Travaux Pratiques de Why3 et MicroPython
Licence Informatique
UE Algo 2

Bogdan Bibyk, Jean-Paul Bodeveix, Antoine Colonna d'Istria, Mamoun Filali

Juillet 2024

Table des matières

Chapitre 1

Exercices de TP

Dans cette partie, nous verrons des exercices qui feront l'objectif des travaux pratiques, ils sont groupés selon le sujet abordé. Certains d'entre eux sont marqués par un ou plusieurs symboles $\star$, ce qui indique leur difficulté par rapport aux autres exercices.

1.1 Somme des n premiers carrés

— Définir le prédicat `carres_post(n:int, sum:int)` pour exprimer la propriété suivante : $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

— Écrire la spécification de l'algorithme de somme des n premiers carrés, en utilisant le prédicat défini précédemment. La tester sur plusieurs exemples ci-dessous.

```
1  def carres(n:int) -> int:
2      #@requires ...
3      #@ensures ...
4      pass
5
6  r = carres(0)
7  #@assert r==0
8  r = carres(1)
9  #@assert r==1
10 r = carres(2)
11 #@assert r==5
12 r = carres(3)
13 #@assert r==14
14 r = carres(4)
15 #@assert r==30
16 r = carres(5)
17 #@assert r==55
18 ...
19
20 # contre exemples
21 r = carres(7)
22 #@ assert r != 26
23 r = carres(8)
24 #@ assert r != 21
25 r = carres(9)
26 #@ assert r != 13
```

1.2 Entier dont la valeur absolue est le maximum parmi 3 entiers

Écrire la spécification de l'algorithme qui renvoie le paramètre dont la valeur absolue est le maximum des valeurs absolues des trois paramètres. Tester sur plusieurs exemples et contre-exemples cette spécification.

```
#@ function abs(n:int) -> int = if n >= 0 then n else -n

def abs_max(a:int,b:int,c:int) -> int:
  #@ ensures ...
  pass

r = abs_max(5,8,2)
#@ assert r == 8
#Parce que abs(8)>abs(5) et abs(8)>abs(2), on retourne donc 8
r = abs_max(-2,4,-6)
#@ assert r == -6
#Parce que abs(-6)>abs(4) et abs(-6)>abs(-2), on retourne donc -6
r = abs_max(17,-3,2)
#@ assert r == 17
r = abs_max(-2,-4,-10)
#@ assert r == -10

r = abs_max(4,-10,8)
#@ assert r != 8
r = abs_max(7,-1,-9)
#@assert r != 9
```

1.3 Calcul du carré

Cet exercice correspond à corriger la fonction de calcul du carré. La spécification de la fonction est la suivante :

```
def Carre(n):
    #@requires n >= 0
    #@ensures result==n*n
    pass
```

Cherchez à trouver quelles lignes modifier pour que le programme soit bien vérifié par l'environnement Why3 :

```
def Carre(n):
    #@requires n >= 0
    #@ensures result==n*n
    i, c, k = 1, 1, 1
    while (i < n):
        #@variant n-i
        #@invariant c==i*i
        #@invariant 0<=i<=n
        #@invariant k==2*i+1
        c = c+k
        k = k+2
        i = i+1
    return c
```

1.4 Tri de 3 variables

Nous avons vu les deux premières versions du tri de 3 variables dans lors de l'introduction aux TPs why3. Testez la variante tri3V2 du chapitre 5 en remplaçant le variant par l'ordre lexicographique `x,y,z`. Pourquoi cela ne marche pas ? Vérifiez que cela fonctionne en restreignant le type des entrées/sorties.

1.5 Calcul itératif de la fonction de Fibonacci

Cet exemple a pour but de vérifier l'implantation itérative du calcul de la fonction `fib` par rapport à sa définition récursive.

On commence par définir récursivement la fonction `fib`.

— Définir récursivement la fonction `fib`. Elle devra être totale sur le type `int`. On la prolongera par l'identité sur les entiers négatifs.

— Démontrer que $\forall n, n > 0 \rightarrow \text{fib}(n) > 0$ en s'inspirant de la preuve d'une propriété similaire sur `fact` (voir Introduction : `6.2 Démonstration d'un lemme`).

— Écrire la spécification de l'algorithme itératif calculant `fib(n)` pour $n > 0$. Vous pouvez tester cette spécification sur quelques exemples et contre-exemples :

```
1  def m_fib(n:int)->int:
2      #@requires ...
3      #@ensures ...
4      pass
5
6  r = m_fib(0)
7  #@assert r == 0
8  r = m_fib(1)
9  #@assert r == 1
10 r = m_fib(2)
11 #@assert r == 1
12 r = m_fib(3)
13 #@assert r == 2
14 r = m_fib(4)
15 #@assert r == 3
16
17 # contre exemples
18 r = m_fib(3)
19 #@ assert r != 5
20 r = m_fib(4)
21 #@ assert r != 7
22 r = m_fib(5)
23 #@ assert r != 13
```

— Développer l'algorithme itératif en se limitant aux entiers strictement positifs (modifier la spécification et les tests en conséquence). On introduira deux variables auxiliaires `fp` et `fn` et on prendra comme invariants :

$$1 \leq i \leq n \tag{1.1}$$

$$\text{fp} = \text{fib}(i-1) \tag{1.2}$$

$$\text{fn} = \text{fib}(i) \tag{1.3}$$

On démarrera la boucle avec $i = 1$. Le résultat sera lu en sortie de boucle dans `fn`. La boucle doit seulement contenir une affectation multiple.

— Remplacer l'affectation multiple par une séquence de quatre affectations et ajouter les cinq assertions intermédiaires en calculant manuellement les WP. On démarrera par la conjonction des invariants à la fin du corps de la boucle.

```
1  while (Condition de boucle) :
2      #@ (Variant)
3      #@ (Invariants)
4      #@assert (4ème assertion à ajouter)
5      (1ère affectation)
6      #@assert (3ème assertion à ajouter)
7      (2ème affectation)
8      #@assert (2ème assertion à ajouter)
```

```
    (3ème affectation)
    #@assert (1ère assertion à ajouter)
    (4ème affectation)
    #@assert 1 <= i <= n and fp == fib(i-1) and fn == fib(i)
return fn
```

— Ajouter le traitement du cas $n = 0$. Modifier la spécification, les tests et le code.

1.6 Cas particulier de l'identité d'Ocagne

Cet exercice est en continuité avec le précédent, portant également sur la suite de Fibonacci. Il propose de prouver le cas particulier suivant de l'identité d'Ocagne : $\forall n \geq 2, fib(n)^2 - fib(n-1) \times fib(n+1) = (-1)^{n+1}$
Par exemple : $1^2 - 1 \times 2 = -1$; $2^2 - 1 \times 3 = 1$; $3^2 - 2 \times 5 = -1$; $5^2 - 3 \times 8 = 1$.

— Reprendre la fonction `fib(n:int)->int` de l'exercice précédent.

