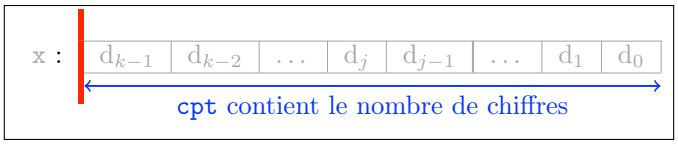
On obtient donc le simple bout de code suivant :

```
unsigned int nb_chiffres(unsigned int x){
  unsigned int cpt = 0;
}//fin nb_chiffres()
```

Critère d'Arrêt et Fonction de Terminaison

Commençons par le Critère d'Arrêt. Pour le trouver, il faut faire glisser la ligne de démarcation de sorte que la zone verte ("à calculer") ait disparu et que la zone bleue ("`cpt` contient le nombre de chiffres") recouvre tout le dessin. Soit :

Nous voyons que l'entièreté du nombre `x` a été amputée (tous les chiffres qui le composent sont grisés). `x` a donc "disparu". Comme c'est une variable de type `unsigned int`, il prend dès lors la valeur minimale, soit 0. Notre Critère d'Arrêt est donc : `x == 0`. 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 : `x != 0`. Soit de manière plus naturelle : `x > 0`. Cette formulation a pour mérite de mieux mettre en valeur la façon dont `x` évolue à chaque itération.

On dérive alors le code suivant :

```
1 while(x > 0){
2   //à compléter
3 }//fin while - x
```

Pour la Fonction de Terminaison, la formulation du Gardien de Boucle nous la donne directement.

La Fonction de Terminaison est donc : $f : x$.

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. Cela signifie qu'il reste encore au moins un chiffre du nombre (`x`) à traiter. Par traiter, on entend incrémenter le compteur de chiffres (`cpt`).

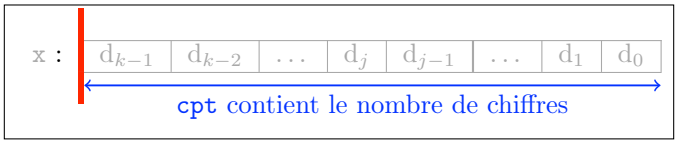
Après, il ne faut pas oublier de faire avancer la zone bleue d'une position dans le nombre `x` (et donc faire diminuer d'autant la zone verte). Voir la ligne 3 dans le segment de code ci-dessous.

On obtient alors le code suivant :

```
1 while(x > 0){
2   cpt++;
3   x /= 10;
4 }//fin while - x
```

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). On est dès lors dans la situation suivante :



Cette situation sert de point de départ pour atteindre la PostCondition (i.e., "`nb_chiffres` vaut le nombre de chiffres dans x."). Le nombre de chiffres dans `x` est connu : c'est `cpt`. On peut donc reformuler la PostCondition comme devant être : "`nb_chiffres` vaut `cpt`". Pour atteindre cette situation, il suffit de retourner la valeur de `cpt`.

Nous avons donc le bout de code suivant :

```
1 return cpt;
```

Code Complet du Module

```
unsigned int nb_chiffres(unsigned int x){
  unsigned int cpt = 0;

  while(x > 0){
    cpt++;
    x /= 10;
  }//fin while - x

  return cpt;
}//fin nb_chiffres()
```

Suite de l'exercice

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

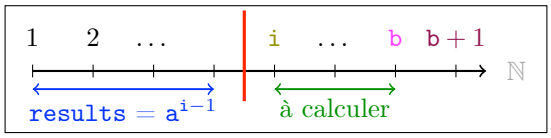
- construction du SP_1 ;
- construction du SP_{2b} ;
- construction du SP_{2c} ;
- code complet pour tout le programme.

