# 1 Expressions

La fonction **abs** : $\mathbb{Z} \longrightarrow \mathbb{N}$ qui renvoie la valeur absolue d'un entier relatif est une fonction prédéfinie de notre langage d'algorithme. On suppose définis les algorithmes suivants :

> **Algorithme** : proche ?
> **Données** : `a` : Nombre, `b` : Nombre
> **Résultat** : Booléen, qui vaut `true` si la distance entre `a` et `b` est inférieure à 2 et `false` sinon.
> **Le résultat est :** `abs(b-a) < 2`
> **fin algorithme**
> **Algorithme** : reste
> **Données** : $x$ : $Nombre$ ; $x$ est entier.
> **Résultat** : Nombre, égal au reste de la division entière de $x$ par 2.
> **Le résultat est :** `x mod 2`
> **fin algorithme**

1. Quels sont les types et valeurs des expressions suivantes :

| | |
|---|---|
| `(2 + (3 * (7 - 5)))` | `2 + (3 * true)` |
| `(true ou false)` | `true et false` |
| `non (true et false)` | `(2 < 3) et false` |
| `proche?(2,7)` | `proche?(1,2,3)` |
| `proche?(1,1) et (proche?(2,3) et proche?(1,3))` | `proche?(1,2) ou proche?((5*48) quo 7,33)` |
| `(reste(3) et reste(5)) < 3` | `proche?(reste(5),2)` |
| `(5 + (reste(5) + 1)) quo 2` | `proche?(reste(12),3-reste(7))` |
| `cond( reste(5)=1, reste(2)+1, reste(2))` | `cond( proche?(4,2), 2/0, 1)` |
| `cond( proche?(1,4), false, proche?(2,3))` | `cond( 1 < 3, reste(3), true)` |

[redacted]

| | | | |
|---|---|---|---|
| colonne gauche | `(2 + (3 * (7 - 5)))` | Nombre | 8 |
| | `(true ou false)` | Booléen | true |
| | `non (true et false)` | Booléen | true |
| | `proche?(2,7)` | Booléen | false |
| | `proche?(1,1) et (proche?(2,3) et proche?(1,3))` | Booléen | false |
| | `(reste(3) et reste(5)) < 3` | Erreur | |
| | `(5 + (reste(5) + 1)) quo 2` | Nombre | 3 |
| | `cond( reste(5)=1, reste(2)+1, reste(2))` | Nombre | 1 |
| | `cond( proche?(1,4), false, proche?(2,3))` | Booléen | true |

[redacted]

| | | | |
|---|---|---|---|
| colonne droite | `2 + (3 * true)` | Erreur | |
| | `true et false` | Booléen | false |
| | `(2 < 3) et false` | Booléen | false |
| | `proche?(1,2,3)` | Erreur | |
| | `proche?(1,2) ou proche?((5*48) quo 7,33)` | Booléen | true |
| | `proche?(reste(5),2)` | Booléen | true |
| | `proche?(reste(12),3-reste(7))` | Booléen | false |
| | `cond( proche?(4,2), 2/0, 1)` | Nombre | 1 |
| | `cond( 1 < 3, reste(3), true)` | Erreur | |

2. On suppose à présent que `x` et `y` sont des paramètres de type `Nombre` et que `z` est un paramètre de type `Booléen`. Quel est le type des expressions suivantes ?

| | |
|---|---|
| `x + y` | `x < y` |
| `z et (x=y)` | `z + x` |
| `reste(x) + y` | `proche?(x,y) et (x < y)` |
| `reste(x) = reste(y)` | `abs(x+y) quo reste(x)` |
| `cond( non(z), reste(y), (x mod y))` | `cond( z, reste(x)<1, proche?(y,3))` |
| `cond( proche?(x,y), z, x)` | `proche?(reste(x),abs(y)) ou z` |

[redacted]

| | | | |
|---|---|---|---|
| `x + y` | Nombre | `x < y` | Booléen |
| `z et (x=y)` | Booléen | `z + x` | Erreur |
| `reste(x) + y` | Nombre | `proche?(x,y) et (x < y)` | Booléen |
| `reste(x) = reste(y)` | Booléen | `abs(x+y) quo reste(x)` | Nombre |
| `cond( non(z), reste(y), (x mod y))` | Nombre | `cond( z, reste(x)<1, proche?(y,3))` | Booléen |
| `cond( proche?(x,y), z, x)` | Erreur | `proche?(reste(x),abs(y)) ou z` | Booléen |

## 2 Algorithmes non récursifs

1. Écrivez l'algorithme `distance` qui étant donné deux nombres calcule leur distance, c'est à dire la valeur absolue de leur différence.

