



# Programmation algorithmique

## Leçon 1 Introduction à l'algorithmie



(Wikipedia)

Muhammad Ibn Mūsā **al-Khuwārizmī**, Perse (Bagdad), 780-850.  
Mathématicien, astronome, géographe et auteur.

# Plan de leçon

## **Qu'est-ce qu'un algorithme**

Le pseudo-code

Exemples

Exactitude et efficacité (survol)

Les opérations de base

Les limites du pseudo-code

# Qu'est-ce qu'un algorithme?

**Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème ou d'obtenir un résultat**

Wikipedia

**Une suite finie d'instructions basiques**

**Opérant sur une entrée**

**Produisant une sortie**

**Dans un temps fini**

Knuth

# Qu'est-ce qu'un algorithme?

**Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème ou d'obtenir un résultat**

Wikipedia

**Une suite finie d'instructions basiques**

**Opérant sur une entrée**

**Produisant une sortie**

Knuth

**Dans un temps fini**

Une recette de biscuits

Une procédure de triage

Une méthode de calcul

# Exemple - L'ajustement du prix du loyer.

<https://www.rdl.gouv.qc.ca/fr/calcul-pour-la-fixation-de-loyer/outil-de-calcul>

Ajustement de loyer du logement ⓘ

Loyer mensuel du logement  
(avant augmentation)

Reporter le pourcentage F

Multiplié par F

Montant de la case G

Total

Après arrondissement

## Exemple - La valeur absolue

Entrée : Un nombre réel  $x$

Sortie : Un nombre réel  $x$

Les instructions:

```
Si  $x < 0$ ,  
Alors  $x \leftarrow x * (-1)$   
Retourner  $x$ 
```

## Exemple - Le pgcd (encore?!)

Entrée : Deux nombres naturels  $x$  et  $y$

Sortie : Un nombre naturel **pgcd**

Les instructions:

$i \leftarrow 1$

$\text{pgcd} \leftarrow 1$

Tant que  $i \leq x$

Faire

    Si  $i$  divise  $x$  et  $y$  sans reste

        Alors  $\text{pgcd} \leftarrow i$

$i \leftarrow i + 1$

Retourner  $\text{pgcd}$

## Exemple - La racine carrée (méthode babylonienne)

Entrée : Un nombre réel  $x$

Sortie : Un nombre réel  $r$

Les instructions :

$r \leftarrow 10$

Tant que  $r*r$  est loin de  $x$ ,

Faire  $r \leftarrow$  la moyenne entre  $r$  et  $x/r$

Retourner  $r$



## Exemple - La racine carrée (méthode babylonienne)

Entrée : Un nombre réel  $x$

Sortie : Un nombre réel  $r$

Les étapes (méthode babylonienne):

$r \leftarrow 10$

Tant que la valeur absolue de  $r*r - x$  est plus grande que  $0.0001$ ,

Faire  $r \leftarrow 0.5 * (r + (x/r))$

Retourner  $r$

# Exercice

## En utilisant

L'affectation ( $\leftarrow$ ),

Si... alors

Tant que... faire

Les opérateurs arithmétiques de base (+, -, \*, /, et %)

**Développez un algorithme en pseudo-code qui traduit un nombre écrit en nombre romain en nombre écrit en décimal.**

## Exercice

**Le Si... alors n'est pas une opération essentielle.**

**En utilisant l'affectation et le Tant que... Faire, programmez le Si... Alors.**

# Plan de leçon

Qu'est-ce qu'un algorithme  
Le pseudo-code  
Exemples

## **Exactitude et efficacité (survol)**

Les opérations de base

Les limites du pseudo-code

# Un bon algorithme

**Un bon algorithme est exact, efficace et élégant.**

## Exactitude et efficacité

Un algorithme est **exact** s'il donne la bonne sortie pour toute les entrées possibles. (Il ne se trompe jamais!)

Un algorithme est **efficace** s'il s'exécute en peu d'opérations basiques.

Un algorithme est **élégant** si ses instructions sont lisibles et peu nombreuses.

## Exactitude et efficacité

Un algorithme est **exact** s'il donne la bonne sortie pour toute les entrées possibles. (Il ne se trompe jamais!)

Un algorithme est **efficace** s'il s'exécute en peu d'opérations basiques.

Comment calcule-ton l'efficacité?

Un algorithme est **élégant** si ses instructions sont lisibles et peu nombreuses.

## Efficacité - Revenons au pgcd

```
i ← 1
pgcd ← 1
Tant que i ≤ x
Faire
    Si i divise x et y
    sans reste
        Alors pgcd ← i
    i ← i + 1
Retourner pgcd
```

**Combien d'opérations  
pour**

**x = 10, y = 8?**

**Combien d'opérations  
pour**

**x = 3, y = 550?**

**y = 550, x = 3?**

Le nombre d'opérations effectuées  
dépend de l'entrée.



## Efficacité - Revenons au pgcd

```
i ← 1
pgcd ← 1
Tant que i ≤ x
Faire
    Si i divise x et y
    sans reste
        Alors pgcd ← i
    i ← i + 1
Retourner pgcd
```

**Combien d'opérations  
pour**

**x = 10, y = 8?**

**Combien d'opérations  
pour**

**x = 3, y = 550?**

**y = 550, x = 3?**

Pouvez-vous faire mieux?

$O(2^n)$

## Efficacité - Revenons au pgcd

$i \leftarrow x$

Si  $(x > y)$ , alors  $i \leftarrow y$

Tant que  $i > 1$ , faire

    Si  $i$  divise  $x$  et  $y$  sans reste

        Alors retourner  $i$

$i \leftarrow i - 1$

Retourner 1

Quel est le pire cas?  
Le meilleur cas?

$O(2^n)$

## Efficacité - Euclide

Tant que  $y$  est  
différent de 0,

Faire

$t \leftarrow y$

$y \leftarrow x \bmod y$

$x \leftarrow t$

Retourner  $y$

**Parcourir tous les  
nombres entre 1 et  $x$   
est très coûteux.**

**Heureusement, Euclide  
(Alexandrie,  $\approx 300$  av. J.-  
C.) a trouvé mieux!**

$O(n^2)$

# Plan de leçon

Qu'est-ce qu'un algorithme  
Le pseudo-code  
Exemples

Exactitude et efficacité (survol)

## **Les opérations de base**

Les limites du pseudo-code

# Qu'est-ce qu'une opération basique?

**C'est une opération qui prend un temps constant.**

# Qu'est-ce qu'une opération basique?

**C'est une opération qui prend un temps constant.**

Est-ce que  
« multiplier 698483 par 2929 »  
  
prend le même temps que  
« multiplier 5 par 6 »?

Est-ce que  
« changer le signe de 68883838 »  
  
prend le même temps que  
« changer le signe de -7 »?

# Qu'est-ce qu'une opération basique?

**C'est une opération qui prend un temps constant.**

Cela dépend du support de calcul

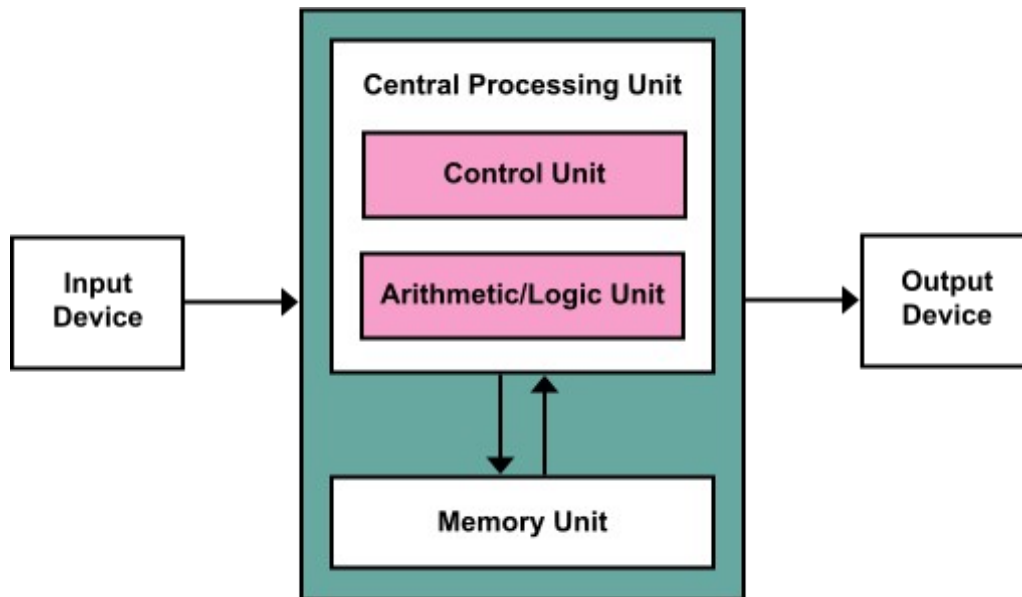


- Multiplier par 10.
- Diviser par 10.
- Comparer deux chiffres.
- Additionner deux chiffres.
- Lire un caractère.
- Changer le signe d'un nombre
- Etc...

# Qu'est-ce qu'une opération basique?

**C'est une opération qui prend un temps constant.**

Cela dépend du support de calcul



- Multiplier par 2.
- Diviser par 2.
- Déplacer la tête de lecture.
- Changer le signe d'un registre.
- Additionner deux registres.
- Comparer deux registres.
- Etc...



# Décomposition en opérations de base - La multiplication

## La multiplication, en général, n'est pas une opération basique...

Entrée : Deux nombres naturels  $x$  et  $y$

Sortie : Le produit des deux naturels.

$i \leftarrow 1$

$t \leftarrow x$

Tant que  $i < y$

Faire

$x \leftarrow x + t$

$i \leftarrow i + 1$

Retourner  $x$

Avec un papier et un crayon,  
comment multipliez-vous?

# Décomposition en opérations de base - La multiplication « à la russe »

Entrée : Deux nombres naturels  $x$  et  $y$

Sortie : Le produit des deux naturels.

Les instructions:

$r = 0$

Tant que  $x$  est différent de 0, Faire

    Si  $x$  est impair, Alors

$r = r + y$

$x = x - 1$

$x = x / 2$

$y = y * 2$

Retourner  $r$

En C, ça ressemble à quoi?

# Plan de leçon

Qu'est-ce qu'un algorithme  
Le pseudo-code  
Exemples

Exactitude et efficacité (survol)

Les opérations de base

**Les limites du pseudo-code**

# Le pseudo-code a ses limites...

## **Il est loin des processeurs et des registres :**

Pas de débordement,

Pas d'erreur d'arrondi,

Pas de types,

La mémoire est arbitrairement grande,

Les opérations basiques ne sont pas basiques pour toutes les implémentations,

Chaque variable est accessible en la même quantité de temps,

...

# Exercices

## **Traduisez les algorithmes du pseudo-code au C.**

Algorithme d'Euclide, méthode de Babylone, multiplication à la russe et traduction de chiffres romains aux décimaux.

## **Testez-les.**

**Avez-vous pensé à tous les cas limites?  
(nombres positifs, négatifs, nuls, etc.)**





# Programmation algorithmique

## Leçon 2 La pile et le tas



**Stack**

Google



**Heap**

@fhinkel

## Quelques questions...

Comment un exécutable fait-il pour savoir l'ordre d'exécution des instructions?

Comment un exécutable fait-il pour savoir quoi exécuter après un **return**?

Comment la mémoire est-elle allouée et libérée?

# Comment les programmes sont-ils exécutés?

Les langages structurés suivent un « pointeur d'exécution »

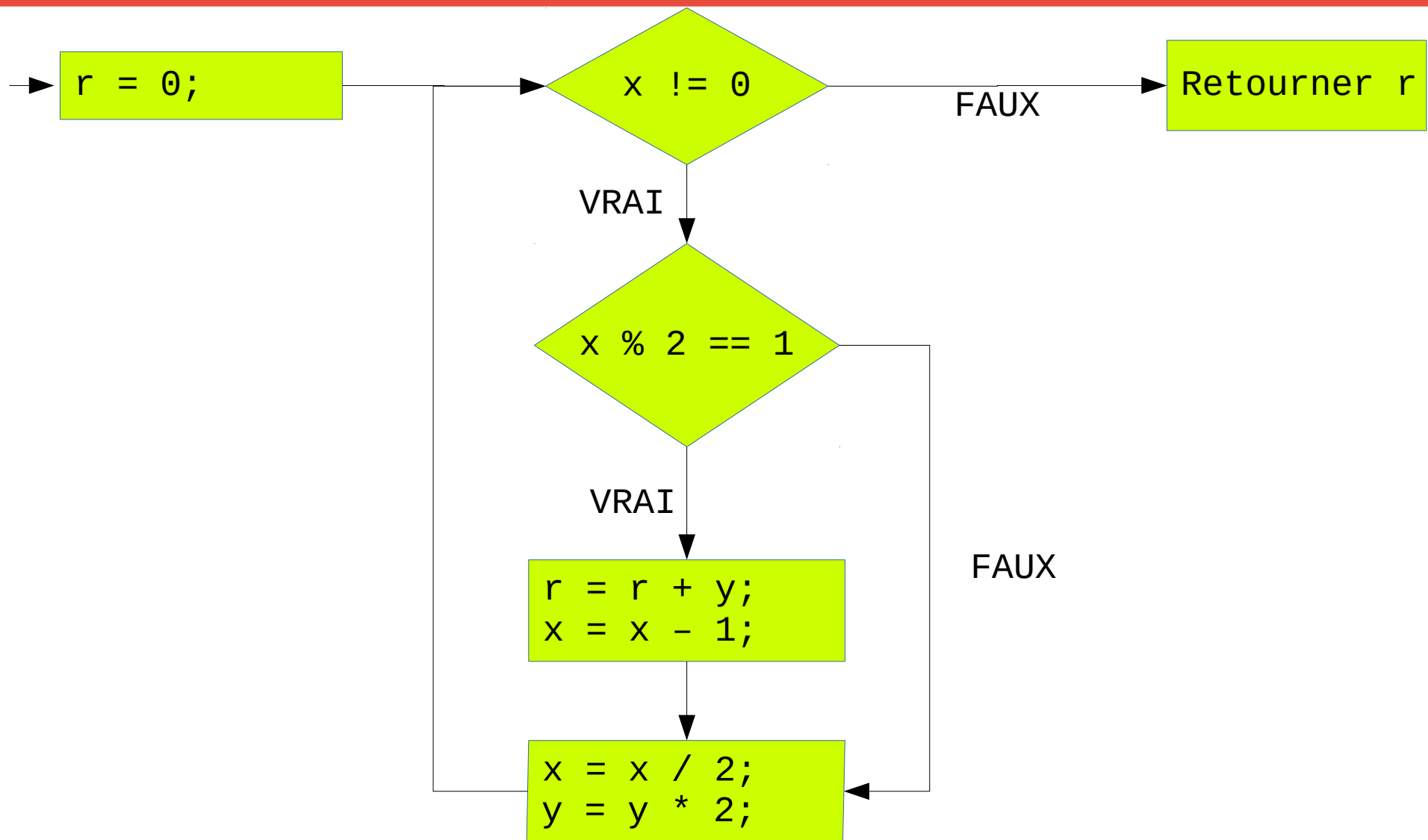
Dans le langage courant, cela signifie « À quel numéro de ligne sommes-nous dans l'exécution du programme ».

On peut visualiser l'exécution d'un programme à l'aide d'un diagramme de flux et en suivant les flèches de haut en bas. Notre doigt, c'est le pointeur d'exécution.





# Le diagramme de flux de la multiplication à la russe.



# Le pointeur d'exécution

**Sur les applications très simples, on peut facilement suivre le diagramme de flux.**

**Mais comment l'exécution s'y retrouve si...**

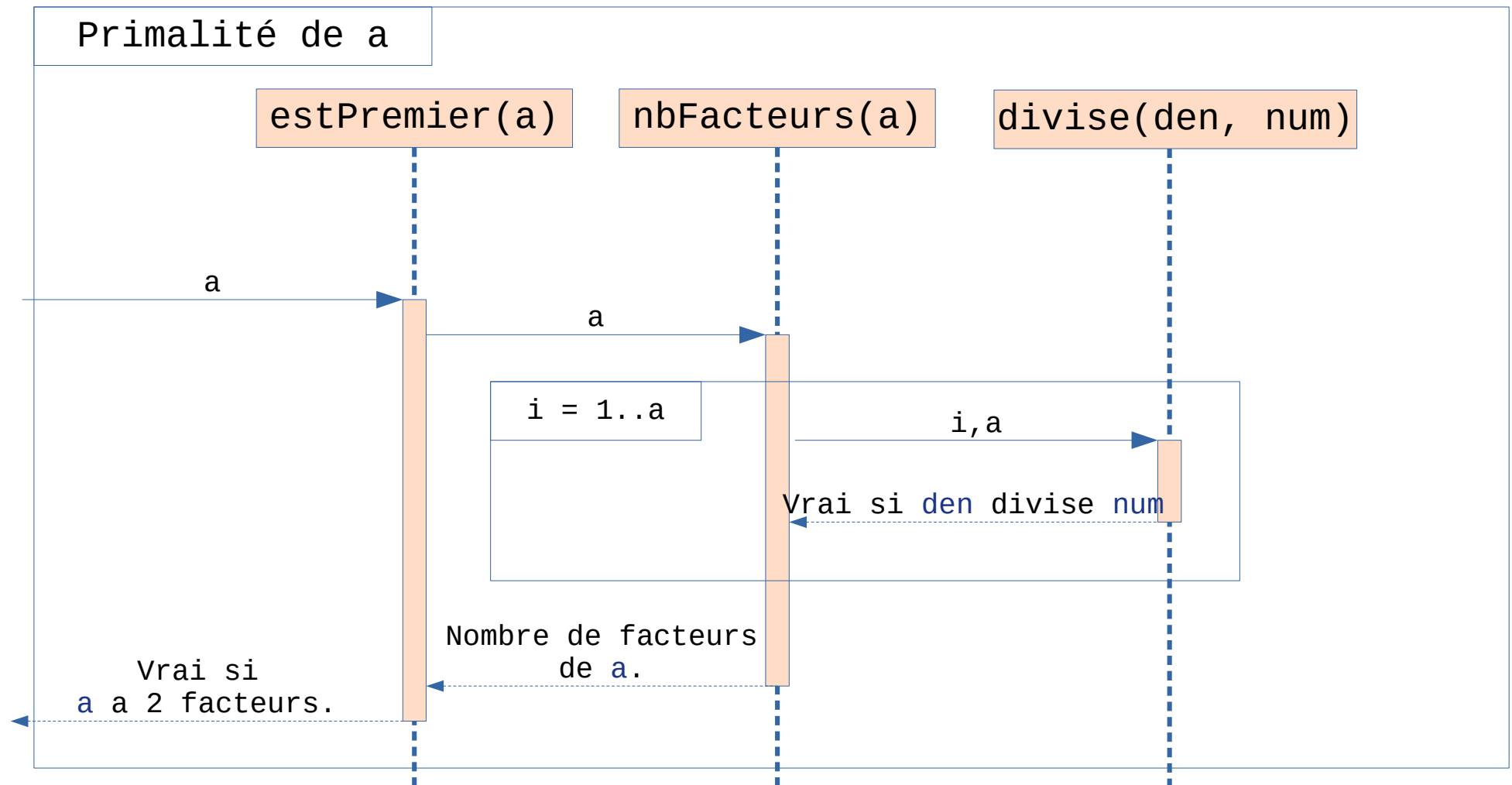
On a beaucoup d'instructions?

On a des appels de fonctions?

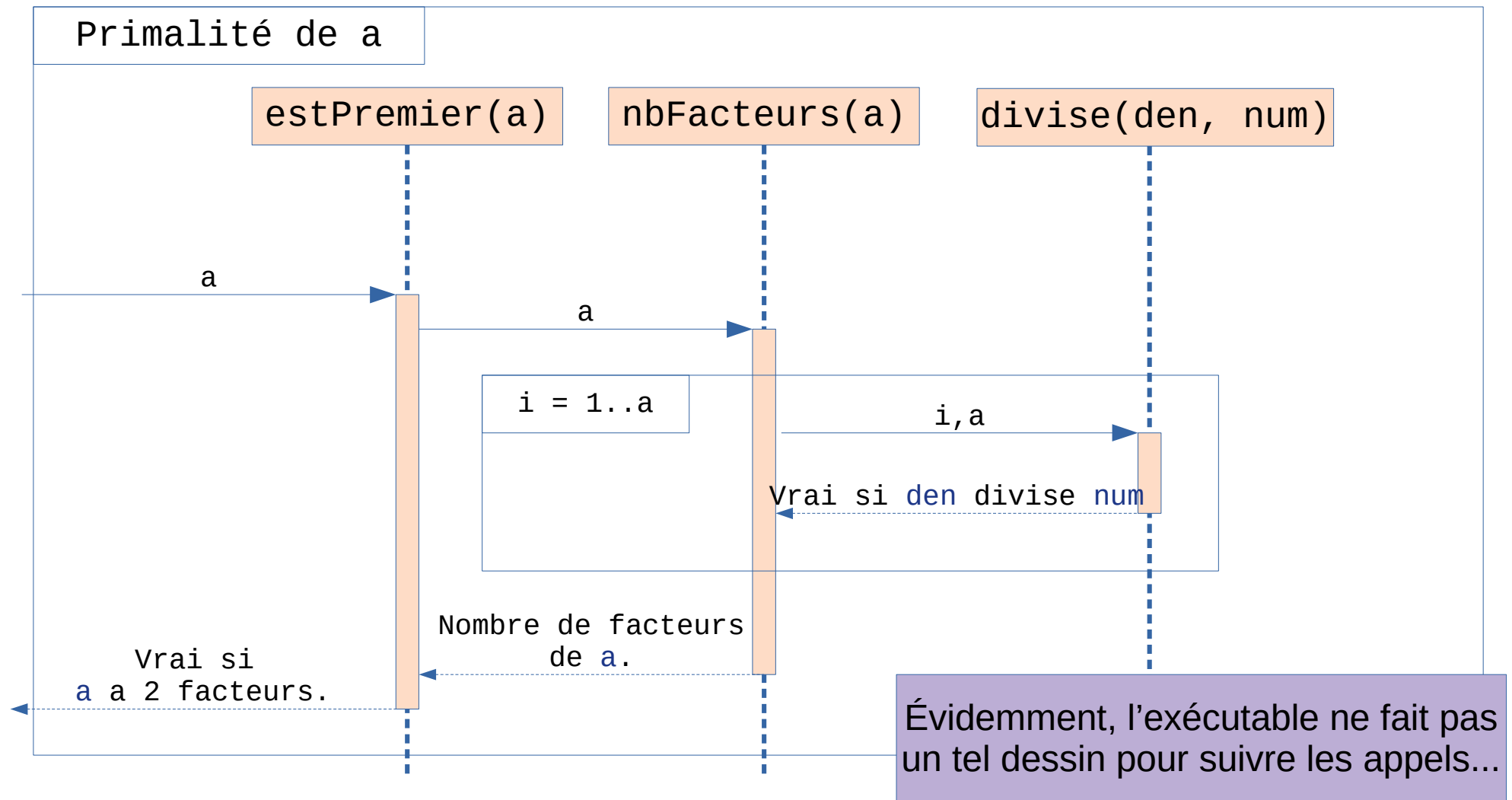
... On fait du multithread? (dans un cours futur)



# Le diagramme de séquence d'un petit programme.



# Le diagramme de séquence d'un petit programme.



# La solution : la pile d'exécution

L'exécution se fait à la fois en conservant un pointeur d'exécution, mais aussi une **pile d'exécution**.

Une pile (***stack***), c'est une structure de données très simple : Dernier arrivé premier servi.

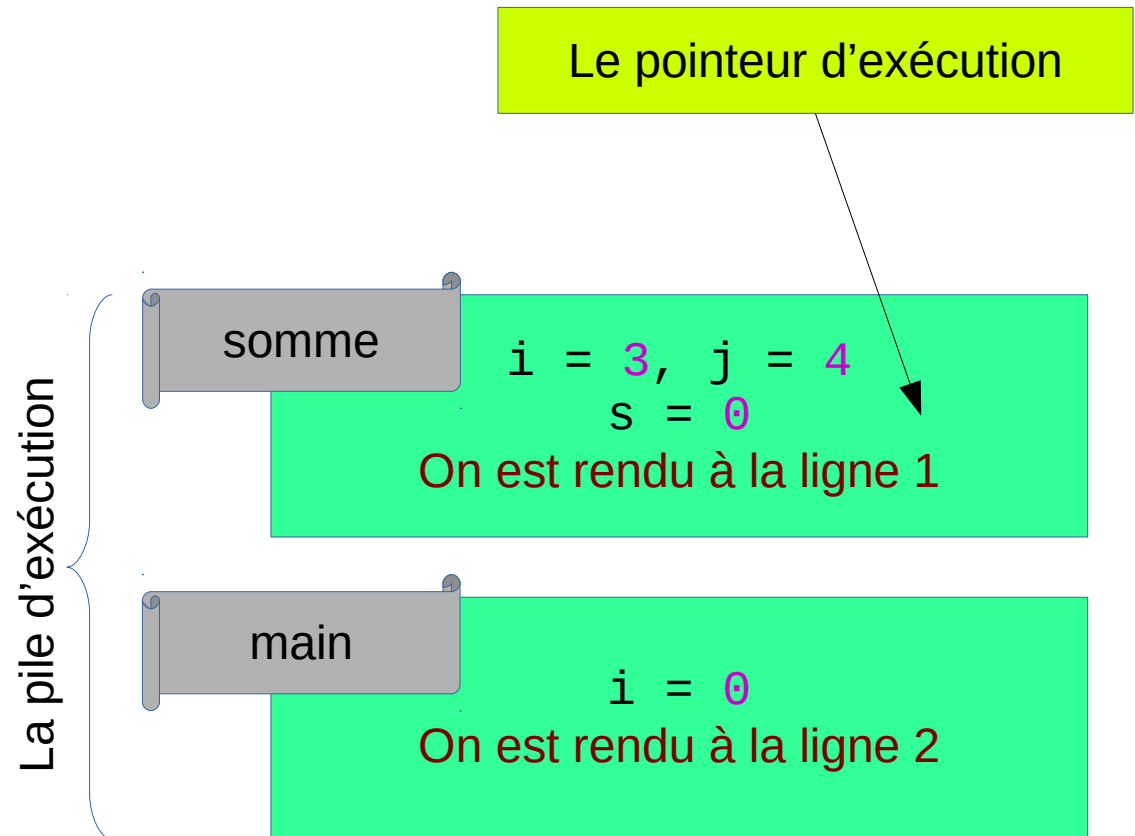


# La pile d'exécution

Voyons un exemple avec des appels de fonction...

```
int somme(int i, int j)
{
    int s = 0;
    s = i + j;
    return s;
}

int main()
{
    int i = 0;
    i = somme(3, 4);
}
```



# La pile d'exécution

```
int somme(int i, int j){  
    int s = i + j;  
    return s;  
}  
int produit (int i, int j){  
    int p = 0;  
    for (int k = 1; k <= j; k ++){  
        p = somme(p, i);  
    }  
    return p;  
}  
int main(){  
    int i = 0;  
    i = produit(4,3);  
}
```

# Exercice : dessiner la pile pour l'appel à produit(4, 3);

```
int somme(int i, int j){  
    int s = i + j;  
    return s;  
}  
int produit (int i, int j){  
    int p = 0;  
    for (int k = 1; k <= j; k ++){  
        p = somme(p, i);  
    }  
    return p;  
}  
int main(){  
    int i = 0;  
    i = produit(4, 3);  
}
```

somme

i = 0, j = 4  
s = 4

On est rendu à la ligne 2

produit

i = 4, j = 3  
p = 0, k = 1

On est rendu à la ligne 3

main

i = 0

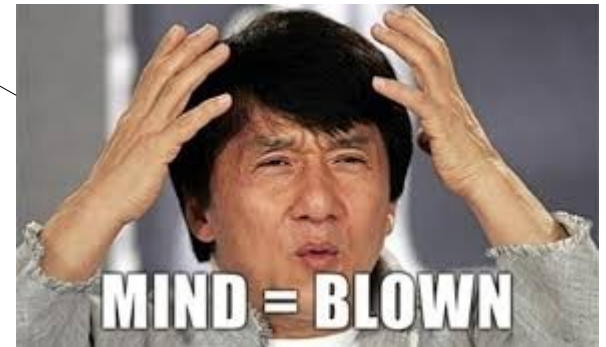
On est rendu à la ligne 2



# Une fonction peut s'appeler elle même! C'est la *récurrence*.

```
int factorielle(int i)
{
    if (i <= 1)
        return i;
    return i * factorielle(i - 1);
}
```

```
int main()
{
    int i = 0;
    i = factorielle(4);
}
```



# Une minute de silence pour apprécier la merveille qu'est la pile d'exécution

À tout instant, nous connaissons la profondeur des appels de fonctions. (`print(Thread.callStackSymbols)`)

On peut avoir n'importe quel ordre d'appel de fonctions sans que le pointeur d'exécution ne se perde.

On peut avoir des fonctions récursives.

Les variables locales ont une portée précise. Elles « meurent » après le retour et n'encombrent plus la mémoire.

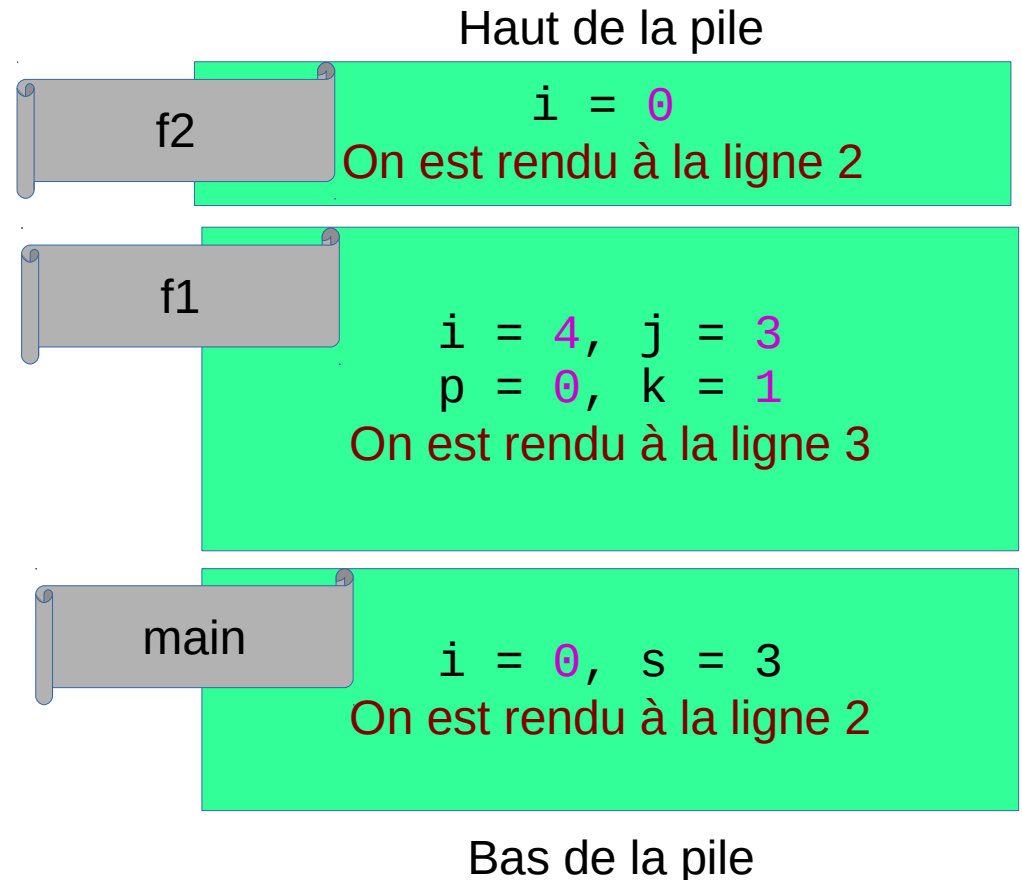


# La pile d'exécution - À retenir

À la base de la pile est le main.

