Le package `tnsalgo`

Code source disponible sur `https://github.com/typensee-latex/tnsalgo.git`.

Version `0.0.0-beta` développée et testée sur Mac OS X.

Christophe BAL

2020-09-12

Table des matières

I. Introduction

Le package `tnsalgo` propose des outils facilitant la rédaction d'algorithmes en langage naturel et si besoin graphiquement via des ordinogrammes [1].

Remarque. Pour faciliter la rédaction des exemples, cette documentation utilise le package `tnsmath` disponible sur `https://github.com/typensee-latex/tnsmath.git`.

II. Noms des macros

Pour les algorithmes en langage naturel, il est fait appel au package `algorithm2e` qui utilise, et abuse [2], de la notation dite en bosses de dromadaire [3] comme par exemple avec `\BlankLine` et `\ElseIf` au lieu de `\blankline` et `\elseif`. Par souci de cohérence les nouvelles macros ajoutées par `tnsalgo` en lien avec les algorithmes utilisent aussi cette convention même si par exemple l'auteur aurait préféré proposer `\putin` et `\forrange` à la place de `\PutIn` et `\ForRange`.

III. Packages utilisés

La roue ayant déjà été inventée, le package `tnsalgo` utilise les packages suivants sans aucun scrupule.

- `algorithm2e`
- `mathtools`
- `simplekv`
- `tcolorbox`
- `tikz`
- `xint`

1. On parle aussi d'organigrammes.
2. Ce type de convention est un peu pénible lors de la saisie au clavier.
3. On parle aussi de casse à la Pascal en référence au langage de programmation.

IV. Algorithmes en langage naturel

1. Comment taper les algorithmes avec `algorithm2e`

Le gros du travail est fait par le package `algorithm2e` qui propose une syntaxe de saisie très simple. La mise en forme par défaut de `algorithm2e` utilise des flottants, chose qui peut poser des problèmes pour de longs algorithmes ou, plus gênant, pour des algorithmes en bas de page. Dans `tnsalgo` il a été fait le choix de ne pas utiliser de flottants avec un placement libre, un choix lié à l'utilisation faite de `tnsalgo` par son auteur pour rédiger des cours de niveau lycée.

Voici ce qu'il est possible d'obtenir sans trop d'efforts avec tous les mots clés en français car c'est langue choisie avec `babel`.

Algorithme 1 : Suite de Collatz (u_k) – Conjecture de Syracuse

Donnée : $n \in \mathbb{N}$
Résultat : le premier indice $i \in [\![0, 10^5]\!]$ tel que $u_i = 1$ ou (-1) en cas d'échec

Actions
- $i, imax \leftarrow 0, 10^5$
- $u \leftarrow n$
- $continuer \leftarrow \top$
- **Tant Que** $continuer = \top$ **et** $i \leq imax$ **:**
  - **Si** $u = 1$ **:**
    - `# C'est gagné !`
    - $continuer \leftarrow \bot$
  - **Sinon**
    - `# Calcul du terme suivant`
    - **Si** $u \equiv 0\ [2]$ **:**
      - $u \leftarrow u/2$; `# Quotient de la division euclidienne.`
    - **Sinon**
      - $u \leftarrow 3u + 1$
    - $i \leftarrow i + 1$
- **Si** $i > imax$ **:**
  - $i \leftarrow (-1)$
- **Renvoyer** i

La rédaction d'un tel algorithme se fait facilement via un code proche de ce que pourrait proposer un langage classique de programmation tel que le `C` *(nous donnons juste après le squelette de la syntaxe propre à* ***`algorithm2e`****)*. Indiquons au passage que le titre s'indique via une option [4] et aussi que sont utilisées des macros additionnelles proposées par `tnsalgo` ainsi que `\NN` et `\ZintervalC` fournies par `tnsmath`.

```
\begin{algo}[title = Suite de Collatz $(u_k)$ -- Conjecture de Syracuse]
  \Data{$n \in \NN$}
  \Result{le premier indice $i \in \ZintervalC{0}{10^5}$ tel que $u_i = 1$
          ou $(-1)$ en cas d'échec}

  \BlankLine  % Pour aérer un peu la mise en forme.

  \Actions{
```

4. Avec `algorithm2e` le titre s'ajoute via la macro `\caption`. Nous verrons que l'option proposée par `tnsalgo` autorise d'avoir sans difficulté un algorithme juste numéroté mais sans titre particulier.

```
    $i, imax \Store 0, 10^5$
    \\
    $u \Store n$
    \\
    $continuer \Store \top$
    \\
    \While{$continuer = \top$ \And $i \leq imax$}{
      \uIf{$u = 1$}{
        \Comment{C'est gagné !}
        $continuer \Store \bot$
      } \Else {
        \Comment{Calcul du terme suivant}
        \uIf{$u \equiv 0 \,\, [2]$}{
          $u \Store u / 2$
          \Comment*{Quotient de la division euclidienne.}
        } \Else {
          $u \Store 3u + 1$
        }
        $i \Store i + 1$
      }
    }
    \If{$i > imax$}{
      $i \Store (-1)$
    }
    \Return{$i$}
  }
\end{algo}
```

Le squelette du code précédent est le suivant. Ceci ressemble bien à du code `C`.

```
\caption{...}

\Data{...}
\Result{...}

\Actions{
    ...
    \While{...}{
        \uIf{...}{
            ...
        } \Else {
            ...
            \uIf{...}{
                ...
            } \Else {
                ...
            }
            ...
        }
    }
    \If{...}{
        ...
    }
```

```
    \Return{...}
}
```

2. Numérotation des algorithmes

Avant de continuer les présentations il faut savoir que tous les algorithmes sont numérotés globalement à l'ensemble du document. C'est plus simple et efficace pour une lecture sur papier.

