Algorithme et structure de données

2023-2024

Eductive

CAMPUS EDUCTIVE AIX-EN-PROVENCE

Campus Eductive – Aix en Provence
ESGI 1

David Palermo

Mail : dpalermo1@myges.fr

Algorithme et structure de données

Semestre 1 : 10 heures

CAMPUS
EDUCTIVE
AIX-EN-PROVENCE

Algorithmes
Les bases


Facile


Normal


Difficile


Professionnel

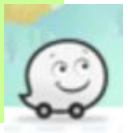

Expert

david.palermo@yantra-technologies.com

https://wiki.waze.com/wiki/Your_Rank_and_Points

Sommaire

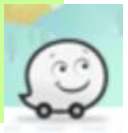
1 – Généralités -> Définition



Définition Larousse:

"Ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations. Un algorithme peut être traduit, grâce à un langage de programmation, en un programme exécutable par un ordinateur."

1 – Généralités -> Définition

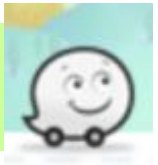


Un algorithme est une méthode ou un procédé décrit pas à pas.

Ce n'est pas :

- un problème de décision, mais une méthode pour résoudre un tel problème
- un langage de programmation
- un codage numérique mais les données et les résultats de n'importe quel algorithme doivent être codés de façon numérique

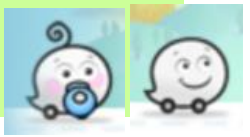
1 – Généralités -> Concept de bases



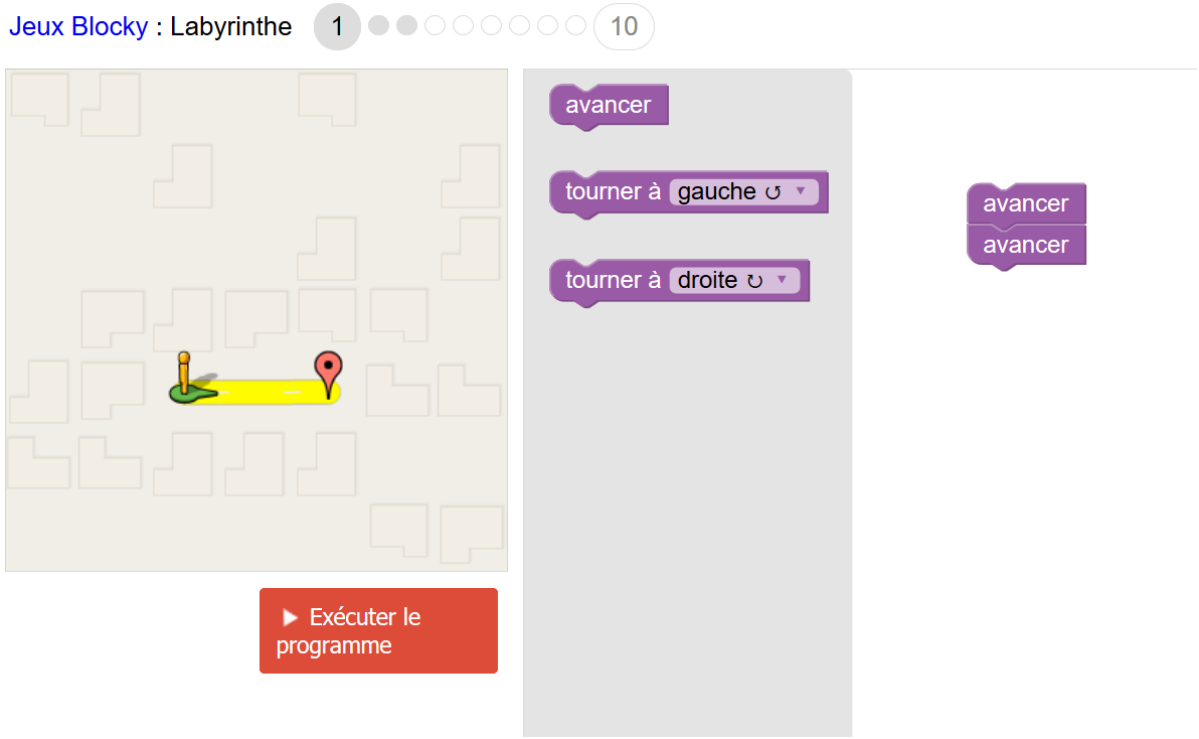
Les concepts en œuvre en algorithmique, par exemple selon l'approche de N. Wirth pour les langages les plus répandus (Pascal, C, etc.), sont en petit nombre. Ils appartiennent à deux classes :

- les structures de contrôle
  - séquences
  - conditionnelles
  - boucles
- les structures de données
  - constantes
  - variables
  - tableaux
  - structures récursives (listes, arbres, graphes)

1 – Généralités -> Exemple

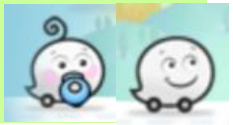


https://blockly-games.appspot.com/maze?lang=fr&level=4&skin=0



Autres => https://www.maths-et-tiques.fr/index.php/logiciels/algorithmique

1 - Généralités -> Pseudo code

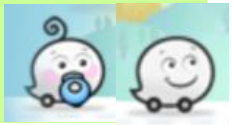


Le pseudo-code est un langage qui permet de décrire facilement un algorithme avec un vocabulaire simple et sans connaissance à priori du langage de programmation.

Ce travail d'algorithmique peut se faire sans ordinateur, sur une simple feuille de papier.

Vous pouvez échanger en pseudo-code avec une autre personne qui utilise un langage de programmation que vous ne maitrisez pas.

1 - Généralités -> Instructions



Instruction

Commande élémentaire interprétée et exécutée par le processeur.

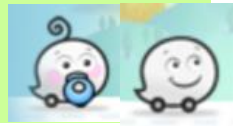
Jeu d'instruction

Dans un processeur, ensemble des instructions que cette puce peut exécuter.

Bloc d'instructions

Dans un algorithme, séquence d'instructions pouvant être vue comme une seule instruction.

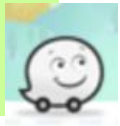
1 - Généralités -> Mot clés du langage



ALGORITHME	PROCEDURE	CONSTANTES	VARIABLES
SINON	POUR	TANT_QUE	JUSQU'A
DIV	CARACTERE	CHAINE	NON
BEBUT	FIN	FONCTION	SI
OU	ET	MOD	PARAMETRES
ALORS	BOOLEEN	ENTIER	REEL
REPETER	SELON	AUTREMENT	RENVOIE
SORTIE	ENTREE	FAUX	VRAI
STRUCTURE	TYPE		

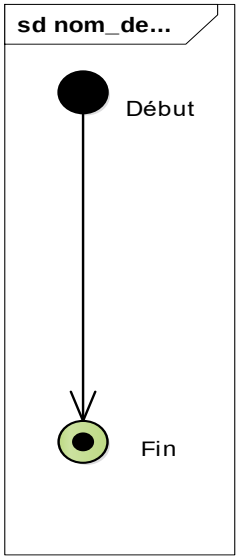
https://fr.wikiversity.org/wiki/Langage_C%2B%2B/Mots_cl%C3%A9s
https://fr.wikibooks.org/wiki/Programmation_Java/Liste_des_mots_r%C3%A9serv%C3%A9s
https://www.gladir.com/CODER/PYTHON/reservedword.htm

1 – Généralités -> Programme



Un programme est la traduction d'un algorithme dans un langage de programmation.

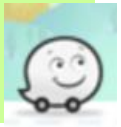
```
ALGORITHME nom_de_algo

   <partie déclarations>

DEBUT

   <partie instructions>

FIN
```



1 – Généralités -> Programme : Exemple en C

```
int main()  {
      return 0;
}
```

2 - Variable, affectation



Base de la programmation, les variables permettent d'associer un nom à une valeur, celle-ci pouvant évoluer au cours du programme

Les variables permettent lors de l'exécution d'un algorithme, de stocker des données, des résultats ...

On attribue un nom à chaque variable.

2 - Variable, affectation -> Type

Type	Domaine
BOOLEAN	{ FAUX, VRAI }
CARACTERE	Symbole typographique
ENTIER	$\mathbb{N}$
REEL	$\mathbb{Z}$
CHAINE	"canard"

2 - Variable, affectation -> Type

Type Numérique	Plage
Byte (octet)	0 à 255
Entier simple	-32 768 à 32 767
Entier long	-2 147 483 648 à 2 147 483 647
Réel simple	$-3{,}40\times10^{38}$ à $-1{,}40\times10^{45}$ pour les valeurs négatives $1{,}40\times10^{-45}$ à $3{,}40\times10^{38}$ pour les valeurs positives
Réel double	$1{,}79\times10^{308}$ à $-4{,}94\times10^{-324}$ pour les valeurs négatives $4{,}94\times10^{-324}$ à $1{,}79\times10^{308}$ pour les valeurs positives

https://fr.wikibooks.org/wiki/Programmation_Java/Types_de_base

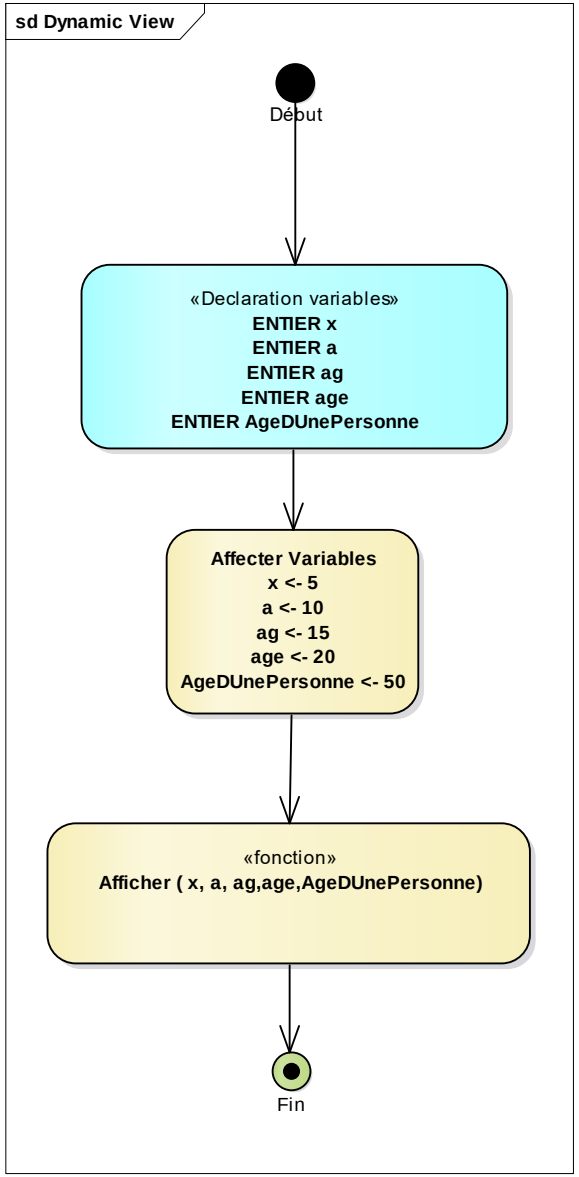
https://fr.wikiversity.org/wiki/Python/Les_types_de_base

https://en.cppreference.com/w/cpp/language/types

2 - Variable, affectation -> Type : Exemple

Je veux mémoriser la taille d'une personne dans une variable, j'ai le choix de le nommer :

```
ALGORITHME nom_de_algo
DEBUT
        ENTIER x
        ENTIER a
        ENTIER ag
        ENTIER age
        ENTIER AgeDUnePersonne

        x <- 5
        a <- 10
        ag <- 15
        age <- 20
        AgeDUnePersonne <- 50
        afficher(x,a,ag,age, AgeDUnePersonne)
 FIN
```



2 - Variable, affectation -> Type : Exemple Python

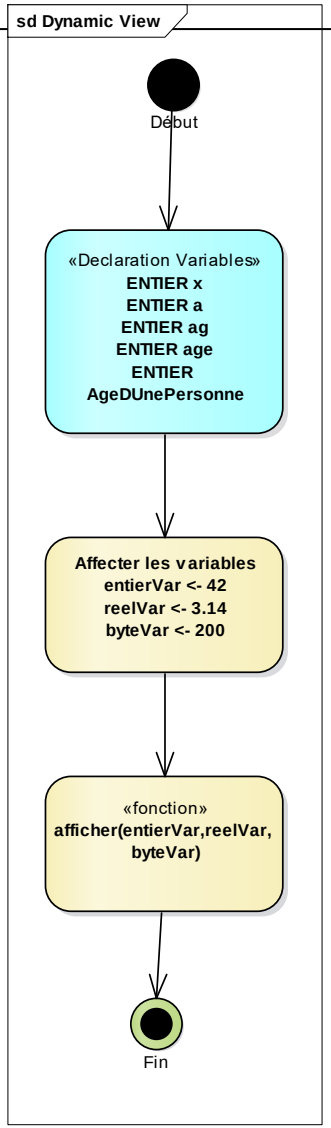
```
if __name__ == "__main__":

    x=0
    a=0
    ag=0
    age=0
    AgeDUnePersonne=0

    x = 5
    a = 10
    ag = 15
    age = 20
    AgeDUnePersonne = 50

    print(5,10,15,20,50)
```

2 - Variable, affectation -> Type : Exemple C

```c
#include <stdio.h>

int main() {
    int x = 0;
    int a = 0;
    int ag = 0;
    int age = 0;
    int AgeDUnePersonne = 0;

    x=5;
    a=10;
    age=15;
    AgeDUnePersonne = 50;
    printf("%d %d %d %d %d \n",x,a,ag,age,AgeDUnePersonne);
    return 0;
}
```

Variable, affectation -> Type : Exemple

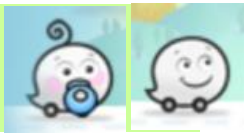
```
ALGORITHME nom_de_algo
DEBUT
        ENTIER : entierVar
        REEL : reelVar
        BYTE : byteVar

        entierVar <- 42
        reelVar <- 3.14
        byteVar <- 200

        afficher(entierVar,reelVar, byteVar)
 FIN
```



2 - Variable, affectation -> Type : Exemple C

```c
#include <stdio.h>

int main() {
    int entierVar;
    float reelVar;
    unsigned char byteVar;

    entierVar = 42;
    reelVar = 3.14;
    byteVar = 200;

    printf("%d %f %c",entierVar,reelVar,byteVar)
    return 0;
}
```

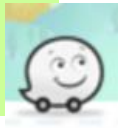
3 - Instruction de base



Les instructions de base sur des variables sont les suivantes :

- **la saisie** : on demande à l'utilisateur de l'algorithme de donner une valeur à la variable
- **l'affectation** : le concepteur de l'algorithme donne une valeur à la variable. Cette valeur peut-être le résultat d'un calcul
- **l'affichage** : on affiche la valeur de la variable.

