INFO0947: Compléments de Programmation
Construction de Programme

(Un exercice dont vous êtes le héros)

Benoit Donnet, Simon Liénardy

Avant-propos

Le présent document PDF est « interactif ». Vous pouvez toujours cliquer sur les liens pour passer d'une section à l'autre (ou revenir en arrière). L'idée de ce PDF est bien de ne pas vous donner la solution de l'exercice telle quelle, mais plutôt d'essayer que vous puissiez la construire pas à pas par vous-même, comme on le ferait en séance.

Il n'est donc pas nécessaire de lire tout le document tant que vous arrivez à la solution finale. Si vous pensez que votre code est sensiblement différent de la solution, discutons-en sur le forum eCampus !

2.6.1 Tri par Insertion

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

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

0				4
3	2	1	1	0

On obtiendra successivement les résultats suivants :

— 1re itération

0				4
3	2	1	1	0

— 2e itération

0				4
2	3	1	1	0

— 3e itération

0				4
1	2	3	1	0

— 4e itération

0				4
1	1	2	3	0

— 5e itération

0				4
0	1	1	2	3

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

1. Formaliser le Problème en proposant des notations (voir Sec. 2.6.2)
2. Spécifier le Problème (Sec. 2.6.3
3. Décomposer le problème en Sous-Problèmes (SP) (Sec. 2.6.4)
4. Spécifier chaque SP (Sec. 2.6.5)

5. Construire le programme de chaque SP (Sec. 2.6.6)
6. Rédiger le programme final(Sec. 2.6.9)

2.6.2 Formaliser le Problème en proposant des notations

Pour définir correctement des notations, il faut se poser la question : De quoi parle le Problème ? Quel est le Sujet ?

Déterminez de quoi parle ici le Problème et proposez un (des notations) adéquates.

— Si vous ne voyez pas quelle notation pourrait être intéressante, voyez la Sec. 2.6.2.1

— Si vous voyez la notation mais que vous ne vous souvenez plus de comment faire voyez la Sec. 2.6.2.2

— Si vous avez réussi à définir une notation, voyez la Sec. 2.6.2.3 pour comparer votre réponse avec la nôtre.

2.6.2.1 De quoi parle le Probleme

On commence par relire l'énoncé en mettant en évidence les notions qui apparaissent :

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

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

Il faudrait une notation pour le tri.

— Si vous ne vous souvenez plus de la façon dont il faut procéder, allez à la Sec. 2.6.2.2.

— Si vous vous en souvenez, proposez une notation et rendez-vous à la Sec. 2.6.2.3 pour la correction.

2.6.2.2 Comment définir une notation

Voici la forme que doit prendre une notation :

NomPredicat(Liste de parametres) ≡ Détail du predicat

NomPredicat est le nom à donner du prédicat. Il faut souvent trouver un nom en rapport avec ce que l'on fait. Mathématiquement parlant, cela ne changera rien donc on pourrait appeler le prédicat « Simon » mais il vaut mieux trouver un nom en rapport avec la définition.

Liste de paramètres C'est une liste de symboles qui pourront être utilisés dans la définition du prédicat. Comment cela fonctionne-t-il ? Le prédicat peut être utilisé en écrivant son nom ainsi qu'une valeur particulière de paramètre. On remplace cette valeur dans la définition du prédicat et on regarde si c'est vrai ou faux (Voir TP1).

Détail du prédicat C'est une combinaison de symboles logiques qui fait intervenir, le plus souvent les paramètres.

Exemple :

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

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

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

2.6.2.2.1 Suite de l'exercice

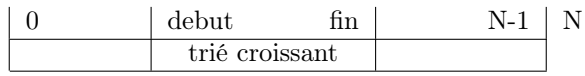
Dans la suite de l'exercice, il faut définir une notation pour le tri. Rendez-vous à la section 2.6.2.3 pour la correction.

