Introduction à la Programmation

Benoit Donnet

Année Académique 2023 - 2024

Agenda

- Introduction
- Chapitre 1: Bloc, Variable, Instruction Simple
- Chapitre 2: Structures de Contrôle
- Chapitre 3: Méthodologie de Développement
- Chapitre 4: Structures de Données
- Chapitre 5: Modularité du Code
- Chapitre 6: Pointeurs
- Chapitre 7: Allocation Dynamique

Agenda

- Chapitre 3: Méthodologie de Développement
  - Schéma Méthodologique
  - Définition du Problème
  - Analyse du Problème
  - Invariant de Boucle
  - Fonction de Terminaison

Agenda

- Chapitre 3: Méthodologie
  - Schéma Méthodologique
  - Définition du Problème
  - Analyse du Problème
  - Invariant de Boucle
  - Fonction de Terminaison

Schéma Méthodo

- Il est (très) difficile d'écrire un programme correct du premier coup, de la première à la dernière ligne
- Il est préférable d'adopter une *démarche méthodologique*
- **Schéma méthodologique**

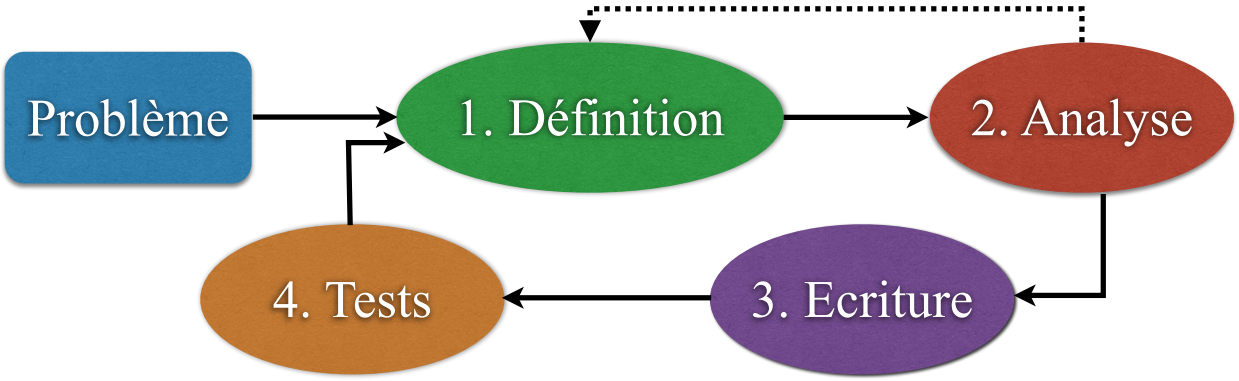


Schéma Méthodo (2)

- 1ère étape: *Définition du problème*
  - définir précisément ce qu'on attend du programme
  - prendre connaissance des informations nécessaires à la résolution du problème
    - ✓ quelles sont les données en entrée?
      - **Input**
    - ✓ quels sont les résultats attendus et sous quelle forme?
      - **Output**
    - ✓ que dois-je manipuler comme données?
      - **Caractérisation des Inputs**
      - lien(s) avec l'Input
  - Slides 9 → 15

Schéma Méthodo (3)

- 2ème étape: *Analyse du problème*
  - découper le problème en parties plus petites et plus faciles à appréhender
    - **découpe en sous-problèmes** (ou **approche systémique**)
      - chaque sous-problème pourra être résolu indépendamment
      - un sous-problème peut admettre plusieurs solutions
      - il est possible de généraliser un sous-problème (cfr. Chap. 6)
  - structurer le problème
    - comment les sous-problèmes s'emboîtent
  - cette étape revient à penser *l'architecture* du programme
  - Slides 16 → 62

Schéma Méthodo (4)

- 3ème étape: *Ecriture du code*
  - implémentation des différents sous-problèmes
    - **Invariant** et **Fonction de Terminaison**
      - si le sous-problème fait intervenir une boucle
  - mise en commun des sous-problèmes
  - Slides 63 → 136
- 4ème étape: *Tests*
  - vérifier que l'implémentation résout bien le problème
  - peut nécessiter de revenir à une étape précédente en cas d'erreur
  - cfr. INFO0030, Partie 2, Chap. 3

Agenda

- Chapitre 3: Méthodologie
  - Schéma Méthodologique
  - Définition du Problème
    - ✓ Principe
    - ✓ Représentation Graphique
    - ✓ Exemple
  - Analyse du Problème
  - Invariant de Boucle
  - Fonction de Terminaison

Principe

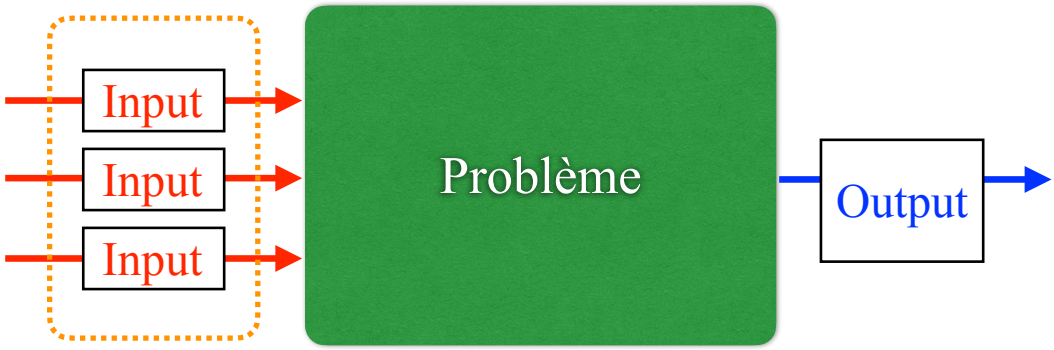
- Il s'agit de définir
  - les données en entrée
    - ✓ **Input(s)**
    - ✓ toujours des variables
    - ✓ possible qu'il n'y en ait pas
  - les données/résultats en sortie
    - ✓ **Output(s)**
    - ✓ variable(s) ou affichage d'un résultat sur la sortie standard
  - quels sont les types des données utilisées en entrée
    - ✓ **Caractérisation des Inputs**
    - ✓ donner un type C et un identifiant de variable aux Inputs.
    - ✓ revient à une déclaration de variable

Principe (2)

- Pour la caractérisation des Inputs, il faut
  - décrire l'utilité de chaque Input
  - donner un nom *évocateur* à chacun en fonction de ce qu'il représente
    - ✓ identifiant de variable
  - donner un *type*
    - ✓ `int`
    - ✓ `double`
    - ✓ `float`
    - ✓ `char`
    - ✓ ...
- Cette étape est à faire *avant* d'écrire la moindre ligne de code

Représentation Graphique

- La Définition d'un Problème peut se représenter graphiquement
- Patterns pour la représentation graphique
  - boite arrondie pour le problème
  - flèches orientées pour les Inputs/Outputs.
    - ✓ toujours étiquetées
      - boites à angles droits pour les variables (Input ou Output)
      - texte pour l'affichage à l'écran (Output)

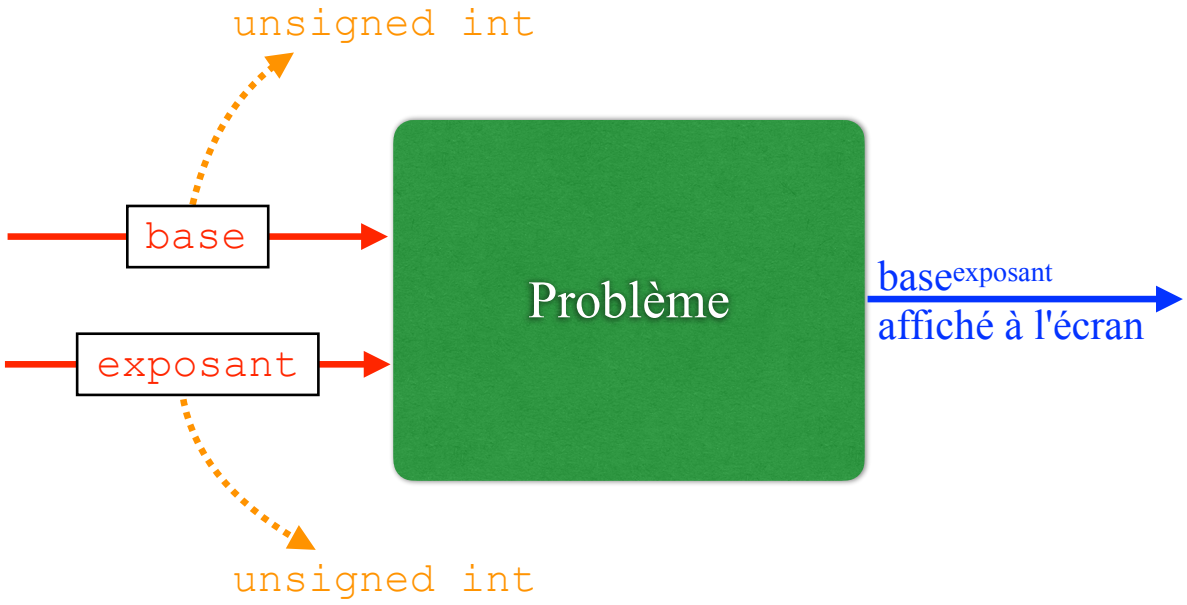


Exemple

- Calculer a^b et afficher le résultat sur la sortie standard
- Définition du problème
  - Input
    - ✓ la *base* et l'*exposant*, lus au clavier
  - Output
    - ✓ le résultat de l'*exponentiation* est affiché à l'écran
  - Caractérisation des Inputs
    - ✓ `base`, la base
      - `base` $\in \mathbb{N}$
      - `unsigned int base;`
    - ✓ `exp`, l'exposant
      - `exp` $\in \mathbb{N}$
      - `unsigned int exp;`

Exemple (2)

- Représentation graphique



Exemple (3)

- Canevas général du code

```
#include <stdio.h>

int main(){
  unsigned int base, exp;

  //déclaration de variables supplémentaires

  //code

}//fin programme
```


Agenda

- Chapitre 3: Méthodologie
  - Schéma Méthodologique
  - Définition du Problème
  - Analyse du Problème
    - ✓ Principe
    - ✓ Méthode de Découpe en SP
    - ✓ Déclaration d'un SP
    - ✓ Typologie des SPs
    - ✓ Structuration des SPs
    - ✓ Représentation Graphique
    - ✓ Exemple Complet 1: Impression de Chiffres
    - ✓ Exemple Complet 2: Nombre Parfait
  - Invariant de Boucle
  - Fonction de Terminaison

Principe

- Une fois les données déterminées et les résultats explicités
  - Etape 1 (Définition) de la méthodologie
  - il reste à trouver la méthode permettant d'obtenir les résultats à partir des données
- Quid si la solution n'est pas évidente du 1er coup d'oeil?
  - essai/erreur?
  - coder comme une bête jusqu'à arriver à une "solution"?
- Solution
  - découper le problème en **sous-problèmes** (SPs)
    - ✓ **approche systémique**
  - les structurer

Méthode de Découpe

- Lorsque le problème est trop complexe
  - il faut le découper en tâches/fonctionnalités distinctes
    - ✓ les différents SPs
- Chaque SP peut être divisé en plusieurs SPs
  - jusqu'à obtenir des SPs évidents à résoudre
  - on peut donc appliquer la méthodologie générale sur chaque SP
- Quelle est la **bonne** granularité pour un SP?
  - un module
    - ✓ `scanf()`, `printf()`
  - un traitement itératif (i.e., boucle)
- **Mauvaise** granularité pour un SP?
  - déclaration/initialisation de variables
  - structure conditionnelle
  - calcul simple

Méthode de Découpe (2)

1. Identifier les grandes fonctionnalités que la solution doit avoir
   - cela revient à répondre à la question "Quoi?"
   - ne surtout pas se demander "Comment"?
     - ✓ c'est l'étape suivante (« Écriture du code ») !!!
2. À chaque fonctionnalité identifiée correspond un SP
3. Pour chaque SP identifié, se poser la question
   - peut-on le décomposer encore de façon à déléguer une tâche à un autre SP?
     - ✓ tout en gardant en tête la granularité (Slide 18)
4. On s'arrête quand on ne peut plus décomposer
   - cfr. granularité (Slide 18)

Déclaration d'un SP

- Comment déclarer un SP?
  1. identification
     - ✓ chaque SP doit avoir un nom significatif qui représente sa tâche
     - ✓ chaque SP est identifié par un numéro unique
       - exemple: SP_1
     - ✓ format général
       - SP_x: nom
  2. définition
     - ✓ chaque SP doit être défini correctement
       - Input(s)
       - Output(s)
       - Caractérisation des/de l'Input(s)

Les types de SPs (1)

1. **Lecture au clavier**
   - lecture d'une ou plusieurs valeurs sur l'entrée standard
     - ✓ pas d'Input(s)
     - ✓ en Output, les variables qui contiennent les valeurs lues
   - identification
     - ✓ SPx: lecture au clavier
2. **Affichage à l'écran**
   - afficher une ou plusieurs valeurs sur la sortie standard
     - ✓ en Inputs, les variables à afficher
     - ✓ en Output, on indique ce qui a été affiché
   - identification
     - ✓ SPx: affichage à l'écran

Les types de SPs (2)

3. **Réalisation d'une action**
   - réaliser une action (compter, compresser, sommer, calculer, convertir, vérifier une propriété, ...) sur une (ou plusieurs) variables
     - ✓ en Inputs, les variables à manipuler pour l'action
     - ✓ en Output, le résultat de l'action
       - ‣ variable ou affichage à l'écran
   - identification
     - ✓ SPx: action
4. **Énumération et Action**
   - énumérer des valeurs et, pour chacune d'elles, effectuer une action
     - ✓ en Inputs, les variables qui délimitent l'intervalle
     - ✓ en Output, le résultat de l'action
       - ‣ variable ou affichage à l'écran
   - identification
     - ✓ SPx: énumération et action

Les types de SPs (3)

- Synthèse

Nom	Input(s)?	Output(s)?	Code Couleur
Lecture au clavier	∅	variable(s)	
Affichage à l'écran	variable(s)	variable(s) ou affichage	
Action	variable(s)	variable(s) ou affichage	
Énumération et Action	variable(s)	variable(s) ou affichage	

Structuration des SPs

- Une fois les différents SPs définis, il faut les structurer
  - comment les différents SPs s'emboîtent?
- Cela revient à penser l'**architecture du code**
- La définition de chaque SP et les interactions Input(s)/Output(s) entre SPs suggèrent la structuration
  - d'où l'importance de bien les définir au préalable
- Le problème général doit paraître résolu avec les différents SPs et leur structuration

Structuration des SPs (2)

- On envisage 3 types de structuration
  1. structuration **isolée**
     - ✓ le SP n'a pas d'Input mais bien un/des Output(s)
     - ✓ représentation textuelle
       - $SP_i \rightarrow$
  2. structuration **linéaire**
     - ✓ l'Output d'un SP est l'Input du SP suivant
     - ✓ représentation textuelle
       - $SP_i \rightarrow SP_{i+1}$
  3. structuration **imbriquée**
     - ✓ un SP est inclu (SP interne) dans un autre (SP externe)
     - ✓ le SP externe est forcément une énumération
       - chaque tour de boucle du SP externe doit fournir un Input pour le SP interne
     - ✓ en interne, il peut y avoir plusieurs SPs
       - structuration linéaire ou imbriquée
     - ✓ représentation textuelle
       - $SP_i \subset SP_j$

