Conception Formelle

TD-TP : Un peu de théorie et de pratique.

Thomas Morin Dimitri Periphanos

2021-2022

Ce travail est à réaliser en binôme. La date de rendu est fixée au 1^{er} mai 2022.

Ce sujet (à part le premier exercice) est librement inspiré de l'algorithme Brute Force pour le tsp vu dans le cours de TAP de L3 [1]. Rassurez-vous, on va y aller pas à pas et ne démontrer que le plus simple.

Les exercices sont par difficulté croissante. Les deux premiers demandent uniquement d'appliquer le calcul de WP et de trouver des invariants de boucle. Les réaliser parfaitement devrait valoir autour de 9. Le troisième demande de produire un code correct et de le démontrer, et de comprendre un prédicat. Avec les trois premiers exercices parfaits, on devrait arriver vers 15. Le quatrième est plus technique, il demande de formaliser un contrat informel et déterminer les bons invariants de boucle est moins immédiat, mais devrait être suffisamment guidé pour rester faisable. Le cinquième est plus exploratoire, et devrait donc être plus vu comme un bonus (la première question devrait cependant rester accessible).

Structure du devoir et rendu Le devoir est constitué du présent document, ainsi que de fichiers de code à compléter en parallèle des questions posées. Le code fourni est découpé ainsi : un fichier .c et .h pour chaque fonction, et un fichier `formalism.h` pour les prédicats. Ces fichiers sont à remplir au fur et à mesure. `formalism.h` contient uniquement deux prédicat à remplir (à l'exercice 3), le reste vous est donné. Il contient des lemmes qu'on admettra.

Pour répondre aux questions, vous avez deux choix :

— Soit compléter le présent document .tex en plaçant vos réponses dans les blocs solutionorbox prévues à cet effet. Pour cela, référez-vous aux macros définies dans les sujets de TD.

— Soit produire un document pdf par d'autres moyens (autre logiciel, scan), tant que c'est lisible, ça me conviendra.

Vous devrez rendre (sur la page moodle du cours) :

— Le présent document de réponse (ou votre version).

— Le dossier `code` complété. Dans les fichiers qui le composent, vous pourrez si besoin rajouter des commentaires expliquant comment prouver certaines spécifications si ce n'est pas immédiat, ou expliquer ce qui vous bloque.

Rappels de logique : On rappelle que :

— $p \Rightarrow q \equiv \neg p \vee q$

— $(p \wedge q) \Rightarrow r \equiv p \Rightarrow q \Rightarrow r$.

Vous aurez à manipuler des expressions comprenant des $\forall$. Pour ces expressions-là, lorsque vous appliquez un pas de WLP, grossièrement il vous faut découper votre formule

1. `https://dept-info.labri.fr/~gavoille/UE-TAP/`

entre partie modifiée et partie non-modifiée, et appliquer le calcul. Plus concrètement ici, ce que vous devriez avoir, c'est quelque chose du genre :

$$\begin{aligned}&\mathrm{WLP}(t[i] = x, \forall j; 0 \leq j \leq i ==> \varphi(t[j]))\\ \equiv&\mathrm{WLP}(t[i] = x, (\forall j; 0 \leq j < i ==> \varphi(t[j])) \wedge \varphi(t[i]))\\ \equiv&(\forall j; 0 \leq j < i ==> \varphi(t[j])) \wedge \varphi(x)\end{aligned}$$

La règle la plus générale d'où vient cela est la suivante : pour φ et ψ des formules quelconques :

$$\forall j; \varphi(j) \equiv \forall j; \psi(j) ==> \varphi(j) \wedge \forall j; \neg\psi(j) ==> \varphi(j);$$

Un autre point, c'est «soyez paresseux» : si vous arrivez à un moment sur un calcul équivalent à $\bot \Rightarrow \mathrm{WLP}(i - j, \phi)$, il est inutile de calculer $\mathrm{WLP}(i - j, \phi)$ pour déterminer que l'implication est vraie.

Vous devrez aussi prouver des prédicats de la forme `Toto{Pre,Post}(...)`. Pour prouver ce genre de prédicats, rappelez-vous, que Pre correspond au début du programme : il ne sera donc pas modifié par le calcul de WP. Post, au contraire, correspond à la fin du programme et donc au label Here : lors du calcul de WP, les valeurs correspondant à la position courante sont modifiées (donc vous gardez le label Here, où vous mettez la ligne de la position courante). Ça correspond simplement au calcul normal de WP où on laisse la formule abstraite : cette formule contient un label implicite qui est bel et bien Here. Une bonne manière de voir cela, est que, lorsque que vous remplacez le nom du prédicat par sa définition, vous remplacerez les `at(x,Pre)` par old(x), et les `at(x,Post)` par x, et effectuez le calcul normalement.

On attendra que vos calculs de WP soient suffisamment détaillés, mais vous pouvez sauter quelques étapes si elles sont faciles (comme par exemples, appliquer plusieurs substitutions d'un coup). Évidemment, tant que cela est correct.

Comme dit souvent, pour les justification de vérité de formule, on n'attendra pas de preuves formelles, mais des justifications convaincantes (i.e., qui n'oublient pas de cas, mais on restera tolérant sur la forme). En gros, quand vous aurez une implication, une possibilité sera de dire un truc du genre «l'implication est vraie car telle et telle partie de la partie gauche impliquent bien la partie droite». Un «ben oui» (ou plutôt un remplacement par $\top$) sera admissible pour des propriétés du genre $\bot \Rightarrow \varphi$, $\varphi \Rightarrow \top$ ou $a < b \Rightarrow a \leq b$. Cependant, pour faire cela de manière convaincante, vous auriez intérêt à simplifier vos formules avant.

Dans le présent sujet, on donne une version normalisée du code pour vous faciliter preuves (if then else développés, un seul return, que des while). Évidemment, vous pouvez coder autrement, mais il sera plus aisé de faire ces restrictions sur papier.