Chaque appel de fonction empile une case sur la pile. Cette case contient toutes les variables locales.

Au return, la case du dessus est effacée et le pointeur d'exécution se retrouve dans la case directement en dessous.

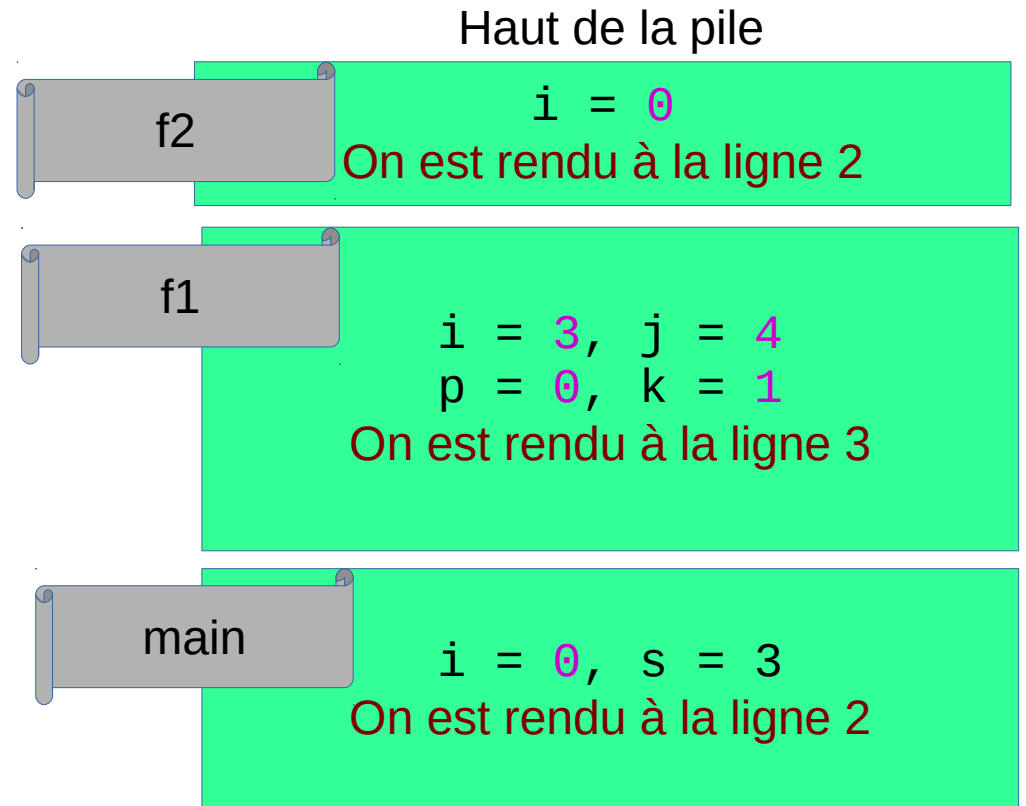


# La pile d'exécution - À retenir

À la base de la pile est le main.

Chaque appel de fonction empile une case sur la pile. Cette case contient toutes les variables locales.

Au return, la case du dessus est effacée et le pointeur d'exécution se retrouve dans la case directement en dessous.



Elle ne suffit pas pour tous les programmes...

# Revenons aux tableaux...

Essayons de retourner une variable...

```
double moyenne(int tableau[], int taille){  
    int m;  
    ...  
    return m;  
}
```

retourne la valeur  
de m.

```
int[] renverser(int source[], int taille){  
    int dest[taille];  
    ...  
    return dest;  
}
```

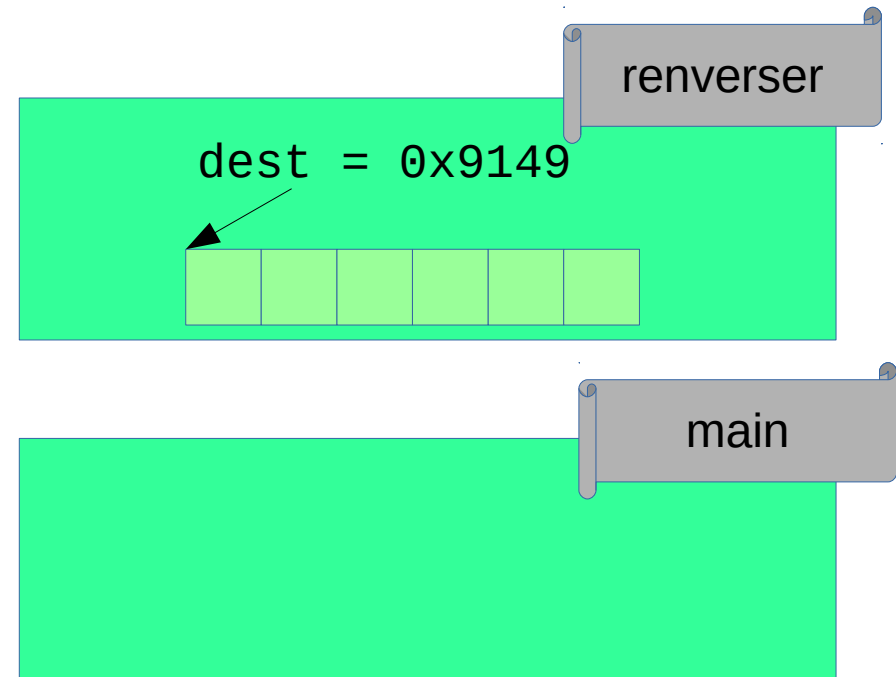
retourne la valeur  
de dest  
Mais ...

# La pile et les pointeurs

```
int* renverser(int source[], int taille){
    int dest[taille];
    for (int i = 0; i < taille; i++)
        dest[i] = source[taille - i - 1];
    return dest;
}

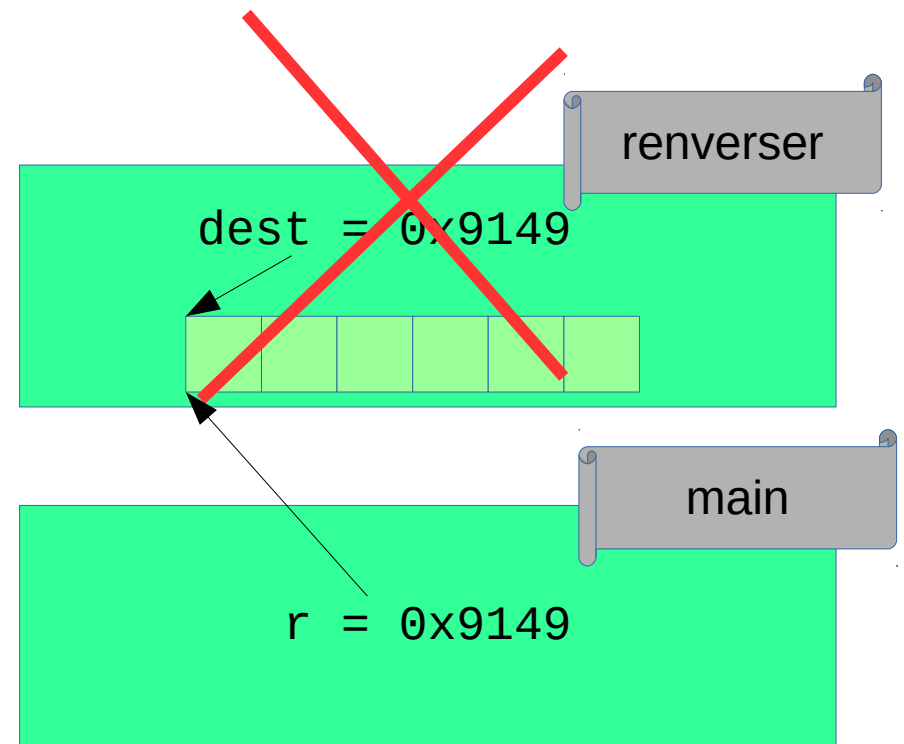
int main(){
    int t[] = {55, 2, 3, 1, 49, 9};
    int* r = renverser(t, 6);

    // Imprimer t et r
}
```



# La pile et les pointeurs

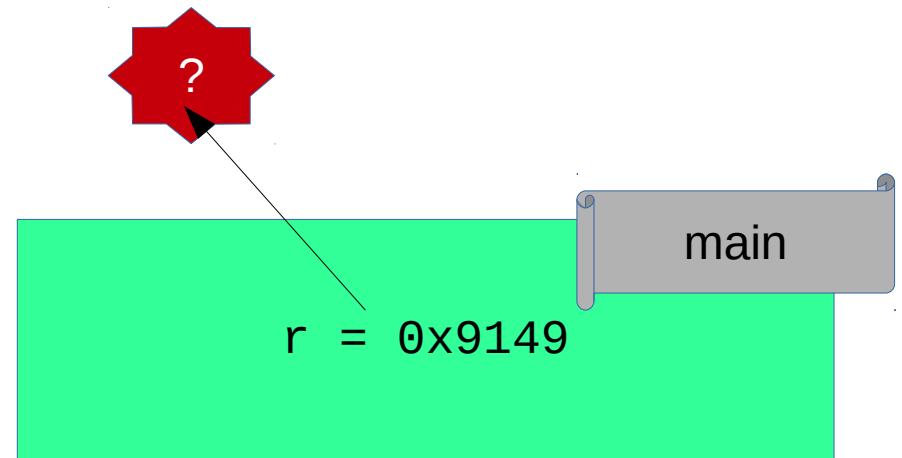
```
int* renverser(int source[], int taille){  
    int dest[taille];  
    for (int i = 0; i < taille; i++)  
        dest[i] = source[taille - i - 1];  
    return dest;  
}  
  
int main(){  
    int t[] = {55, 2, 3, 1, 49, 9};  
    int* r = renverser(t, 6);  
    // Imprimer t et r  
}
```



# La pile et les pointeurs

```
int* renverser(int source[], int taille){
    int dest[taille];
    for (int i = 0; i < taille; i++)
        dest[i] = source[taille - i - 1];
    return dest;
}

int main(){
    int t[] = {55, 2, 3, 1, 49, 9};
    int* r = renverser(t, 6);
    // Imprimer t et r
}
```

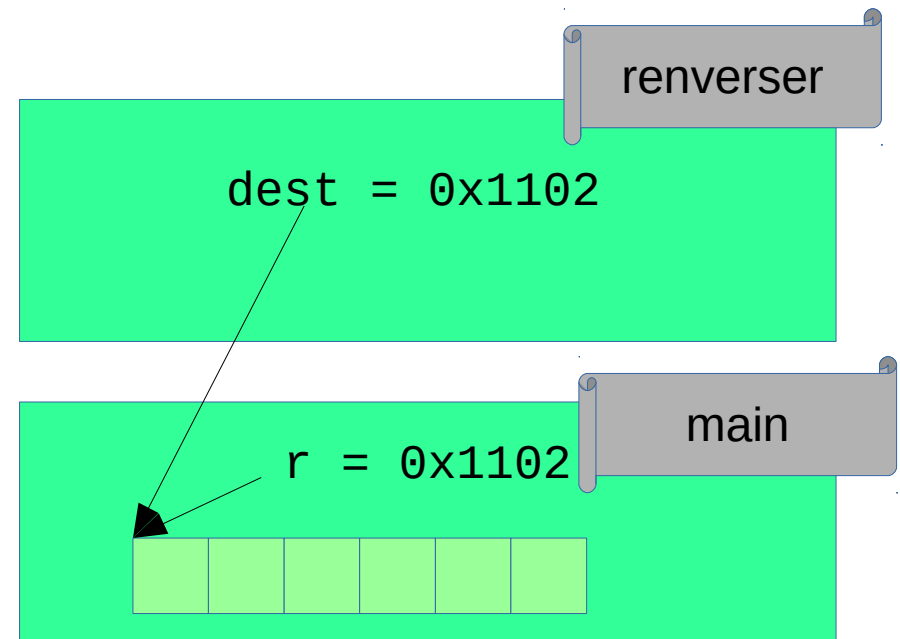




# On ne peut pas pointer vers une case plus haut dans la pile...

## Solution 1 : On passe le tableau par référence.

```
void renverser(int source[], int dest[], int taille){  
    for (int i = 0; i < taille; i++)  
        dest[i] = source[taille - i - 1];  
}  
  
int main(){  
    int t[] = {55, 2, 3, 1, 49, 9};  
    int r[6];  
    renverser(t, r, 6);  
    // Imprimer r et t  
}
```



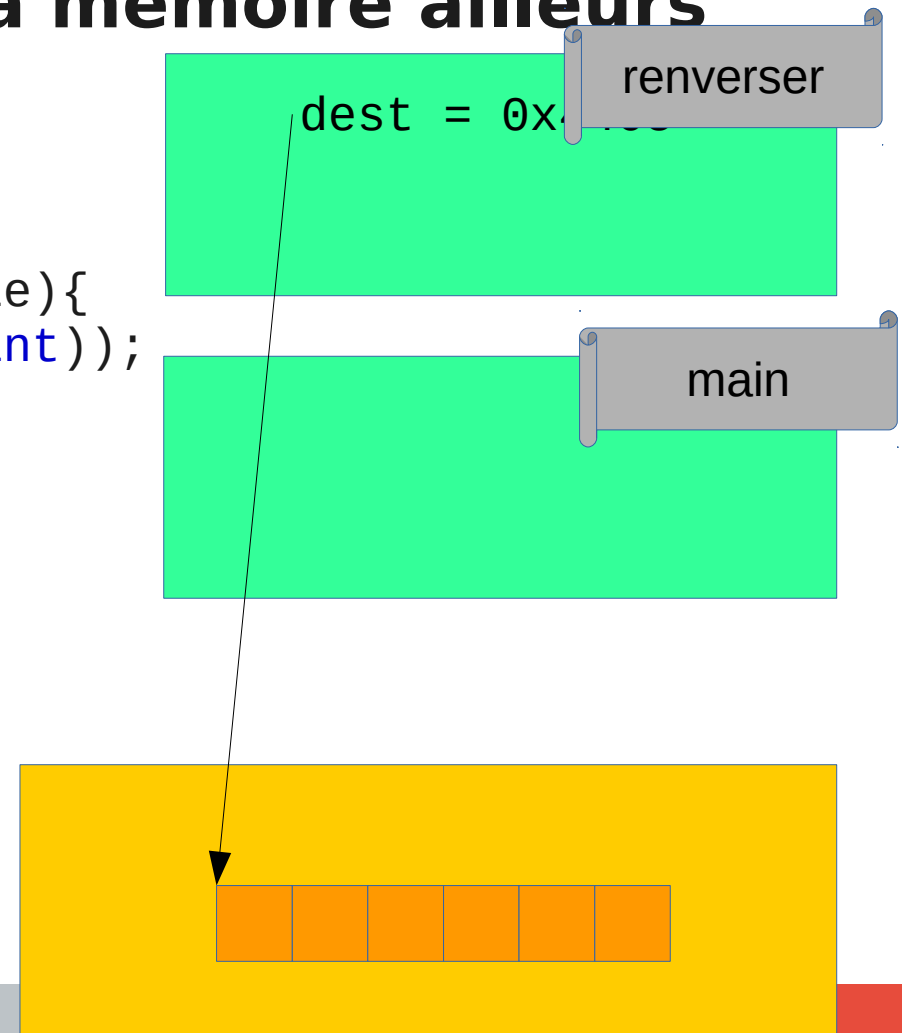
# On ne peut pas pointer vers une case plus haut dans la pile...

## Solution 2 : On réserve de la mémoire ailleurs que dans la pile. Le tas!

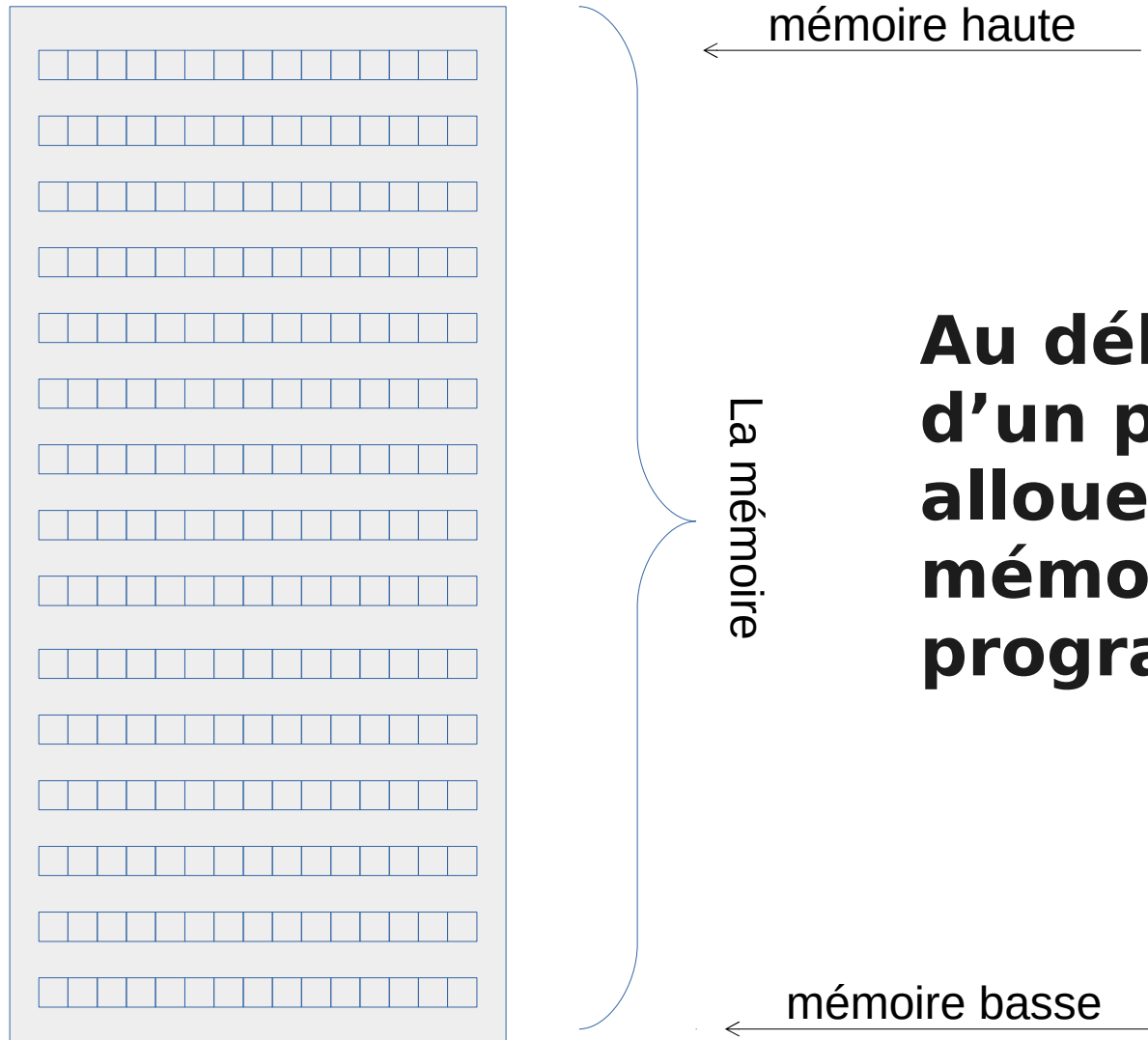
```
int* renverser(int source[], int taille){
    int* dest = malloc(taille * sizeof(int));
    for (int i = 0; i < taille; i++)
        dest[i] = source[taille - i - 1];
    return dest;
}

int main(){
    int t[] = {55, 2, 3, 1, 49, 9};
    int* r = renverser(t, 6);

    // Imprimer t et r
}
```

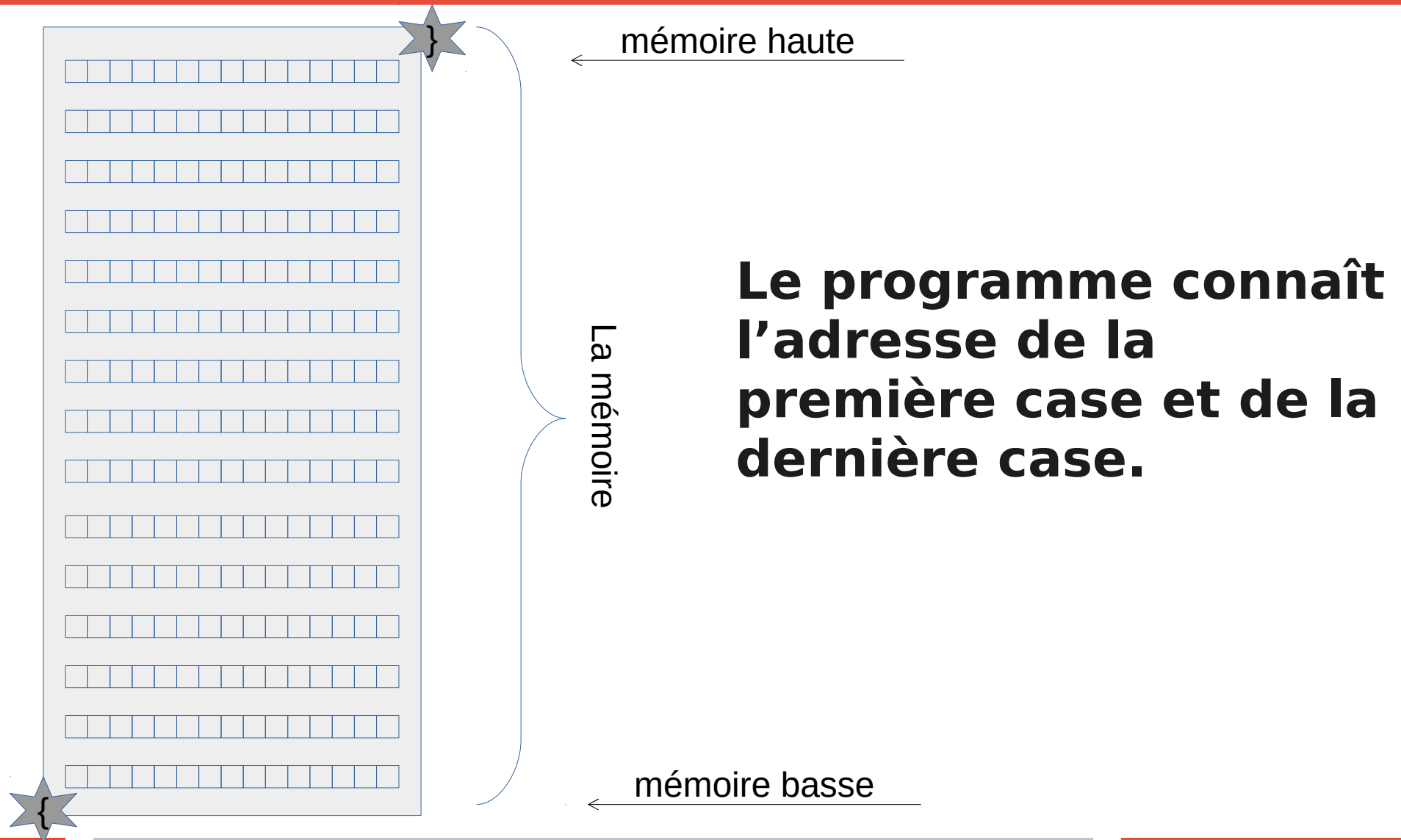


# La mémoire...

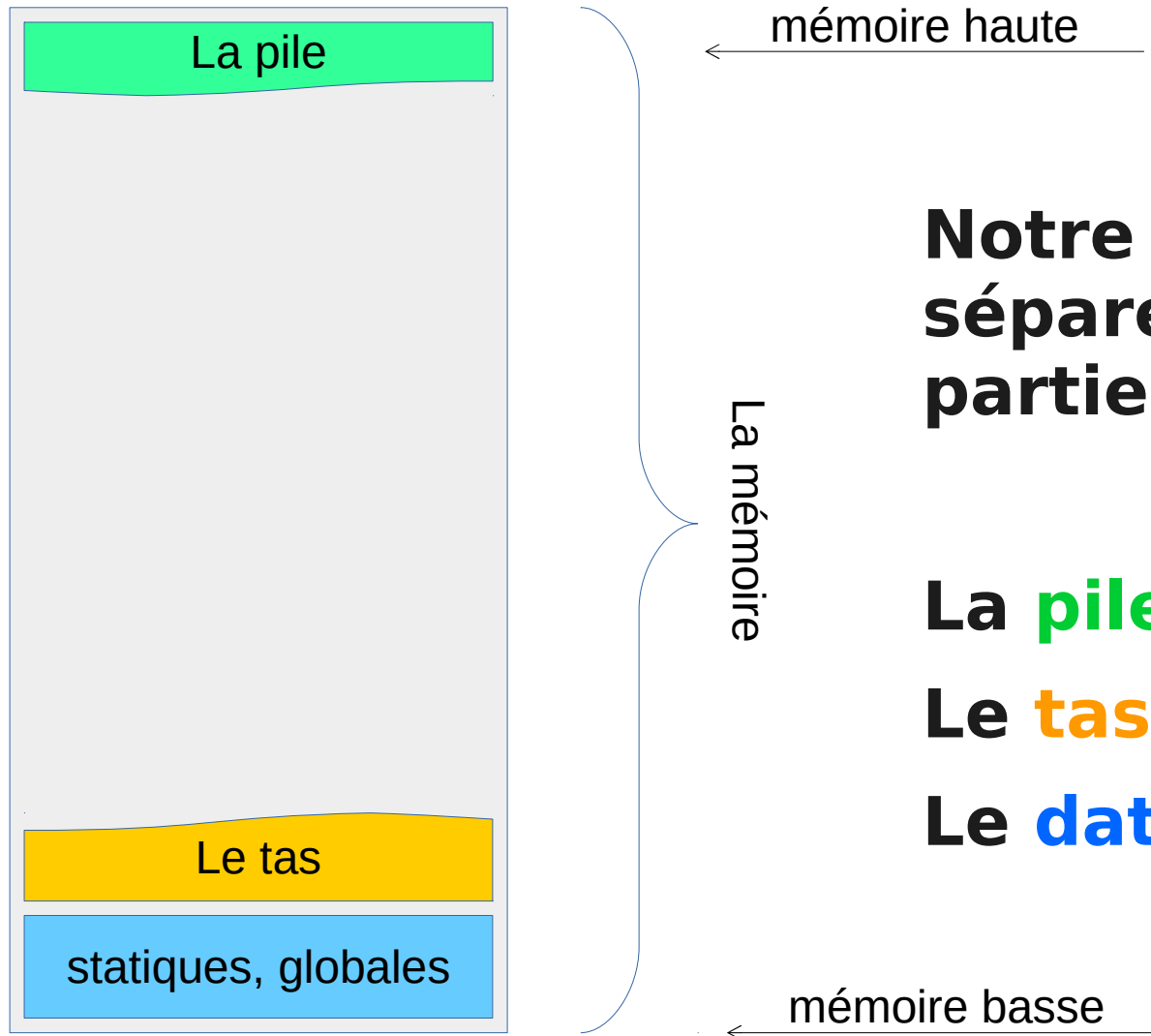


**Au début de l'exécution d'un programme, l'OS alloue un espace mémoire pour notre programme.**

# La mémoire...



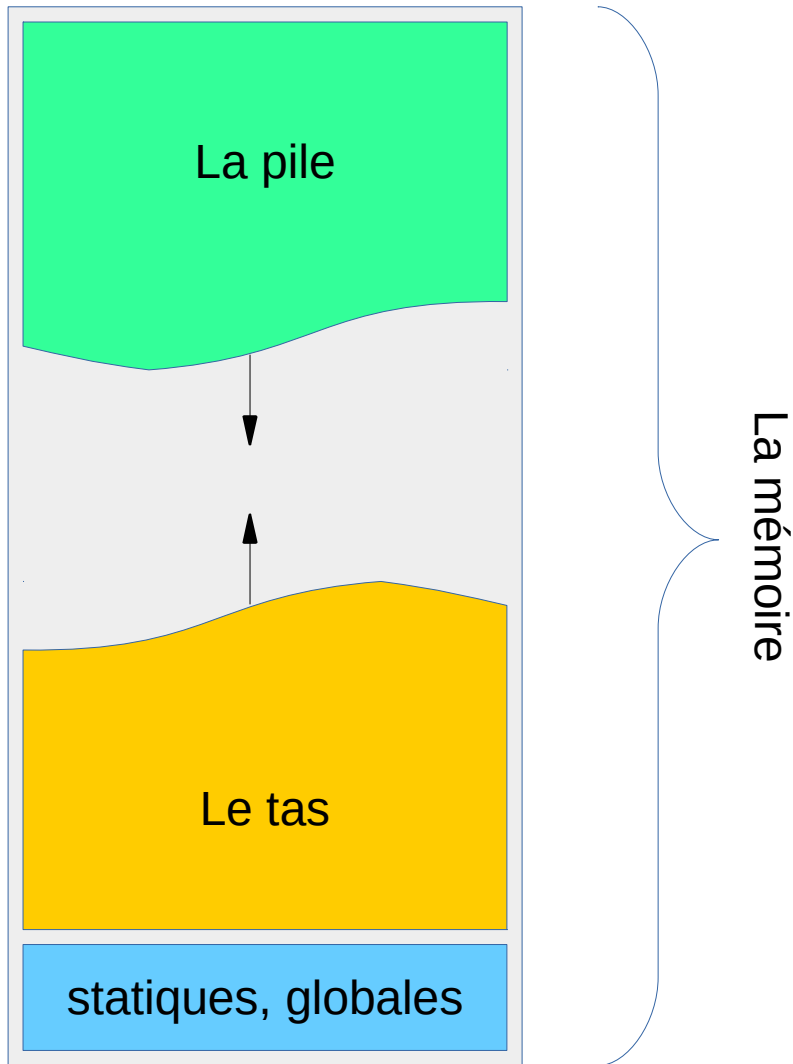
# La mémoire...



