Algorithmique I

L1 et PEIP2 A Maths – Université Clermont Auvergne

Responsable de l'UE :

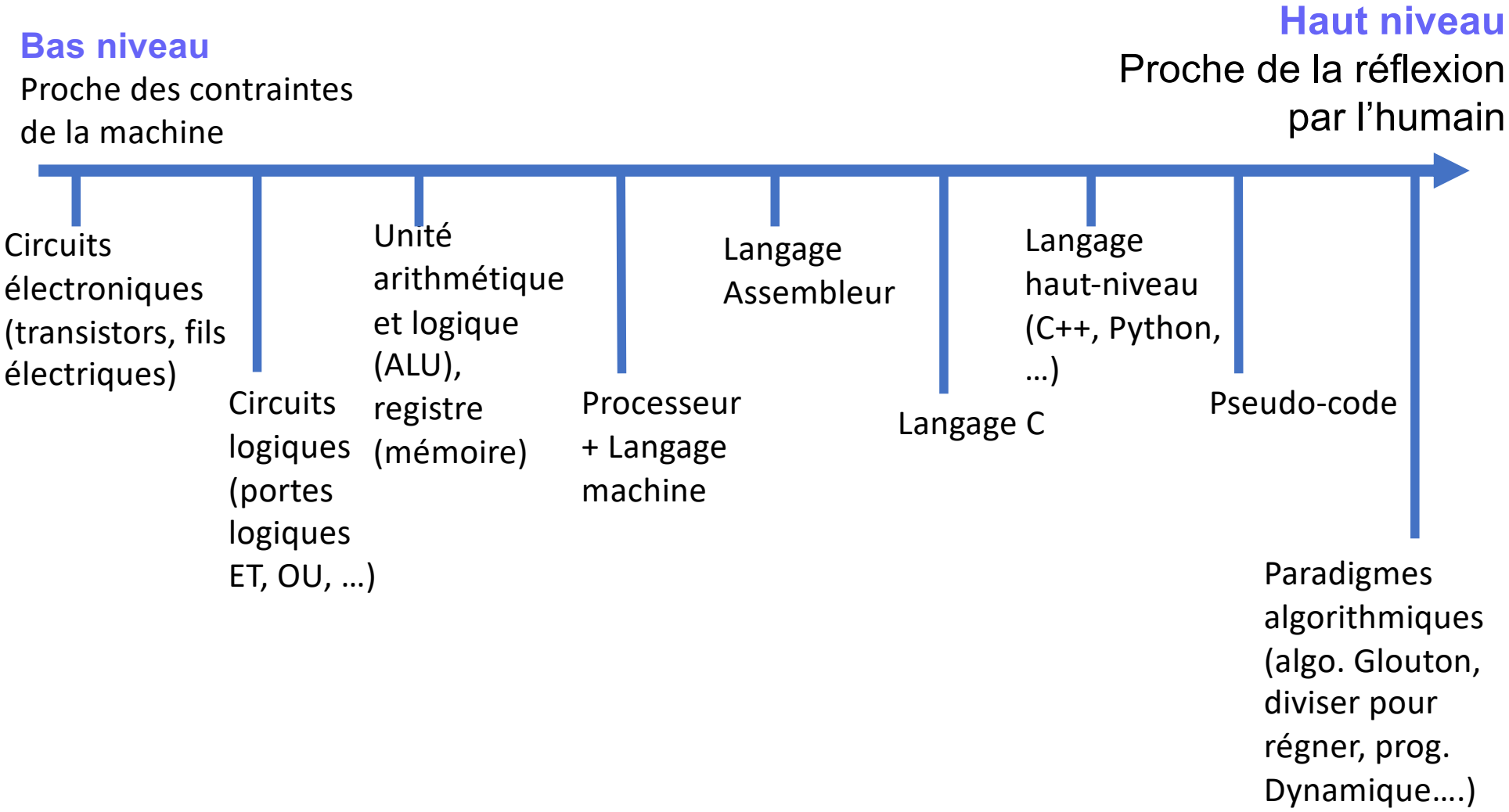
Aurélie Lagoutte

Un petit tour d'horizon

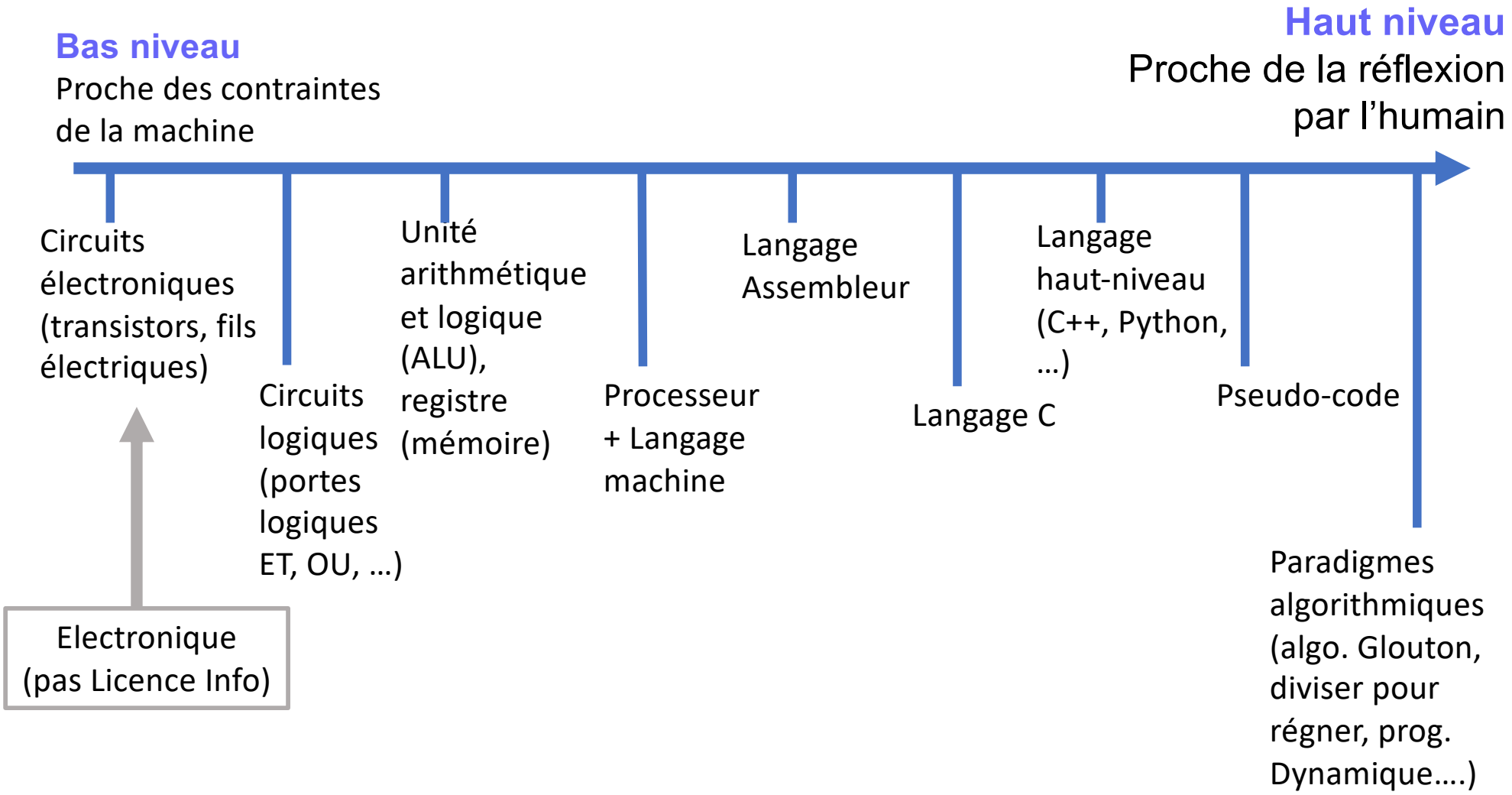
Replaçons ce cours d'Algorithmique 1 dans son contexte:

- Suite logique d'Intro à l'algorithmique (en pseudo-code) et de Programmation en C
- Et sur la "carte mentale" de l'informatique … ?