3. L'environnement `algo`

i. Cadre et largeur

L'option `frame` de l'environnement `algo` demande d'encadrer les algorithmes afin de les rendre plus visibles. Indiquons qu'ici sont utilisées les macros `\NNs` et `\dsum` fournies par le package `tnsmath`.

```
\begin{algo}[
  title = Un truc bidon,
  frame
]
  \Data{$n \in \NNs$}

  \BlankLine

  \Result{$\dsum_{i = 1}^{n} i$}

  \Actions{
    $s \Store 0$
    \\
    \ForRange{i}{1}{n}{
      $s \Store s + i$
    }
    \Return{$s$}
  }
\end{algo}
```

Algorithme 2 : Un truc bidon

Donnée : $n \in \mathbb{N}^*$

Résultat : $\sum_{i=1}^{n} i$

Actions

- $s \leftarrow 0$
- **Pour** i **allant de** 1 **jusqu'à** n **:**
  - $s \leftarrow s + i$
- **Renvoyer** s

L'environnement `algo` propose l'option `scale` pour indiquer la largeur relativement à celle de la ligne et une autre `center` pour centrer l'algorithme qui permettent d'obtenir ce qui suit.

Algorithme 3 : Un truc bidon

Donnée : $n \in \mathbb{N}^*$

Résultat : $\sum_{i=1}^{n} i$

Actions

- $s \leftarrow 0$
- **Pour** i **allant de** 1 **jusqu'à** n **:**
  - $s \leftarrow s + i$
- **Renvoyer** s

Le rendu précédent a été obtenu via le code suivant.

```
\begin{algo}[
  title = Un truc bidon,
  scale = .46,
  center,
  frame
]
  \Data{$n \in \NNs$}

  \BlankLine

  \Result{$\dsum_{i = 1}^{n} i$}

  \Actions{
    $s \Store 0$
    \\
    \ForRange{i}{1}{n}{
      $s \Store s + i$
    }
    \Return{$s$}
  }
\end{algo}
```

ii. Un titre ou pas

Exemple 1 – Algorithme numéroté (par défaut)

```
\begin{algo}
  \Datas{\dots}
  \Result{\dots}
  \Actions{
    \BlankLine\dots
    \\
    \BlankLine
  }
\end{algo}
```

Algorithme 4

Données : ...
Résultat : ...
Actions
└ ...

Exemple 2 – Algorithme numéroté avec un titre

```
\begin{algo}[title = Mon titre]
  \Datas{\dots}
  \Result{\dots}
  \Actions{
    \BlankLine\dots
    \\
    \BlankLine
  }
\end{algo}
```

Algorithme 5 : Mon titre

Données : ...
Résultat : ...
Actions
└ ...

Exemple 3 – Aucun titre

```
\begin{algo}[notitle]
  \Datas{\dots}
  \Result{\dots}
  \Actions{
    \BlankLine\dots
    \\
    \BlankLine
  }
\end{algo}
```

Données : ...
Résultat : ...
Actions
└ ...

4. Des macros pour structurer les algorithmes

i. Entrée / Sortie

Ci-dessous sont données les versions au singulier de mots de type « entrée / sortie ». Toutes les macros présentées ci-après ont une version au pluriel qui s'obtient en rajoutant un `s` à la fin du nom de la macro : par exemple le pluriel de `\In` s'obtient via `\Ins`.

```
\begin{algo}[notitle]
  \In{donnée 1}
  \Out{donnée 2}
\end{algo}
```

Entrée : donnée 1
Sortie : donnée 2

```
\begin{algo}[notitle]
  \Data{donnée 1}
  \Result{donnée 2}
\end{algo}
```

Donnée : donnée 1
Résultat : donnée 2

```
\begin{algo}[notitle]
  \InState{donnée 1}
  \OutState{donnée 2}
\end{algo}
```

État initial : donnée 1
État final : donnée 2

```
\begin{algo}[notitle]
  \PreCond{donnée 1}
  \PostCond{donnée 2}
\end{algo}
```

Précondition : donnée 1
Postcondition : donnée 2

ii. Bloc principal

```
\begin{algo}[notitle]
  % Possibilité 1
  \Actions{Instructions 1}

  % Possibilité 2
  \Begin{Instructions 2}
\end{algo}
```

Actions
└ Instructions 1
Début
└ Instructions 2

iii. Boucles `Tant Que`

```
\begin{algo}[notitle]
  \While{$i \in uneliste$}{
    Instructions 4
  }
\end{algo}
```

Tant Que $i \in uneliste$ **:**
└ Instructions 4

iv. Boucles `Répéter - Jusqu'à Avoir`

```
\begin{algo}[notitle]
  \Repeat{$i \in uneliste$}{
    Instructions
  }
\end{algo}
```

Répéter
| Instructions
Jusqu'à Avoir $i \in uneliste$;

v. Boucles `Pour`

Exemple 1 – Boucles généralistes

```
\begin{algo}[notitle]
  \For{$i \in uneliste$}{
    Instructions 1
  }
\end{algo}
```

Pour $i \in uneliste$ **:**
└ Instructions 1

```
\begin{algo}[notitle]
  \ForAll{$i \in uneliste$}{
    Instructions 2
  }
\end{algo}
```

Pour Tout $i \in uneliste$ **:**
└ Instructions 2

```
\begin{algo}[notitle]
  \ForEach{$i \in uneliste$}{
    Instructions 3
  }
\end{algo}
```

Pour Chaque $i \in uneliste$ **:**
└ Instructions 3

Exemple 2 – Boucles sur des entiers consécutifs

Les boucles présentées ci-dessous passent toutes leurs arguments en mode mathématique.

```
\begin{algo}[notitle]
  \ForRange{a}{0}{12}{
    Instructions 1
  }
\end{algo}
```

Pour a **allant de** 0 **jusqu'à** 12 **:**
└ Instructions 1

```
\begin{algo}[notitle]
  \ForRange*{a}{0}{12}{
    Instructions 1
  }
\end{algo}
```

Pour a **de** 0 **à** 12 **:**
└ Instructions 1

```
\begin{algo}[notitle]
  \ForRange**{a}{0}{12}{
    Instructions 1
  }
\end{algo}
```

Pour $a \in 0\,..\,12$ **:**
└ Instructions 1

Exemple 3 – Boucles spécifiques aux listes

Les boucles présentées ci-après passent elles aussi leurs arguments en mode mathématique.

```
\begin{algo}[notitle]
  \ForInList{e}{L}{
    Instructions 1
  }
\end{algo}
```