2.6.2.2.2 Voir aussi

Comment évaluer un prédicat	TP 1
Signification des opérateurs logiques	Chapitre 1, Slides $6 \rightarrow 9$
Signification des quantificateurs logiques	Chapitre 1, Slides $11 \rightarrow 24$
Signification des quantificateurs numériques	Chapitre 1, Slides $25 \rightarrow 33$

2.6.2.3 Notation que nous utiliserons dans la suite

Nous avons identifié la notion de Tri comme étant importante. Nous introduisons un prédicat qui représente le tri (non strict) d'une portion de tableau. La portion est triée si tous les entiers se suivent selon la relation d'ordre $\leq$ (i.e., tri par ordre croissant). Nous pouvons illustrer ce prédicat de la façon suivante [1] :



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

Nous pouvons traduire ce dessin avec le prédicat suivant :

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

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

2.6.2.3.1 Suite du problème

À l'aide de cette notation, nous allons maintenant spécifier ce problème (2.6.3).

2.6.2.3.2 Voir aussi

Traduire un schéma en assertion logique TP 2

1. Dessiner au préalable la situation que nous souhaitons représenter à l'aide d'un prédicat est toujours une bonne idée (c'est, finalement, le même principe que dessiner un invariant pour, ensuite, le représenter à l'aide d'un prédicat).

2.6.3 Spécifier le Problème

On dispose donc du prédicat $Tri(T, N, \text{debut}, \text{fin})$ qui représente le fait qu'une portion d'un tableau T est trié.

Il faut maintenant définir :

1. la Précondition
2. la Postcondition
3. la signature de la fonction

Si vous voyez comment faire, rendez-vous à la Sec. 2.6.3.1
Sinon, un petit rappel :

La Précondition C'est une condition sur les données. Pour l'établir, il faut se demander quelles doivent être les données du problème. Ici, dans notre exemple, cela parle de tableau. Quelles conditions devons-nous imposer sur ce tableau ?

La Postcondition C'est une condition sur le résultat du problème, si tant est que la précondition est remplie (si tel n'est pas le cas, on pratique la *programmation défensive* et on arrête l'exécution du programme.

La signature est souvent nécessaire lors de l'établissement de la postcondition des fonction. En effet, le résultat retourné par l'invocation de la fonction est représenté par le nom de la fonction dans la postcondition. Le nom de la fonction doit donc être défini avant d'en parler.

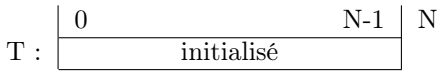
Spécifiez le problème. Rendez-vous à la Sec. 2.6.3.1

2.6.3.1 Spécification du problème

Pour ce faire, on utilise, évidemment la notation que l'on a préalablement définie (**Sinon, on ne se serait pas donné la peine de le faire...**)

On a donc :

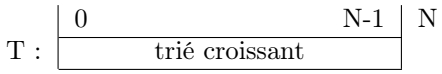
Précondition Il faut demander que le tableau soit initialisé sur toute sa longueur, sinon c'est inutile de le trier. On peut représenter, graphiquement, cette situation particulière de la façon suivante :



Ce qu'on peut traduire, formellemment, de la façon suivante :

$$Pre : T[0..N-1]\ init$$

Postcondition On utilise la notation, le tableau doit être trié sur toute sa longueur. On peut représenter, graphiquement, cette situation particulière de la façon suivante :



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

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

Au final, la Postcondition peut s'exprimer, formellement, de la façon suivante :

$$Post : Tri(T, N, 0, N-1) \wedge T\ perm\ T_0$$

Toute la longueur : debut = 0 et fin est le dernier indice de tableau : N-1. Le module que nous allons écrire est une procédure, elle ne renvoie rien donc `tri_insertion` n'intervient pas dans la Post

Signature

```
1 void tri_insertion(int *T, const unsigned int N);
```

Suite

Passez maintenant à la découpe en Sous-Problème (Sec. 2.6.4).

2.6.4 Décomposer le problème en sous-problème

Il existe de nombreux algorithmes de tri. ici, on est en train de travailler avec le tri par insertion dont le mécanisme est rappelé dans l'énoncé. La découpe en sous-problème est en quelque sorte donnée. Par contre, il convient de relire cet énoncé : (voir Sec. 2.6.1).