**Notre programme va la  
séparer en trois  
parties.**

**La *pile* (*stack*),  
Le *tas* (*heap*) et  
Le *data statique*.**

# La pile... le tas...



La **pile** : on la connaît, c'est les variables locales.

Taille dynamique

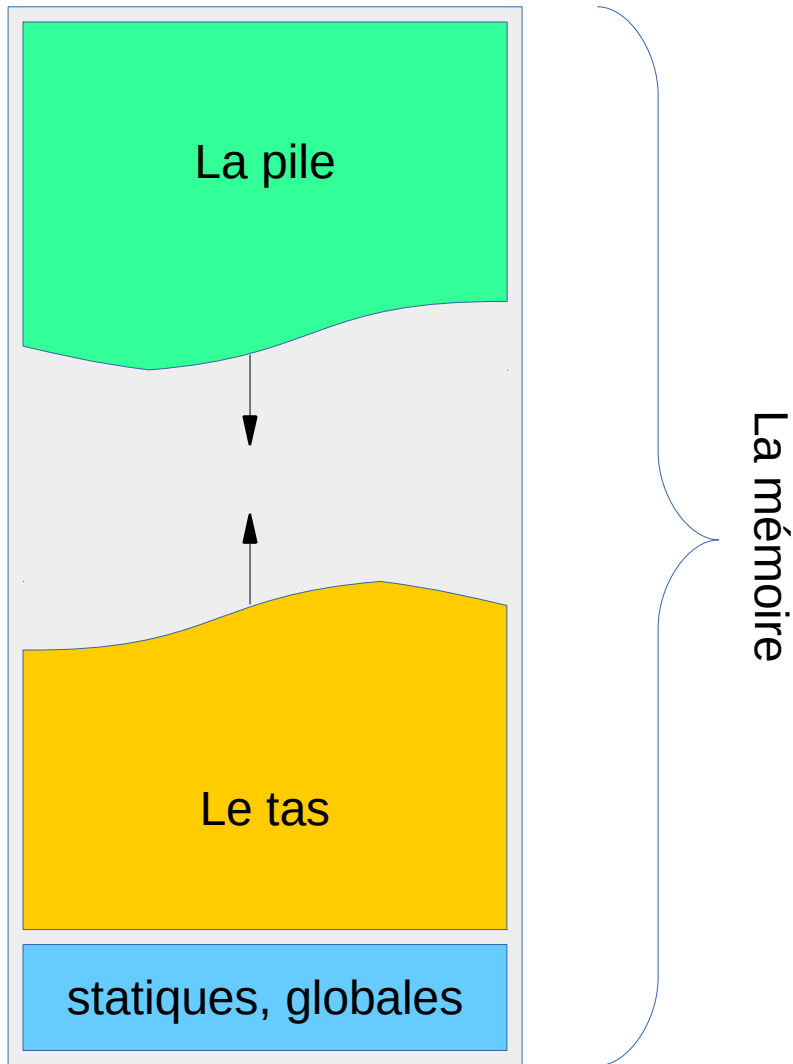
Le **tas** : ce sont les variables allouées dynamiquement (avec un malloc)

Taille dynamique

Le **data** : (entre autres) les instructions, les constantes et les variables globales.

Taille connue d'avance

# La pile... le tas...

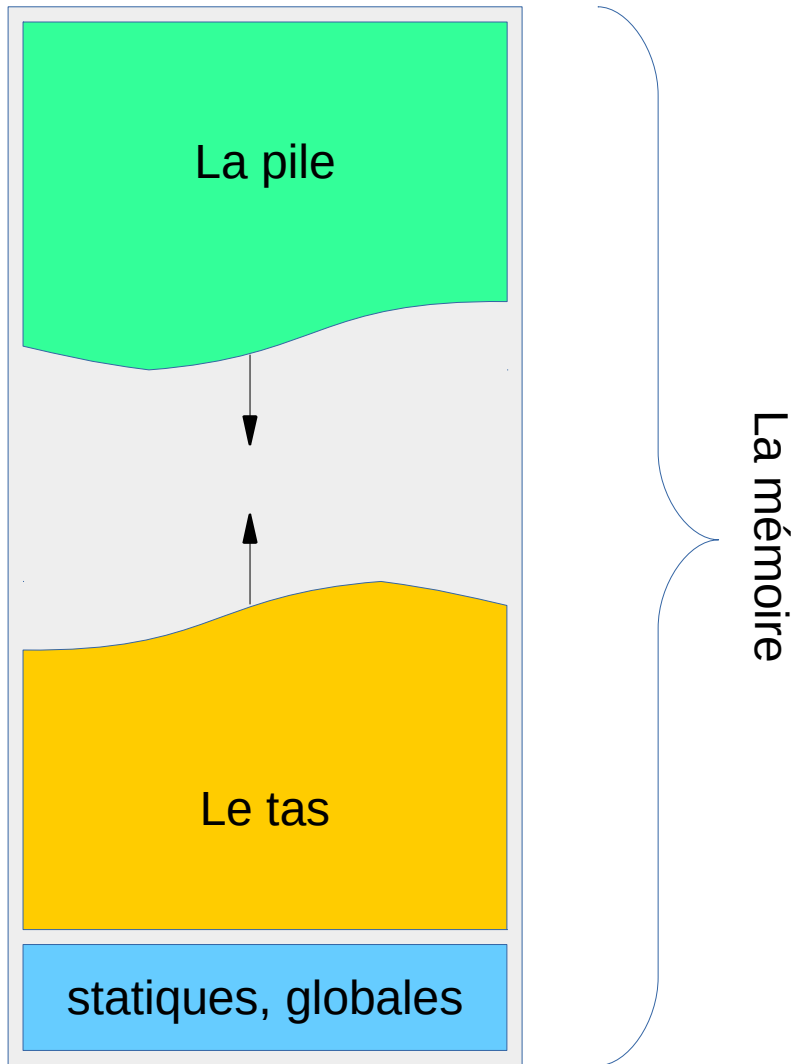


La **pile** : on la connaît, c'est les variables locales.

Le **tas** : ce sont les variables allouées dynamiquement (avec un malloc)

Le **data** : (entre autres) les instructions, les variables globales.

# La pile... le tas...



La **pile** : on la connaît, c'est les variables locales.

Taille dynamique

Le **tas** : ce sont les variables allouées dynamiquement (avec un malloc)

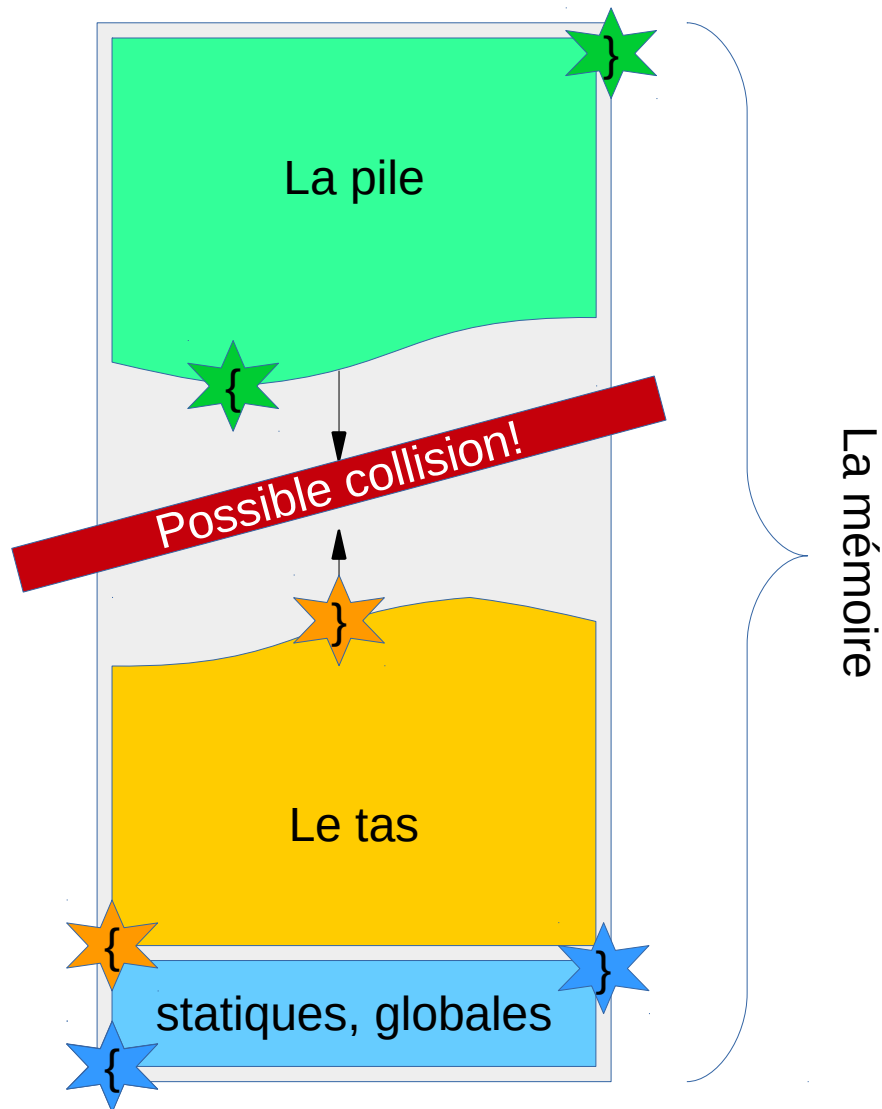
Taille dynamique

Le **data** : (entre autres) les instructions, les constantes et les variables globales.

Taille connue d'avance



# La pile... le tas...



La **pile** : on la connaît, c'est les variables locales.

Taille dynamique

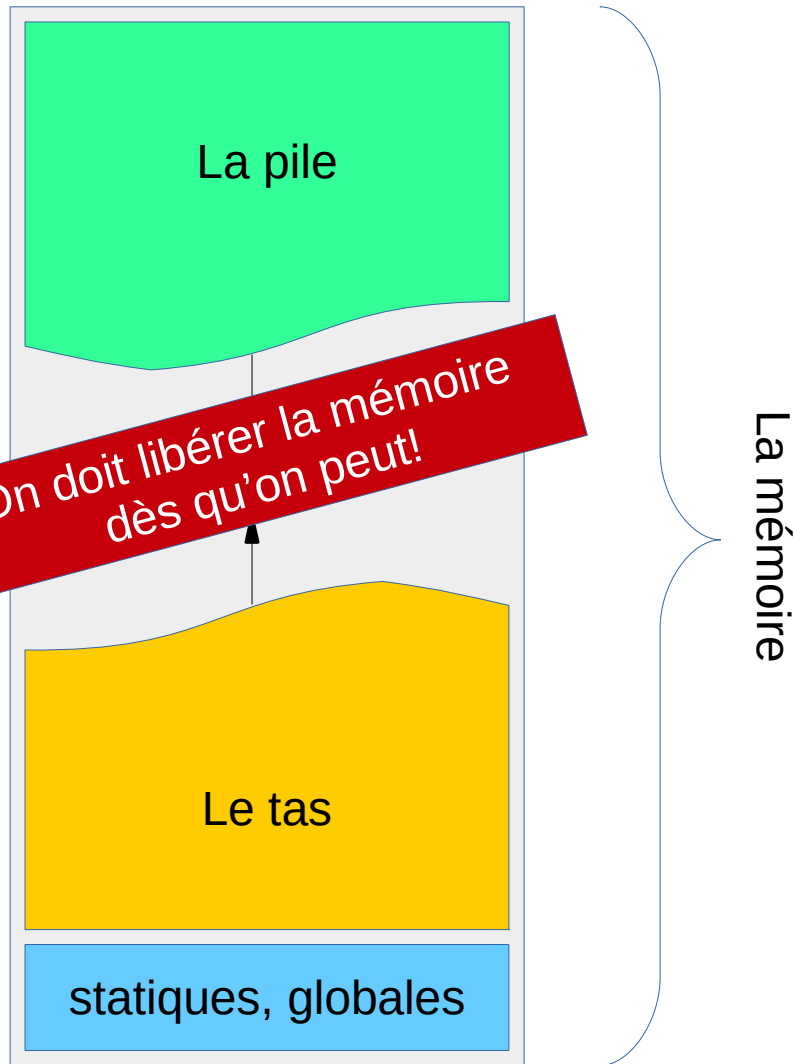
Le **tas** : ce sont les variables allouées dynamiquement (avec un `malloc`)

Taille dynamique

Le **data** : (entre autres) les instructions, les constantes et les variables globales.

Taille connue d'avance

# La pile... le tas...



La **pile** : on la connaît, c'est les variables locales.

Mémoire libérée au return.

Le **tas** : ce sont les variables allouées dynamiquement (avec un malloc)

Mémoire libérée avec la fonction free (ou à la fin de l'exécution).

Le **data** : (entre autres) les instructions, les constantes et les variables globales.

Mémoire libérée à la fin de l'exécution du programme.

# la fonction malloc plus en détail...

*memory allocation.*  
(Java : *new*)

```
void* malloc(size_t taille);
```

Retourne un pointeur vers n'importe quoi (**void**) et réserve *taille* octets à partir de cette adresse.

Par exemple :

**sizeof** retourne le nombre d'octets que prend le type

```
#include <stdlib.h>
```

```
int* a = (int*)malloc(sizeof(int));
```

```
int* b = (int*)malloc(sizeof(int) * 6);
```

```
int* c = (int*)malloc(24);
```

# la fonction malloc plus en détail...

```
void* malloc(size_t taille);
```

Si l'allocation a échoué, le pointeur sera **NULL**.

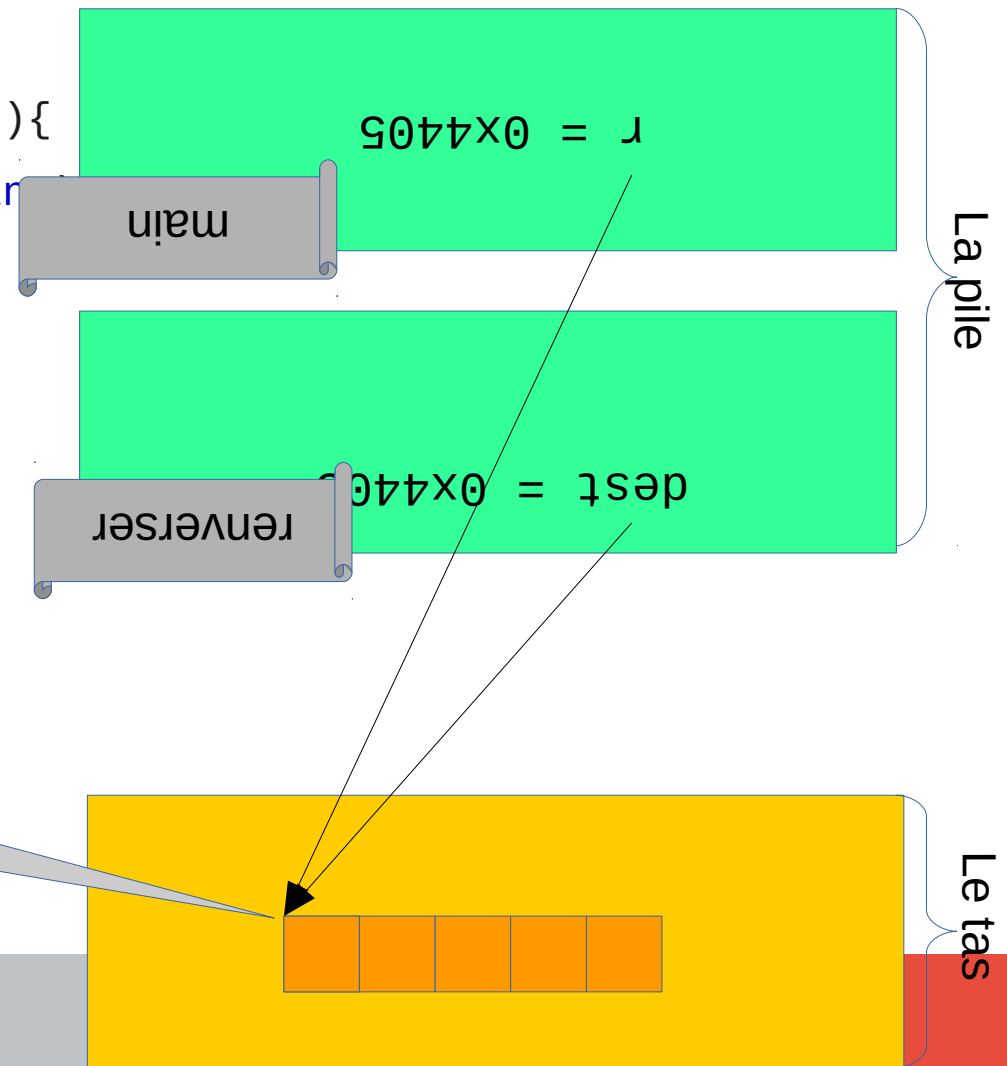
Par exemple :

```
int* a = (int*)malloc(sizeof(int));  
if (a == NULL){  
    fprintf(stderr, "Erreur de mémoire!");  
    return -1;  
}
```

# La libération de la mémoire

## Libérons la mémoire réservée dans le tas

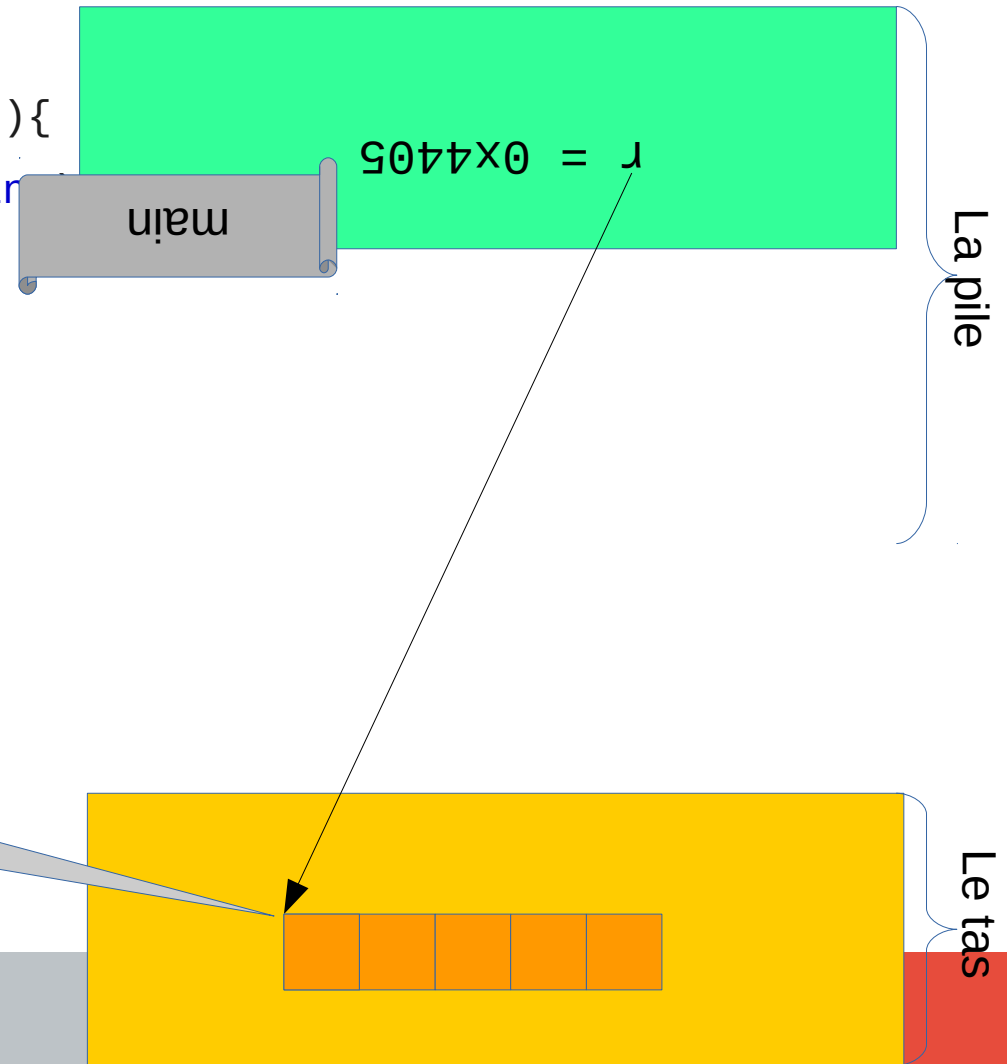
```
int* renverser(int source[], int taille){  
    int* dest = malloc(taille * sizeof(int))  
    ...  
    return dest;  
}  
  
int main(){  
    ...  
    int* r = renverser(t, 6);  
    ...  
    free(r);  
}
```



# La libération de la mémoire

## Libérons la mémoire réservée dans le tas

```
int* renverser(int source[], int taille){  
    int* dest = malloc(taille * sizeof(int));  
    ...  
    return dest;  
}  
  
int main(){  
    ...  
    int* r = renverser(t, 6);  
    ...  
    free(r);  
}
```

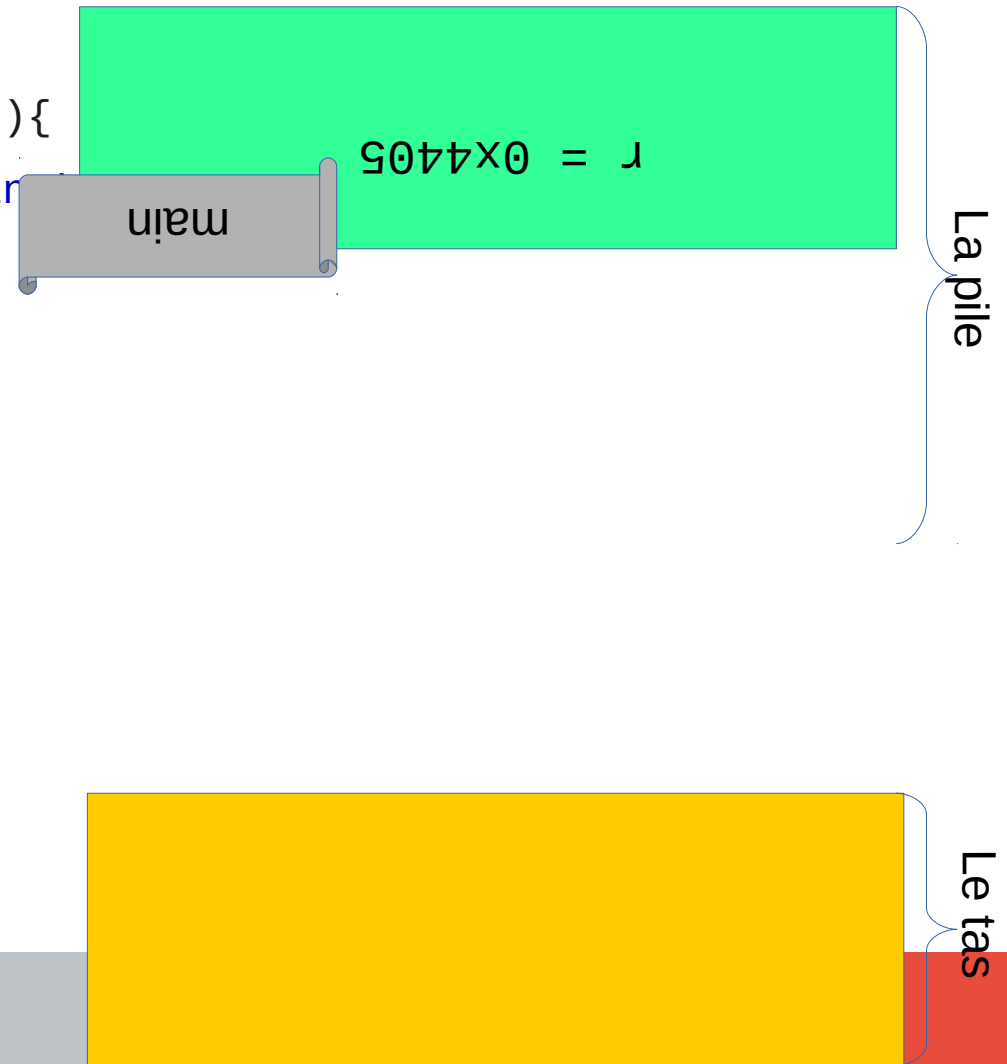


# La libération de la mémoire

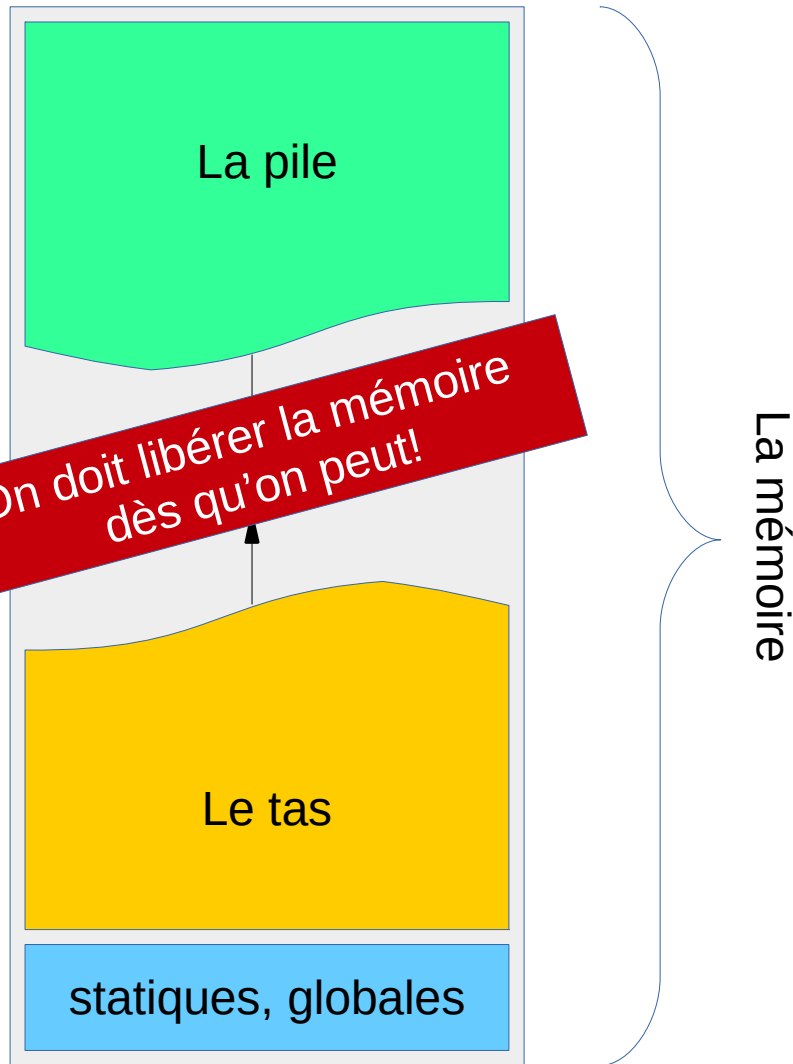
## Libérons la mémoire réservée dans le tas

```
int* renverser(int source[], int taille){
    int* dest = malloc(taille * sizeof(int));
    ...
    return dest;
}

int main(){
    ...
    int* r = renverser(t, 6);
    ...
    free(r);
}
```



# La pile... le tas...



La **pile** : on la connaît, c'est les variables locales.

Mémoire libérée au return.

Le **tas** : ce sont les variables allouées dynamiquement (avec un malloc)

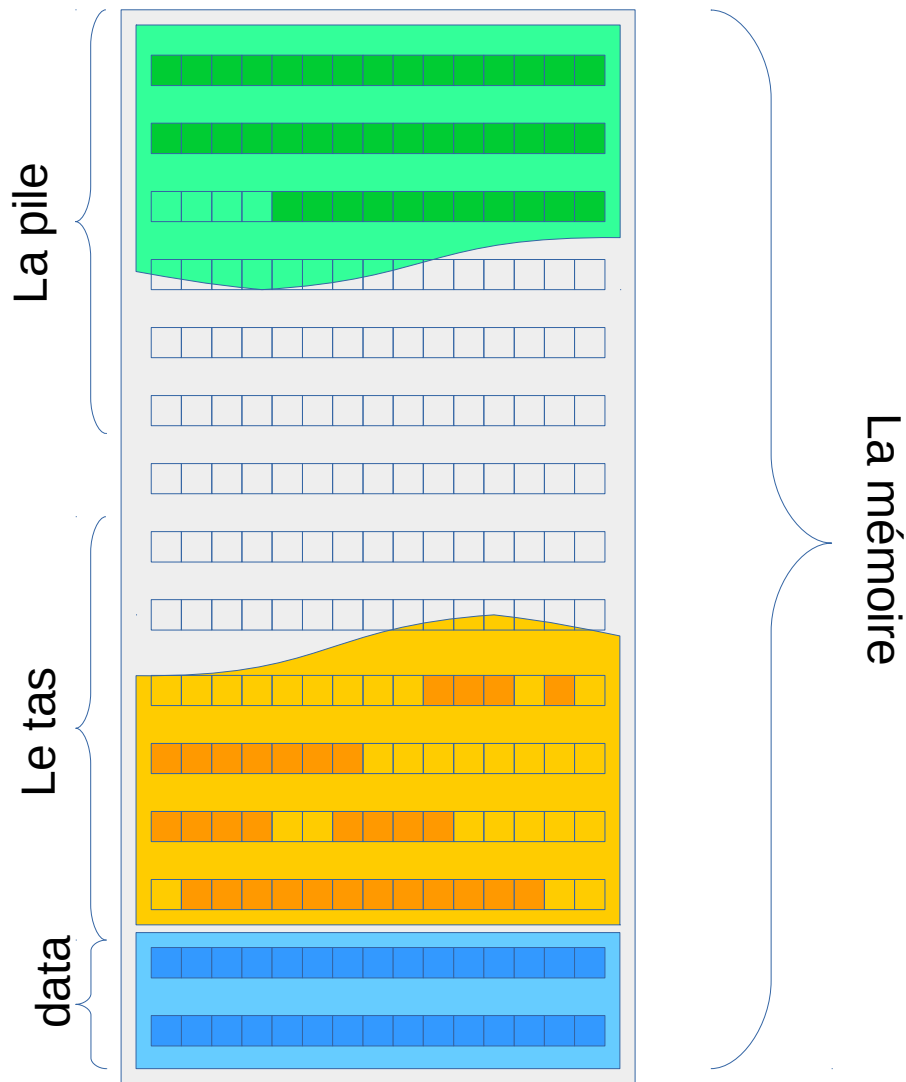
Mémoire libérée avec la fonction free (ou à la fin de l'exécution).