3 - Instruction de base -> Exemple

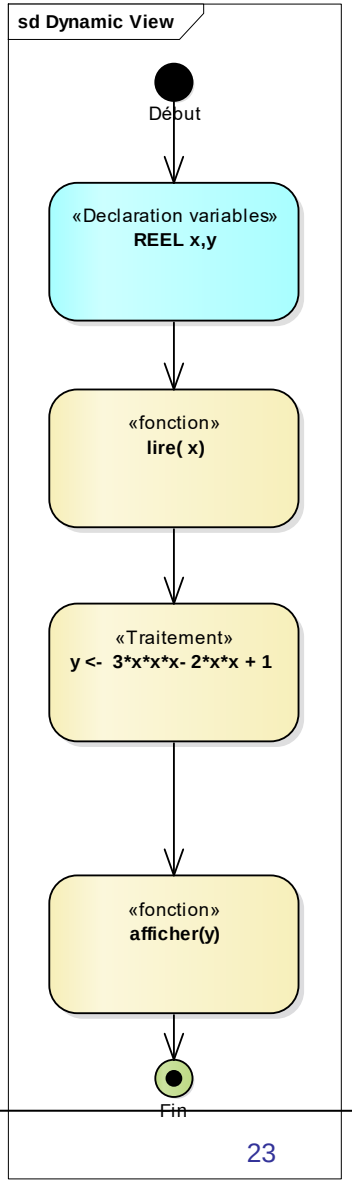


But calculer : $fonction(x) = 3\,x^3 - 2^2 + 1\ avec\ x\ \in\ \mathbb{R}$

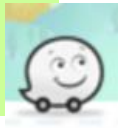
ALGORITHME fonction
DEBUT

```
// Variables Locales
 REEL x , y //nombres réels

// Instructions
 afficher(« Rentrer valeur de x : ")
 lire ( x )  //Entrée
 y <-  3*x*x*x– 2*x*x + 1 //Traitement
 afficher ( y ) //Sortie
```

FIN

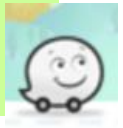


3 - Instruction de base -> Exemple Python



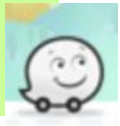
```
if __name__ == "__main__":

    x=0.0
    y=0.0

    print(type(x),type(y))

    x=float(input("rentrer valeur de x : "))
    y = 3*x*x*x-2*x*x+1

    print(y)
```

3 - Instruction de base -> Exemple C



```c
#include <stdio.h>

int main() {
    double x = 0;
    double y = 0;

    printf("Rentrer valeur de x : ") ;

    scanf("%lf",&x);

    y = 3*x*x*x-2*x*x+1;

    printf("y=%lf \n",y);
}
```

4 - Tableau Statique Unidimensionnel



Opération	Spécification	Exemple
Déclaration	Type nom [taille]	ENTIER tab[10]
Initialisation	Type nom[n] <- { v1..,vn}	CARACTERE voyelles[6] ← {'a','e','i','o','u','y'}
Accès	nom[index]	voyelles[1]

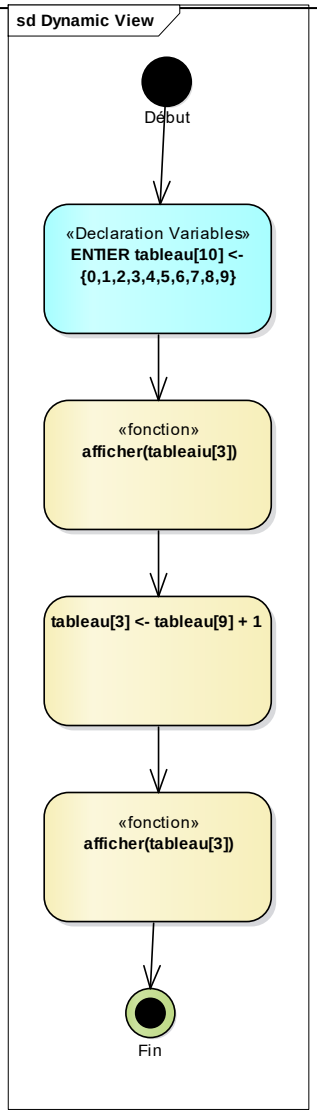
ATTENTION

Le premier index d'un tableau de N éléments est 0 et le dernier index est N − 1.

4 - Tableau : Statique Unidimensionnel -> Exemple

Définir un Tableau de 10 réels

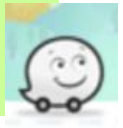
```
ALGORITHME Tableau

DEBUT
    REEL tableau[10] <- {0,1,2,3,4,5,6,7,8,9}

     afficher ( tableau [3] )
     tableau [3] <- tableau [9] + 1
     afficher ( tableau [3] )

FIN
```



4 - Tableau : Statique Unidimensionnel
-> Exemple Python

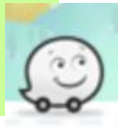
```
import array as arr

if __name__ == "__main__":

    tableau = arr.array('l',[0,1,2,3,4,5,6,7,8,9])

    print(type(tableau))

    print(tableau[3])
    tableau[3] = tableau[9] + 1
    print(tableau[3])

    tableau = [0,1,2,3,4,5,6,7,8,9]

    print(type(tableau))

    print(tableau[3])
    tableau[3] = tableau[9] + 1
    print(tableau[3])
```

4 - Tableau : Statique Unidimensionnel
-> Exemple C/C++



```c
int main(int , char *[]) {
    //C
    {
        double tableau[10] = {0,1,2,3,4,5,6,7,8,9};

        printf("%f %d\n",tableau[3], sizeof(tableau) / sizeof(double));
        tableau[3] = tableau[9] + 1;
         printf("%f\n",  tableau[3]);
    }

    //C++
    {
        std::array<double, 10> tableau ({0,1,2,3,4,5,6,7,8,9});
        std::cout <<  tableau[3] << " " << sizeof(tableau) / sizeof(double) << std::endl;
        tableau[3] = tableau[9] + 1;
        std::cout <<  tableau[3] << std::endl;

    }

   return 0;
}
```

4 - Tableau : Statique Multidimensionnel

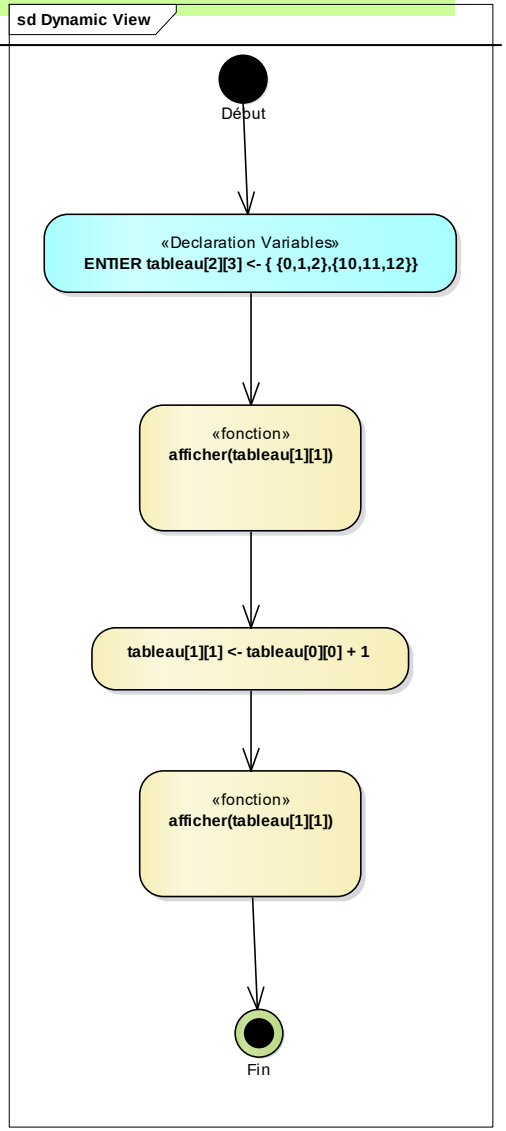


Opération	Spécification	Exemple
Déclaration	Type nom [ligne][colonne]	ENTIER tab[10][5]
Initialisation	Type nom[m][n] <- { { v11..,v1n}, … ,{ vm1..,vmn} }	REEL binaire[2][2] ← {{0,0},{0,1}}
Accès	nom[ligne][colonne]	matrice[i][j]

4 - Tableau : Statique Multidimensionnel -> Exemple

Définir une Tableau de 10 réels

```
ALGORITHME Tableau2D

DEBUT
    REEL tableau[2][3] <- { {0,1,2} , {10,11,12} }

    afficher ( tableau [1][1] )
    tableau [1][1] <- tableau [0][0] + 1
    afficher  ( tableau [1][1] )

FIN
```



4 - Tableau : Statique Multidimensionnel
-> Exemple Python

```
import array as arr

if __name__ == "__main__":

    tableau = [ arr.array('l',[0,1,2]), arr.array('l',[10,11,12]) ]

    print(type(tableau))

    print(tableau[1][1])
    tableau[1][1] = tableau[0][0] + 1
    print(tableau[1][1])

    tableau = [ [0,1,2],[10,11,12] ]

    print(type(tableau))

    print(tableau[1][1])
    tableau[1][1] = tableau[0][0] + 1
    print(tableau[1][1])
```

4 - Tableau : Statique Multidimensionnel
-> Exemple C/C++

```
int main(int , char *[])
{
    //C
    {
        double tableau[2][3] = {{0,1,2},{10,11,12}};

        printf("%f %d\n",tableau[1][1] ,sizeof(tableau) / sizeof(tableau[0]) );
        tableau[1][1] = tableau[0][0] + 1;
        printf("%f\n", tableau[1][1] );
    }

    //C++
    {
        std::array<std::array<double, 3>, 2> tableau  ({std::array<double, 3>({0,1,2}),
                                                        std::array<double, 3>({10,11,12})} );

        std::cout <<  tableau[1][1] << " " << tableau.size() << std::endl;
        tableau[1][1] = tableau[0][0] + 1;
        std::cout <<  tableau[1][1] << std::endl;
    }
```

5 - Operateurs : arithmétiques

Type	Ensemble de valeurs	Opérateurs	Exemple
ENTIER	$\mathbb{N}$	+ (somme) - (différence) * (multiplication) / (division entière)	ENTIER x,y,z,a x <- 1 + 2 y <- 1 - x z <- x * y a <- x /2
REEL	$\mathbb{R}$	+ (somme) - (différence) * (multiplication) / (division)	REEL x,y,z,a x <- 1.2 + 2.2 y <- 1.3 - x z <- x * y a <- x /2.3
CARACTERE	Symbole typographique		
CHAINE (CARACTERE*)	Tableau dynamique de Symbole typographique	+ (Concaténation)	CHAINE a <- "le canard ", b <- " est " a <- a + b + " tres bon"

5 – Operateurs : arithmétiques -> Exemple Python

```
if __name__ == "__main__":
#ENTIER
    x=0
    y=0
    z=0
    a=0

    print(type(x),type(y),type(z),type(a))

    x = 1 + 2
    y = 1 - 2
    z = x * y
    a = x // 2
    print(type(x),type(y),type(z),type(a))
    print(x,y,z,a)
#REEL
    x=0.0
    y=0.0
    z=0.0
    a=0.0

    print(type(x),type(y),type(z),type(a))

    x = 1.2 + 2.2
    y = 1.3 - x
    z = x * y
    a = x / 2.3
    print(type(x),type(y),type(z),type(a))
    print(x,y,z,a)
#CHAINE
    a = "le canard "
    b = " est "
    a = a + b + " tres bon"
    print(type(a),type(b))
    print(a)
```

CAMPUS EDUCTIVE AIX-EN-PROVENCE

Yantra Technologies

5 – Operateurs: arithmétiques -> Exemple C++

```
int main(int , char *[]){

    {
        //ENTIER
        int x=0;
        int y=0;
        int z=0;
        int a=0;

        x = 1 + 2;
        y = 1 - 2;
        z = x * y;
        a = x / 2;
        std:: cout << x << " "<< y << " "<< z << " "<< a << std::endl;
    }
    {
        //REEL
        double x=0.0;
        double y=0.0;
        double z=0.0;
        double a=0.0;

        x = 1.2 + 2.2;
        y = 1.3 - x;
        z = x * y;
        a = x / 2.3;
        std:: cout << x << " "<< y << " "<< z << " "<< a << std::endl;

    }
return 0;
}
```

5 - Operateurs : arithmétiques -> Exemple C/C++

```
int main(int , char *[])    {
        {
            //C CHAINE
            const char* b = " est ";
            char a[256];
            strcpy(a,"le canard ");
            strcat(a,b);
            strcat(a," tres bon");
            printf("%s\n",a);
        }
        {
            //C++ CHAINE
            std::string a = "le canard ";
            std::string b = " est ";
            a = a + b + " tres bon";
            std::cout << a << std::endl;
        }

    return 0;
}
```

5 – Operateur : comparaison

Type	Ensemble de valeurs	Opérateurs	Exemple
ENTIER REEL	comparaison mathématique	= (est égal à) < (est plus petit que) > (est plus grands que) <= (est plus petit ou égal à) >= (est plus grand ou égal à)	REEL x <- 0.1 ,y <- -1.1 ENTIER i <- 2 ,j <- 3 BOOLEAN res res <- x = x res <- x < i res <- i > x res <- i <= j res <- i >= j
CARACTERE CHAINE (CARACTERE*)	comparasion lexicographique	= (est égal à) < (est plus petit que) > (est plus grands que) <= (est plus petit ou égal à) >= (est plus grand ou égal à)	CARACTERE c1<-'a', c2 <- 'b' BOOLEAN res CHAINE a <- "le canard ", b <- " est " res <- a ="le canard" res <- c1 < c2 res <- a > b res <- c1 <= c2 res <- a >= b

CAMPUS EDUCTIVE AIX-EN-PROVENCE

Yantra Technologies

5 – Operateur : comparaison -> Exemple Python

```
if __name__ == "__main__":
#ENTIER
#REEL
    print("\nENTIER&REEL \n")
    x=0.1
    y=1.1
    i=int(2)
    j=int(3)
    res = False
    print(type(x),type(y),type(i),type(j),type(res))
    res = x==x
    print(res)
    res = x < i
    print(res)
    res = x > i
    print(res)
    res = i <= j
    print(res)
    res = i >= j
    print(res)

#CARACTERE
#CHAINE
    print("\nCHAINE&CARACTERE \n")
    c1 = 'a'
    c2 = 'b'
    res = bool(False)
    a = "le canard "
    b = " est "
    print(type(c1),type(c2),type(a),type(b),type(res))
    res = a == "le canard"
    print(res)
    res = c1 < c2
    print(res)
    res = a > b
    print(res)
    res = c1 <= c2
    print(res)
    res = a >= b
    print(res)
```