— Définir la fonction récursive `signe(n:int)->int` qui prend en paramètre n>0, une puissance, et qui retourne $(-1)^n$. Aidez-vous de l'ébauche suivante :

```
#@function
def signe(n:int) -> int:
    #@requires ...
    #@variant ...
    #@ensures ...
    return ... if ... else ...
```

— Faire une preuve par induction de la proposition $\forall n \geq 2, fib(n)^2 - fib(n-1) \times fib(n+1) = (-1)^{n+1}$. Pour cela, compléter sa définition en faisant un appel récursif lorsqu'on n'est pas sur le cas de base.

```
def ocagne(n:int) -> unit:
    #@requires ...
    #@variant ...
    #@ensures ...
    if n>2:
        ...
```

1.7 Division entière

Écrire l'algorithme calculant le quotient q et le reste r de la division entière d'un entier naturel a par un entier $b > 0$. Il renverra un type `Tuple[int, int]`. On commence par écrire la spécification (n'oubliez pas les propriétés vérifiées par le reste) et on la teste sur quelques exemples.

```
def div_mod(a:int,b:int)->Tuple[int,int]:
    #@requires ...
    #@ensures ...
    pass
r = div_mod(5,3)
#@assert r == (1,2)
r = div_mod(2,2)
#@assert r == (1,0)
r = div_mod(17,5)
#@assert r == (3,2)
r = div_mod(24,5)
#@assert r == (4,4)
r = div_mod(1,4)
#@assert r == (0,1)

# contre exemples
r = div_mod(7,5)
#@assert r != (2,-2)
r = div_mod(14,4)
```

```
20 #@assert r != (3,1)
```

En déduire l'invariant et écrire la boucle. On part de $q = 0$ et $r = a$ et à chaque passage de la boucle, on retranchera b à r tout en incrémentant q jusqu'à obtenir le reste. Ajouter le calcul manuel des WP.

```
1  def div_mod(a:int,b:int) -> Tuple[int,int]:
2      #@requires ...
3      #@ensures ... # proprietes du reste
4      #@ensures ...
5      q,r = 0,a
6      while (...):
7          #@invariant ...
8          #@invariant ...
9          #@variant ...
10         ...
11     return (q,r)
```

1.8 Racine carrée entière par dichotomie

Écrire l'algorithme de recherche de la partie entière de la racine carrée d'un entier naturel `N`. Commencer par écrire la spécification. La tester sur quelques exemples. Le résultat r devra être tel que $r^2 \leq N < (r+1)^2$. On procédera par dichotomie en partant de l'intervalle $0..N+1$.

1.9 Exponentiation rapide

Cet exercice a pour but de vérifier un algorithme de calcul de la puissance. La preuve n'est pas entièrement automatique. Il sera nécessaire d'introduire un lemme qui sera prouvé automatiquement.

— Définir la fonction récursive `power(n:int,m:int):int` calculant n^m.

— Écrire la spécification de l'algorithme et la tester sur des exemples simples.

— Écrire l'algorithme d'exponentiation rapide selon le schéma suivant :

```
1  def expR(A:int, B:int) -> int:
2      #@requires ...
3      #@ensures ...
4      x,y = A,B
5      z = 1
6      while (y>0):
7          #@variant y
8          #@invariant z*power(x,y)==power(A,B)
9          #@invariant B >= y
10         if (y%2 == 0):
11             x,y = ..., y//2
12         else:
13             z,y = ..., y-1
14     return z
```

— On notera que la preuve n'est pas automatique. Afin de localiser le problème, ajouter le 1er invariant comme assertion de fin de boucle. Calculer manuellement la WP de l'instruction dans chaque branche du IF et de l'assertion et l'ajouter comme assertion.

```
1  def expR(A:int, B:int) -> int:
2      #@requires ...
3      #@ensures ...
4      ...
5          if (y%2 == 0):
6              #@assert ...==power(A,B)    (*)
7              x,y = ..., y/2
8              #@assert z*power(x,y) == power(A,B)
```

```
9           else:
10              #@assert ...== power(A,B)
11              z,y = ..., y-1
12              #@assert z*power(x,y) == power(A,B)
13          #@assert z*power(x,y) == power(A,B)
14      return z
```

On notera que l'assertion (*) n'est pas démontrée automatiquement.

— Introduire un lemme, voir le chapitre Lemmes de l'introduction, énonçant la propriété de `power` manquante : $(x^2)^y = x^{2y}$:

```
1  def pow_sqr(x,y):
2      #@requires x >= 0
3      #@requires y >= 0
4      #@ensures ... == ...
5      pass   # on ne le demontre pas tout de suite
```

— Vérifier la pertinence du lemme en ajoutant un appel au lemme avant l'assertion (*). Il ne devrait plus y avoir d'erreur (mais le lemme est pris comme un axiome).

```
1      if (y%2 == 0):
2          #@call pow_sqr(...,...)
3          #@assert ... == power(A,B) // (*)
```

— Tenter la preuve automatique du lemme après avoir remplacé `pass` par `None`. C'est encore un échec...

— Faire une preuve par induction de ce lemme : pour cela, compléter sa définition en faisant un appel récursif lorsqu'on n'est pas sur le cas de base. Il faudra aussi indiquer le variant.

```
1  def pow_sqr(x:int,y:int):
2      #@requires x >= 0
3      #@requires y >= 0
4      #@variant ...
5      #@ensures ... == ...
6      if ...:
7        pow_sqr(...,...)
```

1.10 Le pgcd ⋆⋆

— Écrire la spécification de la fonction `pgcd` en utilisant la fonction % pour tester la divisibilité. La tester sur quelques exemples et contre-exemples. On reviendra à la définition mathématique du pgcd : p est le pgcd de deux entiers naturels a et b supposés non simultanément nuls si p divise a et b et si tout diviseur commun de a et b divise p.

Tester cette spécification.

```
1  def pgcd(a:int,b:int) -> int:
2      #@requires a >=0 and b >= 0
3      #@requires a > 0 or b > 0
4      #@ensures result > 0
5      ...
```

```
1  r = pgcd(1,1)
2  #@ assert r == 1
3  r = pgcd(6, 10)
4  #@ assert r == 2
5  r = pgce(8,4)
6  #@ assert r == 4
7  ...
```

— Écrire l'algorithme introduisant deux variables u,v entières positives, non toutes les deux nulles, et telles que les diviseurs communs de u et v soient exactement les diviseurs communs de a et b. On progressera en passant de (u, v) à $(u - v, v)$ lorsque $u \geq v$. On s'arrête lorsque u ou v est nul. On en déduit alors p. Ajouter le variant. Ajouter manuellement le calcul des WP. On aura besoin de trois lemmes et un axiome qu'on ne cherchera pas à démontrer :

```
#@lemma add_mod: forall u,v. forall i . i>0 and u%i==0 and v%i==0 -> (u+v)%i==0

#@lemma diff_mod1: forall u,v. forall i . i>0 and (u-v)%i==0 and v%i==0 -> u%i==0

#@axiom opp_mod: forall u,i. i>0 and u%i==0 -> (-u)%i==0

#@lemma diff_mod2: forall u,v. forall i . i>0 and u%i==0 and v%i==0 -> (u-v)%i==0
```

1.11 Recherche linéaire d'un élément présent dans un tableau

— Spécifier le sous-programme `recherche_lin_present` prenant en argument un tableau d'entiers A et un entier X supposé présent dans le tableau et renvoyant dans `pos` la position de X dans ce tableau. Pour tester la spécification, il faut bien définir la liste testée en dehors du `#@ assert`.

```
    l1 = [5, 4, 3, 2, 1]
    r = recherche_lin_present(l1, 3) #@ assert r == 2
    l2 = [-1, 0, 6, 4, 2]
    r = recherche_lin_present(l2, 4) #@ assert r == 3
    ...
```

— Écrire la boucle. On incrémentera une variable `pos` jusqu'à ce que la valeur de `A` en `pos` soit bien la valeur recherchée. Trouver l'invariant. On pourra voir le tableau comme étant découpé en deux parties à chaque passage de la boucle : celle précédant l'indice `pos` et celle succédant à `pos`. Vous pouvez vous aider de la figure 1.1.

1.12 Recherche linéaire du dernier élément dans un tableau

— Spécifier le sous-programme `recherche_lin` prenant en argument un tableau d'entiers A et un entier X et renvoyant dans *pos* la position du dernier X dans ce tableau s'il est présent, `-1` sinon.
— Trouver l'invariant et écrire la boucle. N'utilisez pas les tranches de tableaux, elles risquent de ne pas fonctionner pour la preuve.

