

# Terminaison et correction

## Partie I : preuves par récurrence

Lycée Thiers

1 Terminaison et variants

2 Correction et invariants

3 Etude de cas : tri fusion

- 1 Terminaison et variants
- 2 Correction et invariants
- 3 Etude de cas : tri fusion

# Démontrer qu'une boucle ou une récursion termine

- Montrer la *terminaison* d'un programme, c'est prouver que le programme termine quel que soit l'état initial.

```

1 int power(int x, int n){
2   assert (n>=0);
3   int r = 1;
4   while (n>0)
5     {r*=x; n-=1;}
6   return r;
7 }
8

```

```

1 let power x n =
2   let rec pow n acc =
3     match n with
4       z when z < 0 -> failwith
5         "exposant négatif"
6       | 0 -> acc
7       | _ -> pow (n-1) (x*acc)
8   in pow n 1;;

```

# Démontrer qu'une boucle ou une récursion termine

- Montrer la *terminaison* d'un programme, c'est prouver que le programme termine quel que soit l'état initial.
- Le problème se pose principalement pour les boucles conditionnelles (`while`) et pour les fonctions récursives.

```

1 int power(int x, int n){
2   assert (n>=0);
3   int r = 1;
4   while (n>0)
5     {r*=x; n-=1;}
6   return r;
7 }
8

```

```

1 let power x n =
2   let rec pow n acc =
3     match n with
4       z when z < 0 -> failwith
5         "exposant négatif"
6       | 0 -> acc
7       | _ -> pow (n-1) (x*acc)
8   in pow n 1;;

```

# Notion de *variant*

## Définition

Soit un algorithme et une boucle de cet algorithme. Un *variant* (de cette boucle) est une fonction des variables de l'algorithme qui :

- ne prend que des valeurs entières positives ou nulles **lorsque la condition de boucle est vérifiée**,
- décroît strictement à chaque tour de boucle

## Définition

Soit un algorithme récursif. Un *variant* (de cette récursion) est une fonction des variables de l'algorithme qui :

- ne prend que des valeurs entières positives ou nulles **lorsque la condition de récursion est vérifiée**,
- décroît strictement à chaque appel récursif.

# Conséquence de l'existence d'un variant

- Il n'existe pas de suite strictement décroissante (infinie) d'entiers positifs.

# Conséquence de l'existence d'un variant

- Il n'existe pas de suite strictement décroissante (infinie) d'entiers positifs.
- En conséquence, si un variant de boucle (resp. de récursion) est identifié, la boucle (resp. la récursion) termine.

# Exemple

Considérons ces 2 codes d'exponentiation naïve.

```

1 int power(int x, int n){
2   assert (n>=0);
3   int r = 1;
4   while (n>0)
5     {r*=x; n-=1;}
6   return r;
7 }
8
```

```

1 let power x n =
2   let rec pow n acc =
3     match n with
4       z when z < 0-> failwith
5         "exposant négatif"
6       | 0 -> acc
7       | _ -> pow (n-1) (x*acc)
8   in pow n 1;;
```

- Pour le code en C, les variables sont  $x, n, r$ . Le variant est  $n$  (quantité entière strictement décroissante à chaque appel).
- Pour le code en OCaml, les variables de la récursion sont **acc**,  $n, x$ . Le variant est encore  $n$ .
- Ces deux programmes terminent.

# Remarque

```
1  int x = 3;  
2  while (true){  
3      x = x - 2;  
4  }  
5
```

- On est tenté de prendre  $x$  comme variant.

# Remarque

```
1 int x = 3;
2 while (true){
3     x = x - 2;
4 }
5
```

- On est tenté de prendre  $x$  comme variant.
- Mais il s'avère que  $x$  devient négatif au bout de quelques tours **sans que ce soit une condition de sortie de boucle**. Ce n'est donc pas un variant. Ouf! Car le programme ne s'arrête manifestement pas.

# Précaution

- Lorsque le candidat variant ne peut prendre que des valeurs positives (par exemple : longueur d'une liste), sa stricte décroissance suffit pour établir la terminaison.

# Précaution

- Lorsque le candidat variant ne peut prendre que des valeurs positives (par exemple : longueur d'une liste), sa stricte décroissance suffit pour établir la terminaison.
- En revanche, il faut faire attention s'il peut prendre des valeurs négatives. Dans ce cas, il faut vérifier que ces valeurs négatives sont bien capturées par la condition de sortie de boucle.

## Variante de récursion

Soit  $v$  le candidat variant.

- On établit que les cas de bases terminent et, d'une façon plus générale, que tout ce qui n'est pas appel récursif interne termine

## Variante de récursion

Soit  $v$  le candidat variant.

- On établit que les cas de bases terminent et, d'une façon plus générale, que tout ce qui n'est pas appel récursif interne termine
- On établit que, lors d'un appel interne, le variant est strictement plus petit que dans l'appel externe.

## Variant de récursion

Soit  $v$  le candidat variant.

- On établit que les cas de bases terminent et, d'une façon plus générale, que tout ce qui n'est pas appel récursif interne termine
- On établit que, lors d'un appel interne, le variant est strictement plus petit que dans l'appel externe.
- Si, par sa nature, le candidat variant ne peut prendre que des valeurs positives (par exemple : longueur d'une liste), les deux premiers points suffisent pour établir la terminaison.

## Variante de récursion

Soit  $v$  le candidat variant.

- On établit que les cas de bases terminent et, d'une façon plus générale, que tout ce qui n'est pas appel récursif interne termine
- On établit que, lors d'un appel interne, le variant est strictement plus petit que dans l'appel externe.
- Si, par sa nature, le candidat variant ne peut prendre que des valeurs positives (par exemple : longueur d'une liste), les deux premiers points suffisent pour établir la terminaison.
- En revanche, si il peut prendre des valeurs négatives, il faut s'assurer que les cas de bases capturent bien les valeurs négatives du variant (*i.e.* le cas  $v \leq 0$  est pris en compte par un des cas de base).

# Fusion de listes triées ↑

## Exercice

Ecrire un programme OCaml

`fusion : 'a list -> 'a list -> 'a list` qui prend en paramètres deux listes triées (par ordre croissant) et les fusionne en une seule liste triée. Établir la terminaison.

# Fusion de listes triées ↑

Solution.

Voici :

```
let rec fusion l1 l2 = match l1, l2 with
| [], l -> l
| l, [] -> l
| x1::t1, x2::t2 -> if x1<=x2
then x1 :: fusion t1 l2
else x2 :: fusion l1 t2 ;;
```



# Terminaison de fusion

- Cette fonction ne manipule pas directement des nombres entiers mais des listes.

## Terminaison de fusion

- Cette fonction ne manipule pas directement des nombres entiers mais des listes.
- On pourrait prendre comme variant la longueur de 11 (resp. 12). Malheureusement, cette longueur est certes décroissante mais pas toujours strictement.

# Terminaison de fusion

- Cette fonction ne manipule pas directement des nombres entiers mais des listes.
- On pourrait prendre comme variant la longueur de 11 (resp. 12). Malheureusement, cette longueur est certes décroissante mais pas toujours strictement.
- En revanche

# Terminaison de fusion

- Cette fonction ne manipule pas directement des nombres entiers mais des listes.
- On pourrait prendre comme variant la longueur de 11 (resp. 12). Malheureusement, cette longueur est certes décroissante mais pas toujours strictement.
- En revanche
  - Les 2 cas de base terminent et

# Terminaison de fusion

- Cette fonction ne manipule pas directement des nombres entiers mais des listes.
- On pourrait prendre comme variant la longueur de 11 (resp. 12). Malheureusement, cette longueur est certes décroissante mais pas toujours strictement.
- En revanche
  - Les 2 cas de base terminent et
  - la somme des longueurs des deux listes est strictement décroissante à chaque appel récursif : c'est un bon candidat à être variant.