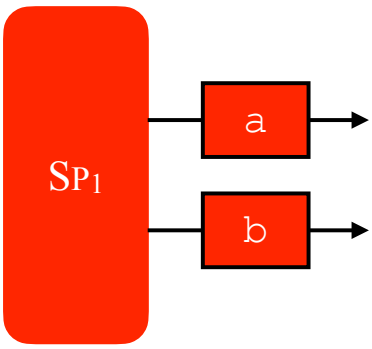
Représentation Graphique

- La structuration des SPs peut se représenter graphiquement
- Patterns pour la représentation graphique :
  - boite arrondie pour un SP
    - ✓ la boite doit être étiquetée avec le numéro unique du SP
    - ✓ respect des codes couleurs
      - cfr. Slide 23
  - flèches orientées pour les Inputs/Outputs
    - ✓ toujours étiquetées
      - boites à angles droits pour les variables (Input et/ou Output)
        - ✦ respect des codes couleurs (cfr Slide 23)
      - texte pour l'affichage à l'écran (Output)

Représentation Graphique (2)

- Application à la structuration et à la typologie

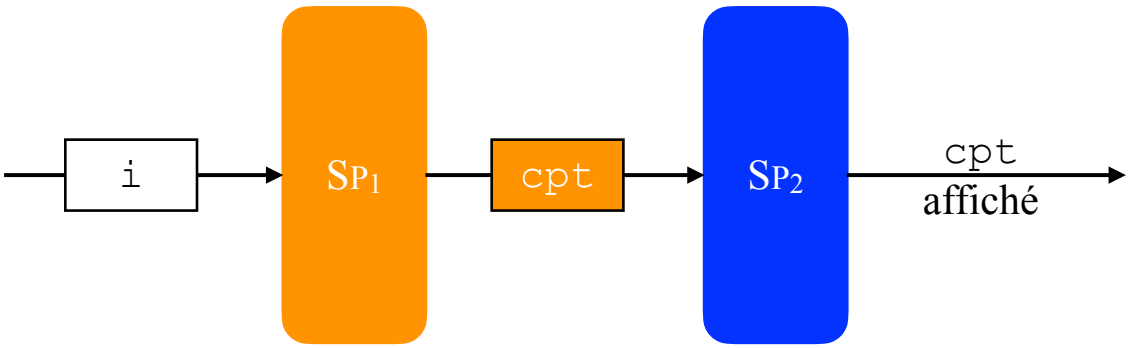
1. Structuration isolée
   - exemple: lire 2 nombres au clavier



Représentation Graphique (2)

- Application à la structuration et à la typologie (cont')

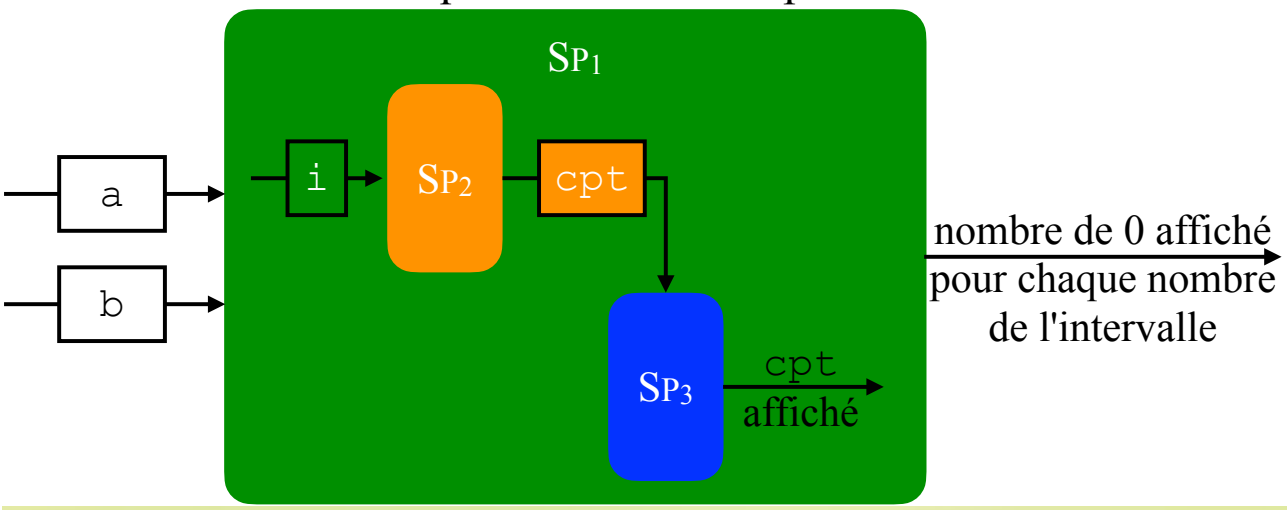
2. Structuration linéaire
   - exemple: afficher le nombre de 0 dans la représentation binaire d'un nombre entier positif



Représentation Graphique (3)

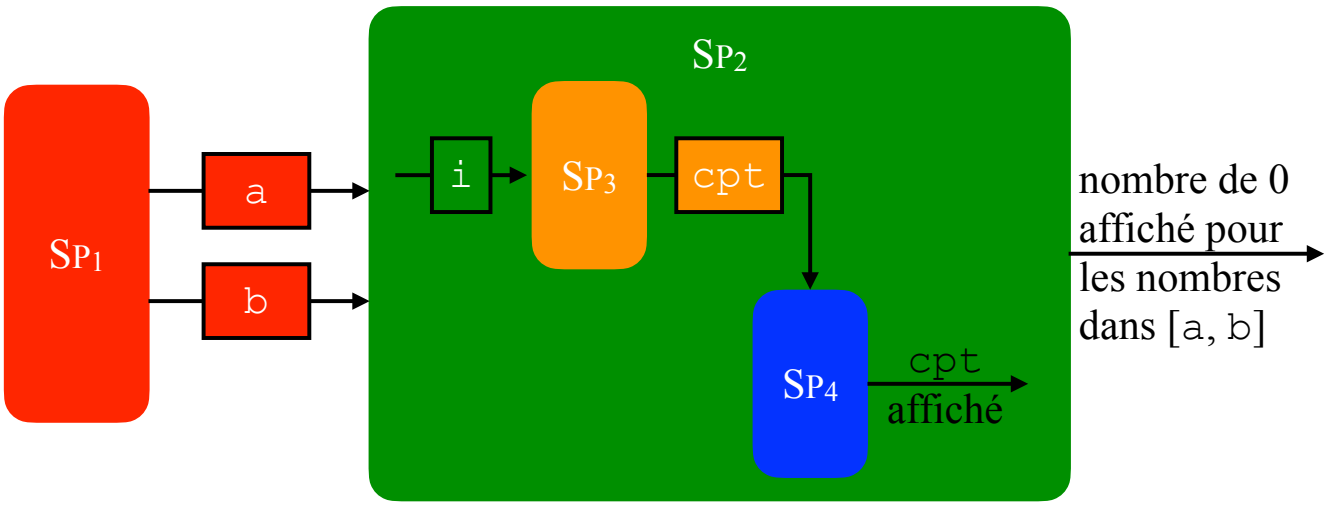
- Application à la structuration et à la typologie (cont')

3. Structuration imbriquée
   - exemple: afficher le nombre de 0 dans la représentation binaire de chaque nombre entier positif dans un intervalle



Représentation Graphique (4)

- Exemple complet
   - afficher à l'écran le nombre de 0 dans la représentation binaire des nombres entiers positifs dans l'intervalle [a, b]



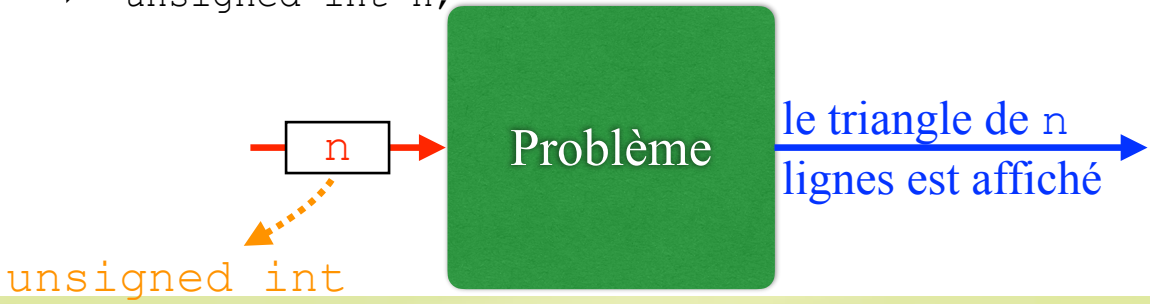
Ex1: Impression Chiffres

- Comment produire le résultat suivant?
  - afficher, sur la sortie standard, une suite de chiffres, avec la limite *n* introduite au clavier

```
1
22
333
4444
...
nnnnnnnnnnnn
```

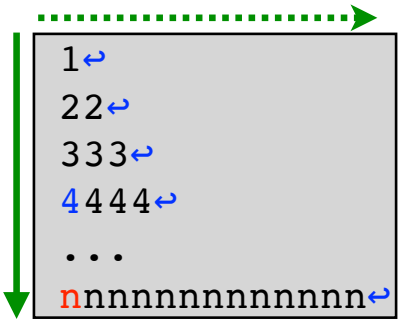

Ex1: Impression Chiffre (2)

- Étape 1: Définition du Problème
  - Input
    - ✓ le nombre de lignes, lu au clavier
  - Output
    - ✓ l'affichage à l'écran, tel que formaté dans l'énoncé du problème
  - Caractérisation de l'Input
    - ✓ `n`, le nombre de lignes
      - `n` $\in \mathbb{N}$
      - `unsigned int n;`



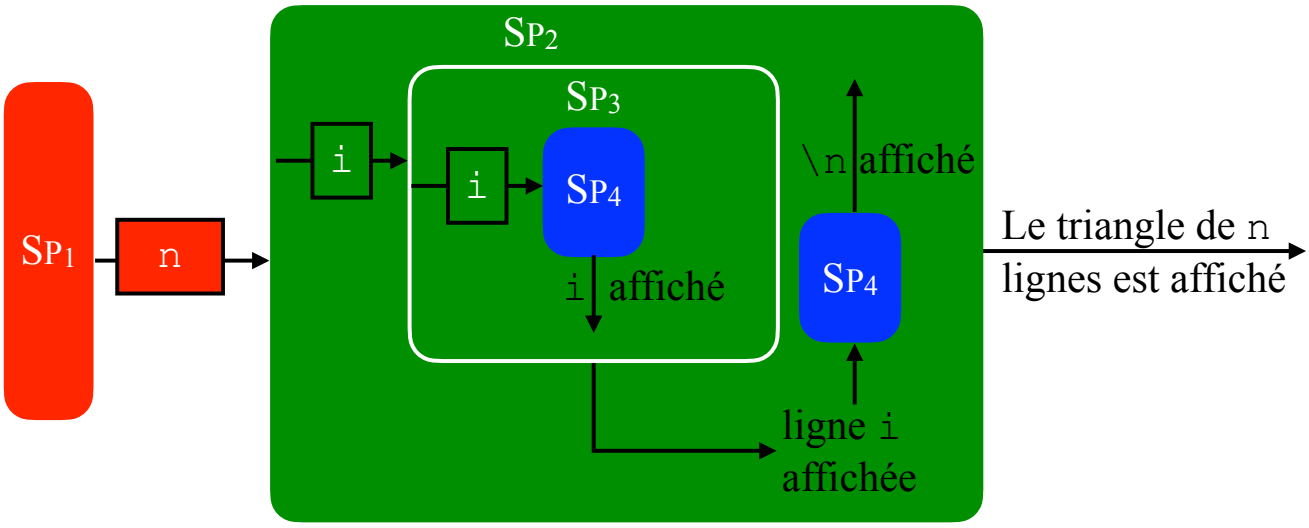
Ex1: Impression Chiffres (3)

- Étape 2: Analyse
- On peut envisager 4 SPs
  - SP_1: Lecture au clavier
    - ✓ lire la valeur de `n` au clavier
  - SP_2: Énumération et Affichage des lignes
    - ✓ énumération de chaque ligne dans l'intervalle [1, **`n`**] et affichage de ces lignes
  - SP_3: Énumération et Affichage d'une ligne
    - ✓ énumération, pour une ligne donnée **`i`**, de **`i`** fois le chiffre **`i`**
  - SP_4: Affichage d'un caractère
    - ✓ affichage du chiffre **`i`** ou du caractère `"\n"`



Ex1: Impression Chiffres (4)

- Structuration des 3 SPs?
  - $SP_1 \rightarrow [((SP_4 \subset SP_3) \rightarrow SP_4) \subset SP_2]$



Ex1: Impression Chiffres (5)

- Étape 3: Ecriture du Code
- On peut se focaliser sur chaque SP indépendamment l'un de l'autre
- Canevas général du code

```
#include <stdio.h>

int main(){
  unsigned int n;

  //déclaration de variables supplémentaires

  //résolution des 3 SPs

}//fin programme
```


Ex1: Impression Chiffres (6)

- Résolution SP_2 (énumération et affichage)
- Définition
  - Input
    - ✓ `n`, le nombre total de lignes à considérer
      - fourni par le SP_1
  - Output
    - ✓ les lignes de 1 à `n` ont été énumérées et affichées à l'écran
  - Caractérisation de l'Input
    - ✓ `n`
    - ✓ obtenu via le SP_1

Ex1: Impression Chiffres (7)

- Il faut écrire une boucle
  1. déclaration compteur
     - ✓ `unsigned int i = 1;`
  2. nombre de tours dans la boucle?
     - ✓ `n`
  3. Gardien de Boucle?
     - ✓ `i<=n`
  4. Corps de Boucle
     - ✓ afficher `i` fois le chiffre `i`
       - écrire la `i`ème ligne
       - SP_3!!
     - ✓ faire un retour à la ligne
       - SP_4!!
     - ✓ incrémenter `i`

Ex1: Impression Chiffres (8)

- Code SP_2 + SP_4

```
unsigned int i=1;

while(i<=n){
  //application du SP3

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

Ex1: Impression Chiffres (9)

- Résolution SP_3 (affichage des chiffres d'une ligne donnée)
- Définition
  - Input
    - ✓ `i`, le numéro de la ligne courante
  - Output
    - ✓ le chiffre `i` a été affiché `i` fois à l'écran
  - Caractérisation de l'Input
    - ✓ `i`
    - ✓ obtenu via le SP_2

Ex1: Impression Chiffres (10)

- Il faut écrire une boucle
  1. déclaration compteur
     - ✓ `unsigned int j = 1;`
  2. nombre de tours dans la boucle?
     - ✓ `i`
  3. Gardien de Boucle?
     - ✓ `j<=i`
  4. Corps de Boucle
     - ✓ écrire le chiffre `i` à l'écran
       - SP_4!!
     - ✓ incrémenter `j`

Ex1: Impression Chiffres (11)

- Code SP_3 + SP_4

```
unsigned int j = 1;

while(j<=i){
  printf("%u", i);
  j++;
}//fin while - j
```


Ex1: Impression Chiffres (12)

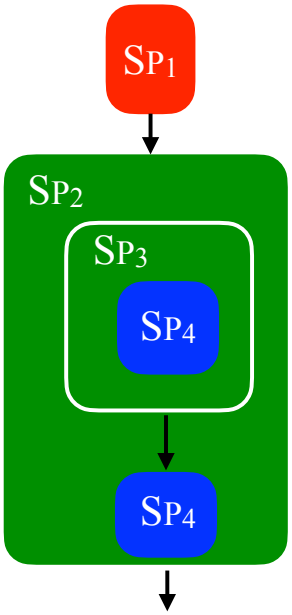
- Résolution SP_1 (lecture au clavier)
- Définition
  - Input
    - ✓ /
  - Output
    - ✓ `n` a été lu au clavier
  - Caractérisation de l'Input
    - ✓ /
- Code SP_1

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

Ex1: Impression Chiffres (13)

```c
#include <stdio.h>

int main(){
  unsigned int n;
  unsigned int i=1, j;

