TD n°3 - Preuves de programmes

Exercice 1

La fonction suivante renvoit le minimum d'un tableau :

```
Fonction minimum :
Entrées : n un entier positif et tab un tableau de n éléments
Soit mini = tab[0]
Pour i allant de 0 a n-1 inclus faire
  si tab[i] < mini alors mini = tab[i]
Retourner mini
```

- Écire une spécification pour cette fonction.
  Précondition : **tab** est un tableau de valeurs ordonnées totalement et **n** est un entier positif qui est sa taille.
  Postcondition : Le programme renvoie le minimum du tableau.
- Trouver un invariant de la boucle pour tout.
  Un invariant est "**mini** est le minimum du tableau réduit à ses indices entre 0 et i".
  En effet supposons cette propriété vraie à l'itération $i-1$, c'est à dire que **mini** est le minimum des éléments du tableau entre les indices 0 et $i-1$. À l'itération i on a deux choix possibles :
  - **tab[i]>mini**. Dans ce cas **mini** ne change pas et est le minimum des éléments du tableau entre les indices 0 et i.
  - **tab[i]<mini**. Dans ce cas **mini** a comme nouvelle valeur **tab[i]**. **tab[i]** est donc plus petit que tous les éléments précédents du tableau, et est le minimum.
- Justifier la correction de cette fonction
  Cette fonction termine car elle ne contient qu'un boucle **for**.
  Montrons que l'invariant est vrai à la première itération de la boucle : on a alors $i = 0$ et **mini = tab[0]**. Or **tab[0]** est bien le minimum des éléments du tableau $[tab[0]]$.
  Ainsi l'invariant est vrai après la boucle (qui termine), par principe de récurrence. Il s'exprime alors "**mini** est le minimum du tableau réduit à ses indices entre 0 et $n-1$". La valeur renvoyée est bien le minimum du tableau.

Exercice 2

La fonction suivante calcule le terme k de la suite $(u_n)_{n\in\mathbb{N}}$ définie par $u_0 = c$ et $\forall n \in \mathbb{N}$, $u_{n+1} = a * u_n + b$ pour a, b, c des entiers :

```
int k_ieme(int a, int b, int c, int k){
  int res = c;
  for(int i=1; i<=k;i+=1){
    res = a*res+b;
  }
  return res;
}
```

1. Écrire une spécification pour cet algorithme.
   Précondition : $a, b, c \in \mathbb{Z}$ et $k \in \mathbb{N}$.
   Postcondition : La valeur renvoyée est u_k.
2. Justifier que ce programme termine. Ce programme ne contient qu'une boucle **for**, donc il termine.
3. Trouver un invariant.
   Un invariant est "**res** = u_i".
   Supposons en effet que ce soit vrai à l'itération $i-1$. Alors **res** = u_{i-1}. La nouvelle valeur de **res** est $a*u_{i-1}+b = u_i$.
4. Justifier la correction de ce programme.
   L'invariant est vrai avant la boucle pour $i = 0$ (ici sous-entendu, si ça vous gene, modifiez le code) puisque **res** = $c = u_0$. Il est donc également vrai après la fin de la boucle puisque celle-ci termine.
   On a donc **res** = u_k à la fin, soit le résultat voulu.

Exercice 3

La fonction suivante a pour but de calculer une valeur approchée de la racine carrée de $n \in \mathbb{N}$.

```
Fonction racine_approchée
Entrées : n un entier positif
Soit res=0 et s=1
Tant que s<=n:
    res = res+1
    s = s+2*res+1
Renvoyer res
```