Le **data** : (entre autres) les instructions, les constantes et les variables globales.

Mémoire libérée à la fin de l'exécution du programme.



# La mémoire, en résumé.



## La **pile**

Elle est dense.

Sa taille s'ajuste au cours de l'exécution

La mémoire est **automatique**.

Son accès est rapide.

## Le **tas**

Il n'a pas de structure particulière.

Sa taille s'ajuste au cours de l'exécution

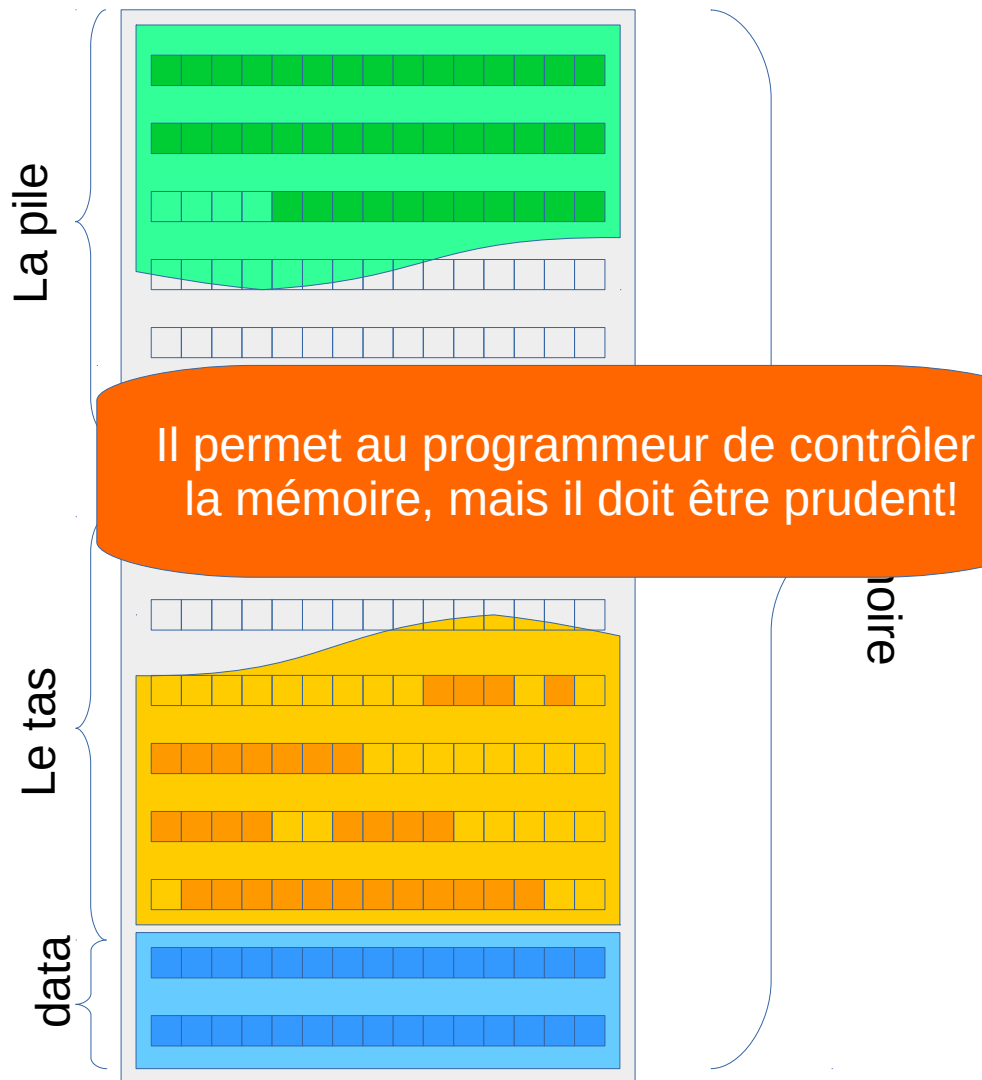
La mémoire est **dynamique**.

## Le **data**

Rapide, compact et **statique**.

Sa taille est fixée à la compilation.

# La mémoire, en résumé.



## La **pile**

Elle est dense.

Sa taille s'ajuste au cours de l'exécution

La mémoire est **automatique**.

Son accès est rapide.

## Le **tas**

Il n'a pas de structure particulière.

Sa taille s'ajuste au cours de l'exécution

La mémoire est **dynamique**.

## Le **data**

Rapide, compact et **statique**.

Sa taille est fixée à la compilation.

# Où se trouvent les données?

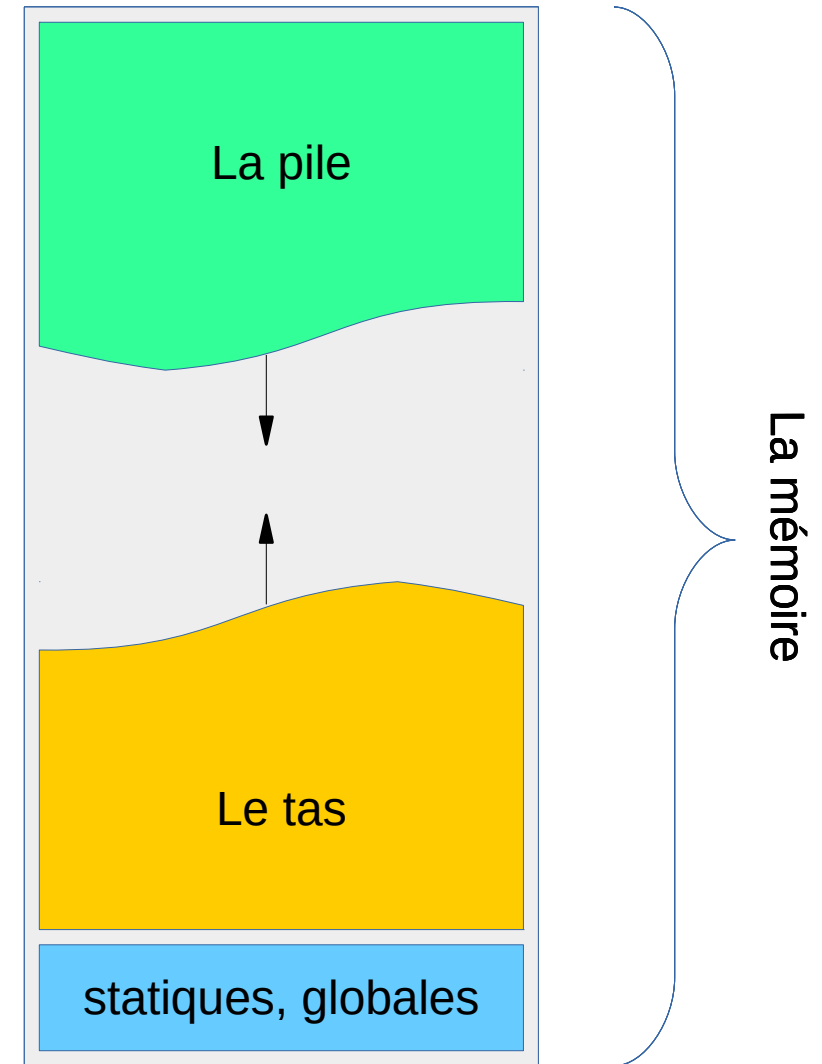
Ouvrez `dynamique.c`

**A)** Quelle est la valeur du pointeur `tabPile`?

**B)** Quelle est la valeur du pointeur `tabTas`?

**C)** Quelle est l'adresse de la fonction `imprimerTableau`?

**D)** Quelle est l'adresse de la variable globale `TAILLE`?



# stdlib

## La librairie stdlib vient avec toutes sortes de fonctions et de macros pour utiliser le tas :

sizeof (c'est le type retourné par `sizeof`)

`NULL` (macro)

`void* calloc(size_t nitems, size_t size);`

`void free(void* ptr);`

`void* malloc(size_t size);`

`void* realloc(void* ptr, size_t size);`

et bien d'autres...

Tableau de `nitems` cases de `size` octets chacun et initialise chaque case à 0.

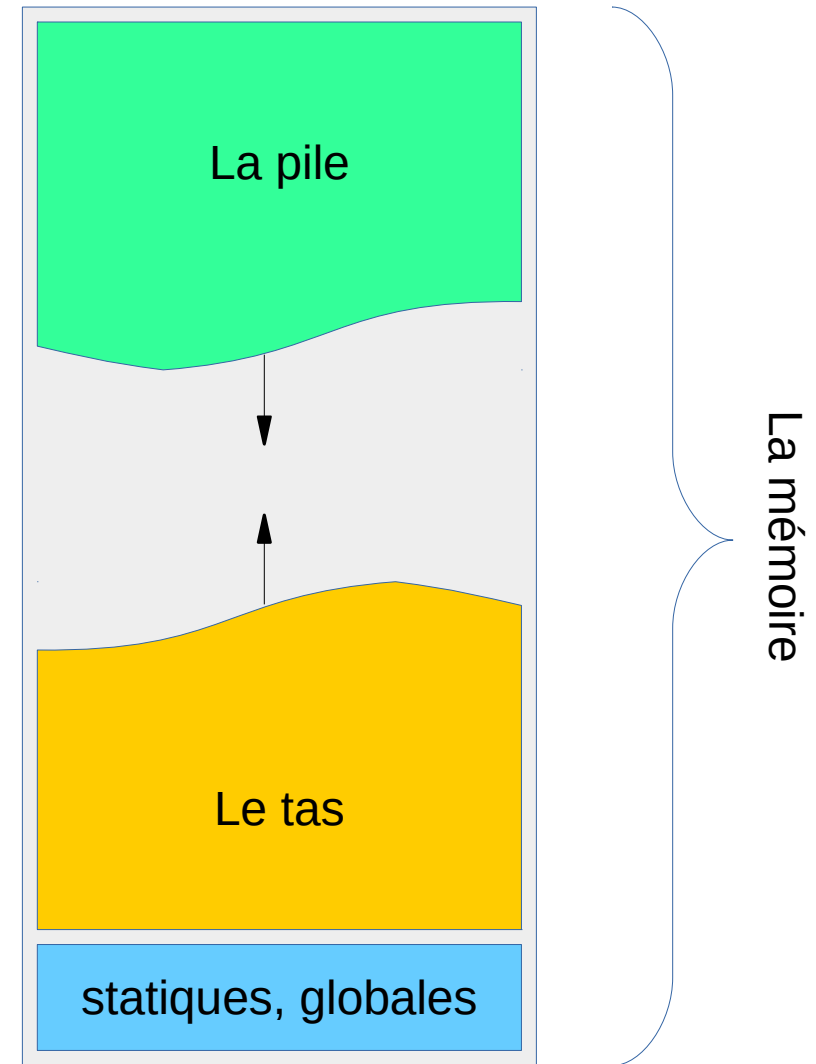
Libère la mémoire utilisée par `ptr` dans le tas.

Modifie la taille réservée pour par `ptr` dans le tas pour la nouvelle taille `size`.

# Les fonctions de `stdlib`

**E)** Combien il faut-il d'octets pour stocker une adresse? (utilisez `sizeof`)

**F)** À quelle valeur les bits sont initialisés dans le tas? Dans la pile?

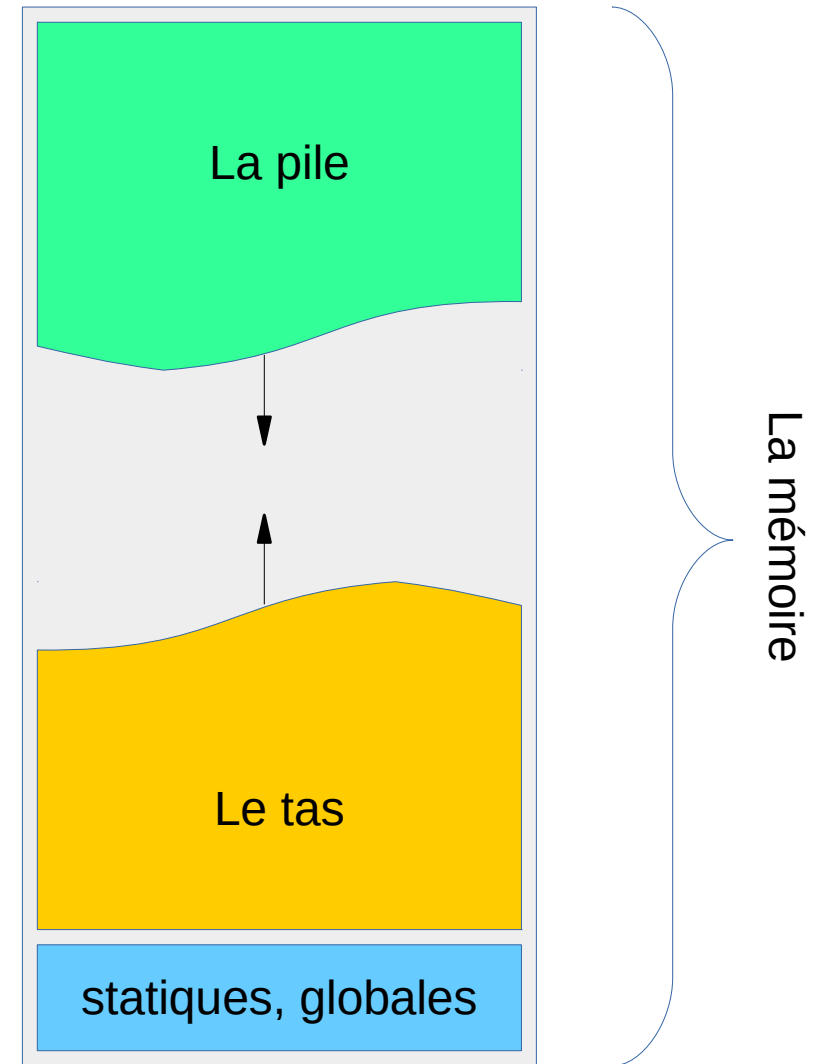


# La pile, le tas et les variables.

**G)** Que se passera-t-il si on ne libère la mémoire utilisée par `tableauTas` à même la fonction `fabriquerTableauTas`?

**H)** Quelle est la portée d'une variable dans la pile? Sa durée de vie?

**I)** Quelle est la portée d'une variable dans le tas? Sa durée de vie?



# Un exercice de programmation

Au TP1, on aura besoin de stocker un nombre arbitraire de **Clients**.

```
typedef struct Client Client;
struct Client{
    int instantArrivee; // L'instant où le client s'est ajouté à une file
    int nbArticles;     // Nb d'articles dans le panier du client
    // Vous pouvez ajouter des membres
};
```

Programmez la fonction

```
Client* creerClient(int instant, int nbArticles);
```

Appelez cette fonction dans le **main**.

*N'oubliez pas de libérer la mémoire du tas!*

« Ramassez les ordures! »

# Un exercice de programmation (suite et fin)

Quelle est la taille maximale que votre système d'exploitation permet d'allouer à la pile?

Sur UNIX : `ulimit -a`

Essayez de faire déborder la pile.  
Essayez de faire déborder le tas.

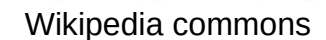
Déclarez de grandes variables  
Faites trop d'appels récursifs.

Demandez un très grand nombre  
d'octets à `malloc` ou `calloc`.





# Les structures de données unidimensionnelles



# Les structures unidimensionnelles

## **Les tableaux dynamiques**

Les listes chaînées

Les listes doublement chaînées

# Les tableaux dynamiques

On vous demande de programmer l'utilitaire suivant.

```
Entrez la mesure des angles (en degrés) qui forment  
le polygone. Pour terminer, entrez un nombre négatif
```

```
> 60, 90, 60, 150, -1
```



# Les tableaux dynamiques

On vous demande de programmer l'utilitaire suivant.

Entrez la mesure des angles (en degrés) qui forment le polygone. Pour terminer, entrez un nombre négatif

> 60, 90, 60, 150, -1



Vous ne connaissez pas d'avance le nombre d'angles à sauvegarder.

# Les tableaux dynamiques

On vous demande de programmer l'utilitaire suivant.

Entrez la mesure des angles (en degrés) qui forment le polygone. Pour terminer, entrez un nombre négatif

> 60, 90, 60, 150, -1



```
double angles[??];  
int i = 0;  
do{  
    scanf("%lf", angles + i);  
}while (angles[i++] >= 0);
```

# Les tableaux statiques

## Une première solution :

on définit un tableau assez long et on espère que ce sera suffisant...

```
double angles[100];  
int i = 0;  
do{  
    scanf("%lf", angles + i);  
}while (angles[i++] >= 0);
```

+ Simple à implémenter

- Peut déborder.

- Gaspille de l'espace.

# Les tableaux dynamique

**Une deuxième solution :**  
on définit un tableau assez  
long et on le rallonge au besoin...

+ Pas de débordement

- Gaspille de l'espace  
(Jusqu'à 100 doubles de trop)
- Les ajouts sont plus complexes

```
#define BUFF 100
int t = BUFF;
double* angles = (double*)
    malloc(BUFF * sizeof(double));
int i = 0;
do{
    scanf("%lf", angles + i);
    i++;
    if (i >= t)
    {
        t += BUFF;
        double* temp = (double*)
            realloc(angles,
                t * sizeof(double));
        if (temp == 0)
        {
            free(angles);
            return -1;
        }
        angles = temp;
    }
}while (angles[i - 1] >= 0);
free(angles);
```

# Les tableaux dynamiques

Exécution : 1 1 3 4 4 2 -6



```
#define BUFF 3
int t = BUFF;
double* angles = (double*)
    malloc(BUFF * sizeof(double));
int i = 0;
do{
    scanf("%lf", angles + i);
    i++;
    if (i >= t)
    {
        t += BUFF;
        double* temp = (double*)
            realloc(angles,
                t * sizeof(double));
        if (temp == 0)
        {
            free(angles);
            return -1;
        }
        angles = temp;
    }
}while (angles[i - 1] >= 0);
free(angles);
```



# Les tableaux dynamiques

Exécution : 1 1 3 4 4 2 -6



```
#define BUFF 3
int t = BUFF;
double* angles = (double*)
    malloc(BUFF * sizeof(double));
int i = 0;
do{
    scanf("%lf", angles + i);
    i++;
    if (i >= t)
    {
        t += BUFF;
        double* temp = (double*)
            realloc(angles,
                t * sizeof(double));
        if (temp == 0)
        {
            free(angles);
            return -1;
        }
        angles = temp;
    }
}while (angles[i - 1] >= 0);
free(angles);
```

# Les tableaux dynamiques

Exécution : 1 1 3 4 4 2 -6



```
#define BUFF 3
int t = BUFF;
double* angles = (double*)
    malloc(BUFF * sizeof(double));
int i = 0;
do{
    scanf("%lf", angles + i);
    i++;
    if (i >= t)
    {
        t += BUFF;
        double* temp = (double*)
            realloc(angles,
                t * sizeof(double));
        if (temp == 0)
        {
            free(angles);
            return -1;
        }
        angles = temp;
    }
}while (angles[i - 1] >= 0);
free(angles);
```

# Les tableaux dynamiques

Exécution : 1 1 3 4 4 2 -6

angles

0	1
1	1
2	3
3	
4	
5	

i = 3  
t = 6

```
#define BUFF 3
int t = BUFF;
double* angles = (double*)
    malloc(BUFF * sizeof(double));
int i = 0;
do{
    scanf("%lf", angles + i);
    i++;
    if (i >= t)
    {
        t += BUFF;
        double* temp = (double*)
            realloc(angles,
                t * sizeof(double));
        if (temp == 0)
        {
            free(angles);
            return -1;
        }
        angles = temp;
    }
}while (angles[i - 1] >= 0);
free(angles);
```

# Les tableaux dynamiques

Exécution : 1 1 3 4 4 2 -6

angles

0	1
1	1
2	3
3	4
4	
5	

i = 4  
t = 6

```
#define BUFF 3
int t = BUFF;
double* angles = (double*)
    malloc(BUFF * sizeof(double));
int i = 0;
do{
    scanf("%lf", angles + i);
    i++;
    if (i >= t)
    {
        t += BUFF;
        double* temp = (double*)
            realloc(angles,
                t * sizeof(double));
        if (temp == 0)
        {
            free(angles);
            return -1;
        }
        angles = temp;
    }
}while (angles[i - 1] >= 0);
free(angles);
```

# Les tableaux dynamiques

Exécution : 1 1 3 4 4 2 -6

angles

0	1
1	1
2	3
3	4
4	4
5	

i = 5  
t = 6

```
#define BUFF 3
int t = BUFF;
double* angles = (double*)
    malloc(BUFF * sizeof(double));
int i = 0;
do{
    scanf("%lf", angles + i);
    i++;
    if (i >= t)
    {
        t += BUFF;
        double* temp = (double*)
            realloc(angles,
                t * sizeof(double));
        if (temp == 0)
        {
            free(angles);
            return -1;
        }
        angles = temp;
    }
}while (angles[i - 1] >= 0);
free(angles);
```

# Les tableaux dynamiques

Exécution : 1 1 3 4 4 2 -6

angles

0	1
1	1
2	3
3	4
4	4
5	2
6	
7	
8	

i = 6  
t = 9

```
#define BUFF 3
int t = BUFF;
double* angles = (double*)
    malloc(BUFF * sizeof(double));
int i = 0;
do{
    scanf("%lf", angles + i);
    i++;
    if (i >= t)
    {
        t += BUFF;
        double* temp = (double*)
            realloc(angles,
                t * sizeof(double));
        if (temp == 0)
        {
            free(angles);
            return -1;
        }
        angles = temp;
    }
}while (angles[i - 1] >= 0);
free(angles);
```

# Les tableaux dynamiques

Exécution : 1 1 3 4 4 2 -6

angles

0	1
1	1
2	3
3	4
4	4
5	2
6	-6
7	
8	

i = 7  
t = 9

```
#define BUFF 3
int t = BUFF;
double* angles = (double*)
    malloc(BUFF * sizeof(double));
int i = 0;
do{
    scanf("%lf", angles + i);
    i++;
    if (i >= t)
    {
        t += BUFF;
        double* temp = (double*)
            realloc(angles,
                t * sizeof(double));
        if (temp == 0)
        {
            free(angles);
            return -1;
        }
        angles = temp;
    }
}while (angles[i - 1] >= 0);
free(angles);
```

# Les tableaux dynamiques

On peut rendre le code plus lisible avec une structure

```
typedef struct TabDyn TabDyn;  
struct TabDyn{  
    double* tab;  
    int dernier;  
    int taille;  
};
```

```
void initialiser(TabDyn* tabdyn);  
int ajouterElement(TabDyn* tabdyn, double e);  
void libererMemoire(TabDyn* tabdyn);  
void imprimerTabDyn(TabDyn tabdyn);
```

```
TabDyn tabDyn;  
initialiser(&tabDyn);  
double a;  
do{  
    scanf("%lf", &a);  
    if (!ajouterElement(&tabDyn, a))  
        return -1;  
}while (a >= 0);  
imprimerTabDyn(tabDyn);  
libererMemoire(&tabDyn);
```



# Les tableaux dynamiques

On peut rendre le code plus lisible avec une structure

```
typedef struct TabDyn TabDyn;  
struct TabDyn{  
    double* tab;  
    int dernier;  
    int taille;  
};
```

```
void initialiser(TabDyn* tabdyn);  
int ajouterElement(TabDyn* tabdyn, double e);  
void libererMemoire(TabDyn* tabdyn);  
void imprimerTabDyn(TabDyn tabdyn);
```

```
TabDyn tabDyn;  
initialiser(&tabDyn);  
double a;  
do{  
    scanf("%lf", &a);  
    if (!ajouterElement(&tabDyn, a))  
        return -1;  
}while (a >= 0);  
imprimerTabDyn(tabDyn);  
libererMemoire(&tabDyn);
```

En déplaçant du code vu plus haut, implémentez ces fonctions.

# Ajout, modification et suppression

Étudions un problème différent.

En plus de stocker un nombre arbitraire d'éléments, on doit permettre à l'utilisateur d'ajouter, de modifier et de supprimer des valeurs

```
Entrez un angle
> 60
Angles entrés : 1:60
(R)etirer - (M)odifier - (A)jouter - (T)erminer
> A
Entrez un angle
> 200
Angles entrés : 1:60, 2:200
(R)etirer - (M)odifier - (A)jouter - (T)erminer
> M
Quel angle voulez-vous modifier?
1:60, 2:200
> 2
Entrez la nouvelle valeur de l'angle 2
> 60
Angles entrés :
1:60, 2:200
(R)etirer - (M)odifier - (A)jouter - (T)erminer
> R
Quel angle voulez-vous retirer 1:60, 2:20
> 1
Angles entrés : 1:20
(R)etirer - (M)odifier - (A)jouter - (T)erminer
> ...
```

# Ajout, modification et suppression

angles  
nbElements = 6

Enlevons l'élément  
à l'indice 2.

0	1
1	1
2	3
3	4
4	4
5	2
6	
7	
8	

# Ajout, modification et suppression

angles  
nbElements = 5

Enlevons l'élément  
à l'indice 2.

0	1
1	1
2	4
3	4
4	2
5	2
6	
7	
8	

Ajout à la fin est rapide  
Retrait à la fin est rapide

Tout autre ajout/retrait est lent...

# Les structures unidimensionnelles

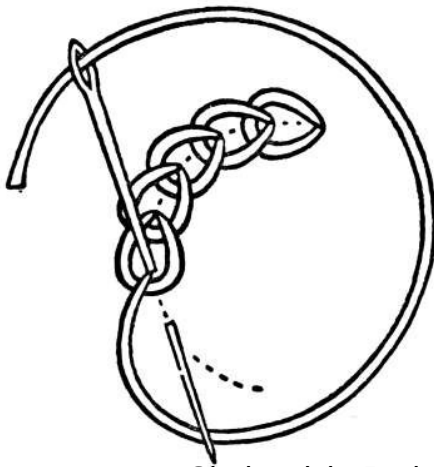
Les tableaux dynamiques

**Les listes chaînées**

Les listes doublement chaînées

# Les listes chaînées

En utilisant une structure différente, on peut accélérer de beaucoup la suppression et l'ajout en position arbitraire...



Chain stich. Project Gutenberg

```
typedef struct Noeud Noeud;  
struct Noeud{  
    int donnee;  
    Noeud* suivant;  
};
```

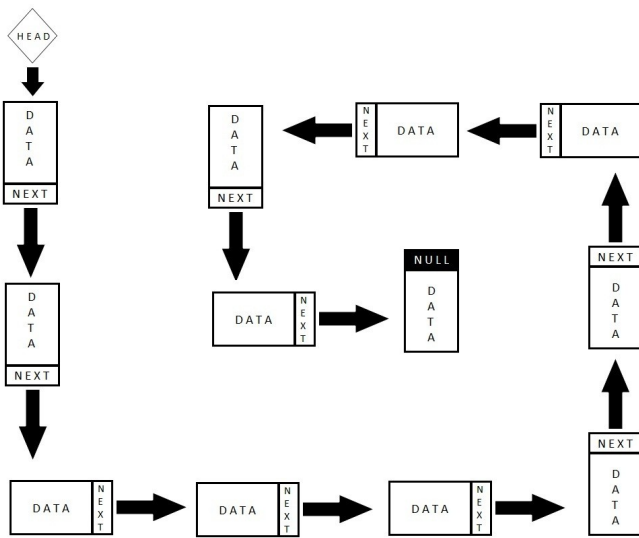
Chaque Noeud connaît  
sa donnée et le Noeud  
suivant.

# Les listes chaînées

En utilisant une structure  
différente, on peut  
accélérer de beaucoup la  
suppression et l'ajout en  
position arbitraire...

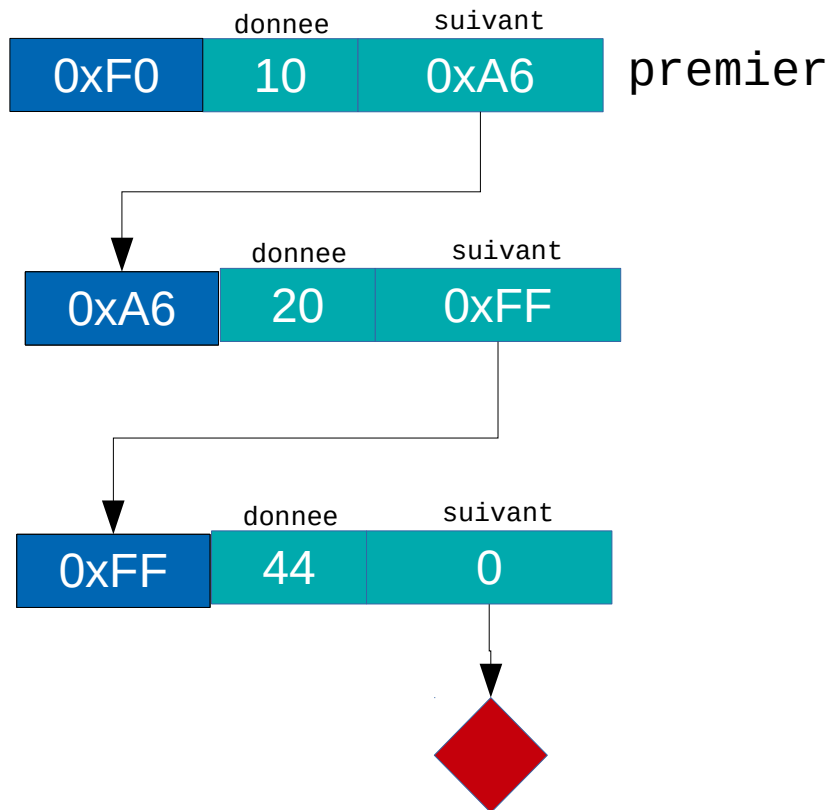
```
typedef struct Noeud Noeud;  
struct Noeud{  
    int donnee;  
    Noeud* suivant;  
};
```

Chaque Noeud connaît sa donnée et le Noeud suivant.