6.8.4.3 Module `puissance()` ($\text{SP}_{2.b}$)

Pour rappel, l'interface du module est la suivante :

```
/*
 * PréCondition : b ≥ 1.
 * PostCondition : puissance vaut a^b.
 */
unsigned int puissance(unsigned int a, unsigned int b);
```

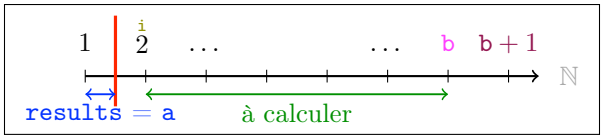
L'Invariant Graphique est le suivant :



Zone 1

Commençons par identifier les variables nécessaires. L'Invariant Graphique utilise quatre variables : `a`, `b`, `i` et `results`. `a` et `b` sont déjà donnés (ce sont les paramètres formels du module) et sont de type `unsigned int`. `i` est la variable d'itération qui va osciller dans l'intervalle d'intérêt (i.e., $\in [1, b]$). Comme `b` est de type `unsigned int`, naturellement, `i` sera aussi de type `unsigned int`. `results` est la variable d'accumulation qui va contenir les multiplications intermédiaires. Par défaut, `results` ne peut pas être négatif (la multiplication de naturels donnera toujours un naturel). Il sera donc, aussi, de type `unsigned int`.

On peut maintenant déplacer la ligne de démarcation vers sa valeur minimale. Il y a une PréCondition sur `b` (≥ 1). On obtient alors la situation suivante :



La ligne de démarcation n'est pas placée à son minimum (i.e., avant 1) car on sait que $b \geq 1$ (PréCondition). On peut donc placer la ligne de démarcation entre 1 et 2. Dès lors, on tire les conclusions suivantes :

- puisque `i` se trouve à droite de la ligne de démarcation, sa valeur d'initialisation est 2 ;
- la zone bleue ("`results` = $\texttt{a}^{\texttt{i}-1}$") porte sur une valeur, i.e., `i`-1 = 2 - 1 = 1. On voit donc qu'il faut initialiser `results` à `a` ("`results` = $\texttt{a}^{1}$").

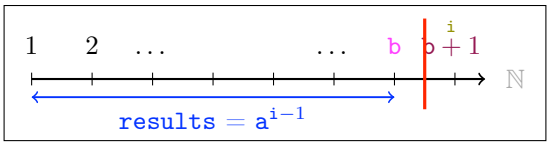
On obtient donc le simple bout de code suivant :

```
unsigned int puissance(unsigned int a, unsigned int b){
  assert(b >= 1);
  unsigned int results = a, i=2;
}//fin nb_chiffres()
```

Critère d'Arrêt et Fonction de Terminaison

Commençons par le Critère d'Arrêt. Pour le trouver, il faut faire glisser la ligne de démarcation de sorte que la zone verte ("à calculer") ait disparu et que la zone bleue ("`results` = $\texttt{a}^{\texttt{i}-1}$") recouvre tout le dessin. Soit :

Nous voyons donc que, dans cette situation particulière, `i` prend la valeur particulière `b` +1. Dès lors, `results` contient $\texttt{a}^{\texttt{b}+1-1}$. On a donc bien le résultat désiré quand `i` prend la valeur `b` +1. Notre Critère



d'Arrêt est donc : `i == b+1`. 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 != b+1`. Soit, de manière plus naturelle : `i <= b` (ce qui a pour mérite de mieux mettre en valeur la position relative de `i` par rapport à `b`, telle qu'illustrée par l'Invariant Graphique).

On dérive alors le code suivant :

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

Passons à la Fonction de Terminaison. Il suffit, ici, d'estimer la taille de la zone verte ("à calculer"). Pour ce faire, estimons d'abord la taille de l'intervalle complet (i.e., $[1, b]$). Soit b. La zone bleue ("`results` = $\texttt{a}^{\texttt{i}-1}$") est comprise dans l'intervalle $[1, i-1]$. Elle a donc une taille de $i-1$. Il suffit alors de faire la différence entre ces deux taille pour obtenir la Fonction de Terminaison. Soit $b-(i-1)$.