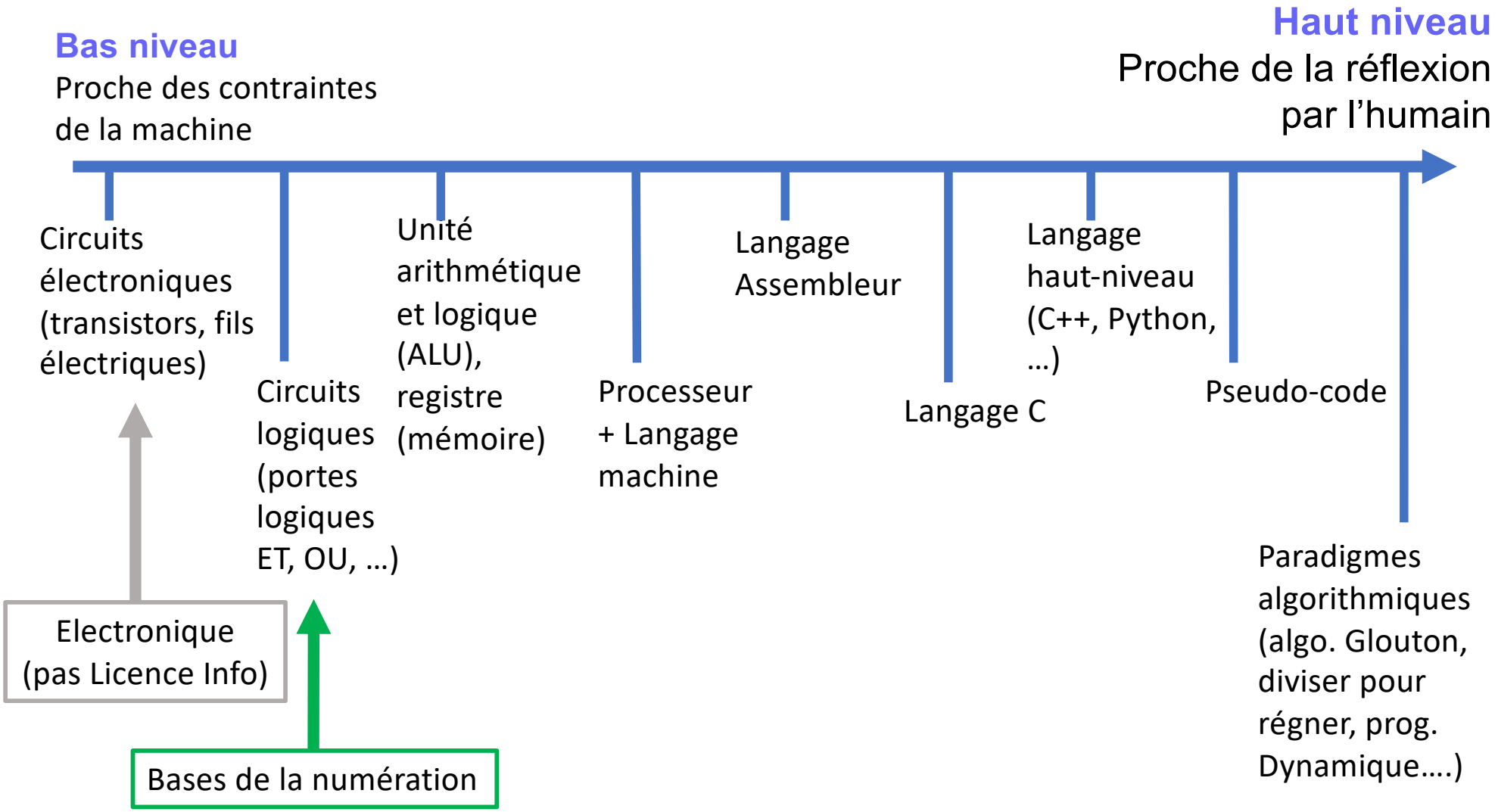
Bas niveau/Haut niveau



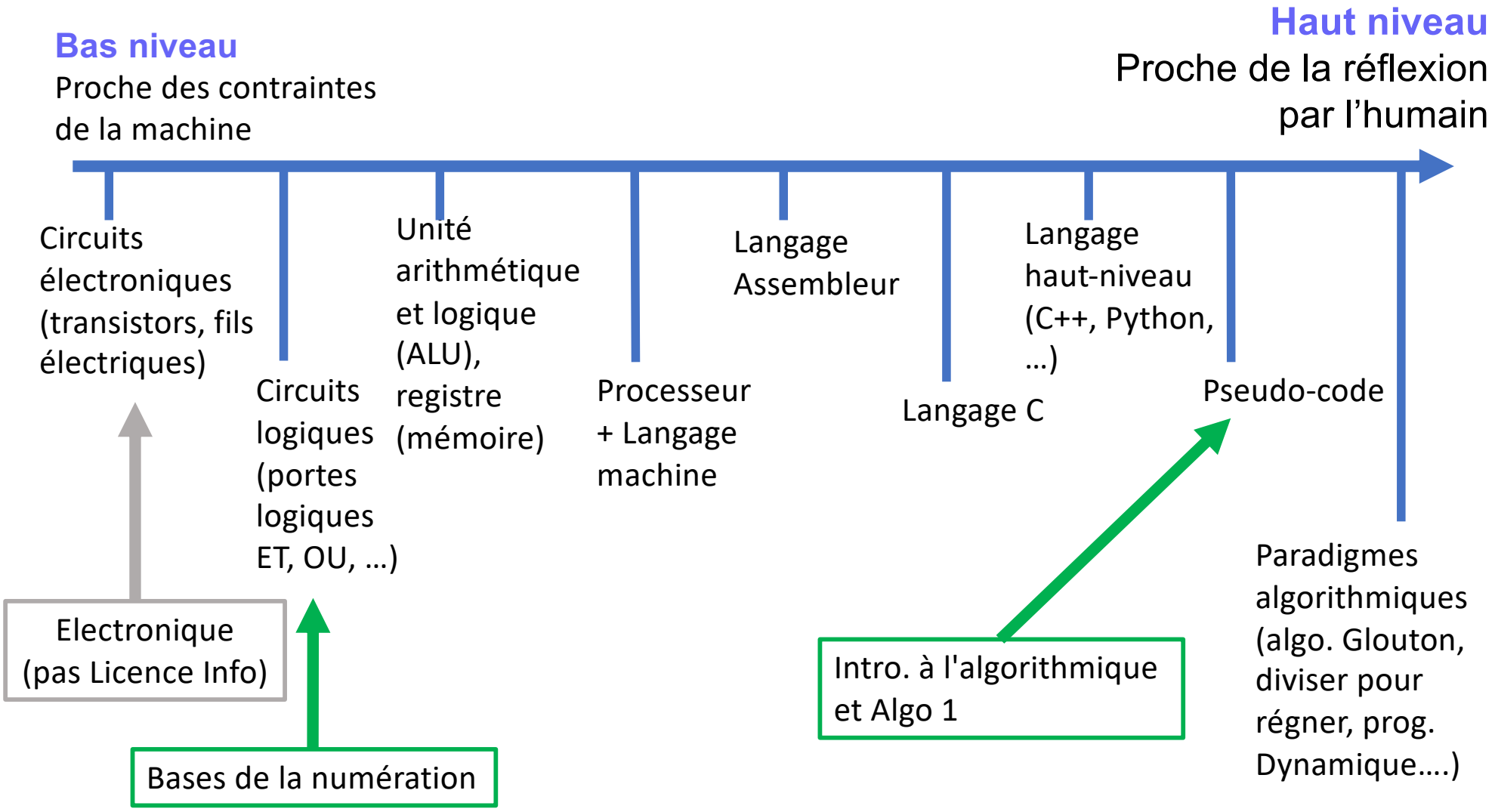
Bas niveau/Haut niveau



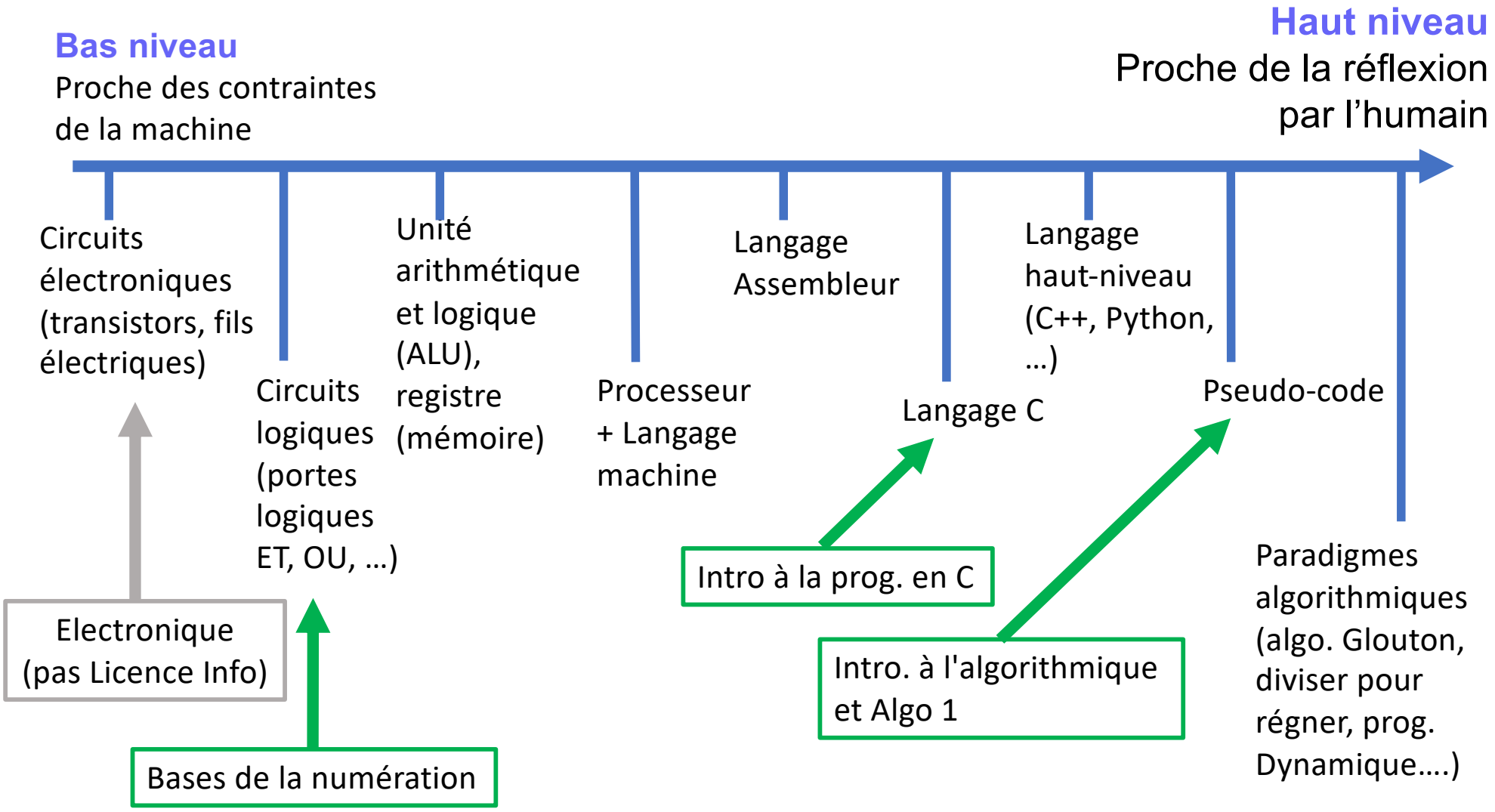
Bas niveau/Haut niveau



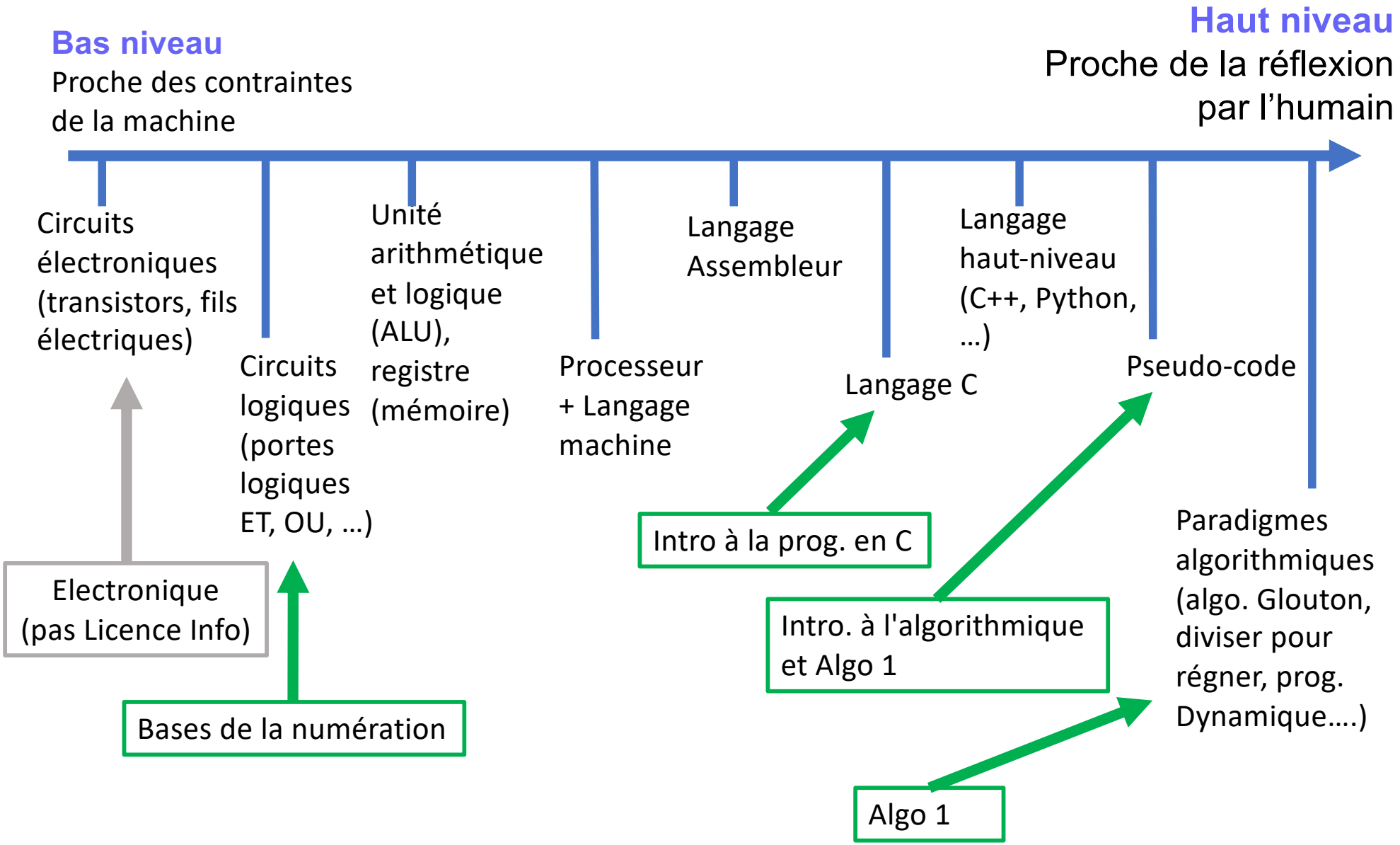
Bas niveau/Haut niveau



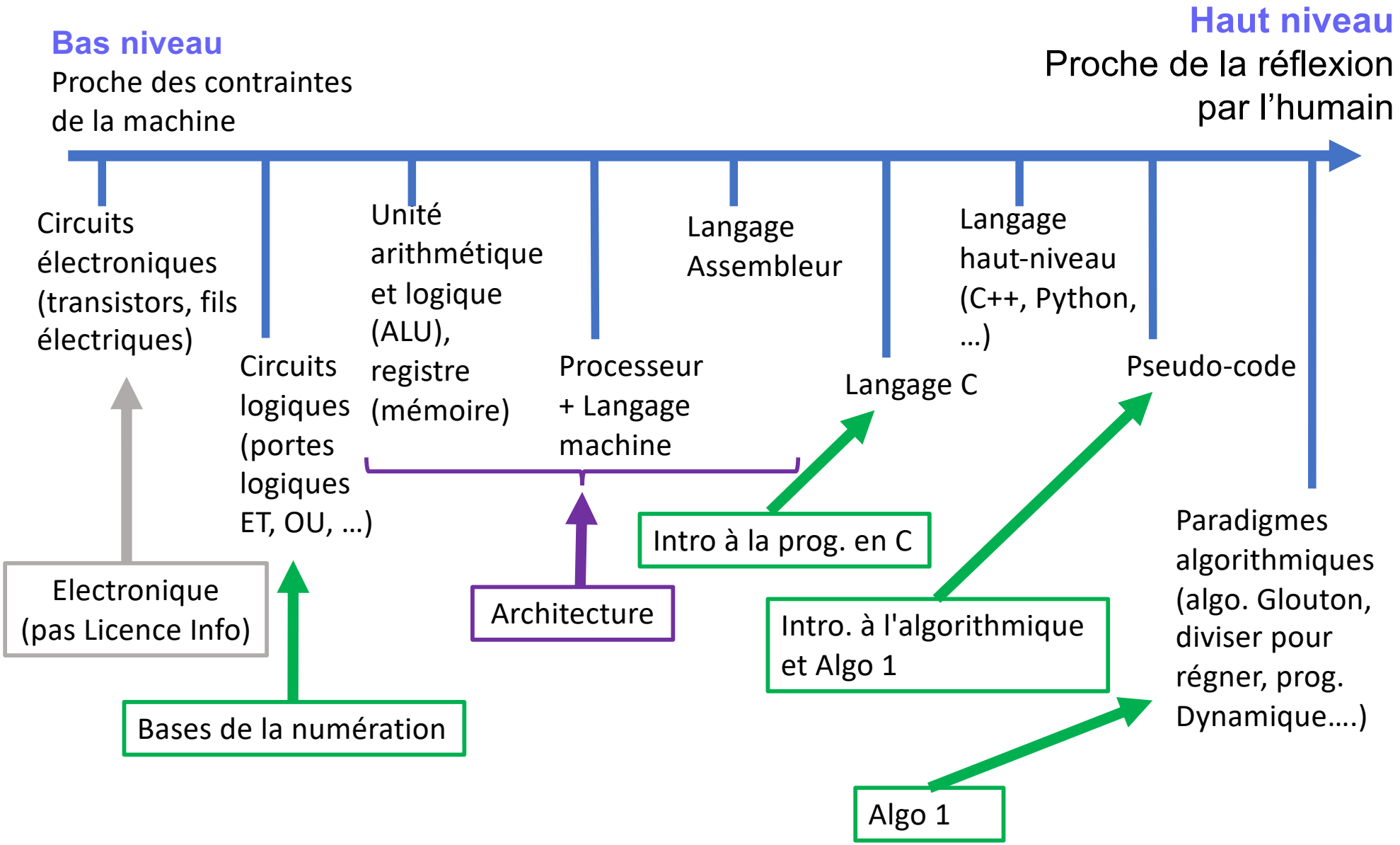
Bas niveau/Haut niveau



Bas niveau/Haut niveau



Bas niveau/Haut niveau



Bibliographie

- *Introduction à l'algorithmique*, T. Cormen, C. Leiserson, R. Rivest

(parfois aussi appelé *Algorithmique* ou surnommé *"le Cormen"*)

- Site OpenClassroom, cours Algorithmique pour l'Apprenti Programmeur

https://openclassrooms.com/courses/algorithmique-pour-l-apprenti-programmeur

Chapitre 1 :
rappels du cours Intro à l'Algo

1) Algorithmique vs. programmation

2) "Syntaxe" du pseudo-code pour ce cours

3) Qu'est qu'un "bon" algorithme?

1) Algorithmique et programmation

- L'**algorithmique** et la **programmation** sont différents
  - L'algorithmique permet de passer **du problème à une méthode de résolution**
  - La programmation permet d'implémenter (= mettre en place) une méthode de résolution dans un **programme concret**

Exemple: défendre votre point de vue sur le réchauffement climatique en anglais

Structurer votre argumentaire au brouillon: thèse, antithèse, synthèse. Dans chacun, 1 à 3 arguments précis. Idées principales lisibles par un autre humain.

Faire des phrases en anglais (respectant le vocabulaire, la grammaire, …)

1) Algorithmique et programmation

- Dans ce cours: les algorithmes seront écrits en **pseudo-code.**

- Le pseudo-code est une façon de décrire un algorithme pour se comprendre **"entre humains",** indépendamment d'un langage de programmation.

- Les règles de pseudo-code choisies sont fortement **inspirées** des règles (syntaxe, mot-clés…) communes à la **plupart des langages de programmation.**

- Pour résoudre un problème complexe, il est souvent nécessaire de **commencer par écrire un algo. en pseudo-code** avant de l'implémenter.

1) Algorithmique et programmation

- *Parfois, on illustrera certains algorithmes en donnant un exemple d'implémentation en C ou en Python, mais les implémentations ne font pas partie des compétences attendues dans ce cours.*