Wikipedia commons

# Les listes chaînées



```
typedef struct Noeud Noeud;
```

```
struct Noeud{  
    int donnee;  
    Noeud* suivant;  
};
```

```
Noeud premier;  
premier.donnee = 10;  
Noeud deuxieme;  
deuxieme.donnee = 20;  
Noeud troisieme;  
troisieme.donnee = 44;
```

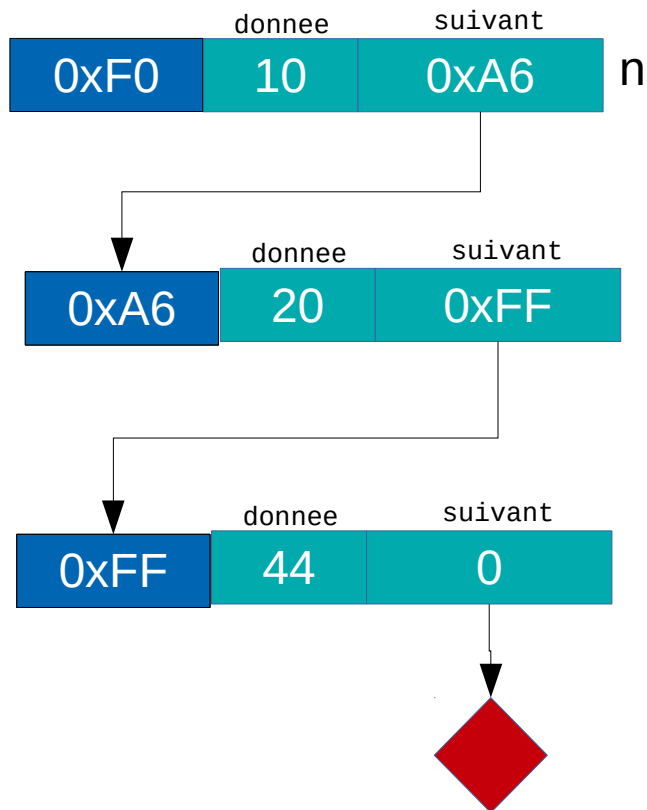
```
premier.suivant = &deuxieme;  
deuxieme.suivant = &troisieme;  
troisieme.suivant = NULL;
```



# Les listes chaînées

La suppression est facile!

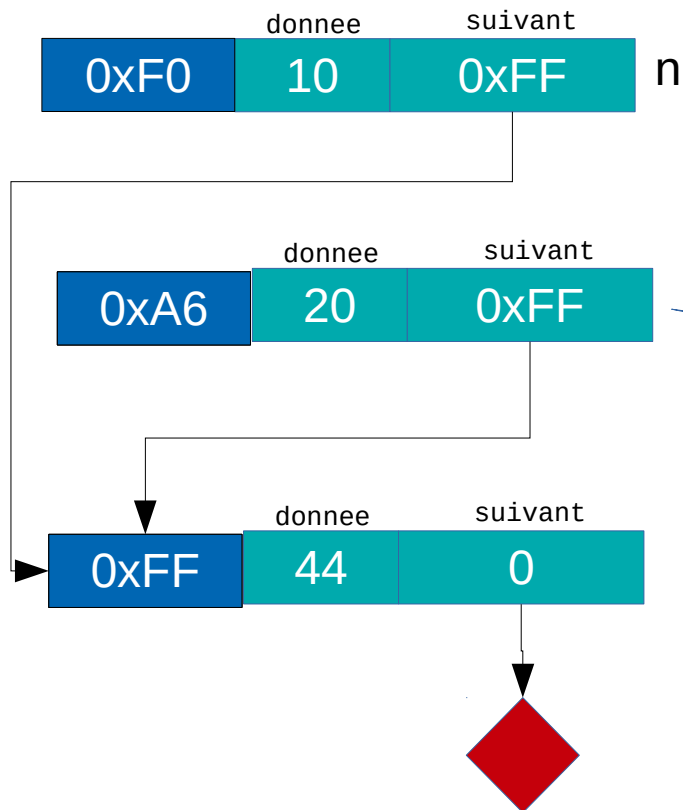
```
n.suivant = n.suivant->suivant;
```



# Les listes chaînées

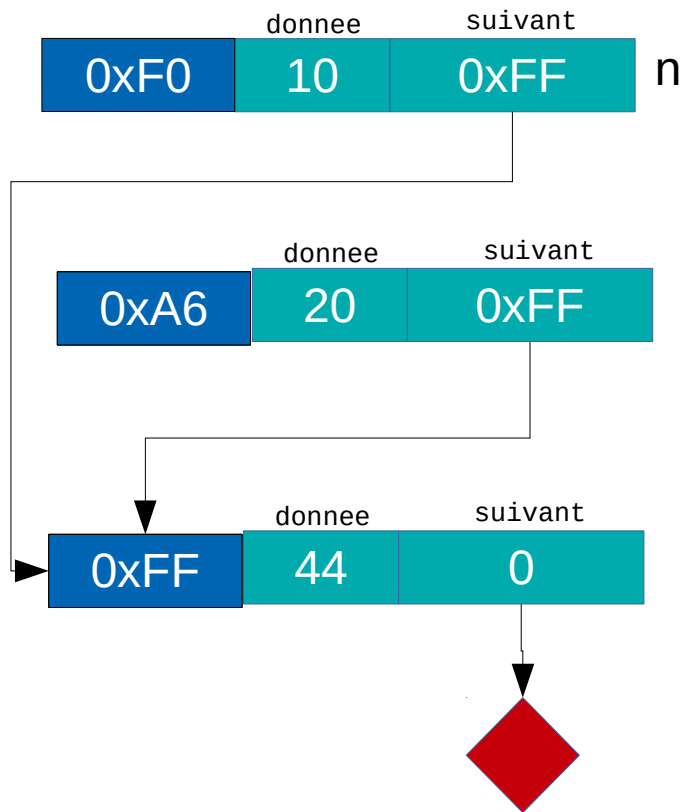
La suppression est facile!

```
n.suivant = n.suivant->suivant;
```



Ce Noeud est perdu...

# Les listes chaînées



Dans une liste chaînée, chaque nœud conserve une référence vers le nœud suivant. Le dernier nœud a une référence nulle.

- + Pas de débordement
- + Pas de sur-réservation d'espace
- + Ajout et suppression possibles en position arbitraire

- Accès séquentiel
- Données dispersées en mémoire
- Plus volumineux en mémoire
- On ne peut pas « reculer »

# Les listes chaînées

**Comment parcourir une liste chaînée?**

**Comment ajouter un élément au début? Au milieu? À la fin?**

```
typedef struct Noeud Noeud;  
struct Noeud{  
    int donnee;  
    Noeud* suivant;  
};  
  
typedef struct Liste Liste;  
struct Liste{  
    Noeud* premier;  
};
```

# Les listes chaînées

**Comment parcourir  
liste chaînée?**

**Comment ajouter un  
élément au début?  
milieu? À la fin?**

```
typedef struct Noeud Noeud;  
  
struct Noeud{  
    int donnee;  
    Noeud* suivant;  
};  
  
typedef struct Liste Liste;  
  
struct Liste{  
    Noeud* premier;  
};
```

*Imprimer la liste*

$p \leftarrow \text{liste.premier}$

Tant que  $p$  n'est pas nul, faire

Imprimer  $p.donnee$

$p \leftarrow p.suivant$

*Ajout d'un élément nouveau au début*

$p \leftarrow \text{liste.premier};$

Si  $p$  est nul, alors

$p \leftarrow \text{nouveau}$

$p.suivant \leftarrow \text{nul}$

Sinon, alors

$\text{nouveau.suivant} \leftarrow p$

$\text{liste.premier} \leftarrow \text{nouveau}$

*Retrait d'un élément au début*

Si  $\text{liste.premier}$  n'est pas nul, alors

$r \leftarrow \text{liste.premier.donnee}$

$\text{liste.premier} \leftarrow \text{liste.premier.suivant}$

retourner  $r$

# Les listes chaînées

**Comment parcourir une liste chaînée?**

**Comment ajouter un élément au début? Au milieu? À la fin?**

```
typedef struct Noeud Noeud;  
struct Noeud{  
    int donnee;  
    Noeud* suivant;  
};  
  
typedef struct Liste Liste;  
struct Liste{  
    Noeud* premier;  
};
```

```
void imprimerListe(Liste liste)  
{  
    Noeud* p = liste.premier;  
    while (p != NULL)  
    {  
        // ...  
    }  
}  
  
void ajouterDebut(Liste* liste,  
                  int data)  
{  
    if (liste->premier == NULL)  
        liste->premier = nouveau;  
    else  
    {  
        // ...  
    }  
}  
  
int retirerDebut(Liste* liste)  
{  
    if (liste->premier != NULL)  
    {  
        // ...  
    }  
}
```

# Les listes chaînées

**Comment parcourir une liste chaînée?**

**Comment ajouter un élément au début? Au milieu? À la fin?**

```
typedef struct Noeud Noeud;  
struct Noeud{  
    int donnee;  
    Noeud* suivant;  
};  
  
typedef struct Liste Liste;  
struct Liste{  
    Noeud* premier;  
};
```

```
void imprimerListe(Liste liste)  
{  
    Noeud* p = liste.premier;  
    while (p != NULL)  
    {  
        // ...  
    }  
}
```

```
void ajouterDebut(Liste* liste,  
                  int data)  
{  
    if (liste->premier == NULL)  
        liste->premier = nouveau;  
    else  
    {  
        // ...  
    }  
}
```

Complétez les fonctions.  
Comment implémenteriez vous une pile?  
Une file?

# Les listes chaînées

**Comment parcourir  
liste chaînée?**

**Comment ajouter un  
élément au début?  
milieu? À la fin?**

```
typedef struct Noeud Noeud;  
struct Noeud{  
    int donnee;  
    Noeud* suivant;  
};  
typedef struct Liste Liste;  
struct Liste{  
    Noeud* premier;  
};
```

*Imprimer la liste*

```
p ← liste.premier  
Tant que p n'est pas nul, faire  
    Imprimer p.donnee  
    p ← p.suivant
```

*Ajout d'un élément nouveau au début*

```
p ← liste.premier;  
Si p est nul, alors  
    p ← nouveau  
    p.suivant ← nul  
Sinon, alors  
    nouveau.suivant ← p  
    liste.premier ← nouveau
```

*Retrait d'un élément au début*

```
Si liste.premier n'est pas nul, alors  
    liste.premier ← liste.premier.suivant
```

Donnez le pseudo-code pour ajouter  
ou retirer un élément à la position *i* dans  
la liste chaînée.



## Les listes chaînées - à retenir

En plus de sa donnée, chaque nœud garde une référence vers le nœud suivant. Le dernier nœud réfère vers le pointeur nul.

On peut ajouter, lire, modifier et supprimer des nœuds à condition qu'on connaisse l'adresse du premier nœud.

La taille d'une liste chaînée est dynamique.

L'ajout et la suppression autour d'une adresse connue sont optimales.

Les données ne sont jamais déplacées, seules les références le sont.



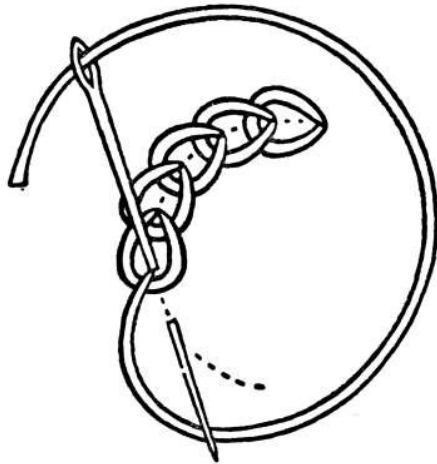
# Les structures unidimensionnelles

Les tableaux dynamiques

Les listes chaînées

**Les listes doublement chaînées**

# Les listes doublement chaînées

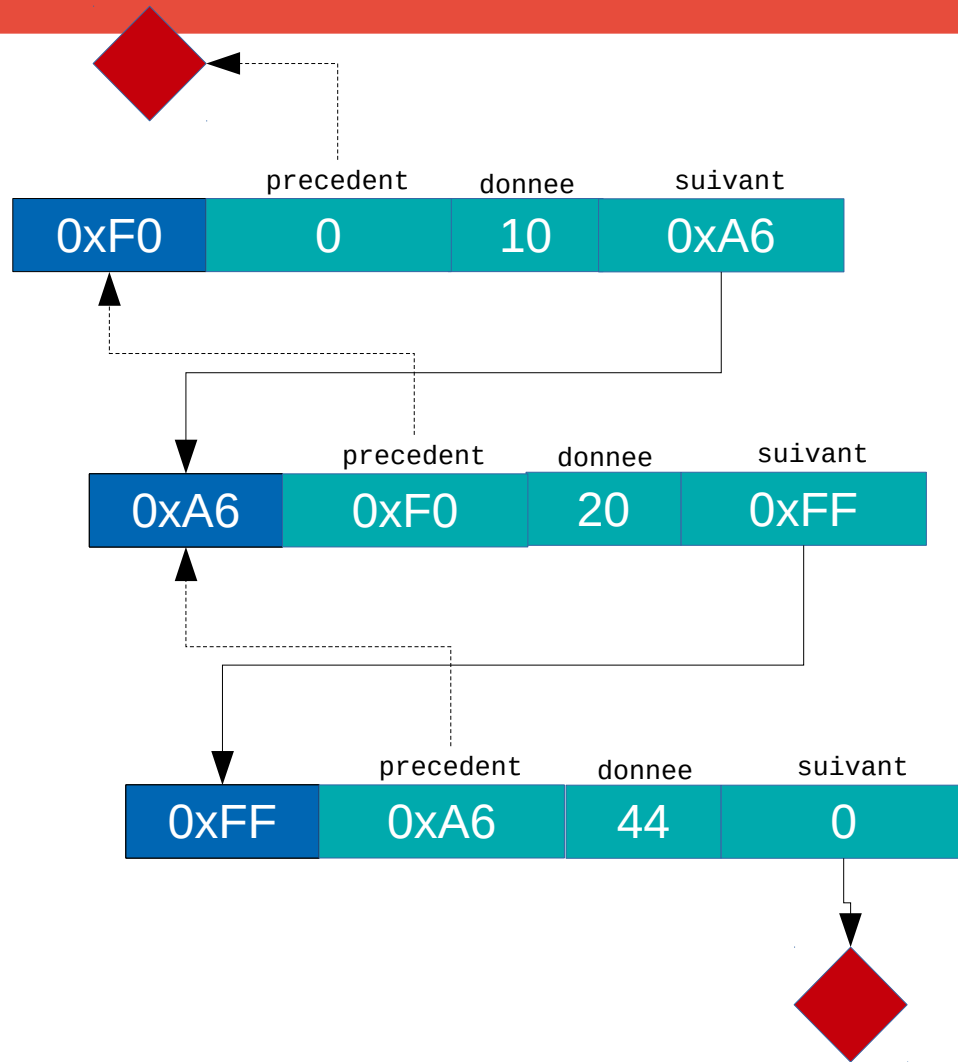


Chain stich. Project Gutenberg

```
typedef struct Noeud Noeud;  
  
struct Noeud{  
    Noeud* precedent;  
    int donnee;  
    Noeud* suivant;  
};
```

Chaque Noeud connaît sa donnée, le Noeud suivant, et le Noeud précédent.

# Les listes doublement chaînées



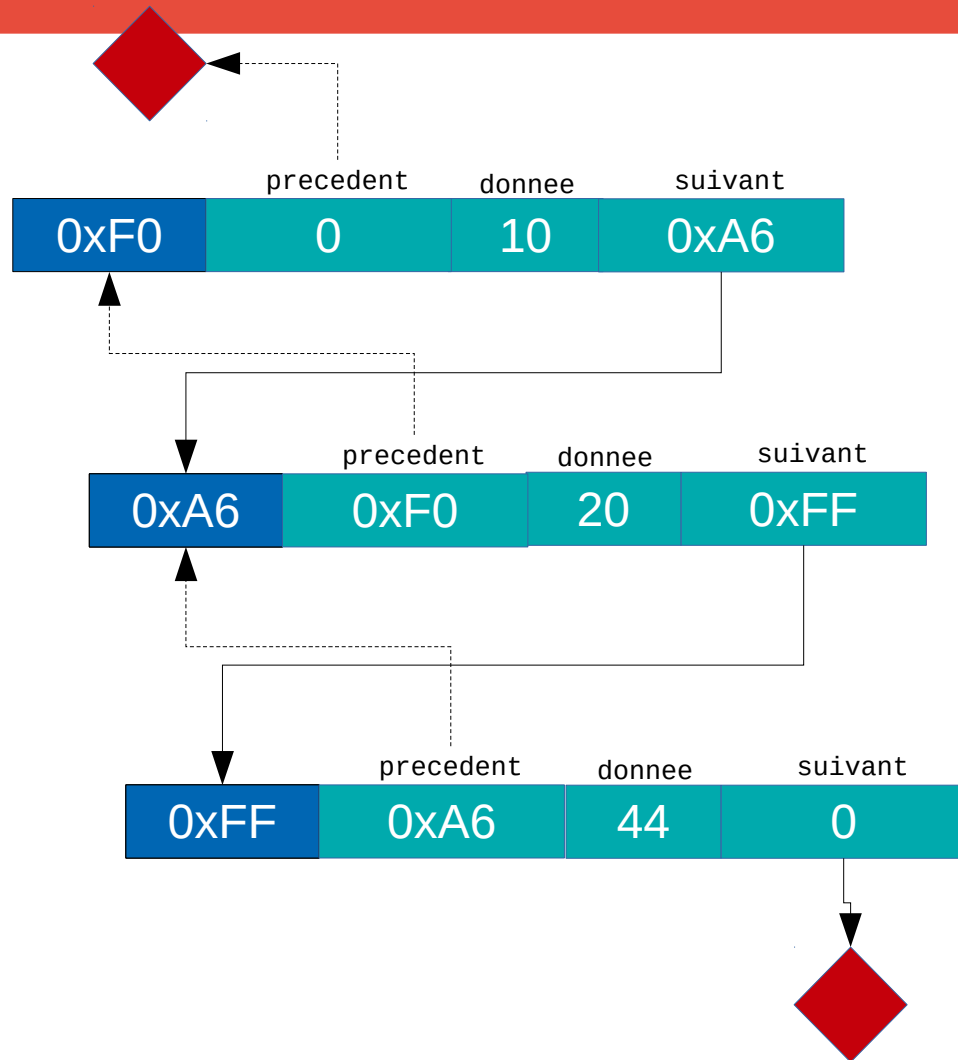
```
typedef struct Noeud Noeud;
```

```
struct Noeud{  
    Noeud* precedent;  
    int donnee;  
    Noeud* suivant;  
};
```

```
Noeud premier;  
premier.donnee = 10;  
Noeud deuxieme;  
deuxieme.donnee = 20;  
Noeud troisieme;  
troisieme.donnee = 44;
```

```
premier.precedent = NULL;  
premier.suivant = &deuxieme;  
deuxieme.precedent = &premier;  
deuxieme.suivant = &troisieme;  
troisieme.precedent = &deuxieme;  
troisieme.suivant = NULL;
```

# Les listes doublement chaînées



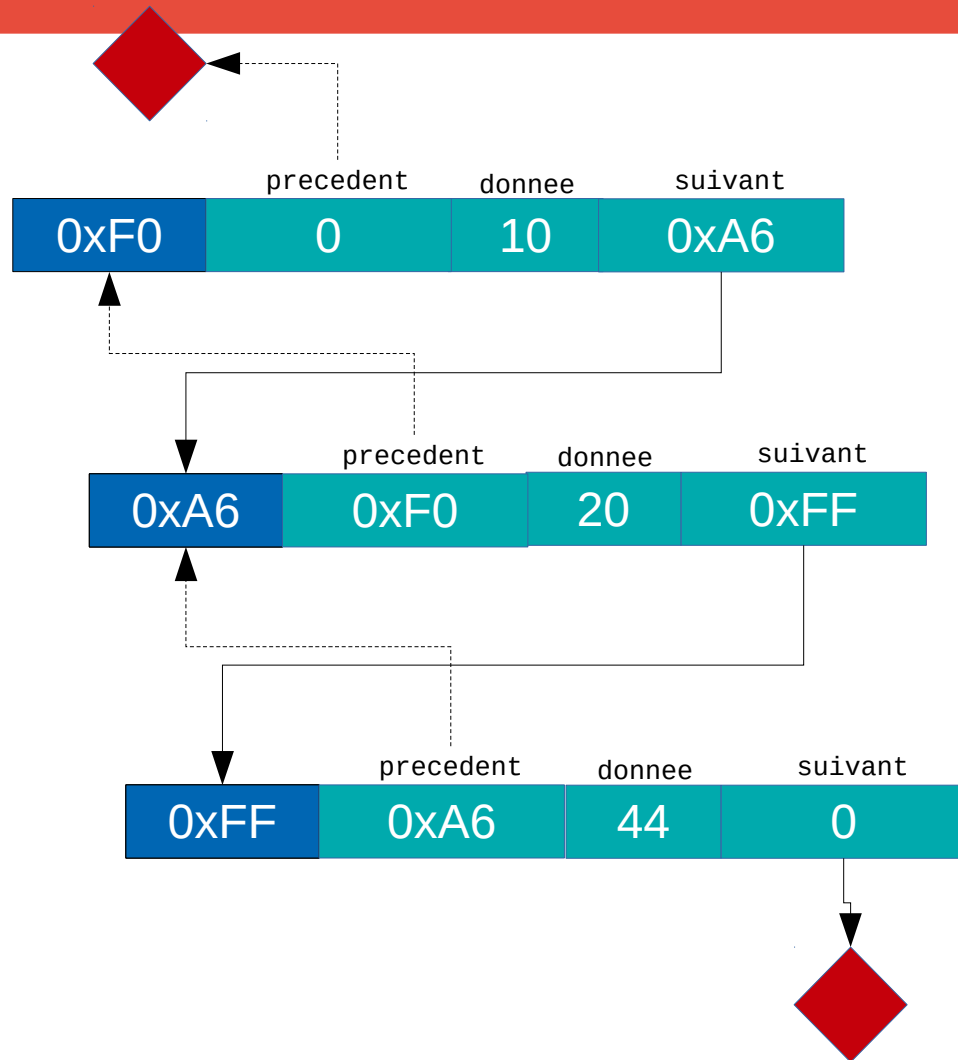
```
typedef struct Noeud Noeud;  
struct Noeud{  
    Noeud* precedent;  
    int donnee;  
    Noeud* suivant;  
};
```

```
Noeud premier;  
premier.donnee = 10;  
Noeud deuxieme;  
deuxieme.donnee = 20;  
Noeud troisieme;  
troisieme.donnee = 44;
```

```
premier.precedent = NULL;  
premier.suivant = &deuxieme;  
deuxieme.precedent = &premier;  
deuxieme.suivant = &troisieme;  
troisieme.precedent = &deuxieme;  
troisieme.suivant = NULL;
```

Chaque nœud prend 8 octets de plus.  
Est-ce que ça vaut la peine?

# Les listes doublement chaînées



```
typedef struct Noeud Noeud;  
struct Noeud{  
    Noeud* precedent;  
    int donnee;  
    Noeud* suivant;  
};
```

```
Noeud premier;  
premier.donnee = 10;  
Noeud deuxieme;  
deuxieme.donnee = 20;  
Noeud troisieme;  
troisieme.donnee = 44;
```

```
premier.precedent = NULL;  
premier.suivant = &deuxieme;  
deuxieme.precedent = &premier;  
deuxieme.suivant = &troisieme;  
troisieme.precedent = &deuxieme;  
troisieme.suivant = NULL;
```

Donnez le pseudo-code pour  
retirer tous les éléments ayant la  
donnée 44

# Les forces et les faiblesses de chaque implémentation

**Nous avons vu quelques structures de données...**

Tableaux dynamiques

Tableaux statiques

Listes chaînées

Laquelle choisir si...

On connaît la taille d'avance?

On ne connaît pas la taille d'avance?

On fera surtout des accès en lecture?

On fera beaucoup d'ajouts et des suppressions?

La mémoire est limitée?

# Les forces et les faiblesses de chaque implémentation

Soit une liste de  $n$  éléments. Combien ça prend d'opérations pour accéder au premier élément dans ...

## Un tableau dynamique

```
tabDyn.tab[0]
```

## Une liste chaînée

```
liste.premier-&gtdonnee
```



# Les forces et les faiblesses de chaque implémentation

Soit une liste de  $n$  éléments. Combien ça prend d'opérations pour **imprimer les données** dans ...

## Un tableau dynamique

```
for(i = 0; i < tabDyn.dernier; i++)  
    printf("%d", tabDyn.tab[i]);
```

## Une liste chaînée

```
Noeud* temp = liste.premier;  
while (temp != NULL){  
    printf("%d", temp->donnee);  
    temp = temp->suivant;  
}
```

# Les forces et les faiblesses de chaque implémentation

Soit une liste de  $n$  éléments. Combien ça prend d'opérations pour accéder au 55<sup>e</sup> élément dans ...

## Un tableau dynamique

```
printf("%d", tabDyn.tab[55]);
```

## Une liste chaînée

```
Noeud* temp = liste.premier;  
for (int j = 0; j < 55; j++)  
    temp = temp->suivant;  
printf("%d", temp->donnee);
```

# Les forces et les faiblesses de chaque implémentation

Soit une liste de  $n$  éléments. Combien ça prend d'opérations pour **ajouter un -6 au début** dans ...

## Un tableau dynamique

Si nécessaire, allonger le tableau.  
Déplacer tous les éléments d'une position vers la droite.

```
tabDyn.tab[0] = -6;
```

## Une liste chaînée

```
Noeud* nouveau = malloc(...);  
nouveau->donnee = -6;  
nouveau->suivant = liste.premier;  
if (liste.premier == null)  
    liste.premier = nouveau;
```

# Les forces et les faiblesses de chaque implémentation

Soit une liste de  $n$  éléments. Combien ça prend d'opérations pour **ajouter un -6 à la 55<sup>e</sup> case** dans ...

## Un tableau dynamique

Si nécessaire, allonger le tableau.  
Déplacer tous les éléments d'une position vers la droite à partir du 55<sup>e</sup>.  
`tabDyn.tab[54] = -6;`

## Une liste chaînée

Avancer à la position 54.  
`Noeud* nouveau = malloc(...);`  
`nouveau->donnee = -6;`  
`nouveau->suivant = temp->suivant;`  
`temp->suivant = nouveau;`

# Les forces et les faiblesses de chaque implémentation

## **Un tableau dynamique**

Meilleur pour

Accéder à une position arbitraire.

## **Une liste chaînée**

Meilleur pour

Ajouter et retirer des éléments.



# Programmation algorithmique

Leçon 4  
Les tris

# Les opérations sur les listes

Un liste doit faire quoi pour nous être utile?

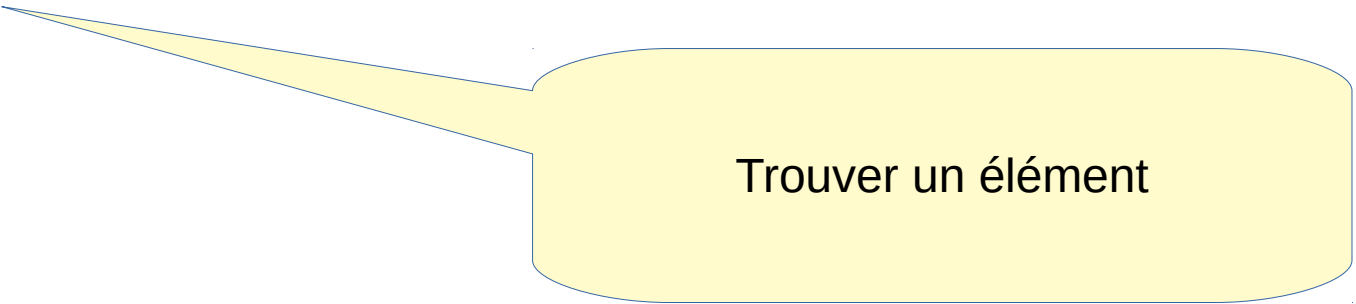
Sélection

Parcours

Insertion

Suppression

Tri



Trouver un élément

# Les opérations sur les listes

Un liste doit faire quoi pour nous être utile?

Sélection

Parcours

Insertion

Suppression

Tri

Effectuer une action pour chaque élément.



# Les opérations sur les listes

Un liste doit faire quoi pour nous être utile?

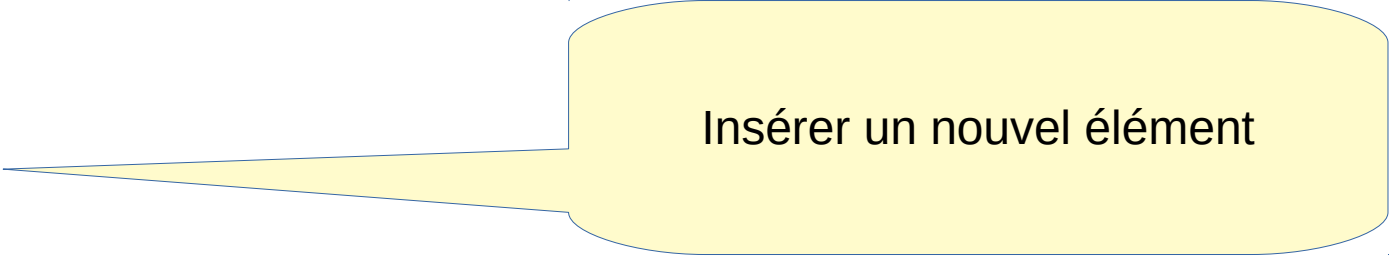
Sélection

Parcours

Insertion

Suppression

Tri



Insérer un nouvel élément

# Les opérations sur les listes

Un liste doit faire quoi pour nous être utile?

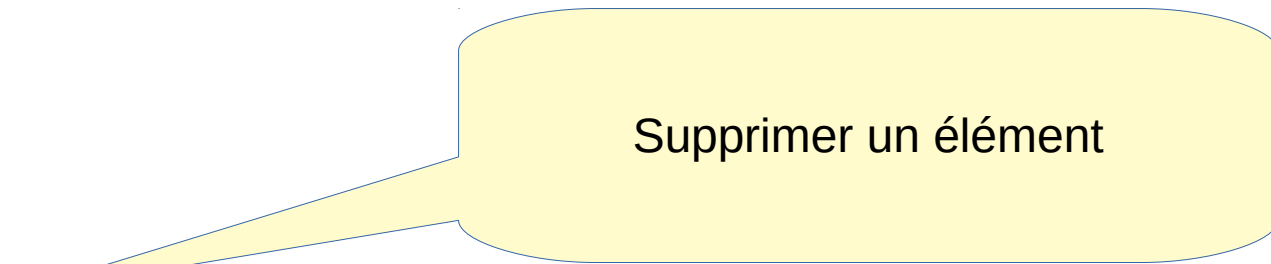
Sélection

Parcours

Insertion

Suppression

Tri



Supprimer un élément

# Les opérations sur les listes

Un liste doit faire quoi pour nous être utile?

Sélection

Parcours

Insertion

Suppression

Tri

Trier/ordonner les éléments selon leur valeur.

# Les opérations sur les listes

Un liste doit faire quoi pour nous être utile?

Sélection

Parcours

Insertion

Suppression

Tri

D'autres opérations possibles :  
Concaténation de listes, renversement, dénombrement, etc.

# Le tri

À quoi ça sert de trier des éléments?