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

Zone 2

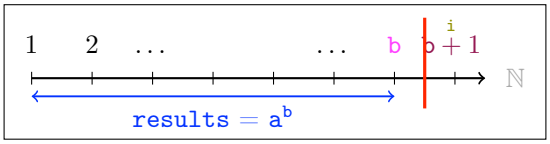
La Zone 2 correspond au Corps de la Boucle. Par conséquent, l'Invariant Graphique est vrai et le Gardien de Boucle vient d'être évalué à vrai. Cela signifie qu'il reste encore au moins un nombre de l'intervalle (`i`) à traiter. Par traiter, on entend effectuer le calcul : `results` = $\texttt{a}^{\texttt{i}-1} \times \texttt{a} = \texttt{a}^{\texttt{i}}$.

Après, il ne faut pas oublier de faire avancer la zone bleue d'une position dans l'intervalle (et donc faire diminuer d'autant la zone verte). Voir la ligne 3 dans le segment de code ci-dessous.

```
1 while(i <= b){
2   results *= a;
3   i++;
4 }//fin while - i
```

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). On est dès lors dans la situation suivante :



Cette situation sert de point de départ pour atteindre la PostCondition (i.e., "puissance vaut a^b."). Or, on voit sur le dessin que `results` contient déjà $\texttt{a}^{\texttt{b}}$. Dès lors, la seule chose à faire dans la Zone 3, c'est de retourner la valeur de `results`. Soit l'instruction suivante :

```
1 return results;
```

Code Complet du Module

```
unsigned int puissance(unsigned int a, unsigned int b){
  assert(b > 1);
  unsigned int results = a, i=2;

  while(i <= b){
    results *= a;
    i++;
  }//fin while - i

  return results;
}//fin puissance()
```

Suite de l'exercice

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

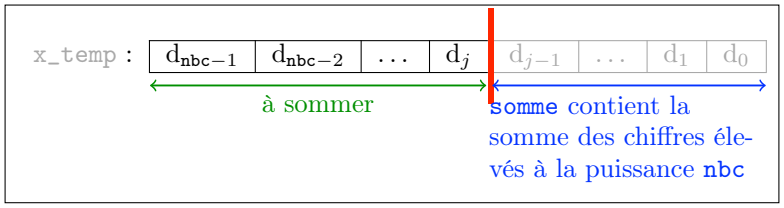
- construction du SP_1 ;
- construction du SP_{2a} ;
- construction du SP_{2c} ;
- code complet pour tout le programme.

6.8.4.4 Module `est_narcissique()` (SP$_{2.c}$)

Pour rappel, l'interface du module est la suivante :

```
/*
 * PréCondition : /
 * PostCondition : est_narcissique vaut 1 si x est un nombre narcissique.
 *                 0 sinon.
 */
int est_narcissique(unsigned int x);
```

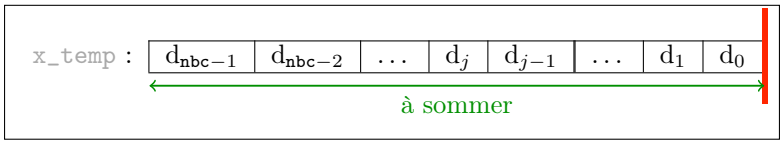
L'Invariant Graphique est le suivant :



Zone 1

Commençons par identifier les variables nécessaires. L'Invariant Graphique utilise trois variables : `x_temp`, `nbc` et `somme`. `x_temp` contient la valeur initiale de `x`, qui est le paramètre formel de la fonction. `x_temp` est donc de type `unsigned int`. `nbc` contient le nombre de chiffres dans `x`. Par conséquent, il ne peut être que de type `unsigned int`. Enfin, `somme` est la variable d'accumulation qui va contenir la somme des chiffres élevés à la puissance `nbc`. Il ne peut donc être que de type `unsigned int`.