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

  while(i<=n){
    j = 1;
    while(j<=i){
      printf("%u", i);
      j++;
    }//fin while - j

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

SP1 SP2 SP3 SP4



Ex 2: Nombre Parfait

- Un **nombre parfait**
  - entier positif égal à la somme de ses diviseurs (excepté lui-même)
  - exemple: 28 = 1 + 2 + 4 + 7 +14
- Problème
  - rechercher tous les nombres parfaits appartenant à un intervalle donné ([1, nMax[)

Ex 2: Nombre Parfait (2)

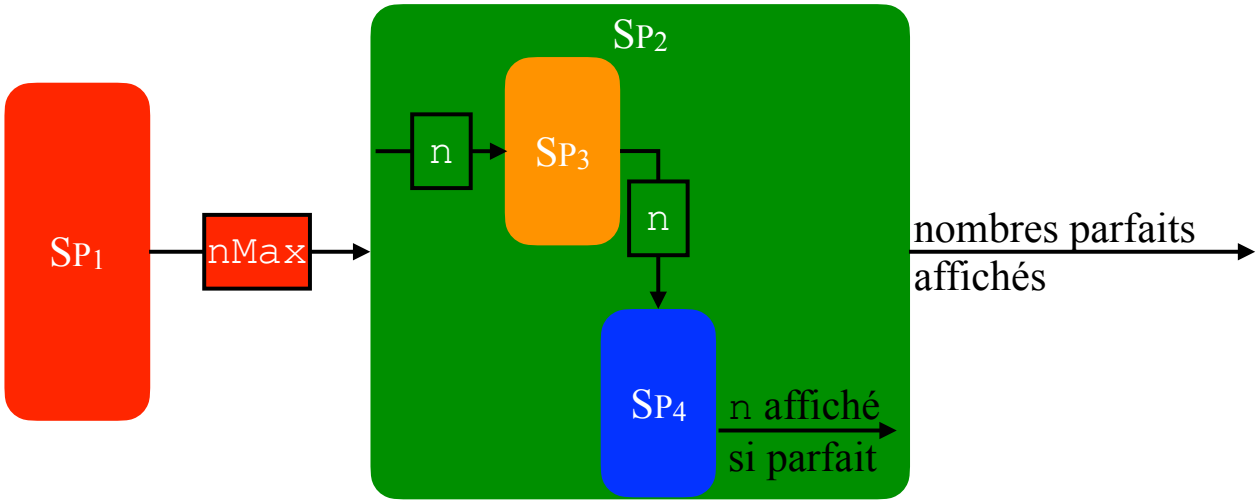
- Étape 1: Définition du Problème
  - Input
    - ✓ la borne supérieure de l'intervalle de recherche, lue au clavier
  - Output
    - ✓ l'affichage à l'écran de tous les nombres parfaits ∈ [1, `nMax`[
  - Caractérisation de l'Input
    - ✓ `nMax`, la borne supérieure de l'intervalle de recherche
      - ✓ `nMax` ∈ ℕ
      - ‣ `unsigned int nMax;`

Ex 2: Nombre Parfait (3)

- Étape 2: Analyse
- On peut envisager 4 SPs
  - SP_1: Lecture au clavier
    - ✓ lire la valeur de `nMax` au clavier
  - SP_2: Enumération et Affichage
    - ✓ énumérer tous les valeurs de `n` ∈ {1, 2, 3, ..., **`nMax`**-1} et afficher les nombres parfaits
  - SP_3: Vérification
    - ✓ vérifier, pour une valeur **`n`** donnée, si elle constitue ou non un nombre parfait
  - ✓ SP_4: Affichage
    - ✓ affichage du nombre parfait **`n`**

Ex 2: Nombre Parfait (4)

- Structuration des SPs
  - $SP_1 \rightarrow [(SP_3 \rightarrow SP_4) \subset SP_2)]$



Ex 2: Nombre Parfait (5)

- Étape 3: Ecriture du code
- Canevas général du code

```
#include <stdio.h>

int main(){
  unsigned int nMax;

  //déclaration de variables supplémentaires

  //résolution des 3 SPs

}//fin programme
```

Ex 2: Nombre Parfait (6)

- Résolution SP_2 (énumération et affichage)
- Définition
  - Input
    - ✓ `nMax`
      - fourni par le SP_1
  - Output
    - ✓ tous les nombres `n` ∈ [1, `nMax`[ont été énumérés et affichés
  - Caractérisation de l'Input
    - ✓ `nMax`
      - donné par le SP_1

Ex 2: Nombre Parfait (7)

- Il faut faire une boucle pour l'itération

1. déclaration compteur
   - ✓ `unsigned int n = 1;`
2. nombre de tours dans la boucle?
   - ✓ `nMax-1`
3. Gardien de Boucle?
   - ✓ `n < nMax`
4. Corps de Boucle
   - ✓ vérifier si `n` est un nombre parfait
     - SP_3!!
   - ✓ afficher n si parfait
     - SP_4!!
   - ✓ incrémenter `n`

Ex 2: Nombre Parfait (8)

- Code SP_2

```
unsigned int n;

for(n=1; n<nMax; n++){
  //application du SP3
  //application du SP4
}//fin for - n
```


Ex 2: Nombre Parfait (9)

- Résolution SP_3 (vérification) + SP_4 (affichage)
- Définition
  - Input
    - ✓ `n`, le nombre à vérifier
  - Output
    - ✓ afficher `n` à l'écran si c'est un nombre parfait
    - ✓ rien sinon
  - Caractérisation de l'Input
    - ✓ `n`
      - donné par le SP_2
- Idée de solution
  - énumérer tous les diviseurs `div` de `n`
    - ✓ envisager tous les cas de `1` à `n-1`
    - ✓ boucle!
  - calculer la somme `som` de ces diviseurs
  - comparer `som` avec `n`

Ex 2: Nombre Parfait (10)

- Code SP_3 + SP_4

```
unsigned int div, som=0;

for(div=1; div<n; div++){
  if(!(n % div))
    som += div;
}//fin for - div

if(som==n)
  printf("%u\n", n);
```


Ex 2: Nombre Parfait (11)

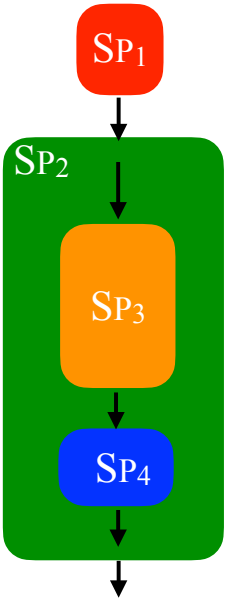
- Résolution SP_1 (lecture au clavier)
- Définition
  - Input
    - ✓ /
  - Output
    - ✓ `nMax` a été lu au clavier
  - Caractérisation de l'Input
    - ✓ /
- Code SP_1

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

Ex 2: Nombre Parfait (12)

```
#include <stdio.h>

int main(){
  unsigned int nMax, n , som, div ;

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

  for(n=1; n<nMax; n++){
    som = 0;
    for(div=1; div<n; div++){
      if(!(n % div))
        som += div;
    }//fin for - div
    if(som==n)

      printf("%u\n", n);
  }//fin for - n
}//fin programme
```

SP1, SP2, SP3, SP4



Ex 2: Nombre Parfait (13)

- Peut-on améliorer (i.e., rendre plus efficace) le SP3 (vérification)?
- Objectif: réduire "l'espace de recherche"
- Idée
  - tous les nombres entiers sont divisibles par 1
    - ✓ énumération des diviseurs à partir de `div`=2
    - ✓ commencer la recherche à partir de `n`=2 (1 ≠ parfait)
  - pour une valeur donnée de `n`
    - ✓ plus grand diviseur à considérer est égal à ⌊`n`/2⌋
  - quand `som > n`
    - ✓ arrêt énumération des diviseurs pour la valeur courante de `n`

Ex 2: Nombre Parfait (14)

```
#include <stdio.h>

int main(){
  unsigned int nMax, n, som, div;

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

  for(n=2; n<nMax; n++){
    som = 1;

    for(div=2; div<=n/2 && som <=n; div++){
      if(!(n % div))
        som += div;
    }//fin for - div
    if(som==n)
      printf("%u\n", n);
  }//fin for - n
}//fin programme
```

SP1

SP3

SP2

SP4

Ex 2: Nombre Parfait (15)

- Peut-on encore faire mieux?
  - observation
    - ✓ Si `div` est un diviseur de `n`, alors `n/div` l'est également
  - on peut dès lors énumérer les diviseurs de `n` (sauf 1 et `n`) de la façon suivante
    - ✓ considérer toutes les valeurs de `div` $\in [2, \lfloor \sqrt{n} \rfloor]$ qui divisent `n`
    - ✓ pour chacune de ces valeurs, aussi considérer `div' = n/div`
  - Attention
    - ✓ si `n` est un *carré parfait*, alors ne pas considérer deux fois le diviseur `div = div' =` $\sqrt{n}$

Ex 2: Nombre Parfait (16)

```
#include <stdio.h>
int main(){
  unsigned int nMax, n, som, div, div2;
  printf("Entrez une valeur pour nMax: ");
  scanf("%u", &nMax);
  for(n=2; n<nMax; n++){
    som = 1;
    for(div=2; div * div <= n && som <= n; div++){
      if(!(n % div)){
        som += div;
        div2 = n/div;
        if(div2 != div)
          som += div2;
      }
    }//fin for - div
    if(som==n)
      printf("%u\n", n);
  }//fin for - n
}//fin programme
```

SP1

SP2

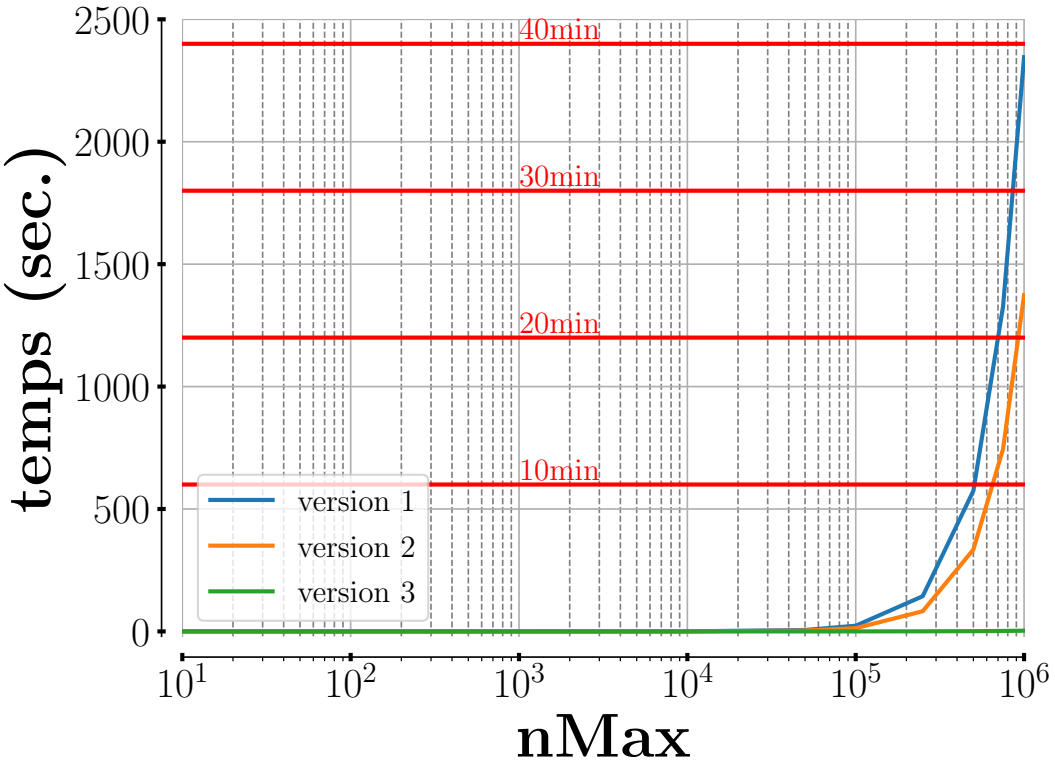
SP3

SP4

Ex 2: Nombre Parfait (17)

- Intérêt de se prendre la tête?
  - Après tout, la version 1 fonctionne très bien et la solution est assez intuitive
- Evaluation expérimentale des performances des 3 versions
  - Intel i7, 2Ghz
    - ✓ quadcore
  - 8Go RAM
  - implémentation C
    - ✓ `nMax` $\in [10, 10^6]$
    - ✓ le temps nécessaire à chaque version est obtenu grâce à `time.h`
      - cfr. INFO0030

Ex 2: Nombre Parfait (18)



Exercice

- Ecrire un programme qui affiche la table de multiplication des nombres de 1 à 10 sous la forme suivante:
  - note: découpe en sous-problèmes requise

	I	1	2	3	4	5	6	7	8	9	10
1	I	1	2	3	4	5	6	7	8	9	10
2	I	2	4	6	8	10	12	14	16	18	20
3	I	3	6	9	12	15	18	21	24	27	30
4	I	4	8	12	16	20	24	28	32	36	40
5	I	5	10	15	20	25	30	35	40	45	50
6	I	6	12	18	24	30	36	42	48	54	60
7	I	7	14	21	28	35	42	49	56	63	70
8	I	8	16	24	32	40	48	56	64	72	80
9	I	9	18	27	36	45	54	63	72	81	90
10	I	10	20	30	40	50	60	70	80	90	100

Agenda

- Chapitre 3: Méthodologie
  - Schéma Méthodologique
  - Définition du Problème
  - Analyse du Problème
  - Invariant de Boucle
    - ✓ Intuition
    - ✓ Définition
    - ✓ Règles pour un Invariant Graphique
    - ✓ Bestiaire
    - ✓ Comment Trouver un Invariant Graphique?
    - ✓ Construction par Invariant Graphique
    - ✓ Exemple Complet 1: Factorielle
    - ✓ Exemple Complet 2: Renversement d'un Nombre
  - Fonction de Terminaison

Intuition

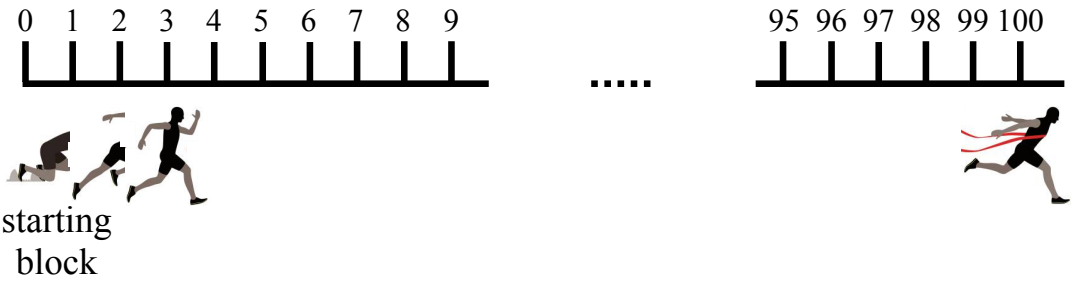
- On cherche à modéliser une course de Usain Bolt
  - 100m
- Une course est composée
  - d'une position de départ
    - ✓ starting block
  - la course proprement dite
    - ✓ chaque pas fait avancer le coureur d'un mètre
    - ✓ modélisation discrète
  - l'arrivée

© LORFM

Intuition (2)