Un mot sur les preuves : À partir de l'exercice 3, certaines des preuves commencent à être difficiles pour les solveurs, aussi faites bien les trois points suivants :

- Vérifiez que le solveur Z3 est activé (il parvient à démontré des propriétés où alt-ergo échoue).
- Si certaines propriétés ne sont pas prouvées, retentez la preuve : quand les solveurs tentent trop de preuves en même temps, il arrive que certaines timeout pour de mauvaises raisons. Les relancer peut régler le problème.
- Si cela ne marche toujours pas (et que vous avez confiance en la propriété), cliquez sur le nom du but, puis sur la tactique «filter». Cela peut parfois régler le problème.

Si rien de tout cela ne marche, vous avez probablement oublié de spécifier certaines hypothèses (ou votre propriété est fausse), donc reprenez votre stylo.

Exercice 1 : Swap par xor

On considère une variante du swap sans variable auxiliaire, en utilisant l'opération xor (notée ^). On va pour l'occasion, augmenter les opérations arithmétiques que l'on s'autorise dans notre langage jouet C1 avec l'opération xor, avec la même sémantique qu'en C. En particulier, on rappelle (et c'est la seule propriété dont on aura besoin) que pour tous entiers n, m, $(n \hat{} m) \hat{} n = m$, et que cette opération est commutative (et associative).

À noter que cette version du swap n'est en réalité pas vraiment plus efficace que celle avec mémoire auxiliaire. On va de plus voir qu'elle dispose d'un léger défaut supplémentaire.

```
1  /*@ ensures *a == \old(*b);
2      ensures *b == \old(*a);
3  */
4  void swap(int *a, int *b)
5  {
6      *a = (*a) ^ (*b);
7      *b = (*a) ^ (*b);
8      *a = (*a) ^ (*b);
9      return;
10 }
```

(a) Si ce n'est pas déjà fait et que vous rendrez le présent .tex, mettez vos noms dans la balise author situé en haut de ce document (celle où il y a écrit un message assez explicite). (0 points)

(b) Calculez WP(swap, ψ) pour ψ la post-condition fournie, et déduisez-en un triplet de Hoare valide.

Réponse:

On a :

$\text{WP}(\texttt{swap}, \psi) = \text{WP}(6 - 8, \psi)$

$\equiv \text{WP}(\texttt{swap}, \psi) = \text{WP}(6 - 7, \psi[*a \leftarrow *a \hat{} * b])$

$\equiv \text{WP}(\texttt{swap}, \psi) = \text{WP}(6, \psi[*a \leftarrow *a \hat{} * b][*b \leftarrow *a \hat{} * b])$

$\equiv \text{WP}(\texttt{swap}, \psi) = \psi[*a \leftarrow *a \hat{} * b][*b \leftarrow *a \hat{} * b][*a \leftarrow *a \hat{} * b]$

Et en remplacant ψ par sa valeur, c'est-à-dire $*a == \text{old}(*b) \wedge *b == \text{old}(*a)$, on obtient :

$\text{WP}(\texttt{swap}, \psi) = (*a == \text{old}(*b) \wedge *b == \text{old}(*a))[*a \leftarrow *a \hat{} * b][*b \leftarrow *a \hat{} * b][*a \leftarrow *a \hat{} * b]$

$\equiv \text{WP}(\texttt{swap}, \psi) = ((*a \hat{} * b) == \text{old}(*b) \wedge *b == \text{old}(*a))[*b \leftarrow *a \hat{} * b][*a \leftarrow *a \hat{} * b]$

$\equiv \text{WP}(\texttt{swap}, \psi) = ((*a \hat{} (*a \hat{} * b)) == \text{old}(*b) \wedge (*a \hat{} * b) == \text{old}(*a))[*a \leftarrow *a \hat{} * b]$

$\equiv \text{WP}(\texttt{swap}, \psi) = ((*a \hat{} * b) \hat{} ((*a \hat{} * b) \hat{} * b)) == \text{old}(*b) \wedge ((*a \hat{} * b) \hat{} * b) == \text{old}(*a)$

Et en appliquant la règle de simplification du xor, on obtient :

$\text{WP}(\texttt{swap}, \psi) = ((*a \hat{} * b) \hat{} (*a) == \text{old}(*b) \wedge (*a) == \text{old}(*a)$

$\equiv \text{WP}(\texttt{swap}, \psi) = (*b) == \text{old}(*b) \wedge (*a) == \text{old}(*a)$

Puis en retirant les old, on obtient :
$\text{WP}(\texttt{swap}, \psi) = (*b) == (*b) \wedge (*a) == (*a)$
$\equiv \text{WP}(\texttt{swap}, \psi) = \top$ (ben c'est égal)
Et on en déduit ainsi que un triplet de hoare valide pour cette fonction est :
$\{ \top \}$ swap $\{ *a == \text{old}(*b) \wedge *b == \text{old}(*a) \}$

(c) Évidemment, on a ici une correction partielle, dans le sens où on n'a pas tenu compte des comportements indéterminés. Quelle information manque-t'il pour assurer qu'il n'y aura pas d'erreur à l'exécution ?

Réponse:
Bien entendu, il faut que les deux pointeurs données soit valides et lecture et en écriture.

(d) Quel autre problème voyez-vous à notre preuve ? Quelle hypothèse cachée faut-il rajouter pour qu'elle soit correcte ?

Réponse:
Il faut s'assurer que les pointeurs a et b soit séparés. En effet, si a et b ne sont pas différents, alors lorsque l'on change l'un, on change l'autre, et donc on a plus l'assurance d'avoir la proprieté de simplification du xor.

(e) Compléter le fichier `swap-xor.h` de manière à ce que Frama-C puisse démontrer la post-condition fournie, en ajoutant les assertions RTE. Il est bien évidemment possible (et encouragé) de répondre aux questions précédentes grâce à celle-ci.