On peut maintenant déplacer la ligne de démarcation vers sa valeur minimale. Il n'y a aucune PréCondition sur `x`. On obtient donc la situation particulière suivante :



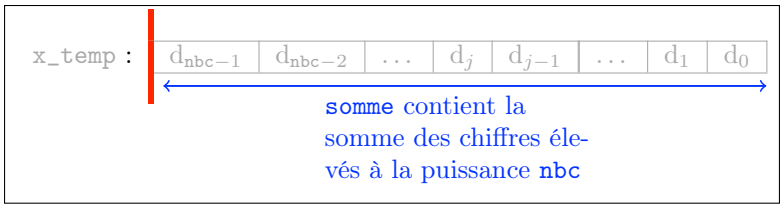
De la figure, on voit clairement les éléments suivants apparaître :

- `x_temp` n'a été amputé d'aucun chiffre. Il est donc à sa valeur initiale, soit `x` ;
- `nbc` est déjà obtenu. L'initialisation de `nbc` se fait en invoquant le SP$_{2.a}$ et en affectant à `nbc` son résultat ;
- puisque `x_temp` est à sa valeur initiale, `somme` contient une somme à zéro terme, soit 0.

Enfin, puisque la fonction n'a aucune PréCondition, il n'y a rien à faire en terme de programmation défensive.

On obtient donc le simple bout de code suivant :

```
int est_narcissique(unsigned int x){
  unsigned int nbc = nb_chiffres(x);
  unsigned int somme = 0, x_temp = x;
}//fin est_narcissique()
```



Critère d'Arrêt et Fonction de Terminaison

Commençons par le Critère d'Arrêt. Pour le trouver, il faut faire glisser la ligne de démarcation de sorte que la zone verte ("à sommer") ait disparu et que la zone bleue ("`somme` contient la somme des chiffres élevés à la puissance `nbc`") recouvre tout le dessin. Soit :

Nous voyons que l'entièreté du nombre `x_temp` a été amputée (tous les chiffres qui le composent sont grisés). `x_temp` a donc "disparu". Comme c'est une variable de type `unsigned int`, il prend dès lors la valeur minimale, soit 0. Notre Critère d'Arrêt est donc : `x_temp == 0`. 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 : `x_temp != 0`. Soit de manière plus naturelle : `x_temp > 0`. Cette formulation a pour mérite de mieux mettre en valeur la façon dont `x_temp` évolue à chaque itération.

On dérive alors le code suivant :

```
1 while(x_temp > 0){
2   //à compléter
3 }//fin while - x_temp
```

Pour la Fonction de Terminaison, la formulation du Gardien de Boucle nous la donne directement.

La Fonction de Terminaison est donc : $f : x\_temp$.

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. Cela signifie qu'il reste encore au moins un chiffre du nombre (`x_temp`) à traiter. Par traiter, on entend incrémenter la somme des chiffres à la puissance `nbc` (`somme` – ligne 2 dans le segment de code ci-dessous).

Après, il ne faut pas oublier de faire avancer la zone bleue d'une position dans le nombre `x_temp` (et donc faire diminuer d'autant la zone verte). Voir la ligne 3 dans le segment de code ci-dessous.

On obtient alors le code suivant :

```
1 while(x_temp > 0){
2   somme += puissance(x_temp%10, nbc);
3   x_temp /= 10;
4 }//fin while - x_temp
```

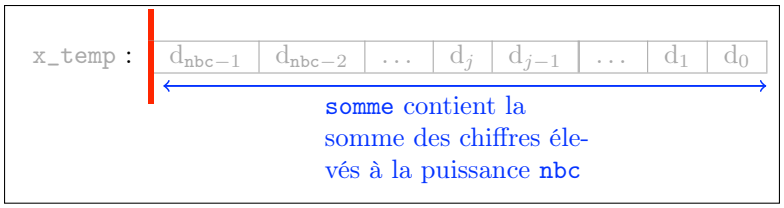
Zone 3

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). On est dès lors dans la situation suivante :

Cette situation sert de point de départ pour atteindre la PostCondition (i.e., "`est_narcissique` vaut 1 si `x` est un nombre narcissique. 0 sinon"). A ce stade, `somme` contient la somme de tous les chiffres de `x_temp` (et donc `x`), chacun élevé à la puissance `nbc`.