[redacted]

**Algorithme** : `distance`
**Données** : `a : Nombre, b : Nombre`
**Résultat** : `Nombre, la distance de a à b`

**Le résultat est :** `abs(a-b)`
**fin algorithme**

2. La fonction `max` calculant le maximum de deux nombres et la fonction `abs` calculant la valeur absolue d'un nombre sont deux fonctions prédéfinies de notre langage d'algorithme. On peut cependant définir leur algorithme en utilisant l'opérateur conditionnel. Écrivez ces deux algorithmes.

[redacted]

**Algorithme** : `max`
**Données** : `a : Nombre, b : Nombre`
**Résultat** : `Nombre, le maximum entre a et b`

**Le résultat est :** `cond( a<b, b, a)`
**fin algorithme**

**Algorithme** : `abs`
**Données** : `a : Nombre`
**Résultat** : `Nombre, la valeur absolue de a`

**Le résultat est :** `cond( a<0, -1*a, a)`
**fin algorithme**
`FIN TD1`

3. La règle du max de la Faculté des Sciences, ou comment calculer la note finale à une UE :
« Soit `noteExam` et `noteCC` vos deux notes (sur 20) au contrôle terminal et au contrôle continu. Soit `coeffExam` et `coeffCC` les poids respectifs de ces notes. On calcule la note sur 20 qui est la moyenne pondérée de `NoteExam` et `noteCC`. Votre note finale sur 20 est le max entre la note précédente et la note à l'examen. » Écrire un algorithme qui prend en paramètre les deux notes et les deux coefficients et donne comme résultat la note finale.

[redacted]

**Algorithme** : `noteUE`
**Données** : `noteExam : Nombre, noteCC : Nombre, coeffExam : Nombre, coeffCC : Nombre ; noteExam et noteCC appartiennent à [0,20]; coeffExam et coeffCC sont strictement positifs`
**Résultat** : `Nombre, la note de l'UE calculée selon la règle du Max`

**Le résultat est :**
```
cond( (noteExam*coeffExam + noteCC*coeffCC) / (coeffExam+coeffCC) > noteExam,
(noteExam*coeffExam + noteCC*coeffCC) / (coeffExam+coeffCC),
noteExam)
ou bien :
```
**Le résultat est :**
```
max( noteExam , (noteExam*coeffExam + noteCC*coeffCC) / (coeffExam+coeffCC) )
```
**fin algorithme**

4. Définissez l'algorithme `max3` qui calcule le maximum de 3 nombres. Vous donnerez deux versions d'algorithme, l'une n'utilisant que l'opérateur conditionnelle, l'autre n'utilisant que la fonction `max`.

[redacted]

**Algorithme** : max3
**Données** : a : Nombre, b : Nombre, c : Nombre
**Résultat** : Nombre, le plus grand de a, b ou c

**Le résultat est :**
cond( a≥b et a≥c, a,
cond( b≥c, b,
c))
**fin algorithme**

**Algorithme** : max3
**Données** : a : Nombre, b : Nombre, c : Nombre
**Résultat** : Nombre, le plus grand de a, b ou c

**Le résultat est :** max(a,max(b,c))
**fin algorithme**

5. Écrivez l'algorithme `plusProche` qui, étant donné trois nombres `a`, `b` et `x`, renvoie le nombre choisi parmi `a` et `b` qui est le plus proche de `x`. Par exemple, si `a=5`, `b=9` quand `x=6` le plus proche est `5`, quand `x=10` le plus proche est `9`, quand `x=7` le résultat est soit `5` soit `9`.

[redacted]

**Algorithme** : plusProche
**Données** : a : Nombre, b : Nombre, x : Nombre
**Résultat** : Nombre, a ou b le plus proche de x

**Le résultat est :** cond( abs(a-x) ≤ abs(b-x), a, b)
ou bien
**Le résultat est :** cond( distance(a,x) ≤ distance(b,x), a, b)
**fin algorithme**

6. Écrivez un algorithme pour la fonction **`médian`** qui étant donné trois nombres fournit l'élément médian de ces nombres. Par exemple l'élément médian de 12, 3 et 8 est 8.