5 – Operateur : comparaison -> Exemple C

```c
int main(int , char *[]){

    //ENTIER
    //REEL
    printf("\nENTIER&REEL \n");
    float x=0.1;
    float y=1.1;
    int i=int(2);
    int j=int(3);
    bool res = 0;
    printf("%s\n", (res?"true":"false"));
    res = x < i;
    printf("%s\n", (res?"true":"false"));
    res = x > i;
    printf("%s\n", (res?"true":"false"));
    res = i <= j;
    printf("%s\n", (res?"true":"false"));
    res = i >= j;
    printf("%s\n", (res?"true":"false"));
```

```c
    //CARACTERE
    //CHAINE
    printf("\nCHAINE&CARACTERE \n");
    char c1 = 'a';
    char c2 = 'b';
    res = 0;
    const char* a = "le canard ";
    const char* b = " est ";

    res = strcmp(a,"le canard")==0;
    printf("%s\n", (res?"true":"false"));
    res = c1 < c2;
    printf("%s\n", (res?"true":"false"));
    res = strcmp(a,b)  > 0;
    printf("%s\n", (res?"true":"false"));
    res = c1 <= c2;
    printf("%s\n", (res?"true":"false"));
    res = strcmp(a,b)  >= 0;
    printf("%s\n", (res?"true":"false"));

    return 0;

}
```

5 - Operateur : comparaison -> Exemple C++

```cpp
int main(int , char *[]){

    //ENTIER
    //REEL
    std::cout << "\nENTIER&REEL \n";
    float x=0.1;
    float y=1.1;
    int i=int(2);
    int j=int(3);
    bool res = false;
    std::cout <<std::boolalpha<<res << std::endl;
    res = x < i;
    std::cout <<std::boolalpha<<res << std::endl;
    res = x > i;
    std::cout <<std::boolalpha<<res << std::endl;
    res = i <= j;
    std::cout <<std::boolalpha<<res << std::endl;
    res = i >= j;
    std::cout <<std::boolalpha<<res << std::endl;
```

```cpp
    //CARACTERE
    //CHAINE
    std::cout << "\nCHAINE&CARACTERE \n";
    char c1 = 'a';
    char c2 = 'b';
    res = false;
    std::string a = "le canard ";
    std::string b = " est ";

    res = a == "le canard";
    std::cout <<std::boolalpha<<res << std::endl;
    res = c1 < c2;
    std::cout <<std::boolalpha<<res << std::endl;
    res = a > b;
    std::cout <<std::boolalpha<<res << std::endl;
    res = c1 <= c2;
    std::cout <<std::boolalpha<<res << std::endl;
    res = a >= b;
    std::cout <<std::boolalpha<<res << std::endl;

    return 0;

}
```

5 - Operateur : logique

Type	Ensemble de valeurs	Opérateurs	Exemple
BOOLEAN	{VRAI ; FAUX}	NON (négation) ET (conjonction) OU (disjontion)	BOOLEAN a <- VRAI, b <- FAUX a <- NON b a <- a ET b b <- a OU b

Exemple Python

```
if __name__ == "__main__":
    a = True
    b = False
    print(a,b)
    a = not b
    print(a,b)
    a = a and b
    print(a,b)
    b = a or b
    print(a,b)
```

5 - Operateur : logique -> Exemple C++

```
int main(int , char *[]){
    bool a = true;
    bool b = false;
    std::cout <<std::boolalpha<<a << " "<<b << std::endl;
    a = !b;
    std::cout <<std::boolalpha<<a << " "<<b << std::endl;
    a = a && b;
    std::cout <<std::boolalpha<<a << " "<<b << std::endl;
    b = a || b;
    std::cout <<std::boolalpha<<a << " "<<b << std::endl;

    return 0;
}
```

6 - Expression

En algorithmique, une expression est une combinaison de valeurs, d'opérateurs et de fonctions qui, une fois évaluée, produit un résultat.

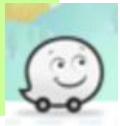
Les expressions peuvent être simples, comme une seule valeur, ou plus complexes, impliquant des opérations mathématiques, logiques ou d'autres types

(1 + 2) / x

(0 > x) ET (z < 3)

racineCarree(x) + 2 * sin(y)

7 - Instruction conditionnelle



La résolution des certains problèmes nécessite la mise en place d'un test pour savoir si l'on doit exécuter une tâche.

Si la condition est remplie alors on effectue la tâche, sinon on exécute (éventuellement) une autre tâche.

Dans un algorithme, on code la structure du « Si... Alors.. Sinon » sous la forme suivante :

SI *condition*
ALORS
    Tâche 1
    Tâche 2
    ...
SINON
    Tâche 1bis
    Tâche 2bis
    ...
FIN SI

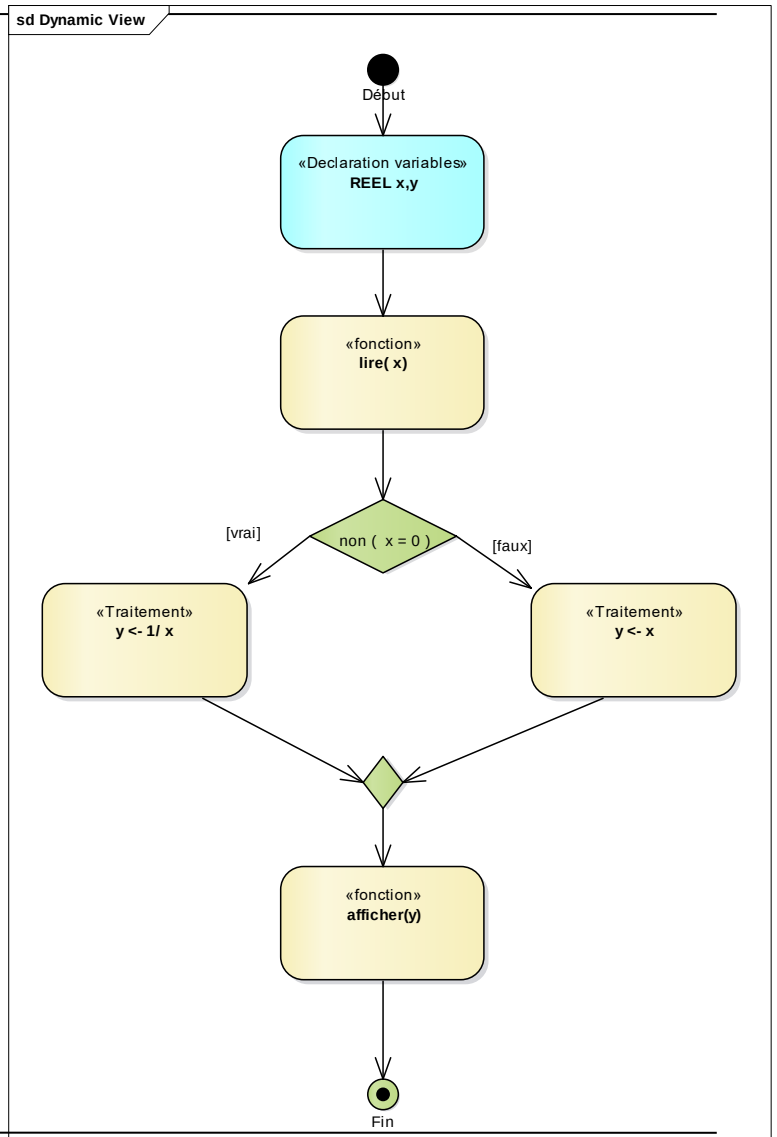
SI *condition*
ALORS
    Tâche 1
    Tâche 2
    ...
FIN SI

7 - Instruction conditionnelle -> Exemple

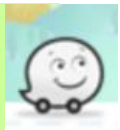
But calculer : $f(x) = {}^{1}/_{x}\ avec\ x \in \mathbb{R}\ et\ x \neq 0$
$f(x) = 0\ avec\ x \in \mathbb{R}\ si\ x = 0$

ALGORITHME

DEBUT
REEL x , y
lire (x)
SI non (*x = 0*)
ALORS
y <- 1/x
SINON
y <- x
FIN SI
afficher (y)
FIN

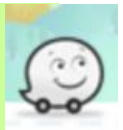


7 - Instruction conditionnelle -> Exemple Python



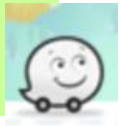
```
if __name__ == "__main__":
    x = 0.0
    y = 0.0
    x = float(input("x : "))
    if not (x == 0.0 ):
        y = 1 / x
    else :
        y = x
```

7 - Instruction conditionnelle -> Exemple C



```c
#include <stdio.h>

int main() {
    double x = 0.0;
    double y = 0.0;

    printf("Rentrer valeur de x : ") ;
    scanf("%lf",&x);

    if ( ! (x == 0.0) )
        y = 1. / x;
    else
        y = x;

    printf("y=%lf \n",y);

    if ( x != 0.0)
        y = 1. / x;
    else
        y = x;

    printf("y=%lf \n",y);

    if ( ! (x > 0.0 && x < 0.0 ) ) y = 1. / x;

    printf("y=%lf \n",y);
}
```

8 - Structure itérative



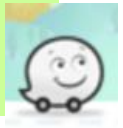
Il existe trois types de structures itératives :

- la structure « POUR… FIN POUR » :
  Le nombre de répétitions est connu (i= 1 à 10)

- la structure « TANT QUE… FAIRE » :
  Le nombre de répétitions n'est pas connu et peut être nul : 0 à n répétitions

- la structure « REPETER… JUSQU'À » :
  Le nombre de répétitions n'est pas connu mais ne peut pas être nul : 1 à n répétitions

8 - Structure itérative -> La boucle Pour



POUR variable **DE** valeur_depart **A** valeur_fin FAIRE

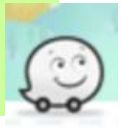
Tâche 1

Tâche 2

…

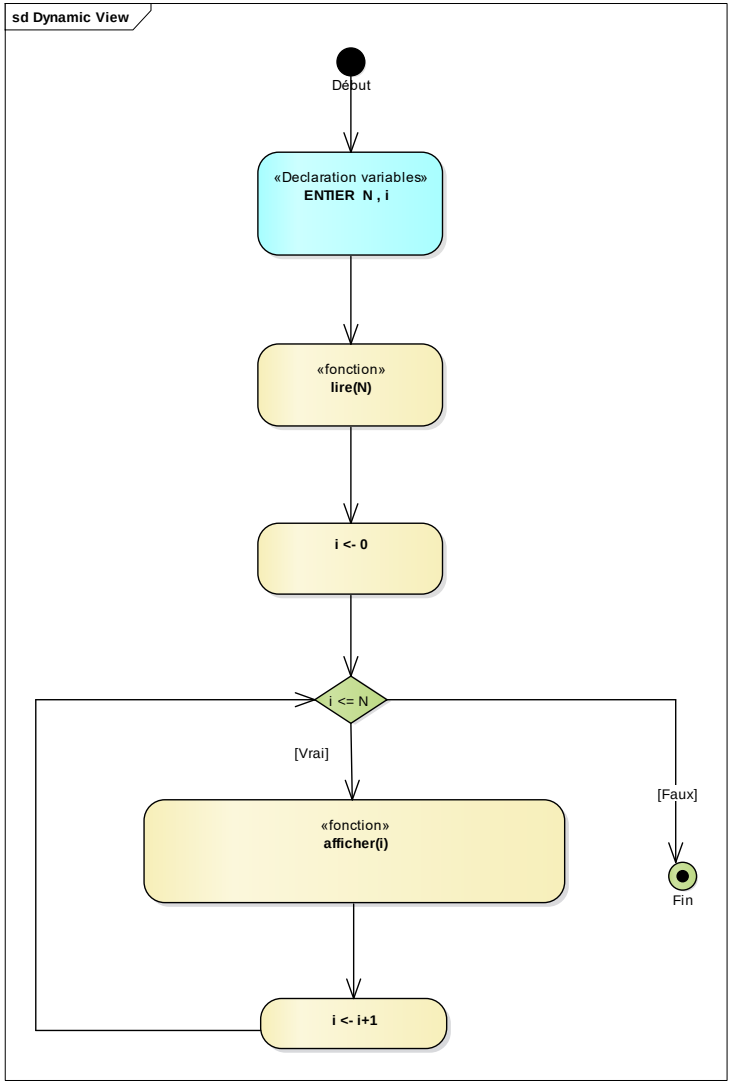
FIN POUR

8 - Structure itérative -> La boucle Pour : Exemple

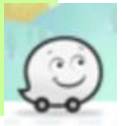


But calculer : afficher le N premier entier

```
ALGORITHME

DEBUT
    ENTIER  N , i

    lire ( N )
    POUR i DE 0 A N FAIRE
        afficher ( i )
    FIN POUR
FIN
```

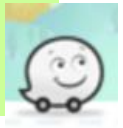


8 - Structure itérative
-> La boucle Pour : Exemple Python



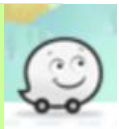
```
if __name__ == "__main__":

    N =0
    i =0
    N = int(input ("i :"))
    for i in range(N+1):
        print(i)
```

8 - Structure itérative
-> La boucle Pour : Exemple C



```c
#include <stdio.h>

int main() {
    int N =0;
    int i;

    printf("Rentrer valeur de N : ") ;
    scanf("%d",&N);
    for ( i=0; i <= N; ++i)
        printf ("%d ",i);
    return 0;
}
```

8 - Structure itérative -> La boucle Tant que



TANT QUE *condition* **FAIRE**

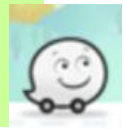
Tâche 1

Tâche 2

…

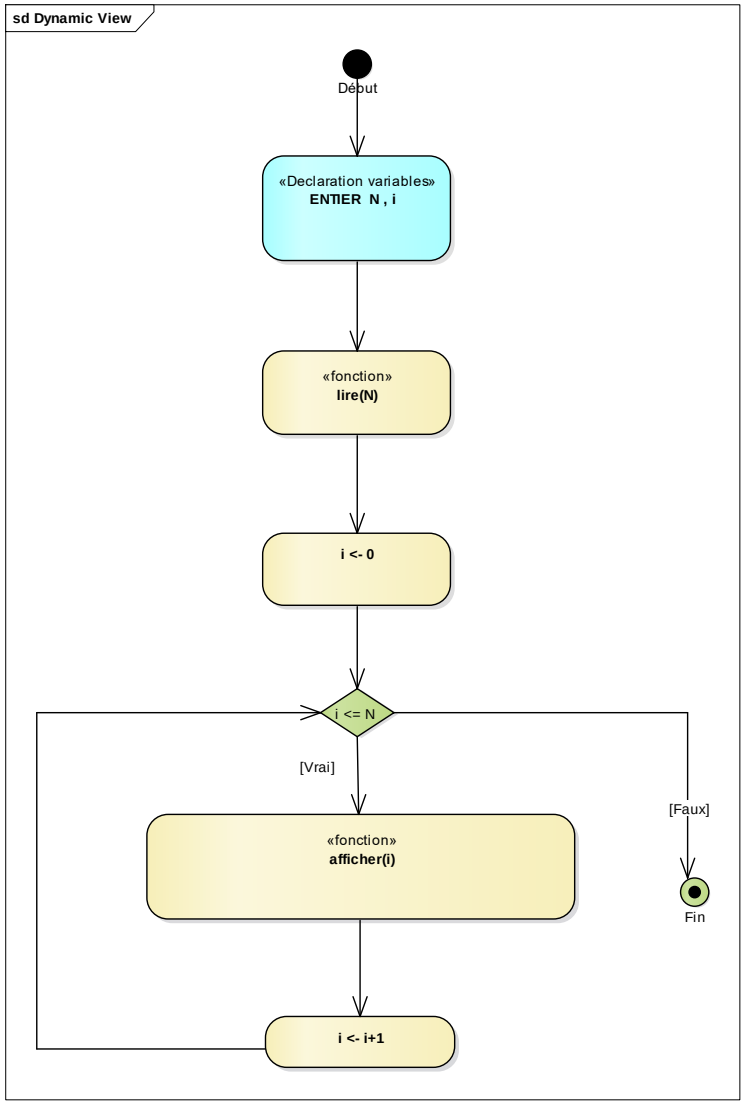
FIN TANT QUE

8 - Structure itérative -> La boucle Tant que : Exemple



But calculer : afficher les N premiers entiers

```
ALGORITHME
DEBUT
    ENTIER  N , i
    lire ( N )
    TANT QUE ( indice  <= N ) FAIRE
        afficher ( indice )
        indice <- indice + 1
    FIN TANT QUE
FIN
```