Pour rappel, un nombre est narcissique si il est égal à somme de chiffres qui le composent, chacun des chiffres étant élevé à la puissance correspondant à la quantité de chiffres dans le nombre de base. Dit autrement, `x` est narcissique si il est égal à `somme`.

Dans la Zone 3, il suffit donc de retourner au code appelant le résultat du test `x == somme`. Soit :




```
return (x==somme);
```

Code Complet du Module

```
int est_narcissique(unsigned int x){
  unsigned int nbc = nb_chiffres(x);
  unsigned int somme = 0, x_temp = x;

  while(x_temp > 0){
    somme += puissance(x_temp%10, nbc);
    x_temp /= 10;
  }//fin while - x_temp

  return (x==somme);
}//fin est_narcissique()
```

Suite de l'exercice

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

- construction du SP_1 ;
- construction du SP_{2a} ;
- construction du SP_{2b} ;
- code complet pour tout le programme.

On peut maintenant continuer l'écriture du code, en particulier mettre les différentes parties ensemble. Voir Sec. 6.9.

6.9 Code Final

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

Extrait de code 8 – Fichier `narcissique.h`

```h
/*
 * PréCondition : x > 0.
 * PostCondition : Affiche tous les nombres narcissiques dans [1, x].
 */
void affiche_narcissique(unsigned int x);

/*
 * PréCondition : /
 * PostCondition : nb_chiffres vaut le nombre de chiffres dans x.
 */
unsigned int nb_chiffres(unsigned int x);

/*
 * PréCondition : b ≥ 1.
 * PostCondition : puissance vaut a^b.
 */
unsigned int puissance(unsigned int a, unsigned int b);

/*
 * PréCondition : /
 * PostCondition : est_narcissique vaut 1 si x est un nombre narcissique.
 *                 0 sinon.
 */
int est_narcissique(unsigned int x);
```

Extrait de code 9 – Fichier `narcissique.c`

```c
#include <assert.h>
#include <stdio.h>

#include "narcissique.h"

void affiche_narcissique(unsigned int x){
  assert(x>0);

  unsigned int i = 1;

  while(i <= x){
    if(est_narcissique(i))
      printf("%u␣", i);

    i++;
  }//fin while - i

  printf("\n");
}//fin affiche_narcissique()

unsigned int nb_chiffres(unsigned int x){
  unsigned int cpt = 0;

  while(x > 0){
    cpt++;
    x /= 10;
  }//fin while - x

  return cpt;
}//fin nb_chiffres()
```

```c

unsigned int puissance(unsigned int a, unsigned int b){
  assert(b > 1);
  unsigned int results = a, i=2;

  while(i <= b){
    results *= a;
    i++;
  }//fin while - i

  return results;
}//fin puissance()

int est_narcissique(unsigned int x){
  unsigned int nbc = nb_chiffres(x);
  unsigned int somme = 0, x_temp = x;

  while(x_temp > 0){
    somme += puissance(x_temp%10, nbc);
    x_temp /= 10;
  }//fin while - x_temp

  return (x==somme);
}//fin est_narcissique()
```

Extrait de code 10 – Programme principal à compléter (fichier `main-narcissique.c`)

```c
#include <stdio.h>

#include "narcissique.h"

int main(void){
  unsigned int x;

  printf("Entrez une valeur pour x: ");
  scanf("%u", &x);

  printf("Liste des nombres narcissiques entre 1 et %u: \n", x);
  affiche_narcissique(x);

  return 0;
}//fin programme
```

Pour rappel, pour compiler et tester le code, on procède comme suit :

```
$>gcc -o pgm-narcissique main-narcissique.c narcissique.c
```