Exercice 2 : Reverse

On veut maintenant une fonction qui va inverser une portion consécutive d'un tableau. Le but de cet exercice est de formaliser le contrat et de le prouver. On considère le code de la fonction, ainsi que la précondition que l'on souhaite démontrer.

```
1  /*@
2  ensures Phi: \forall integer k;
3               i <= k <= j ==> t[k] == \old(t[j-k+i]);
4  */
5  void reverse(int *t, unsigned int i, unsigned int j,
6   unsigned int n)
7  {
8      while (i < j)
9      {
10         int aux = t[i];
11         t[i] = t[j];
12         t[j] = aux;
13         i++;
14         j--;
15     }
16     return;
17 }
```

(a) Quelle condition doit-être vraie à la sortie de la boucle (question sur 0 points, c'est quand même trivial) ?

Réponse:
C'est Φ : il n'y a rien après, donc il faut qu'à la sortie de la boucle la post-condition soit vraie.

(b) Calculez $\text{WLP}(10-14, I)$ pour un invariant I quelconque.

Réponse:
On a :
$\text{WLP}(10-14, I) = \text{WLP}(10-13, I[j \leftarrow j-1])$
$\equiv \text{WLP}(10-14, I) = \text{WLP}(10-12, I[j \leftarrow j-1][i \leftarrow i+1])$
$\equiv \text{WLP}(10-14, I) = \text{WLP}(10-11, I[j \leftarrow j-1][i \leftarrow i+1][t[j] \leftarrow aux])$
$\equiv \text{WLP}(10-14, I) = \text{WLP}(10, I[j \leftarrow j-1][i \leftarrow i+1][t[j] \leftarrow aux][t[i] \leftarrow t[j]])$
$\equiv \text{WLP}(10-14, I) = I[j \leftarrow j-1][i \leftarrow i+1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]$

(c) Proposez des invariants de boucle pour votre boucle. Vous devriez en avoir qui détermine un ordre sur les variables i, j et les comparent avec leurs valeurs initiales (un seul peut suffire, mais vous pouvez bien sûr le découper). Vous devriez également avoir des invariants qui parlent de la partie du tableau déjà modifiée, et d'autres qui parlent de la manières non encore modifier. Au total, 3 invariants peuvent suffire, mais ça dépend de comment vous les écrivez (en compressant on peut faire 2, et on peut les découper en plus). Vous leur donnerez des noms (par exemple I1, I2, etc, ou des noms plus explicites). On appellera I leur conjonction. Justifiez que $\neg i < j \wedge I \Rightarrow \text{WLP}(16, \varphi)$.
Pour chaque Ii, précisez $\text{WLP}(10-14, Ii)$ (utilisez le calcul de la question b), et justifiez que $i < j \wedge I \Rightarrow \text{WLP}(10-14, Ii)$.

Réponse:
Les invariants choisis sont au nombre de 5 :
$I_1 = 0 \leq \backslash at(i, Pre) \leq i <= j+1 \leq \backslash at(j, Pre)+1 \leq n$. Ce premier invariant donne un encadrement de la valeur de i et de j.
$I_2 = \forall$ integer $k; i \leq k \leq j \Rightarrow t[k] == \backslash at(t[k], Pre)$. Cet invariant donne une indication sur la partie du tableau non parcourue (la portion non parcourue n'a pas été modifiée).
$I_3 = i == \backslash at(j, Pre) - j + \backslash at(i, Pre)$. Cet invariant exprime le fait que l'écart entre i et la valeur de départ de i est égal à à l'écart entre j et la valeur de départ de j.
$I_4 = \forall$ integer $k; (\backslash at(i, Pre) \leq k < i) \Rightarrow t[k] == \backslash at(t[j-k+i], Pre)$. Cet invariant exprime la valeur de la partie traitée du tableau entre la valeur de départ de i et i.
$I_5 = \forall$ integer $k; j < k \backslash at(j, Pre) \Rightarrow t[k] == \backslash at(t[j-k+i], Pre)$. Cet invariant exprime la valeur de la partie traitée du tableau entre la valeur de départ de j et j.

On va maintenant poruver pour chaque I_i la chose demandée.

On a :

— $\text{WLP}(10-14, I_1) = 0 \leq \backslash at(i, Pre) \leq i <= j+1 \leq \backslash at(j, Pre)+1 \leq n[j \leftarrow j-1][i \leftarrow i+1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]$
$\equiv \text{WLP}(10-14, I_1) = 0 \leq \backslash at(i, Pre) \leq i+1 <= j \leq \backslash at(j, Pre)+1 \leq n.$

Ainsi, $i < j \wedge I \Rightarrow \text{WLP}(10-14, I_1)$ est vrai car si $i < j$ alors $i+1 <= j$; la valeur de $\backslash at(i, Pre)$ ne change pas, donc son rapport à 0 ne change pas, et comme on a I_1 dans I, on a $\backslash at(i, Pre) \leq i$ et donc $\backslash at(i, Pre) \leq i+1$. De manière analogue, on montre que $j \leq \backslash at(j, Pre)+1 \leq n$ est vrai.

— $\text{WLP}(10-14, I_2) = \forall \text{ integer } k; i \leq k \leq j \Rightarrow t[k] == \backslash at(t[k], Pre)[j \leftarrow j-1][i \leftarrow i+1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]$
$\equiv \text{WLP}(10-14, I_2) = \forall \text{ integer } k; i+1 \leq k \leq j-1 \Rightarrow t[k] == \backslash at(t[k], Pre).$

Ainsi, $i < j \wedge I \Rightarrow \text{WLP}(10-14, I_2)$ est vrai car on a I_2 dans I, et donc si $\forall$ integer $k; i \leq k \leq j \Rightarrow t[k] == \backslash at(t[k], Pre)$ est vrai, on a $\forall$ integer $k; i+1 \leq k \leq j-1 \Rightarrow t[k] == \backslash at(t[k], Pre)$ (Après un tour de boucle, le k est dans une plage plus petite et incluse dans la plage avant la boucle).