# Terminaison de fusion

- Cette fonction ne manipule pas directement des nombres entiers mais des listes.
- On pourrait prendre comme variant la longueur de 11 (resp. 12). Malheureusement, cette longueur est certes décroissante mais pas toujours strictement.
- En revanche
  - Les 2 cas de base terminent et
  - la somme des longueurs des deux listes est strictement décroissante à chaque appel récursif : c'est un bon candidat à être variant.
  - Par sa nature (somme de longueurs de listes), le candidat variant ne peut prendre que des valeurs positives.

# Terminaison de fusion

- Cette fonction ne manipule pas directement des nombres entiers mais des listes.
- On pourrait prendre comme variant la longueur de 11 (resp. 12). Malheureusement, cette longueur est certes décroissante mais pas toujours strictement.
- En revanche
  - Les 2 cas de base terminent et
  - la somme des longueurs des deux listes est strictement décroissante à chaque appel récursif : c'est un bon candidat à être variant.
  - Par sa nature (somme de longueurs de listes), le candidat variant ne peut prendre que des valeurs positives.
- La somme des longueurs des listes est donc bien un variant. Le programme termine.

# Un mauvais exemple

Ou comment conclure un peu trop vite qu'une fonction termine

```
(*précondition : n > 0*)  
let rec f n = match n with  
  | 0 -> 0  
  | n -> f(n-2)
```

- Le cas de base termine

# Un mauvais exemple

Ou comment conclure un peu trop vite qu'une fonction termine

```
(*précondition : n > 0*)  
let rec f n = match n with  
  | 0 -> 0  
  | n -> f(n-2)
```

- Le cas de base termine
- On a bien que `f 0` termine.

# Un mauvais exemple

Ou comment conclure un peu trop vite qu'une fonction termine

```
(*précondition : n > 0*)
let rec f n = match n with
  | 0 -> 0
  | n -> f(n-2)
```

- Le cas de base termine
- On a bien que `f 0` termine.
- `n` semble un bon candidat variant. L'argument de l'appel interne respecte la règle de stricte décroissance :  $n - 2 < n$ .

# Un mauvais exemple

Ou comment conclure un peu trop vite qu'une fonction termine

```
(*précondition : n > 0*)
let rec f n = match n with
  | 0 -> 0
  | n -> f(n-2)
```

- Le cas de base termine
- On a bien que `f 0` termine.
- `n` semble un bon candidat variant. L'argument de l'appel interne respecte la règle de stricte décroissance :  $n - 2 < n$ .
- On conclut un peu vite que `f n` termine. Mais `f 1` nous prouve le contraire !

# Un mauvais exemple

Ou comment conclure un peu trop vite qu'une fonction termine

```
(*précondition : n > 0*)
let rec f n = match n with
  | 0 -> 0
  | n -> f(n-2)
```

- Le cas de base termine
- On a bien que `f 0` termine.
- `n` semble un bon candidat variant. L'argument de l'appel interne respecte la règle de stricte décroissance :  $n - 2 < n$ .
- On conclut un peu vite que `f n` termine. Mais `f 1` nous prouve le contraire !
- C'est parce que, en prenant  $n$  comme variant, on a oublié de vérifier si  $n < 0$  est pris en compte par les cas de base. Or, ce n'est pas traité.

# La factorielle

```
1 let factorielle n =  
2   let rec facto acc n = match n with  
3     | 0 -> acc  
4     | _ -> facto (acc*n) (n-1)  
5   in facto 1 n;;
```

- Il faut étudier la terminaison de la fonction auxiliaire.

# La factorielle

```
1 let factorielle n =  
2   let rec facto acc n = match n with  
3     | 0 -> acc  
4     | _ -> facto (acc*n) (n-1)  
5 in facto 1 n;
```

- Il faut étudier la terminaison de la fonction auxiliaire.
- Le candidat variant est  $n$ . C'est bien une quantité strictement décroissante.

# La factorielle

```
let factorielle n =  
  let rec facto acc n = match n with  
    | 0 -> acc  
    | _ -> facto (acc*n) (n-1)  
  in facto 1 n;;
```

- Il faut étudier la terminaison de la fonction auxiliaire.
- Le candidat variant est  $n$ . C'est bien une quantité strictement décroissante.
- Mais par sa nature (un entier),  $n$  peut prendre des valeurs négatives. Et celles-ci ne sont pas couvertes par les cas de base : il est clair que `facto` ne termine pas si  $n < -1$ . Encore une fois, la stricte décroissance ne suffit pas.

# La factorielle

Deux possibilités :

- On peut ajouter le filtrage

| `x when x < 0 -> failwith "cas <0"` . L'hypothèse  $n \leq 0$  est alors bien capturée par les cas de base.

# La factorielle

Deux possibilités :

- On peut ajouter le filtrage  
`| x when x < 0 -> failwith "cas < 0"`. L'hypothèse  $n \leq 0$  est alors bien capturée par les cas de base.
- On peut aussi ajouter la *précondition*  $n \geq 0$  avant l'appel à `factorielle`.

# La factorielle

Deux possibilités :

- On peut ajouter le filtrage  
`| x when x < 0 -> failwith "cas < 0"`. L'hypothèse  $n \leq 0$  est alors bien capturée par les cas de base.
- On peut aussi ajouter la *précondition*  $n \geq 0$  avant l'appel à `factorielle`.
  - On numérote les appels internes à partir de 0, et on note  $n_i$  la valeur du paramètre `n` lors de l'appel récursif  $n$  ( $n_0$  est la valeur de `n` lors de l'appel `factorielle n`). On trouve la récurrence  $n_{i+1} = n_i - 1$ .

# La factorielle

Deux possibilités :

- On peut ajouter le filtrage  
`| x when x < 0 -> failwith "cas < 0"`. L'hypothèse  $n \leq 0$  est alors bien capturée par les cas de base.
- On peut aussi ajouter la *précondition*  $n \geq 0$  avant l'appel à `factorielle`.
  - On numérote les appels internes à partir de 0, et on note  $n_i$  la valeur du paramètre `n` lors de l'appel récursif  $n$  ( $n_0$  est la valeur de `n` lors de l'appel `factorielle n`). On trouve la récurrence  $n_{i+1} = n_i - 1$ .
  - On en déduit que  $n_i = n_0 - i$ . Et donc  $n_{n_0} = n_0 - n_0 = 0$ . Comme 0 est bien capturé par le cas de base, la terminaison pour la precondition  $n \geq 0$  est établie.

# La factorielle

Deux possibilités :

- On peut ajouter le filtrage  
`| x when x < 0 -> failwith "cas < 0"`. L'hypothèse  $n \leq 0$  est alors bien capturée par les cas de base.
- On peut aussi ajouter la *précondition*  $n \geq 0$  avant l'appel à `factorielle`.
  - On numérote les appels internes à partir de 0, et on note  $n_i$  la valeur du paramètre `n` lors de l'appel récursif  $n$  ( $n_0$  est la valeur de `n` lors de l'appel `factorielle n`). On trouve la récurrence  $n_{i+1} = n_i - 1$ .
  - On en déduit que  $n_i = n_0 - i$ . Et donc  $n_{n_0} = n_0 - n_0 = 0$ . Comme 0 est bien capturé par le cas de base, la terminaison pour la precondition  $n \geq 0$  est établie.
  - Mais c'est quand même plus simple d'ajouter un cas de base !!