Pour e **dans** L **parcourue de gauche à droite :**
└ Instructions 1

```
\begin{algo}[notitle]
  \ForInListRev{e}{L}{
    Instructions 1
  }
\end{algo}
```

Pour e **dans** L **parcourue de droite à gauche :**
└ Instructions 1

vi. Disjonction conditionnelle `Si`

Ci-dessous le préfixe `u`, pour `u`-nclosed soit *« non fermé »* en anglais, demande de ne pas fermer horizontalement le bloc.

```
\begin{algo}[notitle]
  \uIf{$i = 0$}{
    Instructions 1
  }
  \uElseIf{$i = 1$}{
    Instructions 2
  }
  \Else{
    Instructions 3
  }
\end{algo}
```

Si $i = 0$ **:**
| Instructions 1
Sinon Si $i = 1$ **:**
| Instructions 2
Sinon
└ Instructions 3

vii. Disjonction de cas via `Suivant - Cas`

```
\begin{algo}[notitle]
  \Switch{$i$}{
    \uCase{$i = 0$}{Instructions 1}
    \uCase{$i = 1$}{Instructions 2}
    \uCase{$i = 2$}{Instructions 3}
    \Other         {Instructions 4}
  }
\end{algo}
```

Suivant i **:**
- **Cas** $i = 0$ **:** Instructions 1
- **Cas** $i = 1$ **:** Instructions 2
- **Cas** $i = 2$ **:** Instructions 3
- **Autre :** Instructions 4

viii. Citer les structures algorithmiques

Pour faciliter la rédaction de textes sur les algorithmes, des macros standardisent l'impression des noms des outils de structure. On peut ainsi parler de tests **Si** , **Si – Sinon Si – Sinon** et de boucles **Tant Que** ou si l'on préfère de `Si` , `Si - Sinon Si - Sinon` et `Tant Que` avec une police de type True Type à chasse fixe. Ci-dessous, le préfixe `txt` est pour `text` eest la casse est celle des macros utilisables dans en algorithme.

1. Liste des commandes de type « algorithme ».
   a) `\txtIf` donne **Si**.
   b) `\txtIfElse` donne **Si – Sinon**.
   c) `\txtIfElseIf` donne **Si – Sinon – Si**.
   d) `\txtIfElseIfElse` donne **Si – Sinon Si – Sinon**.
   e) `\txtSwitch` donne **Suivant – Cas**.
   f) `\txtFor` donne **Pour**.
   g) `\txtWhile` donne **Tant Que**.
   h) `\txtRepeat` donne **Répéter – Jusqu'à Avoir**.
2. Liste des commandes de type « True Type ».
   a) `\txtIf*` donne `Si`.
   b) `\txtIfElse*` donne `Si - Sinon`.
   c) `\txtIfElseIf*` donne `Si - Sinon - Si`.
   d) `\txtIfElseIfElse*` donne `Si - Sinon Si - Sinon`.
   e) `\txtSwitch*` donne `Suivant - Cas`.
   f) `\txtFor*` donne `Pour`.
   g) `\txtWhile*` donne `Tant Que`.
   h) `\txtRepeat*` donne `Répéter - Jusqu'à Avoir`.

5. Affectations simples ou multiples

i. Affectation simple avec une flèche

```
$x \Store 3$ ou
$3 \PutIn x$
```

$x \leftarrow 3$ ou $3 \rightarrow x$

ii. Affectation simple avec un signe égal décoré

Deux notations alternatives existent [5]. Voici comment les obtenir.

`$x \Store* 3$` ou `$x \Store** 3$`	$x \mathrel{:=} 3$ ou $x \triangleq 3$

iii. Affectations multiples en parallèle

`$a, b, c \MStore x, y, z$` ou `$x, y, z \MPutIn a, b, c$`	$a, b, c \Lleftarrow x, y, z$ ou $x, y, z \Rightarrow a, b, c$

Remarque. La multi-affectation se faisant en parallèle, le résultat de $a, b, c \Lleftarrow 2, a + b, c - b$ ne sera pas semblable à celui de $a \leftarrow 2$ suivi de $b \leftarrow a + b$ puis de $c \leftarrow c - b$ car dans le second cas les variables a et b évoluent avant de nouvelles affectations simples. En fait la multi-affectation précédente correspond aux actions suivantes.

Algorithme 6 : Comment $a, b, c \Lleftarrow 2, a + b, c - b$ fonctionne-t-il ?

$a_{memo} \leftarrow a$; $b_{memo} \leftarrow b$; $c_{memo} \leftarrow c$
$a \leftarrow 2$
$b \leftarrow a_{memo} + b_{memo}$
$c \leftarrow c_{memo} - b_{memo}$

6. Intervalles discrets d'entiers

Il est d'usage en informatique théorique de poser $4\,..\,7 = \{4; 5; 6; 7\}$ où $4\,..\,7$ a été obtenu en tapant `\CSinterval{4}{7}`. Le préfixe `CS` fait référence à `Computer Science` soit *« Informatique Théorique »* en anglais.

7. Listes

Avant de commencer il faut savoir que la convention retenue pour la numérotation des indices des listes est de commencer à 1 *(ceci est plus naturel pour un humain)*.

i. Opérations de base.