— $\text{WLP}(10-14, I_3) = i == \backslash at(j, Pre) - j + \backslash at(i, Pre)[j \leftarrow j-1][i \leftarrow i+1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]$
$\equiv \text{WLP}(10-14, I_3) = i+1 == \backslash at(j, Pre) - j + 1 + \backslash at(i, Pre)$
$\equiv \text{WLP}(10-14, I_3) = i == \backslash at(j, Pre) - j + \backslash at(i, Pre) = I_3.$

Ainsi, $i < j \wedge I \Rightarrow \text{WLP}(10-14, I_3)$ car $\text{WLP}(10-14, I_3) = I_3$ et I_3 inclu dans I.

— $\text{WLP}(10-14, I_4) = \forall \text{ integer } k; (\backslash at(i, Pre) \leq k < i) \Rightarrow t[k] == \backslash at(t[j-k+i], Pre)[j \leftarrow j-1][i \leftarrow i+1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]$
$\equiv \text{WLP}(10-14, I_4) = \forall \text{ integer } k; (\backslash at(i, Pre) \leq k < i+1) \Rightarrow t[k] == \backslash at(t[j-1-k+i+1], Pre)$
$\equiv \text{WLP}(10-14, I_4) = \forall \text{ integer } k; (\backslash at(i, Pre) \leq k < i+1) \Rightarrow t[k] == \backslash at(t[j-k+i], Pre)$

Ainsi, $i < j \wedge I \Rightarrow \text{WLP}(10-14, I_4)$ est vrai : pour $\forall$ integer $k; (\backslash at(i, Pre) \leq k < i \Rightarrow t[k] == \backslash at(t[j-k+i], Pre))$ est vrai car I_4 est inclus dans I. Reste à montrer que $t[i] == \backslash at(t[j-i+i], Pre))$, soit $t[i] == \backslash at(t[j], Pre))$. Ceci est vrai, car, après la boucle, on a $t[i] == t[j]$; et que, avant le tour de boucle, par I_2 qui est inclus dans I, on a $t[j] == \backslash at(t[j], Pre)$.

— $\text{WLP}(10-14, I_5) = \forall \text{ integer } k; j < k \backslash at(j, Pre) \Rightarrow t[k] == \backslash at(t[j-k+i], Pre)[j \leftarrow j-1][i \leftarrow i+1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]$
$\equiv \text{WLP}(10-14, I_5) = \forall \text{ integer } k; j-1 < k \backslash at(j, Pre) \Rightarrow t[k] == \backslash at(t[j-k+i], Pre).$

$i < j \wedge I \Rightarrow \text{WLP}(10-14, I_5)$ est vrai par le même type d'argument

que pour I_4.

(d) Déduisez-en un triplet de HOARE valide pour votre fonction.

Réponse:
Avant de déduire le triplet de HOARE, il faut réussir à trouver les conditions pour que ces invariants soient vrais au début de la fonction. On applique donc $\text{WP}(7, I) = I$. Pour que nos invariants soient vérifiés à l'entrée de la boucle, il faut donc que :

- $0 \leq \backslash at(i, Pre) \leq i <= j+1 \leq \backslash at(j, Pre)+1 \leq n = 0 \leq i \leq j+1 \leq n$, ce qui implique donc qu'il faut les préconditions $0 \leq i \leq n$, $i \leq j+1$ et $0 \leq j < n$.
- $\forall$ integer $k; i \leq k \leq j \Rightarrow t[k] == \backslash at(t[k], Pre)$, ce qui est vrai par nature : avant de modifier les valeurs, les valeurs sont égales à leurs valeurs avant modification (ou ... ben oui ...)
- $i == \backslash at(j, Pre) - j + \backslash at(i, Pre)$ ce qui est équivalent en début de boucle à $i == j - j + i$, soit $i == i$, ce qui semble aussi vachement vrai.
- $\forall$ integer $k; (\backslash at(i, Pre) \leq k < i) \Rightarrow t[k] == \backslash at(t[j - k + i], Pre)$, ce qui est aussi vrai : $\forall$ integer $k; (\backslash at(i, Pre) \leq k < i)$ est équivalent à $\perp$ car i vaut sa valeur avant la boucle, et donc l'implication est vraie.
- $\forall$ integer $k; j < k \backslash at(j, Pre) \Rightarrow t[k] == \backslash at(t[j - k + i], Pre)$, ce qui est aussi vrai car la partie gauche de l'implication est toujours vraie.

On en déduit ainsi que le triplet de HOARE $\{ 0 \leq i \leq n \wedge i \leq j+1 \wedge 0 \leq j < n\}$ reverse $\{\Phi\}$ est un triplet valide.

(e) Démontrez la terminaison de la fonction en donnant un variant de boucle et en démontrant qu'il décroît à chaque tour de boucle et qu'il est toujours positif (vous pouvez utiliser le calcul de la question b).

Réponse:
On utilise le variant $V = j - i + 1$.
C'est bien un variant : en effet, d'une part, on a $i < j \wedge I \Rightarrow j - i + 1 \geq 0$: $i < j$ donc $j - i > 0$ donc $j - i + 1 \geq 0$.
D'autre part, on a $\text{WLP}(10 - 14, V) - \backslash at(V, Pre) = (j - 1) - (i + 1) + 1 - \backslash at(V, Pre) = j - 1 - i - \backslash at(V, Pre) = j - 1 - i - j + i - 1 = -2$ et donc V est strictement décroissant.

(f) Quelles sont les valeurs mémoires modifiées par cette fonction (et la boucle) ? Vous répondrez en donnant les clauses assigns et loop assigns correspondantes.

Réponse:
La boucle modifie les valeurs $i, j, t[\backslash at(i, Pre)..\backslash at(j, Pre)]$; et donc, la fonction modifie les valeurs $t[i..j]$. On a donc les clauses suivantes :

```
1
2 assigns t[i..j];
```

```
3
4  ...
5
6  loop assigns i,j,t[\at(i,Pre)..\at(j,Pre)] ;
```

(g) Que manque-t'il comme précondition pour que la fonction ne puisse pas faire d'erreur à l'exécution ?

> **Réponse:**
> Il faut exprimer qu'il est valide de lire et d'écrire pour toute adresse entre $t+i$ et $t+j$.