1. Écrire une spécification de l'algorithme (précondition, postcondition)
   Précondition : $n \in \mathbb{N}$
   Postcondition : la fonction renvoie un entier r positif tel que $r \leq \sqrt{n} < r+1$ avec $r = \sqrt{n}$ si n est carré d'entier.
2. Démontrer qu'il se termine à l'aide d'un variant de boucle.
   On commence par remarquer que **res** est positif (c'est un invariant !). La preuve est simple, **res** part de 0 et augmente de 1 à chaque tour de boucle.
   Un variant est $n - s$. Puisque **res** est positif, la nouvelle valeur s' de s au cours d'une itération, qui vaut $s' = s + 2 * res + 1$, est strictement plus grande que s. $n - s$ reste un entier positif tant que la boucle ne s'arrête pas au vu de la condition de fin.
3. Montrer que $(res+1)^2 = s$ est un invariant de la boucle.
   Supposons que l'invariant est vrai à la i-ème itération. Au début de la i+1-ème itération on a donc $(res+1)^2 = s$. À la fin de la i+1-ème itération, la nouvelle valeur de **res** est $res' = res + 1$. La nouvelle valeur de s est $s' = s + 2 * res' + 1$.
   On a donc $s' = res'^2 + 2res' + 1 = (res' + 1)^2$.
4. En déduire que la postcondition est vérifiée.
   L'invariant est vrai avant la boucle, donc il vrai à la fin de la dernière itération. On note res_{fin} et res_{ad} les valeurs de **res** pour la dernière itération et l'avant dernière. On note de la même manière s_{fin} et s_{ad}.
   On a d'après l'invariant $(res_{fin} + 1)^2 = s_{fin}$ et $(res_{ad} + 1)^2 = res_{fin}^2 = s_{ad}$. Puisqu'on s'intéresse à la dernière itération, on a $s_{fin} > n$ car la boucle s'arrête ensuite et $n \geq s_{ad}$ car la boucle ne s'est pas arrêtée, d'où, par croissance de la racine carrée sur $\mathbb{R}^+$, $res_{fin} + 1 > \sqrt{n} \geq res_{fin}$. En particulier si $n = k^2$ pour un $k \in \mathbb{m}$, alors k est un entier qui vérifie $res_{fin} + 1 > k \geq res_{fin}$. On a donc $k = res_{fin}$.

Exercice 4

Outre le fait de permettre de prouver la correction et la terminaison, réfléchir aux invariants en avance permet d'écrire du code bien ordonné.

Dans cet exercice on cherche à écrire un algorithme réalisant la division euclidienne de a par b avec $a, b \in \mathbb{N}$ différents de $(0, 0)$. (Précondition)

On cherche donc un couple $q, r \in \mathbb{N}$ tel que $a = bq + r$ avec $r \in [|0, b-1|]$. (Postcondition)

On vous donne le squelette de code suivant, pour ne pas avoir à renvoyer q et r (ce qui en C n'est pas immédiat), on les affiche :

```
void affiche_div(int a, int b){
 assert(a>=0);
 assert(b>=0);
 assert( !(a==0&&b==0) );

 int q = 0;
 int r = a;

 while(r>=b){
   q = q+1;
   r = r-b;
 }
 printf("quotient = %d et reste = %d",q,r);
}
```

Le code précédent utilise des variables q et r qui vérifient toujours $a = qb + r$ et $0 \leq r$ tout au long de l'algorithme, mais pas forcément $r < b$. De plus, un variant de la boucle doit être r.

1. En considérant les deux invariants et le variants, déduire la condition de fin de la boucle.
   On a tout le temps $a = bq + r$ et $0 \leq r$, donc si on s'arrête quand $r < b$, on aura trouvé le reste voulu.
2. Compléter l'initialisation de q et r pour que les invariants soient vérifiés.
   On va au plus simple, $a = b * 0 + a$. (d'autres choix sont possibles : $a = b * 1 + a - b$, ...)

3. Compléter le corps de la boucle pour que les invariants et le variant soient satisfaits.
   On augmente q de 1, on diminue r de b (c'est l'algorithme habituel)

Exercice 5

Pour chacune des fonctions récursives suivantes, montrer qu'elles sont correctes (et qu'elles terminent).

Cette fonction calcule x^n

```
int exp_semi_rapide(int x, int n){
  if (n==0){return 1;}
  else if (n%2==0){return exp_semi_rapide(x*x,n/2);}
  else {return x*exp_semi_rapide(x,n-1);}
}
```

Soit $x, n \in \mathbb{N}$, on veut montrer que **`exp_semi_rapide(x,n)`** renvoie x^n. Pour cela on procède par disjonction de cas :

- Si n = 0 alors on renvoie $1 = x^0$
- Si n est pair alors on fait un appel récursif. On suppose que cet appel est correct, c'est à dire qu'il renvoie bien $(x*x)^{\frac{n}{2}} = x^n$. On ne fait que renvoyer cette valeur, donc on renvoie bien x^n.
- Si n est impair alors on fait un appel récursif On suppose que cet appel est correct, c'est à dire qu'il renvoie bien $(x)^{n-1}$. On renvoie alors $x * x^{n-1} = x^n$.

La fonction renvoie toujours x^n, elle est correcte.