Fouiller rapidement

Trouver les doublons

# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

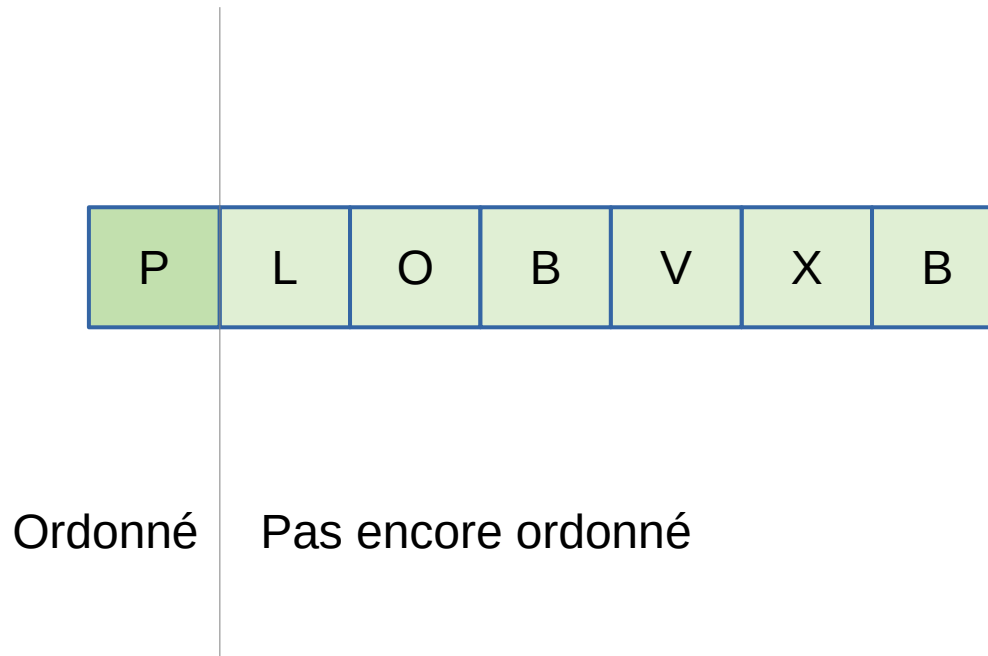
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.

P	L	O	B	V	X	B
---	---	---	---	---	---	---

# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

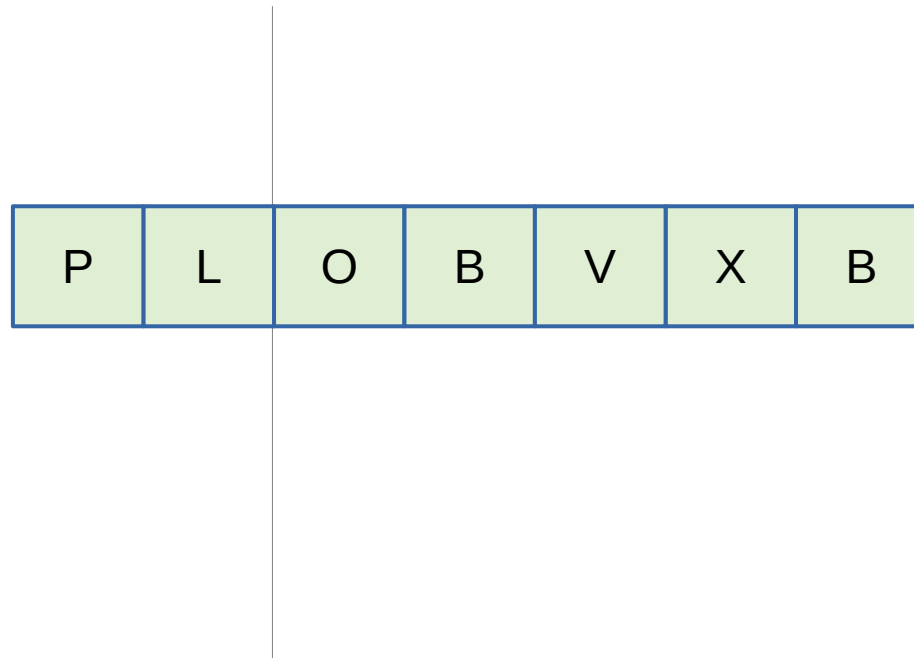
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.

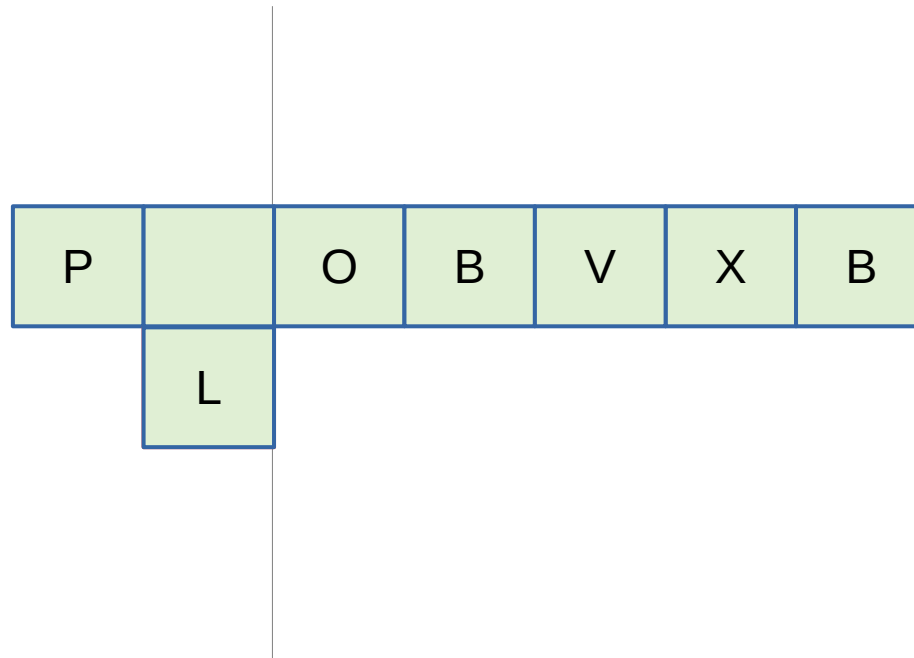




# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

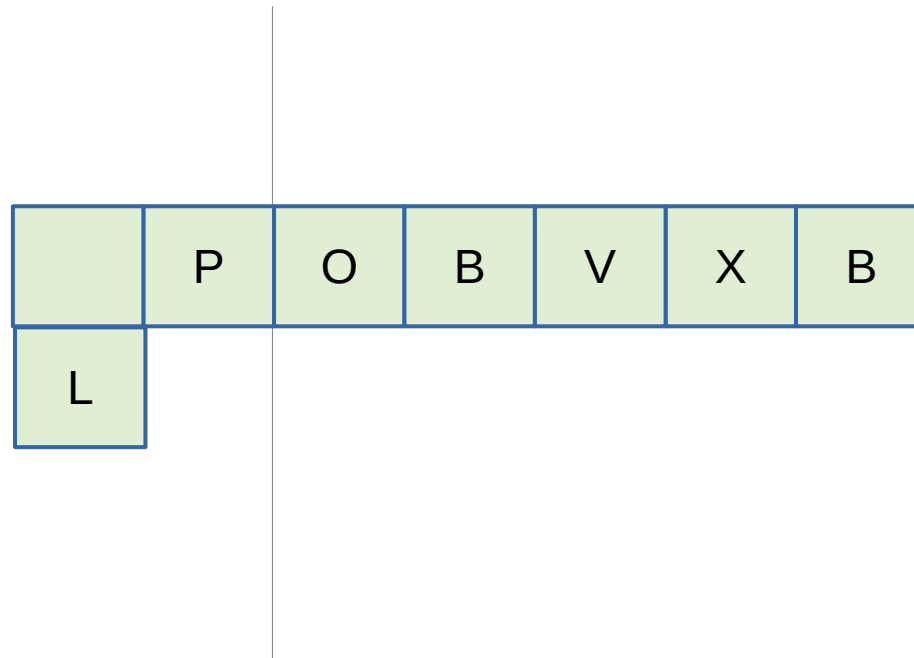
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

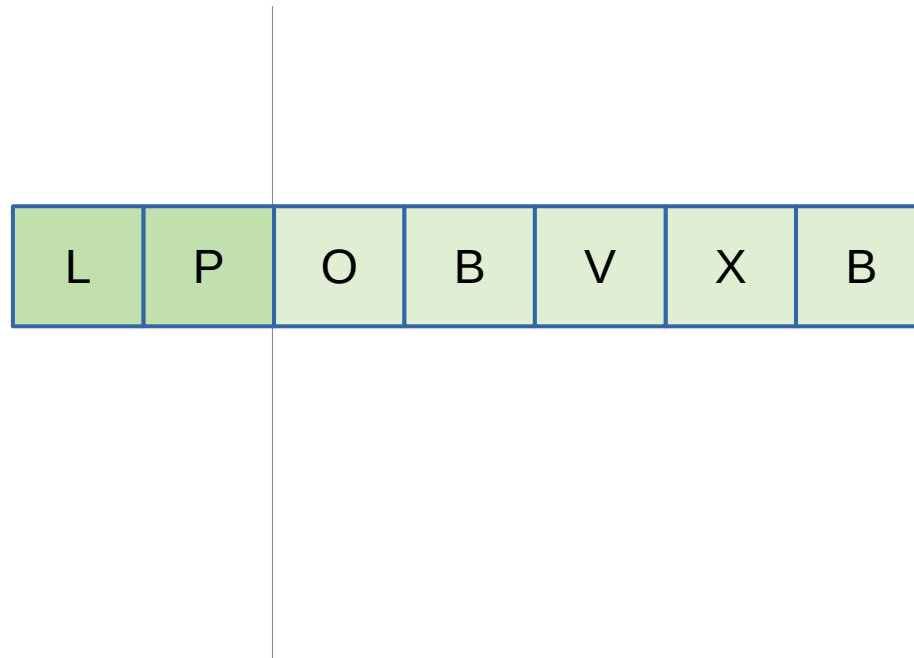
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

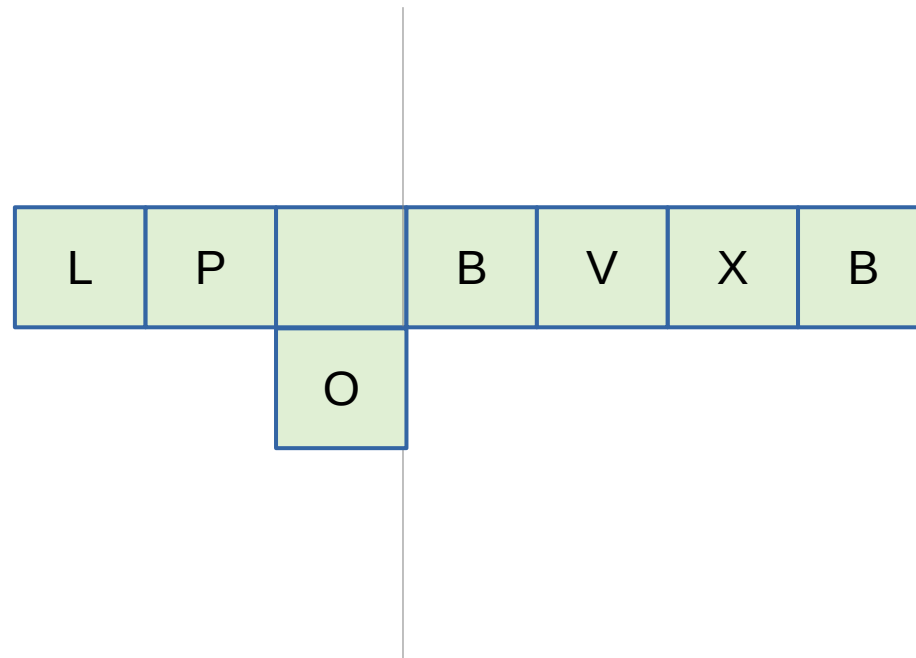
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

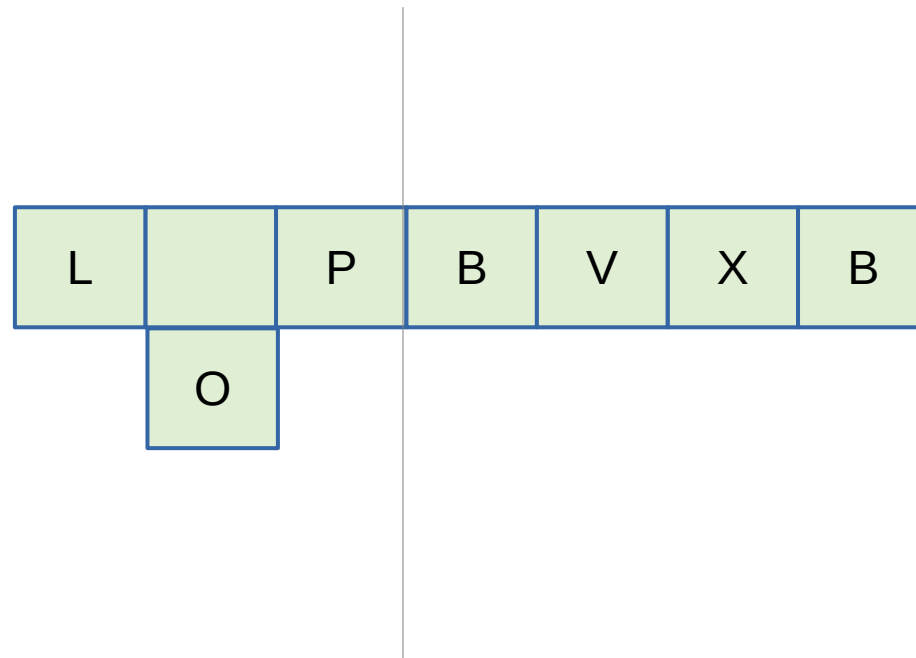
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

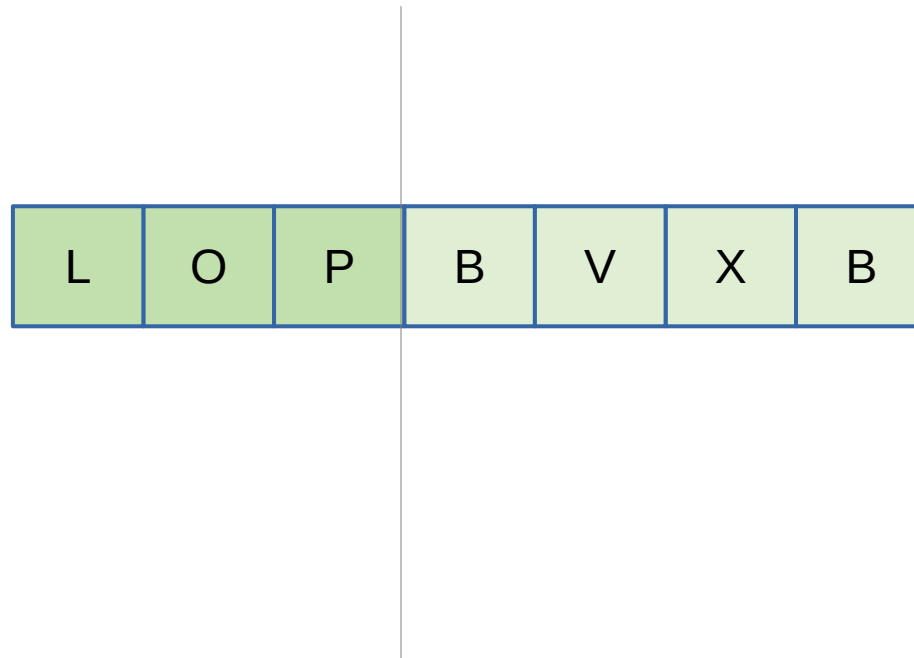
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

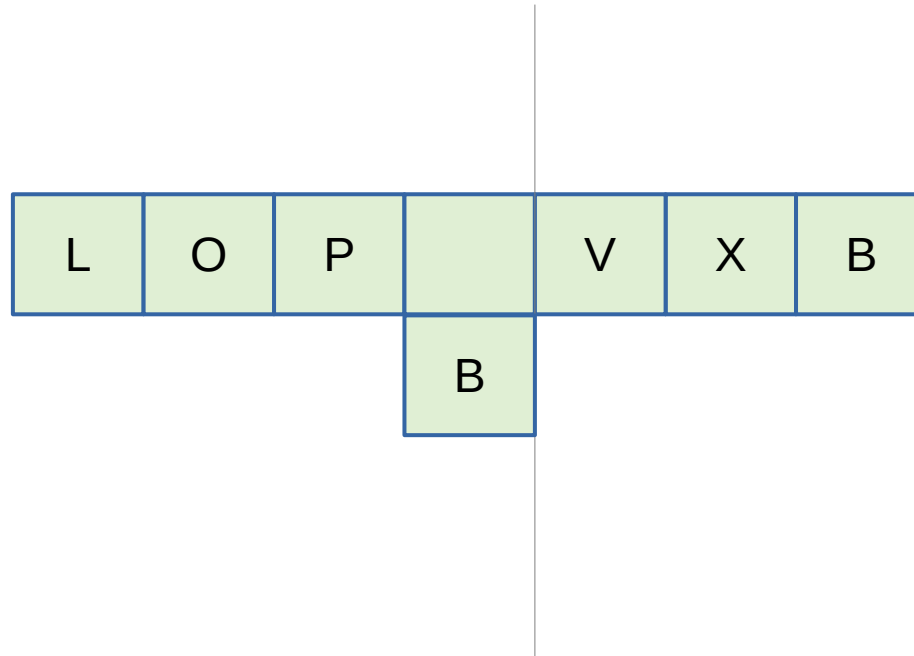
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

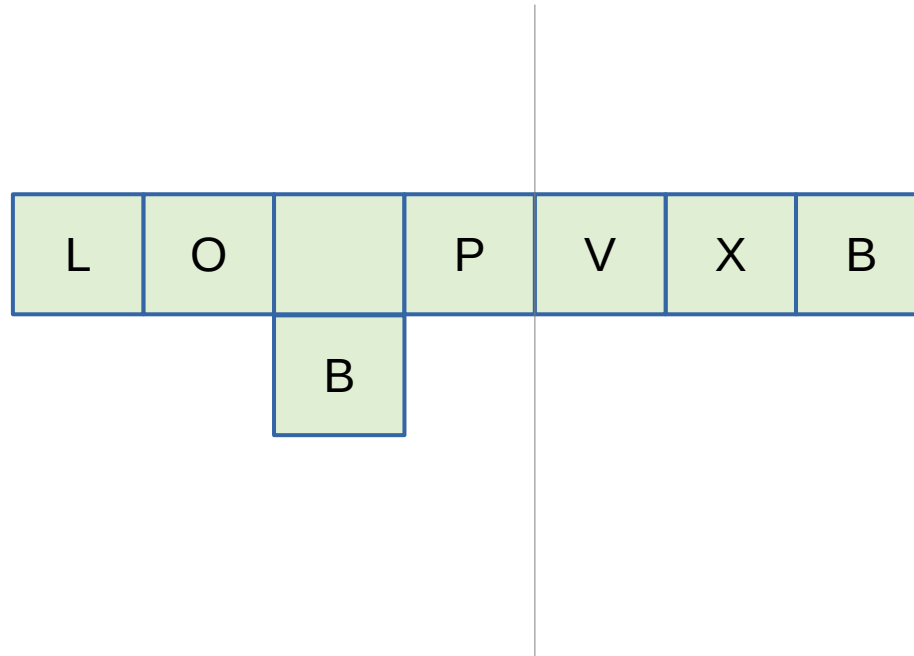
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.

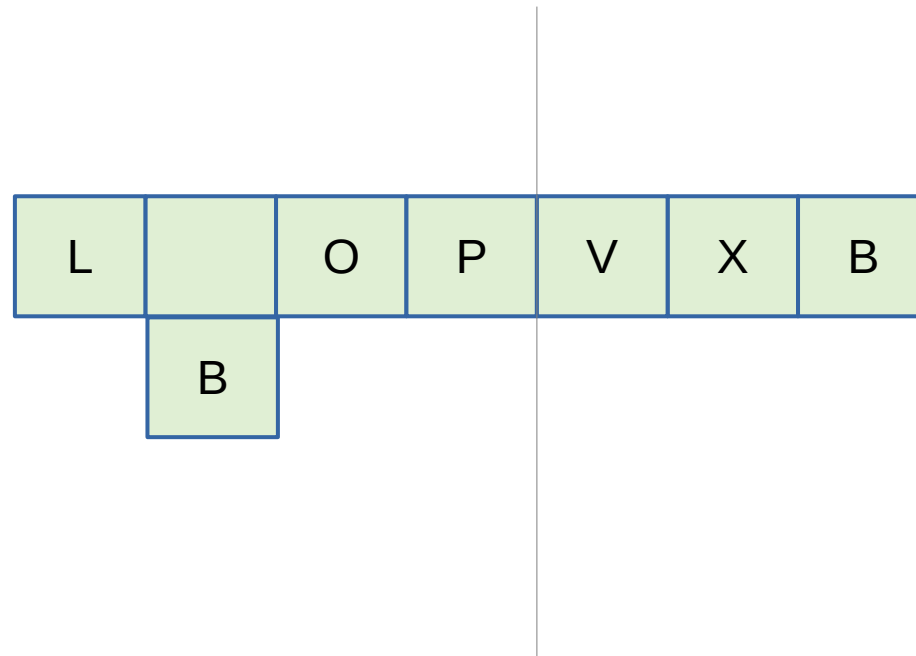




# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

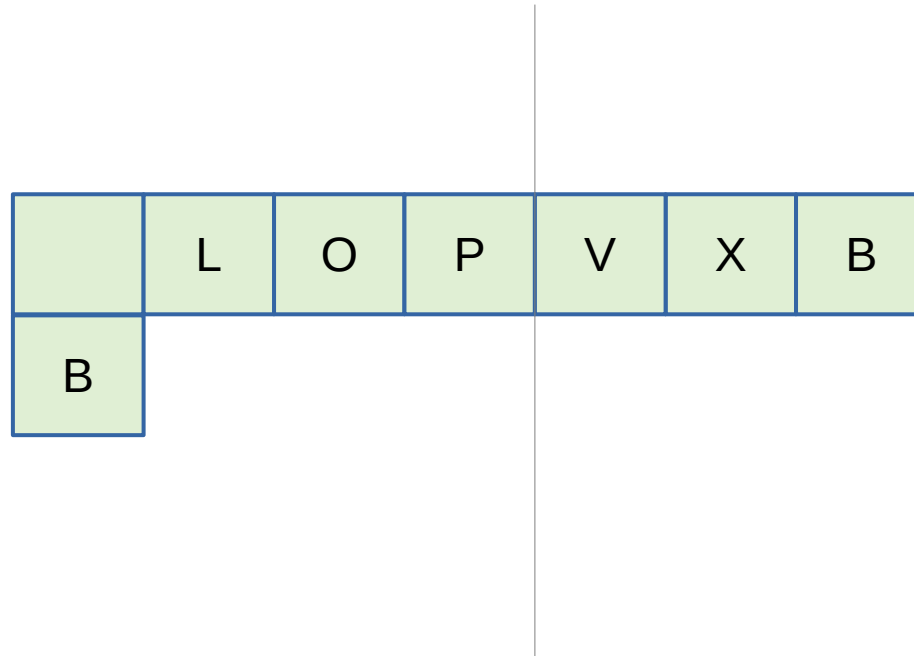
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

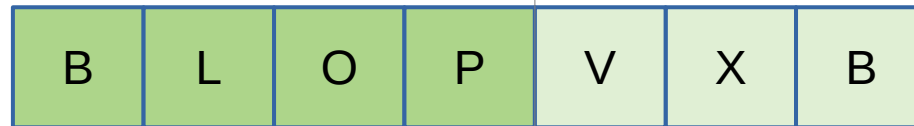
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

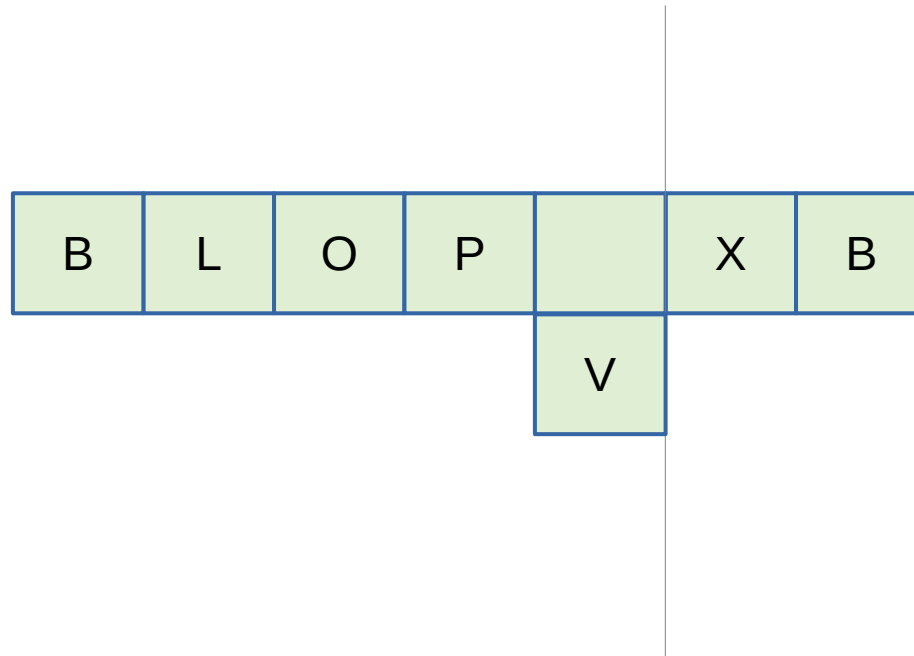
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

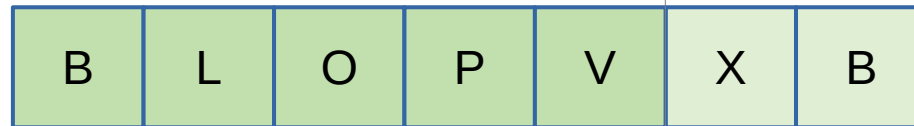
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

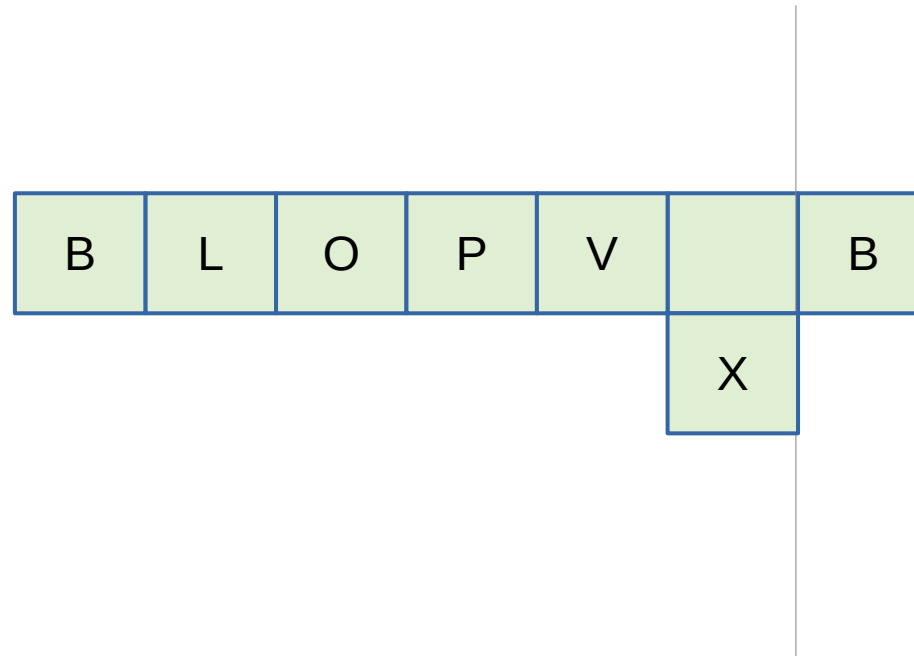
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

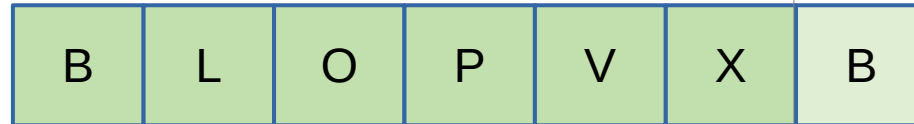
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

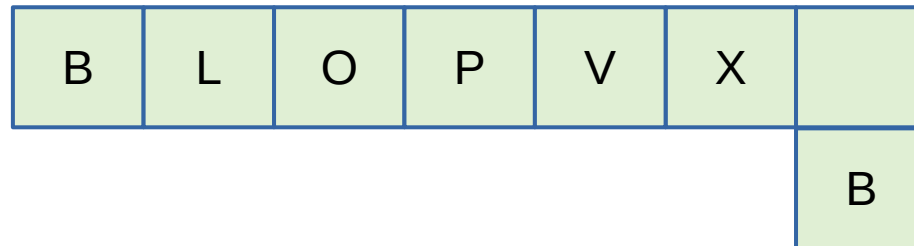
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.