(h) Complétez les fichiers `reverse.c` et `reverse.h` de manière à ce que le contrat reflète tous les points vus ici et que frama-c accepte de prouver le contrat. Vous pouvez bien évidemment vous servir de cette question pour travailler les précédentes (c'est recommandé, d'ailleurs). On attendra bien sûr que toutes les assertions RTE soient démontrée, ainsi qu'une clause `terminates \true`.

(i) (Bonus) Proposez une version récursive de la fonction et démontrez (avec Frama-C) qu'elle vérifie le même contrat. En particulier, vous fournirez une clauses `decreases` permettant à la clause `terminates \true` d'être satisfaite (frama-c ne la vérifiera pas, mais moi oui).

Exercice 3 : Permutations minimales et maximales

On s'intéresse maintenant à des fonctions générant des permutations sur l'ensemble $\{0,\dots,n-1\}$, vues comme des tableaux.

Définition 0.1. *Une permutation sur* $\{0,\dots,n-1\}$ *est une fonction* P *qui à chaque entier* $i \in \{0,\dots,n-1\}$ *associe un entier* $P[i]$ *de* $\{0,\dots,n-1\}$ *et qui est une bijection.*

En C, on va stocker une bijection dans un tableau. On définit donc un bijection avec le prédicat suivant.

```
1  predicate isPermutation(int *t, integer n) =
2  \forall integer i; 0 <= i < n ==> 0 <= v[i] < n &&
3  \forall integer i,j; 0 <= i < j < n ==> v[i] != v[j];
```

(a) Justifiez (rapidement) que ce prédicat est bien équivalent à la définition d'une permutation.

> **Réponse:**
> Une application est bijective si tout élément de son ensemble d'arrivée a un et un seul antécédent. Ici, grâce à $\forall integer i; 0 <= i < n ==> 0 <= v[i] < n$ on voit bien que chaque nombre entre 0 et n possède une image dans le tableau comprise entre 0 et n. La deuxième moitié de la définition nous assure l'unicité de l'image dans l'ensemble d'arrivée, en effet on ne peut pas avoir deux valeurs identiques dans le tableau. Cette description du prédicat avec une clause atleast et une clause atmost nous assure la bijection entre les deux ensembles.

On va également considérer un ordre entre les permutations. On le définit via les trois prédicats suivants :

```
predicate unchangedTab{L,M}(int *s, int *t, integer i,
integer n) = \forall integer j; i <= j < n ==>
\at(s[j],L) == \at(t[j],M);

predicate isStrictlyBiggerPerm{L,M}(int *t, int *s,
integer n) = (\exists integer m; 0 <= m <= n-1 &&
(\at(t[m],L) < \at(s[m],M) &&
unchangedTab{L,M}(t,s,0,m)));

predicate isBiggerPerm{L,M}(int* t,int* s, integer n) =
isStrictlyBiggerPerm{L,M}(t,s,n) ||
(unchangedTab{L,M}(t,s,0,n));
```

(b) Quelles sont, selon cet ordre, les permutations minimales et maximales ? Donnez des prédicats `isMinPerm(int *t, integer n)` et `isMaxPerm(int *t, integer n)` qui ne sont vrais que si le tableau `t` est la permutation minimale (resp. maximale). Complétez ces prédicats dans `formalism.h`. Les deux premiers lemmes de la section exercice 4 devraient être démontrés, mais pas les autres (c'est normal).

Réponse:

```
/*@ predicate isMinPerm{L}(int *t, integer n) =
    $\backslash$ forall integer i ; 0<=i<n-1 ==>
    t[i] < t[i+1] ;

    predicate isMaxPerm{L}(int *t, integer n) =
    $\backslash$ forall integer i ; 0<=i<n-1 ==>
    t[i] > t[i+1] ;
*/
```

(c) Implémentez maintenant une fonction qui place dans `P` la permutation minimale, avec le prototype et le contrat partiel suivants. Respectez les styles que l'on sait démontrer sur papier, à savoir un seul point de sortie de la fonction et uniquement des boucles while (pas de for).

```
/*@
terminates \true;
ensures isPermutation(P,n);
ensures isMinPerm(P,n);
*/
void initPerm(int *P, unsigned int n);
```

(d) Déterminez un contrat complet pour cette fonction. Vous aurez évidemment besoin de déterminer des invariants de boucles. On attend à cette question que vous fournissiez les calculs de WLP adéquats.

Réponse:

Soit $\phi = isMinPerm(P, n) \wedge isPermutation(P, n)$

et $I = I_1 \wedge I_2 = 0 \leq i \leq n \wedge \forall 0 \leq k < n - i \Rightarrow P[k] == k$ $\text{WLP}(initPerm, \phi) \equiv \text{WLP}(5, \text{WLP}(13 - 16, \phi))$

$\equiv \text{WLP}(5, I) \equiv 0 \leq n \leq n \wedge \forall 0 \leq k < n - n \Rightarrow P[k] == k$

La plus faible précondition avec ces annotations est donc, si elle est définie, $0 < n$

Il nous reste à démontrer que $\text{WLP}(13 - 16, \phi)$ est bien définie. Pour cela il nous faut prouver les deux lemmes qui suivent.

— Lemme 1 : $\neg i > 0 \wedge I \Rightarrow \phi$:

$$\equiv \neg i > 0 \wedge 0 \leq i \leq n \wedge \forall 0 \leq k < n - i \Rightarrow P[k] == k \Rightarrow \phi$$

$\equiv i == 0 \wedge \forall 0 \leq k < n \Rightarrow P[k] == k \Rightarrow \forall 0 \leq k < n - 1 \Rightarrow P[k] < P[k+1] \wedge \forall 0 \leq k < n \Rightarrow 0 \leq P[k] < n \wedge \forall 0 \leq k < j < n \Rightarrow P[k] \neq P[j]$

$\equiv i == 0 \wedge \forall 0 \leq k < n \Rightarrow P[k] == k \Rightarrow \forall 0 \leq k < n - 1 \Rightarrow P[k] < P[k + 1] \equiv \top$

Car $P[k + 1] == k + 1$ et $P[k] == k$ donc $P[k + 1] > p[k]$

$\equiv i == 0 \wedge \forall 0 \leq k < n \Rightarrow P[k] == k \Rightarrow \forall 0 \leq k < n \Rightarrow 0 \leq P[k] < n \equiv \top$

Car k est compris entre 0 et $n - 1$, et $P[k] == k$

$\equiv i == 0 \wedge \forall 0 \leq k < n \Rightarrow P[k] == k \Rightarrow \wedge \forall 0 \leq k < j < n \Rightarrow P[k] \neq P[j] \equiv \top$

Car $P[k] == k$ donc pour chaque indice différent, on a une valeur différente.