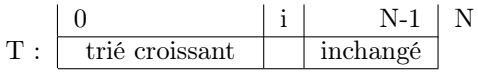
Après avoir fais votre découpe en sous-problème, allez en Sec. 2.6.4.1.

2.6.4.1 Découpe en sous-problèmes

De l'énoncé, on note :

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

L'algorithme revient donc à être dans cette situation :



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

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

Ces étapes forment un premier sous-problème qui consiste à insérer `T[i]` dans `T[0 ... i-1]`.
On pourrait partir sur la découpe suivante :

SP1 Insérer une valeur dans un sous-tableau déjà trié

PP Le problème principal

Pour le SP1, on peut encore être précis : en effet, le décalage est une opération générique et peut être effectué dans un second sous-problème et on obtient la découpe suivante :

SP1 Insérer une valeur dans un sous-tableau déjà trié

SP2 Décaler un sous-tableau d'une cas vers la droite

PP Le problème principal

L'emboitement est donc : $SP2 \subset SP1 \subset PP$

Suite

Il faut maintenant spécifier ces sous-problèmes. Faites-le et rendez-vous à la Sec. 2.6.5.1 pour la correction.

2.6.5 Spécifier chaque sous-problème

Il n'y a rien à rappeler du point de vue théorique ici. La correction se trouve à la suite : Sec. 2.6.5.1

2.6.5.1 Spécification des sous-problèmes

2.6.5.1.1 SP1 – Insérer une valeur dans un tableau trie

On commence par représenter la situation par un schéma :

	0 i-1	i	N-1	N
T :	trié croissant			

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

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

2.6.5.1.2 SP2 – Décaler une portion de tableau d'une case vers la droite

Ici, on va modifier le tableau T, il faudra bien distinguer :
— T_0 qui est le tableau initial (avant l'invocation du sous-problème)
— T qui est le tableau final

```
// Pre : T[begin .. end+1] init
// Post : ∀i, begin < i ≤ end, T[i] = T_0[i − 1]
void shift(int *T, unsigned begin, unsigned end);
```

Dans la Postcondition, $\mathtt{T}[i] = \mathtt{T}_0[i-1]$ représente bien un décalage vers la droite : l'ancienne valeur qui était en $i-1$ dans T_0 est décalée dans la case de droite (i).

2.6.5.1.3 Suite du problème

Passons maintenant à l'implémentation des ces sous-problèmes (2.6.6).

2.6.6 Construire le programme de chaque sous-problème

Petit rappel de la structure du code (cfr. Chapitre 2, Slide 28) :

```
// {Pre}
INIT
// {Inv}
while(B){
   // {Inv ∧ B}
   ITER
   // {Inv}
}
// {Inv ∧ ¬B}
END
// {Post}
```

La première chose à faire, pour chaque sous-problème, est de déterminer un Invariant. La meilleure façon de procéder est de d'abord produire un Invariant Graphique et de le traduire ensuite formellement. Le code sera ensuite construit sur base de l'Invariant.

Pour l'Invariant, n'hésitez pas à vous appuyer sur le GLI : `https://gli.montefiore.ulg.ac.be`

2.6.6.1 Suite de l'exercice

Résolvez les sous-problèmes dans l'ordre de votre choix.

Pour	Voir la Sec. :
L'Invariant du SP1	2.6.6.2.1
L'Invariant du SP2	2.6.6.3.1

2.6.6.2 SP1 – Insérer une valeur dans un tableau trié

2.6.6.2.1 Invariant

Dans la résolution du SP1, nous avons préalablement analysé qu'il fallait :

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

Le point 1 nécessite une boucle, donc, un Invariant.
Réfléchissez d'abord à un Invariant Graphique [2] et allez en Sec. 2.6.6.2.2.