Exemple: Deux implémentations

Algorithme (pseudo-code)

```
Programme principal
Début
        print( "Hello World")
Fin
```

Programme en C

```
#include <stdio.h>

int main() {
      printf("hello world\n");

      return 0;
}
```

Programme en Python

```
print("Hello World!")
```

1) Algorithmes

Vous connaissez déjà des algorithmes, qui ne sont pas en pseudo-code!

- Recettes de cuisine
- Itinéraire à suivre
- Algorithme d'Euclide pour le calcul du PGCD
- …

2) Pseudo-code

- Contrairement aux langages de programmation, il n'y a pas de syntaxe précise à respecter impérativement en pseudo-code.
- Chaque groupe de personnes doit se mettre d'accord sur la "syntaxe" à utiliser (instructions, mot-clés, …) pour le pseudo-code, pour bien se comprendre.
- **Les slides suivants vont fixer les règles que l'on devra respecter dans le cadre de ce cours.**

2.1) Variables & Types

- Les **noms de variable**s ne contiendront que des caractères **alpha-numériques** (pas d'espace, de ponctuation, …) et éventuellement le caractère _
- Il faudra **déclarer une variable** en donnant son type et son nom avant sa première utilisation.

(comme en C, mais contrairement au Python)

- Une même variable ne pourra donc recevoir **qu'un seul type de valeur** (entier, flottant, caractère…)

(comme en C, mais contrairement au Python)

2.1) Variables & Types

- Dans un premier temps, on utilisera les types suivants:
  - **entier**, appelé aussi **int**
  - **flottant** (nombre à virgule), appelé aussi **float**
  - **booléen**, appelé aussi **bool**
  - **caractères**, appelé aussi **char**

2.2) Affectation, opérations

- Le symbole utilisé pour l'**affectation** sera **:=** ou éventuellement ← .
- On supposera que les symboles **+, -, *** fonctionnent comme les opérateurs mathématiques classiques.
- On supposera que le symbole **/** fait une **division** en gardant les chiffres après la virgule.
- Pour tronquer les chiffres après la virgule d'un flottant x, on utilisera **floor(x)** (partie entière inférieure)
- Pour l'opérateur "modulo", on utilisera **%** ou éventuellement **mod**.

2.2) Affectation, opérations

- On pourra **incrémenter** (ou **décrémenter**) de un une variable entière avec le symbole **++** (ou **--**):
- Exemple:

```