[redacted]

**Algorithme** : median
**Données** : a : Nombre, b : Nombre, c : Nombre
**Résultat** : Nombre, le nombre médian entre a, b et c

**Le résultat est :**
cond( a≤b, cond( b≤c, b, max(a,c)),
cond( a≤c, a,
max(b,c)))
ou bien
**Le résultat est :** (max(a,b)+max(a,c)+max(b,c))-2*max3(a,b,c) (*solution d'un étudiant*)
**fin algorithme**

7. Le ≪ ou exclusif ≫ est la fonction booléenne dont la signature est
**ouExcl** : $Bool \times Bool \longrightarrow Bool$ et dont la sémantique est donnée par la table :

| a | b | ouExcl(a,b) |
|---|---|---|
| true | true | false |
| true | false | true |
| false | true | true |
| false | false | false |

Écrivez deux versions d'algorithme calculant cette fonction, la première utilisant l'opérateur conditionnel, la seconde ne l'utilisant pas.

[redacted]

**Algorithme** : ouExcl
**Données** : a : Booléen, b : Booléen
**Résultat** : Booléen, a ou exclusif b

**Le résultat est :**
cond( a et non(b), true,
cond( non(a) et b, true,
false))
**fin algorithme**

**Algorithme** : ouExcl
**Données** : a : Booléen, b : Booléen
**Résultat** : Booléen, a ou exclusif b

**Le résultat est :** (a et non(b)) ou (non(a) et b)
**fin algorithme**

# 3 Algorithmes récursifs

1. Que calcule l'algorithme suivant ? Exprimez le résultat en fonction du paramètre $n$.

**Algorithme** : mystere
**Données** : n : Nombre ; n est un entier naturel.
**Résultat** : Nombre, ? ?

**Le résultat est :** cond( n=0, 0, n*n + mystere(n-1))
**fin algorithme**

[redacted]
vu en cours
$0^2 + 1^2 + ... + n^2$
FIN TD2

2. En utilisant la **récursivité** vous écrirez un algorithme qui calcule la somme $1+2+...+n$ en fonction de son paramètre $n$.

[redacted]
Pour calculer somme(n) on utilise le schéma récursif :
| Cas de base : quand n=0, somme(n) vaut 0
| Équation de récurrence : quand n>0, somme(n) vaut n + somme(n-1)

**Algorithme** : somme
**Données** : n : Nombre ; n est un entier naturel
**Résultat** : Nombre, 0 + 1 + 2 + ... + n

**Le résultat est :** cond( n=0, 0, n + somme(n-1))
**fin algorithme**

3. Écrivez le corps de l'algorithme dont la spécification est :

**Algorithme** : sommeInterv
**Données** : $a$ : Nombre, $b$ : Nombre ; $a$ et $b$ sont 2 entiers tels que $a \leq b$.
**Résultat** : Nombre, somme des entiers $a + (a + 1) + \ldots + b$

[redacted]
Pour calculer sommeInterv(a,b) on utilise le schéma récursif :
| Cas de base : quand a=b, sommeInterv(a,b) vaut a (ou b)
| Équation de récurrence : quand a<b, sommeInterv(a,b) vaut a + sommeInterv(a+1,b) (ou b + sommeInterv(a,b-1))

D'où : **Le résultat est :** cond( a=b, a, a + sommeInterv(a+1,b))
**fin algorithme**

4. Écrivez un algorithme `suiteU` qui étant donné un entier naturel $n$, calcule le $n^{ème}$ terme de la suite $(U_n)_{n\in\mathbb{N}}$ définie par : $U_0 = 1$ et $\forall n > 0$ $U_n = 2 \times U_{n-1} + 3$

[redacted]

```
Pour calculer suiteU(n) on utilise le schéma récursif :
     | Cas de base : quand n=0, suiteU(n) vaut 1
     | Équation de récurrence : quand n>0, suiteU(n) vaut 2*suiteU(n-1) +3)
D'où :
```

**Algorithme** : suiteU
**Données** : n : Nombre ; n est un entier naturel
**Résultat** : Nombre, $U_n$

**Le résultat est :** cond( n=0 , 1, 2*suiteU(n-1) + 3)
**fin algorithme**

5. On cherche un algorithme calculant le carré d'un entier naturel $n$, sans utiliser de multiplication. Appelons `carré` cet algorithme et $n$ son paramètre.

   Pour calculer `carré(n)` on utilise un schéma récursif :
   — **Cas de base :** quand `n=0` que vaut `carré(n)` ?
   — **Équation de récurrence :** quand `n`$\neq$`0` essayez de définir `carré(n)` en fonction de `carré(n-1)` (pour cela développez l'expression $(n-1)^2$).