2. Si nécessaire, aidez-vous du GLI (`https://gli.montefiore.ulg.ac.be`)

2.6.6.2.2 Invariant Graphique pour le SP1

On essaie en fait d'arriver à cette situation :

	0	$\texttt{k}_f$-1	$\texttt{k}_f$	i-1	i	N-1	N
T :	$\cdot <$ T[i]		$\leq T[i]$				

On aura donc un **Invariant** de ce type :

	0	k-1	k	i-1	i	N-1	N
T :	$\cdot <$ T[i]		à investiguer				

Quelle est la différence entre les deux dessins ? Pourquoi le premier n'est pas un invariant alors que l'autre en est bien un Invariant ?

$\Rightarrow$ Dans le dessin du dessus, on a représenté une **situtation particulière** où tous les éléments avant `k` sont plus petit que `T[i]` et tous les éléments après sont plus grands ou égaux. c'est un cas particulier !

Dans le second dessin, on a assuré que les éléments avant `k` sont plus petit que `T[i]` mais on a pas encore atteint le point particulier où les éléments dans les indices suivants du tableaux ne représentent plus cette propriété (que l'on note d'ailleurs $\texttt{k}_f$ dans le schéma du dessus).

Remarque C'est un bon Invariant parce que :

1. La structure manipulée est un tableau, il est correctement représenté et nommé
2. Les bornes du problèmes (0 et i) sont représentées
3. Il y a une ligne de division
4. La ligne de division est annotée par une variable : k
5. Ce qu'on a déjà fait est indiqué par le texte $< T[i]$
6. Ce qu'on doit encore faire est mentionné

Ce sera un encore meilleur Invariant dès que le code construit sur sa base sera en lien avec lui.

Suite de l'exercice Traduisez maintenant cet Invariant graphique en Invariant formel et rendez-vous à la Sec. 2.6.6.2.3 pour le résultat.

2.6.6.2.3 Invariant formel pour le SP1

Partant du schéma, on a :

	0 … k-1	k … i-1	i	N-1 … N
T :	$\cdot <$ T[i]	à investiguer		

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

On écrit cela parce que :

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

Suite de l'exercice

Si ce n'est pas encore fait, passez à l'Invariant Graphique et formel du SP2 (Sec. 2.6.6.3.1).
Si c'est déjà fait, passez à l'écriture du code des SP (Sec. 2.6.6.4)

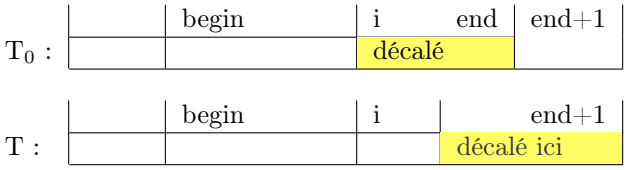
2.6.6.3 SP2 – Décaler une portion de tableau vers la droite

2.6.6.3.1 Invariant

Commencez par un Invariant Graphique. Ici, vu qu'il faut dessiner le tableau de départ et le tableau d'arrivée, on peut représenter 2 tableaux : T et T_0.

solution de l'Invariant Graphique : Sec. 2.6.6.3.2.

2.6.6.3.2 Invariant Graphique pour le SP2



On représente le décalage des éléments depuis T_0 jusque T. Le sous-problème `shift` ne prends que les paramètres `T`, `begin` et `end`.

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

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

Suite de l'exercice Traduisez maintenant cet invariant graphique ne invariant formel et rendez-vous à la Sec. 2.6.6.3.3 pour le résultat.

2.6.6.3.3 Invariant formel pour le SP2

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

On écrit cela parce que :

1. On représente la position de `begin`, `i` et `end` ;
2. On rappelle qu'on conserve la position des bornes
3. On décrit le décalage vers la droite. Ici, j'ai représenté la quantification dans le tableau `T`, par rapport à $\texttt{T}_0$ donc les j vont de $i+1$ jusque $end+1$. J'aurais pu faire l'inverse et écrire :

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

   qui revient exactement au même !

Suite de l'exercice

Si ce n'est pas encore fait, passez à l'Invariant Graphique et formel du SP1 (Sec. 2.6.6.2.1).
Si c'est déjà fait, passez à l'écriture du code des SP (Sec. 2.6.6.4)

2.6.6.4 Rédiger le programme de chaque sous-problème

Il faut maintenant, sur bases des invariants, rédiger les programmes finaux pour les sous-problèmes. Il vaut mieux commencer par le SP2. Pourquoi ? Parce que suivant la découpe en SPs, nous en aurons besoin lors de la rédaction du SP1.