1.13 Recherche d'un élément commun à deux tableaux

— Définir le prédicat `sorted(A: list[int])` indiquant si A est trié.
— Définir le prédicat `isIn(x : int, A : list[int])` indiquant si A contient x.
— Définir le prédicat `hasCommonSub(A : list[int], ad : int, af : int, B : list[int], bd : int, bf : int)` indiquant s'il y a au moins un élément qui apparait dans A entre les indices ad et af, ainsi que dans B entre les indices bd et bf.
— Écrire la spécification de la fonction `recherche_commun(A : list[int], B : list[int])` en utilisant les prédicats définis précédemment. Elle prend en paramètre deux tableaux d'entiers, ayant au moins un élément commun, et retourne le premier élément commun aux deux listes.
— Tester la spécification :

```
r = recherche_commun([3], [3])
#@assert r==3
r = recherche_commun([4,5,7,9,18,19], [3,6,8,9,10])
#@assert r==9
```

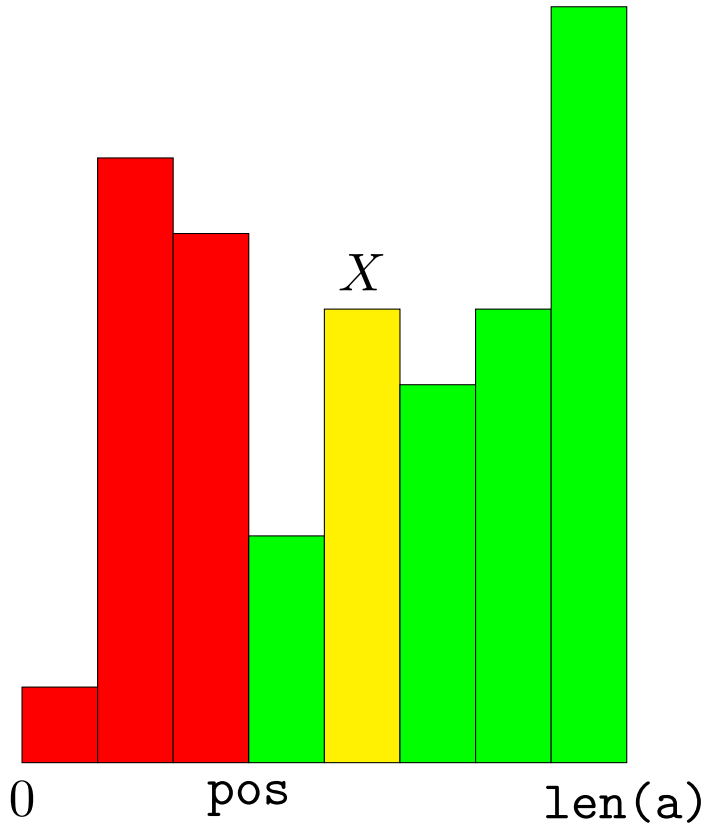


FIGURE 1.1 – Algorithme de recherche linéaire

```
5 ...
```

— Compléter le corps de la fonction avec deux indices qui parcourent les listes, une boucle while, un variant et des invariants. La contrainte ici est de réaliser la recherche dans un temps linéaire par rapport à la taille des listes. On exprimera dans l'invariant les faits suivants :
  — Chaque indice reste dans les bornes de sa liste.
  — Il y a un élément commun entre l'indice et la fin de la liste, pour les deux listes.
  — Aucun élément avant l'indice n'est présent dans l'autre liste, autrement dit, on n'a pas encore vu l'élément commun.

1.14 Recherche dichotomique d'un élément dans un tableau trié

— Définir le prédicat `sorted(A: list[int])` indiquant si A est trié.
— Définir le prédicat `isInSub(X,A,D,F)` indiquant si X apparaît dans le tableau A entre les indices D (compris) et F (non compris).
— En déduire la définition du prédicat `isIn(X,A)`
— Écrire la spécification du sous-programme de recherche prenant en argument un tableau trié A et un entier, et renvoyant dans `pos` l'indice de l'élément s'il est présent, `len(A)` sinon. La tester sur des exemples et contre-exemples simples.
— Déterminer un invariant en introduisant deux variables i et j telles que la présence de X dans le tableau soit équivalente à sa présence entre les indices i et j (j non compris). On peut s'aider de la figure 1.2.
— Écrire le code de la recherche dichotomique.

Ci-joint une illustration de la méthode par dichotomie. Méthode de dichotomie

1.15 Recherche simultanée ⋆

Maintenant, nous allons nous pencher sur la preuve d'un algorithme de recherche plus efficace dans un tableau non trié. C'est ce qu'on appelle la recherche simultanée. Étant donné un tableau d'entiers et un entier `c` représentant le nombre de flux de recherches parallèles, on parcourt tout le tableau en vérifiant un élément dans chaque secteur de notre tableau. Voir l'illustration 1.3.

— Reprendre les prédicats `isIn` et `isInSub` définis dans les exercices précédents.
— Voici le lemme dont on aura besoin pour effectuer la preuve de cet algorithme :

```
1 #@lemma NotisInSubNext: forall X, A, D, F. 0<=D<=F<len(A) and not isInSub(X,A,D,F) and
      not (A[F] == X) -> not isInSub(X,A,D,F+1)
```

Définissez ce lemme, mais sous forme d'une fonction qui ne retourne rien (c'est-à-dire de type `unit`). Elle prend en paramètres les quatre variables du lemme. En préconditions, reprenez les antécédents de l'implication. La postcondition sera quant à elle la conséquence de l'implication du lemme.

— Spécifier la fonction `recherche_simultanee(n : int, liste : list[int], c : int, k : int) -> int` de recherche parallèle. La fonction recherche `n` dans `liste`, en réalisant `c` recherches parallèles, portant chacune sur un secteur de `k` éléments. La fonction renvoie l'indice de l'élément trouvé s'il y en a un, sinon elle renvoie `len(liste)`. On supposera pour faciliter le traitement que `len(liste)` est multiple de `c` et de `k`.
— Écrire le corps de la fonction avec les deux boucles suivantes, en remplissant les points de suspension (sans toucher les assertions) :

```
1 i = ...
2 #@assert True
3 while ...:
4     s = ...
5     while ...:
```

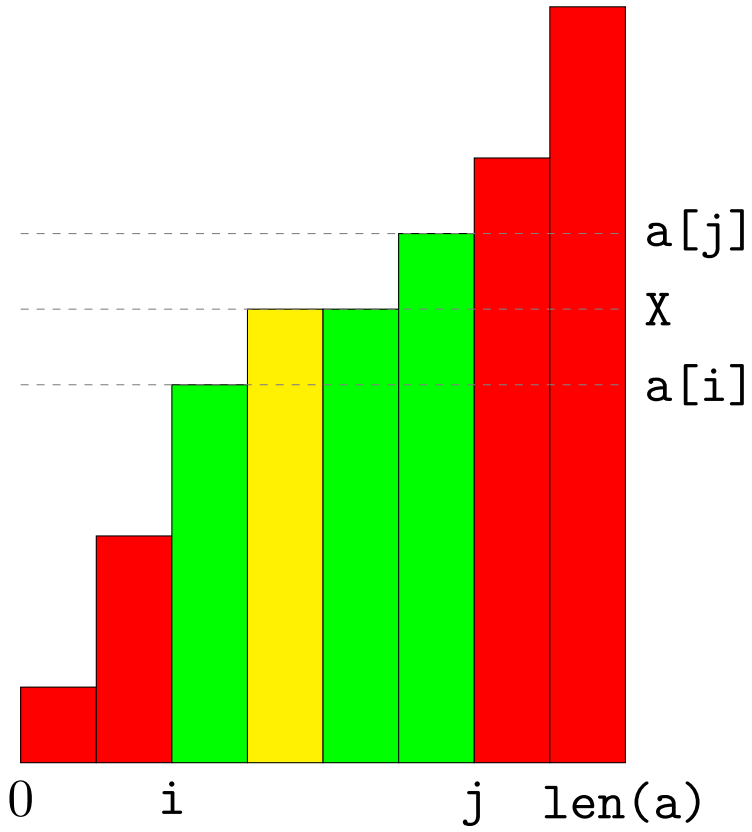
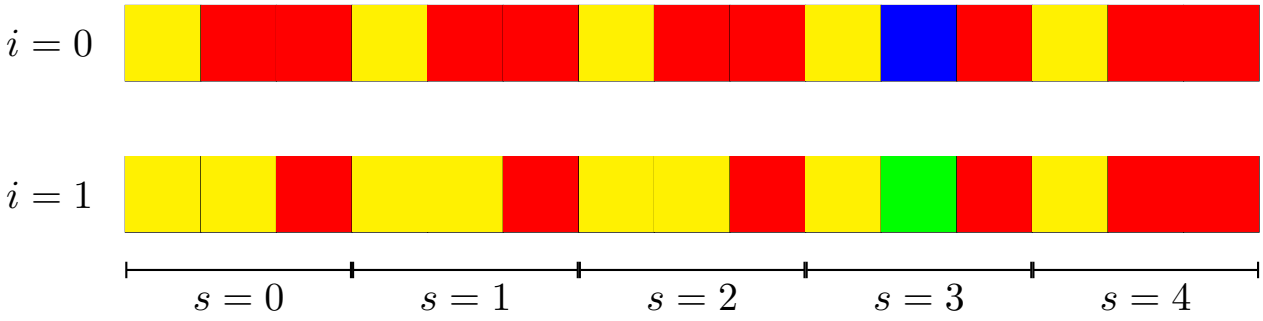