   Écrivez l'algorithme `carré`.

[redacted]

```
Pour calculer carré(n) on utilise le schéma récursif :
     | Cas de base : quand n=0, carré(n) vaut 0
     | Équation de récurrence : quand n>0, carré(n) vaut carré(n-1)+n+n-1
D'où :
```

**Algorithme** : carré
**Données** : n : Nombre ; n est un entier naturel
**Résultat** : Nombre, $n^2$

**Le résultat est :** cond( n=0, 0, carré(n-1) + n + n - 1)
**fin algorithme**

6. Écrire un algorithme récursif `estPair` qui, étant donné un entier `n` positif ou nul renvoie `true` si `n` est un entier pair, `false` sinon. Les seules opérations autorisées sont la soustraction, les comparaisons et la conditionnelle.

[redacted]

```
Pour calculer estPair(n) on utilise le schéma récursif :
     | Cas de base : quand n=0, estPair(n) vaut true
     | Équation de récurrence : quand n>0, estPair(n) vaut non(estPair(n-1))
ou
     | Cas de base :
        | quand n=0, estPair(n) vaut true
        | quand n=1, estPair(n) vaut false
     | Équation de récurrence : quand n>1, estPair(n) vaut estPair(n-2)
D'où :
```

**Algorithme** : estPair
**Données** : n : Nombre ; n est un entier naturel
**Résultat** : Nombre, true si n est pair, false sinon

**Le résultat est :** cond( n=0, true, non(estPair(n-1)))
**fin algorithme**

FIN TD3

7. Écrivez un algorithme qui renvoie la somme des entiers impairs inférieurs ou égaux à $n$ : $1+3+5+\ldots n$ (si $n$ est impair) ou $1+3+5+\ldots n-1$ (si $n$ est pair).

[redacted]

```
Pour calculer somImp(n) on utilise le schéma récursif :
     | Cas de base : quand n=0, somImp(n) vaut 0
     | Équation de récurrence :
        | quand n>0 et n pair, somImp(n) vaut somImp(n-1)
        | quand n>0 et n impair, somImp(n) vaut n + somImp(n-1)
D'où :
```

**Algorithme** : somImp
**Données** : n : Nombre ; n est un entier naturel
**Résultat** : Nombre, somme des entiers impairs ≤ n)

**Le résultat est :**
```
cond( n=0, 0,
cond( estPair(n), somImp(n-1),
n + somImp(n-1)))
```
**fin algorithme**

8. Complétez l'algorithme dont les spécifications sont :

**Algorithme** : plusPetitDiv
**Données** : $a$ : Nombre, $b$ : Nombre ; $a$ et $b$ sont 2 entiers naturels tels que $a \leq b$.
**Résultat** : Nombre, le plus petit entier $d$ tel que $a \leq d \leq b$ et $d$ divise $b$.

*Exemple :* `plusPetitDiv(2,35)` vaut 5, `plusPetitDiv(3,9)` vaut 3 et `plusPetitDiv(8,35)` = 35.

[redacted]

```
Pour calculer plusPetitDiv(a,b) on utilise le schéma récursif :
     | Cas de base : quand a divise b, plusPetitDiv(a,b) vaut a
     | Équation de récurrence : quand a ne divise pas b, plusPetitDiv(a,b) vaut
       plusPetitDiv(a+1,b) (comme a ≤ b et b divise b le cas de base sera atteint)
D'où :
```

**Le résultat est :**
```
cond( b mod a = 0, a, plusPetitDiv(a+1,b))
```
**fin algorithme**

Utilisez l'algorithme `plusPetitDiv` pour définir un algorithme qui teste si un nombre est un nombre premier. Les spécifications de cet algorithme sont :

**Algorithme** : estPremier
**Données** : $n$ : Nombre ; $n$ est un entier natuel
**Résultat** : Booléen, qui vaut `true` si $n$ est un nombre premier, $false$ sinon.

On rappelle qu'un nombre $n \neq 1$ est premier si ses seuls diviseurs sont 1 et $n$. On rappelle également que 0 et 1 ne sont pas des nombres premiers.