On a donc bien $\neg i > 0 \wedge I \Rightarrow \phi$

— Lemme 2 : $i > 0 \wedge I \Rightarrow \text{WLP}(14 - 15, I)$:

On commence par calculer $\text{WLP}(14 - 15, I)$: $\text{WLP}(14 - 15, I) \equiv \text{WLP}(14, i - 1 > 0 \wedge 0 \leq i - 1 \leq n \wedge \forall 0 \leq k < n - (i - 1) \Rightarrow P[k] == k)$

$\equiv i - 1 > 0 \wedge 0 \leq i - 1 \leq n \wedge \forall 0 \leq k < n - i - 1 \Rightarrow P[k] == k$

Or, si $i - 1 > 0$, alors $i > 1$ et donc on a bien $0 \leq i \leq n \forall n \geq 0$

D'autre par on remarque si on a $\forall 0 \leq k < n - (i - 1) \Rightarrow P[k] == k$ alors on a aussi $\forall 0 \leq k < n - i \Rightarrow P[k] == k$ Car si la proprieté est vraie pour les $n - i + 1$ premiers indices de P alors elle est vraie pour les $n - i$ premiers indices aussi

On a donc bien $i > 0 \wedge I \Rightarrow \text{WLP}(14 - 15, I)$

Avec ces deux lemmes on peut donc conclure que $\text{WLP}(14 - 15, \phi)$ est bien défini, et donc que le calcul de weakest precondition effectué plus haut est correct.

On peut en conclure que $\{0 \leq n\} initPerm \{\phi\}$ est un triplet de Hoare valide.

(e) Démontrez la terminaison de cette fonction en donnant un variant pour votre boucle, et en démontrant que c'en est un.

Réponse:

On choisit le variant $n - i$, pour prouver que c'est un variant valide, il nous faut montrer qu'il est toujours positif à l'entrée de la boucle, et qu'il décroit

strictement à chaque tour de boucle.
$n - i \geq 0$ Car on a pour invariant $0 \leq i \leq n$ (ben voilà).
$\text{WLP}(14-15, n-i) - \backslash at(n-i, Pre) = n-(i+1)-n+i = n-i-1-n+i = -1 < 0$, Donc n-i strictement décroissant

(f) Votre implémentation de `initPerm` doit valider le contrat précédent dans Frama-C, ainsi que valider les assertions RTE, et comprendre la preuve de terminaison.

(g) Donnez une implémentation de la fonction dont le contrat partiel et le prototype suivent. Attention, alt-ergo n'arrivera a priori pas à tout démontrer, mais Z3 y arrivera. On ne demande pas de démontrer sur papier cette fonction, vous êtes donc libre sur le style. On attend que les assertions RTE soient démontrées, et que vous démontriez que la fonction termine.

```
/*@
terminates \true;
ensures isMaxPerm(P,n) ==> \result == true;
ensures !isMaxPerm(P,n) ==> \result == false;
*/
bool isMaxPerm(int *P, unsigned int n);
```

Cela dit, si vous ne parvenez pas à tout démontrer et souhaitez me montrer sur papier ce que vous essayez de faire et faire des preuves partielles, vous pouvez le faire en suivant :

Réponse:
Normalement notre implémentation réponds à tout

Exercice 4 : NextPermutation

On va maintenant s'intéresser à la fonction qui permet de considérer la prochaine permutation dans l'ordre défini précédemment.

```
bool NextPermutation(int *P, unsigned int n)
{
    unsigned int i = n - 1, j;
    j = n - 1;
    while (j > 0 && P[j - 1] >= P[j])
    {
        j--;
    }
    if (j == 0)
    {
        return false;
    }
    i = j - 1;
    while ((j < n) && (P[i] < P[j]))
        j++;
    j--;
```

```
17      swap(P + i, P + j);
18      reverse(P, i + 1, n - 1, n);
19      return true;
20  }
```

Le but de cet exercice est de déterminer un contrat de fonction pour cette fonction et de le prouver (avec Frama-C). Un mot d'avertissement : sur une telle fonction (et avec le genre de propriété qu'on souhaite montrer), les solveurs vont commencer à suer un peu : certaines propriétés ne seront démontrées qu'avec Z3 (et pas alt-ergo), et il y aura besoin de placer certaines assertions en cours de code pour les aider. En l'occurence, je vous fournis dans le fichier `NextPermutation.c` les assertions dont j'ai besoin pour démontrer mon contrat (et qui sont sans doute liées au fait qu'on a fait des appels de fonction). Les preuves utilisent les lemmes de la section Exercice 4 de `formalism.h` (qu'on ne demande pas de démontrer).

(a) Le premier but est donc de formaliser le contrat qui suit :
- Si la fonction reçoit une permutation maximale (i.e., qui satisfait `maxPerm(P,n)`), la fonction renvoie faux et ne modifie rien.
- Si la fonction reçoit une permutation non-maximale, alors elle renvoie vrai et place dans P une permutation strictement plus grande que celle qui y était à l'entrée de la fonction.
- Dans tous les cas, il y aura une permutation dans P à la fin de l'appel de la fonction.

Donnez une formalisation de ces post-conditions :

Réponse:

Logiquement, on a deux cas différents. Une formalisation en utilisant des comportements est donc de mise.