## Rédaction (cas d'un variant de boucle)

Quand le candidat variant  $v$  est une expression complexe imposant d'être très précis, on raisonne par récurrence pour montrer qu'il s'agit bien d'un variant :

- On numérote les tours de boucles à partir de 1 ; le tour 0 désignant ce qui se passe avant la boucle.

## Rédaction (cas d'un variant de boucle)

Quand le candidat variant  $v$  est une expression complexe imposant d'être très précis, on raisonne par récurrence pour montrer qu'il s'agit bien d'un variant :

- On numérote les tours de boucles à partir de 1 ; le tour 0 désignant ce qui se passe avant la boucle.
- Les variables et le candidat variant sont indicés par les tours de boucles :  $x_i$  désigne la valeur de la variable  $x$  à la fin du tour  $i$  ;  $x_0$  la valeur de  $x$  avant la boucle.

## Rédaction (cas d'un variant de boucle)

Quand le candidat variant  $v$  est une expression complexe imposant d'être très précis, on raisonne par récurrence pour montrer qu'il s'agit bien d'un variant :

- On numérote les tours de boucles à partir de 1 ; le tour 0 désignant ce qui se passe avant la boucle.
- Les variables et le candidat variant sont indicés par les tours de boucles :  $x_i$  désigne la valeur de la variable  $x$  à la fin du tour  $i$  ;  $x_0$  la valeur de  $x$  avant la boucle.
- Si le candidat variant  $v$  peut prendre des valeurs négatives, on montre que **si  $v_i$  est négatif au tour  $i$ , alors il n'y a pas de tour de boucle supplémentaire.**

## Rédaction (cas d'un variant de boucle)

Quand le candidat variant  $v$  est une expression complexe imposant d'être très précis, on raisonne par récurrence pour montrer qu'il s'agit bien d'un variant :

- On numérote les tours de boucles à partir de 1 ; le tour 0 désignant ce qui se passe avant la boucle.
- Les variables et le candidat variant sont indicés par les tours de boucles :  $x_i$  désigne la valeur de la variable  $x$  à la fin du tour  $i$  ;  $x_0$  la valeur de  $x$  avant la boucle.
- Si le candidat variant  $v$  peut prendre des valeurs négatives, on montre que si  $v_i$  est négatif au tour  $i$ , alors il n'y a pas de tour de boucle supplémentaire.
- Pour  $i \in \mathbb{N}$ , on suppose qu'il y a un tour  $i + 1$  (s'il n'y en a pas, le programme s'arrête et on est content). **Il s'agit alors de prouver que  $v_i > v_{i+1}$ .** Remarque : une conséquence de l'hypothèse est que  $v_i \geq 0$ .

## Rédaction (cas d'un variant de boucle)

Quand le candidat variant  $v$  est une expression complexe imposant d'être très précis, on raisonne par récurrence pour montrer qu'il s'agit bien d'un variant :

- On numérote les tours de boucles à partir de 1 ; le tour 0 désignant ce qui se passe avant la boucle.
- Les variables et le candidat variant sont indicés par les tours de boucles :  $x_i$  désigne la valeur de la variable  $x$  à la fin du tour  $i$  ;  $x_0$  la valeur de  $x$  avant la boucle.
- Si le candidat variant  $v$  peut prendre des valeurs négatives, on montre que si  $v_i$  est négatif au tour  $i$ , alors il n'y a pas de tour de boucle supplémentaire.
- Pour  $i \in \mathbb{N}$ , on suppose qu'il y a un tour  $i + 1$  (s'il n'y en a pas, le programme s'arrête et on est content). Il s'agit alors de prouver que  $v_i > v_{i+1}$ . Remarque : une conséquence de l'hypothèse est que  $v_i \geq 0$ .
- Si la décroissance stricte est prouvée, le candidat variant est bien un variant. On en déduit la terminaison du programme.

# Recherche dichotomique dans une liste triée

L'idée centrale de cette approche repose sur l'idée de réduire de moitié l'espace de recherche à chaque étape : on regarde la valeur du milieu et si ce n'est pas celle recherchée, on sait qu'il faut continuer de chercher dans la première moitié ou dans la seconde.

# Recherche dichotomique dans une liste triée

## Exercice

Écrire en C une fonction impérative

`bool dichot(int n, int t[], int x)` qui indique par un booléen si  $x$  est présent dans le tableau trié (par ordre croissant)  $t$ . Établir la terminaison.

# Recherche dichotomique dans une liste triée ↑

Sémantique : `g` est l'indice à partir duquel on cherche, `d` celui à partir duquel on ne cherche plus.

```

1 bool dichot(int n, int t[n], int x){
2     int g = 0, d = n; // on cherche x entre g et d-1
3     while(g<d){// hypothèses : n>=0, |t| = n, t trié
4         int m = (g+d)/2;// milieu
5         if (x == t[m]) return true;
6         else if (x < t[m]) d=m;
7         else g = m+1;
8     }// while
9     return false; }//dichot
10

```

# Terminaison de dichot

- Les variables sont  $x, g, d, m, t$  .

# Terminaison de dichot

- Les variables sont  $x, g, d, m, t$ .
- $x$  (resp.  $t$ ) ne varie pas selon les tours de boucle,  $g$  est croissante dans la boucle,  $m$  n'est pas monotone dans la boucle. Ces quantités ne peuvent donc être des variants de boucles.

# Terminaison de `dichot`

- Les variables sont `x, g, d, m, t` .
- `x` (resp. `t` ) ne varie pas selon les tours de boucle, `g` est croissante dans la boucle, `m` n'est pas monotone dans la boucle. Ces quantités ne peuvent donc être des variants de boucles.
- `d` est bien décroissante à chaque tour de boucle mais pas strictement décroissante : `d` ne peut pas être un variant.

# Terminaison de `dichot`

- Les variables sont `x, g, d, m, t`.
- `x` (resp. `t`) ne varie pas selon les tours de boucle, `g` est croissante dans la boucle, `m` n'est pas monotone dans la boucle. Ces quantités ne peuvent donc être des variants de boucles.
- `d` est bien décroissante à chaque tour de boucle mais pas strictement décroissante : `d` ne peut pas être un variant.
- Notons  $d_i, g_i$  les valeurs respectives de `d, g` à la fin de chaque tour de boucle  $i \geq 0$ . ( $i = 0$  signifie « avant la boucle »).

# Terminaison de `dichot`

- Les variables sont `x, g, d, m, t`.
- `x` (resp. `t`) ne varie pas selon les tours de boucle, `g` est croissante dans la boucle, `m` n'est pas monotone dans la boucle. Ces quantités ne peuvent donc être des variants de boucles.
- `d` est bien décroissante à chaque tour de boucle mais pas strictement décroissante : `d` ne peut pas être un variant.
- Notons  $d_i, g_i$  les valeurs respectives de `d, g` à la fin de chaque tour de boucle  $i \geq 0$ . ( $i = 0$  signifie « avant la boucle »).
- Si  $d_i - g_i \leq 0$  (donc  $d_i \leq g_i$ ) : il n'y a pas de tour de boucle supplémentaire. De plus,  $d_i - g_i$  est manifestement toujours entier. Ainsi,  $d_i - g_i$  est un bon candidat variant.