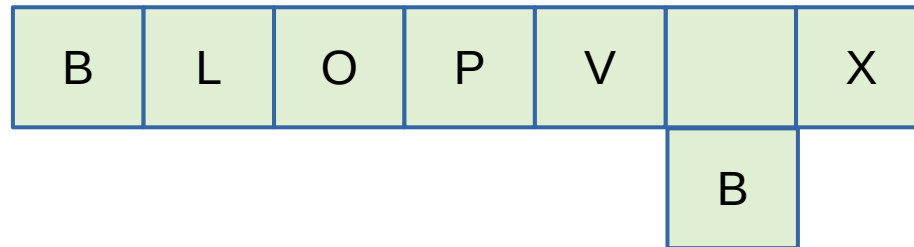




# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

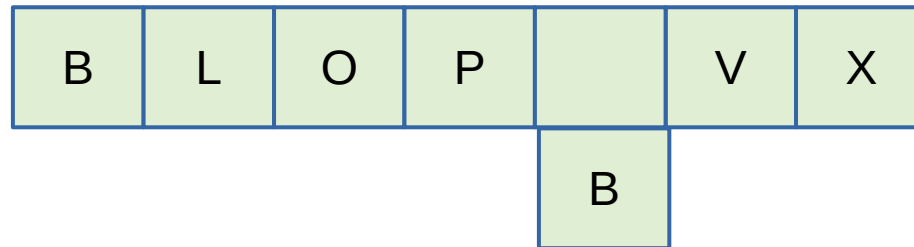
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

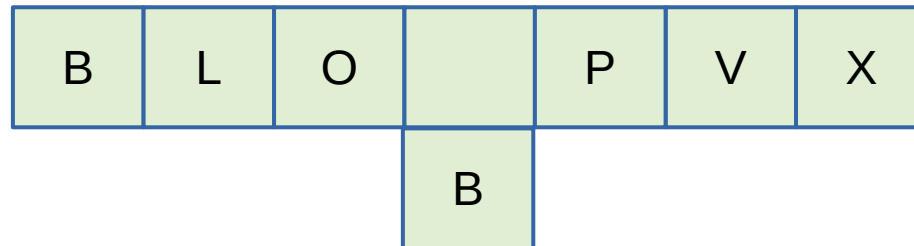
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

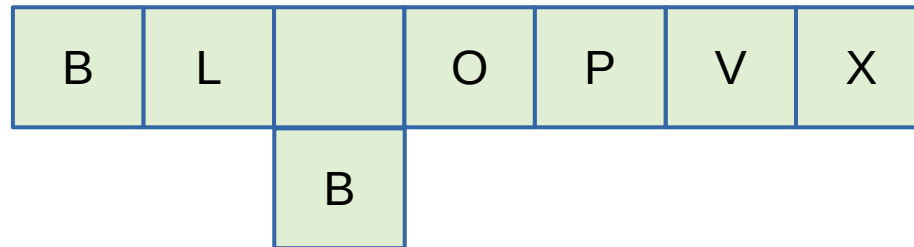
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

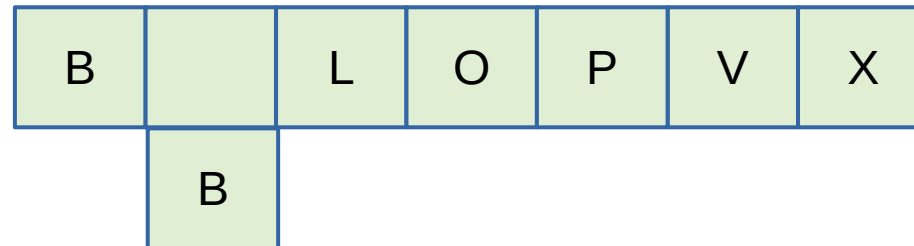
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

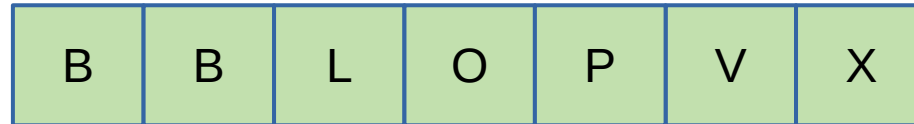
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

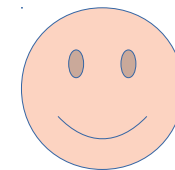
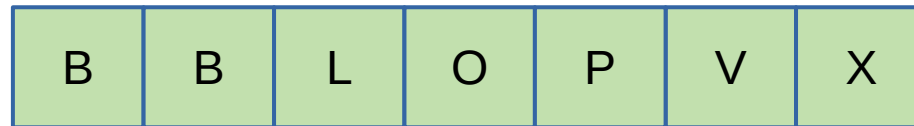
On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

Un croupier distribue des cartes.

On ramasse une carte à la fois et on l'insère au bon endroit dans sa main.



# Un premier tri - Le tri par insertion

## Pseudo-code

Entrée : T un tableau de longueur n

Sortie : T trié en ordre croissant

**i** ← 1

**Tant que** i < n, **Faire**

**j** ← i

**Tant que** j > 0 et que T[j - 1] > T[j], **Faire**  
        échanger T[j] et T[j - 1]

        j ← j - 1

    i ← i + 1

Efficace sur les petits tableaux  
Efficace pour les tableaux  
presque en ordre.  
« En ligne »

$O(n^2)$



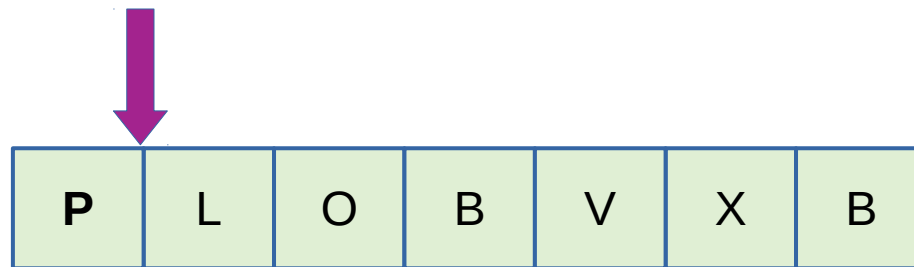
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...

P	L	O	B	V	X	B
---	---	---	---	---	---	---

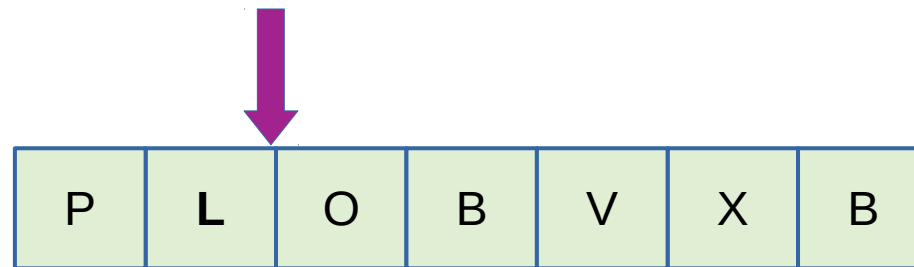
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



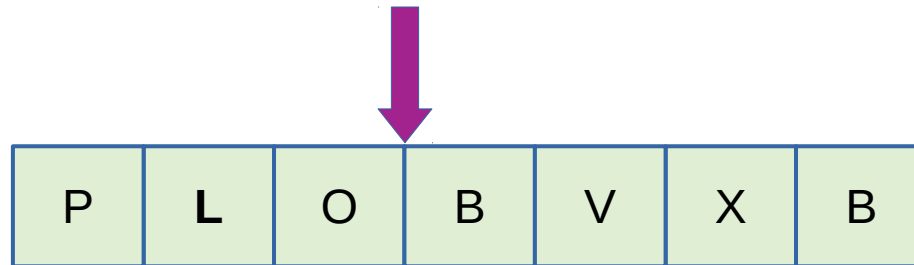
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



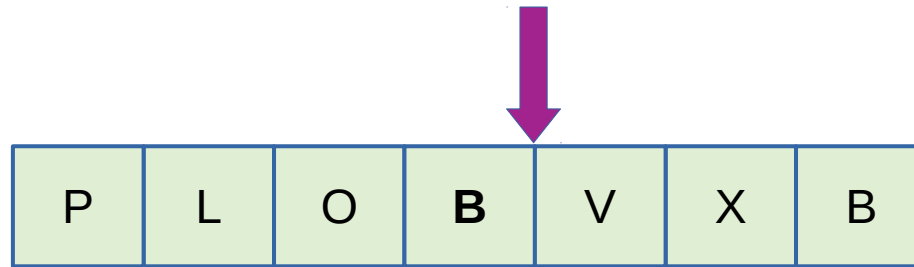
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



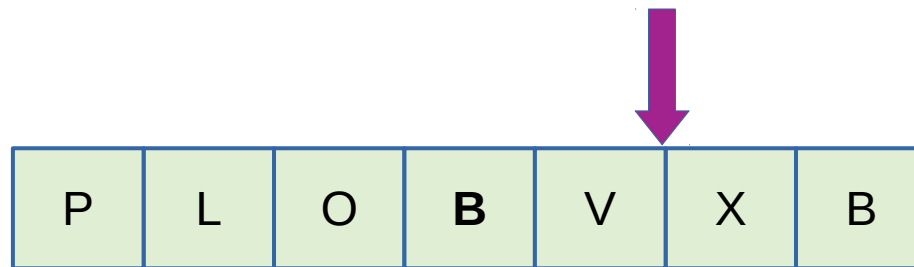
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



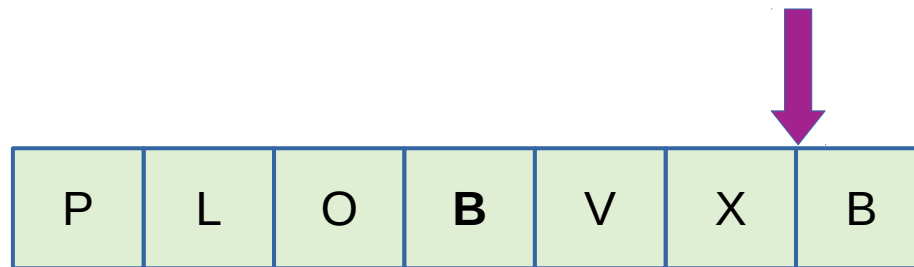
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



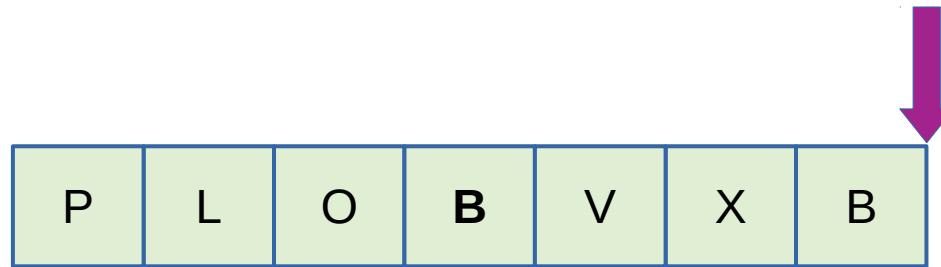
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...





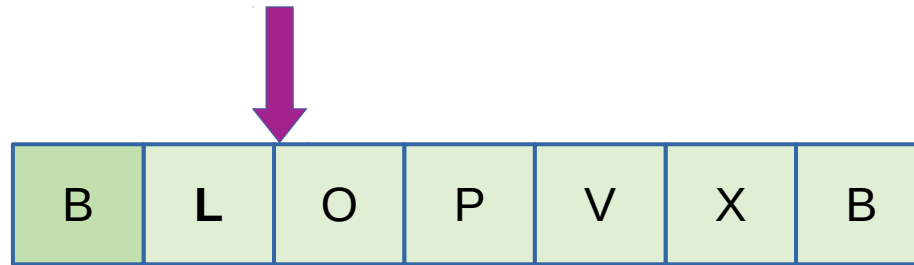
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...

B	L	O	P	V	X	B
---	---	---	---	---	---	---

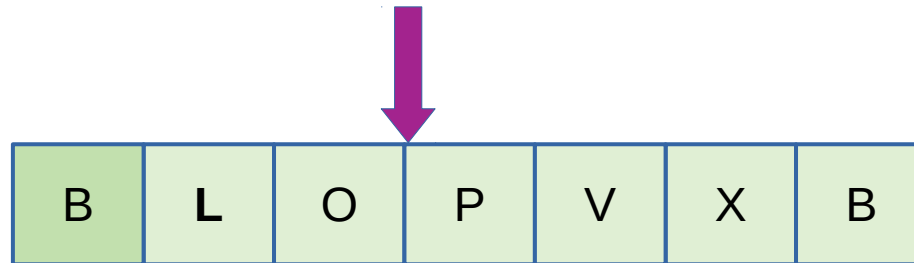
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



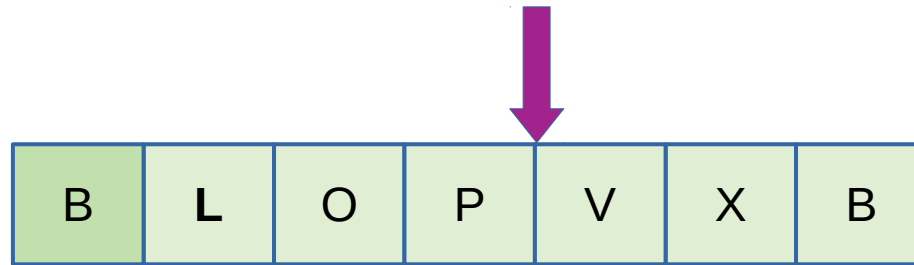
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



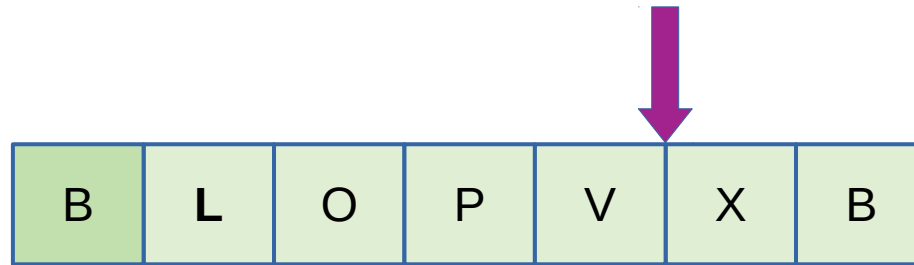
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



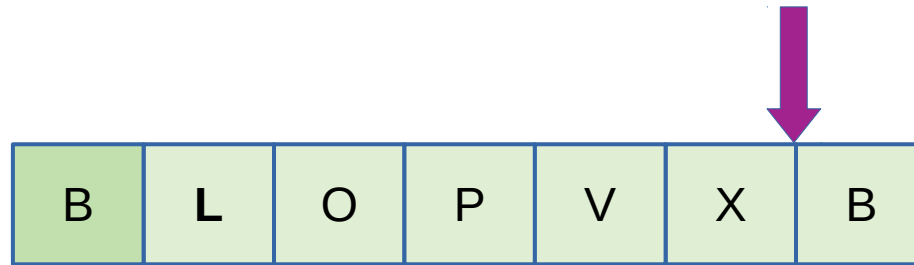
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



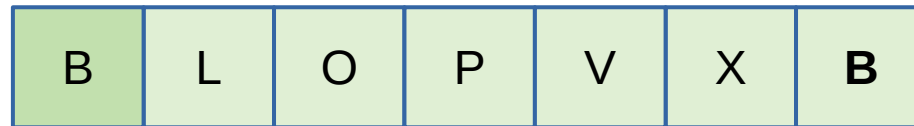
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



# Un autre tri - Le tri par sélection

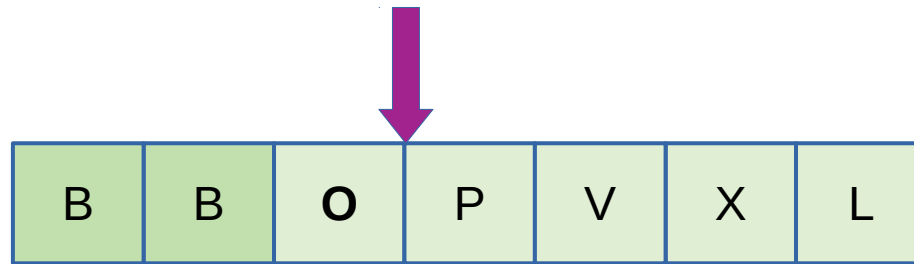
On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...

B	B	O	P	V	X	L
---	---	---	---	---	---	---



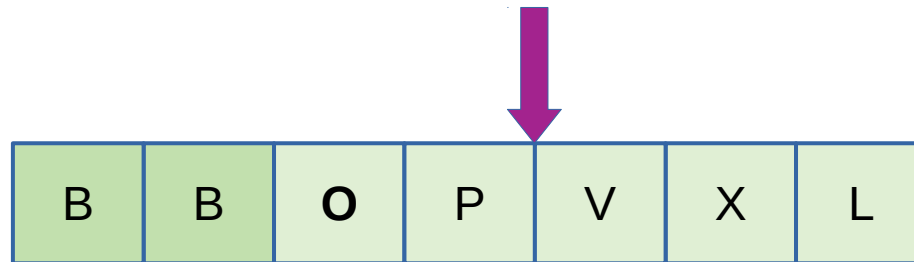
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



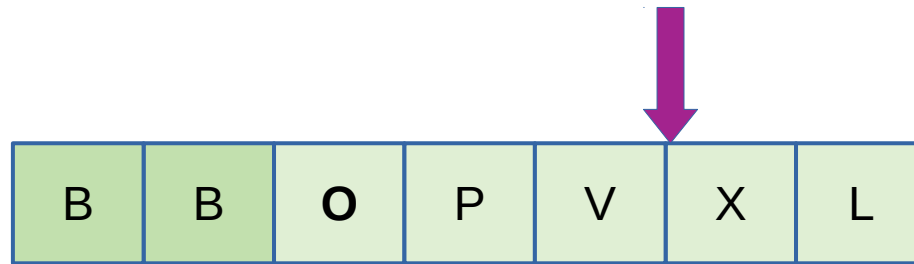
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



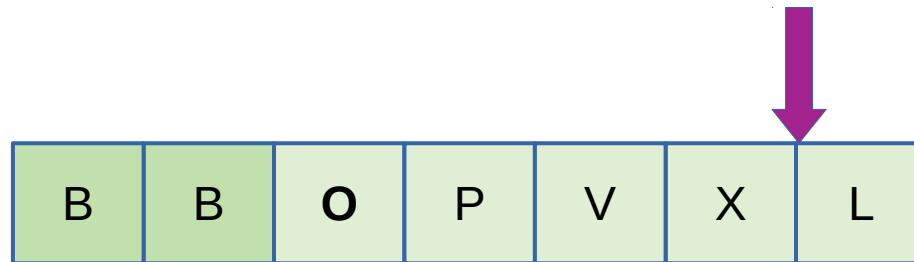
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



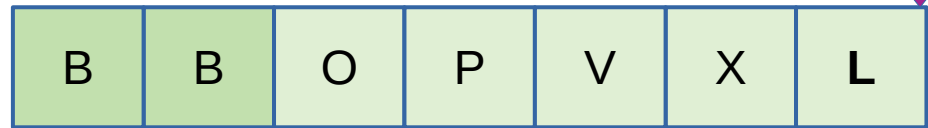
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



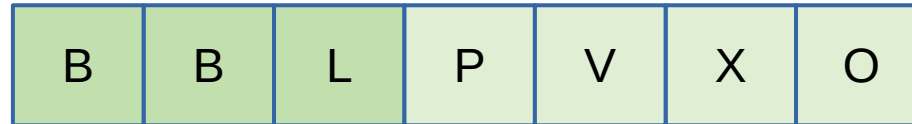
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



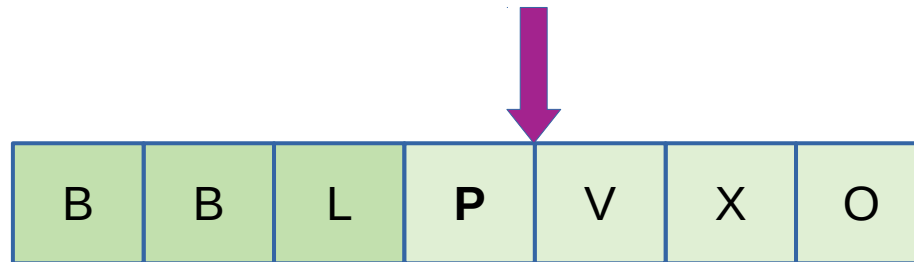
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



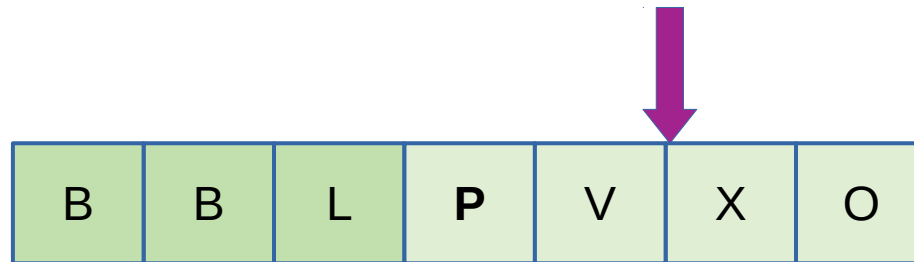
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



# Un autre tri - Le tri par sélection

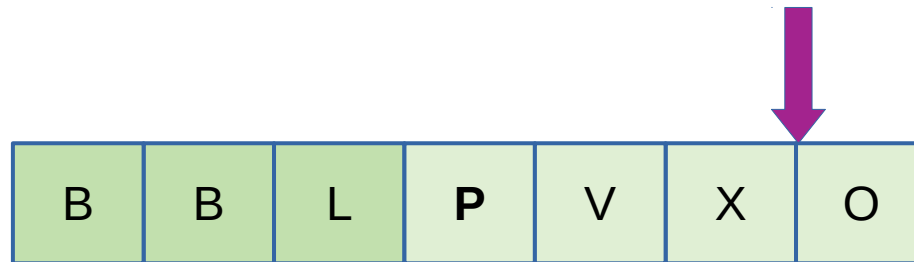
On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...





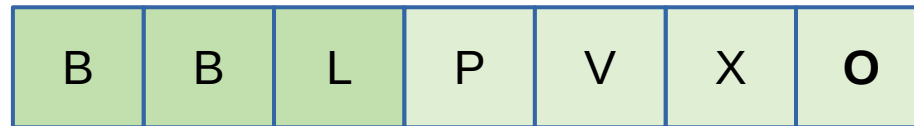
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



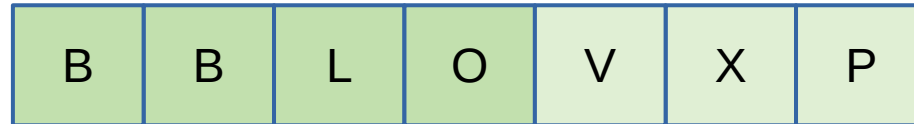
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



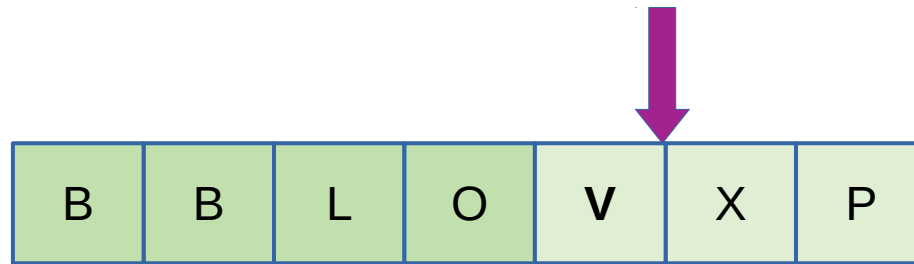
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



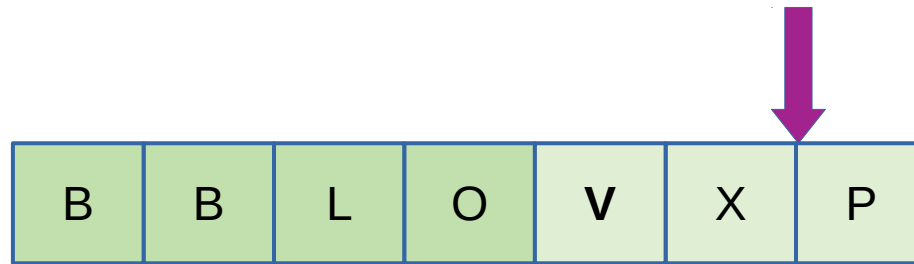
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



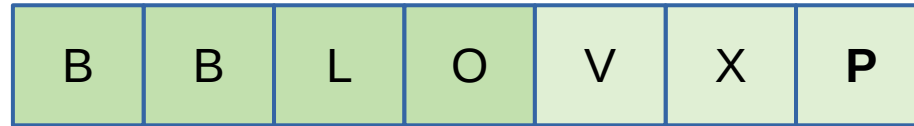
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



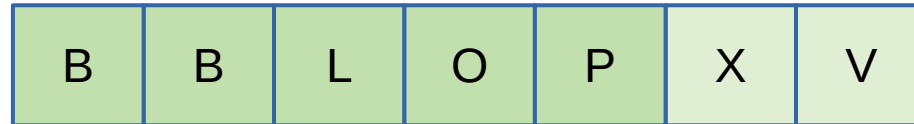
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



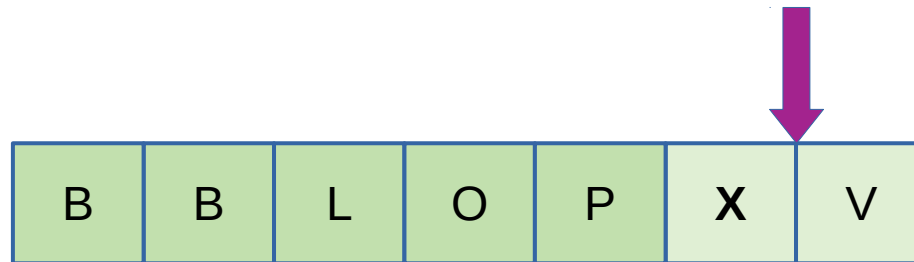
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



# Un autre tri - Le tri par sélection

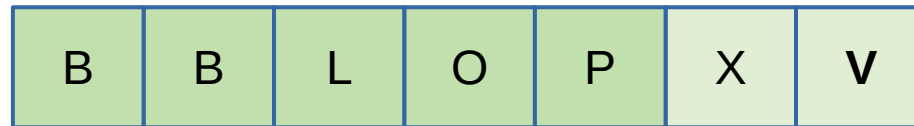
On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...





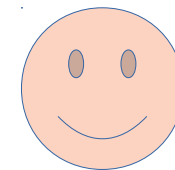
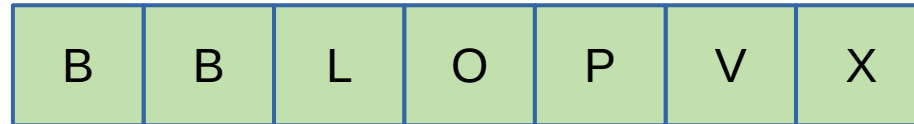
# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



# Un autre tri - Le tri par sélection

On trouve le plus petit,  
puis le deuxième plus petit,  
et ainsi de suite...



# Un autre tri - Le tri par sélection

## Pseudo-code

Entrée :  $T$  un tableau de longueur  $n$

Sortie :  $T$  trié en ordre croissant

$i \leftarrow 0$

Tant que  $i < n - 1$ , Faire

$\text{min} \leftarrow i$

$j \leftarrow i + 1$

    Tant que  $j < n$ , Faire

        Si  $T[\text{min}] > T[j]$

$\text{min} = j$

$j \leftarrow j + 1$

    échanger  $T[i]$  et  $T[\text{min}]$

$i \leftarrow i + 1$

Pas beaucoup d'échanges  
Pratique sur les listes chaînées

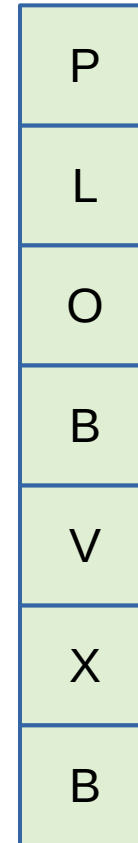
$O(n^2)$

# Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

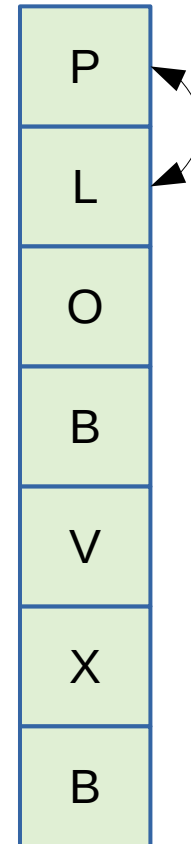
La plupart d'entre vous avez choisi une  
variante de cet algorithme au TP3  
(Yahtzee)



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

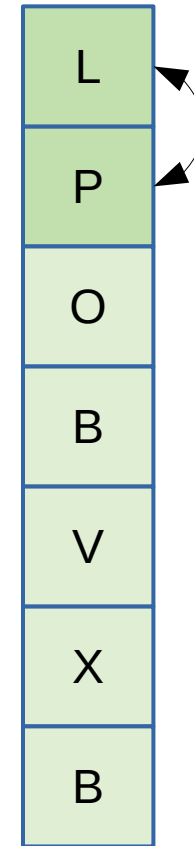
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

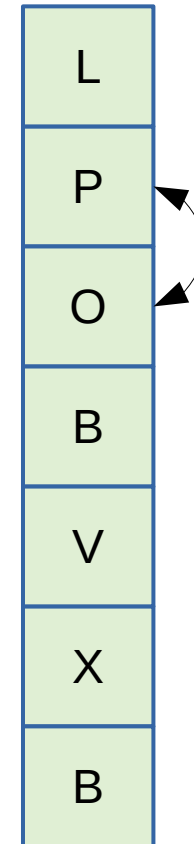
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

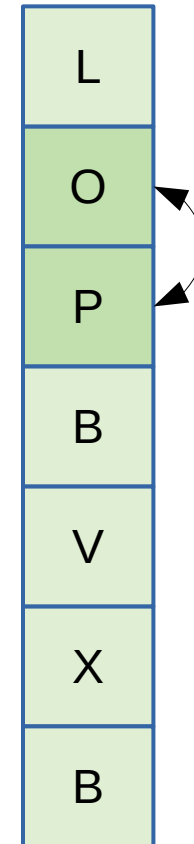
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

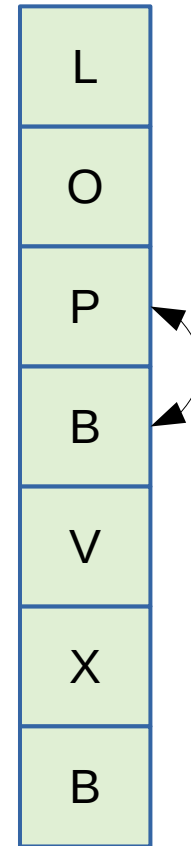




## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

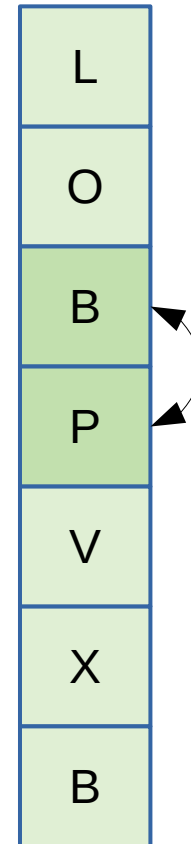
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

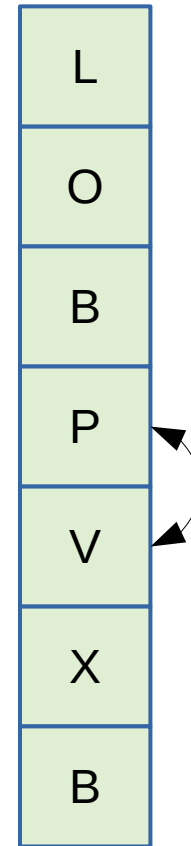
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

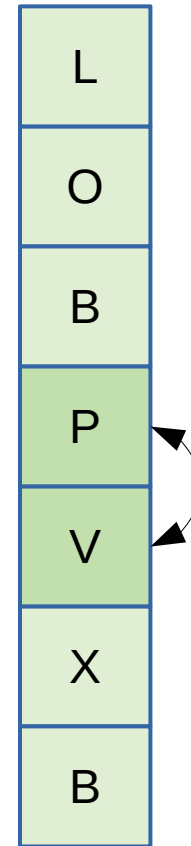
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

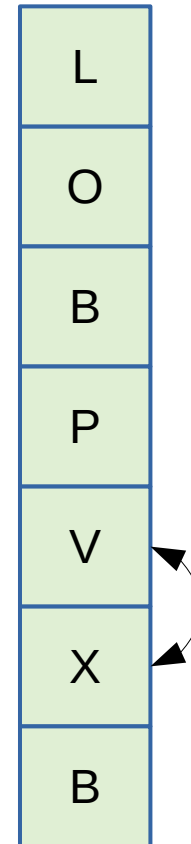
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

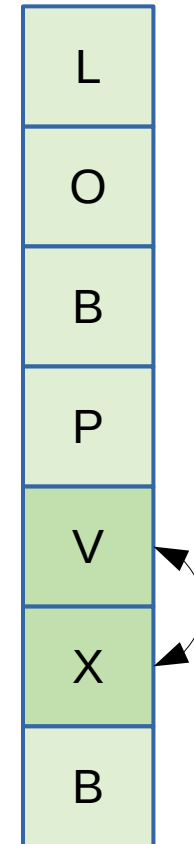
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

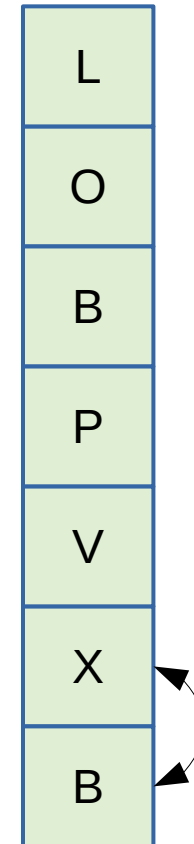
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

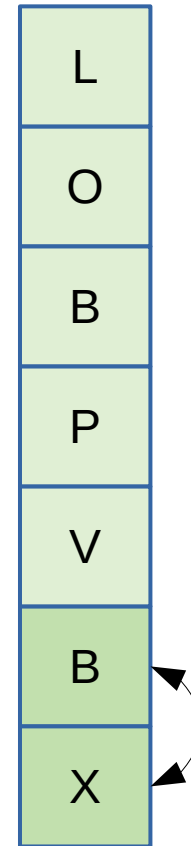
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.