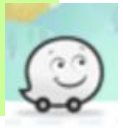
8 - Structure itérative
-> La boucle Tant que : Exemple Python

```
if __name__ == "__main__":
    N =0
    i =0
    N = int(input ("i :"))
    while i <= N:
        print(i)
        i = i + 1
```

8 - Structure itérative
-> La boucle Tant que : Exemple C++

```c
#include <stdio.h>

int main() {
    int N = 0;
    int i = 0;

    printf("Rentrer valeur de N : ") ;
    scanf("%d",&N);
    while ( i <= N) {
        printf ("%d ",i);
        i = i + 1;
    }
    return 0;
}
```

8 - Structure itérative -> La boucle REPETER



REPETER

Tâche 1

Tâche 2

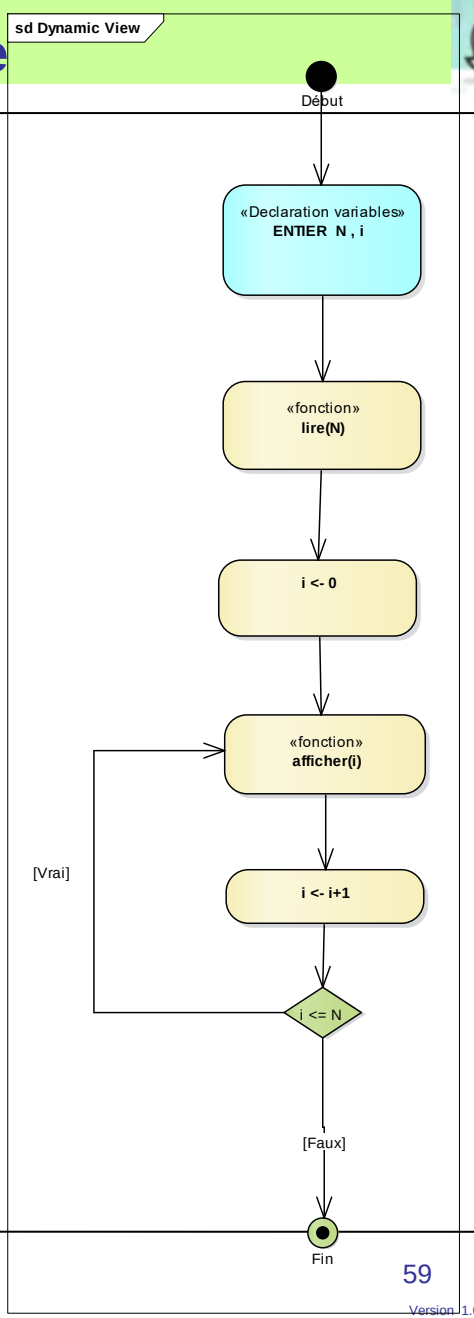
...

JUSQU'A *condition*

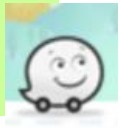
8 - Structure itérative
-> La boucle REPETER : Exemple

But calculer : afficher les N premiers entiers

```
ALGORITHME
DEBUT
    ENTIER  N , i

    lire ( N )
    indice  <- 0
    REPETER
        afficher( indice )
        indice <- indice + 1
    JUSQU'A ( indice > N )
FIN
```



8 - Structure itérative
-> La boucle REPETER : Exemple C



```c
#include <stdio.h>

int main() {
    int N = 0;
    int i = 0;

    printf("Rentrer valeur de N : ") ;
    scanf("%d",&N);
    do {
        printf ("%d ",i);
        i = i + 1;
    } while ( i <= N);
    return 0;
}
```

8 - Structure itérative
-> La boucle REPETER : Exemple Python

```
if __name__ =="__main__":
    N=0
    i=0
    N=int(input("N : "))
    print(i)
    i=i+1
    while (i <= N):
        print(i);
        i=i+1;
```

9 - Sous programme

Un algorithme ne devrait pas dépasser une page !

Pour respecter ce principe, il convient de nommer certaines séquences d'actions qui correspondront à des **procédures** ou à des **fonctions**

Ces **actions nommées** seront décrites dans des **algorithmes auxiliaires** et seront **utilisées dans un algorithme principal**.

9 - Sous programme -> Concept de bases : spécification

Paramètre d'entrée (donnée) : valeur(s) à fournir à l'algorithme

Pré-condition : condition que doivent vérifier les paramètres d'entrée

Paramètre de sortie (résultat) : valeur(s) que fourni l'algorithme

Post-condition : une condition que doivent vérifier les paramètres d'entrée et de sortie

9 - Sous programme -> Concept de bases : spécification

Paramètre entrer : X un REEL

Pré-condition : X != 0

Paramètre de sortie : Y un REEL

Post-condition : Y != 0

Post-condition : Y <- 1/X

9 - Sous programme -> Concept de bases : spécification

Paramètre entrer : X un REEL

Pré-condition : X >= 0

Paramètre de sortie : Y un REEL

Post-condition : Y >= 0

Post-condition : Y <- $\sqrt{x}$

9 - Sous programme -> Fonction

FONCTION *nom_de_la_fonction* (*liste_des_paramètres_formels*) **RENVOIE** *type_de_la_valeur_de_retour*

DEBUT

Tâche 1

Tâche 2

…

RENVOIE *une_valeur*

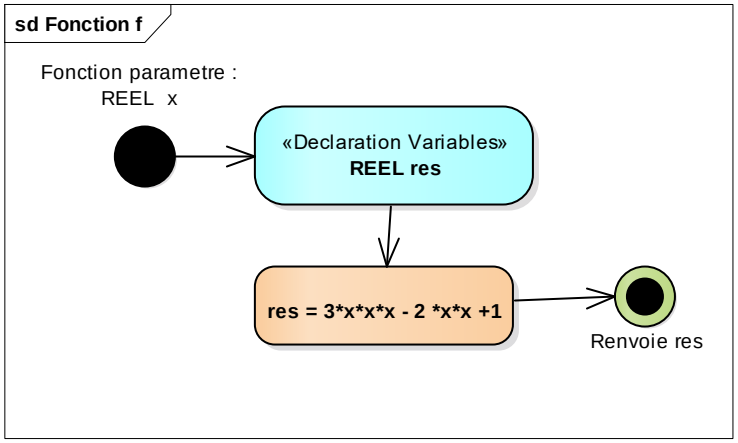
FIN

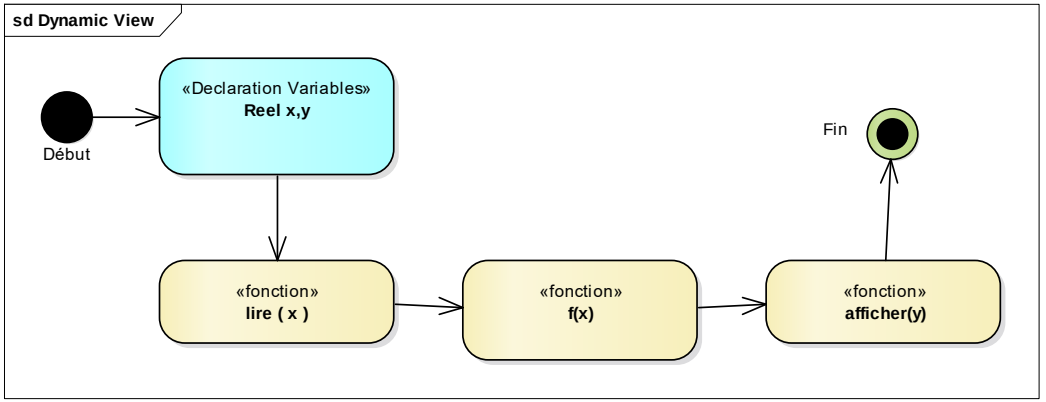
9 - Sous programme -> Fonction : Exemple

But calculer : $f(x) = 3\,x^3 - 2^2 + 1\ avec\ x\ \in\ \mathbb{R}$

```
FONCTION f ( REEL x ) RENVOIE REEL
DEBUT
    REEL  res
    res <- 3x3– 2x2 + 1
    RENVOIE res
FIN
```

```
ALGORITHME
DEBUT
    REEL  x,y
    lire(  x )
    y <- f(x)
    afficher ( y )
FIN
```





9 - Sous programme -> Fonction : Exemple

FONCTION f (REEL x) RENVOIE *REEL*

Paramètre entrer : x un REEL

Pré-condition : min < x < max

Paramètre de sortie : res un REEL

Post-condition : res <- $3x^3 - 2x^2 + 1$

9 - Sous programme
-> Fonction : Exemple Python

```
def f(x):
    res = 3*x*x*x-2*x*x+1
    return res

if __name__ == "__main__":

    x=0.0
    y=0.0

    print(type(x),type(y))

    x=float(input("rentrer valeur de x : "))

    y=f(x)

    print(y)
```

9 - Sous programme
-> Fonction : Exemple C

```c
#include <stdio.h>

double f(double x) {
    double res = 3*x*x*x - 2*x*x +1;
    return res;
    // <=>  return 3*x*x*x - 2*x*x +1;
}


int main() {
    double x = 0.0;
    double y = 0.0;

    printf("Rentrer valeur de x : ") ;
    scanf("%lf",&x);
    y = f(x);
    printf("y=%lf \n",y);
}
```

9 - Sous programme -> Fonction : Exemple

PROCEDURE *nom_procedure*
(*liste_paramètres_formels*)

DEBUT

Tâche 1

Tâche 2

...

FIN

9 - Sous programme-> PROCEDURE : Exemple 1

But calculer : $f(x) = 3\,x^3 - 2^2 + 1\ avec\ x\ \in\ \mathbb{R}$

```
PROCEDURE f (REEL ENTREE x, REEL SORTIE res)
DEBUT
        res <- 3x3– 2x2 + 1
FIN
```

```
ALGORITHME
DEBUT
        REEL x , y
        lire( x )
        f(x,y)
        afficher( y )
FIN
```

9 - Sous programme
-> PROCEDURE : Exemple 1 - Python

```
def f(x,res):
    res[0] = 3*x*x*x-2*x*x+1


if __name__ == "__main__":

    x=0.0
    y=[0.0]

    print(type(x),type(y))

    x=float(input("rentrer valeur de x : "))

    f(x,y)

    print(y," " ,y[0])
```

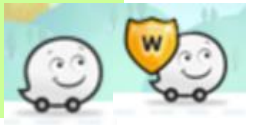
9 - Sous programme
-> PROCEDURE : Exemple 1 - C

```c
#include <stdio.h>

void f(double x, double * y) {

      (*y) = 3*x*x*x - 2*x*x +1;

}


int main() {
     double x = 0.0;
     double y = 0.0;

     printf("Rentrer valeur de x : ") ;
     scanf("%lf",&x);
     f(x,&y);
     printf("y=%lf \n",y);
}
```

9 - Sous-programme
-> PROCEDURE & FONCTION : Récursivité

La récursivité est un concept où une fonction s'appelle elle-même pour résoudre un problème plus large en le décomposant en sous-problèmes plus petits et similaires.

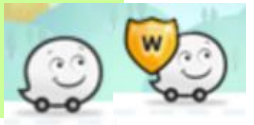
9 - Sous-programme
-> FONCTION : Exemple : Récursivité



```
FONCTION boucle_recursive (ENTIER ENTREE i)
            RENVOIE ENTIER
DEBUT
      SI i = 0 ALORS
            RENVOIE 0
      FIN SI
     afficher( boucle_recursive(i-1) )
     RENVOIE i
FIN
```

```
ALGORITHME
DEBUT
      ENTIER N
      lire( N)
      afficher( boucle_recursive(N))
FIN
```

9 - Sous-programme
-> FONCTION : Exemple : Récursivité - Python



```
def boucle_recursive (N):
    if   N == 0 :
         return 0
    print ( boucle_recursive(N-1) )
    return N


if __name__ == "__main__":
    N = int(input ("i :"))
    print(boucle_recursive (N))
```

9 - Sous-programme
-> FONCTION : Exemple : Récursivité – C++

```
int boucle_recursive(int i ) {
    if ( i == 0 ) return i;
    printf ("%d ",boucle_recursive( i-1 ));
    return i;
}


int main() {
    int N = 0.0;
    printf("Rentrer valeur de N : ") ;
    scanf("%d",&N);
    printf ("\n fin = %d \n",boucle_recursive( N ));
    return 0;
}
```

9 - Sous-programme
-> FONCTION : Exemple : Récursivité – C++

```
unsigned int boucle_recurcive(unsigned i) {
    if (i == 0) return 0;
    std::cout <<  boucle_recurcive(i-1) << std::endl;
    return i;
}
int main(int , char *[]) {
    unsigned  N;
    std::cin >> N;
    std::cout << boucle_recurcive(N) << std::endl;
    return 0;
}
```

9 - Sous-programme
-> PROCEDURE & FONCTION : Récursivité

Avantages de la Récursivité

- Permet de résoudre des problèmes complexes en les décomposant en problèmes plus simples.
- Peut rendre le code plus lisible et élégant pour certains problèmes.

Limitations de la Récursivité

- Peut-être moins efficace que les approches itératives pour certains problèmes.
- Une mauvaise utilisation peut entraîner une consommation excessive de mémoire et une récursion infinie.