# Terminaison de dichot

- Soit  $i \in \mathbb{N}$ . S'il y a un tour  $i + 1$ , alors  $d_i > g_i$  donc  $d_i - g_i > 0$ . On a 
$$m_{i+1} = \lfloor \frac{d_i + g_i}{2} \rfloor$$

# Terminaison de dichot

- Soit  $i \in \mathbb{N}$ . S'il y a un tour  $i + 1$ , alors  $d_i > g_i$  donc  $d_i - g_i > 0$ . On a 
$$m_{i+1} = \lfloor \frac{d_i + g_i}{2} \rfloor$$
  - si  $t_{m_{i+1}} = x$ , alors le programme s'arrête : on est content.

# Terminaison de dichot

- Soit  $i \in \mathbb{N}$ . S'il y a un tour  $i + 1$ , alors  $d_i > g_i$  donc  $d_i - g_i > 0$ . On a 
$$m_{i+1} = \lfloor \frac{d_i + g_i}{2} \rfloor$$
  - si  $t_{m_{i+1}} = x$ , alors le programme s'arrête : on est content.
  - Sinon, comme  $d_i - g_i > 0$ , on a que  $g_i \leq m_{i+1} < d_i$ .

# Terminaison de dichot

- Soit  $i \in \mathbb{N}$ . S'il y a un tour  $i + 1$ , alors  $d_i > g_i$  donc  $d_i - g_i > 0$ . On a 
$$m_{i+1} = \lfloor \frac{d_i + g_i}{2} \rfloor$$
  - si  $t_{m_{i+1}} = x$ , alors le programme s'arrête : on est content.
  - Sinon, comme  $d_i - g_i > 0$ , on a que  $g_i \leq m_{i+1} < d_i$ .
    - Si  $d_{i+1} = m_{i+1}$ , alors  $g_{i+1} = g_i$  et  $d_{i+1} - g_{i+1} = m_{i+1} - g_i < d_i - g_i$  (décroissance stricte)

# Terminaison de dichot

- Soit  $i \in \mathbb{N}$ . S'il y a un tour  $i + 1$ , alors  $d_i > g_i$  donc  $d_i - g_i > 0$ . On a 
$$m_{i+1} = \lfloor \frac{d_i + g_i}{2} \rfloor$$
  - si  $t_{m_{i+1}} = x$ , alors le programme s'arrête : on est content.
  - Sinon, comme  $d_i - g_i > 0$ , on a que  $g_i \leq m_{i+1} < d_i$ .
    - Si  $d_{i+1} = m_{i+1}$ , alors  $g_{i+1} = g_i$  et  $d_{i+1} - g_{i+1} = m_{i+1} - g_i < d_i - g_i$  (décroissance stricte)
    - Si  $g_{i+1} = m_{i+1} + 1$ , alors  $d_{i+1} = d_i$  et  $d_{i+1} - g_{i+1} = d_i - m_{i+1} - 1 < d_i - m_{i+1} \leq d_i - g_i$  (décroissance stricte)

# Terminaison de dichot

- Soit  $i \in \mathbb{N}$ . S'il y a un tour  $i + 1$ , alors  $d_i > g_i$  donc  $d_i - g_i > 0$ . On a 
$$m_{i+1} = \lfloor \frac{d_i + g_i}{2} \rfloor$$
  - si  $t_{m_{i+1}} = x$ , alors le programme s'arrête : on est content.
  - Sinon, comme  $d_i - g_i > 0$ , on a que  $g_i \leq m_{i+1} < d_i$ .
    - Si  $d_{i+1} = m_{i+1}$ , alors  $g_{i+1} = g_i$  et  $d_{i+1} - g_{i+1} = m_{i+1} - g_i < d_i - g_i$  (décroissance stricte)
    - Si  $g_{i+1} = m_{i+1} + 1$ , alors  $d_{i+1} = d_i$  et  $d_{i+1} - g_{i+1} = d_i - m_{i+1} - 1 < d_i - m_{i+1} \leq d_i - g_i$  (décroissance stricte)
    - Dans tous les cas de figure la décroissance est stricte.

# Terminaison de dichot

- Soit  $i \in \mathbb{N}$ . S'il y a un tour  $i + 1$ , alors  $d_i > g_i$  donc  $d_i - g_i > 0$ . On a 
$$m_{i+1} = \lfloor \frac{d_i + g_i}{2} \rfloor$$
  - si  $t_{m_{i+1}} = x$ , alors le programme s'arrête : on est content.
  - Sinon, comme  $d_i - g_i > 0$ , on a que  $g_i \leq m_{i+1} < d_i$ .
    - Si  $d_{i+1} = m_{i+1}$ , alors  $g_{i+1} = g_i$  et  $d_{i+1} - g_{i+1} = m_{i+1} - g_i < d_i - g_i$  (décroissance stricte)
    - Si  $g_{i+1} = m_{i+1} + 1$ , alors  $d_{i+1} = d_i$  et  $d_{i+1} - g_{i+1} = d_i - m_{i+1} - 1 < d_i - m_{i+1} \leq d_i - g_i$  (décroissance stricte)
    - Dans tous les cas de figure la décroissance est stricte.
  - Ainsi  $d_i - v_i$  est un variant. L'algorithme s'arrête à un moment.

- 1 Terminaison et variants
- 2 Correction et invariants**
- 3 Etude de cas : tri fusion

# Spécification d'un algorithme

- La *spécification* d'un problème algorithmique est une description des attendus en deux parties :

# Spécification d'un algorithme

- La *spécification* d'un problème algorithmique est une description des attendus en deux parties :
  - la description des entrées admissibles, et

# Spécification d'un algorithme

- La *spécification* d'un problème algorithmique est une description des attendus en deux parties :
  - la description des entrées admissibles, et
  - la description du résultat et/ou des effets attendus.

# Spécification d'un algorithme

- La *spécification* d'un problème algorithmique est une description des attendus en deux parties :
  - la description des entrées admissibles, et
  - la description du résultat et/ou des effets attendus.
- On décrit les entrées admissibles par des contraintes appelées *préconditions*. Exemple :

# Spécification d'un algorithme

- La *spécification* d'un problème algorithmique est une description des attendus en deux parties :
  - la description des entrées admissibles, et
  - la description du résultat et/ou des effets attendus.
- On décrit les entrées admissibles par des contraintes appelées *préconditions*. Exemple :
  - les entrées sont des entiers positifs

# Spécification d'un algorithme

- La *spécification* d'un problème algorithmique est une description des attendus en deux parties :
  - la description des entrées admissibles, et
  - la description du résultat et/ou des effets attendus.
- On décrit les entrées admissibles par des contraintes appelées *préconditions*. Exemple :
  - les entrées sont des entiers positifs
  - les entrées sont un tableau de flottants trié par ordre croissant et un flottant ;

# Spécification d'un algorithme

- La *spécification* d'un problème algorithmique est une description des attendus en deux parties :
  - la description des entrées admissibles, et
  - la description du résultat et/ou des effets attendus.
- On décrit les entrées admissibles par des contraintes appelées *préconditions*. Exemple :
  - les entrées sont des entiers positifs
  - les entrées sont un tableau de flottants trié par ordre croissant et un flottant ;
  - Les entrées sont deux listes sans intersection...