[redacted]

**Le résultat est :**
```
cond( n = 0 ou n = 1, false, plusPetitDiv(2,n) = n)
```
**fin algorithme**

9. (**) La multiplication dite « du paysan russe ». On souhaite multiplier deux entiers positifs $a$ et $b$ en n'utilisant que des additions, la multiplication par 2 et la division par 2. Soit `mul` le nom de cet algorithme. Pour définir `mul` on utilise un schéma récursif exprimant `mul(a,b)` en fonction de `mul(a quo 2,b)`. Complétez ce schéma :
   - **Cas de base :** quand `a=0` `mul(a,b)` vaut ...
   - **Équation de récurrence :** quand `a≠0` il existe 2 cas :
     - quand `a` est pair `mul(a,b)` vaut ... `mul(a quo 2, b)` ...
     - quand `a` est impair `mul(a,b)` vaut ... `mul(a quo 2, b)` ...

   Écrivez l'algorithme `mul`.

[redacted]

```
Pour calculer mul(a,b) on utilise le schéma récursif :
     | Cas de base : quand a=0, mul(a,b) vaut 0
     | Équation de récurrence : quand a≠0 il existe 2 cas :
        | quand a est pair, mul(a,b) vaut 2 * mul(a div 2, b)
        | quand a est impair, mul(a,b) vaut 2 * mul(a div 2, b) + b
D'où :
```

**Algorithme** : mul
**Données** : a : Nombre, b : Nombre ; a et b sont des entiers naturels
**Résultat** : Nombre, a*b

**Le résultat est :**
```
cond( a=0, 0,
cond( a mod 2 = 0, 2 * mul(a div 2, b),
2 * mul(a div 2, b) + b))
```
**fin algorithme**

FIN TD4

10. (***) De façon analogue, l'exponentiation rapide, se définit comme suit. On souhaite calculer $a^b$ où $a, b$ sont deux entiers positifs. Mais on ne peut utiliser que des multiplications.

Le schéma récursif d'une telle exponentiation est le suivant :

$$\begin{cases} \texttt{si } b \texttt{ pair} & a^b = a^{2*b'} = a^{b'} * a^{b'} \\ \texttt{si } b \texttt{ impair} & a^b = a^{2*b'+1} = (a^{2*b'}) * a \end{cases}$$

Écrivez l'algorithme récursif de cette exponentiation.

[redacted]

**Algorithme** : expo
**Données** : a : Nombre, b : Nombre ; a et b sont des entiers naturels
**Résultat** : Nombre, $a^b$

**Le résultat est :**
```
cond( b=0, 1,
cond( b mod 2 = 0, expo(a, b div 2) * expo(a, b div 2),
expo(a, b div 2) * expo(a, b div 2) * a))
```
**fin algorithme**

ou

**Algorithme** : expo
**Données** : a : Nombre, b : Nombre ; a et b sont des entiers naturels
**Résultat** : Nombre, $a^b$

**Le résultat est :**
```
cond( b=0, 1,
cond( b mod 2 = 0, expo(a, b div 2) * expo(a, b div 2),
expo(a, b-1) * a))
```
**fin algorithme**

11. (****) On cherche à calculer le nombre de façons de répartir $n$ objets dans $b$ boîtes différenciées. On compte les répartitions avec boîte vide. Par exemple les répartitions de 3 objets dans 2 boîtes sont | 0 | 3 | | 1 | 2 | | 2 | 1 | | 3 | 0 | et donc le nombre de répartitions est 4. Complétez l'algorithme :

**Algorithme** : nbRepart
**Données** : $n$ : Nombre, $b$ : Nombre ; $n$ et $b$ sont 2 entiers naturels, $b \neq 0$.
**Résultat** : Nombre, le nombre de façons de répartir $n$ objets dans $b$ boîtes différenciées.

Le schéma de récurrence permettant de définir `nbRepart` utilise une récurrence sur les deux paramètres $n$ et $b$.

[redacted]

[redacted]

```
Pour calculer nbRepart(n,b) on utilise le schéma récursif :
     | Cas de base :
        | quand n=0, nbRepart(n,b) vaut 1
        | quand b=1, nbRepart(n,b) vaut 1
     | Équation de récurrence : quand n>0 et b>1, nbRepart(n,b) vaut nbRepart(n,b-1) +
       nbRepart(n-1,b) (on n'en met pas dans la première boîte ou on en met au moins 1)
D'où :
```

**Le résultat est :** `cond( n=0 ou b=1, 1, nbRepart(n,b-1)+nbRepart(n-1,b))`

**fin algorithme**

## 4 C/C++

Traduisez en C/C++ l'algorithme `plusProche` de l'exercice 2.5 et l'algorithme `carré` de l'exercice 3.5.