int n          // création d'une variable n de type int
n:=5           // initialisation de n à 5
n++            // n vaut maintenant 6
n- -           // n vaut maintenant 5
```

- On pourra utiliser aussi le raccourci **+=**:

```
int somme:=1
somme += 4     // raccourci de somme:=somme+4
```

Exemples

```
int a                  // déclaration d'une variable a de type int
int b,c                //decl. de b et c, de type int
a:=5                   // a prend la valeur 5
b:=8                   // b prend la valeur 8
c:= a+b                //c prend la valeur 13
a:=a*b                 // a prend la valeur  5*8 donc 40
c:= b % 2              // 8 modulo 2 égal 0, donc c vaut 0
a:=floor(14.9)         // a prend la valeur 14
a++                    // a augmente de 1, donc a vaut 15
b- -                   // b diminue de 1, donc b vaut 7
```

Exemples

```
float x, y, z      // déclaration de 3 variables de type float
x:=5.6             // x prend la valeur 5.6
y:=2               // y prend la valeur 2.0
z:=x/y             //z prend la valeur 5.6/2.0 donc 2.8
z:= 1/4            // z prend la valeur  0.25
x:= x+y            // x prend la valeur 5.6+2.0 donc 7.6
```

2.2) Opérations

- Un booléen pourra prendre la valeur **0** (aussi appelée **False** ou **Faux**), ou la valeur **1** (aussi appelée **True** ou **Vrai**).
- Pour les opérateurs booléens, on utilisera **et**, **ou**, **non**, **xor** **(ou exclusif)**.
- Pour tester l'égalité, on utilisera le symbole **==** *(comme en C et en Python)*
- Pour tester la non-égalité, on utilisera **!=** ou, au choix, **≠**

Exemples

```
bool b          // déclaration d'une variable booléenne b
int n,m         // déclaration de deux variables n et m
n:= 5           // n vaut 5
m:= 4           // m vaut 4
b:= (n>m)       // b vaut Vrai
b:= b ou (n<3)  // b vaut (Vrai ou (n<3)) donc Vrai
b:= non(b)      // b vaut non(Vrai), donc Faux
b:=(n==5)       // b vaut Vrai car n est bien égal à 5
```

2.3) Divers

- Pour écrire des commentaires, on utilisera **//** ou les balises **/*** (début) et ***/** (fin) *(comme en C)*

- Parfois (pas systématiquement), on écrira **Début** et **Fin** pour délimiter notre algorithme (ce qui correspondra à notre "programme principal").