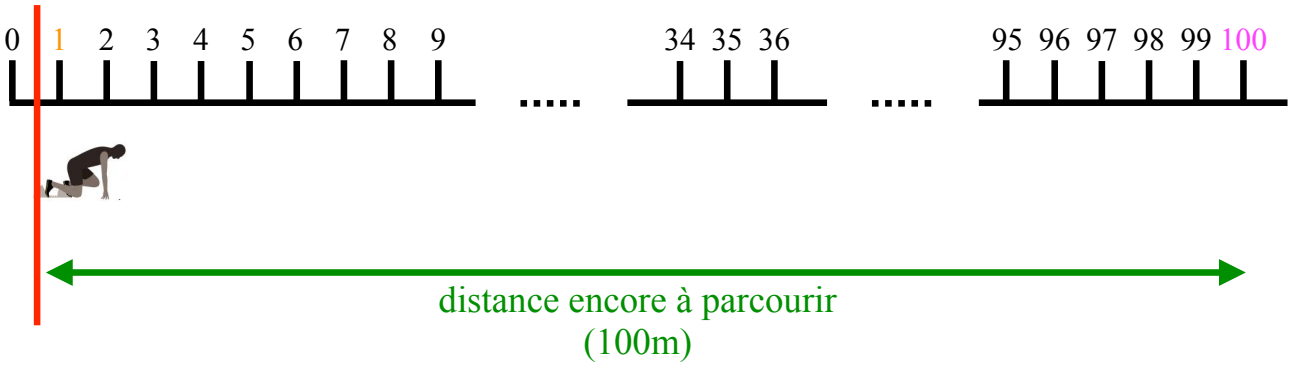
- Anatomie d'un sprint sur 100m

signal de départ



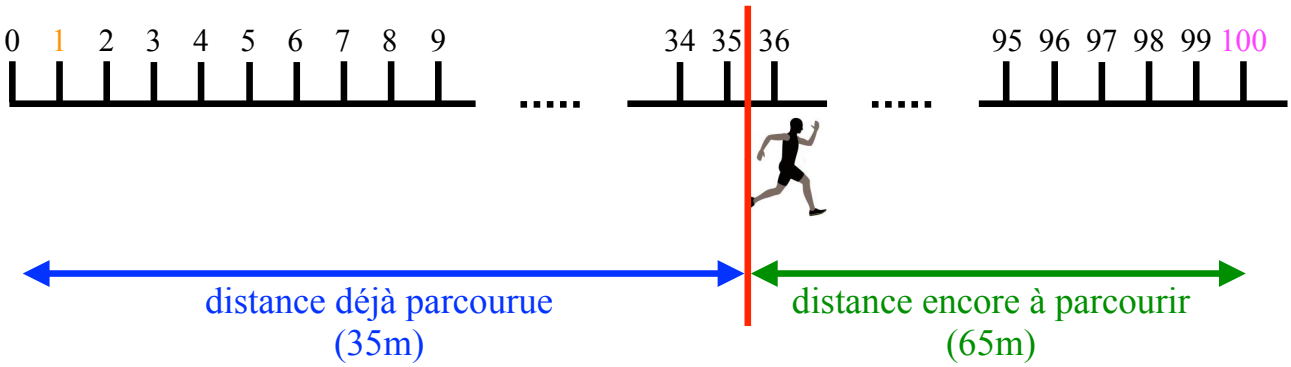
Intuition (3)

- Analyse de la course par un journaliste sportif
  - **photo** prise en cours de sprint



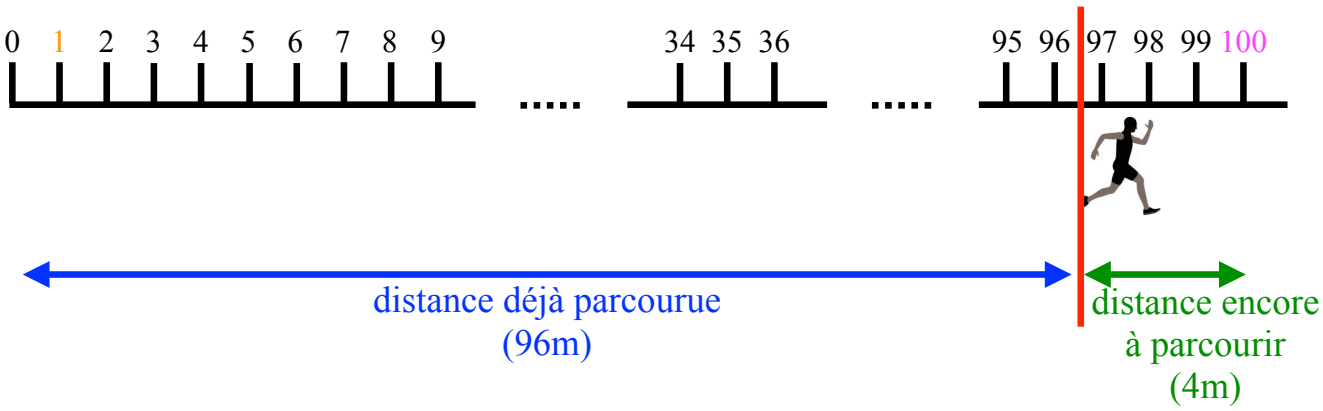
Intuition (4)

- Analyse de la course par un journaliste sportif (cont')
  - **photo** prise en cours de sprint



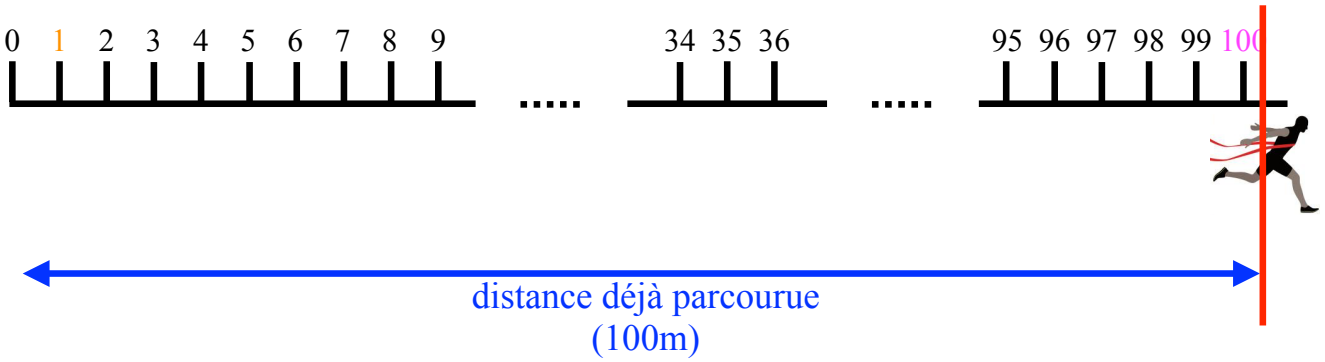
Intuition (5)

- Analyse de la course par un journaliste sportif (cont')
  - **photo** prise en cours de sprint



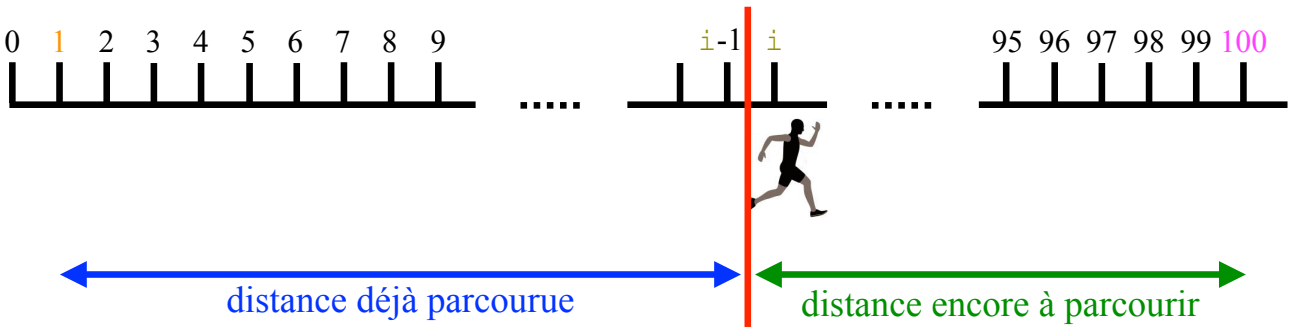
Intuition (6)

- Analyse de la course par un journaliste sportif (cont')
  - **photo** prise en cours de sprint



Intuition (7)

- Analyse de la course par un journaliste sportif (cont')
  - **photo** prise en cours de sprint
  - généralisation



Intuition (8)

- Cette photo généralisée décrivant la distance déjà parcourue par Usain Bolt est l'**Invariant de la Boucle**

Définition

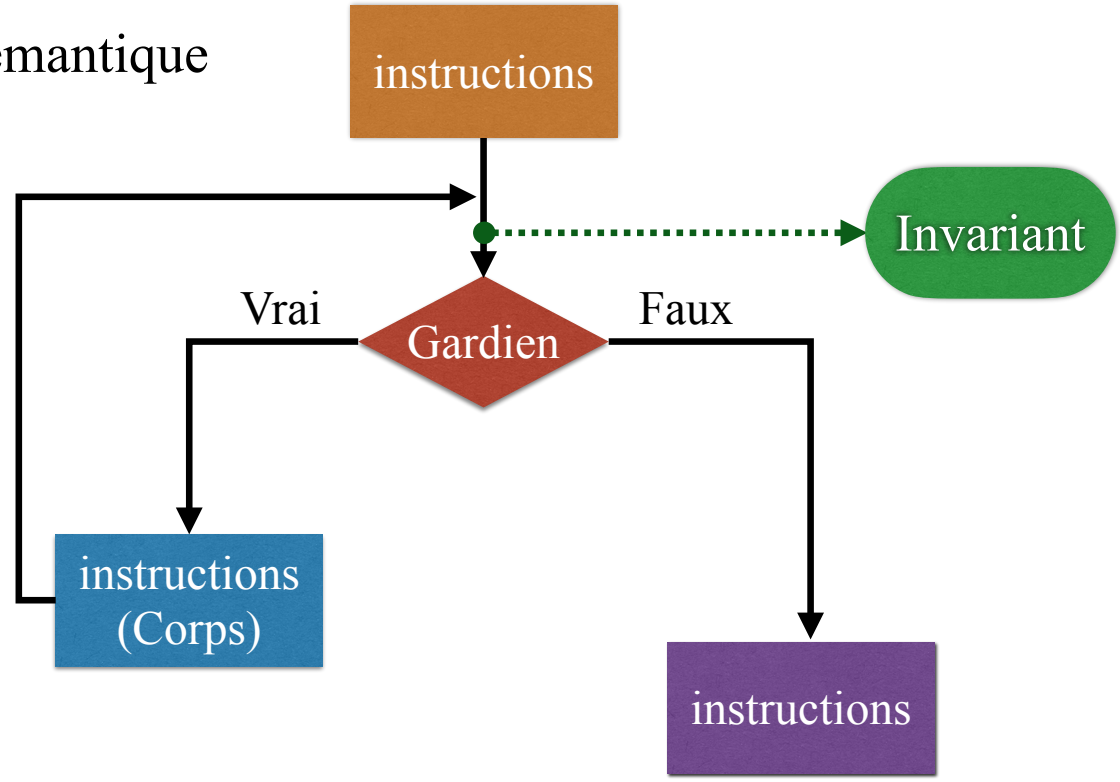
- **Invariant de Boucle**
  - *propriété* vérifiée à chaque exécution de la boucle
  - résumé de tout ce qui a déjà été calculé *jusqu'à maintenant*
  - R. W. Floyd. *Assigning Meanings to Programs*. In Proc. Symposium on Applied Mathematics. 1967.
  - C. A. R. Hoare. *An Axiomatic Basis for Computer Programming*. In Communications of the ACM, 12(10), pg.576-580. 1969.
- Raisonnement sur la boucle avant de l'écrire
  - Invariant *informel*
    - ✓ Invariant Graphique
      - O. Astrachan. *Pictures as Invariants*. In Proc. ACM Technical Symposium on Computer Science Education (SIGCSE). March 1991.
      - G. Brieven, S. Liénardy, L. Malcev, B. Donnet. *Graphical Loop Invariant Based Programming*. In Proc. Formal Method Teaching (FMTea). March 2023.
    - ✓ Invariant en français
      - W. C. Tam. *Teaching Loop Invariants to Beginners by Examples*. In Proc. ACM Technical Symposium on Computer Science Education (SIGCSE). March 1992.
  - Invariant *formel*
    - ✓ prédicat
    - ✓ cfr. INFO0947

Définition (2)

- L'Invariant de Boucle exprime/résume ce qui a déjà été calculé par la boucle *jusqu'à maintenant*
  - à chaque évaluation du Gardien de Boucle

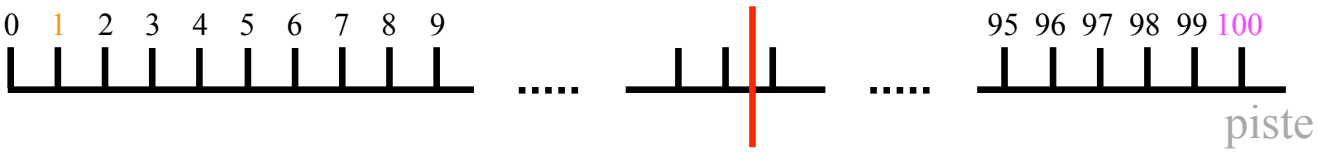
Définition (3)

- Sémantique



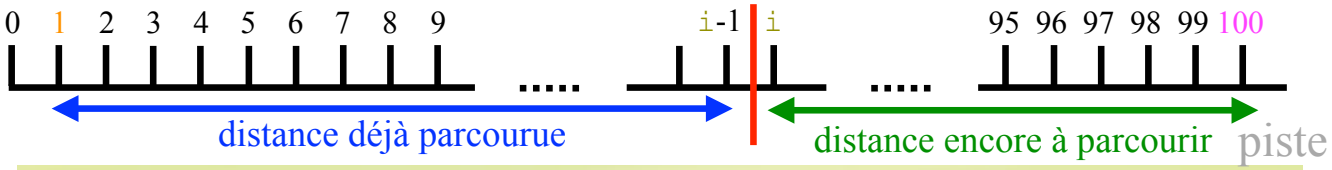
Règles

- Règles pour un bon Invariant Graphique
  1. réaliser un dessin pertinent et le nommer
  2. placer sur le dessin les bornes de début et de fin
     - ✓ on peut aussi identifier la taille de la structure
  3. placer une (ou plusieurs) ligne(s) de démarcation qui sépare(nt) ce qui a déjà été calculé dans les itérations précédentes et ce qu'il reste à faire
     - ✓ permet(tent) de création des zones sur lesquelles on va pouvoir décrire des propriétés



Règles (2)

- Règles pour un bon Invariant Graphique (cont')
  4. étiqueter chaque ligne de démarcation avec une variable d'itération
     - ✓ à gauche ou à droite
  5. décrire ce que les itérations précédentes ont déjà calculé en utilisant des variables
     - ✓ ces variables devront se retrouver dans le programme
     - ✓ questions à se poser
       - où est stocké ce résultat?
       - comment peut-on décrire ce résultat (forcément partiel)?
  6. identifier ce qu'il reste à faire dans les itérations suivantes
  7. toutes les structures et variables identifiées sont présentes dans le code.



Règles (3)

- Règles pour un bon Invariant Graphique (all together)
  1. réaliser un dessin pertinent et le nommer
  2. placer sur le dessin les bornes de début et de fin
     - on peut aussi identifier la taille de la structure
  3. placer une (ou plusieurs) ligne(s) de démarcation qui sépare(nt) ce qui a déjà été calculé dans les itérations précédentes et ce qu'il reste à faire
  4. étiqueter chaque ligne de démarcation avec une variable d'itération
     - à gauche ou à droite
  5. décrire ce que les itérations précédentes ont déjà calculé en utilisant des variables
     - ces variables devront se retrouver dans le programme
     - questions à se poser
       - où est stocké ce résultat?
       - comment peut-on décrire ce résultat (forcément partiel)?
  6. identifier ce qu'il reste à faire dans les itérations suivantes
  7. toutes les structures et variables identifiées sont présentes dans le code.