Exemple 1 – Longueur d'une liste

`$\Len(L)$`	$\text{taille}(L)$

Exemple 2 – Liste vide et liste en extension

`$\EmptyList$`
 	
`$\List{4 ; 7 ; -1}$`
	$[\,]$
	$[\,4; 7; -1\,]$

5. La 2e notation vient du langage B qui permet de spécifier et prouver des programmes.

Exemple 3 – Extraire certains éléments

```
k\ieme{} élément :
$\ListElt{L}{k}$

Du 1\ier{} au k\ieme{} :
$\ListUntil{L}{k}$

À partir du k\ieme{} :
$\ListFrom{L}{k}$
```

k^{e} élément : $L\,[\,k\,]$
Du 1er au k^{e} : $L\,[\,..\,k\,]$
À partir du k^{e} : $L\,[\,k\,..\,]$

Exemple 4 – Concaténer des listes

```
$L =          \ListUntil{L}{k-1}
     \AddList \ListElt{L}{k}
     \AddList \ListFrom{L}{k+1}$
```

$$L = L\,[\,..\,k-1\,] \boxplus L\,[\,k\,] \boxplus L\,[\,k+1\,..\,]$$

ii. Modifier une liste avec un élément

Exemple 1 – Versions textuelles

`append` et `prepend` signifient *« ajouter »* et *« préfixer »* en anglais. Quant à `pop`, il signifie *« éclater »*.

```
Ex.1 : \Append{L}{x}

Ex.2 : \Prepend{L}{y}

Ex.3 : \PopAt{L}{3}
```

Ex.1 : Ajouter le nouvel élément x après la fin de la liste L.
Ex.2 : Ajouter le nouvel élément y avant le début de la liste L.
Ex.3 : Élément à la position 3 dans la liste L, cet élément étant retiré de la liste.

Exemple 2 – Versions POO

Voici des notations à la fois concises et aisées à comprendre [6] avec une syntaxe de type POO [7].

```
Ex.1 : \Append*{L}{x}

Ex.2 : \Prepend*{L}{y}

Ex.3 : \PopAt*{L}{3}
```

Ex.1 : $L\,.\,\text{ajouter-droite}(x)$
Ex.2 : $L\,.\,\text{ajouter-gauche}(y)$
Ex.3 : $L\,.\,\text{extraire}(3)$

Exemple 3 – Versions symboliques

Pour finir il est possible d'utiliser des notations symboliques qui sont très efficaces lorsque l'on rédige les algorithmes à la main [8]. Notez au passage qu'ici de petits calculs automatiques facilitent la rédaction et surtout que la macro `\PopAt**` prend un argument de plus que les macros `\PopAt` et `\PopAt*`, cet argument étant pour indiquer où stocker l'élément retiré de la liste.

6. L'opérateur point `.` est défini dans la macro `\POOpoint`. Ceci permet de personnaliser facilement cet opérateur.
7. « POO » est l'acronyme de « Programmation Orientée Objet ».
8. Rappelons que l'opérateur $\boxplus$ est défini dans la macro `\AddList`.

`Ex.1 : \Append**{L}{x}`
 	
`Ex.2 : \Prepend**{L}{y}`
 	
`Ex.3 : \PopAt**{e}{L}{3}`
	Ex.1 : $L \leftarrow L \boxplus [\,x\,]$
	Ex.2 : $L \leftarrow [\,y\,] \boxplus L$
	Ex.3 : $e, L \leftleftarrows L\,[\,3\,], L\,[\,..\,2\,] \boxplus L\,[\,4..\,]$

Remarque. Voici des points importants à connaître.

1. `\PopAt**{e}{L}{1}` produit $e, L \leftleftarrows L\,[\,1\,], L\,[\,2..\,]$ sans écrire $L\,[\,..\,0\,] \boxplus L\,[\,2\,..\,]$ puisque pour le package les indices des listes commencent toujours à 1.
2. L'écriture symbolique $e, L \leftleftarrows L\,[\,k\,], L\,[\,..\,k-1\,] \boxplus L\,[\,k+1..\,]$ s'obtient tout simplement via `\PopAt**{e}{L}{k}`.
3. Par contre `\PopAt**{e}{L}{k-1}` aboutit à $e, L \leftleftarrows L\,[\,k-1\,], L\,[\,..\,k-1-1\,] \boxplus L\,[\,k-1+1..\,]$ ce qui n'est pas joli ! Dans ce cas, taper `\PopAt**{e}{L}{k-2 | k-1 | k}` aide la macro à produire $e, L \leftleftarrows L\,[\,k-1\,], L\,[\,..\,k-2\,] \boxplus L\,[\,k\,..\,]$.

iii. Modifier une liste en extrayant des sous-listes

Exemple 1

`\KeepLR` vient de « keep left and right » soit *« garder à droite et à gauche »* en anglais.

`\KeepLR{L}{a}{b}`	$L \leftarrow L\,[\,..\,a\,] \boxplus L\,[\,b\,..\,]$

Exemple 2

`\KeepL{L}{a}`
 	
`\KeepR{L}{b}`
	$L \leftarrow L\,[\,..\,a\,]$
	$L \leftarrow L\,[\,b\,..\,]$

8. Diverses commandes

Pour finir voici un ensemble de mots supplémentaires qui pourront vous rendre service. Le préfixe `m` permet d'utiliser des versions « masculinisées » des textes proposés.

Exemple 1 – Opérateurs booléens

`A \And B \Or C`	A **et** B **ou** C

Exemple 2 – Interface

`\Return RÉSULTAT`
 	
`\Ask "Quelque chose"`
 	
`\Print "Quelque chose"`
	Renvoyer RÉSULTAT
	Demander "Quelque chose"
	Afficher "Quelque chose"

Exemple 3 – Itérations sur des entiers consécutifs

`$k$ \From $1$ \To $n$`	k **de** 1 **à** n
`$k$ \ComingFrom $1$ \GoingTo $n$`	k **allant de** 1 **jusqu'à** n

Exemple 4 – Itération sur un ensemble

`$e$ \InThis $\{ 1 , 4 , 16 \}$`	e **dans** $\{1, 4, 16\}$

Exemple 5 – Itération sur une liste

`LtoR` est pour « Left to Right » soit *« de droite à gauche »* en anglais, et `RtoL` est pour « Right to Left ».