10 - Type complexe : Les structures

Une structure est un type composite formé par plusieurs types groupés ensemble

```
STRUCTURE Pixel
        REEL x
        REEL y
FIN STRUCTURE

TYPE Pixel Ecran[1024]
```

```
ALGORITHME
DEBUT
    Pixel p
    Ecran e
    ENTIER index
    lire ( p.x )
    lire ( p.y )
    POUR index DE 0 A 1024 FAIRE
        e[i].x <- p.x * ( index+1)
        e[i].y <- p.y * ( index+1)
    FIN POUR
    POUR index DE 0 A 1024 FAIRE
        afficher ( index, e[i].x , e[i].y)
    FIN POUR
FIN
```

10 - Type complexe : Les structures -> Exemple Python

```
from copy import deepcopy

class Pixel:
    def __init__(self):
        self.x=0.0
        self.y=0.0

Ecran = [Pixel()]*2

if __name__ == "__main__":
        p = Pixel()
        e = deepcopy(Ecran)
        index=0
        print(type(p),type(e))
        p.x = float(input("x :"))
        p.y = float(input("y :"))
        for index  in range(len(e)):
            e[index].x= p.x * (index+1)
            e[index].y= p.y * (index+1)

        for index  in range(0,len(e)):
            print(index,e[index].x,e[index].y)
```

10 - Type complexe : Les structures -> Exemple C

```c
#include <stdio.h>

struct Pixel{
    double x;
    double y;
};

typedef  struct Pixel Ecran[1024];

int main() {
    struct Pixel p;
    Ecran e;

    unsigned int index;
    printf("Rentrer valeur d'un Pixel : ") ;
    scanf("%lf %lf",&p.x,&p.y);
    for ( index=0; index < 1024; ++index) {
        e[index].x=p.x *(index+1);
        e[index].y=p.y *(index+1);;
    }
    for ( index=0; index < 1024; ++index) {
        printf("%d %lf %lf\n",index, e[index].x,e[index].y);
    }

    return 0;
}
```

10 - Type complexe : Les structures -> Exemple C++

```
class Pixel {
public:
    double x;
    double y;

    Pixel(double px=0.0, double py=0.0):
        x(px),y(py){}
};

using Ecran = Pixel[1024];


int main(int , char *[]) {

    Pixel p;
    Ecran e;
    std::cin >> p.x;
    std::cin >> p.y;
    for (int index =0; index < 1024; ++index){
        e[index].x=p.x*(index+1);
        e[index].y=p.y*(index+1);
    }

    for (int index =0; index < 1024; index = index+100) {
        std::cout << e[index].x << " " << e[index].y << std::endl;
    }

    return 0;
}
```

10 - Type complexe : Tableau Dynamique Unidimensionnel -> Les listes (ou Vecteurs)

Un tableau dynamique ou liste ou vecteur est une séquence de données de même type, la taille de la séquence est variable

Opération	Spécification	Exemple
Déclaration	Vecteur *TYPE* nom	Vecteur REEL v
Initialisation	Vecteur *TYPE* nom(taille, valeur)	Vecteur REEL w(5,1)
Copie	nom 1 <- nom2	v <- w
Accès élément	Nom[index]	v[i]
Accès à la taille	longueur(nom) : ENTIER	longueur (v)
Test vecteur vide	vide(nom) : BOOLEAN	SI vide(v) ALORS etendre(v,5,0) FIN SI
Ajout d'élément à la fin	etendre(nom,taille,valeur)	etendre(v,5,0)

10 - Type complexe : Tableau Dynamique Unidimensionnel -> Les listes (ou Vecteurs) : Exemple

```
ALGORITHME
DEBUT
    Vecteur REEL v
    ENTIER N
    ENTIER index
    lire ( N )
    etendre (v,N)
    POUR index DE 0 A N-1 FAIRE
        v[index] <- index * 2
    FIN POUR
    POUR index DE 0 A N-1 FAIRE
        afficher (v[index])
    FIN POUR

FIN
```

10 - Type complexe : Tableau Dynamique Unidimensionnel -> Les listes (ou Vecteurs) : Exemple Python

```
if __name__ == "__main__":
    liste=[]
    N=0
    index = 0

    N = int(input("rentrer la taille du vecteur : "))
    liste.extend([0.0]*N)

    for index in range(len(liste)):
        liste[index]= index *2

    for index in range(len(liste)):
        print(liste[index])
```

10 - Type complexe : Tableau Dynamique Unidimensionnel -> Les listes (ou Vecteurs) : C

```c
int main(int , char *[]) {
    double*  v;
    unsigned N;
    unsigned index;

   printf("Entrer la taille du vecteur");
   scanf("%d",&N);
   getchar();

    v = (double*) malloc( sizeof(double)*N);

    for (index=0; index < N ;++index) {
        v[index]=index*2;
    }

    for (index=0; index < N ;++index) {
         printf("v[%u]-> %f\n", index,v[index]);
    }
    free(v);
    return 0;
}
```

10 - Type complexe : Tableau Dynamique Unidimensionnel -> Les listes (ou Vecteurs) : Exemple C++

```
int main(int , char *[]) {
    std::vector<double> v;
    unsigned N;
    unsigned index;

    std::cout << "Entrer la taille du vecteur";
    std::cin >> N;
    v.resize(N,0);
    for (index=0; index < N ;++index) {
        v[index]=index*2;
    }

    for (index=0; index < N ;++index) {
         std::cout <<"v[" << index << "] -> " << v[index] << std::endl;
    }
    return 0;
}
```

10 - Type complexe : Chaine

Une chaine est un tableau dynamique unidimensionnel composé de caractères ascii.

Les chaines permettent des comparaisons lexicographiques

10 - Type complexe : Chaine

Opération	Spécification	Exemple
Déclaration	CHAINE nom	CHAINE c
Initialisation	CHAINE nom <- *chaine constante*	CHAINE s <- "canard"
Copie	nom 1 <- nom2	c <- s
Accès élément	nom[index]	s[i]
Accès à la taille	longueur(nom) : ENTIER	longueur (s)
Test vecteur vide	vide(nom) : BOOLEAN	SI vide(s) ALORS s <- "roti" FIN SI
Ajout d'une chaine Concaténation	nom <- nom1 + nom2	c = " j'aime le " + c + " " + s
Comparaison	=,<>,<=,>=	

10 - Type complexe : Chaine -> Exemple

```
ALGORITHME
    Chaine v <- "Le canard est "
    Chaine suite
DEBUT
    lire ( suite )
    v <- v+ suite
    POUR index DE 0 A longueur(v) -1 FAIRE
        afficher(v[index] )
    FIN POUR
    afficher (v)
FIN
```

10 - Type complexe : Chaine -> Exemple Python

```
if __name__ == "__main__":
    v="le canard est "
    suite=""
    suite = input("rentrer la suite de la phrase : "+v)
    v = v + suite;
    for index in range(len(v)):
        print(index," ",v[index])

    print(v)
```

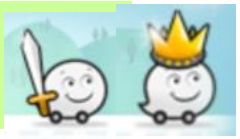
10 - Type complexe : Chaine -> Exemple C

```c
#include <stdio.h>


int main() {

    char v[256] = {'l','e',' ','c','a','n','a','r','d',' ','e','s','t',' ','\0'};
    char suite[256];
    printf(" rentrer la suite de la pharse '%s' : ",v);
    scanf("%255[^\n]",suite);
    strcat(v,suite);
    printf("%s",v);
    return 0;
}
```

10 - Type complexe : Chaine -> Exemple C++

```
int main(int , char *[]) {
    std::string v = "le canard est ";
    std::string suite;
    std::cin >> suite;
    v += suite;

    std::cout << v << std::endl;
    return 0;
}
```

10 - Type complexe : Les pointeurs

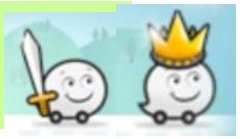


Un **pointeur** est une **adresse mémoire**: il permet de désigner directement une zone de la mémoire et donc l'objet dont la valeur est rangée à cet endroit.

- Un pointeur est souvent typé de manière à préciser quel type d'objet il désigne dans la mémoire.

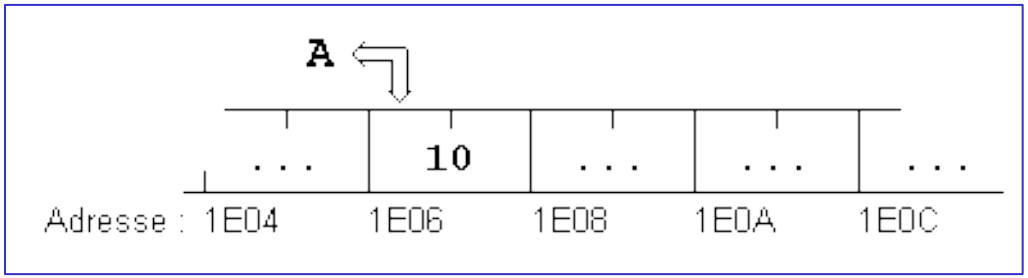
- Un type pointeur est défini par **le symbole ^ ou *** suivi par le nom du type de l'objet pointé

10 - Type complexe : Les pointeurs - Adressage



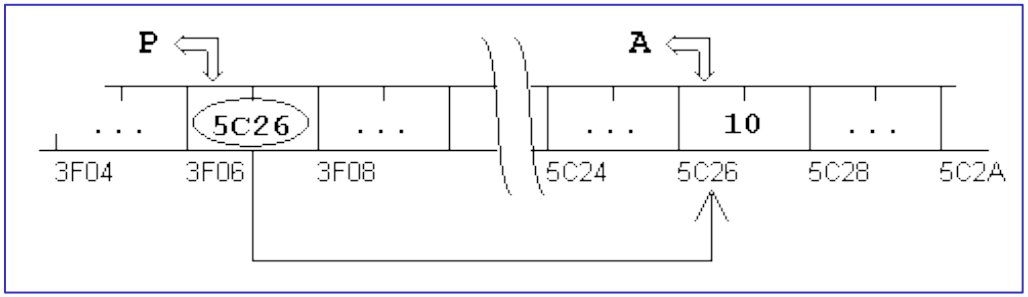
Adressage direct

```
ENTIER A
A <-10

afficher(a)
```

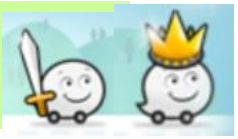


Adressage indirect

```
ENTIER *P
P <- &A

afficher(*p)
```

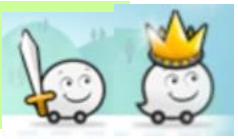


10 - Type complexe : Les pointeurs – Adressage
-> Exemple 1 - Python



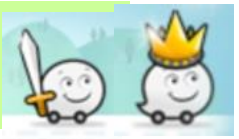
```
if __name__ == "__main__":
    a = 10
    print (a," ",id(a)," " ,hex(id(a) ))

    p = [a]

    print (p,id(p)," " ,hex(id(p) ))
    print (p[0],id(p[0])," " ,hex(id(p[0]) ))

    p[0]=20
    print (p[0],id(p[0])," " ,hex(id(p[0]) ))
    print (a," ",id(a)," " ,hex(id(a) ))
```

10 - Type complexe : Les pointeurs – Adressage
-> Exemple 1 - Python



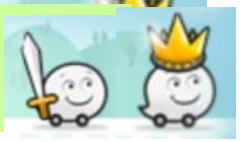
```
if __name__ == "__main__":
    a = 10
    print (a," ",id(a)," " ,hex(id(a) ))

    p = [a]

    print (p,id(p)," " ,hex(id(p) ))
    print (p[0],id(p[0])," " ,hex(id(p[0]) ))

    p[0]=20
    print (p[0],id(p[0])," " ,hex(id(p[0]) ))
    print (a," ",id(a)," " ,hex(id(a) ))
```

10 - Type complexe : Les pointeurs – Adressage
-> Exemple 1 – C/C++

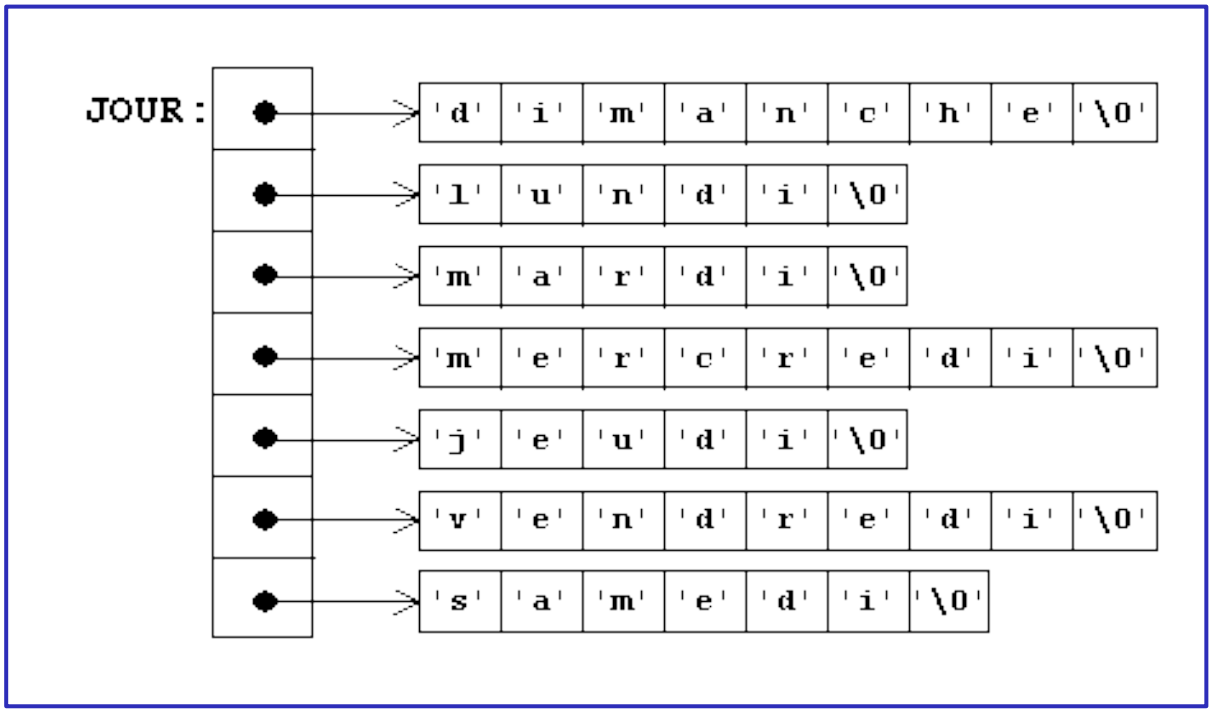


```
int main(int , char *[]) {
    //C
    {
        int a= 10;
        int *p =&a;
        printf("%d %p \n",a,&a);
        printf("%d %p %p\n",*p,p,&p);
        printf("%d %p %p\n",p[0],p,&p);

        p[0] = 20;
        printf("%d %p \n",a,&a);
        printf("%d %p %p\n",p[0],p,&p);
    }

    //C++
    {
        int a= 10;
        int *p =&a;
        std::cout << a << " " << &a << std::endl;
        std::cout << *p << " " << p << " " << &p << std::endl;
        std::cout << p[0]<< " " << p<< " "<< &p<< std::endl;

        p[0] = 20;
        std::cout << a << " " << &a << std::endl;
        std::cout << p[0]<< " " << p<< " "<< &p<< std::endl;
    }

   return 0;
}
```

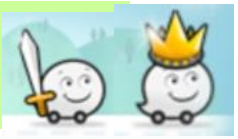
10 - Type complexe : Les pointeurs – Adressage -> Exemple 2



```
CHAINE *JOUR[7] =
 { "dimanche" , "lundi" , "mardi" , "mercredi" , "jeudi" , "vendredi" , "samedi" }
```