FIGURE 1.2 – Algorithme de dichotomie



Légende :

— Les éléments en rouge n'ont pas encore été visités
— On recherche l'élément colorié bleu
— Les éléments en jaune ont été visités
— L'élément en vert est l'élément recherché, une fois qu'il a été visité

FIGURE 1.3 – Recherche parallèle avec `c=5` secteurs de `k=3` éléments

```
6           #@assert True
7           if ...:
8               return ...
9           s = ...
10      i = ...
11  #@assert True
12  #@assert True
13  return ...
```

— Annoter les deux boucles avec les variants et les invariants. Astuce : pensez à la recherche linéaire, on peut représenter l'algorithme de recherche simultanée comme `c` recherches linéaires parallèles.
— Enfin, pour aider le prouveur, complétez vos annotations avec un appel au lemme en utilisant `#@call NotIsInSubNext(...)` sur les bons paramètres juste après le premier `return`, attention le `#@call` est en dehors de l'instruction conditionnelle.
— Le prouveur n'arrive toujours pas, afin de terminer la preuve, remplacez les `True` des assertions par les propositions logiques suivantes, en trouvant le bon ordre :

```
1  #@assert forall i. 0<=i<c*k -> (forall I. (i//k)*k<=I<((i//k)+1)*k and I==i -> liste[I
       ] != n)
2  #@assert s*k+k == (s+1)*k
3  #@assert forall S. 0<=S<c -> not isInSub(n, liste, S*k, S*k)
4  #@assert forall i. 0<=i<c*k -> (i//k)*k<=i<((i//k)+1)*k
```

1.16 Recherche d'un sous-tableau dans un tableau

— Définir le prédicat `subT(s,t,p)` indiquant si le tableau s se retrouve à la position p dans t.

```
1  #@predicate subT(s:list[int],t:list[int],p:int) = ...
```

— Définir en utilisant une boucle la fonction `sous_tableau(s,t,p)` renvoyant `True` si et seulement si `s` est un sous-tableau de `t` à la position `p`. Écrire la spécification de la fonction, la tester sur des exemples simples, puis écrire son corps.

```
1  s1 = [1, -1, 5]
2  t = [7, 1, -1, 5, 6, 7]
3  r = sous_tableau(s1, t, 1)
4  #@ assert r == True
5  s2 = [-1, 6, 5]
6  r = sous_tableau(s2, t, 2)
7  #@ assert r != True
```

— Écrire la spécification du sous-programme `recherche_derniere_position` prenant en argument 2 tableaux d'entiers `s` et `t` tels que `s` est un sous-tableau de `t`. Ce sous-programme renvoie la position de la dernière occurrence de `s` dans `t`. La spécification pourra utiliser `subT`. Tester cette spécification.
— Déterminer l'invariant de la boucle de recherche linéaire ainsi que la condition de boucle.
— Écrire le corps de la boucle. On utilisera une variable `pos` que l'on décrémentera après chaque passage de la boucle. On pourra réutiliser la fonction `sous_tableau`.

1.17 Somme maximale

— Définir la fonction récursive `sum(a,i)` calculant la somme des `i` premiers éléments de `a`. On retournera 0 si on n'a pas $0 \leq i \leq$ `len`(a).

```
1  #@function
2  def sum(a:list[int], i:int) -> int:
3    #@variant ...
4    return ... if 0 < i and i <= len(l) else 0
```