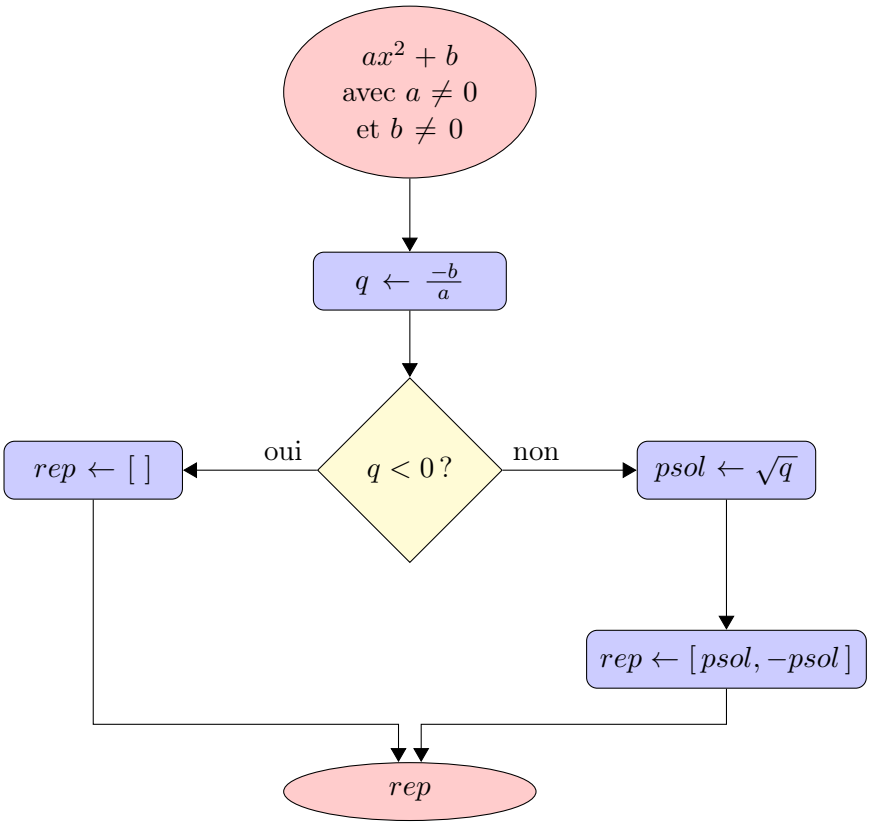
`$L$ \LtoR`	L **parcourue de gauche à droite**
`$L$ \mLtoR`	L **parcouru de gauche à droite**
`$L$ \RtoL`	L **parcourue de droite à gauche**
`$L$ \mRtoL`	L **parcouru de droite à gauche**

V. Ordinogrammes

1. C'est quoi un ordinogramme

Les ordinogrammes [9] sont des diagrammes que l'on peut utiliser pour expliquer des algorithmes très simples [10].

Voici un exemple expliquant comment résoudre dans $\mathbb{R}$ l'équation en x $ax^2 + b = 0$ lorsque $a \neq 0$ et $b \neq 0$: le code utilisé est donné plus tard dans la section 6. *(ce code sera très aisé à comprendre une fois lues les sections à venir).*



2. L'environnement `algochart`

Tous les codes seront placés dans l'environnement `algochart` qui pour le moment est juste un alias de l'environnement `tikzpicture` proposé par `TikZ` qui fait le principal du travail [11]. `tnsalgo` définit juste quelques styles et quelques macros pour faciliter la saisie des ordinogrammes afin de travailler efficacement avec les macros `\node` et `\path` proposées par `TikZ`.

3. Convention pour les noms des styles et des macros

Toutes les fonctionnalités proposées par `tnsalgo` seront nommées en minuscule en utilisant toujours le préfixe `ac` pour `a`-lgorithmic `c`-hart soit *« diagramme algorithmique »* en anglais.

9. Le mot « ordinogramme » vient des mots « ordinateur », du latin « ordinare » soit « mettre en ordre », et du grec ancien « gramma » soit « lettre, écriture ».

10. Cet outil pédagogique montre très vite ses limites. Essayez par exemple de tracer un ordinogramme pour expliquer comment résoudre une équation du 2e degré.

11. `algochart` vient de la contraction de « algorithmic » et « flowchart » soit « algotithmique » et « diagramme » en anglais.

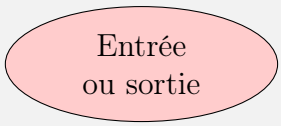
4. Les styles proposés

À la norme officielle, nous avons préféré un style plus percutant où les formes des cadres sont bien différenciées.

i. Entrée et sortie

L'entrée et la sortie de l'algorithme sont représentés par des ovales comme dans l'exemple ci-après. Par convention, l'entrée se situe tout en haut de l'ordinogramme, et la sortie tout en bas. Indiquons que `io` dans `acio` fait référence à « input / output » soit « entrée / sortie » en anglais.

```
\begin{algochart}
  \node[acio] {Entrée ou sortie};
\end{algochart}
```



Dans le code ci-dessus, nous utilisons le style `acio` en l'indiquant entre des crochets. La machinerie `TikZ` permet de changer localement un réglage comme ci-après où l'on modifie la largeur de l'ellipse pour n'avoir qu'une seule ligne de texte.

```
\begin{algochart}
  \node[acio, text width = 8em]
  {Entrée ou sortie};
\end{algochart}
```

ii. Les instructions

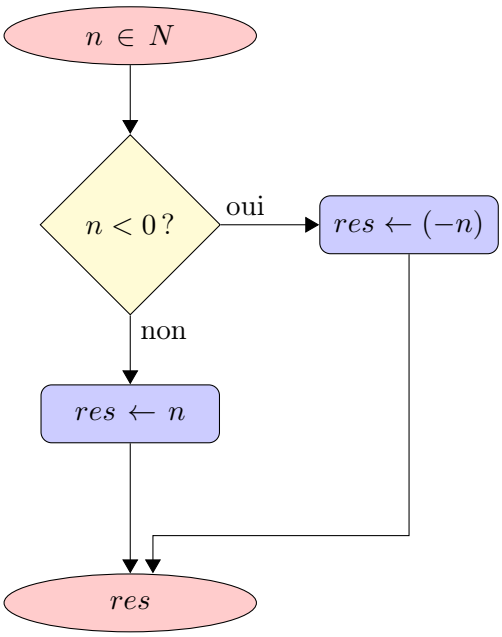
Dans l'exemple suivant, qui ne nécessite aucun commentaire [12], nous avons dû régler à la main la largeur du cadre pour ne pas avoir un retour à la ligne.

```
\begin{algochart}
  \node[acinstr, text width = 8em]
  {Une instruction};
\end{algochart}
```

iii. Les tests conditionnels via un exemple complet

Nous allons voir comment obtenir le résultat suivant qui contient une structure conditionnelle. Nous allons en profiter pour expliquer comment placer les noeuds les uns par rapport aux autres, et aussi voir comment ajouter des connexions via la macro `\path` proposée par `TikZ`.