Cette fonction détermine si n est pair.

```
bool est_pair(int n) =
  if (n==0){return true;}
  if (n==1){return false;}
  return est_pair(n-2);
```

Soit $n \in \mathbb{N}$, on veut montrer que **`est_pair(n)`** renvoie true si n est pair et false sinon. Pour cela on procède par disjonction de cas :

- Si n = 0 alors on renvoie true, qui est le bon résultat.
- Si n = 1 on renvoie false, qui est le bon résultat.
- Sinon on renvoie **`est_pair(n-2)`**. On suppose que cet appel récursif renvoie le bon résultat, c'est à dire true si $n-1$ est pair et false sinon. Alors puisque n et $n-2$ ont même parité, on renvoie bien le bon résultat.

Le but de cette fonction est de déterminer la somme des éléments d'un tableau, mais il faut que vous commenciez par préciser exactement la somme de quels éléments du tableau elle calcule.

```
int somme_rec(int* tab, int i, int taille_tab){
  if (i==taille_tab){return 0;}
  else {return tab[i]+somme_rec(tab,i+1,taille_tab);}
}
```

Soit $t = [e_0; e_1; ...; e_{n-1}]$ un tableau d'entiers, on veut montrer que **`somme_rec(t, i, n)`** renvoie $\sum_{k=i}^{n-1} e_k$. Pour cela on procède par disjonction de cas :

- Si $i = n$, alors $\sum_{k=n}^{n-1} e_k = 0$. Le résultat correct est donc bien 0
- Sinon on renvoie e_i ajouté au résultat de **`somme_rec(tab,i+1,taille_tab)`**. On suppose que cet appel est correct, donc qu'il renvoie $\sum_{k=i+1}^{n-1} e_k$.

  Alors le résultat final renvoyé est $\sum_{k=i}^{n-1} e_k$

Exercice 6 (*)

On considère l'algorithme suivant, dit "algorithme de recherche dichotomique".

Précondition : tab est un tableau trié dans l'ordre croissant et valeur est une valeur du bon type.

Postcondition : la fonction renvoie un indice i tel que $tab[i] = valeur$ si un tel i existe et -1 sinon.

```
Fonction recherche
Entrées tab un tableau et valeur un élément
Soit gauche = 0 et droite = longueur(tab)-1
Tant que gauche <= droite:
```

```
    m = (gauche+droite)/2
    Si valeur = tab[m] alors renvoyer m et arrêter l'algorithme
    Sinon, si valeur < tab[m] alors droite = m-1
          sinon gauche = m+1
Renvoyer -1
```

1. droite est il un variant de la boucle tant que? gauche est il un variant? Comment construire un variant avec droite et gauche? (faire des dessins de l'évolution de droite et gauche sur plusieurs exemples)

gauche n'est pas un variant car il aumente au lieu de diminuer. droite (et -gauche) n'est pas un variant car une itération change SOIT droite, SOIT gauche. Il y a donc des itérations où droite ne change pas.

droite-gauche est un variant. En effet on a toujours gauche$\leq$droite dans la boucle, donc $0 \leq$ droite-gauche et à chaque tour de boucle on a SOIT droite qui diminue de 1 et droite-gauche aussi, SOIT gauche qui augmente de 1 et droite-gauche qui diminue de 1. Dans les deux cas droite-gauche diminue de 1.

2. En déduire la terminaison.

On a trouvé un variant.

3. Supposons que valeur est dans tab. Quel peut être un invariant? Prouver l'invariant.

Un invariant est : "Il existe un indice i entre droite et gauche tel que $tab[i]$ =valeur"

4. En déduire que soit valeur est dans le tableau et la fonction renvoie son indice, soit valeur n'est pas dans le tableau et la fonction renvoie -1.

Supposons que valeur est dans le tableau. Alors l'invariant nous indique que la solution à renvoyer, i, est toujours situé entre droite et gauche.

L'invariant est vrai avant la boucle car l'intervalle entre gauche t droite contient tous les indices du tableau. Comme la boucle termine, l'invariant est également vrai après.

La boucle peut se terminer de deux façons : soit on trouve m tel que $tab[m]$=valeur et on renvoie m, la fonction est alors correcte, soit gauche devient strictement plus grand que droite, donc valeur est dans un tableau vide, ce qui est absurde.

Donc si valeur est dans le tableau, la boucle ne peut se terminer que par le renvoi de m.

Supposons maintenant que valeur n'est pas dans le tableau. La boucle se terminera sans jamais passer par le renvoi, puisque la condition si ne peut pas être vérifiée. Donc on renverra -1, ce qui est le résultat attendu.

Dans tous les cas, on a montré que la fonction renvoie le bon résultat.