2.6.6.4.1 suite de l'exercice

Pour chaque SP, il faut préciser (cfr. Chapitre 2, Slides 25 $\rightarrow$ 37) :

INIT De la Précondition, il faut arriver dans une situation où l'Invariant est vrai

B Il faut trouver d'abord la condition d'arrêt $\neg B$ et en déduire le gardien

ITER Il faut faire progresser le programme puis restaurer l'Invariant.

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

Fonction t Il faut donner une fonction de terminaison pour montrer que la boucle se termine (cfr. INFO0946, Chapitre 3, Slides 97 $\rightarrow$ 107).

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

Construction	Voir la Sec :
Du SP1	2.6.7
Du SP2	2.6.8

2.6.7 Rédiger le programme du SP1

2.6.7.1 INIT

Sur base de l'invariant :

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

Il faut rendre ceci vrai. Clairement, cela consiste à trouver une valeur pour k. Cette valeur de k doit vérifier la dernière partie de l'invariant : $\forall j, 0 \leq j < k, T[j] < T[i]$. Rendons trivialement vrai cette expression en choisissant un k qui fasse que l'ensemble des j possibles soit vide. Seul $k = 0$ convient.

```
// Pre
int k  0;
// Inv
```

2.6.7.2 B

La position finale a été représentée plus haut. En fait, il faut s'arrêter de faire grandir la zone d'éléments inférieurs à `T[i]` si l'élément suivant est plus grand ou égal.

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

Ce qui donne

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

2.6.7.3 ITER

```
while(k < i && T[k] < T[i])
// {Inv ∧ B} : (∀j, 0 ≤ j < k, T[j] < T[i]) ∧ T[k] < T[i]
// ∀j, 0 ≤ j ≤ k, T[j] < T[i]
++k;
//{Inv}
```

De $\{Inv \wedge B\}$, on peut conclure $\forall j, 0 \leq j \leq k, T[j] < T[i]$ (Voir le $<$ qui s'est changé en $\leq$). On doit maintenant rétablir l'Invariant en incrémentant k.

2.6.7.4 END

On a assuré, avec ce morceau de code que tous les éléments avant T[k] sont inférieurs à T[i] et tous ceux après sont supérieurs. Il reste encore à décaler le tableau et à insérer l'élément au bon endroit

```
//{Inv ∧ ¬B}
//∀j, 0 ≤ j < k, T[j] < T[i] ∧ ∀j, k ≤ j < i, T[j] < T[i]
// Sauvegarde de T[i]
int save = T[i];
// save = T0[i]
// Décalage du tableau
//Pre_shift T[k, i] init
shift(T, k, i-1);
// Post_shift : ∀j, k < j ≤ i − 1, tab[j] = tab0[j − 1] ⇒ ∀j, k + 1 ≤ j < i + 1, T[j] ≥ save
//De plus, on a toujours ∀j, 0 ≤ j < k, T[j] < save
T[k] = save
// T perm T0 ∧ Tri(T, N, 0, i)
```

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

Remarque Avant d'appeler `shift`, il faut que sa Précondition soit vraie. Après l'appel, la Postcondition de shift est vraie. C'est une **propriété fondamentale** de l'approche constructive.

2.6.7.5 Fonction de Terminaison

Il faut trouver une combinaison de variables du programme qui diminue à chaque itération. En gros, c'est la distance entre k et i.

$$t = i - k$$

Sera > 0 qand le gardien est vrai et a une valeur qui décroit strictement à chaque itération.