12. Chercher l'erreur...



Voici le code utilisé pour obtenir l'ordinogramme ci-dessus.

```
\begin{algochart}
  % Placement des noeuds.
  \node[acio]
  (input) {$n \in N$};

  \node[acif, below of = input]
  (is-neg) {$n < 0$ ?};

  \node[acinstr, right] at ($(is-neg) + (2.5,0)$)
  (neg) {$res \Store (-n)$};

  \node[acinstr, below of = is-neg]
  (not-neg) {$res \Store n$};

  \node[acio, below of = not-neg]
  (output) {$res$};

  % Ajout des connexions.
  \path[aclink] (input) -- (is-neg);

  \path[aclink] (is-neg) -- (neg) \aclabelabove{oui};
  \path[aclink] (is-neg) -- (not-neg) \aclabelright{non};

  \path[aclink] (not-neg) -- (output);
  \path[aclink] (neg) to[aczigzag] (output);
\end{algochart}
```

Donnons des explications sur les points délicats du code précédent.

1. `\node[acio] (input) {$n \in \NN$}`

   Ici on définit `input` comme alias du noeud via `(input)`, un alias utilisable ensuite pour différentes actions graphiques.

2. `\node[acif, below of = input] (is-neg) {$n < 0$ ?}`

   Ici on demande de placer le noeud nommé `is-neg` sous celui nommé `input` via `below of = input` où « below of » se traduit par *« en dessous de »* en anglais. Attention au signe égal dans `below of = input`.

3. `\node[acinstr, right] at ($(is-neg) + (2.5,0)$) (neg) {$res \Store (-n)$}`

   Dans cette commande un peu plus mystique, l'emploi de `right` indique de se placer à droite du dernier noeud. Vient ensuite la cabalistique instruction `at ($(is-neg) + (2.5,0)$)`. Comme « at » signifie *« à (tel endroit) »* en anglais, on comprend que l'on demande de placer le noeud à une certaine position. Il faut alors savoir que pour `TikZ` l'usage de `($...$)` indique de faire un calcul qui ici est celui d'addition de coordonnées cartésiennes via `(is-neg) + (2.5,0)`.

4. `\path[aclink] (input) -- (is-neg)`

   Cette instruction plus simple demande de tracer, avec le style `aclink`, un trait entre les noeuds `input` et `is-neg`.

5. `\path[aclink] (is-neg) -- (neg) \aclabelabove{oui}`

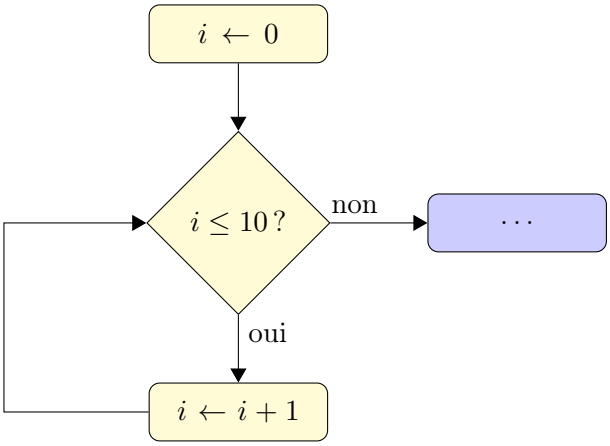
   La nouveauté ici est l'utilisation de la macro `\aclabelabove{oui}` proposée par `tnsalgo` pour placer du texte au début et au dessus de la connexion. Rappelons que « above » signifie *« au-dessus »* en anglais.

6. `\path[aclink] (neg) to[aczigzag] (output);`

   Cette instruction permet d'obtenir « l'orthopolyligne »[13] de $\boxed{res \leftarrow (-n)}$ vers $\boxed{res}$. Vous pouvez utiliser `to[aczigzag = {xstart = ... , x = ... , y = ....}]` pour des réglages personnalisés. Par défaut les clés optionnelles suivent le réglage `xstart = 0mm` , un décalage au début de la connexion , `x = 3mm` , un décalage à la fin de la connexion, et `y = 5mm`, un autre décalage à la fin de la connexion.

iv. Les boucles

L'exemple très farfelu qui suit montre comment dessiner une petite boucle en faisant ressortir les instructions liées au fonctionnement de la boucle *(cet effet est impossible à obtenir en mode noir et blanc : voir la section 5. à ce sujet)*. On voit au passage la limite d'utilisabilité des ordinogrammes car ces derniers ne proposent pas de mise en forme efficace pour les boucles.



Voici le code que nous avons utilisé. Les seules vraies nouveautés sont l'utilisation du style `acifinstr` pour mieux visualiser les instructions liées à la boucle, et l'usage de `to[acbackloopleft]` pour la connexion en retour arrière par la gauche. Vous pouvez utiliser `to[acbackloopleft = {x = ... }]`

13. Un néologisme ?

pour régler le décalage vers la gauche. Par défaut, `x = 5em`. De façon analogue, on peut utiliser `to[acbackloopright]` pour un retour arrière par la droite.

```
\begin{algochart}
  % Placement des noeuds.
  \node[acifinstr]
  (loop-init) {$i \Store 0$};

  \node[acif, below of = loop-init]
  (loop-test) {$i \leq 10$ ?};

  \node[acifinstr, below of = loop-test]
  (loop-next) {$i \Store i+1$};

  \node[acinstr, right] at ($(loop-test) + (2.5,0)$)
  (loop-out) {\dots};

  % Ajout des connexions.
  \path[aclink] (loop-init) -- (loop-test);

  \path[aclink] (loop-test) -- (loop-out) \aclabelabove{non};

  \path[aclink] (loop-test) -- (loop-next) \aclabelright{oui};
  \path[aclink] (loop-next) to[acbackloopleft] (loop-test);
\end{algochart}
```