[redacted]

```
int plusProche(int a, int b, int x)
{
return abs(a-x) < abx(b-x) ? a : b ;
}
int carre(int n)
{
return n==0 ? 0 : carre(n-1) + 2*n - 1;
}
```

## 5 Listes.

1. Supposons que `li` soit un paramètre de type `Liste de Nombres`. Comment obtenir la valeur du premier élément de `li` ? du deuxième ? Comment savoir si `li` a plus d'un élément ? Comment insérer la valeur `5` en tête de la liste `li` ?

[redacted]

`tête(li) ; tête(queue(li)) ; non( estVide(li) ) et non( estVide(queue(li)) ) ; cons(5,li)`

2. Dans les expressions suivantes, substituez à `li` la liste `( 2 3 2 4 )`, puis calculez la valeur de l'expression obtenue :

| | |
|---|---|
| `queue(queue(li))` | `tête(queue(queue(li)))` |
| `tête(queue(queue(queue(queue(li)))))` | `estVide(queue(queue(queue(queue(li)))))` |
| `estVide(tete(li))` | `cons(7,li)` |
| `cons(7,queue(li))` | `cons(tête(li),queue(li))` |
| `cons(tête(queue(li)),queue(li))` | |

[redacted]

| | |
|---|---|
| (2 4) | 2 |
| ERREUR | true |
| ERREUR | ( 7 2 3 2 4 ) |
| ( 7 3 2 4 ) | ( 2 3 2 4 ) (on retrouve li ; cons(tete(l),queue(l) = identite(l)) |
| ( 3 3 2 4 ) | |

3. Écrire un algorithme qui inverse les deux premiers éléments d'une liste de nombres composée d'au moins deux éléments.

[redacted]

**Algorithme** : `inverse2PremiersElements`
**Données** : `li : Liste de Nombres ; li contient au moins 2 éléments.`
**Résultat** : `Liste de Nombres, la liste li dont les 2 premiers éléments sont inversés`

**Le résultat est :** `cons(tête(queue(li)),cons(tête(li),queue(queue(li))))`

**fin algorithme**

FIN TD5

4. Soit l'algorithme `listeEntiers`, qui étant donné un entier naturel `n`, calcule la liste `(n n-1 ... 1 0)`. Complétez la récurrence :
   — **Cas de base :** quand `n=0` que vaut `listeEntiers(0)` ?
   — **Équation de récurrence :** quand `n>0`, définissez `listeEntiers(n)` en fonction de `listeEntiers(n-1)`.
   Écrivez l'algorithme `listeEntiers`.

[redacted]

**Algorithme** : `listeEntiers`
**Données** : `n:Nombre ; n est un entier naturel`
**Résultat** : `Liste de Nombres, la liste (n n-1 ... 1 0)`

**Le résultat est :**
`cond( n=0, cons(0,liVide), cons(n,listeEntiers(n-1)))`
**fin algorithme**

5. On cherche à écrire un algorithme `maxListe` qui étant donné `li`, une liste non vide de nombres, calcule le plus grand de ses éléments. Pour cela complétez la récurrence :
   — **Cas de base :** quand `li` ne contient qu'un élément que vaut `maxListe(li)` ?
   — **Équation de récurrence :** quand `li` contient plus d'un élément, définissez `maxListe(li)` en fonction de `maxListe(queue(li))` en utilisant la fonction `max` (maximum de 2 nombres).
   Complétez alors l'algorithme :

**Algorithme** : maxListe
**Données** : li : Liste de Nombres ; li n'est pas vide.
**Résultat** : Nombre, le plus grand élément de li

[redacted]

**Algorithme** : `maxListe`
**Données** : `li : Liste de Nombres ; li n'est pas vide.`
**Résultat** : `Nombre, le plus grand élément de li`

**Le résultat est :**
`cond( estVide(queue(li)), tête(li), max(tête(li),maxListe(queue(li))))`
**fin algorithme**

6. Écrire l'algorithme `derListe` qui, étant donnée une liste non vide `li`, calcule la valeur du dernier élément de `li`. Traduisez l'algorithme `derListe` en une fonction C/C++.