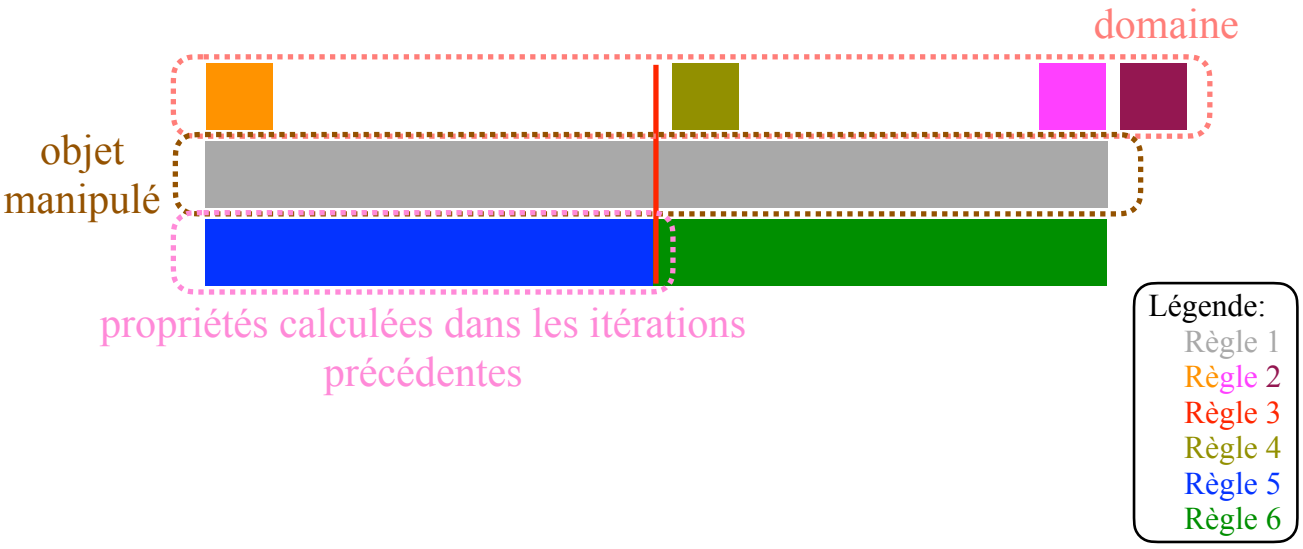
Bestiaire

- Code couleur

Éléments du dessin	Code Couleur	Règle(s) Associée(s)
Nom de la structure		Règle 1
Borne minimale		Règle 2
Borne maximale		
Taille de la structure		
Lignes de démarcation		Règle 3
Étiquette des lignes de démarcation		Règle 4
Ce qui a été réalisé jusqu'à maintenant		Règle 5
zones "à faire"		Règle 6
Propriétés qui sont conservées		Règle 5 + Règle 6

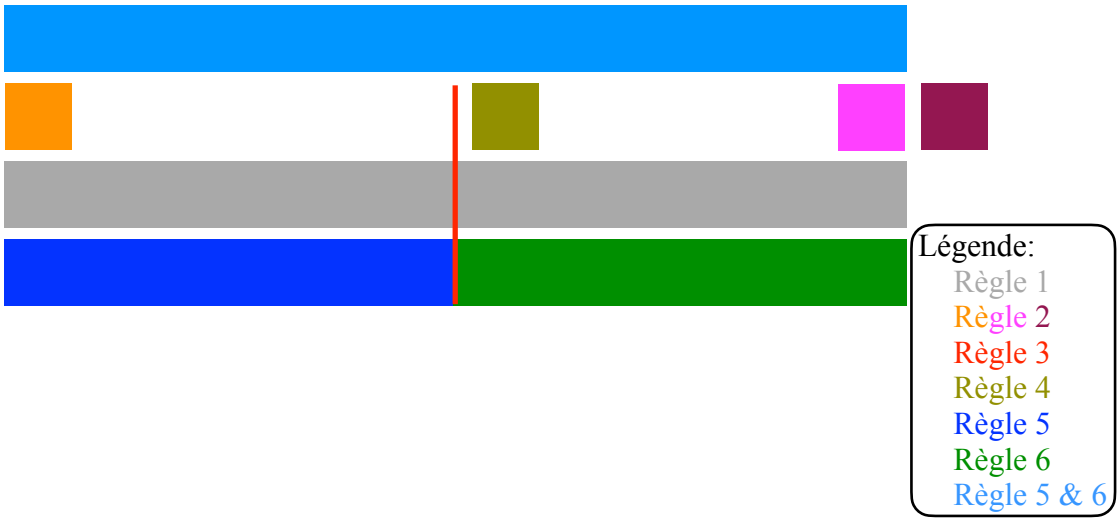
Bestiaire (2)

- Formats génériques d'un Invariant Graphique
  1. Invariant Graphique simple



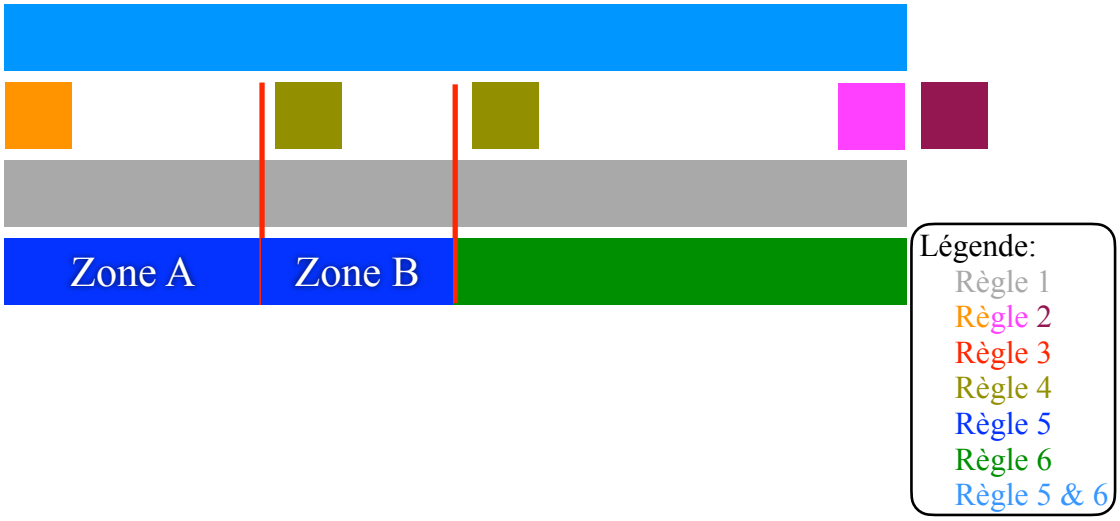
Bestiaire (3)

- Formats génériques d'un Invariant Graphique (cont')
  2. Invariant Graphique simple avec propriété(s) conservée(s)



Bestiaire (4)

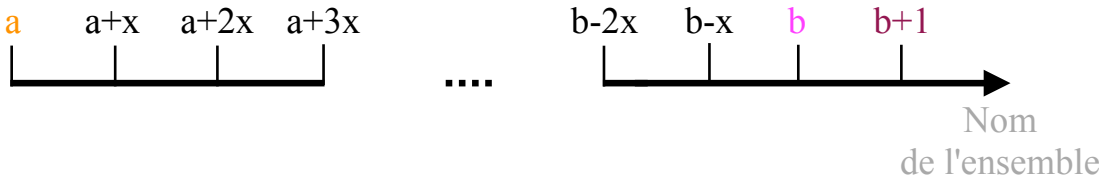
- Formats génériques d'un Invariant Graphique (cont')
  3. Invariant Graphique simple avec plusieurs zones traitées (avec propriété(s) conservée(s))



Bestiaire (5)

- Quels objets manipule-t-on?

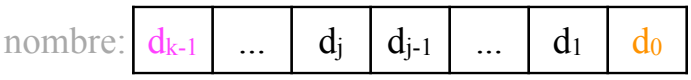
1. Ligne graduée
   - permet la représentation d'un ensemble ordonné
     - ✓ $\mathbb{N}, \mathbb{Z}$
   - Construction
     - ✓ chaque tiret représente une valeur
     - ✓ chaque valeur est décalée d'un même pas
     - ✓ la flèche indique l'ordre croissant des valeurs



Bestiaire (6)

- Quels objets manipule-t-on? (cont')

2. Représentation d'un nombre
   - peu importe la base
     - ✓ binaire, décimal, hexadécimal, ...
   - Construction
     - ✓ cfr. Introduction (Système de Numération)
     - ✓ séquence de symboles d_j
     - ✓ le symbole de poids faible est à droite
     - ✓ le symbole de poids fort est à gauche



Bestiaire (7)

- Quels objets manipule-t-on? (cont')

3. Tableau
   - ✓ cfr. Chap. 5
4. Matrice
   - ✓ cfr. Chap. 5
5. Fichier
   - ✓ cfr. INFO0947
6. Liste
   - ✓ cfr. INFO0947
7. Pile
   - ✓ cfr. INFO0947
8. File
   - ✓ cfr. INFO0947

Bestiaire (8)

- Outil GLIDE
  - https://cafe.uliege.be
  - développé par Simon Liénardy & Lev Malcev
    - G. Brieven, S. Liénardy, L. Malcev, B. Donnet. *Graphical Loop Invariant Based Programming*. In Proc. Formal Method Teaching (FMTea). March 2023.
  - permet la création et la manipulation d'un Invariant Graphique
- Vous êtes invités à utiliser l'outil
  - en séance de répétition et en labos
  - durant les séances CDB
  - à la maison, pour vous exercer/travailler le cours
    - préparation interro
    - préparation examen
  - à la maison, pour vous aider dans les Challenges

Trouver un Invariant Graphique

- Comment trouver un Invariant Graphique?
- Il existe plusieurs techniques
  - intuition
  - raisonnement inductif
  - combiner Input et Output
  - travailler sur l'Output pour en tirer une situation générale
    - **éclatement de la Postcondition**
    - **constant relaxation**
      - C. A. Furia, B. Meyer, S. Velder. *Loop Invariants: Analysis, Classification, and Examples*. In ACM Computing Surveys, 46(3), pg. 1-51. January 2014.
      - G. Brieven, S. Liénardy, L. Malcev, B. Donnet. *Graphical Loop Invariant Based Programming*. In Proc. Formal Method Teaching (FMTea). March 2023.

Trouver un Invariant Graphique (2)

- Éclatement de la Postcondition
  1. faire un dessin représentant l'Output pour faire apparaitre une situation particulière
     - ✓ appliquer les règles 1, 2 et 5
  2. appliquer les règles sur le dessin pour faire apparaitre une situation générale
     - ✓ appliquer la règle 3 fait apparaitre une situation générale
     - ✓ appliquer la règle 4 pour introduire les délimitations de zones
     - ✓ introduire la règle 6 pour faire apparaitre "ce qu'il reste à faire"
     - ✓ modifier la règle 5 de l'Output pour faire apparaitre un résultat intermédiaire
       - introduction d'une (ou plusieurs) variable(s) pour les calculs "en cours de boucle"

Trouver un Invariant Graphique (3)

- Exemple
  - calcul de la factorielle de n
  - notation: $n!$
  - $\forall n \in \mathbb{N},\ n! =$
    - ✓ 1 si $n \leq 1$
    - ✓ $n \times (n-1)!$ sinon

$(n-1) \times (n-2)!$

$(n-2) \times (n-3)!$

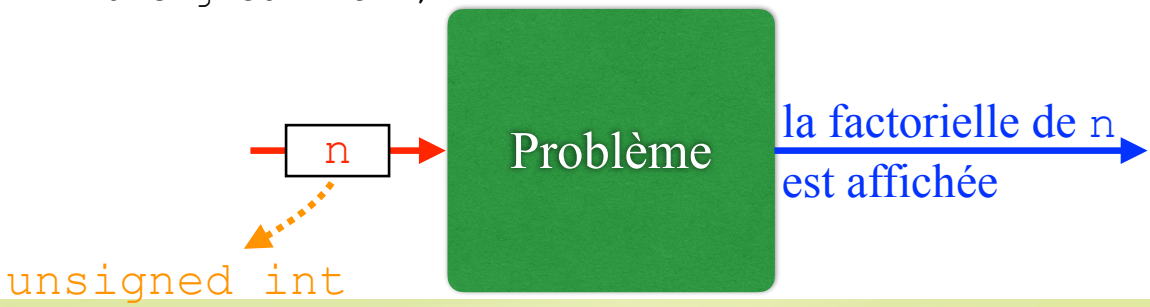
...

$n \times n-1 \times \ldots \times 2 \times 1$

n-1 multiplications

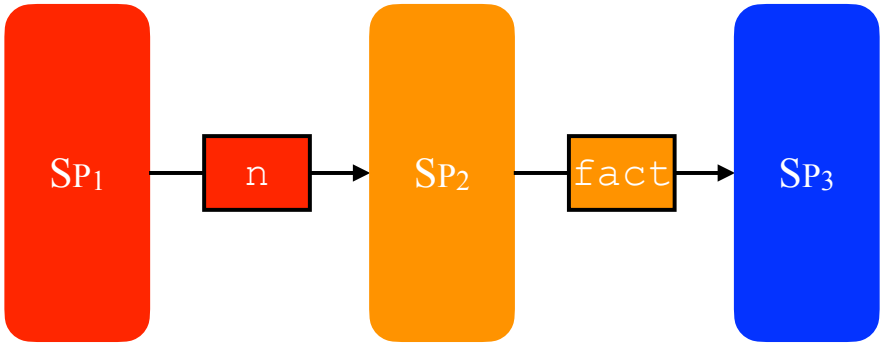
Trouver un Invariant Graphique (4)

- Définition du problème
  - Input
    - ✓ le nombre pour lequel il faut calculer la factorielle, lu au clavier
  - Output
    - ✓ le résultat de la factorielle est affiché à l'écran
  - Caractérisation de l'Input
    - ✓ `n`, le nombre pour lequel il faut calculer la factorielle
      - `n` ∈ ℕ
      - `unsigned int n;`



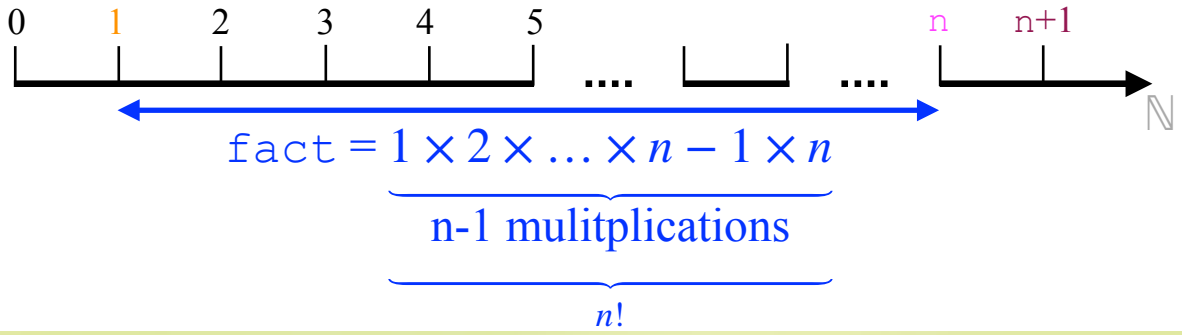
Trouver un Invariant Graphique (5)

- Analyse
  - SP1: lecture de `n` au clavier
  - SP2: calcul de la factorielle de **`n`** dans `fact`
  - SP3: affichage la **factorielle** à l'écran
  - SP1 → SP2 → SP3