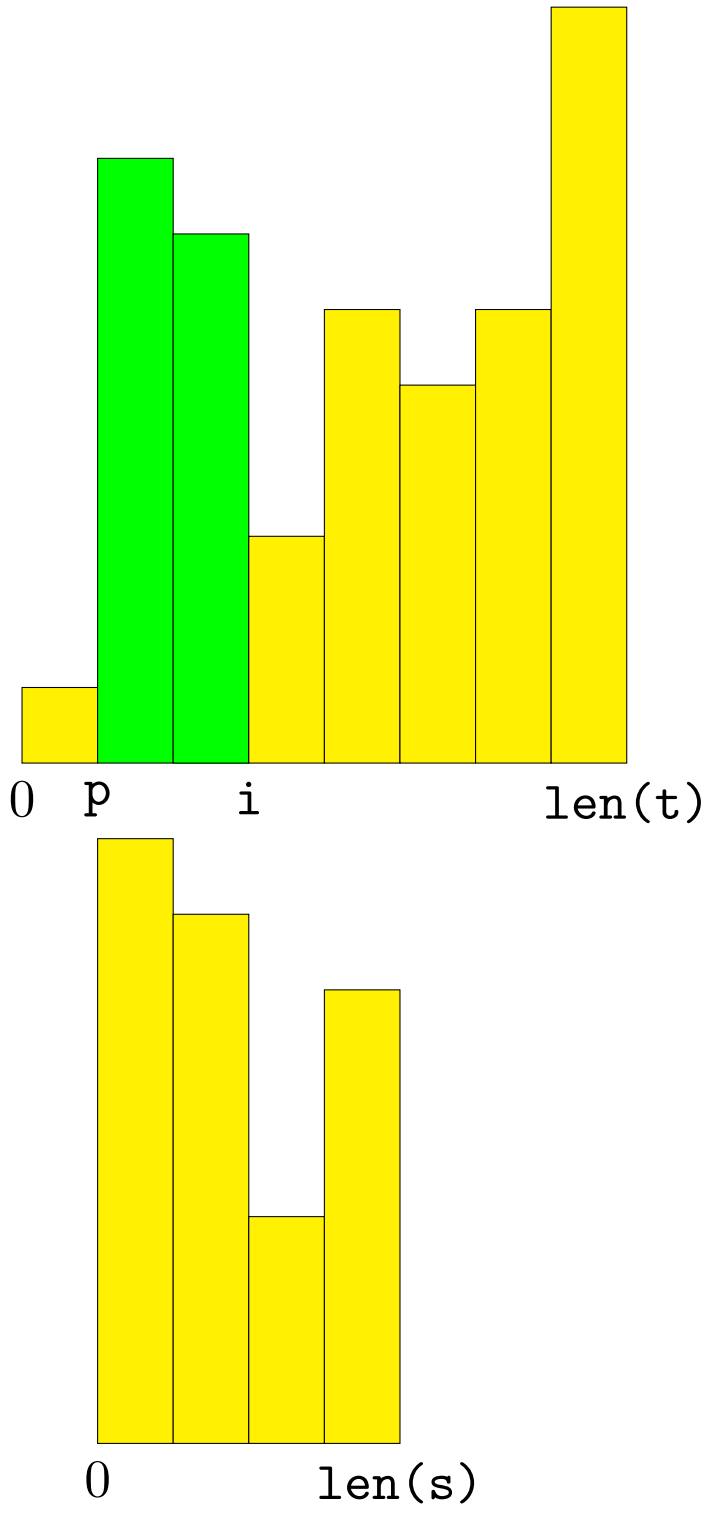


FIGURE 1.4 – Sous tableau

— Spécifier le sous-programme `pmax` prenant en argument une liste non vide d'entiers de premier élément 0 et renvoyant l'indice du plus grand élément qui est la somme de tous les éléments qui le précèdent dans le tableau. On utilisera `sum` dans les assertions. Puis tester cette spécification.
— Déterminer l'invariant et écrire la boucle. On utilisera `sum` pour exprimer l'invariant.

1.18 Liste palindrome

— Écrire la spécification du sous-programme `palindrome` prenant en argument un tableau non vide d'entiers `A` et retournant le premier indice qui témoigne que le tableau n'est pas un palindrome si c'est le cas, ou bien `len(A)` sinon. Tester la spécification sur quelques exemples.
— Écrire le corps de la fonction contenant une boucle while, avec son variant et son invariant.

1.19 Longueur du plus long plateau

— Écrire la spécification du sous-programme `plateau` prenant en argument un tableau trié non vide d'entiers et renvoyant dans `p` la longueur de son plus long plateau. La tester sur des exemples simples.
— En déduire l'invariant et écrire le code du sous-programme. On utilisera une variable `i` qu'on incrémentera à chaque passage de la boucle jusqu'à arriver à la fin du tableau ainsi qu'une variable `p` qui sera la longueur du plus grand plateau précédant l'indice `i`.

1.20 Inverser une liste

Cet exercice vous servira d'introduction dans la preuve des algorithmes qui modifient des tableaux. Il propose de spécifier et développer un algorithme simple qui inverse une liste passée en paramètres.

— Écrire la spécification du sous-programme qui, prenant un tableau non vide d'entiers en entrée, l'inverse et ne retourne rien à la fin de l'exécution, c'est-à-dire la fonction d'entête suivante :
`inverser(liste : list[int]) -> unit`.
Pour ce faire, vous aurez besoin de l'opérateur `old` (voir appendice 2.3.2) pour désigner le tableau reçu en entrée. Le tableau obtenu sera désigné comme le nom du paramètre, i.e. `liste`.

— Définir le prédicat `equals(a : list[int], b : list[int])` qui indique si les listes `a` et `b` sont bien les mêmes dans le sens logique (par opposition au sens physique) du terme.

— Avec plusieurs exemples, tester la spécification définie précédemment. Attention, ne définissez aucune liste dans à l'intérieur des annotations, utilisez plutôt des variables. La preuve de cette spécification est d'une grande complexité donc n'hésitez pas à utiliser l'`auto level 3` même avec des listes de petite taille.

— Développez le corps de la fonction qui satisfait la spécification. En utilisant deux indices, une à gauche et une à droite, initialisées comme sur la figure 1.5 elle parcourt le tableau en inversant deux éléments entre eux à l'aide d'une affectation multiple 1.6. La boucle termine lorsque l'indice de gauche plus un dépasse l'indice de droite, comme sur la figure 1.7. Cela permettra de traiter le cas de liste paire aussi bien que le cas de liste impaire.
— Introduire un `label start` avant la boucle. Pour pouvoir faire référence à la liste avant le début de l'exécution.
— Annoter la boucle avec le variant et les invariants. En ce qui concerne les invariants, vous pourrez utiliser l'opérateur `at` pour faire référence à un élément du tableau avant la boucle (voir 2.3.1). Pour faciliter la preuve, ajoutez un invariant qui dans une formule mettra en relation les indice `l` et `r` avec la taille de la liste `len(liste)`.

FIGURE 1.5 – Illustration de l'initialisation de l'inversion

FIGURE 1.6 – Illustration après 2 tours de boucle d'inversion

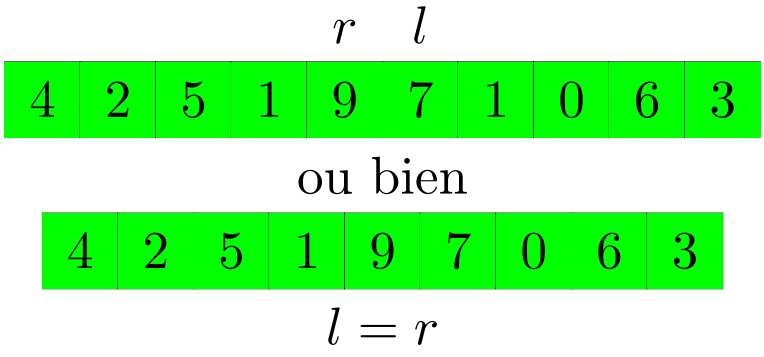


FIGURE 1.7 – Illustration de la fin de l'inversion pour liste paire et impaire respectivement

1.21 Le drapeau ⋆

On introduit 3 constantes représentant des couleurs. La présence du marqueur `#@constant` permet d'utiliser ces constantes dans les assertions et interdit toute modification des variables.

```
#@constant
BLEU = 1

#@constant
BLANC = 2

#@constant
ROUGE = 3
```

— Écrire la spécification du sous-programme de tri (nommé `drapeau`) d'un tableau de couleurs. On notera la post-condition exprimant que les valeurs contenues dans le tableau sont inchangées, c'est-à-dire que le tableau final est une permutation du tableau initial (désigné via le mot clé `old`, voir appendice 2.3.2). On utilisera pour ce faire la fonction `occurrence` qui compte pour un certain `v`, le nombre de `v` dans le tableau (voir appendice 2.4)

```
def drapeau(a:list[int]):
    #@requires forall i. 0<=i<len(a) -> BLEU <= a[i] <= ROUGE
    #@ensures forall v. occurrence(v, a, 0, len(a)) == old(occurrence(v, a, 0, len(a))
        )
    ...
    pass
```

— Il faudra introduire un label `start` avant la boucle afin de capturer l'état du programme avant la boucle. Voir l'appendice. 2.3

— Sachant que l'algorithme doit effectuer le tri en une passe, en déduire l'invariant puis le code de la boucle. On introduira 3 variables `i`, `j`, `k` auxiliaires pour délimiter 4 zones : les bleus, les blancs, les inconnus et les rouges. Vous pouvez vous aider de l'illustration1.8. On pourra écrire `a[i],a[j] = a[j],a[i]` pour échanger deux éléments de tableau. La preuve n'est pas automatique. On ajoutera le lemme suivant indiquant que la permutation de deux cases préserve les éléments du tableau :

```
#@lemma L2: forall i1, i2, a, v:int. 0 <= i1 < len(a) and 0 <= i2 < len(a) ->
    occurrence(v, a[i1<-a[i2]][i2<-a[i1]],0,len(a)) == occurrence(v,a,0,len(a))
```

En effet, cela signifie que pour toute valeur `v` du tableau (donc pour `BLEU`, `BLANC` et `ROUGE`) et pour tous `i1` et `i2` indices du tableau, le nombre d'occurrences de la valeur `v` dans le tableau `a` reste inchangé après permutation. Cela revient à dire que le tableau après permutation de deux éléments garde les mêmes éléments (seul l'ordre change). Le tableau `a[i1<-a[i2]][i2<-a[i1]]` désigne le tableau `a` après permutation des éléments en `i1` et en `i2`.