[redacted]

**Algorithme** : `derListe`
**Données** : `li : Liste de Nombres ; li est non vide`
**Résultat** : `Nombre, dernier élément de li`

**Le résultat est :**
`cond( estVide(queue(li)), tête(li), derListe(queue(li)))`
**fin algorithme**

```
int derListe(list<int> li)
{
return
estVide(queue(li)) ? tete(li) :
derListe(tl(li)) ;
}
```

7. Écrire l'algorithme `appartientLi` qui étant donnés un nombre `n`, et une liste de nombres `li` vaut `true` si `n` est la valeur d'un élément de `li`, `false` sinon.

[redacted]

```
Fait en cours
    Algorithme : appartientLi
    Données : n : Nombre ; li : Liste de Nombres
    Résultat : Booléen, true si n est un élément de la liste li, false sinon.

        Le résultat est :
        cond( estVide(li), false,
        cond( tête(li)=n, true,
        appartientLi(n,queue(li))))
    fin algorithme
```

Écrire l'algorithme `nbOccListe` qui étant donnés un entier `n`, et une liste d'entiers `li` calcule le nombre d'occurrences de `n` dans `li`, c'est à dire le nombre d'éléments de la liste `li` égaux à `n`.

[redacted]

```
    Algorithme : nbOccListe
    Données : n : Nombre, li : Liste de Nombres
    Résultat : Nombre, le nombre d'occurrences de n dans la liste li

        Le résultat est :
        cond( estVide(li), 0,
        cond( tête(li)=n, 1+nbOccListe(n,queue(li)),
        nbOccListe(n,queue(li))))
    fin algorithme
```

8. Écrire l'algorithme `tousPairs` qui étant donné une liste de nombres `li` vaut `true` si chaque élément de `li` est un nombre pair, `false` sinon.

   Exemple, `tousPairs((2 8 2))` et `tousPairs(( ))` valent `true`. `tousPairs((2 5 2))` vaut `false`.

[redacted]

```
    Algorithme : tousPairs
    Données : li : Liste de Nombres
    Résultat : Booléen, true si tous les éléments de li sont pairs, false sinon

        Le résultat est :
        cond( estVide(li), true,
        cond( (tête(li) mod 2) = 1, false,
        tousPairs(queue(li))))
    fin algorithme

FIN TD6
```

9. Écrire l'algorithme `listesEgales` qui, étant données deux listes `li1` et `li2` quelconques, vaut `true` si ces deux listes sont « égales », `false` sinon.

[redacted]

```
    Algorithme : listesEgales
    Données : li1 : Liste de Nombres, li2 : Liste de Nombres
    Résultat : Booléen, ...

        Le résultat est :
        cond( estVide(li1) et estVide(li2), true,
        cond( estVide(li1) ou estVide(li2), false,
        (tête(li1)=tête(li2)) et listesEgales(queue(li1),queue(li2))) )
    fin algorithme
```

10. Soient les définitions suivantes ; `l1` et `l2` étant 2 listes :
    - `l1` est **préfixe** de `l2` si la séquence des éléments de `l2` est composée des éléments de `l1` dans le même ordre, puis d'éléments en nombre et valeur quelconques.
    - `l1` est **suffixe** de `l2` si la séquence des éléments de `l2` est composée d'éléments quelconques, suivis des éléments de `l1` dans le même ordre.
    - `l1` est **facteur** de `l2` si la séquence des éléments de `l2` est composée d'élements quelconques, suivis des éléments de `l1` dans le même ordre, suivis d'éléments quelconques.

    *Exemples :*
    - `(3 2 3)` est préfixe et facteur de la liste `(3 2 3 4 2)`.
    - `(3 2 3)` est préfixe, suffixe et facteur de la liste `(3 2 3)`.

— `()` est préfixe, suffixe et facteur de toutes les listes.
— `(3 2 3)` est suffixe et facteur de la liste `(4 3 2 3)`.
— `(3 2 3)` est facteur de la liste `(4 2 3 2 3 8)`.

Écrivez les algorithmes `prefixe`, `suffixe` et `facteur`.