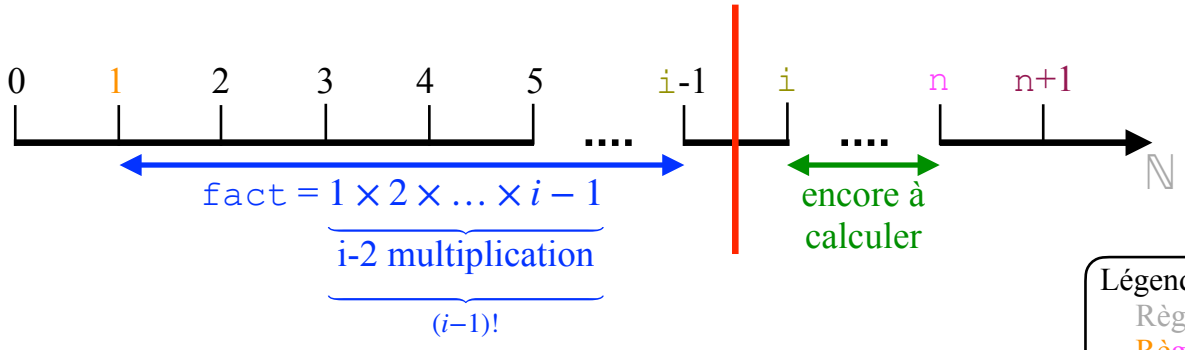
Trouver un Invariant Graphique (6)

- Définition SP_2
  - Input
    - ✓ `n` (obtenu par le SP_1)
  - Output
    - ✓ une variable `fact` contient `n`!
  - Caractérisation de l'Input
    - ✓ `unsigned int n;`
- Représentation graphique de l'Output



Trouver un Invariant Graphique (7)

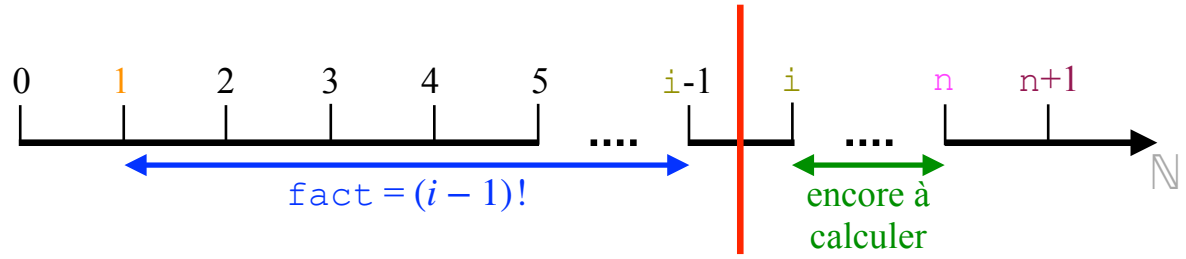
- De l'Output à l'Invariant Graphique



Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Trouver un Invariant Graphique (8)

- Invariant Graphique SP_2



Trouver un Invariant Graphique (9)

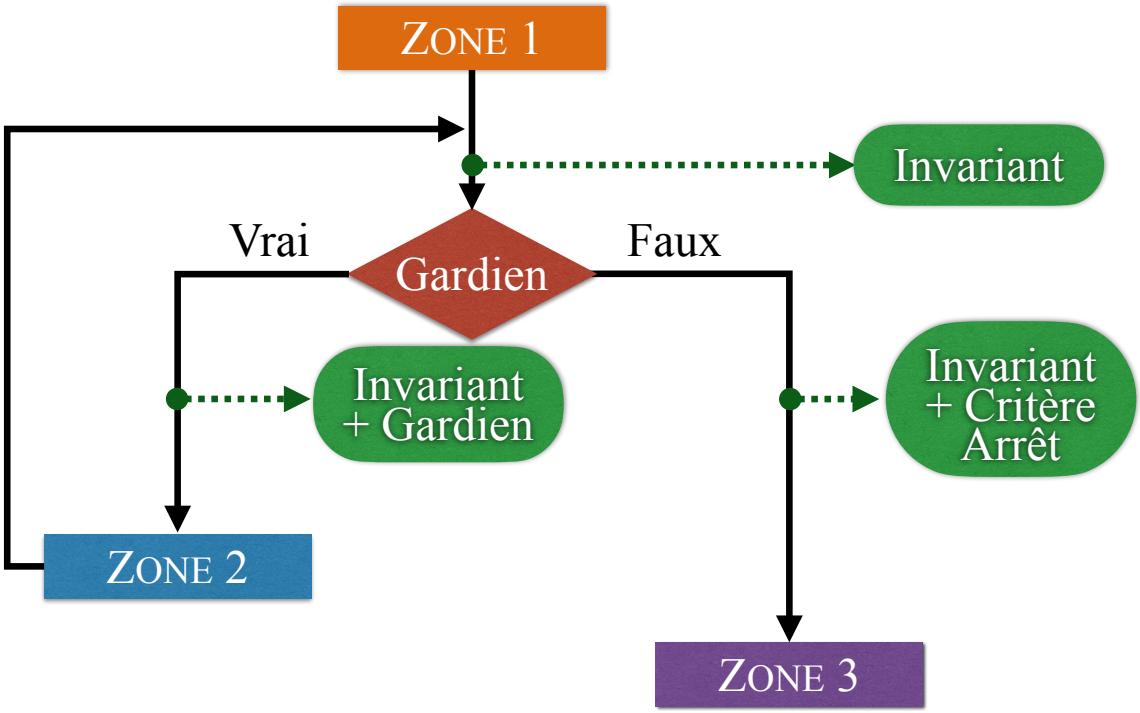
- Il est **obligatoire** que l'Invariant parle des variables "importantes" manipulées par la boucle
  - dans cet exemple, il s'agit
    - ✓ de l'indice de boucle, `i`
      - entre quelles bornes évolue `i`?
    - ✓ de la variable servant au produit cumulatif, `fact`
      - que vaut `fact` maintenant?

Construction par Invariant

- L'Invariant est la base d'une construction rigoureuse de programme
  - D. G. Kourie, B. W. Watson. *Correctness-by-Construction Approach to Programming*. In Springer. January 2012
  - G. Brieven, S. Liénardy, L. Malcev, B. Donnet. *Graphical Loop Invariant Based Programming*. In Proc. Formal Method Teaching (FMTea). March 2023.
- Partant de la sémantique de l'Invariant de Boucle
  - on peut définir une *stratégie* de résolution du problème
    - stratégie == élaboration d'un plan
  - portant sur le comportement des instructions
    - avant la boucle
      - ZONE 1
    - le corps de la boucle
      - ZONE 2
    - après la boucle
      - ZONE 3

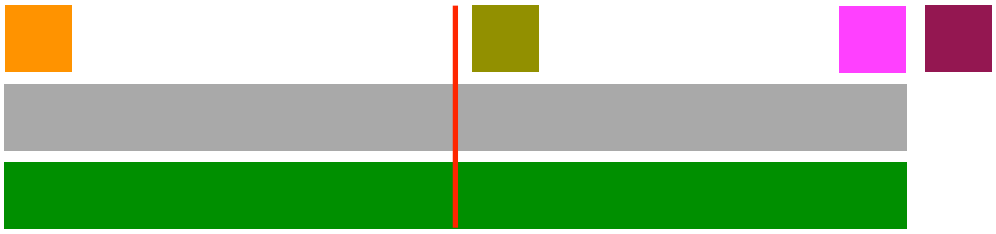
Construction par Invariant (2)

- Invariant et Zones



Construction par Invariant (3)

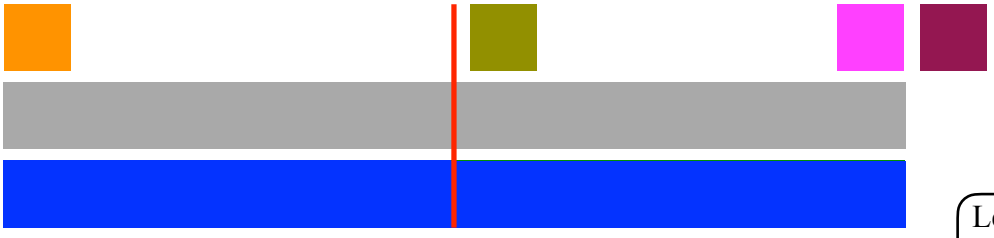
- *ZONE 1*
  - le code d'initialisation (avant la boucle) doit amener l'Invariant de Boucle
    - ✓ l'Invariant de Boucle sera vrai à la toute 1ère évaluation du Gardien de Boucle
      - i.e., avant d'entrer la 1ère fois dans le Corps de Boucle
  - l'Invariant de Boucle indique donc
    - ✓ les variables dont j'ai besoin
    - ✓ comment les initialiser pour pouvoir aborder la boucle
  - l'Invariant est donc bien une stratégie me permettant d'être dans les "starting blocks" pour la boucle



Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Construction par Invariant (4)

- Critère d'Arrêt et Gardien de Boucle
  - on sort de la boucle quand l'entièreté du travail a été fait
  - l'Invariant de Boucle permet de trouver le Critère d'Arrêt
    - ✓ le Gardien de Boucle est la négation du Critère d'Arrêt



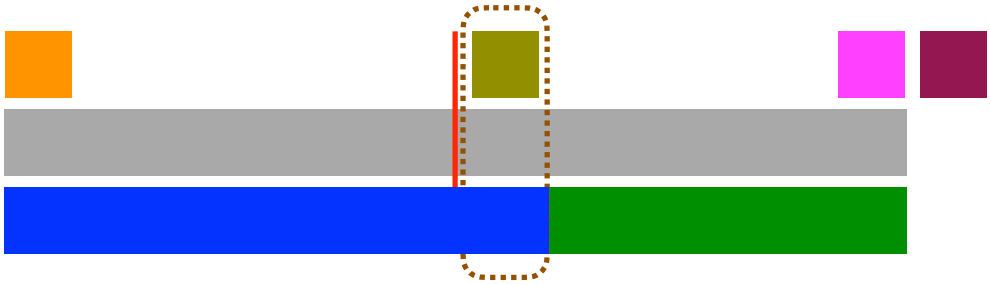
Critère: ==

Gardien: <

Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Construction par Invariant (5)

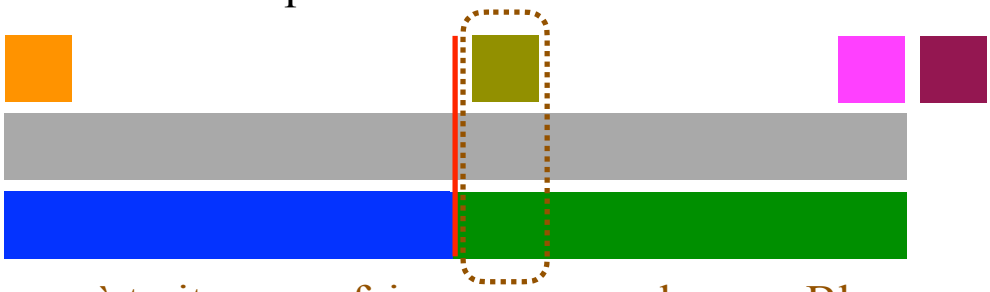
- *ZONE 2*
  - j'entre dans le Corps de Boucle
    - ✓ le Gardien de Boucle vient d'être évalué à vrai
  - mon Invariant de Boucle est donc vrai et, en plus, le Gardien est vrai
    - ✓ sur base de ces 2 propriétés, il faut trouver une suite d'instructions qui permet de faire avancer le problème



à traiter pour faire progresser la zone Bleue

Construction par Invariant (6)

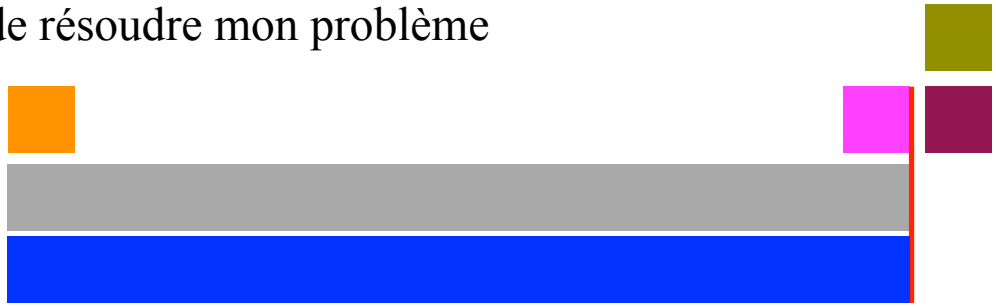
- *ZONE 2* (cont')
  - après la dernière instruction du Corps de Boucle, le Gardien va être évalué
    - ✓ par définition, l'Invariant de Boucle doit de nouveau être vrai à ce moment là
    - ✓ la dernière instruction doit donc me permettre de restaurer l'Invariant de Boucle
  - l'Invariant de Boucle est bien une stratégie pour construire le Corps de Boucle



à traiter pour faire progresser la zone Bleue

Construction par Invariant (7)

- *ZONE 3*
  - le Gardien de Boucle vient d'être évalué à faux
    - ✓ je sors donc de la boucle
  - mon Invariant de Boucle doit toujours être vrai et, en plus, le Critère d'Arrêt est atteint
  - sur base de ces 2 propriétés, je dois trouver une suite d'instructions permettant de clôturer mon problème
    - ✓ de la sortie de la boucle vers l'Output
  - l'Invariant de Boucle est bien une stratégie me permettant de résoudre mon problème



Construction par Invariant (8)

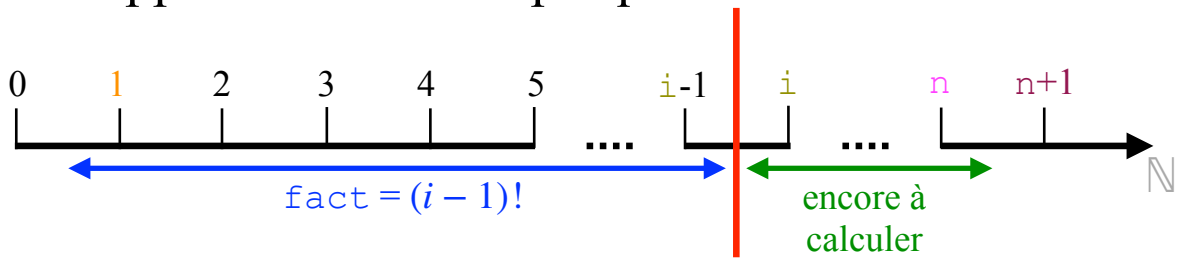
- En résumé, l'Invariant de Boucle
  - exprime ce qui a déjà été calculé, jusqu'à présent, par la boucle
- En particulier, l'Invariant de Boucle
  - décrit
    - ✓ une propriété respectée par
      - les variables du programme
      - les constantes
      - les structures de données
    - ✓ les liens qu'elles entretiennent ensemble

Construction par Invariant (9)

- Attention, l'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
    - ✓ il doit être vrai pendant et après l'itération
  - ne garantit pas que la boucle se termine
    - ✓ Fonction de Terminaison
    - ✓ cfr. Slides 114 → 124
  - n'est pas une assurance de l'efficacité du programme