2.6.7.6 SP1 final

```
void insertion(int *T, unsigned int i){
   assert(T); // Progra def
   // Pre
   int k  0;
   // Inv
   while(k < i && T[k] < T[i])
   // {Inv ∧ B} : (∀j, 0 ≤ j < k, T[j] < T[i]) ∧ T[k] < T[i]
   // ∀j, 0 ≤ j ≤ k, T[j] < T[i]
      ++k;
      //{Inv}
   }
   //{Inv ∧ ¬B}
   //∀j, 0 ≤ j < k, T[j] < T[i] ∧ ∀j, k ≤ j < i, T[j] < T[i]
   // Sauvegarde de T[i]
   int save = T[i];
   // save = T0[i]
   // Décalage du tableau
   //Pre_shift T[k, i] init
   shift(T, k, i-1);
   // Post_shift : ∀j, k < j ≤ i − 1, tab[j] = tab0[j − 1] ⇒ ∀j, k + 1 ≤ j < i + 1, T[j] ≥ save
   //De plus, on a toujours ∀j, 0 ≤ j < k, T[j] < save
   T[k] = save
   // TpermT0 ∧ Tri(T, N, 0, i)
   return
   // Post
```

2.6.7.7 Suite de l'exercice

Passez soit au SP2 (Sec. 2.6.8)
Ou alors passez au programme final (Sec. 2.6.9))

2.6.8 Rédiger le programme du SP2

2.6.8.1 INIT

Sur base de l'Invariant :

$$Inv = \texttt{begin} \leq \texttt{i} \leq \texttt{end} + 1 \wedge \texttt{begin} = \texttt{begin}_0 \wedge \texttt{end} = \texttt{end}_0 \wedge \forall j, i < j \leq end + 1, \texttt{T}[j] = \texttt{T}_0[j-1]$$

Il faut rendre la troisième partie ($\forall j, i < j \leq end + 1, \texttt{T}[j] = \texttt{T}_0[j-1]$) vraie. On peut le faire en sélectionnant une valeur particulière de `i`. On pourrait faire en sorte que l'intervalle sur les j soit vide (cela rend le $\forall$ trivialement vrai [3]). C'est possible en initialisant `i` à `end` + 1.

```
// Pre
int i = end + 1;
// Inv
```

2.6.8.2 B

Vu ce que l'on cherche à faire, il faut s'arrêter lorsque `i` = `begin`

$$\neg B \equiv i = begin$$

Donc

$$B \equiv i > begin$$

Qui représente bien la position relative de `i` et `begin`.

2.6.8.3 ITER

```
while(i > begin){
 //{Inv ∧ ¬B}
 //begin < i ≤ end + 1 ∧ begin = begin0 ∧ end = end0 ∧ ∀j, i < j ≤ end + 1, T[j] = T0[j − 1]
 T[i] = T[i - 1];
 //begin < i ≤ end + 1 ∧ begin = begin0 ∧ end = end0 ∧ ∀j, i ≤ j ≤ end + 1, T[j] = T0[j − 1]
 --i;
 //begin ≤ i ≤ end + 1 ∧ begin = begin0 ∧ end = end0 ∧ ∀j, i < j ≤ end + 1, T[j] = T0[j − 1] = {Inv}
 }
```

On fait progresser le problème en décalant la case T[i-1] dans la case T[i] et on restaure l'Invariant.

2.6.8.4 END

On a que $Inv \wedge \neg B = Post$:

$$i = begin \wedge \forall j, i < j \leq end + 1, \texttt{T}[j] = \texttt{T}_0[j-1]$$

$$\forall j, begin < j \leq end + 1, \texttt{T}[j] = \texttt{T}_0[j-1] = Post$$

3. Pour rappel (Chap. 1, Slides 12 → 16), le quantificateur universel se présente comme suit : $\forall(l) \cdot (P \Rightarrow Q)$. Si P est faux (e.g., ensemble vide), l'expression $P \Rightarrow Q$ est vraie (cfr. Chapitre 1, Slide 6).