On a ainsi un premier comportement :

```
1  behavior maxPerm:
2      assumes isMaxPerm{Pre}(P, n);
3      ensures \result == false;
4      ensures unchangedTab{Pre,Post}(P, P, 0, n)  ;
```

Celui-ci exprime logiquement le fait que la permutation passée en paramètre est maximale. Nextpermutation doit donc ne pas changer la permutation et de renvoyer faux.

Le deuxième comportement est le suivant :

```
1  behavior notMaxPerm:
2      assumes !isMaxPerm{Pre}(P, n);
3      ensures \result == true;
4      ensures isStrictlyBiggerPerm{Pre,Post}(P, P, n) ;
```

Dans ce cas, la permutation P n'est pas une permutation maximale. On remarque que l'on demande simplement d'avoir une permutation plus grande et pas la permutation juste après. Afin d'exprimer une telle post-condition, on pourrai écrire une telle condition :

```
ensures \forall int * t;
    isStrictlyBiggerPerm{Pre,Post}(P,t,n)
    ==> isBiggerPerm{Post, Post}(P,t,n);
```

Ainsi on décrit que pour toute permutation strictement plus grande que la permutation P avant l'éxécution de NextPermutation, est plus grande ou égale à P après l'éxécution. Ce qui nous prouve bien que la permutation P à la fin de NextPermutation est bien la permutation qui vient juste après la permutation passée en paramètre de la fonction.

Bien entendu, on n'oublie pas la partie du contrat qui dit que l'on a bien une permutation à la fin ; présente hors des comportements.

```
ensures isPermutation(\at(P, Post), n) ;
```

Bon, je ne vais pas vous faire l'insulte de vous demander de calculer les WLP des deux boucles, puisque leur corps n'est pas ultra compliqué.

(b) Pour la première boucle, déterminez des invariants. Le but est surtout de déterminer ce que fait la boucle. Indication : la boucle s'arrête dès qu'on n'a trouvé un indice j pour lequel $P[j-1] > P[j]$. Que peut-on donc dire sur la fin du tableau si on est en j ? (ceci est une question réthorique pour que vous trouviez l'invariant). Déterminez ces invariants et justifiez (rapidement) que c'en sont : il faut deux invariants. Déterminez une clause assigns et un variant.

Réponse:

Lorsque l'on est en j, on a que des indices après qui sont décroissants, et comme on a une permutation, tous les indices sont différents donc c'est une décroissance stricte.

On a donc un premier invariant qui est le suivant :

```
loop invariant \forall integer i ; j<=i<n-1
==> P[i] > P[i+1] ;
```

C'est bien un invariant : c'est en fait l'essence même de la condition de boucle. En effet, on sort de la boucle seulement lorsque $P[j-1]$ est inférieur à $P[j]$. Donc, tant qu'on est dans la boucle, on est décroissant, et par construction, tout ce qui est après j est décroissant.

Le deuxième invariant est :

```
loop invariant n > j >= 0;
```

Il s'agit aussi d'un invariant... j décroit dans la boucle et commence à $n-1$ donc il est strictement inférieur à n ; et si $j == 0$, on sort de la boucle.

On a aussi les clauses assigns et variant suivantes :

```
    loop assigns j;
    loop variant j;
```

(c) Même question pour la seconde boucle. De la même manière, il faut être capable de décrire ce qui est vrai sur la portion du tableau exploré. Intuitivement, cette boucle place j sur l'élément suivant celui qui est le plus petit élément plus grand que P[i] (c'est pour ça qu'on fait j– après la boucle). Il y a besoin de deux invariants. Donnez la clause assigns et un variant.

Réponse:

Les deux invariants utilisés sont les suivants :

```
1 loop invariant i < j <= n ;
2 loop invariant \forall integer k ; i<k<j
3 ==> P[i] < P[k] ;
```

Le premier donne un encadrement sur j. Comme i vaut $j-1$ au départ de la boucle et que j ne peut que croître ; et qu'en plus on a la condition $j < n$ dans la boucle, c'est bien un invariant.

Le deuxième est plus intéressant ; il permet de comprendre le sens de la boucle. Il dit en effet que chaque case du tableau compris entre i et j et plus grande que $P[i]$. C'est aussi un invariant par construction même de la boucle : on augmente j seulement si le $P[j]$ actuel est plus grand que $P[i]$.

Les clauses assigns et variant sont les suivantes :

```
1     loop assigns j;
2     loop variant n-j;
```

(d) Terminez les annotations dans le fichier pour démontrer le contrat de cette fonction. En particulier, précisez les requires pertinent, la bonne clause assigns pour le contrat, et la clause terminates qui assure que cette fonction termine. Si vous n'utilisez pas de comportements, alors il vous faut un ensures de plus (qui dit que la permutation n'est pas modifiée si la permutation en entrée est maximale). Si vous utilisez des comportements, il est possible d'avoir des clauses assigns différentes et de se passer de cet ensures.

(e) Le contrat qu'on a donné ici n'est en réalité pas complet (ce qui manque est explicité dans l'exercice 5). Donnez une fonction qui respecte le même contrat (mais plus simple), mais qui ne répond pas à l'idée intuitive de ce que fait `nextPermutation` (on n'attend pas nécessairement que vous la prouviez avec Frama-C, vous pouvez vous contenter d'expliquer pourquoi elle satisfait le contrat). Indication : cette fonction pourrait systématiquement retourner la même permutation.