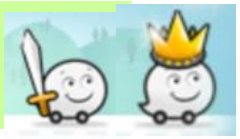


10 - Type complexe : Les pointeurs – Adressage
-> Exemple 2 - Python



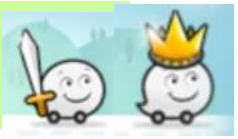
```
if __name__ == "__main__":
    JOUR=["dimanche" , \
          "lundi" , \
          "mardi" , \
          "mercredi" , \
          "jeudi" , \
          "vendredi" , \
          "samedi" ]
    for index in range(len(JOUR)):
        print (index, " ", JOUR[index])
        for index2 in range(len(JOUR[index])):
            print (index,index2, " ", JOUR[index][index2])
```

10 - Type complexe : Les pointeurs – Adressage -> Exemple C++



```
int main(int , char *[]) {

    std::array<std::string,7>JOUR ({ "dimanche",
                                     "lundi",
                                     "mardi",
                                     "mercredi",
                                     "jeudi",
                                     "vendredi",
                                     "samedi"});
    for (unsigned i=0; i < JOUR.size();++i)
    {
        std::cout << JOUR[i] << std::endl;
        for ( char c : JOUR[i])
        {
            std::cout << c << std::endl;
         }
     }
    return 0;
}
```

10 - Type complexe : Les pointeurs – Adressage -> Exemple C



```c
int main(int , char *[]) {
    const unsigned N = 7;
    const unsigned T = 10;
    char* JOUR[N];
    for (unsigned i=0; i < N;++i)
        JOUR[i] = (char*)malloc(sizeof(char)* T);
    strcpy (JOUR[0],"dimanche");
    strcpy (JOUR[1],"lundi");
    strcpy (JOUR[2],"mardi");
    strcpy (JOUR[3],"mercredi");
    strcpy (JOUR[4],"jeudi");
    strcpy (JOUR[5],"vendredi");
    strcpy (JOUR[6],"samedi");
    for (unsigned i=0; i < N;++i)
    {
        printf ("%s\n",JOUR[i]);
        for (unsigned j=0; j < strlen(JOUR[i]);++j)
        {
            printf ("%c\n",JOUR[i][j]);
         }

     }
    return 0;
}
```

11 - Complexité algorithmique : Analyse de la complexité d'un algorithme

L'**analyse de la complexité d'un algorithme** consiste en l'étude formelle de la quantité de ressources (par exemple de **temps** ou **d'espace**) nécessaire à l'exécution de cet algorithme

- la **complexité en temps** est une mesure du temps utilisé par un algorithme, exprimé comme fonction de la taille de l'entrée. Le temps compte le nombre d'étapes de calcul avant d'arriver à un résultat.

- **La complexité spatiale** permet d'évaluer la consommation de l'algorithme en espace mémoire

https://fr.wikipedia.org/wiki/Analyse_de_la_complexit%C3%A9_des_algorithmes

11 - Complexité algorithmique : la complexité en temps

On peut distinguer deux formes de complexité en temps :

- la complexité dans le **meilleur des cas**
- la complexité dans le **pire des cas**
- la complexité dans le cas général

11 - Complexité algorithmique : Ordre de grandeur

Pour comparer des algorithmes, suffit de connaitre l'ordre de grandeur asymptotique, noté O (« grand O »).

Soit une fonction T(n)

- $T_1(n) = 7 = \mathcal{O}(1)$
- $T_2(n) = 12n + 5 = \mathcal{O}(n)$
- $T_3(n) = 4n^2 + 2n + 6 = \mathcal{O}(n^2)$
- $T_4(n) = 2 + (n-1) \times 5 = \mathcal{O}(n)$

11 - Complexité algorithmique : Analyse de la complexité d'un algorithme

Ordre de grandeur du temps nécessaire à l'exécution d'un algorithme d'un type de complexité

Temps	Type de complexité	Temps pour n = 5	Temps pour n = 10	Temps pour n = 20	Temps pour n = 50	Temps pour n = 250	Temps pour n = 1 000	Temps pour n = 10 000	Temps pour n = 1 000 000	Problème exemple
$O(1)$	complexité constante	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	accès à une cellule de tableau
$O(\log(n))$	complexité logarithmique	10 ns	10 ns	10 ns	20 ns	30 ns	30 ns	40 ns	60 ns	recherche dichotomique
$O(\sqrt{n})$	complexité racinaire	22 ns	32 ns	45 ns	71 ns	158 ns	316 ns	1 µs	10 µs	test de primalité naïf
$O(n)$	complexité linéaire	50 ns	100 ns	200 ns	500 ns	2.5 µs	10 µs	100 µs	10 ms	parcours de liste
$O(n\ log^*(n))$	complexité quasi-linéaire	50 ns	100 ns	200 ns	501 ns	2.5 µs	10 µs	100,5 µs	10,05 ms	triangulation de Delaunay
$O(n\log(n))$	complexité linéarithmique	40 ns	100 ns	260 ns	850 ns	6 µs	30 µs	400 µs	60 ms	tris par comparaisons optimaux (comme le tri fusion ou le tri par tas)
$O(n^2)$	complexité quadratique (polynomiale)	250 ns	1 µs	4 µs	25 µs	625 µs	10 ms	1 s	2.8 heures	parcours de tableaux 2D
$O(n^3)$	complexité cubique (polynomiale)	1.25 µs	10 µs	80 µs	1.25 ms	156 ms	10 s	2.7 heures	316 ans	multiplication matricielle naïve
$2^{\text{poly}(\log(n))}$	complexité sous-exponentielle	30 ns	100 ns	492 ns	7 µs	5 ms	10 s	3.2 ans	10^{20} ans	factorisation d'entiers avec GNFS (le meilleur algorithme connu en 2018)
$2^{\text{poly}(n)}$	complexité exponentielle	320 ns	10 µs	10 ms	130 jours	10^{59} ans	...	...	...	problème du sac à dos par force brute
$O(n!)$	complexité factorielle	1.2 µs	36 ms	770 ans	10^{48} ans	...	...	...	...	problème du voyageur de commerce avec une approche naïve
$2^{2^{\text{poly}(n)}}$	complexité doublement exponentielle	4.3 s	10^{278} ans	...	...	...	...	...	...	décision de l'arithmétique de Presburger

$log^*(n)$ est le logarithme itéré.

https://docplayer.fr/176318917-Chapitre-3-introduction-a-l-algorithmique.html

11 - Complexité algorithmique : Exemple

```
def factorielle(n):
    fact = 1
    i = 2

    while i <= n:
        fact = fact * i
        i = i + 1
    return fact
```

```
.
affectation : 1
affectation : 1
.
itérations : au plus x(n - 1)
    comparaison : 1
    multiplication + affectation : 2
    addition + affectation : 2
```

https://info.blaisepascal.fr/nsi-complexite-dun-algorithme/

11 - Complexité algorithmique : Exemple

Exemple : Considérons le programme de détermination du nombre d'occurrences dans un tableau de type `list`. On veut déterminer la complexité de la fonction `nbre_occurrences`.

```
2 def nbre_occurrences(x:object, t:list) -> int:
3     c = 0
4     for i in range(len(t)):
5         if t[i] == x:
6             c = c + 1
7     return c
```

Complexité :

- L'affectation `c = 0` compte pour une opération élémentaire (on dit qu'elle est en $\mathcal{O}(1)$, car majorée par une constante).
- La boucle bornée `for i in range(len(t))` se déroule `len(t)` fois, où `len(t)` est une opération élémentaire.
- L'instruction de contrôle `if`, l'accès à `t[i]`, ainsi que le test d'égalité `t[i] == x` se déroulent en temps constant (3 opérations).
- Le calcul de l'expression `c+1` ainsi que l'affectation `c = c+1` se déroulent en temps constant (2 opérations).
- `return c` est une opération élémentaire.

Conclusion : Si `n = len(t)` (*taille de l'entrée*), le traitement se réalise en un maximum de $5n+2$ opérations élémentaires. Or, $5n+2 \leq 6n$, donc que **la complexité temporelle dans le pire des cas** de la fonction `nbre_occurrences` est $\mathcal{O}(n)$. On dit aussi que la complexité temporelle est **linéaire**.

https://cahier-de-prepa.fr/psi-michelet/download?id=239

12 – Algorithmes de recherches

La recherche est l'une des opérations les plus fondamentales en informatique et en traitement de données.

Les algorithmes de recherche permettent de localiser des éléments dans un ensemble de données : Google, Base de données, IA ….

Donc un algorithme de recherche est une séquence d'instructions conçue pour trouver la position ou la valeur d'un élément cible au sein d'un ensemble de données.

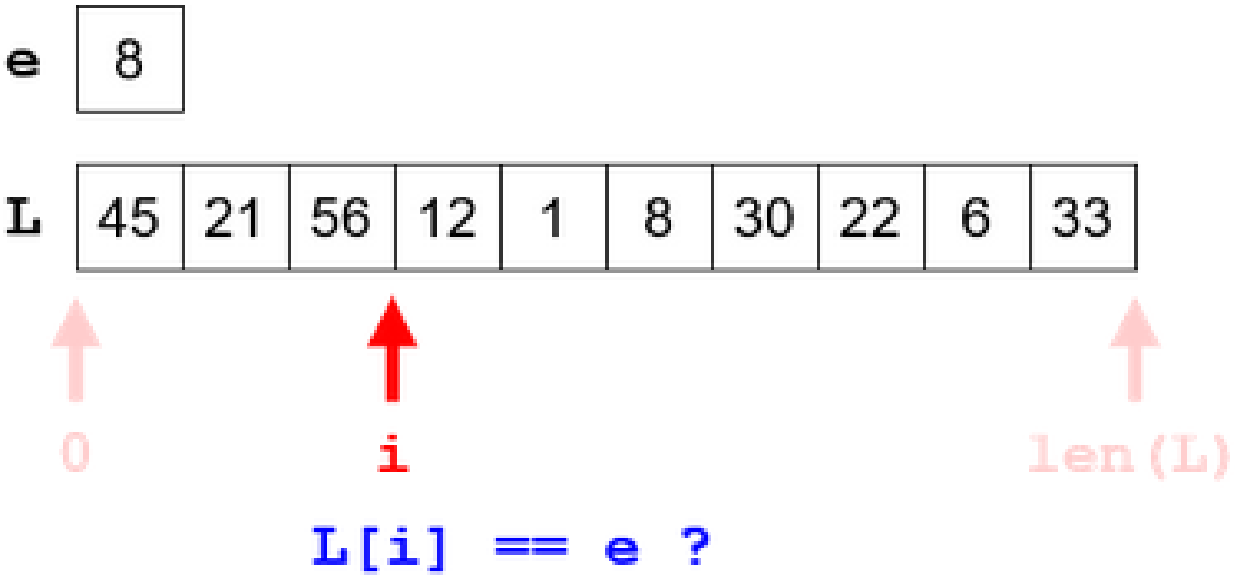
12 – Algorithmes de recherches -> Types d'Algorithmes de Recherche

- **Recherche Séquentielle (Linéaire) :** Parcourt les éléments un par un jusqu'à trouver la cible.

- **Recherche Binaire (dichotomique) :** Applique une stratégie de "diviser pour régner" dans un ensemble trié pour trouver la cible.

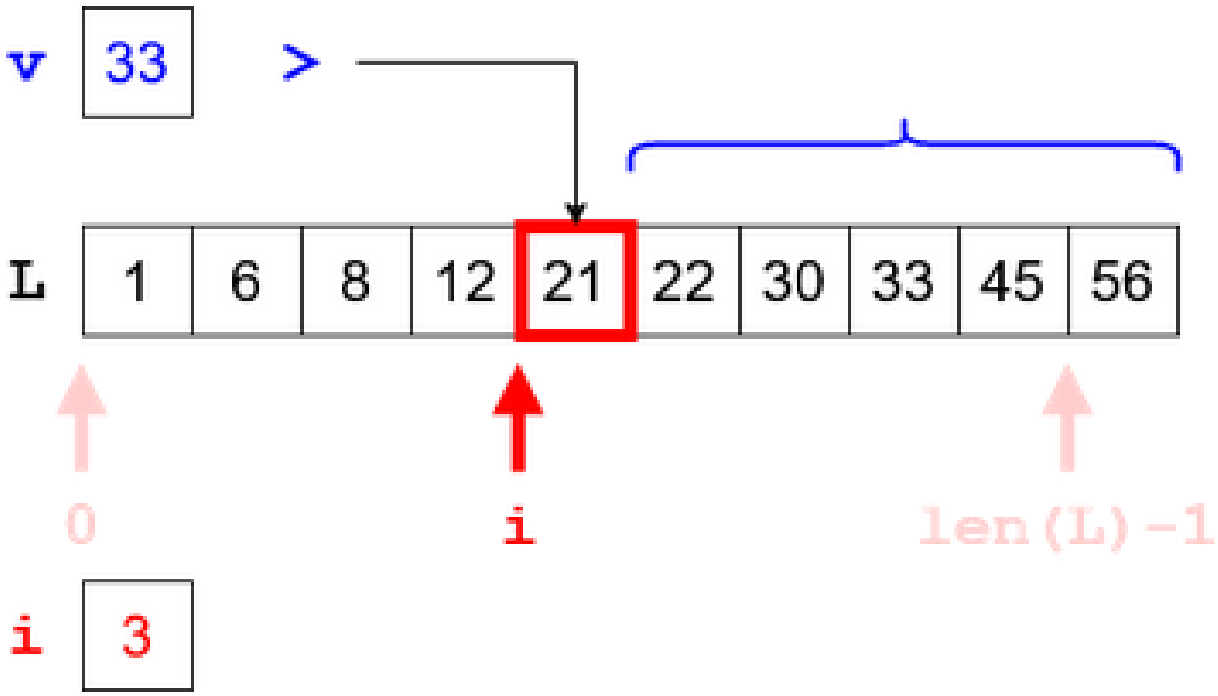
https://fr.wikipedia.org/wiki/Algorithme_de_recherche

12 –Algorithmes de recherches -> Recherche Séquentielle



https://info.blaisepascal.fr/nsi-parcours-sequentiel-dun-tableau/

12 –Algorithmes de recherches -> Recherche dichotomique



https://info.blaisepascal.fr/nsi-recherche-dichotomique/

13 - Algorithmes de tri : Définition

Un algorithme de tri est, en informatique ou en mathématiques, un algorithme qui permet d'organiser une collection d'objets selon **un ordre déterminé.**

Les objets à trier font partie d'un ensemble muni d'une relation d'ordre

Les ordres les plus utilisés sont l'ordre numérique et l'ordre lexicographique (dictionnaire).

Suivant la relation d'ordre considérée, une même collection d'objet peut donner lieu à divers arrangements, pourtant il est possible de définir un algorithme de tri indépendamment de la fonction d'ordre utilisée.

Celui-ci ne fera qu'utiliser une certaine **fonction d'ordre** correspondant à une relation d'ordre qui doit **permettre de comparer** tout **couple d'éléments** de la collection.