2.4) Entrée/Sortie en console

- Pour afficher quelque chose dans la console, on utilisera l'instruction (ou plutôt la fonction) **print(…).**
- On sera **souple sur le contenu des parenthèses** (pas de syntaxe stricte), du moment que votre ligne est facilement compréhensible, par exemple:

```
int n:=5
int m:=8
print("Valeur de n:", n, "et de m:", m)
// Ou, comme en C:
print("Valeur de n: %d et de m: %d", n, m)
```

2.4) Entrée/Sortie en console

- Pour demander à l'utilisateur de taper quelque chose au clavier dans la console, et récupérer cette valeur/chaîne de caractères, on utilisera la syntaxe suivante (proche du Python):

```
nomVariable:=input("Phrase affichée avant le curseur")
```

- S'il n'y a pas d'ambiguité sur le type de la variable qui récupère la valeur tapée (entier, flottant, caractère….), on pourra supposer que la conversion est automatique.

2.4) Entrée/Sortie en console

- Exemple:

```
Début
        int n
        n:=input("Donnez un entier:")
        int m:=2*n
        print("Son double est", m)
Fin
```

/ Ce programme demande un entier à l'utilisateur (entrée clavier en console) et affiche son double dans la console*/*

2.4) Instruction conditionnelle

- On utilisera **Si…. SinonSi…. Sinon.**
- On se servira de l'indentation et d'une barre verticale, ou un FinSi pour délimiter nos blocs
- Exemple:

```
Debut
        int a:=56+31
        Si a%2==0:
        |       print("La somme est paire")
        Sinon:
        |       print("La somme est impaire")
Fin
```

2.4) Instruction conditionnelle

- Ou:

```
Debut
        int a:=56+31
        Si a%2==0:
                print("La somme est paire")
        Sinon:
                print("La somme est impaire")
        FinSi
Fin
```

2.4) Instruction conditionnelle

- Rappels:
  - il peut y avoir **0, 1 ou plusieurs SinonSi**
  - il peut y avoir **0 ou 1 Sinon**
- Exemple:

```
Debut
    int a:=input("Donnez un entier")
    Si a==0:
    |    print("L'entier donné est nul")
    SinonSi a>0:
    |    print("L'entier donné est strict. positif")
    Sinon:
    |    print("L'entier donné est strict. négatif")
Fin
```

2.5) Boucle Pour

- On utilisera les boucles Pour ainsi:

Pour *indice* de *indiceDebut* à *indiceFin* :

```
    instructions
```

- On délimitera notre bloc de la boucle pour par l'indentation + une barre verticale, ou en ajoutant un FinPour
- Exemple:

```
Debut
    int somme:=0
    int n:=15
    int i
    Pour i de 1 à n:
    |       somme:= somme + i
    print("Somme totale:", somme)
Fin
```

2.5) Boucle Pour

- Si le pas n'est pas précisé, il s'agit d'un pas de 1. On peut choisir un pas différent (pos. ou nég.) ainsi:

Pour *indice* de *indiceDebut* à *indiceFin* par pas de *P*

- Exemple:

```
Debut
        int somme:=0
        int n:=15
        int i
        Pour i de 1 à n, par pas de 2:
                somme:= somme + i
        Fin Pour
        print("Somme totale des impairs:", somme)
Fin
```

2.6) Boucle Tant Que

- On utilisera une boucle Tant Que ainsi:

Tant que condition:
    instructions

- Là encore, on délimitera notre bloc par l'indentation + une barre verticale, ou en ajoutant un FinTantQue.

Tant que condition:
    instructions
FinTantQue

2.6) Boucle Tant Que

- Exemple:

```
Debut
        int somme:=0
        int note:=0
        int nbNotes:=0
        TantQue note>=0 et nbNotes<5:
        |       note:=input("Prochaine note?")
        |       somme:=somme+note
        |       nbNotes++
        print("Moyenne:", somme/nbNotes)
Fin
```

QCM Vote électronique

```
Debut
    int n,  produit, i
    produit:=1
    n:=3
    Pour i de 1 à n:
    |       produit:=produit*i
    Si n%2==1:
    |       produit:=produit*n
    Si n>10:
    |       produit:=2*produit
Fin
```

Que vaut produit à la fin de l'algorithme?

1) 3
2) 6
3) 12
4) 18
5) 36

QCM Vote électronique

```
Debut
    int n,  produit, i
    produit:=1
    n:=3
    Pour i de 1 à n:
    |     produit:=produit*i
    Si n%2==1:
    |     produit:=produit*n
    Si n>10:
    |     produit:=2*produit
Fin
```

Que vaut produit à la fin de l'algorithme?

1) 3
2) 6
3) 12
4) **18**
5) 36

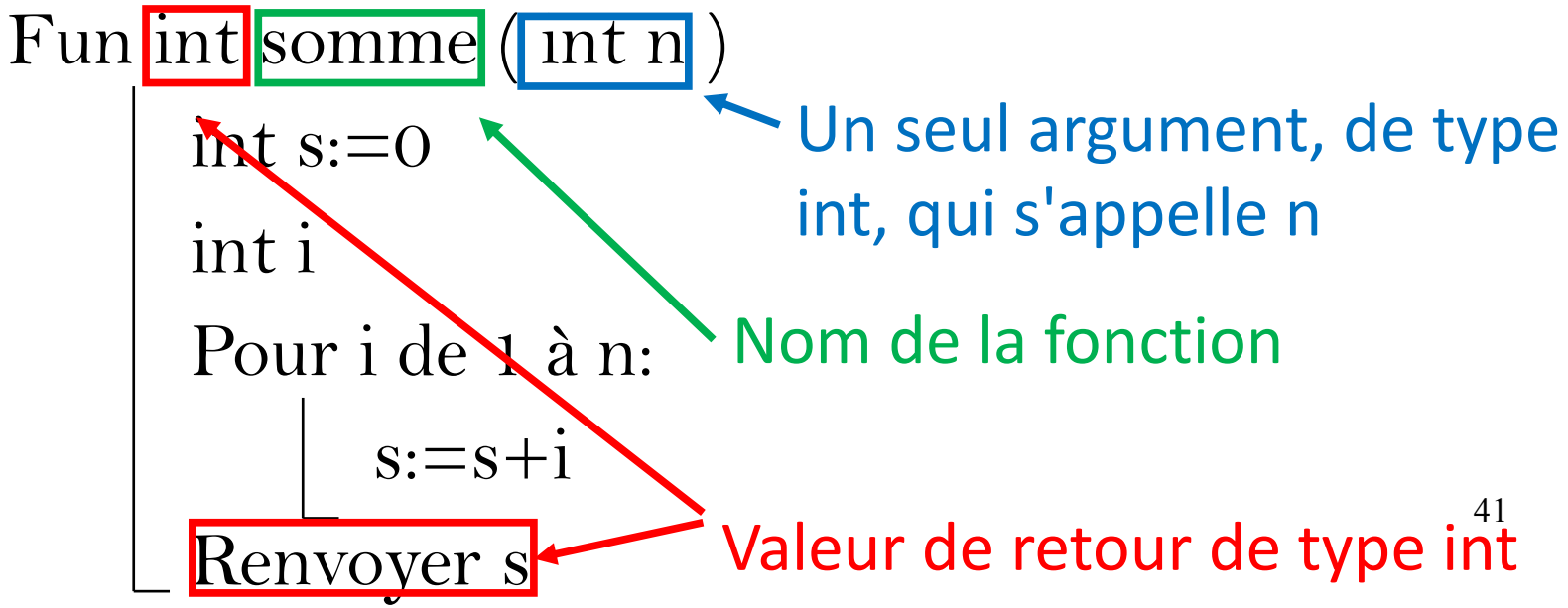
2.7) Fonctions

- La première ligne du code source d'une **fonction** devra être **sa signature,** incluant le type de la valeur de retour et le type de chaque argument (le tout éventuellement précédé du mot **Fun**)
- On délimitera le bloc de la fonction par l'indentation + barre verticale, ou par une paire d'accolades.
- On utilisera **Retourner** ou **Renvoyer** pour la valeur de retour.
- Exemple:

```
Fun int somme ( int n )
    int s:=0
    int i
    Pour i de 1 à n:
        s:=s+i
    Renvoyer s
```