# Spécification d'un algorithme

- La *spécification* d'un problème algorithmique est une description des attendus en deux parties :
  - la description des entrées admissibles, et
  - la description du résultat et/ou des effets attendus.
- On décrit les entrées admissibles par des contraintes appelées *préconditions*. Exemple :
  - les entrées sont des entiers positifs
  - les entrées sont un tableau de flottants trié par ordre croissant et un flottant ;
  - Les entrées sont deux listes sans intersection...
- La *spécification d'un algorithme* est la spécification du problème résolu par cet algorithme.

# Spécification d'un algorithme

- La *spécification* d'un problème algorithmique est une description des attendus en deux parties :
  - la description des entrées admissibles, et
  - la description du résultat et/ou des effets attendus.
- On décrit les entrées admissibles par des contraintes appelées *préconditions*. Exemple :
  - les entrées sont des entiers positifs
  - les entrées sont un tableau de flottants trié par ordre croissant et un flottant ;
  - Les entrées sont deux listes sans intersection...
- La *spécification d'un algorithme* est la spécification du problème résolu par cet algorithme.
- Exemple : Les questions de code dans un DS d'informatique sont des spécifications de l'algorithme attendu.

# Usages de la spécification

- En phase de conception, la spécification décrit ce qui doit être réalisé.

# Usages de la spécification

- En phase de conception, la spécification décrit ce qui doit être réalisé.
- En phase d'utilisation, la spécification indique à l'utilisateur ce qu'il doit entrer et le résultat auquel il peut s'attendre.

# Usages de la spécification

- En phase de conception, la spécification décrit ce qui doit être réalisé.
- En phase d'utilisation, la spécification indique à l'utilisateur ce qu'il doit entrer et le résultat auquel il peut s'attendre.
- En phase de test, la spécification donne les critères pour se prononcer sur la réussite ou l'échec d'un test.

# Usages de la spécification

- En phase de conception, la spécification décrit ce qui doit être réalisé.
- En phase d'utilisation, la spécification indique à l'utilisateur ce qu'il doit entrer et le résultat auquel il peut s'attendre.
- En phase de test, la spécification donne les critères pour se prononcer sur la réussite ou l'échec d'un test.
- En phase de preuve de correction, la spécification énonce ce qui doit être démontré.

# Invariant de boucle

- Objectif : Déterminer si un programme contenant une boucle est correct vis à vis de sa *spécification* (c.a.d s'il fait bien ce qu'on attend de lui).

# Invariant de boucle

- Objectif : Déterminer si un programme contenant une boucle est correct vis à vis de sa *spécification* (c.a.d s'il fait bien ce qu'on attend de lui).
- Moyen : On utilise un *invariant* de boucle, c'est-à-dire une propriété :

# Invariant de boucle

- Objectif : Déterminer si un programme contenant une boucle est correct vis à vis de sa *spécification* (c.a.d s'il fait bien ce qu'on attend de lui).
- Moyen : On utilise un *invariant* de boucle, c'est-à-dire une propriété :
  - 1 qui est vérifiée avant d'entrer dans la boucle.

# Invariant de boucle

- Objectif : Déterminer si un programme contenant une boucle est correct vis à vis de sa *spécification* (c.a.d s'il fait bien ce qu'on attend de lui).
- Moyen : On utilise un *invariant* de boucle, c'est-à-dire une propriété :
  - 1 qui est vérifiée avant d'entrer dans la boucle.
  - 2 qui si elle est vérifiée avant une itération est vérifiée après celle-ci,

# Invariant de boucle

- Objectif : Déterminer si un programme contenant une boucle est correct vis à vis de sa *spécification* (c.a.d s'il fait bien ce qu'on attend de lui).
- Moyen : On utilise un *invariant* de boucle, c'est-à-dire une propriété :
  - ① qui est vérifiée avant d'entrer dans la boucle.
  - ② qui si elle est vérifiée avant une itération est vérifiée après celle-ci,
- Un invariant de boucle pas très utile est par exemple `1==1` . Il s'applique à toutes les boucles ! C'est pour cette raison qu'on ajoute souvent une troisième contrainte :

# Invariant de boucle

- Objectif : Déterminer si un programme contenant une boucle est correct vis à vis de sa *spécification* (c.a.d s'il fait bien ce qu'on attend de lui).
- Moyen : On utilise un *invariant* de boucle, c'est-à-dire une propriété :
  - ① qui est vérifiée avant d'entrer dans la boucle.
  - ② qui si elle est vérifiée avant une itération est vérifiée après celle-ci,
- Un invariant de boucle pas très utile est par exemple `1==1` . Il s'applique à toutes les boucles ! C'est pour cette raison qu'on ajoute souvent une troisième contrainte :
  - ③ lorsque la propriété est vérifiée en sortie de boucle, elle permet de déduire que le programme est correct (ou du moins la partie du programme délimitée par la boucle).

# Calcul de $2^n$

Montrer que ce programme de calcul de  $2^n$  est correct pour tout entier  $n \geq 0$ , c.a.d qu'en sortie de boucle `p` contient  $2^n$ .

```
1 int p=1, c = n; // précondition : n entier >=0
2 while (c > 0){
3     p = p * 2;
4     c = c - 1; }
5
```

# Calcul de $2^n$

On note  $c_i, p_i$  les contenus des variables `c` et `p` après l'itération  $i$ .

- Avant l'entrée dans la boucle :  $c_0 = n, p_0 = 1$ .

# Calcul de $2^n$

On note  $c_i, p_i$  les contenus des variables **c** et **p** après l'itération  $i$ .

- Avant l'entrée dans la boucle :  $c_0 = n, p_0 = 1$ .
- Après l'itération  $i$  :  $c_{i+1} = c_i - 1$  et  $p_{i+1} = 2p_i$ .

# Calcul de $2^n$

On note  $c_i, p_i$  les contenus des variables **c** et **p** après l'itération  $i$ .

- Avant l'entrée dans la boucle :  $c_0 = n, p_0 = 1$ .
- Après l'itération  $i$  :  $c_{i+1} = c_i - 1$  et  $p_{i+1} = 2p_i$ .
- Invariant potentiel **Inv**( $i$ ) :  $c_i \geq 0$  et  $p_i = 2^{n-c_i}$ .

# Calcul de $2^n$

On note  $c_i, p_i$  les contenus des variables **c** et **p** après l'itération  $i$ .

- Avant l'entrée dans la boucle :  $c_0 = n, p_0 = 1$ .
- Après l'itération  $i$  :  $c_{i+1} = c_i - 1$  et  $p_{i+1} = 2p_i$ .
- Invariant potentiel **Inv**( $i$ ) :  $c_i \geq 0$  et  $p_i = 2^{n-c_i}$ .
- Cas de base, itération 0 (avant l'entrée dans la boucle) :  $c_0 = n \geq 0, p_0 = 1 = 2^0 = 2^{n-c_0}$  : OK.

# Calcul de $2^n$

On note  $c_i, p_i$  les contenus des variables **c** et **p** après l'itération  $i$ .