Réponse:

Le contrat ne demande pas à ce que la fonction renvoie la permutation suivante. En particulier, une fonction qui vérifie si la permutation est maximale, qui renvoie faux si oui, ou qui sinon met la permutation maximale dans P serait vérifiée :

```
1  bool NextPermutation(int *P, unsigned int n)
2  {
3      unsigned int i = n - 1, j;
4      j = n - 1;
5      while (j > 0 && P[j - 1] >= P[j])
6      {
7          j--;
8      }
9      if (j == 0)
10     {
11         return false;
12     }
13     i = 0 ;
14     while (i < n) {
15         P[i] = n-i-1 ;
16         i++ ;
17     }
18     return true;
19 }
```

Exercice 5 : tsp

On va maintenant tenter de démontrer une fonction qui implémente un algo Brute Force pour le travelling salesman problem. Pour cela, on va éluder un détail : on admet l'existence d'une fonction value qu'on abstrait (techniquement, il faudrait plus d'arguments, mais ici on s'en fiche). On va également avoir besoin de quelques lemmes qu'on admettra (les solveurs n'arrivent pas à les démontrer). On va également ajouter la partie du contrat qui manque à la fonction précédente, qu'on admettra également (enfin, si vous arrivez à le démontrer, je serai impressionné). Tout cela est définit dans `tsp.h` et dans `formalism.h`.

À partir de ça, on considère le code suivant :

```
1  /*@
2  ensures \forall int* t; \separated(t+(0..n-1),P+(0..n-1))
3  ==> isPermutation(t,n) ==> value(t,n) >= \result;
4   */
5  int tsp(unsigned int n, int *P)
6  {
7      initPerm(P, n);
8      int val = value(P, n);
9      while (!(isMaxPerm(P, n)))
10     {
11         NextPermutation(P, n);
12         if (val > value(P, n))
13             val = value(P, n);
14     }
15     return val;
16 }
```

Une note : dans le contrat que nous avons fourni, vous noterez la présence d'une clause `\separated`. Sans cela, Frama-C est incapable de démontrer le contrat. Cela est dû au fait qu'on formalise nos permutations comme des tableaux, et non réellement comme des objets abstraits, et que comme on modifie la permutation courante via `NextPermutation`, les solveurs ne sont pas capables de montrer qu'une formule quantifiant universellement sur les permutations n'est pas impactée par l'appel de `NextPermutation`. En précisant qu'on ne regarde que les permutations qui ne recoupent pas P, ils s'en sortent par contre. Ça devrait suffire pour être convaincant, mais ça rend la formalisation un peu plus pénible. Mais je ne vois pas d'autre moyen de quantifier universellement sur des permutations.

Vous aurez bien évidemment besoin de garder cette hypothèse de séparation dans vos invariants.

Frama-C n'arrivera pas à prouver le contrat sans assertions. Celles qui sont déjà placées dans `tsp.c` devraient être suffisantes.

(a) Ajoutez les annotations de boucle et les préconditions nécessaires pour démontrer la correction partielle de cette fonction.

Réponse:

Un petit point que nous souhaitons aborder : il nous a été impossible de valider sans condition la correction partielle de cette fonction.

En effet, tout est prouvé en vert ou vert/orange sauf un assert, qui est en orange. Il s'agit de l'assert suivant :

```
1     assert \forall int* t;
2     \separated(t+(0..n-1),P+(0..n-1))
3     ==> isBiggerPerm{Here,L}(t,P,n)
4     ==> isBiggerPerm{L,L}(t,P,n);
```

Il nous a néanmoins semblé inutile de se triturer (vraiment) trop la tête, car il nous semble logique que ceci soit vrai.

En effet, cet assert demande à ce que si un tableau t est séparé de P, et que P au temps L est plus grand que t maintenant, alors P au temps L est plus grand que t au temps L.

La raison pour laquelle il nous semble logique que cet assert soit vrai est que frama-c réussit à prouver sans problème l'assert suivant :

```
1 assert \forall int* t;
2 \separated(t+(0..n-1),P+(0..n-1))
3 ==> unchangedTab{L,Here}(t,t,0,n);
```

Qui dit que si t est séparé de P, alors t au temps L est égal à t maintenant. Ainsi, si un tableau t est séparé de P, et que P au temps L est plus grand que t maintenant, comme t au temps L est égal à t maintenant, P au temps L est plus grand que t au temps L.

(b) Démontrez sur papier que la fonction termine, en donnant un variant de boucle. Ce variant pourra être dans un autre ordre bien fondé que sur $\mathbb{N}, <$.

Je ne demande pas de le faire sur Frama-C (vous pouvez si vous le souhaitez, mais

je pense que c'est un peu technique à écrire proprement).

Réponse:

On peut prouver que la fonction termine en utilisant un variant sur $[[1..n]]^n$, et l'ordre utilisé est l'ordre classique sur $\mathbb{N}^n$: c'est à dire $(x_1, x_2, ..., x_n) < (x'_1, x'_2, ..., x'_n)$ si il y a un i compris entre 1 et n tel que $x_j == x'_j$ pour chaque j strictement plus petit que i ; et $x_i < x'_i$ (on a pas de condition sur les indices après i). Cet ordre est bien un wqo car l'ensemble considéré est fini.

En outre, on a bien à chaque tour de boucle un accroissement de P : si on note P_{pre} la valeur de P avant le tour de boucle, et P_{post} la valeur après le tour de boucle, comme si on entre dans la boucle on effectue un NextPermutation, et que si on entre dans la boucle, P_{pre} n'est pas maximal, on est sûr d'avoir le comportement $notMaxPerm$ qui s'applique, et on a donc $P_{pre} < P_{post}$: c'est exactement la proprieté de ce comportement.

En outre, comme on a une permutation, P est toujours dans $[[1..n]]^n$. Ainsi, la valeur de P par les tours de boucles effectuent une chaine strictement croissante d'éléments de $[[1..n]]^n$ sur un ordre wqo, et donc comme on a un élément maximal car $[[1..n]]^n$ est fini, on est sûr de terminer la boucle.

(c) (Bonus ++) Frama-C n'arrive pas à démontrer certains des lemmes qu'on a admit. Faites-le vous sur papier.

Réponse:

(d) (Bonus ++++) Démontrez que la fonction `nextPermutation` vérifie le contrat supplémentaire qu'on a ajouté à cet exercice. Sur papier, c'est faisable (avec quelques arguments). Sur Frama-C, je n'ai pas réussi (mais pas tenté tant que ça), donc si vous y arrivez, je serai impressionné.

Réponse:

METTEZ VOTRE RÉPONSE ICI.