2.7) Fonctions

- La première ligne du code source d'une **fonction** devra être **sa signature,** incluant le type de la valeur de retour et le type de chaque argument (le tout éventuellement précédé du mot **Fun**)
- On délimitera le bloc de la fonction par l'indentation + barre verticale, ou par une paire d'accolades.
- On utilisera **Retourner** ou **Renvoyer** pour la valeur de retour.
- Exemple:



2.7) Fonctions

- Dans le cas où la **fonction ne renvoie rien**, on utilisera **void** comme type de retour, ou bien on utilisera le mot-clé **Proc** (pour procédure) à la place de Fun
- Exemple:

```
Proc affiche(int n)
    int i
    Pour i de 1 à n:
        print(i)
```

ou:

```
Fun void affiche(int n)
    int i
    Pour i de 1 à n:
        print(i)
```

2.7) Fonctions

- On s'autorisera à utiliser un type "fictif" **nombre** lorsque l'argument d'une fonction peut être indifféremment un entier ou un flottant.
- Exemple:

```
Fun float div (nombre numerateur, nombre denominateur)
    float resultat:=numerateur/denominateur
    Renvoyer resultat
```

2.7) Fonctions: d'autres exemples

```
Fun float max (float x, float y)
    float m
    Si x>y:
        m:= x
    Sinon:
        m:= y
    Renvoyer m

Proc demandeNombreEtAfficheOpposé()
    float x
    x:=input("Donnez un nombre")
    print("Son opposé est", -x)
```

2.8) Fonctions: rappels

Rappels: il ne faut pas confondre

- les **arguments**, aussi appelés **paramètres**, qui sont donnés à la fonction lors de l'appel
- la **valeur de retour**
- les **effets de bord**: **affichage** en console, **entrée clavier** en console, modification de la mémoire (**modification d'une case d'un tableau**, ...)

Lorsqu'une instruction **Renvoyer est exécutée, on sort de la fonction,** les éventuelles instructions suivantes dans le bloc de la fonction sont ignorées.

2.8) Fonctions: appels

Exemple: pour **récupérer la valeur de retour** d'une fonction dans une variable, par exemple la fonction somme définie précédemment:

```
s:=somme(5)
// s vaut 1+2+3+4+5, c'est-à-dire 15
```

Exemple: si la fonction n'a **pas de valeur de retour**, l'appel de la fonction ne doit pas faire partie d'une affectation

```
print("Affichage des nombres de 1 à 7")
affiche(7) // provoque des affichages en console seulement
```

2.8) Fonctions: portée des variables

- Les variables ne sont pas accessibles depuis n'importe quelle endroit (ligne) de notre programme. La **"portée"** ou **"visibilité"** d'une variable désigne **l'ensemble des endroits où cette variable est accessible**.

- **Une fonction ne peut lire/écrire que dans ses variables locales**, constituées des arguments de la fonction et des variables déclarées à l'intérieur de la fonction.

- La même règle s'applique au programme principal.

Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
        int n, m
        n:=4
        m:=suivant(n)
Fin
```

Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
    int n, m
    n:=4
    m:=suivant(n)
Fin
```

Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
    int n, m
    n:=4
    m:=suivant(n)
Fin
```

Début

Variables du prog. princ.

n (int)

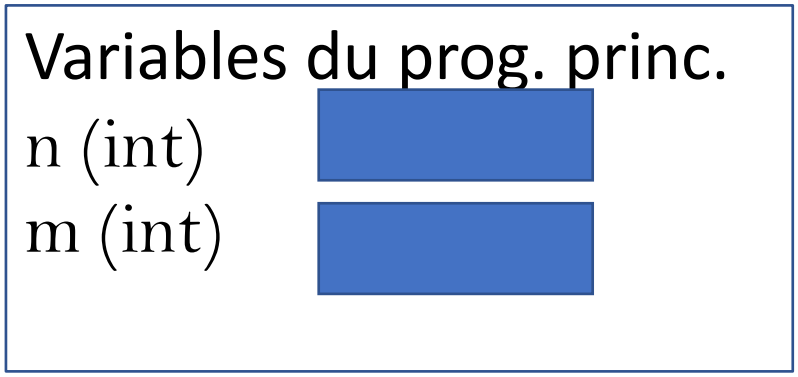
m (int)

Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
        int n, m
        n:=4
        m:=suivant(n)
Fin
```

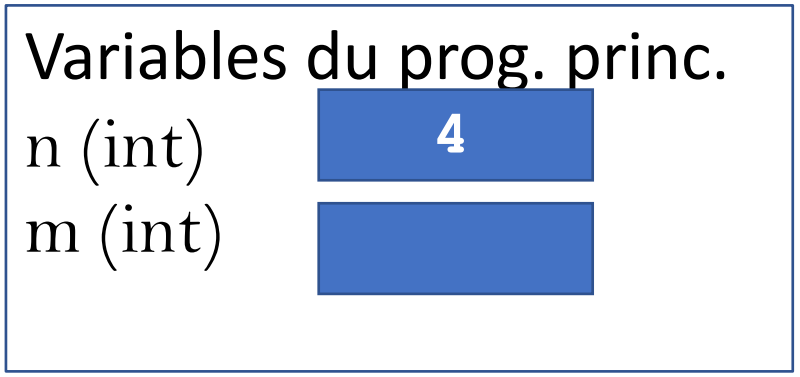


Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
        int n, m
        n:=4
        m:=suivant(n)
Fin
```

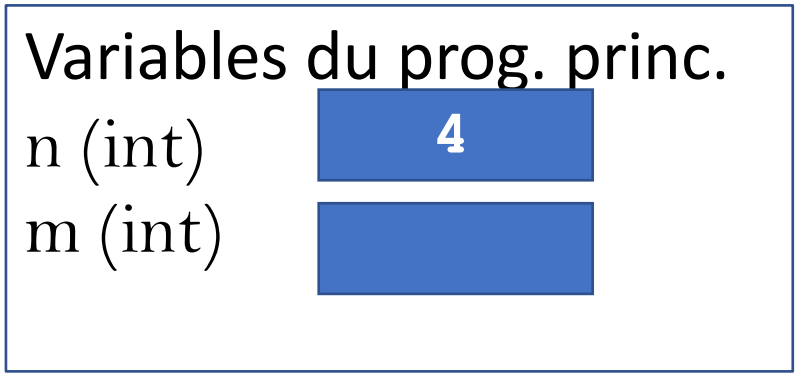


Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
        int n, m
        n:=4
        m:=suivant(n)
Fin
```