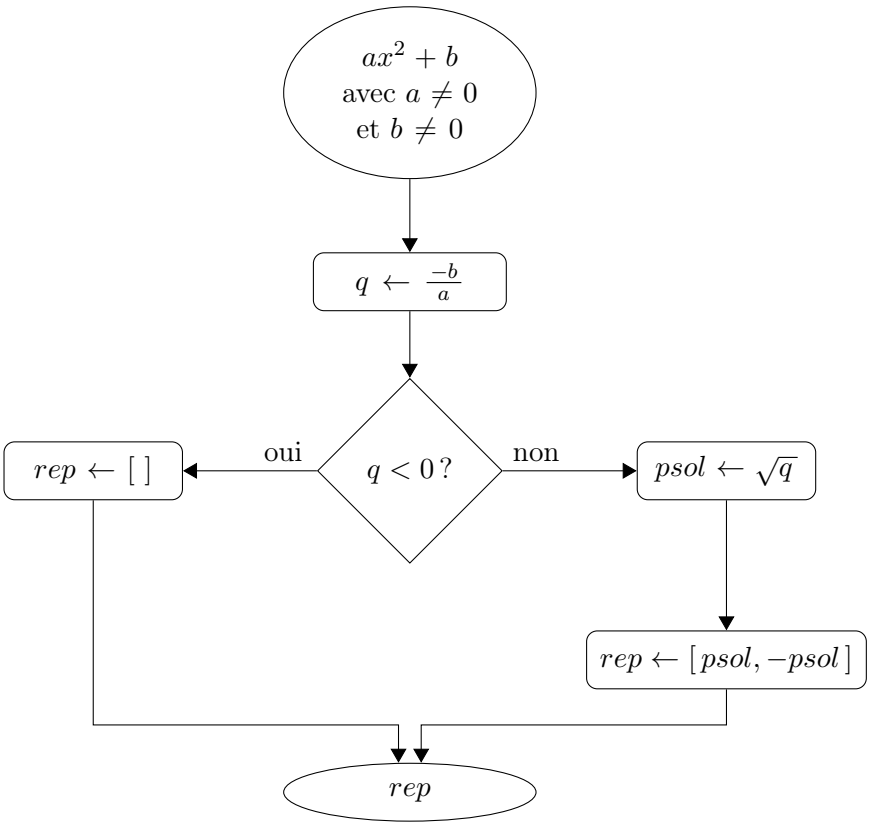
5. Passer de la couleur au noir et blanc, et vice versa

Pour l'impression papier, n'avoir que du noir et blanc peut rende service. Les commandes `\acusebw` et `\acusecolor` permettent d'avoir du noir et blanc ou de la couleur pour tous les ordinogrammes qui suivent l'utilisation de ces macros.

```
\begin{algochart}
  % Passons au noir et blanc.
  \acusebw
  \node[acinstr] {Instruction};
  % Retour à la couleur pour la suite.
  \acusecolor
\end{algochart}
```

6. Code du tout premier exemple

Nous (re)donnons la version noir et blanc de l'ordinogramme présenté au début de la section 1..



Ce diagramme s'obtient via le code suivant.

```
\begin{algochart}
  % Placement des noeuds.
  \node[acio]
  (input) {$a x^2 + b$ avec $a \neq 0$ et $b \neq 0$};

  \node[acinstr, below of = input, text width =  6em]
  (square) {$q \Store \frac{-b}{a}$};

  \node[acif, below of = square]
  (is-square-neg) {$q < 0$ ?};

  \node[acinstr,left] at ($(is-square-neg) + (-3,0)$)
  (no-sol) {$rep \Store \EmptyList$};

  \node[acinstr,right] at ($(is-square-neg) + (3,0)$)
  (pos-sol) {$psol \Store \sqrt{q}$};

  \node[acinstr, below of = pos-sol, text width = 9em]
  (all-sol) {$rep \Store \List{psol , - psol}$};

  \node[acio, below of = is-square-neg] at ($(is-square-neg) + (0,-1.75)$)
  (output) {$rep$};

  % Ajout des connexions.
  \path[aclink] (input) -- (square);
  \path[aclink] (square) -- (is-square-neg);

  \path[aclink] (is-square-neg) -- (no-sol) \aclabelabove{oui};
```

```
  \path[aclink] (is-square-neg) -- (pos-sol) \aclabelabove{non};

  \path[aclink] (pos-sol) -- (all-sol);

  \path[aclink] (all-sol) to[aczigzag={x=1.5mm}] (output);
  \path[aclink] (no-sol) to[aczigzag={x=-1.5mm}] (output);
\end{algochart}
```

VI. Historique

Nous ne donnons ici qu'un très bref historique récent [14] de `tnsalgo` à destination de l'utilisateur principalement. Tous les changements sont disponibles uniquement en anglais dans le dossier `change-log` : voir le code source de `tnsalgo` sur `github`.

2020-09-12 Première version `0.0.0-beta`.

14. On ne va pas au-delà de un an depuis la dernière version.

VII. Toutes les fiches techniques

1. Algorithmes en langage naturel

i. L'environnement `algo`

```
\begin{algo}[#opt]
   ...
\end{algo}
```

— `Option:` on utilise un système clé/valeur.

1. `notitle` permet de ne pas avoir de titre. Par défaut `notitle = false`.
2. `title` est le titre complémentaire. Par défaut `title = {}`.
3. `frame` demande d'ajouter un cadre.
4. `center` sert à centrer le contenu.
5. `scale` donne le coefficient multiplicateur à appliquer à la largeur de la ligne. Par défaut `scale = 1`.

ii. Des macros pour structurer les algorithmes

```
\In       {#1}
\Ins      {#1}
\Out      {#1}
\Outs     {#1}
\Data     {#1}
\Datas    {#1}
\Result   {#1}
\Results  {#1}
\InState  {#1}
\InStates {#1}
\OutState {#1}
\OutStates{#1}
\PreCond  {#1}
\PreConds {#1}
\PostCond {#1}
\PostConds{#1}
```

— `Argument:` du texte décrivant une ou des entrées, une ou des sorties, ...

```
\Actions{#1}
\Begin  {#1}
```

— `Argument:` le bloc des instructions via la syntaxe proposée par `algorithm2e`.

```
\ForAll {#1..#2}
\ForEach{#1..#2}
```

— `Argument 1:` les éléments à boucler.