13 - Algorithmes de tri : Complexité

Nom	Cas optimal	Cas moyen	Pire des cas	Complexité spatiale	Stable
Tri rapide	$n \log n$	$n \log n$	n^2	$\log n$ en moyenne, n dans le pire des cas ; variante de Sedgewick : $\log n$ dans le pire des cas	Non
Tri fusion	$n \log n$	$n \log n$	$n \log n$	n	Oui
Tri par tas	$n \log n$	$n \log n$	$n \log n$	1	Non
Tri par insertion	n	n^2	n^2	1	Oui
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	Non
Tri par sélection	n^2	n^2	n^2	1	Non
Timsort	n	$n \log n$	$n \log n$	n	Oui
Tri de Shell	n	$n \log^2 n$ ou $n^{3/2}$	$n \log^2 n$pour la meilleure suite d'espacements connue	1	Non
Tri à bulles	n	n^2	n^2	1	Oui
Tri arborescent	$n \log n$	$n \log n$	$n \log n$ (arbre équilibré)	n	Oui
Smoothsort	n	$n \log n$	$n \log n$	1	Non
Tri cocktail	n	n^2	n^2	1	Oui
Tri à peigne	n	$n \log n$	n^2	1	Non
Tri pair-impair	n	n^2	n^2	1	Oui

13 - Algorithmes de tri : Tri par sélection

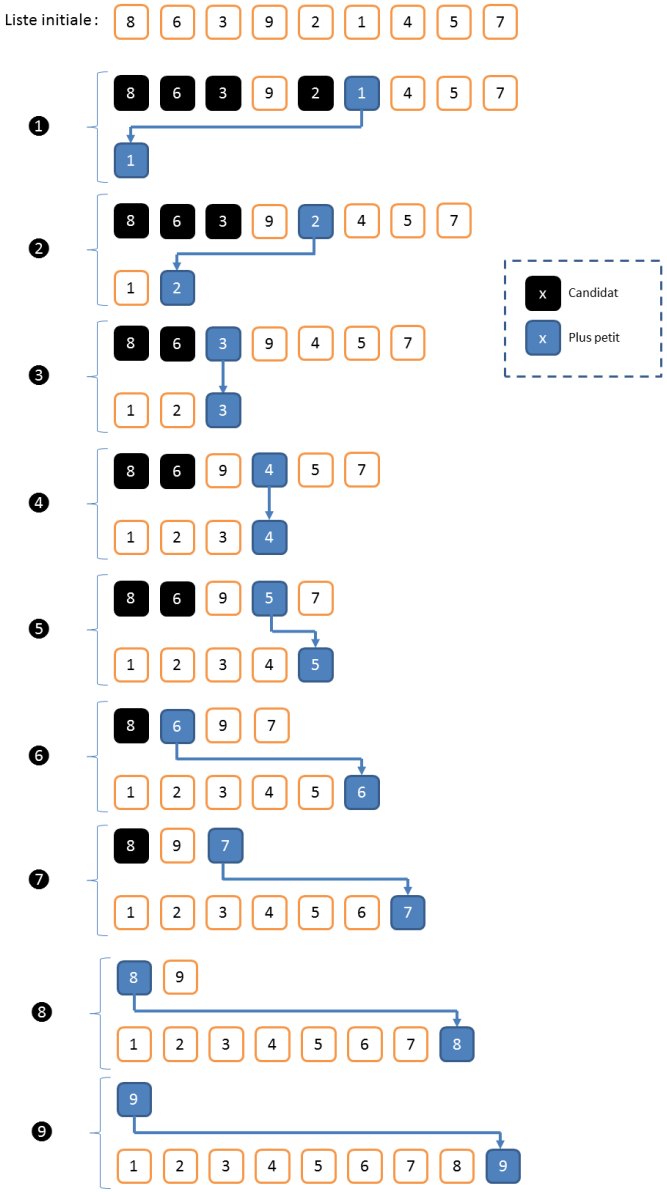
Le tri par sélection (ou tri par extraction) est un algorithme de tri par comparaison.

Cet algorithme est simple, mais considéré comme inefficace car il s'exécute en temps quadratique (n2) en le nombre d'éléments à trier, et non en temps pseudo linéaire (n ou nlog(n)).

Le principe du tri par sélection/échange (ou tri par extraction) est d'aller chercher le plus petit élément du vecteur pour le mettre en premier, puis de repartir du second élément et d'aller chercher le plus petit élément du vecteur pour le mettre en second, etc..

13 - Algorithmes de tri : Tri par sélection

Tri par sélection



https://www.podcastscience.fm/dossiers/2014/09/04/les-tris/

13 - Algorithmes de tri : Tri par sélection

```
PROCEDURE echanger(Vecteur ENTIER T,ENTIER a, ENTIER b)
                          ENTIER tmp
  DEBUT
     tmp <- a
     a   <- b
     b   <- tmp
  FIN

PROCEDURE tri_selection(Vecteur ENTIER T, ENTIER n)
  ENTIER min,i,j
  DEBUT
    POUR i DE 0 A n - 2
       min <- i
       POUR j DE i + 1 A n – 1 FAIRE
          SI t[j] < t[min] ALORS min <-j
       FIN POUR
       SI min ≠ i ALORS echanger(T,min,i)
    FIN POUR
 FIN
```

13 - Algorithmes de tri : Tri par insertion

Le tri par insertion est beaucoup plus lent que d'autres algorithmes comme le tri rapide (ou *quicksort*) et le tri fusion pour traiter de grandes séquences, car sa complexité asymptotique est quadratique (n2).

Remarque : Le tri par insertion est cependant considéré comme le tri le plus efficace sur des entrées de petite taille. Il est aussi très rapide lorsque les données sont déjà presque triées.

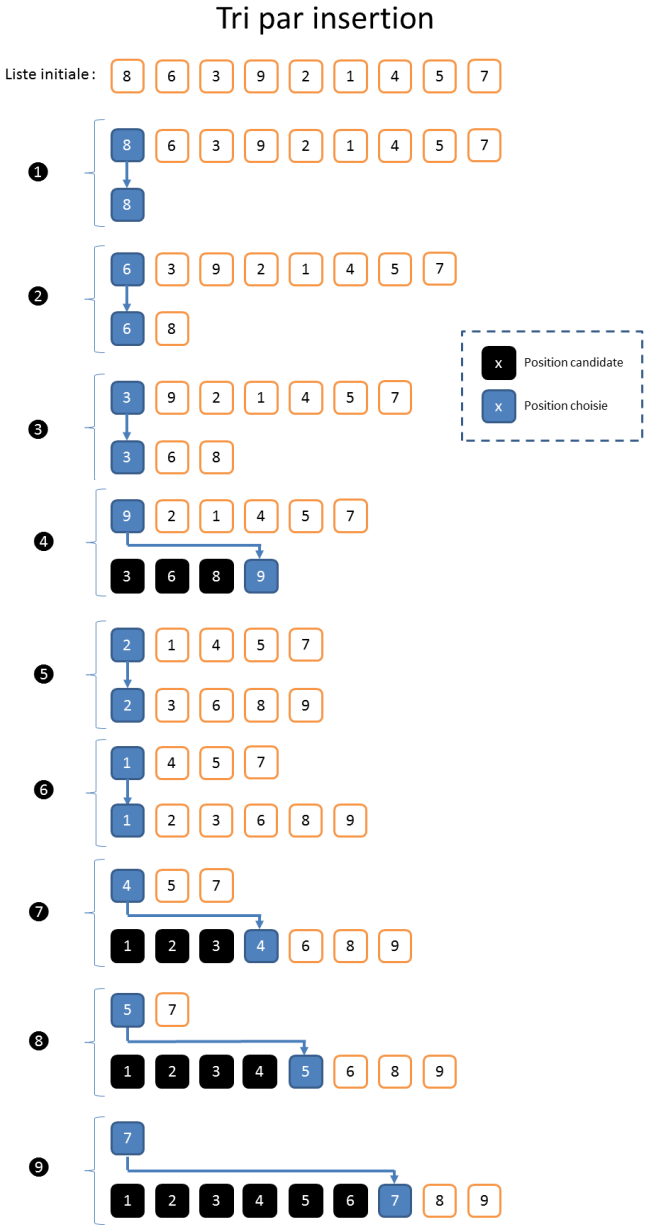
Pour ces raisons, il est utilisé en pratique en combinaison avec d'autres méthodes comme le tri rapide.

Le Tri par insertion est le tri du joueur de cartes. On fait comme si les éléments à trier étaient donnés un par un, le premier élément constituant, à lui tout seul, une liste triée de longueur 1.

On range ensuite le second élément pour constituer une liste triée de longueur 2, puis on range le troisième élément pour avoir une liste triée de longueur 3 et ainsi de suite...

Le principe du tri par insertion est donc d'insérer à la $n_{ième}$ itération le $n_{ième}$ élément à la bonne place.

13 - Algorithmes de tri : Tri par insertion



https://www.podcastscience.fm/dossiers/2014/09/04/les-tris/

13 - Algorithmes de tri : Tri par insertion

```
PROCEDURE tri_insertion(Vecteur ENTIER T, ENTIER n)
  ENTIER x,i,j
DEBUT
      POUR i DE 1 A n - 1
          // mémoriser T[i] dans x
          x <- T[i]
          // décaler vers la droite les éléments de T[0]..T[i-1] qui sont plus grands que x
en partant de T[i-1]
          j <- i
          TANT QUE j > 0 ET T[j - 1] > x FAIRE
                T[j] <- T[j - 1]
                j <- j - 1
          FIN TANT QUE
          // placer x dans le "trou" laissé par le décalage
          T[j] <- x
      FIN POUR
FIN
```

https://fr.wikipedia.org/wiki/Tri_par_insertion

13 - Algorithmes de tri : Tri à bulles

Le tri à bulles ou tri par propagation1 est un algorithme de tri.

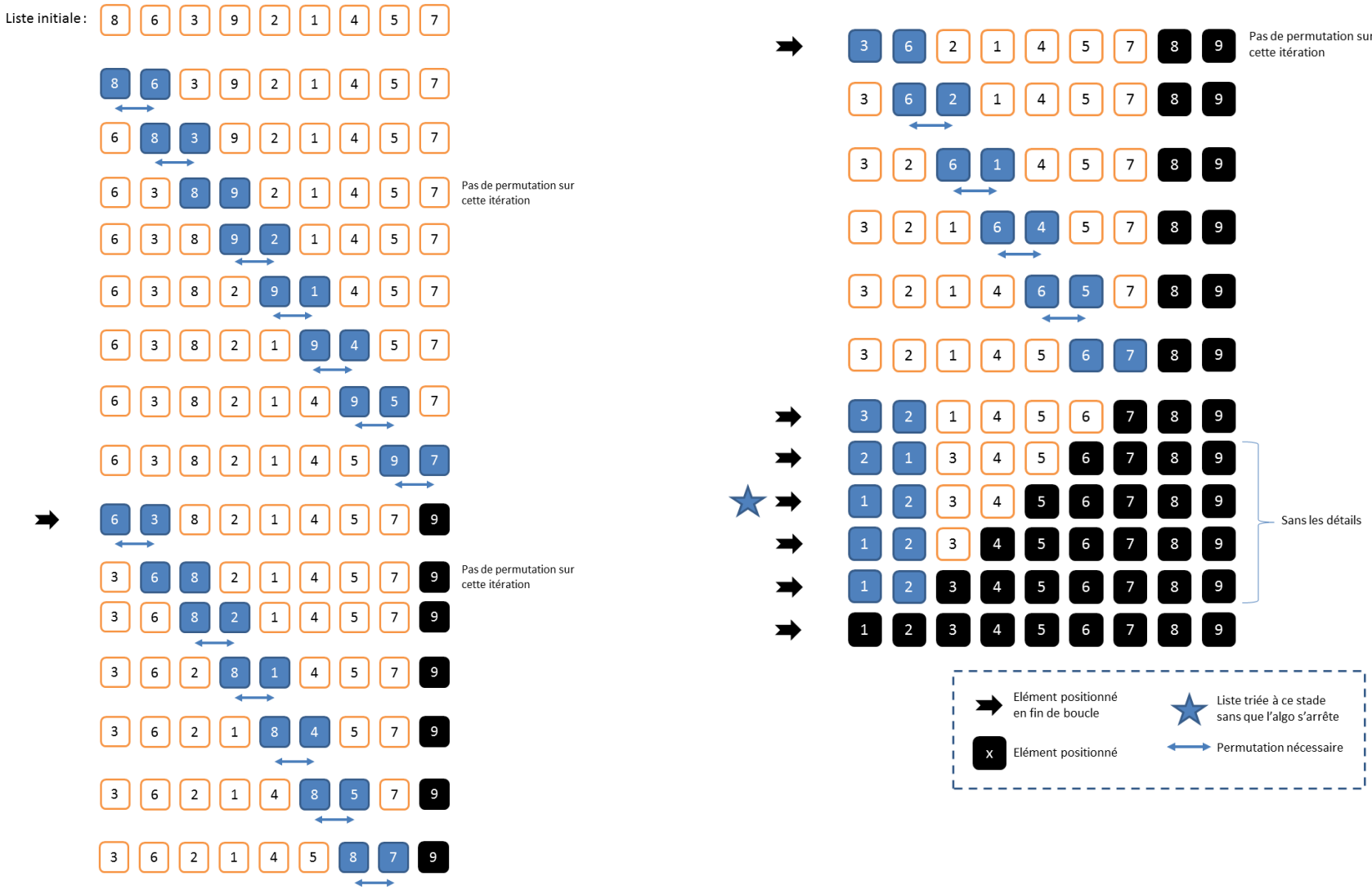
Il consiste à comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils sont mal triés. Il doit son nom au fait qu'il déplace rapidement les plus grands éléments en fin de tableau, comme des bulles d'air qui remonteraient rapidement à la surface d'un liquide.

Le tri à bulles est souvent enseigné en tant qu'exemple algorithmique, car son principe est simple.

Mais c'est le plus lent des algorithmes de tri communément enseignés, et il n'est donc guère utilisé en pratique.

13 - Algorithmes de tri : Tri à bulles

Tri à bulle



13 - Algorithmes de tri : Tri à bulles

```
PROCEDURE tri_à_bulles(Vecteur ENTIER T)
   ENTIER i,j
   BOOLEAN  tableau_trié
DEBUT

     POUR i DE ( longeur (T)-1 ) A 1
          tableau_trié <- VRAI
          POUR j DE 0 A i-1 FAIRE
               SI T[j+1] < T[j] ALORS
                    echanger(T[j+1], T[j])
                    tableau_trié <- FAUX
               FIN SI
          FIN POUR
          SI tableau_trié ALORS FIN
     FIN POUR

FIN
```

13 - Algorithmes de tri : Le tri rapide

Le tri rapide - aussi appelé "tri de Hoare" (du nom de son inventeur Tony Hoare) ou "tri par segmentation" ou "tri des bijoutiers" ou, en anglais "quicksort" - est certainement l'algorithme de tri interne le plus efficace.

13 - Algorithmes de tri : Le tri rapide

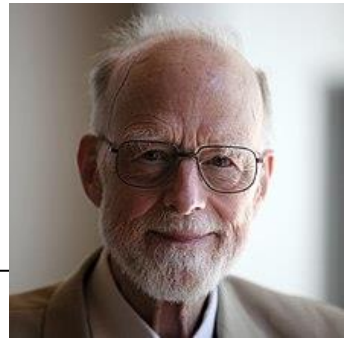
Charles Antony Richard Hoare, né le 11 janvier 1934 à Colombo au Ceylan (maintenant Sri Lanka)

Il est connu pour avoir inventé en 1959/1960 l'algorithme de tri rapide encore très utilisé de nos jours quicksort.