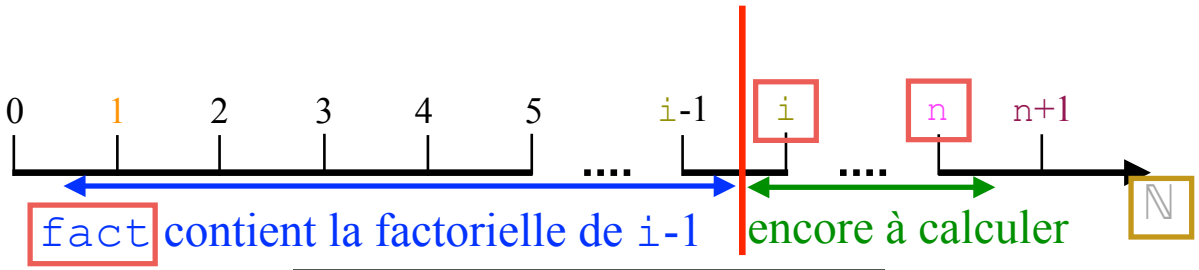
Factorielle

- Définition, Analyse et Invariant Graphique déjà faits
  - cfr. Slides 78 → 81
- Rappel Analyse
  - SP_1: lecture de `n` au clavier
  - SP_2: calcul de la factorielle de **`n`** dans `fact`
  - SP_3: affichage la **factorielle** à l'écran
  - $SP_1 \rightarrow SP_2 \rightarrow SP_3$
- Rappel Invariant Graphique SP_2



Factorielle (2)

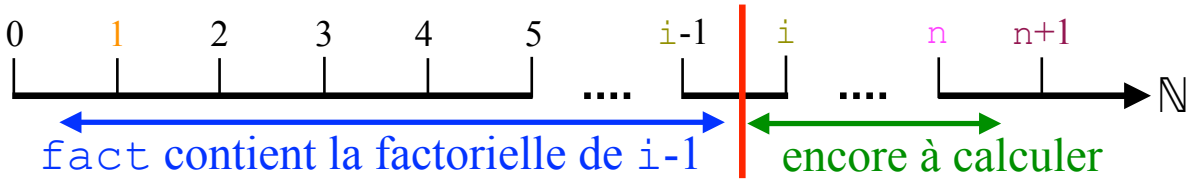
- Construction du code sur base de l'Invariant
  - construction de la *ZONE 1*
    - ✓ déclaration et initialisation des variables avant la boucle
      (1) quelles sont les variables dont j'ai besoin?




```
#include <stdio.h>

int main(){
  unsigned int i, fact, n;

  //à suivre
}//fin programme
```


Factorielle (3)

- Construction du code sur base de l'Invariant
  - construction de la *ZONE 1*
    - ✓ déclaration et initialisation des variables avant la boucle
      (2) quelles sont les valeurs initiales de ces variables?



fact contient la factorielle de i-1

0!

```
int main(){
  i = 1;
  fact = 1;
  scanf("%u", &n);
  //à suivre
}//fin programme
```

Factorielle (4)

- Construction du code sur base de l'Invariant (cont.)
  - construction du *Gardien de Boucle*
    - ✓ variable(s) d'itération et valeur(s) maximale(s) donnent le Critère d'Arrêt
    - ✓ le Gardien de Boucle est donné par la négation du Critère d'Arrêt
      - (1) quelle est la variable d'itération?
      - (2) quelle est sa valeur maximale?

Condition d'arrêt:

$i == n+1$

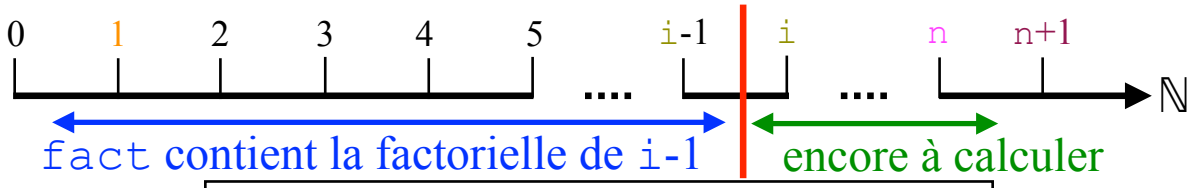
$\Rightarrow !(i == n+1)$

$\Rightarrow i <= n$

```
int main(){
  //ZONE 1

  while(i <= n){
    //à suivre
  }//fin while - i
  //à suivre
}//fin programme
```


Factorielle (5)

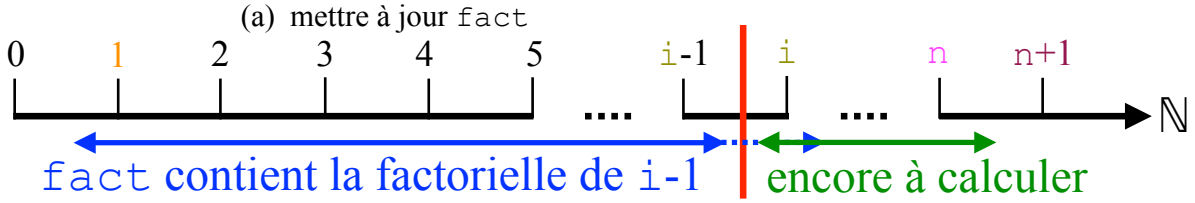
- Construction du code sur base de l'Invariant (cont.)
  - construction de la *ZONE 2*
    - ✓ construire le Corps de la Boucle
      - (1) l'Invariant est vrai
      - (2) le Gardien de Boucle est vrai
      - (3) dériver les instructions du Corps de la Boucle




```
int main(){
  //ZONE 1

  while(i <= n){
    //???
  }//fin while - i
  //à suivre
}//fin programme
```

Factorielle (6)

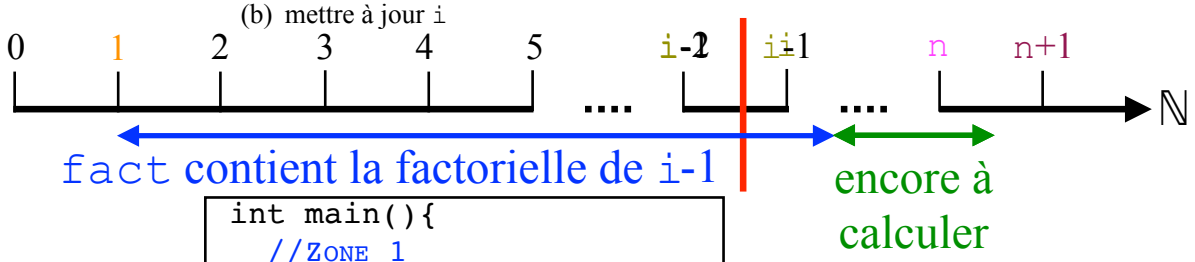
- Construction du code sur base de l'Invariant (cont.)
  - construction de la *ZONE 2*
    - ✓ construire le Corps de la Boucle
      (1) l'Invariant est vrai
      (2) le Gardien de Boucle est vrai
      (3) dériver les instructions du Corps de la Boucle
        (a) mettre à jour `fact`




```
int main(){
  //ZONE 1

  while(i <= n){
    fact *= i;
  }//fin while - i
  //à suivre
}//fin programme
```


Factorielle (7)

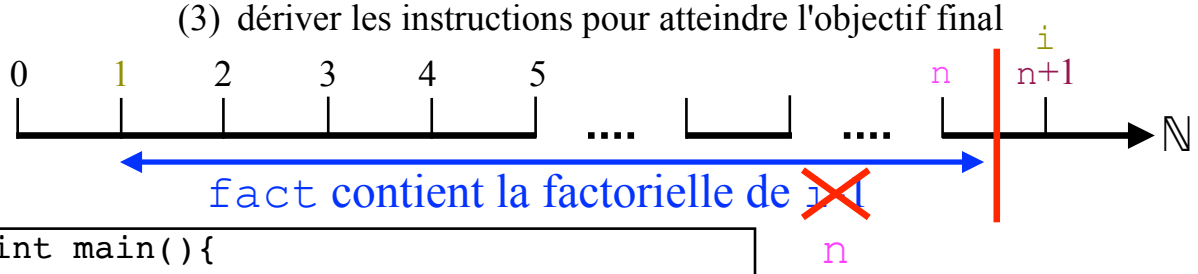
- Construction du code sur base de l'Invariant (cont.)
  - construction de la *Zone 2*
    - ✓ construire le Corps de la Boucle
      (1) l'Invariant est vrai
      (2) le Gardien de Boucle est vrai
      (3) dériver les instructions du Corps de la Boucle
        (a) mettre à jour `fact`
        (b) mettre à jour `i`




```
int main(){
  //ZONE 1

  while(i <= n){
    fact *= i;
    i++;
  }//fin while - i
  //à suivre
}//fin programme
```

Factorielle (8)

- Construction du code sur base de l'Invariant (cont.)
  - construction de la *ZONE 3*
    - ✓ construire le code après la boucle
      (1) l'Invariant est vrai
      (2) le Critère d'Arrêt est atteint
      (3) dériver les instructions pour atteindre l'objectif final




```
int main(){
  //ZONE 1

  while(i <= n){
    //ZONE 2
  }//fin while - i
  printf("factorielle: %u\n", fact);
}//fin programme
```


Factorielle (9)

- Code Complet

```
#include <stdio.h>

int main(){
  unsigned int i=1, fact=1, n;
  scanf("%u", &n);

  while(i<=n){
    fact *= i;
    i++;

  }//fin while - i

  printf("factorielle: %u\n", fact);
}//fin programme
```

ZONE 1

ZONE 2

ZONE 3

Renversement Nombre

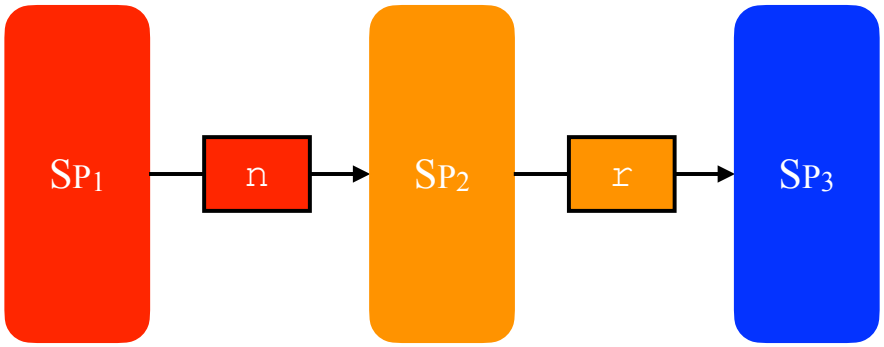
- Problème
  - renverser les chiffres d'un entier positif en base 10 lu au clavier et afficher à l'écran le nombre renversé
- Exemples
  - 35276 → 67253
  - 19 → 91
  - 3 → 3
  - 0 → 0

Renversement Nombre (2)

- Définition du problème
  - Input
    - ✓ le nombre à renverser, lu au clavier
  - Output
    - ✓ un nombre correspondant au renversement du nombre en entrée est affiché à l'écran
  - Caractérisation de l'Input
    - ✓ n, le nombre à renverser
      - $n \in \mathbb{N}$
      - `unsigned int n;`

Renversement Nombre (3)

- Analyse du problème
  - SP_1: lecture de `n` au clavier
  - SP_2: renversement de **`n`** dans `r`
  - SP_3: affichage de **`r`** à l'écran
  - $SP_1 \rightarrow SP_2 \rightarrow SP_3$

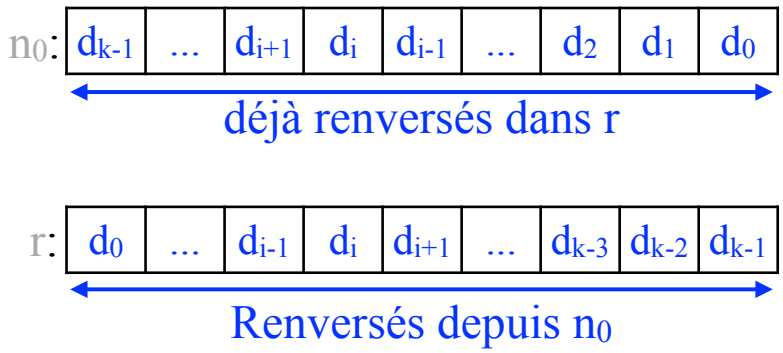


Renversement Nombre (4)

- Le SP_2 nécessite un traitement itératif
- Trouver un Invariant Graphique pour le SP_2
  - éclatement de la PostCondition
- Définition du SP_2
  - Input
    - ✓ `n`, le nombre à renverser
  - Output
    - ✓ `r` contient le renversement de `n`
  - Caractérisation des Inputs
    - ✓ $n \in \mathbb{N}$
      - `unsigned int n;`
    - ✓ $r \in \mathbb{N}$
      - `unsigned int r;`

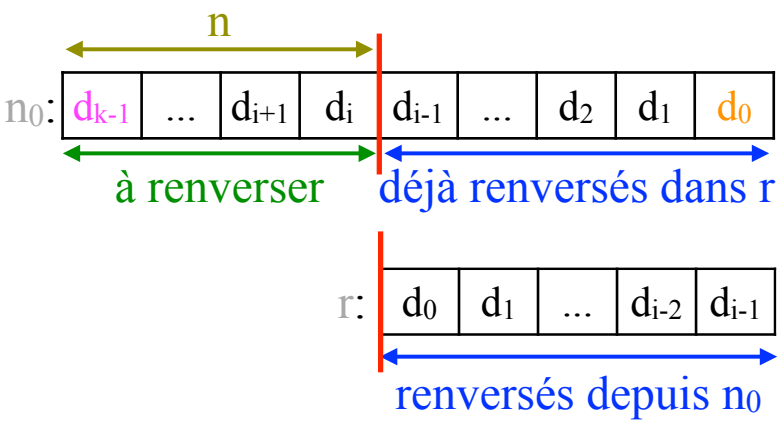
Renversement Nombre (5)

- Représentation graphique de l'Output du SP_2



Renversement Nombre (6)

- Construction de l'Invariant Graphique



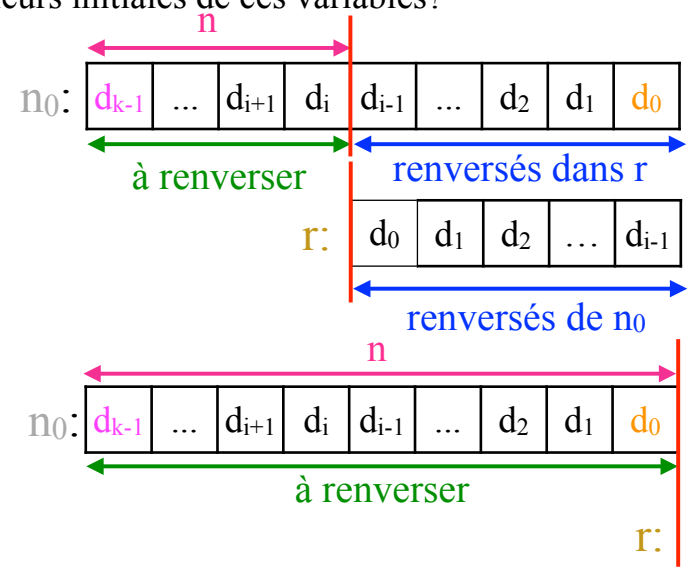
Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Renversement Nombre (7)

- Construction du code sur base de l'Invariant
  - *ZONE 1*
    - ✓ déclaration et initialisation des variables avant la boucle
      (1) quelles sont les variables dont j'ai besoin?
      (2) quelles sont les valeurs initiales de ces variables?
        (a) `n`
        (b) `r`

```
#include <stdio.h>

int main(){
  unsigned int r;
  unsigned int n;

  scanf("%u", &n);
  r = 0;

  //à suivre
}//fin programme
```

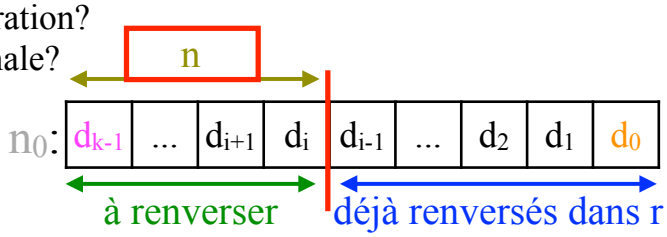
n lu au clavier (SP_1)