- Avant l'entrée dans la boucle :  $c_0 = n, p_0 = 1$ .
- Après l'itération  $i$  :  $c_{i+1} = c_i - 1$  et  $p_{i+1} = 2p_i$ .
- Invariant potentiel **Inv**( $i$ ) :  $c_i \geq 0$  et  $p_i = 2^{n-c_i}$ .
- Cas de base, itération 0 (avant l'entrée dans la boucle) :  $c_0 = n \geq 0$ ,  $p_0 = 1 = 2^0 = 2^{n-c_0}$  : OK.
- Hérité. On suppose **Inv**( $i$ ) vérifié.

# Calcul de $2^n$

On note  $c_i, p_i$  les contenus des variables **c** et **p** après l'itération  $i$ .

- Avant l'entrée dans la boucle :  $c_0 = n, p_0 = 1$ .
- Après l'itération  $i$  :  $c_{i+1} = c_i - 1$  et  $p_{i+1} = 2p_i$ .
- Invariant potentiel **Inv**( $i$ ) :  $c_i \geq 0$  et  $p_i = 2^{n-c_i}$ .
- Cas de base, itération 0 (avant l'entrée dans la boucle) :  $c_0 = n \geq 0, p_0 = 1 = 2^0 = 2^{n-c_0}$  : OK.
- Hérité. On suppose **Inv**( $i$ ) vérifié.
  - 1 Si on entre dans la boucle, alors  $c_i > 0$  et  $c_{i+1} = c_i - 1 \geq 0$ . De plus  $p_{i+1} = 2p_i = 2^{n-c_i+1} = 2^{n-(c_i-1)} = 2^{n-c_{i+1}}$ . Donc **Inv**( $i+1$ ) vérifié.

# Calcul de $2^n$

On note  $c_i, p_i$  les contenus des variables **c** et **p** après l'itération  $i$ .

- Avant l'entrée dans la boucle :  $c_0 = n, p_0 = 1$ .
- Après l'itération  $i$  :  $c_{i+1} = c_i - 1$  et  $p_{i+1} = 2p_i$ .
- Invariant potentiel **Inv**( $i$ ) :  $c_i \geq 0$  et  $p_i = 2^{n-c_i}$ .
- Cas de base, itération 0 (avant l'entrée dans la boucle) :  $c_0 = n \geq 0, p_0 = 1 = 2^0 = 2^{n-c_0}$  : OK.
- Hérité. On suppose **Inv**( $i$ ) vérifié.
  - ① Si on entre dans la boucle, alors  $c_i > 0$  et  $c_{i+1} = c_i - 1 \geq 0$ . De plus  $p_{i+1} = 2p_i = 2^{n-c_i+1} = 2^{n-(c_i-1)} = 2^{n-c_{i+1}}$ . Donc **Inv**( $i+1$ ) vérifié.
  - ② Si on sort de la boucle, alors  $c_i \leq 0$  (condition de sortie) et  $c_i \geq 0$  (par **Inv**( $i$ )) donc  $c_i = 0$ , donc  $p_i = 2^{n-c_i} = 2^n$ .

# Vocabulaire

- Il y a correction *totale* lorsque le programme termine toujours et vérifie la spécification.

# Vocabulaire

- Il y a correction *totale* lorsque le programme termine toujours et vérifie la spécification.
- Il y a correction *partielle* lorsque le programme vérifie la spécification dans tous les cas où il termine.

- 1 Terminaison et variants
- 2 Correction et invariants
- 3 Etude de cas : tri fusion

# Avertissement

Dans tout ce qui suit :

- Par *liste trié*, on sous-entend implicitement « liste triée par ordre croissant ».

# Avertissement

Dans tout ce qui suit :

- Par *liste trié*, on sous-entend implicitement « liste triée par ordre croissant ».
- Par *permutation d'une liste* on sous-entend une nouvelle liste ayant la même taille que la première, et contenant tous les éléments de la première avec le même nombre d'occurrences.

# Avertissement

Dans tout ce qui suit :

- Par *liste trié*, on sous-entend implicitement « liste triée par ordre croissant ».
- Par *permutation d'une liste* on sous-entend une nouvelle liste ayant la même taille que la première, et contenant tous les éléments de la première avec le même nombre d'occurrences.
- Par *version triée d'une liste*, on sous entend une permutation de la liste mais triée par ordre croissant.

# Historique

- Écrit vers 1945.

# Historique

- Écrit vers 1945.
- Attribué au Hongro-Américain John Von Neumann.

# Historique

- Écrit vers 1945.
- Attribué au Hongro-Américain John Von Neumann.
- Un des premiers algorithmes de tris proposé pour les ordinateurs.

# Historique

- Écrit vers 1945.
- Attribué au Hongro-Américain John Von Neumann.
- Un des premiers algorithmes de tris proposé pour les ordinateurs.
- Utilise le principe *diviser pour régner* qui consiste à décomposer récursivement un gros problème en plus petits sous-problèmes.

# Principe

On veut une version triée d'une liste **1**.

- Découper la liste en deux parties de tailles proches.

# Principe

On veut une version triée d'une liste **1**.

- Découper la liste en deux parties de tailles proches.
- Trier ces deux parties.

# Principe

On veut une version triée d'une liste `l`.

- Découper la liste en deux parties de tailles proches.
- Trier ces deux parties.
- Fusionner les deux listes triées obtenues.

# Fusion (rappel)

```
let rec fusion l1 l2 = match l1, l2 with
| [], l -> l
| l, [] -> l
| x1::t1, x2::t2 -> if x1<=x2
then x1 :: fusion t1 l2
else x2 :: fusion l1 t2 ;;
```

- Terminaison : déjà établie

# Fusion (rappel)

```

let rec fusion l1 l2 = match l1, l2 with
| [], l -> l
| l, [] -> l
| x1::t1, x2::t2 -> if x1<=x2
then x1 :: fusion t1 l2
else x2 :: fusion l1 t2 ;;

```

- Terminaison : déjà établie
- Correction : on veut montrer que pour deux listes triées par ordre croissant  $l_1, l_2$ , la liste obtenue par l'appel `fusion l1 l2` est une version triée de  $l_1 @ l_2$ .

# Correction de la fusion

Préconditions :  $l_1$  et  $l_2$  sont des listes triées d'entiers.

- On note  $n_1 = |l_1|$  et  $n_2 = |l_2|$  les longueurs des listes opérandes.

# Correction de la fusion

Préconditions : `l1` et `l2` sont des listes triées d'entiers.

- On note  $n_1 = |l_1|$  et  $n_2 = |l_2|$  les longueurs des listes opérandes.
- On montre par récurrence sur  $n_1 + n_2 = n$  la propriété  $P(n) \ll$  la liste résultat de `fusion l1 l2` est une version triée de `l1@l2`  $\gg$ .

Remarque : sa taille est donc  $n_1 + n_2$

# Correction de la fusion

Préconditions : `l1` et `l2` sont des listes triées d'entiers.

- On note  $n_1 = |l_1|$  et  $n_2 = |l_2|$  les longueurs des listes opérandes.
- On montre par récurrence sur  $n_1 + n_2 = n$  la propriété  $P(n) \ll$  la liste résultat de `fusion l1 l2` est une version triée de `l1@l2`  $\gg$ .

Remarque : sa taille est donc  $n_1 + n_2$

- Cas de base. Lorsque  $n_1 + n_2 = 0$ , alors les deux listes sont vides. On retourne la liste vide qui est triée et contient les éléments de `l1,l2`.

# Correction de la fusion

## Hérédité

On suppose  $P(n)$  vérifiée. On réalise l'appel `fusion l1 l2` avec  $n_1 + n_2 = n + 1$ .