Il est le premier à avoir écrit un compilateur complet pour le langage Algol 60, y compris l'appel de procédures récursives,

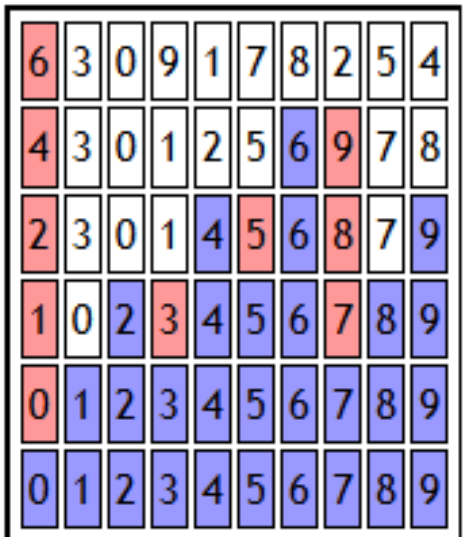
Il est à l'origine de la logique de Hoare qui sert à la vérification de la correction de programmes et du langage formel Communicating sequential processes (CSP) qui permet de spécifier l'interaction de processus concurrents et qui a inspiré les langages de programmation Occam ou Ada ainsi que le concept de moniteur

https://fr.wikipedia.org/wiki/Charles_Antony_Richard_Hoare



13 - Algorithmes de tri : Le tri rapide

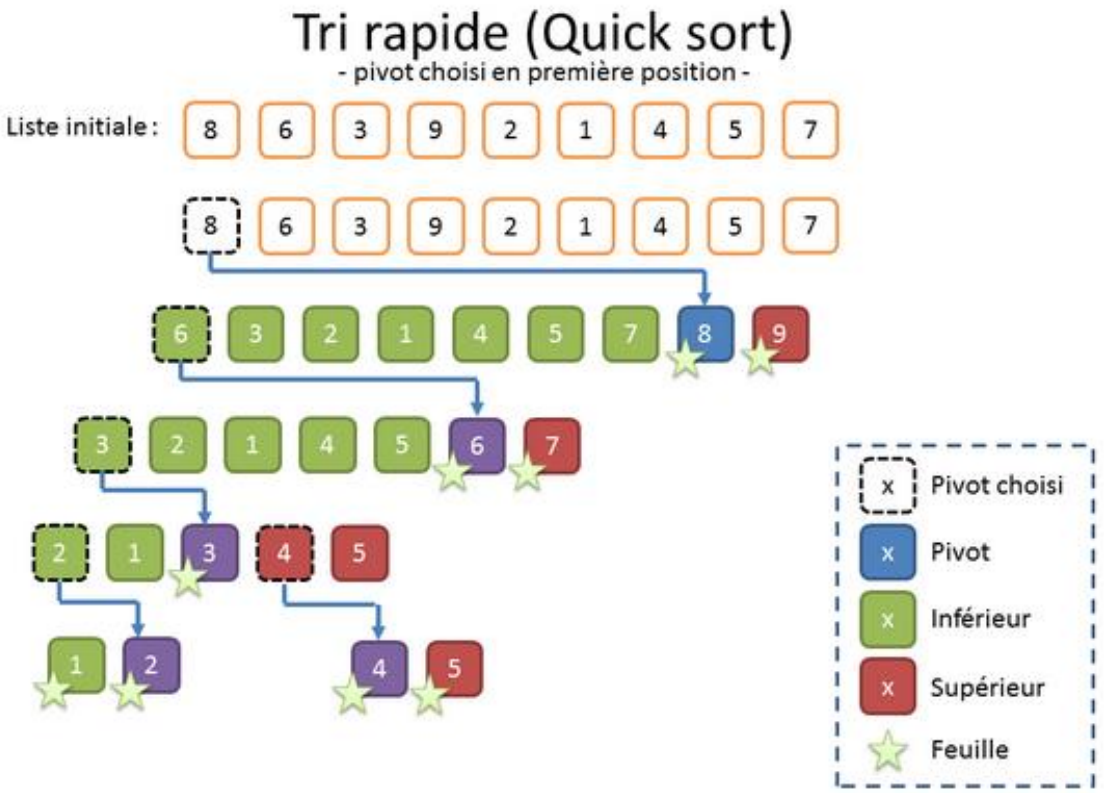
La méthode consiste à placer un élément du tableau *(appelé pivot)* à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont **inférieurs au pivot soient à sa gauche** et que tous ceux qui sont **supérieurs au pivot soient à sa droite**.
(Cette opération s'appelle le partitionnement)



Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement.

Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

13 - Algorithmes de tri : Le tri rapide



https://www.podcastscience.fm/dossiers/2014/09/04/les-tris/

13 - Algorithmes de tri : Le tri rapide

```
PROCEDURE echanger(Vecteur ENTIER T,ENTIER a, ENTIER b)
      ENTIER tmp
   DEBUT
      tmp <- a
      a   <- b
      b   <- tmp
   FIN

FONCTION partition(Vecteur ENTIER T, ENTIER premier, ENTIER dernier) RENVOIE ENTIER
      ENTIER compteur, pivot, i
   DEBUT
      compteur<-premier
      pivot<-T(premier)

      POUR i DE premier+1 A dernier FAIRE
         SI T(i)<pivot ALORS //si l'élément est inférieur au pivot
            compteur<-compteur+1 //on incrémente le compteur (modification de la place finale du pivot)
            echanger(T, compteur, i) //on place l'élément à la position finale du pivot
         FIN SI
      FIN POUR

      echanger(T, compteur, premier) //on place le pivot à sa place
      RENVOIE (compteur) //on renvoie la position finale du pivot

   FIN
```

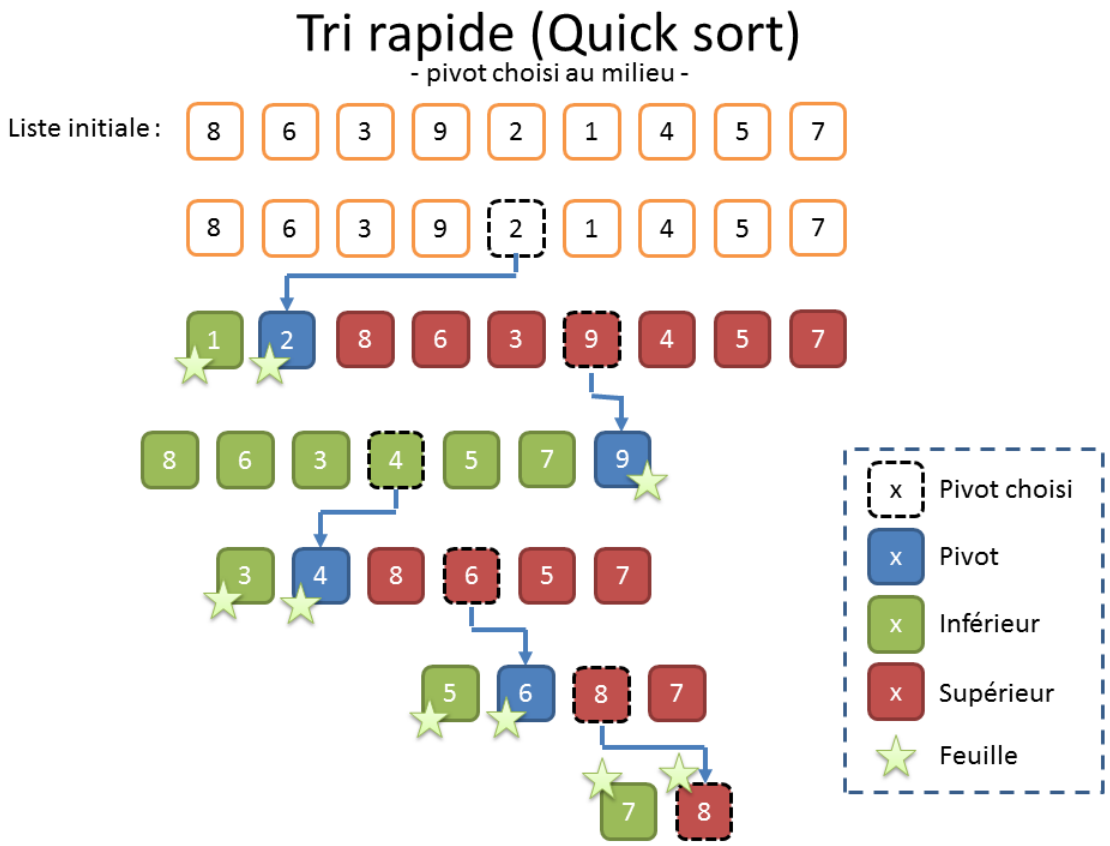
http://www.dailly.info/Tri-rapide

13 - Algorithmes de tri : Le tri rapide

```
PROCEDURE tri_rapide_bis(Vecteur ENTIER T, ENTIER premier, ENTIER dernier)

    ENTIER pivot
  DEBUT
    SI premier<dernier ALORS //condition d'arret de la récursivité
      pivot <- partition(T,premier,dernier) //partition du tableau en 2
      tri_rapide_bis(T,premier,pivot-1) //tri de la première partition
      tri_rapide_bis(T,pivot+1,dernier) //tri de la seconde partition
    FIN SI
  FIN

PROCEDURE tri_rapide(Vecteur ENTIER T)
  DEBUT
    tri_rapide_bis(T,0,longueur(T)-1)) //tri du tableau entier
  FIN
```

http://www.dailly.info/Tri-rapide

13 - Algorithmes de tri : Le tri rapide



https://www.podcastscience.fm/dossiers/2014/09/04/les-tris/

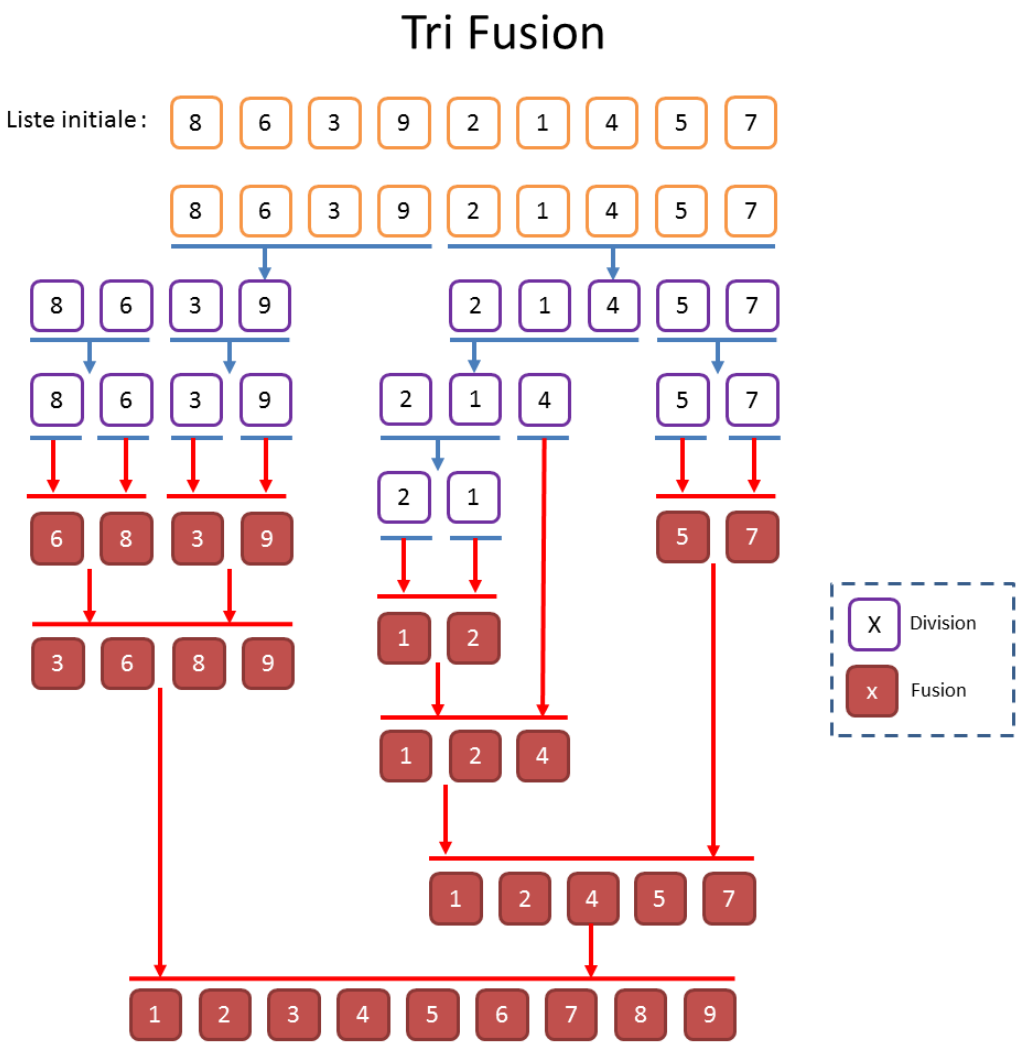
13 - Algorithmes de tri : Tri fusion

le **tri fusion** est un algorithme de tri par comparaison stable. Sa complexité temporelle pour une entrée de taille *n* est de l'ordre de *n* log *n*, ce qui est asymptotiquement optimal. Ce tri est basé sur la technique algorithmique diviser pour régner.

L'opération principale de l'algorithme est la *fusion*, qui consiste à réunir deux listes triées en une seule.

L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire.

13 - Algorithmes de tri : Tri fusion



13 - Algorithmes de tri : Tri fusion

```
PROCEDURE fusion(Vecteur ENTIER T, ENTIER debut, ENTIER milieu, ENTIER fin)*
    ENTIER i,indexG, indexD,lg,ld
    Vecteur G,D
DEBUT
    lg <- milieu - debut + 1
    ld <- fin - milieu
    etendre (G,lg,0)
    etendre (D,ld,0)
    POUR i DE 0 A lg - 1
        G[i] <- T[ i + debut]
    POUR i DE 0 A ld - 1
        D[j] <- T[milieu + i + 1]
    indexG <- 0
    indexD <- 0
    POUR i DE debut A fin
        SI G[indexG] <= D[indexD]
        ALORS
                T[i] = G[indexG]
                indexG <- indexG + 1
        SINON
                T[i] = D[indexD]
                indexD <- indexD + 1
        FIN SI
    FIN POUR
FIN
```

```
// les vecteurs (tableaux) commence a 0 : T[0..lt-1] avec lt longueur du tableau

PROCEDURE triFusion(Vecteur ENTIER T, ENTIER debut, ENTIER fin)
    ENTIER milieu
DEBUT
    SI debut < fin  ALORS
        milieu <- (debut+fin) / 2
        triFusion(T,debut,milieu)
        triFusion(T,milieu+1,fin)
        fusion(T,debut, milieu, fin )
    FIN SI
FIN
```

14 - Bibliographie

http://lwh.free.fr/pages/algo/algorithmes.htm