— `Argument 2:` le bloc des instructions via la syntaxe proposée par `algorithm2e`.

`\ForRange  {#1..#4}`
`\ForRange* {#1..#4}`
`\ForRange**{#1..#4}`

— `Argument 1:` la variable de la boucle.

— `Argument 2:` la 1er valeur entière.

— `Argument 3:` la dernière valeur entière.

— `Argument 4:` le bloc des instructions via la syntaxe proposée par `algorithm2e`.

`\ForInList   {#1..#3}`
`\ForInListRev{#1..#3}`

— `Argument 1:` la variable de la boucle.

— `Argument 2:` la liste.

— `Argument 3:` le bloc des instructions via la syntaxe proposée par `algorithm2e`.

`\ForInList   {#1..#3}`
`\ForInListRev{#1..#3}`

— `Argument 1:` la variable de la boucle.

— `Argument 2:` la liste.

— `Argument 3:` le bloc des instructions via la syntaxe proposée par `algorithm2e`.

`\txtIf`
`\txtIf*`
`\txtIfElse`
`\txtIfElse*`
`\txtIfElseIf`
`\txtIfElseIf*`
`\txtIfElseIfElse`
`\txtIfElseIfElse*`
`\txtSwitch`
`\txtSwitch*`

`\txtFor`
`\txtFor*`
`\txtWhile`
`\txtWhile*`
`\txtRepeat`
`\txtRepeat*`

iii. Affectations simples ou multiples

`\Store`
`\Store*`
`\Store**`

`\PutIn`

`\MStore`
`\MPutIn`

iv. Intervalles discrets d'entiers

`\CSinterval{#1..#2}` `CS = C-omputer S-cience`

— `Argument 1:` la borne entière inférieure.

— `Argument 2:` la borne entière supérieure.

v. Listes

`\Len`

`\EmptyList`

`\List{#1}`

— `Argument:` le contenu explicite de la liste.

`\ListElt{#1..#2}`

— `Argument 1:` le nom de la liste.

— `Argument 2:` le rang de l'élément à lire.

`\ListUntil{#1..#2}`

— `Argument 1:` le nom de la liste.

— `Argument 2:` le rang du dernier élément de la sous-liste extraite.

`\ListFrom{#1..#2}`

— `Argument 1:` le nom de la liste.

— `Argument 2:` le rang du premier élément de la sous-liste extraite.

`\Append  {#1..#2}`
`\Append* {#1..#2}`
`\Append**{#1..#2}`

— `Argument 1:` le nom de la liste.

— `Argument 2:` élément à ajouter après la fin de la liste.

`\Prepend  {#1..#2}`
`\Prepend* {#1..#2}`
`\Prepend**{#1..#2}`

— `Argument 1:` le nom de la liste.

— `Argument 2:` élément à ajouter avant le début de la liste.

`\PopAt {#1..#2}`
`\PopAt*{#1..#2}`

— `Argument 1:` le nom de la liste.

— `Argument 2:` indice de l'élément à retirer.

`\PopAt**{#1..#3}`

— `Argument 1:` variable où stocker l'élément retiré.

— `Argument 2:` le nom de la liste.

— `Argument 3:` indice de l'élément à retirer.

`\KeepLR{#1..#3}`

— `Argument 1:` le nom de la liste.

— `Argument 2:` indice où commencer l'extraction.

— `Argument 3:` indice où finir l'extraction.

`\KeepL{#1..#2}`

— `Argument 1:` le nom de la liste.

— `Argument 2:` indice où finir l'extraction.

`\KeepR{#1..#2}`

— `Argument 1:` le nom de la liste.

— `Argument 2:` indice où commencer l'extraction.

vi. Diverses commandes

`\And`
`\Or`

`\Ask`
`\Print`
`\Return`

\From
\To
\ComingFro
\GoingTo

\InThis

\LtoR LtoR = L-eft to R-ight
\mLtoR m = m-asculine
\RtoL LtoR = R-ight to L-eft
\mRtoL

2. Ordinogrammes

i. Code du tout premier exemple

```
\begin{algochart}
   ...
\end{algochart}
```

Cet environnement s'utilise avec les styles `TikZ` suivants.

1. `acio` pour un noeud indiquant l'entrée ou la sortie de l'algorithme.
2. `acinstr` pour un noeud indiquant une instruction.
3. `acif` pour un noeud indiquant un test.
4. `aclink` pour un chemin liant deux noeuds.
5. `acbackloopleft` à utiliser avec `to` pour une boucle de rétroaction par la gauche entre deux noeuds. Ce style peut recevoir l'argument optionnel `x` permet d'indiquer le décalage vers la gauche. Par défaut `x = 5em`.
6. `acbackloopright` à utiliser avec `to` pour une boucle de rétroaction par la droite entre deux noeuds. Ce style peut recevoir l'argument optionnel `x` permet d'indiquer le décalage vers la droite. Par défaut `x = 5em`.
7. `aczigzag` à utiliser avec `to` pour un chemin » cassé à un angle droit ». Ce style peut recevoir les arguments optionnels suivants.
   (a) `xstart` permet d'indiquer un décalage horizontal au début. Par défaut `xstart = 0mm`.
   (b) `x` permet d'indiquer un décalage horizontal à la fin. Par défaut `x = 3mm`.
   (c) `y` permet d'indiquer un décalage vertical à la fin. Par défaut `y = 5mm`.

\aclabelabove{#1}
\aclabelbelow{#1}
\aclabelleft{#1}
\aclabelright{#1}

— `Argument:` le texte au début du chemin.

\acusebw bw = b-lack & w-hite
\acusecolor