2.6.8.5 Fonction de Terminaison

Il faut trouver une combinaison de variable qui diminue au cours des itérations. La distance entre `i` et `begin` fait l'affaire :

$$t = i - begin$$

2.6.8.6 SP2 final

```
void shift(int *tab, int begin, int end){
   assert(tab);
   // Pre
   int i = end + 1;
   // Inv
   while(i > begin){
      //{Inv ∧ ¬B}
      //begin < i ≤ end + 1 ∧ begin = begin_0 ∧ end = end_0 ∧ ∀j, i < j ≤ end + 1, T[j] = T_0[j − 1]
      T[i] = T[i - 1];
      //begin < i ≤ end + 1 ∧ begin = begin_0 ∧ end = end_0 ∧ ∀j, i ≤ j ≤ end + 1, T[j] = T_0[j − 1]
      --i;
      //begin ≤ i ≤ end + 1 ∧ begin = begin_0 ∧ end = end_0 ∧ ∀j, i < j ≤ end + 1, T[j] = taTb_0[j − 1] = {Inv}
   }
   //{Inv ∧ ¬B} = {Post}
   return;
}
```

2.6.8.7 Suite de l'exercice

Passez soit au SP1 (Sec. 2.6.7)
Ou alors passez au programme final (Sec. 2.6.9))

2.6.9 Rédiger le programme final

2.6.9.1 Suite de l'exercice

Il faut commencer par l'Invariant Graphique (Sec. 2.6.9.2).
Ensuite le traduire en Invariant formel (Sec. 2.6.9.3)
Ensuite, il faut construire le problème principal (Sec. 2.6.9.4).

2.6.9.2 Invariant Graphique

On part sur un Invariant qui rappelle ce que mentionnait l'énoncé :

	0	i N-1	N
T :	Trié croissant	à investiguer	

Rendez-vous à la Sec. 2.6.9.3 pour la version formelle

2.6.9.3 Invariant formel

$$
\begin{aligned}
Inv = {} & 0 \leq i \leq N && (1)\\
& \wedge Trie(T, N, 0, i-1) && (2)\\
& \wedge TpermT_0 && (3)
\end{aligned}
$$

1. Position relative de i par rapport aux bornes du tableau
2. Portion du tableau déjà triée
3. On n'ajoute pas des éléments dans T donc T est une permutation du tableau initial.

2.6.9.4 Code du Programme Principal

2.6.9.5 INIT

Rien est encore tiré dans T. Dès lors, le prédicat $Trie(T, N, 0, -1)$ est trivialement vrai [4].

```
// Pre
int i = 0;
// Inv
```

2.6.9.6 B

On compare l'Invariant et la Postcondition :

$Inv = 0 \leq i \leq N \wedge Trie(T, N, 0, i-1) \wedge TpermT_0$

$Post = Tri(T, N, 0, N-1) \wedge T\ perm\ T_0$

On voit que dès que $i = N$, on a atteint la Postcondition.

$\neg B \equiv i = N$

$B \equiv i < N$

2.6.9.7 ITER

```
while(i < N){
    //{Inv ∧ B}
    // 0 ≤ i < N ∧ Trie(T, N, 0, i − 1) ∧ TpermT0
    // Pre_insertion
    insertion(T, i);
    // Post_insertion : Trie(T, N, 0, i) ∧ T perm T0
    ++i;
    //0 ≤ i ≤ N ∧ Trie(T, N, 0, i − 1) ∧ T perm T0
    //{Inv}
}
```

2.6.9.8 END

$\{Inv \wedge \neg B\} = \{Post\}$
Il n'y a donc rien à faire.

2.6.9.9 Fonction de Terminaison

$t = N - i$

Est positive si B et décroit d'une itération à l'autre.

4. "Un bibliothèque vide est, par définition, triée". (© B. Donnet)

2.6.9.10 code complet final

```
void tri_insertion(T, N){
   assert(T);
   // Pre
   int i = 0;
   // Inv
   while(i < N){
      //{Inv ∧ B}
      //0 ≤ i < N ∧ Trie(T, N, 0, i − 1) ∧ TpermT_0
      // Pre_insertion
      insertion(T, i);
      // Post_insertion : Trie(T, N, 0, i)
      ++i;
      // 0 ≤ i ≤ N ∧ Trie(T, N, 0, i − 1) ∧ TpermT_0
      //{Inv}
   }
   //{Inv ∧ ¬B} = {Post}
   return;
}
```