On aura aussi besoin d'effectuer une preuve par cas. Elle peut être générée automatiquement en ajoutant au code une instruction de la forme `if i == j: None`. Ce code ne fera rien du point de vue de l'interpréteur Python mais sera utile pour indiquer à Why3 de faire une preuve par cas.

On peut illustrer l'algorithme avec la vidéo suivante Drapeau

1.22 Séparation de valeurs spécifiques ⋆

Écrire la spécification du sous-programme `separer` prenant en argument 2 entiers différents `A` et `B` et un tableau d'entiers, et effectuant (un minimum) de permutations du tableau de sorte que les `A` viennent avant les `B`. Le tableau peut contenir n'importe quels entiers. Le sous-programme retournera un indice `i` séparant les `A` des `B` : les `B` `#@invariant forall` ne pourront apparaître qu'après `i` (ou en `i`). Tester cette spécification puis l'implémenter. Pour rendre la preuve automatique, on ajoutera une assertion avant de permuter les éléments du tableau :

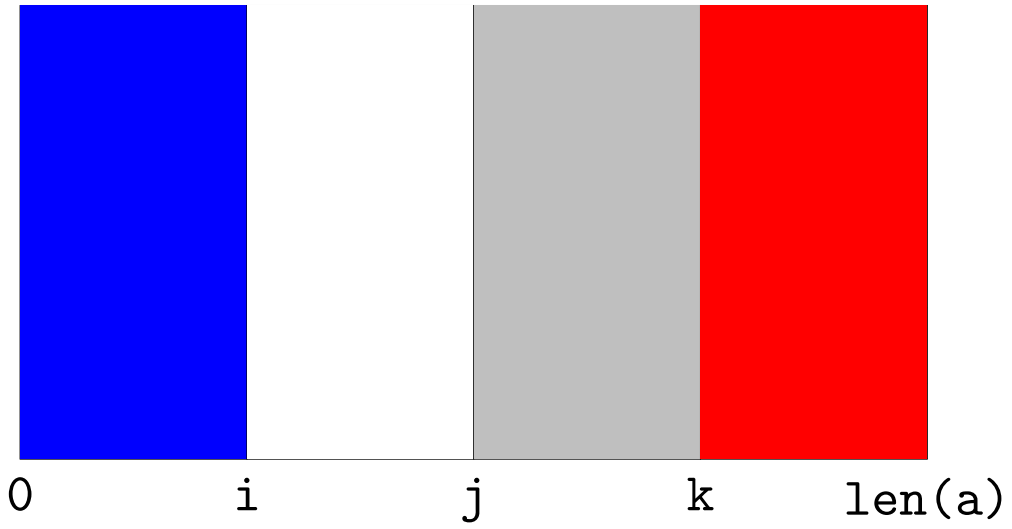


FIGURE 1.8 – Illustration de l'algorithme du drapeau

```
1     #@assert forall v. occurrence(v,a,0,len(a)) == occurrence(v,a[i<-a[j]][j<-a[i]],0,len(a)
          )
2     a[i],a[j] = a[j],a[i]
```

C'est-à-dire que le nombre d'occurrences pour chaque valeur `v` reste le même après avoir permuté `a[i]` et `a[j]`.

Afin de montrer que les éléments de la liste ne sont pas perdus lors de la transformation, on ajoutera l'invariant ci-dessous après avoir déclaré l'étiquette `start` en avant de la boucle.

```
1 #@invariant forall v. occurrence(v,a,0,len(a)) == at(occurrence(v,a,0,len(a)), start)
```

On peut illustrer cet algorithme par cette vidéo Algorithme de séparation

1.23 Tri bulle ⋆⋆

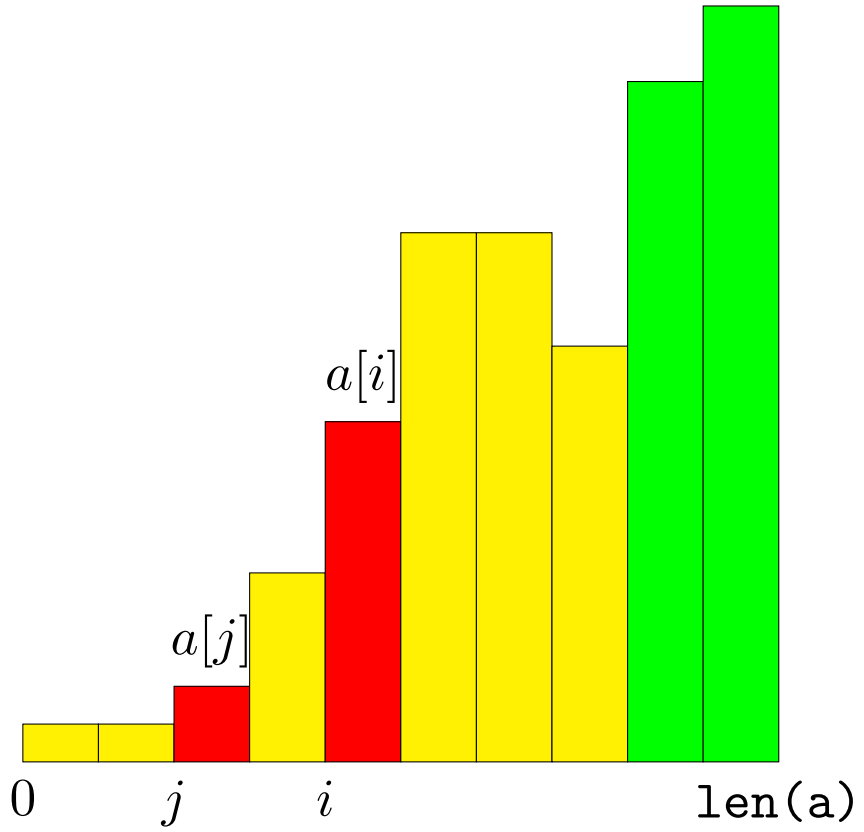
Cet exercice se penchera sur la vérification de l'algorithme du tri bulle. Il s'agit d'un algorithme de tri de tableau de complexité moyenne de l'ordre de $O(n^2)$, donc peu efficace face à d'autres algorithmes comme le tri rapide qui a une complexité de l'ordre de $O(n \log(n))$.

— Écrire la spécification de l'algorithme de tri bulle. On rappelle qu'il existe deux prédicats distincts, mais équivalents `trie1` et `trie2` pour exprimer qu'un tableau est trié. On utilisera pour la spécification le prédicat `trie2`. Exprimer le fait que le tableau trié doit être une permutation du tableau initial à l'aide de `occurrence`.

```
1 #@ predicate trie1(a:list[int], d:int, f:int) = d >= 0 and f <= len(a) and forall I. d
      <= I < f - 1 -> a[I] <= a[I + 1]
2 #@ predicate trie2(a:list[int], d:int, f:int) = d >= 0 and f <= len(a) and forall I, J
      . d <= I < J < f -> a[I] <= a[J]
```

— Montrer que ces deux prédicats sont équivalents. Pour ce faire, créer un lemme `L1` qui montre que `trie1` $\implies$ `trie2` ainsi qu'un lemme `L2` qui montre que `trie2` $\implies$ `trie1`. Vérifier ces deux lemmes. Pourquoi le lemme `L1` ne peut pas être montré automatiquement ?

— Écrire le lemme `L1` sous forme de fonction récursive qui s'appelle elle-même afin de simuler une récurrence ayant comme précondition le prédicat `trie1` et comme postcondition le prédicat `trie2`.