- Si l'une des deux listes est vide, on retourne celle qui ne l'est pas. L'invariant est vérifié trivialement.

# Correction de la fusion

## Hérédité

On suppose  $P(n)$  vérifiée. On réalise l'appel `fusion l1 l2` avec  $n_1 + n_2 = n + 1$ .

- Si l'une des deux listes est vide, on retourne celle qui ne l'est pas. L'invariant est vérifié trivialement.
- Si les deux listes sont non vides, supposons que l'appel interne soit `fusion t1 l2` (se produit si le premier élément  $x_1$  de `l1` est inférieur au premier de `l2` ; l'autre cas est laissé au lecteur).

# Correction de la fusion

## Hérédité

On suppose  $P(n)$  vérifiée. On réalise l'appel `fusion l1 l2` avec  $n_1 + n_2 = n + 1$ .

- Si l'une des deux listes est vide, on retourne celle qui ne l'est pas. L'invariant est vérifié trivialement.
- Si les deux listes sont non vides, supposons que l'appel interne soit `fusion t1 l2` (se produit si le premier élément  $x_1$  de `l1` est inférieur au premier de `l2` ; l'autre cas est laissé au lecteur).
  - L'hypothèse de récurrence s'applique puisque  $|t_1| + |l_2| = n_1 - 1 + n_2 = n$  et `t1,l2` sont triés

# Correction de la fusion

## Hérédité

On suppose  $P(n)$  vérifiée. On réalise l'appel `fusion l1 l2` avec  $n_1 + n_2 = n + 1$ .

- Si l'une des deux listes est vide, on retourne celle qui ne l'est pas. L'invariant est vérifié trivialement.
- Si les deux listes sont non vides, supposons que l'appel interne soit `fusion t1 l2` (se produit si le premier élément  $x_1$  de `l1` est inférieur au premier de `l2` ; l'autre cas est laissé au lecteur).
  - L'hypothèse de récurrence s'applique puisque  $|t_1| + |l_2| = n_1 - 1 + n_2 = n$  et `t1,l2` sont triés
  - Donc `fusion t1 l2` est une version triée de `t1@l2`.

# Correction de la fusion

## Hérédité

On suppose  $P(n)$  vérifiée. On réalise l'appel `fusion l1 l2` avec  $n_1 + n_2 = n + 1$ .

- Si l'une des deux listes est vide, on retourne celle qui ne l'est pas. L'invariant est vérifié trivialement.
- Si les deux listes sont non vides, supposons que l'appel interne soit `fusion t1 l2` (se produit si le premier élément  $x_1$  de `l1` est inférieur au premier de `l2` ; l'autre cas est laissé au lecteur).
  - L'hypothèse de récurrence s'applique puisque  $|t_1| + |l_2| = n_1 - 1 + n_2 = n$  et `t1,l2` sont triés
  - Donc `fusion t1 l2` est une version triée de `t1@l2`.
  - Comme `t1,l2` ne contiennent que des éléments plus grands que  $x_1$ , `x1::fusion t1 l2` est triée et contient les bons éléments. IZP<sup>1</sup>!!

## Fonction de séparation

La fonction suivante coupe en deux parties de longueurs presque égales une liste :

```
let rec separer l =
  match l with
  | [] | [_] -> l, []
  | x::y::q -> let l1, l2 = separer q in x::l1, y::l2;;
```

Cette fonction termine car 1) la longueur de l'argument de `separer` diminue strictement à chaque appel récursif et 2) les cas de bases terminent.

### Exercice

Produire une version en récurrence terminale.

## Fonction de séparation

On montre que `separer l` renvoie 2 listes de tailles  $\lceil \frac{l}{2} \rceil$  et  $\lfloor \frac{l}{2} \rfloor$  (dans cet ordre) et que la concaténation des deux listes renvoyées est une permutation de la liste de départ :

- Cas de bases : C'est vrai pour les cas d'arrêts (vérification immédiate).

## Fonction de séparation

On montre que `separer l` renvoie 2 listes de tailles  $\lceil \frac{\ell}{2} \rceil$  et  $\lfloor \frac{\ell}{2} \rfloor$  (dans cet ordre) et que la concaténation des deux listes renvoyées est une permutation de la liste de départ :

- Cas de bases : C'est vrai pour les cas d'arrêts (vérification immédiate).
- Hérité : Si la propriété est vraie pour une liste  $\ell$  de taille égale ou inférieure à  $n > 0$  (récurrence forte), considérons une liste de taille  $n + 1$  et effectuons `separer l`.

Par HR, l'appel interne (fait avec des listes de tailles  $n - 1$ ) renvoie deux listes de tailles  $\lceil \frac{n-1}{2} \rceil$  et  $\lfloor \frac{n-1}{2} \rfloor$  dont la concaténation est une permutation de `q`.

La preuve en deux points (contenu et taille) est détaillée dans le transparent suivant.

# Fonction de séparation

## Hérédité

On admet HR pour  $n > 0$ . On effectue `separer l` pour  $l = n + 1$ .

- Contenu : En ajoutant les deux éléments  $x, y$  à chacune des deux listes résultats (dont la concaténation est une permutation de `q`), et en les concaténant, on obtient bien une permutation de `l`.

# Fonction de séparation

## Hérédité

On admet HR pour  $n > 0$ . On effectue `separer l` pour  $l = n + 1$ .

- Contenu : En ajoutant les deux éléments  $x, y$  à chacune des deux listes résultats (dont la concaténation est une permutation de `q`), et en les concaténant, on obtient bien une permutation de `l`.
- Taille :

# Fonction de séparation

## Hérédité

On admet HR pour  $n > 0$ . On effectue `separer 1` pour  $\ell = n + 1$ .

- Contenu : En ajoutant les deux éléments  $x, y$  à chacune des deux listes résultats (dont la concaténation est une permutation de `q`), et en les concaténant, on obtient bien une permutation de `1`.
- Taille :
  - Si  $n$  est de la forme  $2k + 1$ , alors  $\lceil \frac{n-1}{2} \rceil = \lceil k \rceil = k$  et  $\lfloor \frac{n-1}{2} \rfloor = \lfloor k \rfloor = k$ . Du fait de la concaténation, les listes retournées sont de tailles  $k + 1 = \lceil \frac{n+1}{2} \rceil$  et  $k + 1 = \lfloor \frac{n+1}{2} \rfloor$ .

# Fonction de séparation

## Hérédité

On admet HR pour  $n > 0$ . On effectue `separer 1` pour  $\ell = n + 1$ .

- Contenu : En ajoutant les deux éléments  $x, y$  à chacune des deux listes résultats (dont la concaténation est une permutation de `q`), et en les concaténant, on obtient bien une permutation de `l`.

- Taille :

- Si  $n$  est de la forme  $2k + 1$ , alors  $\lceil \frac{n-1}{2} \rceil = \lceil k \rceil = k$  et  $\lfloor \frac{n-1}{2} \rfloor = \lfloor k \rfloor = k$ . Du fait de la concaténation, les listes retournées sont de tailles  $k + 1 = \lceil \frac{n+1}{2} \rceil$  et  $k + 1 = \lfloor \frac{n+1}{2} \rfloor$ .
- Si  $n$  est de la forme  $2k$ , alors  $\lceil \frac{n-1}{2} \rceil = \lceil k - \frac{1}{2} \rceil = k$  et  $\lfloor \frac{n-1}{2} \rfloor = \lfloor k - \frac{1}{2} \rfloor = k - 1$ . Du fait de la concaténation, les listes retournées sont de tailles  $k + 1 = \lceil \frac{n+1}{2} \rceil$  et  $k = \lfloor \frac{n+1}{2} \rfloor$ .