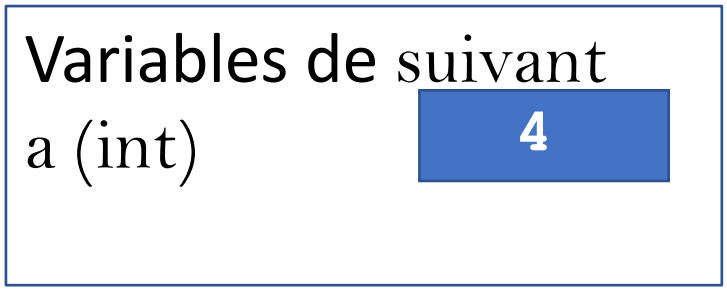


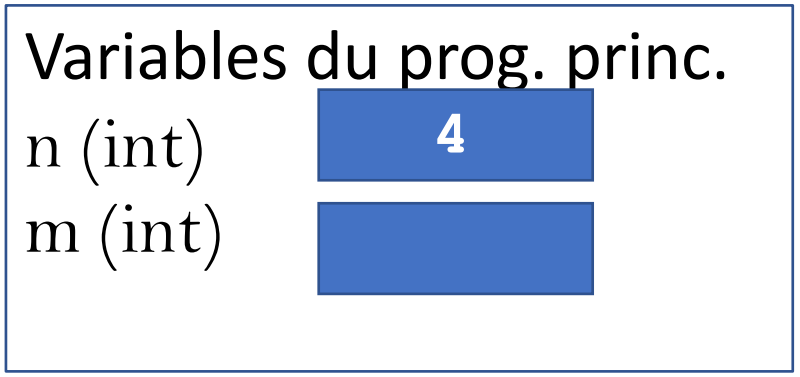
Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
        int n, m
        n:=4
        m:=suivant(n)
Fin
```



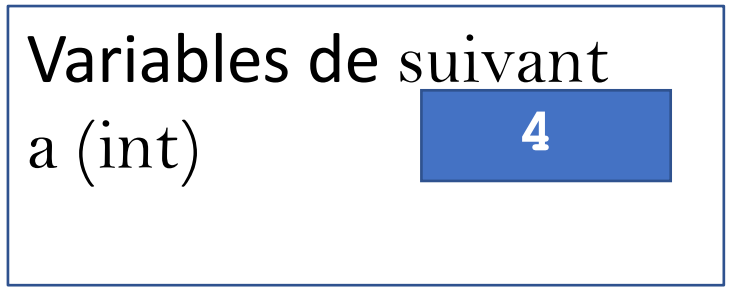


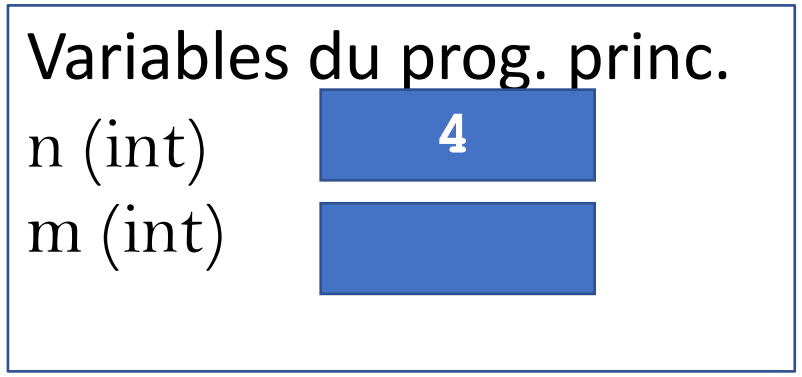
Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
        int n, m
        n:=4
        m:=suivant(n)
Fin
```



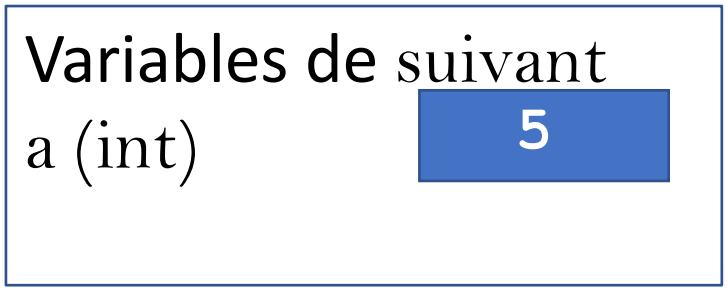


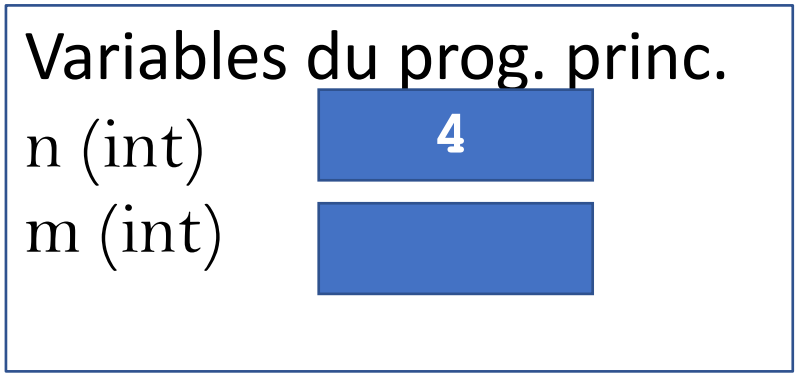
Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
        int n, m
        n:=4
        m:=suivant(n)
Fin
```



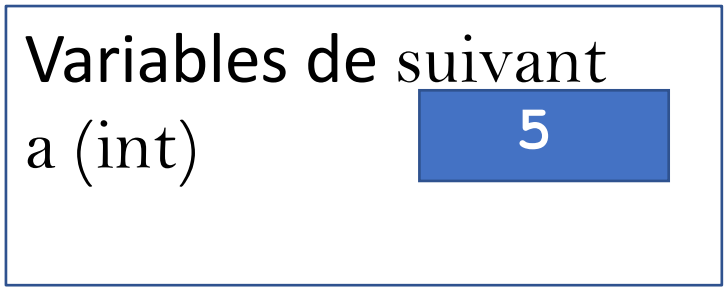


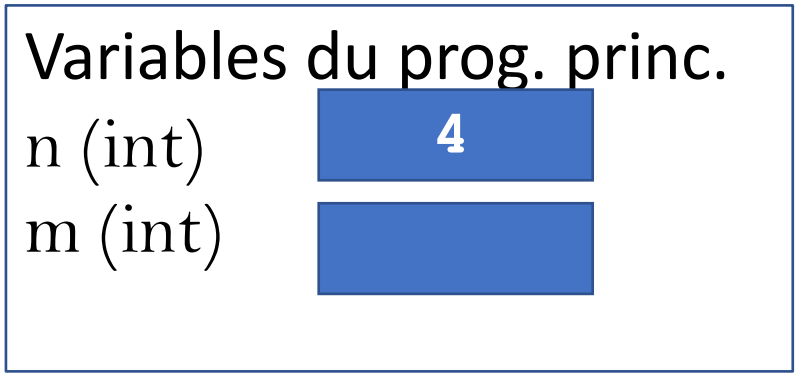
Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
    int n, m
    n:=4
    m:=suivant(n)
Fin
```



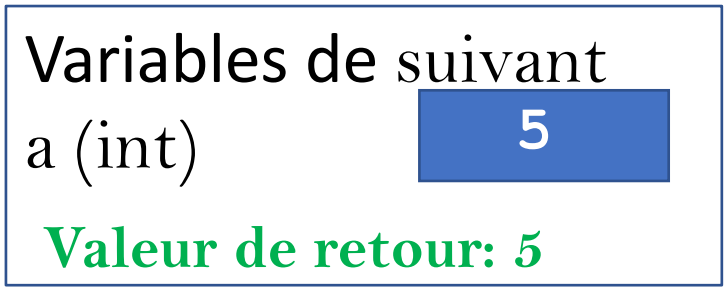


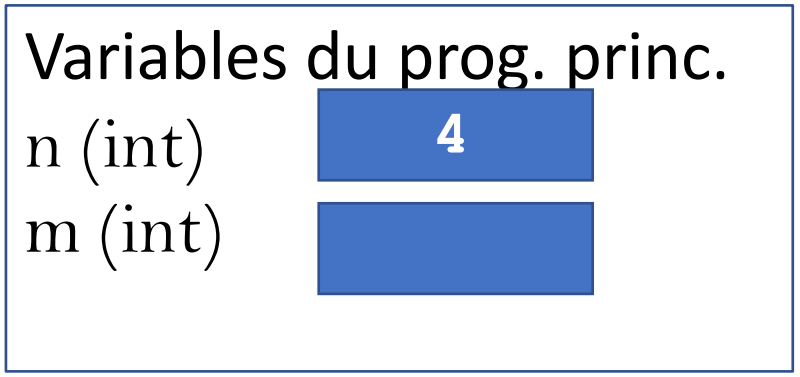
Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
    int n, m
    n:=4
    m:=suivant(n)
Fin
```



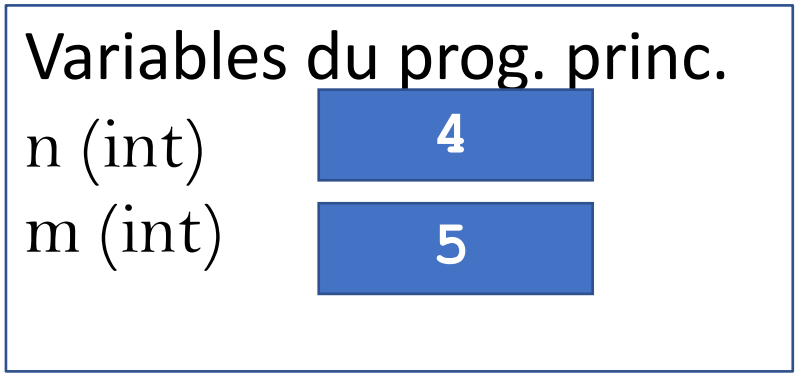


Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
        int n, m
        n:=4
        m:=suivant(n)
Fin
```