— Le tableau est trié de 0 à `i`
— Tous les éléments du tableau d'indice inférieur à `i` sont inférieurs à `a[i]`
— Tous les éléments du tableau d'indice supérieur à `i` sont supérieurs à `a[i]`
— Tous les éléments du tableau d'indice inférieur à `j` sont inférieurs ou égaux à `a[j]`

FIGURE 1.9 – Illustration de l'algorithme du tri bulle

— Écrire l'algorithme du tri bulle. On écrira deux boucles imbriquées, dont la boucle externe fera varier `i` de `len(a)-1` à 1 en décrémentant de 1 à chaque itération, et la boucle interne fera varier `j` de 0 à `i - 1`. On compare alors les deux cases consécutives d'indice `j` et `j + 1` et si elles ne sont pas dans le bon ordre, on les permute. On utilisera comme invariants pour la boucle externe que :
  — Les éléments de 0 à `i` sont bien triés en utilisant les deux prédicats `trie1` et `trie2`
  — Le tableau reste une permutation du tableau avant la boucle externe
  — Tous les éléments du tableau d'indice inférieur à `i` sont inférieurs à `a[i]`
  — Tous les éléments du tableau d'indice supérieur à `i` sont supérieurs à `a[i]`

  Et pour la boucle interne :
  — Les éléments de 0 à `i` sont bien triés en utilisant les deux prédicats `trie1` et `trie2`
  — Le tableau reste une permutation du tableau avant la boucle interne
  — Tous les éléments du tableau d'indice inférieur à `i` sont inférieurs à `a[i]`
  — Tous les éléments du tableau d'indice supérieur à `i` sont supérieurs à `a[i]`
  — Tous les éléments du tableau d'indice inférieur à `j` sont inférieurs ou égaux à `a[j]`

  On utilisera le lemme `L1` à l'aide de `#@ call` sur les bons paramètres en l'ajoutant avant la boucle interne et en fin de chaque itération. Il faudra en outre ajouter deux étiquettes `start1` et `start2` et l'opérateur `at` pour exprimer les invariants de permutation pour chaque boucle. Ajouter l'assertion suivante avant la permutation des éléments d'indice `j` et `j + 1` pour exprimer le fait que la permutation des deux éléments conserve l'invariant de permutation.

```
#@assert forall v. occurrence(v,a[j<-a[j+1]][j+1<-a[j]],0,len(a)) == at(occurrence(v,a
    ,0,len(a)), start2)
```

Ci-joint une illustration de l'algorithme Tri bulle

1.24 Tri décroissant par fusion ⋆⋆⋆

On ignorera ici la propriété de préservation de la liste par permutations.

1. Définir le prédicat `isSorted` prenant en argument une liste d'entiers et deux indices `d` et `f` et spécifiant que la tranche de tableau entre `d` (compris) et `f` (non compris) est triée par ordre décroissant.
2. Écrire la spécification du sous-programme `copie` prenant en argument un tableau `a:list[int]`, deux indices `d` et `f`, et un tableau `t` et réalisant l'affectation `a[d:f] = t`. La spécification devra comporter les propriétés suivantes sans utiliser de tranches de tableaux :
   — les dimensions sont cohérentes
   — les éléments `a[:d]` restent inchangés.
   — les éléments de `a[f:]` restent inchangés
   — `t` a été copié dans `a[d:f]`
3. Implanter le sous-programme à l'aide d'une boucle en suivant le schéma ci-dessous :

```
def copie(...):
  #@requires ...
  #@ensures ...
  #@label start
  i = 0
  while ... :
    #@invariant i est dans les bornes
    #@invariant a[d:i] a été copié
    #@invariant la tranche copiée est triée
    #@invariant a[:d] est inchangé / start
    #@invariant a[f:] est inchangé / start
    #@variant ...
     ...
```

4. Ajouter la preuve du fait qu'après la copie d'un tableau trié dans la tranche de tableau, cette tranche est triée :

```
def copie_sort(a:list[int],d:int,f:int,tmp:list[int]):
  #@requires 0<=d<=f<=len(a)
  #@requires len(tmp) == f-d
  #@requires isSorted(tmp,0,len(tmp))
  #@requires forall j.0<=j<f-d -> a[d+j] == tmp[j]
  #@ensures isSorted(a,d,f)
  if d < f:
    #@assert forall i,j. d<=i<=j<f and tmp[i-d] <= tmp[j-d] -> a[i] <= a[j]
  None
```

5. Écrire, sans utiliser de tranches de tableau, la spécification du sous-programme `merge` prenant en argument un tableau `a` et des indices $d \leq m \leq f$ et devant fusionner les deux sous-tableaux triés `a[d:f]` et `a[m:f]` dans la tranche `a[d:f]` tout en laissant inchangé le reste du tableau.
6. Implémenter le sous-programme `merge` en suivant le schéma ci-dessous :

```
def merge(a,d,m,f):
  #@requires ...
  #@ensures ...
  tmp = [0] * (f-d) # création d'un tableau auxiliaire pour la fusion
  i,d1,d2 = 0,d,m
  # on fusionne dans tmp
  while ... :
    #@invariant d <= d1 <= m <= d2 <=f
    #@invariant i == ...
    #@invariant les éléments de tmp[:i] sont >= à ceux de a[d1:m]
    #@invariant les éléments de tmp[:i] sont >= à ceux de a[d2:f]
    #@invariant tmp[:i] est trié
    if ...:
      tmp[i] = ...
    elif ...:
      tmp[i] = ...
  copie(a,d,f,tmp) # mise à jour de a
  #@call copie_sort(a,d,f,tmp)
```

7. Écrire la spécification du sous-programme `mergeSort(a,d,f)` qui doit trier la tranche `a[d:f]` sans modifier `a[:d]` ni `a[f:]`.
8. Implémenter le sous-programme `mergeSort`.

Ci-joint une illustration de l'algorithme de tri croissant Tri fusion

Chapitre 2

Appendice

2.1 Types de données

`bool`	true, false	
	not p	négation
	x == y	égalité
	x != y	différence
	p and q	conjonction
	p or q	disjonction
	`->`	implication
	`<->`	équivalence
	`if` ... `then` ... `else` ...	expression conditionnelle
	`forall` x,y. ...	quantification universelle
	`exists` x,y. ...	quantification existentielle
`int`	entiers relatifs	
	`x<=y<z`	comparaisons multiples
`list[T]`	**listes d'éléments de type T**	
	len(a)	longueur
	a[i]	ième élément ($0 \leq i < len(a)$)
	[v]*n	liste de n copies de v
	a[i :j],a[i :],a[:j],a[:]	sous liste (j non compris)
	a[i] = v	mise à jour d'une liste
	is_permutation(l1,l2)	si l1 et l2 sont les mêmes listes à permutation près
	occurrence(x, a)	nombre d'occurences d'un élément x dans une liste a

2.2 Règles WP

1. Règle de l'affectation simple

$$\frac{P \Rightarrow Q[x \leftarrow E]}{\{P\}x := E\{Q\}}$$

   Q est vérifiée si on substitue x par E dans Q

2. Règle de l'affectation simple

$$\frac{P \Rightarrow Q[x \leftarrow E]}{\{P\}x := E\{Q\}}$$

   Q est vérifiée si on substitue x par E dans Q

3. Règle de l'affectation multiple
$$\frac{P \Rightarrow Q[x \leftarrow E, y \leftarrow F...]}{\{P\}x, y... := E, F...\{Q\}}$$
Q est vérfiée si on substitue x par E, y par F, etc. dans Q

4. Règle de la séquence
$$\frac{\{P\}I\{R\} \qquad \{R\}J\{Q\}}{\{P\}I; J\{Q\}}$$
P implique un prédicat R avec exécution de I et R implique le prédicat Q avec exécution de J.

5. Règle de la condition
$$\frac{\{P \wedge C\}I\{Q\} \qquad \{P \wedge \neg C\}J\{Q\}}{\{P\}\texttt{if C then I else J}\{Q\}}$$
Si la condition C est vérifiée alors P implique Q avec exécution de I, et si elle n'est pas vérifiée alors P implique Q avec exécution de J.