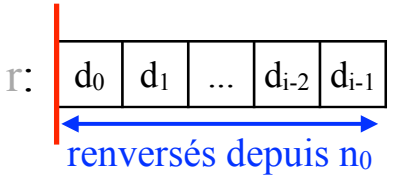
Renversement Nombre (8)

- Construction du code sur base de l'Invariant (cont.)
  - construction du *Gardien de Boucle*
    - ✓ variable(s) d'itération et valeur(s) maximale(s) donnent le Critère d'Arrêt
    - ✓ le Gardien de Boucle est donné par la négation du Critère d'Arrêt
      (1) quelle est la variable d'itération?
      (2) quelle est sa valeur minimale?

```
#include <stdio.h>

int main(){
  //ZONE 1

  while(n!=0){
    //à suivre
  }//fin while - n
  //à suivre
}//fin programme
```



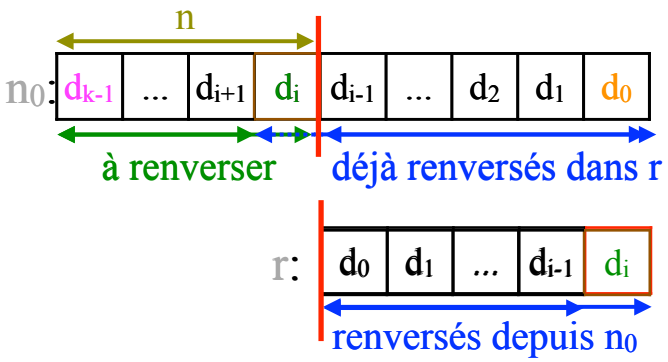
Critère d'Arrêt:
n==0
⇒ !(n==0)
⇒ n!=0



Renversement Nombre (9)

- Construction du code sur base de l'Invariant (cont.)
  - construction de la *ZONE 2*
    - ✓ construire le Corps de Boucle
      (1) l'Invariant est vrai avant la première instruction de la boucle
      (2) le gardien de boucle est vrai avant la première instruction de la boucle
      (3) dériver les instructions du Corps de Boucle
        (a) mettre à jour `r`
        (b) mettre à jour `n`

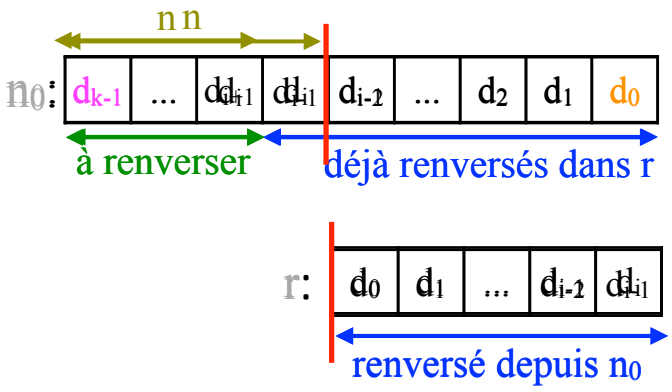
```
#include <stdio.h>

int main(){
  //ZONE 1

  while(n!=0){
    r = 10*r + n%10;
    //???
  }//fin while - n
  //à suivre
}//fin programme
```



Renversement Nombre (10)

- Construction du code sur base de l'Invariant (cont.)
  - construction de la *ZONE 2* (cont.)
    - ✓ construire le corps de la boucle
      (1) l'Invariant est vrai après la dernière instruction de la boucle
      (2) dériver les instructions du corps de la boucle
        (a) mettre à jour `r`
        (b) mettre à jour `n`

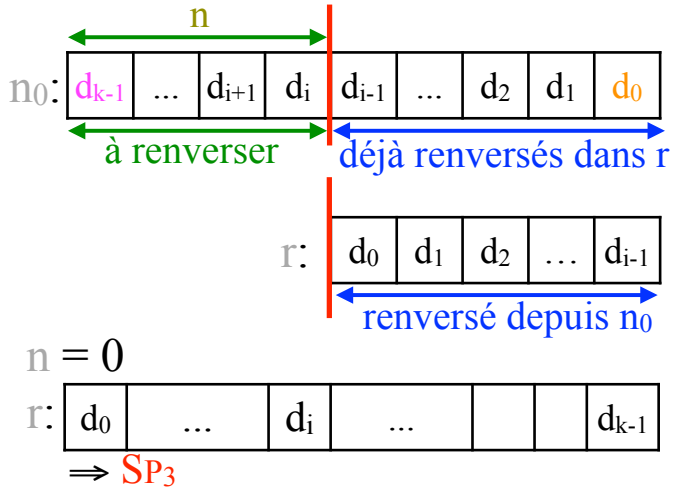
```
#include <stdio.h>

int main(){
  //ZONE 1

  while(n!=0){
    r = 10*r + n%10;
    n = n/10;
  }//fin while - n
  //à suivre
}//fin programme
```



Renversement Nombre (11)

- Construction du code sur base de l'Invariant (cont.)
  - construction de la *ZONE 3*
    - ✓ construire le code après la boucle
      (1) l'Invariant est vrai
      (2) le Critère d'Arrêt est atteint
      (3) dériver les instructions pour atteindre l'objectif final (Output)

```c
#include <stdio.h>

int main(){
  //ZONE 1

  while(n!=0){
    //ZONE 2
  }//fin while - n
  printf("%u\n", r);
}//fin programme
```



Renversement Nombre (12)

- Code complet

```c
#include <stdio.h>

int main(){
  unsigned int r=0;
  unsigned int n;

  scanf("%u", &n);

  while(n!=0){
    r = 10*r + n%10;
    n = n/10;
  }//fin while - n

  printf("le nombre retourné: %u\n", r);
}//fin programme
```

ZONE 1

ZONE 2

ZONE 3

Exercices

- Construire un Invariant Graphique pour
  - SP_2 de l'impression de chiffres
    - ✓ cfr. Slide 21
  - SP_3 de l'impression de chiffres
    - ✓ cfr. Slide 21
  - SP_3 nombres parfaits (version 1)
    - ✓ cfr. Slide 33

Agenda

- Chapitre 3: Méthodologie
  - Schéma Méthodologique
  - Définition du Problème
  - Analyse du Problème
  - Invariant de Boucle
  - Fonction de Terminaison
    - ✓ Principe
    - ✓ Construction
    - ✓ Exemple

Principe

- L'Invariant permet de raisonner sur une boucle
- En outre, il permet de déterminer que la boucle est correcte
  - *à condition* que la boucle se termine
- Comment déterminer, formellement, qu'une boucle se termine?
  - **Fonction de Terminaison**

Principe (2)

- Fonction de Terminaison?
  - *fonction entière* portant sur des variables du programme
  - doit avoir une *valeur > 0* avant toute exécution du corps de la boucle
  - *décroît strictement* à chaque exécution du corps de la boucle
- Plus formellement, f une Fonction de Terminaison:
  - f: {valeurs des variables} $\rightarrow \mathbb{Z}$ t.q.
    - ✓ Invariant et gardien $\Rightarrow f > 0$
    - ✓ $f = f_0$ et Invariant et Gardien
      - itération
    - ✓ $f < f_0$
- Conséquence?
  - on est sûr que, partant de tout entier positif, après un nombre fini d'itérations, on doit sortir de la boucle!

Principe (3)

- Attention, une Fonction de Terminaison
  - n'est pas le Gardien de Boucle
  - n'est pas le Critère d'Arrêt
  - n'est pas l'Invariant de Boucle
  - n'a pas forcément une valeur négative ou nulle lorsque la boucle se termine
    - ce n'est pas demandé par les propriétés que doit respecter une Fonction de Terminaison

Construction

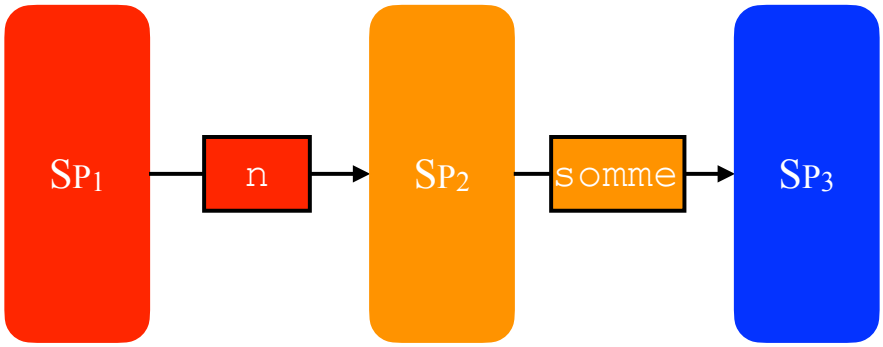
- Comment déterminer une Fonction de Terminaison?
  - **technique 1**: raisonnement sur base de l'Invariant Graphique
    - déterminer, graphiquement, la taille de la zone "encore à ..."
    - probablement la technique la plus simple
      - simple manipulation graphique
    - G. Brieven, S. Liénardy, L. Malcev, B. Donnet. *Graphical Loop Invariant Based Programming*. In Proc. Formal Method Teaching (FMTea). March 2023.
  - **technique 2**: raisonnement sur base du Gardien de Boucle
    - transformer le Gardien en une fonction qui doit être > 0
    - technique plus compliquée car nécessite d'avoir le Gardien
      - globalement, une simple manipulation mathématique

Exemple

- Afficher la somme des *n* premiers entiers positifs
  - *n* donné par l'utilisateur
- Définition du problème
  - Input
    - ✓ `n`, lu au clavier
  - Output
    - ✓ la somme des `n` premiers entiers positifs est affichée sur la sortie standard
  - Caractérisation de l'Input
    - ✓ $n \in \mathbb{N}$
    - ✓ `unsigned int n;`

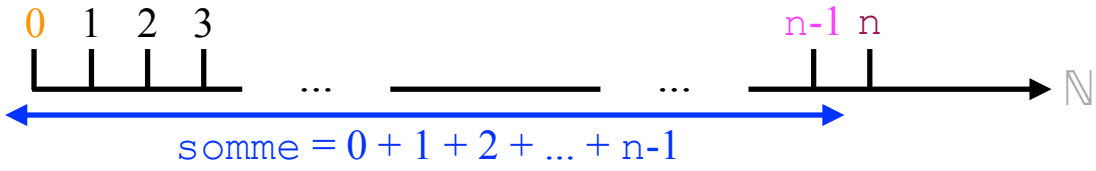
Exemple (2)

- Analyse du Problème
  - SP_1: lecture au clavier
    - ✓ lire la valeur de `n` au clavier
  - SP_2: sommation
    - ✓ calculer la somme de tous les `i` $\in \{0, \ldots, $ **`n`**$-1\}$ dans `somme`
  - SP_3: affichage
    - ✓ affichage de **`somme`**
  - $SP_1 \rightarrow SP_2 \rightarrow SP_3$

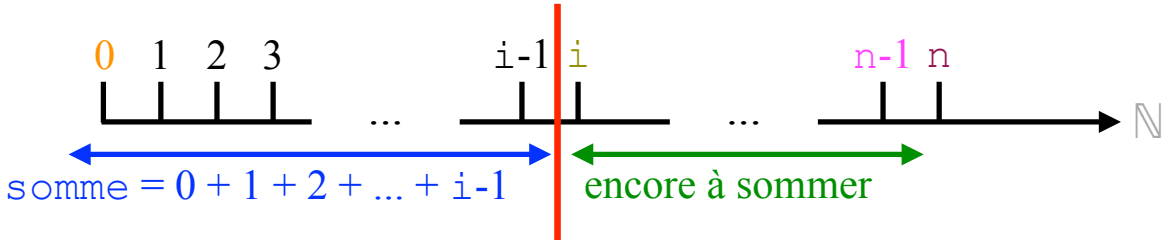


Exemple (3)

- Trouver un Invariant Graphique pour le SP_2
- Représentation graphique de l'Output

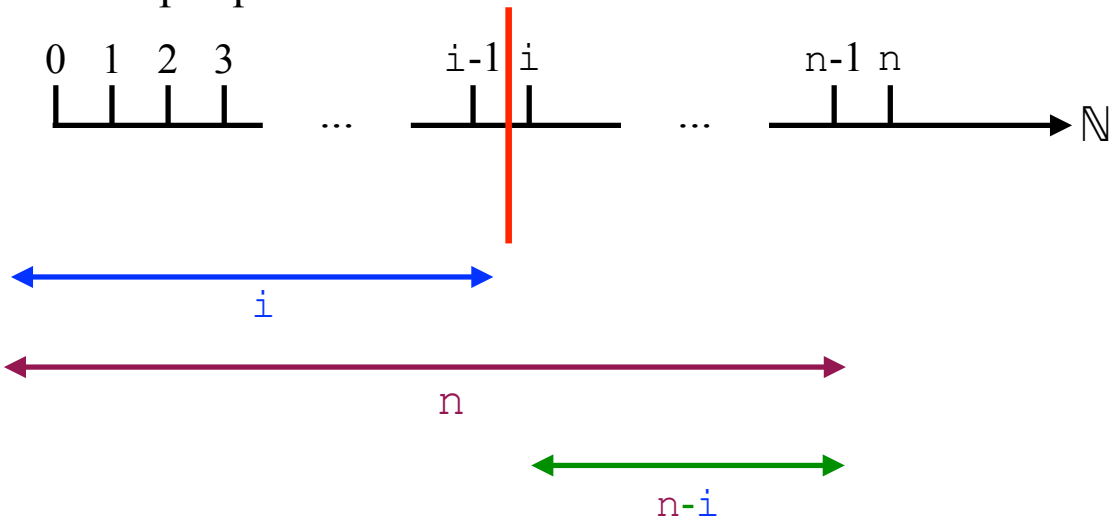


- Éclatement de la PostCondition



Exemple (4)

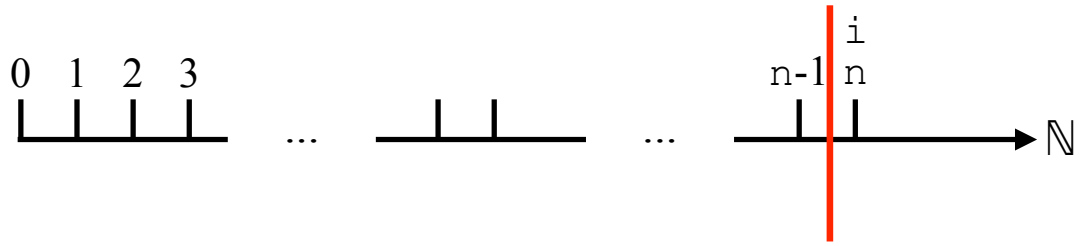
- Déterminer la Fonction de Terminaison
  - technique 1: raisonnement sur base de l'Invariant Graphique



⇒ Fonction de Terminaison: `n-i`

Exemple (5)

- Déterminer la Fonction de Terminaison (cont')
  - technique 2: raisonnement sur base du Gardien de Boucle



⇒ Critère d'Arrêt: `i == n`

⇒ Gardien de Boucle: `i < n`

⇒ `n - i > 0`

⇒ Fonction de Terminaison: `n-i`

Exemple (6)

- Code

```
#include <stdio.h>

int main(){
  unsigned int n, i, somme;

  scanf("%u", &n);

  somme = 0;
  i = 0;
  while(i<n){
    somme += i;
    i++;
  }//fin while - i

  printf("somme: %u\n", somme);
}//fin programme
```

Exercices

- Trouver la fonction de terminaison pour le renversement des chiffres d'un nombre en base 10
- Trouver la fonction de terminaison pour la recherche des nombres parfaits
  - version 1
  - version 2
  - version 3