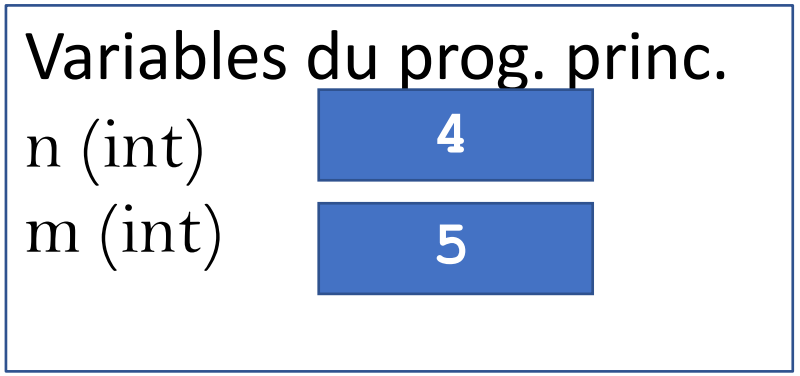


Passage par copie/variable locales

Exemple (fonction peu utile mais pédagogique):

```
Fun int suivant(int a)
    a++
    Renvoyer a

Début
        int n, m
        n:=4
        m:=suivant(n)
Fin
```



Variable locales : un exemple

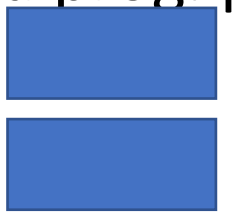
```
Fun int suivant(int n)
    int decalage:=4
    Renvoyer n+decalage
```

Variables de suivant
n (int)
decalage (int)

Ici: on ne peut pas se servir de n et m

```
Début
        int n, m
        n:=4
        m:=suivant(n)
        Renvoyer 0
Fin
```

Variables du prog. princ.
n (int)
m (int)



Ici: on ne peut pas se servir de n et decalage

QCM Vote électronique

```
Fun float valeurAbs (float x)
    Si x>=0:
        Renvoyer x
    Sinon:
        Renvoyer –x
```

```
Fun float écart(float x, float y)
    float d
    d:=valeurAbs(x-y)
    Renvoyer d
```

Quelle est la valeur de écart(-1, 3.5) ?

1) -1
2) 2.5
3) 3.5
4) 4.5
5) -4.5
6) On ne peut pas savoir

QCM Vote électronique

```
Fun float valeurAbs (float x)
    Si x>=0:
        Renvoyer x
    Sinon:
        Renvoyer –x

Fun float écart(float x, float y)
    float d
    d:=valeurAbs(x-y)
    Renvoyer d
```

Quelle est la valeur de écart(-1, 3.5) ?

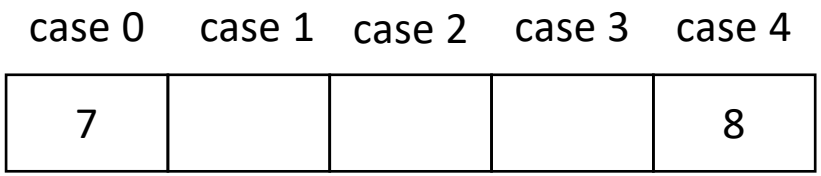
1) -1
2) 2.5
3) 3.5
4) **4.5**
5) -4.5
6) On ne peut pas savoir

2.8) Tableaux

- Pour **déclarer un tableau**, il faut préciser le **type de ses cases**, le **nom de la variable** tableau et entre crochets le **nombre de cases.**
- On s'autorisera à créer des tableaux dont la taille est une variable *(contrairement au C)*.
- **Les cases d'un tableau à n cases sont numérotées de 0 à n-1.**

Exemple:

```
int n:=5
int tab[n]
tab[0]:=7
tab[4]:=8
```



2.8) Tableaux

On supposera qu'on peut **accéder à la taille d'un tableau** en utilisant la fonction **longu(…)**

(contrairement au C).

Exemple:

```
int n,m // déclaration de 2 variables entières n et m
n:=10
int tab[n]
m:= longu(tab) // m contient la valeur 10
```

2.8) Tableaux et fonctions

- On s'autorisera à parler d'un tableau de nombre s'il n'y a pas de différence de traitement entre les tableaux d'entiers et les tableaux de flottants.
- Une fonction qui prend un tableau comme argument peut modifier le tableau (autrement dit, le tableau n'est pas recopié dans une variable locale: on manipule vraiment le véritable tableau)

Exemple:

```
Proc doubleContenuCase(nombre tab[ ])
|       int i
|       Pour i de 0 à longu(tab)-1:
|       |       tab[i]:=2*tab[i]
```

2.8) Tableaux: grands classiques

3 grands classiques à bien connaître:

1. Fonction **d'affichage d'un tableau**

```
Proc afficheTab(nombre tab[ ])
    int i
    Pour i de 0 à longu(tab)-1:
        print(tab[i])
```

2.8) Tableaux: grands classiques

2.Fonction qui renvoie la **somme des cases d'un tableau**

```
Fun nombre sommeTab(nombre tab[ ])
    nombre s:=0
    int i
    Pour i de 0 à longu(tab)-1:
        s:=s+tab[i]
    Renvoyer s
```

(nombre désignant donc soit int, soit float)

2.8) Tableaux: grands classiques

3. Fonction pour trouver le **max dans un tableau**

```
Fun nombre maxTab(nombre tab[ ])
    nombre maxJusquIci:=tab[0]
    int i
    Pour i de 1 à longu(tab)-1:
        Si tab[i]>maxJusquIci
            maxJusquIci:=tab[i]
    Renvoyer maxJusquIci
```

3) Qu'est un bon algorithme?

Il y a plusieurs critères qui peuvent définir un bon algo:

- Il est **facile à lire**, comprendre
- Il est **facile à analyser** (pour être sûr qu'il répond au problème posé)
- Le nombre d'instructions/opérations élémentaires effectuées pendant l'exécution du programme est faible → **complexité temporelle** **(voir chap suivant**)
- La quantité de mémoire utilisée par l'algorithme est petite. → **complexité spatiale**
- …

→ En général, on doit faire un compromis entre tous ces critères

Fin chapitre 1: rappels

Récapitulatif des notions qui ont été abordées:

- *différence entre algorithme et implémentation*
  - *Algorithme: phrase (dans la vie) ou pseudo-code (dans ce cours)*
  - *Implémentation: dans un langage précis (C, Python…)*
- *"Syntaxe" du pseudo-code que l'on va utiliser*
- *Qu'est-ce qu'un bon algo?*
  - *facile à lire*
  - *facile à analyser*
  - *bonne complexité temporelle (peu de temps)*
  - *bonne complexité spatiale (peu de mémoire)*

QCM Vote électronique

```
Proc remplirTableau(nombre tab[])
    int i
    Pour i de 0 à longu(tab)-2:
        tab[i]:=2*i
Début
    int t[6]
    remplirTableau(t)
Fin
```

Comment est rempli le tableau t à la fin ?

1)

2	4	6	8	10	12

2)

0	2	4	6	8	10

3)

2	4	6	8	10	

(Rien dans la dernière case)

4)

0	2	4	6	8	

(Rien dans la dernière case)

QCM Vote électronique

```
Proc remplirTableau(nombre tab[])
    int i
    Pour i de 0 à longu(tab)-2:
        tab[i]:=2*i
Début
    int t[6]
    remplirTableau(t)
Fin
```

Comment est rempli le tableau t à la fin ?

1)	2	4	6	8	10	12
2)	0	2	4	6	8	10
3)	2	4	6	8	10	

(Rien dans la dernière case)

4)	0	2	4	6	8	

(Rien dans la dernière case)