# Fonction de séparation

## Hérédité

On admet HR pour  $n > 0$ . On effectue `separer 1` pour  $\ell = n + 1$ .

- Contenu : En ajoutant les deux éléments  $x, y$  à chacune des deux listes résultats (dont la concaténation est une permutation de `q`), et en les concaténant, on obtient bien une permutation de `l`.
- Taille :
  - Si  $n$  est de la forme  $2k + 1$ , alors  $\lceil \frac{n-1}{2} \rceil = \lceil k \rceil = k$  et  $\lfloor \frac{n-1}{2} \rfloor = \lfloor k \rfloor = k$ . Du fait de la concaténation, les listes retournées sont de tailles  $k + 1 = \lceil \frac{n+1}{2} \rceil$  et  $k + 1 = \lfloor \frac{n+1}{2} \rfloor$ .
  - Si  $n$  est de la forme  $2k$ , alors  $\lceil \frac{n-1}{2} \rceil = \lceil k - \frac{1}{2} \rceil = k$  et  $\lfloor \frac{n-1}{2} \rfloor = \lfloor k - \frac{1}{2} \rfloor = k - 1$ . Du fait de la concaténation, les listes retournées sont de tailles  $k + 1 = \lceil \frac{n+1}{2} \rceil$  et  $k = \lfloor \frac{n+1}{2} \rfloor$ .
- Hérédité OK.

# Le tri fusion

## Exercice

Écrire la fonction `tri_fusion : 'a list -> 'a list` qui réalise le tri fusion. Etablir terminaison et correction.

# Le tri fusion

Solution.

```
let rec tri_fusion l = match l with
| [] -> l
| [_] -> l
| _ -> let l1,l2 = separer l in
        let l1' = tri_fusion l1 and l2' = tri_fusion l2 in
        fusion l1' l2';;
```



# Terminaison

- Dans les cas de bases, (liste vide ou singleton), la fonction termine de façon évidente.

# Terminaison

- Dans les cas de bases, (liste vide ou singleton), la fonction termine de façon évidente.
- Si `tri_fusion l` termine pour tout `l` de taille  $\leq n$ , considérons l'appel `tri_fusion l` pour `l` de taille  $n + 1$  avec  $n > 0$ .

# Terminaison

- Dans les cas de bases, (liste vide ou singleton), la fonction termine de façon évidente.
- Si `tri_fusion l` termine pour tout `l` de taille  $\leq n$ , considérons l'appel `tri_fusion l` pour `l` de taille  $n + 1$  avec  $n > 0$ .
- L'appel à `separer` termine et retourne deux listes de tailles  $\lfloor \frac{n+1}{2} \rfloor$  et  $\lceil \frac{n+1}{2} \rceil$ . Ces deux listes sont passées chacune en argument de `tri_fusion`.

# Terminaison

- Dans les cas de bases, (liste vide ou singleton), la fonction termine de façon évidente.
- Si `tri_fusion l` termine pour tout `l` de taille  $\leq n$ , considérons l'appel `tri_fusion l` pour `l` de taille  $n + 1$  avec  $n > 0$ .
- L'appel à `separer` termine et retourne deux listes de tailles  $\lfloor \frac{n + 1}{2} \rfloor$  et  $\lceil \frac{n + 1}{2} \rceil$ . Ces deux listes sont passées chacune en argument de `tri_fusion`.
- Les deux appels ci-dessus terminent par hypothèse de récurrence puisque,  $n + 1$  étant plus grand que 2, les deux parties entières de  $\frac{n + 1}{2}$  sont strictement plus petites que  $n + 1$ .

# Terminaison

- Dans les cas de bases, (liste vide ou singleton), la fonction termine de façon évidente.
- Si `tri_fusion l` termine pour tout `l` de taille  $\leq n$ , considérons l'appel `tri_fusion l` pour `l` de taille  $n + 1$  avec  $n > 0$ .
- L'appel à `separer` termine et retourne deux listes de tailles  $\lfloor \frac{n + 1}{2} \rfloor$  et  $\lceil \frac{n + 1}{2} \rceil$ . Ces deux listes sont passées chacune en argument de `tri_fusion`.
- Les deux appels ci-dessus terminent par hypothèse de récurrence puisque,  $n + 1$  étant plus grand que 2, les deux parties entières de  $\frac{n + 1}{2}$  sont strictement plus petites que  $n + 1$ .
- Enfin, la fusion de deux listes termine toujours donc `tri_fusion l` termine. Hérité OK.

## Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.

## Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.
- Si `tri_fusion l` renvoie une version triée de `l` pour tout `l` de taille  $\leq n$ , considérons l'appel `tri l` pour `l` de taille  $n + 1$  avec  $n > 0$ .

## Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.
- Si `tri_fusion l` renvoie une version triée de `l` pour tout `l` de taille  $\leq n$ , considérons l'appel `tri l` pour `l` de taille  $n + 1$  avec  $n > 0$ .
- L'appel à `separer` termine et retourne 2 listes `l1,l2` qui, à elle deux, contiennent exactement les éléments de `l`. Elles sont chacune strictement plus courtes que `l` puisque  $n + 1 > 1$ .

## Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.
- Si `tri_fusion l` renvoie une version triée de `l` pour tout `l` de taille  $\leq n$ , considérons l'appel `tri l` pour `l` de taille  $n + 1$  avec  $n > 0$ .
- L'appel à `separer` termine et retourne 2 listes `l1,l2` qui, à elle deux, contiennent exactement les éléments de `l`. Elles sont chacune strictement plus courtes que `l` puisque  $n + 1 > 1$ .
- Ces deux listes sont passées chacune en argument de `tri_fusion` et, par HR, on obtient une version triée de chacune. On les note `l1',l2'`.

## Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.
- Si `tri_fusion l` renvoie une version triée de `l` pour tout `l` de taille  $\leq n$ , considérons l'appel `tri l` pour `l` de taille  $n + 1$  avec  $n > 0$ .
- L'appel à `separer` termine et retourne 2 listes `l1,l2` qui, à elle deux, contiennent exactement les éléments de `l`. Elles sont chacune strictement plus courtes que `l` puisque  $n + 1 > 1$ .
- Ces deux listes sont passées chacune en argument de `tri_fusion` et, par HR, on obtient une version triée de chacune. On les note `l1',l2'`.
- Enfin, `fusion` appliquée à ces 2 listes retourne une version triée de `l1'@l2'`.

## Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.
- Si `tri_fusion l` renvoie une version triée de `l` pour tout `l` de taille  $\leq n$ , considérons l'appel `tri l` pour `l` de taille  $n + 1$  avec  $n > 0$ .
- L'appel à `separer` termine et retourne 2 listes `l1,l2` qui, à elle deux, contiennent exactement les éléments de `l`. Elles sont chacune strictement plus courtes que `l` puisque  $n + 1 > 1$ .
- Ces deux listes sont passées chacune en argument de `tri_fusion` et, par HR, on obtient une version triée de chacune. On les note `l1',l2'`.
- Enfin, `fusion` appliquée à ces 2 listes retourne une version triée de `l1'@l2'`.
- Cette liste est une version triée de `l1@l2` donc de `l`. IZP