[redacted]

**Algorithme** : `prefixe`
**Données** : `l1 : Liste de Nombres, l2 : Liste de Nombres`
**Résultat** : `Booléen, ‘‘l1 est préfixe de l2’’`

**Le résultat est :**
```
cond( estVide(l1), true,
cond( estVide(l2), false,
cond( tête(li1)≠tête(li2), false,
prefixe(queue(l1),queue(l2)))))
```
**fin algorithme**

**Algorithme** : `suffixe`
**Données** : `l1 : Liste de Nombres, l2 : Liste de Nombres`
**Résultat** : `Booléen, ‘‘l1 est suffixe de l2’’`

**Le résultat est :**
```
cond( listesEgales(l1,l2), true,
cond( estVide(l2), false,
suffixe(l1,queue(l2))))
```
**fin algorithme**

**Algorithme** : `facteur`
**Données** : `l1 : Liste de Nombres, l2 : Liste de Nombres`
**Résultat** : `Booléen, ‘‘l1 est facteur de l2’’`

**Le résultat est :**
```
cond( prefixe(l1,l2), true,
cond( estVide(l2), false,
facteur(l1,queue(l2))))
```
**fin algorithme**

11. Expliquez pourquoi l'algorithme suivant n'est pas correct :

**Algorithme** : ajoutFin
**Données** : n : Nombre, li : Liste de Nombres
**Résultat** : Liste de Nombres, la liste `li` à laquelle on a ajouté `n` comme dernier élément.

**Le résultat est :** `cons(li,n)`
**fin algorithme**

Modifiez le corps de cet algorithme pour obtenir une version correcte de `ajoutFin`.

[redacted]

**Algorithme** : `ajoutFin`
**Données** : `n : Nombre, li : Liste de Nombres`
**Résultat** : `Liste de Nombres, la liste li à laquelle on a ajouté n comme dernier élément.`

**Le résultat est :**
```
cond( estVide(li), cons(n,liVide), cons(tête(li),ajoutFin(n,queue(li))))
```
**fin algorithme**

12. En utilisant `ajoutFin`, écrivez un algorithme `listeInversee` qui étant une liste `li`, calcule la liste composée des éléments de `li`, mais dans l'ordre inverse.

[redacted]

**Algorithme** : `listeInversee`
**Données** : `li : Liste de Nombres`
**Résultat** : `Liste de Nombres, la liste composée des éléments de li, mais dans l’ordre inverse.`

**Le résultat est :**
```
cond( estVide(li), li, ajoutFin(tête(li),listeInversee(queue(li))))
```
**fin algorithme**

13. (***) `l1` et `l2` étant 2 listes de nombres triées dans l'ordre croissant, la fusion de `l1` et `l2` est la liste triée composée des éléments de `l1` et des éléments de `l2`. Par exemple la fusion des listes `(2 2 4 7)` et `(1 2 3 4 4)` est la liste `(1 2 2 2 3 4 4 4 7)`.

    Écrivez un algorithme réalisant cette opération.

[redacted]

**Algorithme** : `fusionListes`
**Données** : `l1 : Liste de Nombres, l2 : Liste de Nombres ; l1 et l2 sont triées croissantes`
**Résultat** : `Liste de Nombres, fusion des listes l1 et l2`

**Le résultat est :**
```
cond( estVide(l1), l2,
cond( estVide(l2), l1,
cond( tête(l1)<tête(l2), cons(tête(l1),fusionListes(queue(l1),l2)),
cons(tête(l2),fusionListes(l1,queue(l2))))))
```
**fin algorithme**

14. (****) Écrire une fonction C/C++ `listeSuffixes` qui étant donnée une liste de nombres `l1` calcule la liste des listes suffixes de `l1`. Le résultat est donc une liste dont les éléments sont des listes de nombres. L'ordre des préfixes dans la liste résultat est quelconque. Par exemple la valeur de `listeSuffixes((2 5 3))` est `( (2 5 3) (5 3) (3) () )` ou toute autre liste contenant les 4 éléments `() (3) (5 3) (2 5 3)`.

[redacted]

```
list<list<int>> liSuff(list<int> li)
{
return
estVide(li) ? cons(liVide<int>(),liVide<list <int>>()) :
cons(li,liSuff(queue(li)));
}
```

Écrire une fonction C/C++ `listePréfixes` qui étant donnée une liste de nombres `l1` calcule la liste des listes préfixes de `l1`. Par exemple la valeur de `listePréfixes((2 5 3))` est `( (2 5 3) (2 5) (2) () )` ou toute autre liste contenant les 4 éléments `(2 5 3) (2 5) (2) ()`.

[redacted]

```
list<list<int>> ajoutTete(int e, list<list<int>> li)
{
return
estVide(li) ? li :
cons(cons(e,tete(li)),ajoutTete(e,queue(li)));
}

list< list<int> > liPref(list<int> li)
{
return
estVide(li) ? cons(liVide<int>(),liVide<list<int>>()) :
cons(liVide,ajoutTete(tete(li),liPref(queue(li))));
}
FIN TD7
```