6. Règle du while
$$\frac{\{I \wedge C\}B\{I\}}{\{I\}\texttt{while C do B}\{I \wedge \neg C\}}$$
On exécute le while tant que C est vérifiée. On introduit un invariant I qui est vérifiée avant de rentrer dans la boucle et qui se préserve à chaque itération. De plus, on introduit également un variant V qui est un entier strictement positif qui décroît strictement à chaque itération. Celui-ci permet de montrer la terminaison.

7. Règle de l'appel de fonction
$$\frac{\texttt{fP}(\texttt{fargs} \leftarrow \texttt{eargs}) \qquad \texttt{fQ}(\texttt{args} \leftarrow \texttt{eagrs}, \texttt{result} \leftarrow \texttt{res}) \rightarrow Q}{\{P\}\texttt{res = foo(eargs)}\{Q\}}$$
Lorsqu'on fait appel à une fonction, on substitue l'argument de la fonction par la valeur où on évalue la fonction. Ainsi il se doit de respecter la précondition fP de la fonction, dans ce cas on a donc qu'elle implique fQ. Pour montrer que P implique Q, il faut donc que P $\rightarrow$ fP $\rightarrow$ fQ $\rightarrow$ Q.

2.3 Labels

Un label (ou étiquette en français) permet d'enregistrer l'état de l'algorithme à un certain moment. On l'indique à l'aide de l'annotation `#@ label name`. Par exemple : `#@ label start` avant une boucle pour plus tard faire référence à la valeur qu'avait une variable avant la boucle.

2.3.1 Opérateur `at`

L'opérateur `at` permet d'évaluer une expression dans l'état associé au label à l'intérieur des annotations d'invariant. Il s'utilise en indiquant en paramètres, en premier, l'expression à évaluer et en second le nom du label : `at(E, name)`. Par exemple, `at(li[i], start)` permet de récupérer la valeur de l'élément d'indice `i` du tableau `li` telle qu'elle était à l'interprétation de la ligne avec le `label start`.

2.3.2 Opérateur `old`

L'opérateur `old` permet d'évaluer, dans les annotations de spécification, une expression comportant un paramètre mutable (comme un tableau) dans l'état dans lequel était la fonction avant exécution du corps de celle-ci. Il s'utilise en donnant en paramètre l'expression qu'on cherche à évaluer : `old(E)`. Par exemple, `old(li[i])` renverra la valeur initiale à l'indice `i` du tableau `li`, même si celle-ci a été modifiée lors de l'exécution de la fonction.

2.3.3 Exemple

Voici un exemple illustrant les trois concepts dans une seule fonction. Cette fonction prend une liste et incrémente de un tous ses éléments.

```
1  def increment(li : list[int]) -> unit:
2    #@ensures forall I. 0<=I<len(li) -> old(li[I])+1==li[I]
3    i = 0
4    #@label start
5    while i<len(li):
6      #@variant len(li)-i
7      #@invariant 0<=i<=len(li)
8      #@invariant forall I. 0<=I<i -> at(li[I], start)+1==li[I]
9      #@invariant forall I. i<=I<len(li) -> at(li[I], start)==li[I]
10     li[i] = li[i] + 1
11     i += 1
```

2.4 Opérateur Occurrence

L'opérateur `occurrence` compte le nombre d'occurrences d'une valeur `v` dans un tableau `a` entre deux bornes `c` et `d`. Il prend en paramètres en premier la valeur `v` et en second le tableau `a` ainsi que les deux bornes `c` et `d`. Il s'utilise donc comme il suit : `occurrence(v, a, c, d)`. Si on veut compter le nombre d'occurrences dans le tableau `a`, il suffit d'utiliser `occurrence(v, a, 0, len(a))`.

2.5 Opérateur Is_Permutation

L'opérateur `is_permutation` permet de déterminer si deux tableaux `a1` et `a2` sont des permutations l'un de l'autre. Autrement dit, ils ont les mêmes éléments, par exemple $is\_permutation([1,2,3],[2,3,1]) \equiv \top$ et $is\_permutation([1,3],[2,1]) \equiv \bot$.

2.6 Preuves de correction

2.6.1 Variant

Un variant permet de prouver qu'un algorithme termine (ne s'exécute pas indéfiniment). Il diminue strictement à chaque itération de la boucle ou à chaque appel récursif, et est positif tant que l'exécution se poursuit.

Un variant est soit un entier simple, soit un nuplet d'entiers ordonné selon l'ordre lexicographique $x_1, \ldots, x_n \prec y_1, \ldots, y_n$ si $(x_1 < y_1)$ ou $(x_1 = y_1 \wedge x_2 < y_2)$ ou $(x_1 = y_1 \wedge \ldots \wedge x_{n-1} = y_{n-1} \wedge x_n < y_n)$.
Par exemple :

- $1,3 \prec 2,3$, parce que $1 < 2$;
- $3,1,8 \prec 3,2,3$, parce que $(3 = 3) \wedge (1 < 2)$;
- $7,5,3 \prec 7,5,4$, parce que $(7 = 7) \wedge (5 = 5) \wedge (3 < 4)$.

Terminaison des fonctions récursives

Le variant lexicographique, est utilisé dans la définition récursive de la fonction d'Ackermann. La fonction d'Ackermann n'étant pas primitive récursive, il n'y a pas de variant simple (un entier).

```
1  #@function
2  def ack(m:int, n:int) -> int:
3      #@variant m,n   # ordre lexicographique
4      return n if m <= 0 else ack(m-1,1) if n<=0 else ack(m-1,ack(m,n-1))
```

Terminaison des boucles

La variable i dans la définition impérative de la fonction factorielle. Elle diminue avec chaque tour de boucle en restant positive.
Par exemple : avec l'indice $i = k > 0$ à une certaine itération, puis on décrémente : $i = i - 1$; et pour l'itération suivante i sera égal à $k - 1 < k$, et ainsi de suite.

2.6.2 Invariant

L'invariant de boucle permet d'appuyer la preuve que la boucle satisfait bien les post-conditions. Nous décrivons ci-dessous plusieurs stratégies utilisées pour formuler des invariants efficaces, avec des exemples pratiques pour chaque approche.

— **Invariant trivial :** Utilisation d'une vérité absolue ($\top$) qui reste invariante tout au long de la boucle
Exemple :

```
while (condition):
    ...
```

Invariant : $\top$ reste vrai indépendamment des opérations dans la boucle

— **Éliminer un conjoint de postcondition :** Diviser la postcondition en plusieurs parties et utiliser celles vérifiées après l'initialisation comme invariants
Exemple :

```
# Calcul de la racine carrée entière
n = 25
a = 0
while (a + 1)**2 <= n:
    a += 1
```

Invariant : $a^2 \leq n$ alors que la postcondition est $a \geq 0 \wedge a^2 \leq n < (a+1)^2$

— **Remplacer une sous-expression par une variable :** Remplacer une constante ou une sous-expression dans la postcondition par une variable mise à jour dans la boucle
Exemple :

```
# Trouver le maximum dans un tableau
max_val = float("-inf")
for i in range(len(array)):
    if array[i] > max_val:
        max_val = array[i]
```

Invariant : La variable *max_val* est le plus grand élément examiné jusqu'à présent

— **Combiner pré et postcondition :** Utiliser à la fois la précondition et la postcondition pour définir un invariant
Exemple :

```
# Séparation des éléments 'a' et 'b' dans un tableau
i, j = 0, len(array) - 1
while i < j:
    if array[i] == 'a':
        i += 1
    else:
        array[i], array[j] = array[j], array[i]
        j -= 1
```

Invariant : Les éléments avant i sont tous 'a' et ceux après j sont 'b'
Ci-joint la documentation officielle de Micro-Python : Documentation Micro-Python