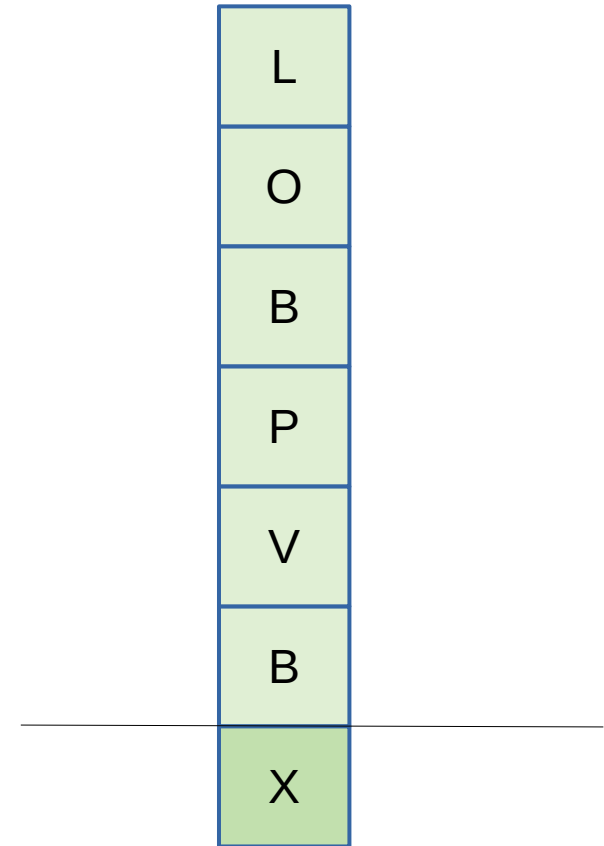


## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

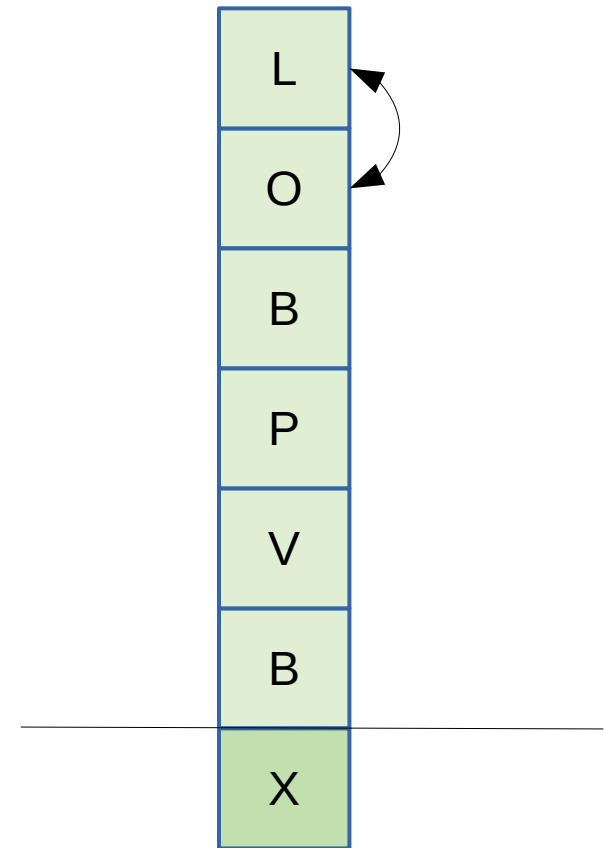
Fin du premier tour



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

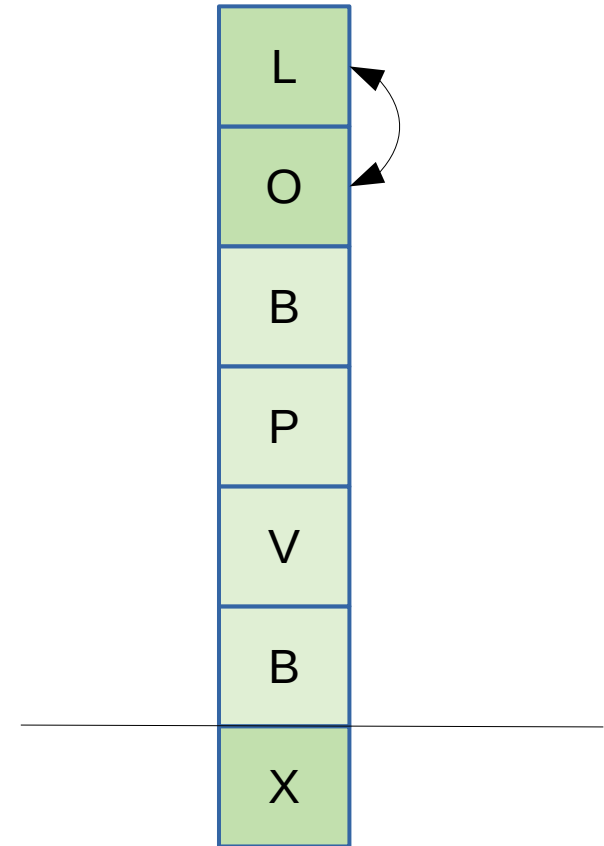
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

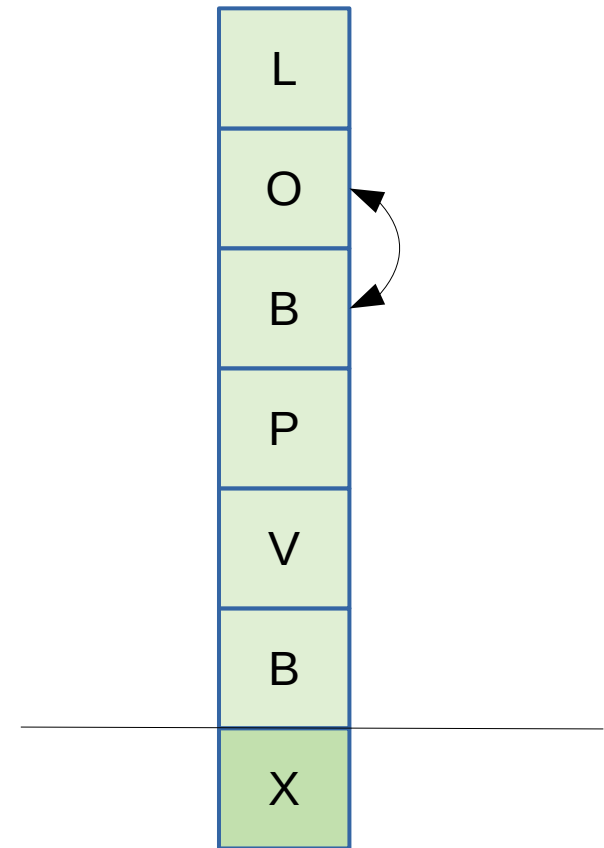
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

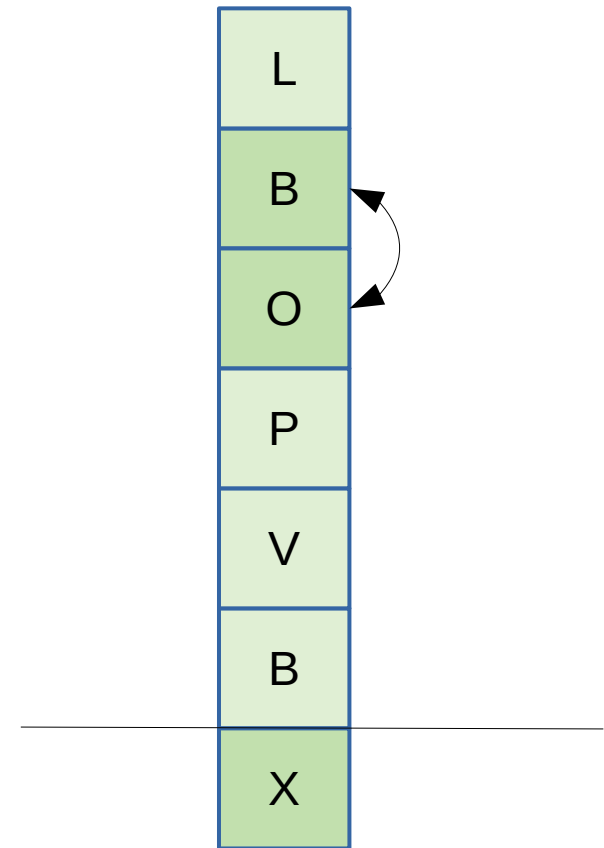
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

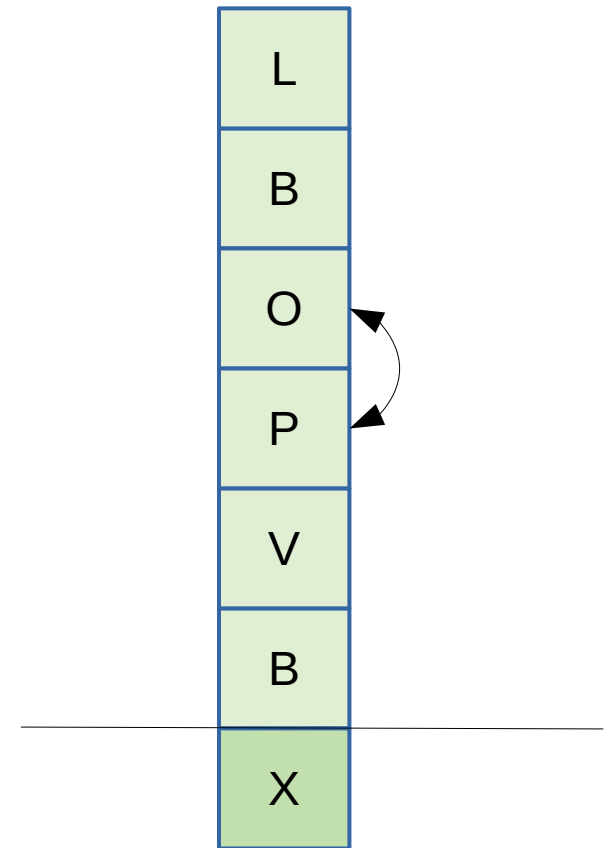
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

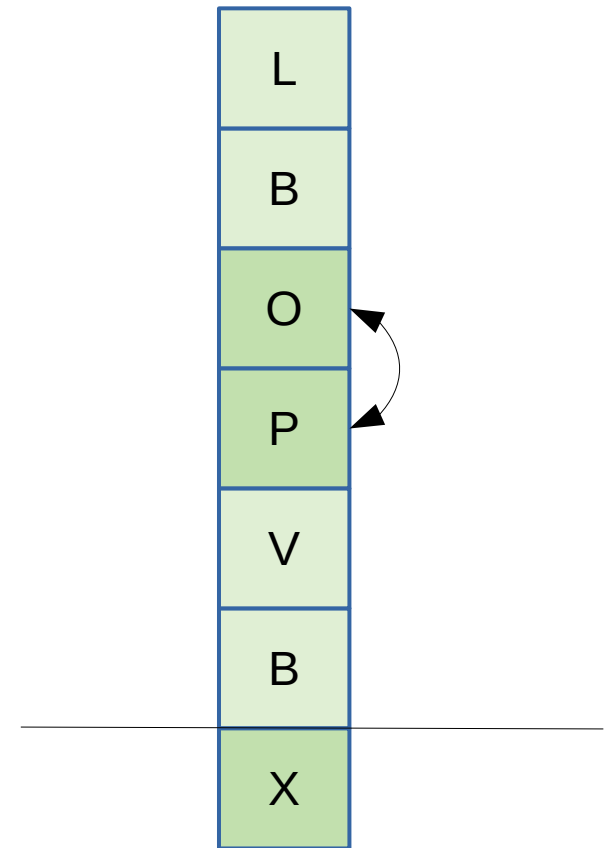
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

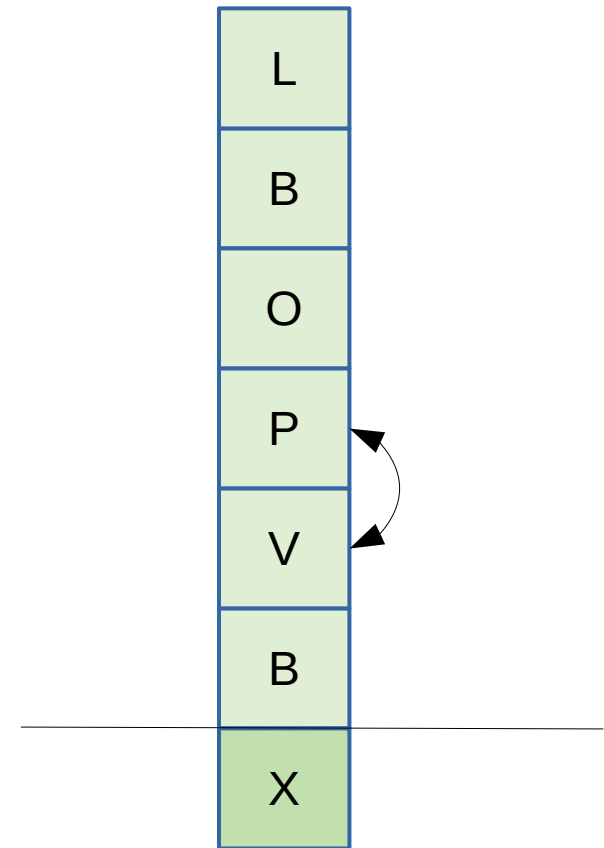
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

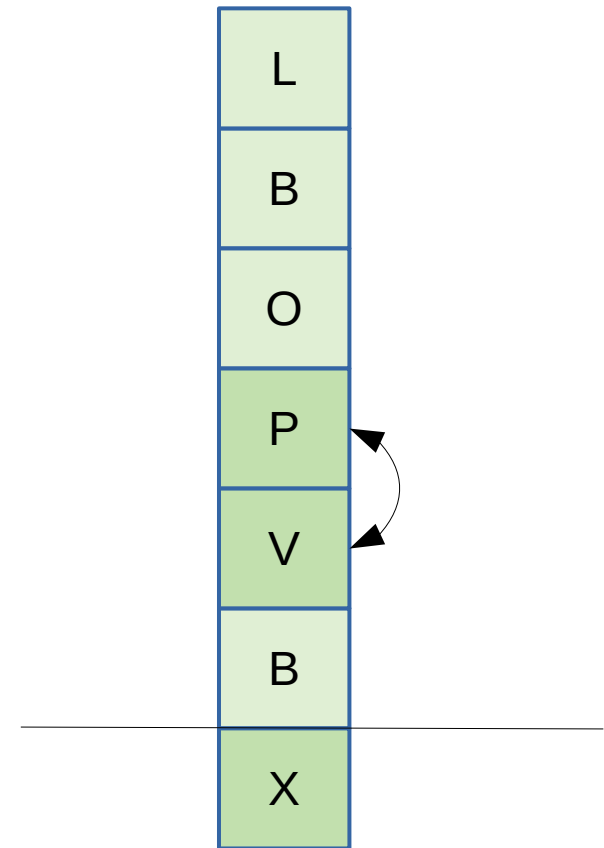




## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

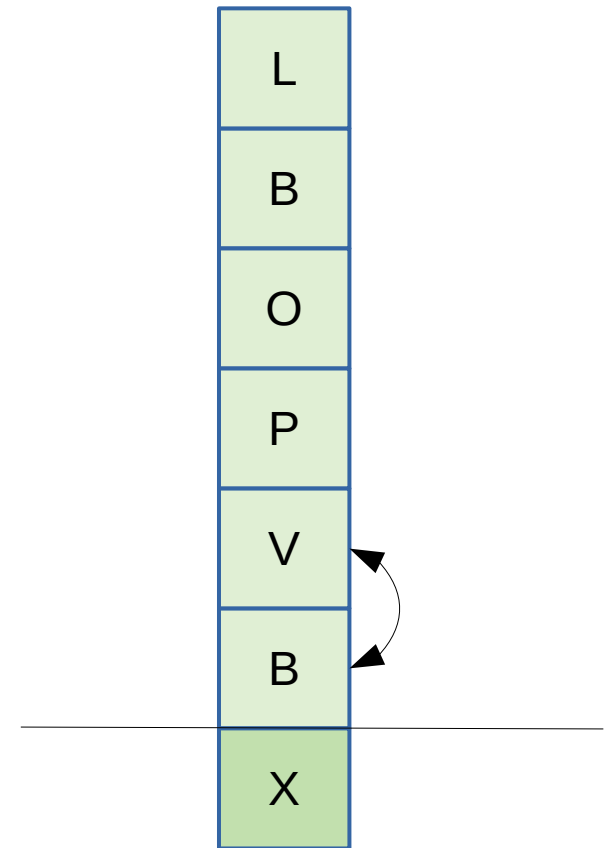
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

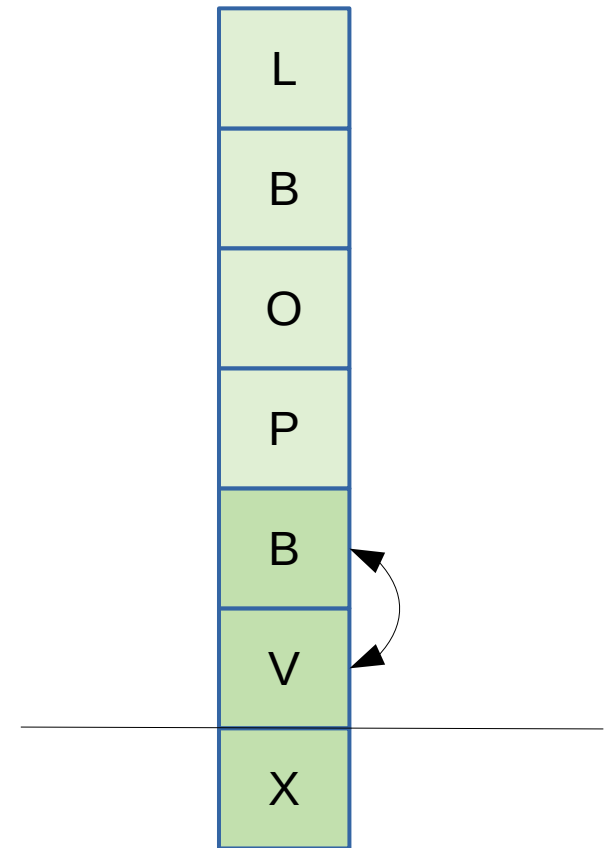
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

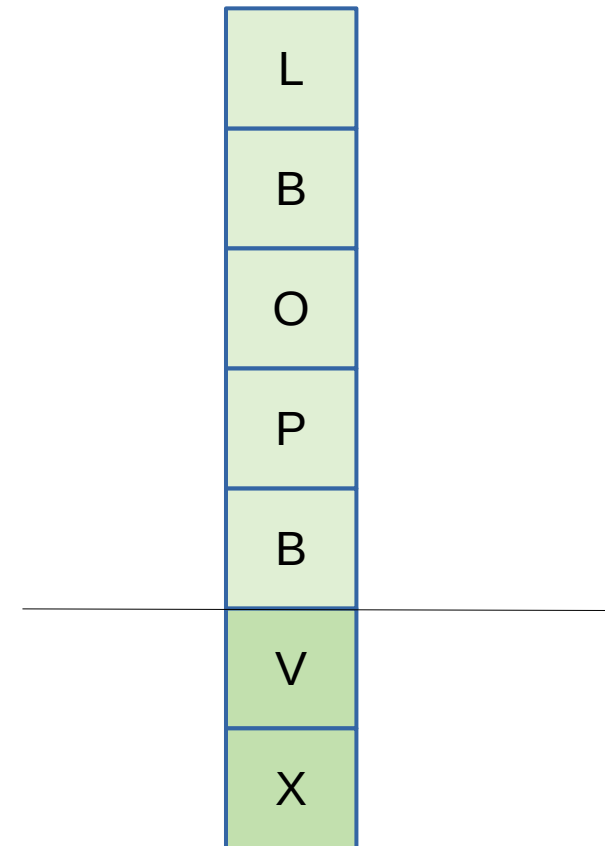


## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

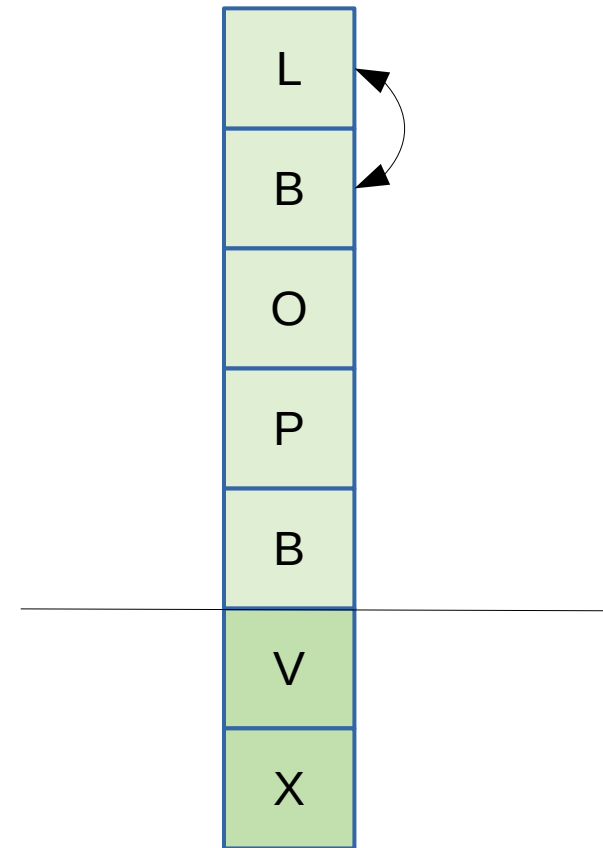
Fin du deuxième tour.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

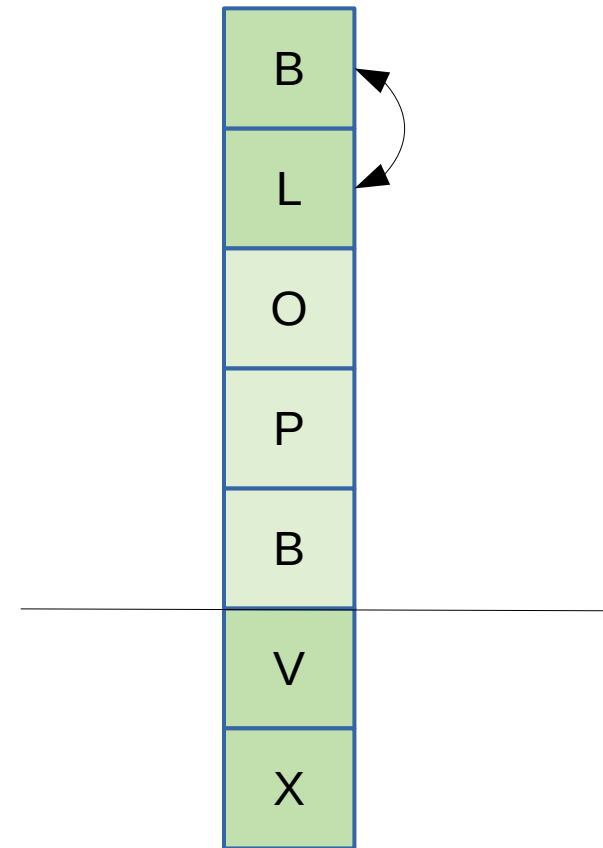
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

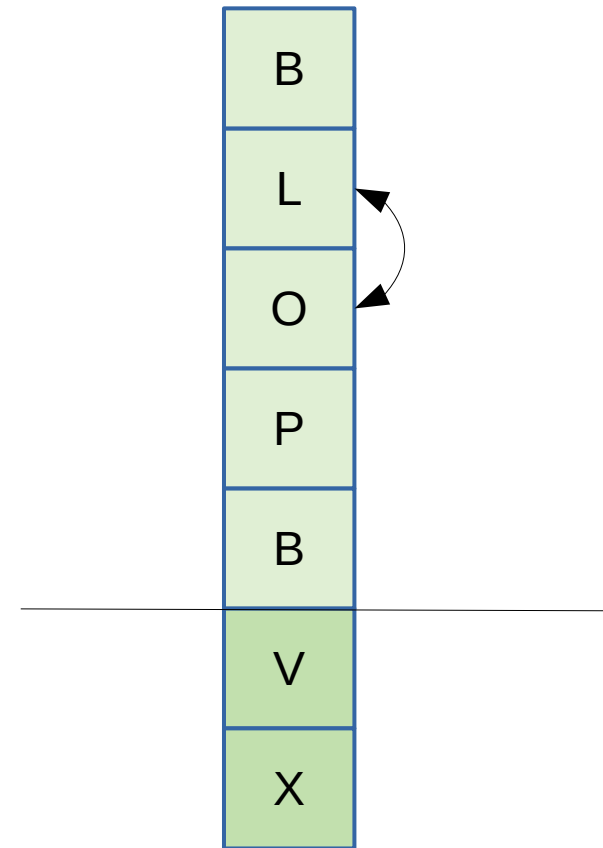
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

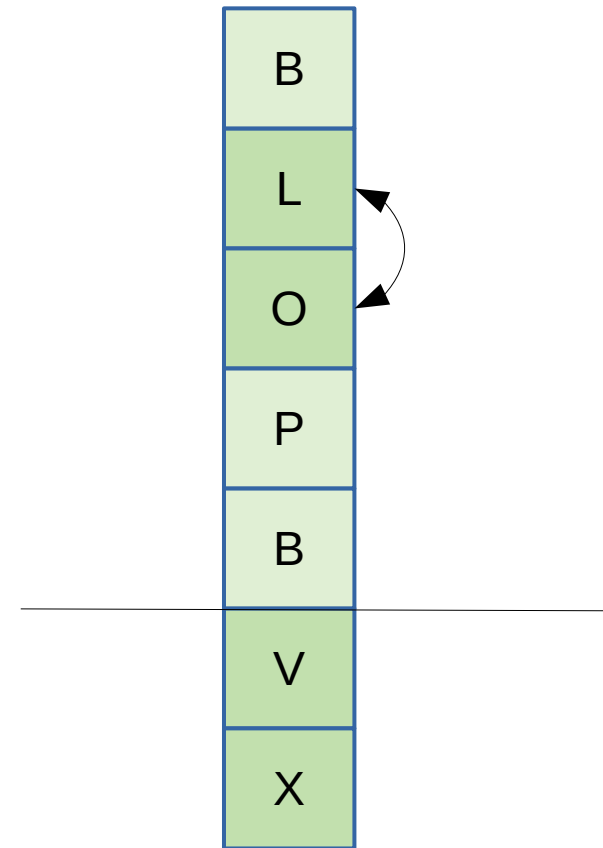
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

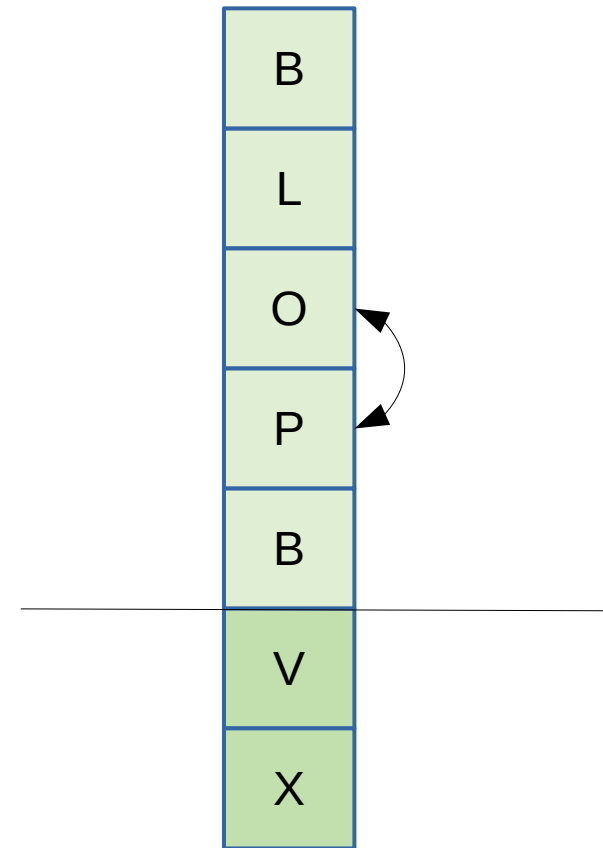




## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

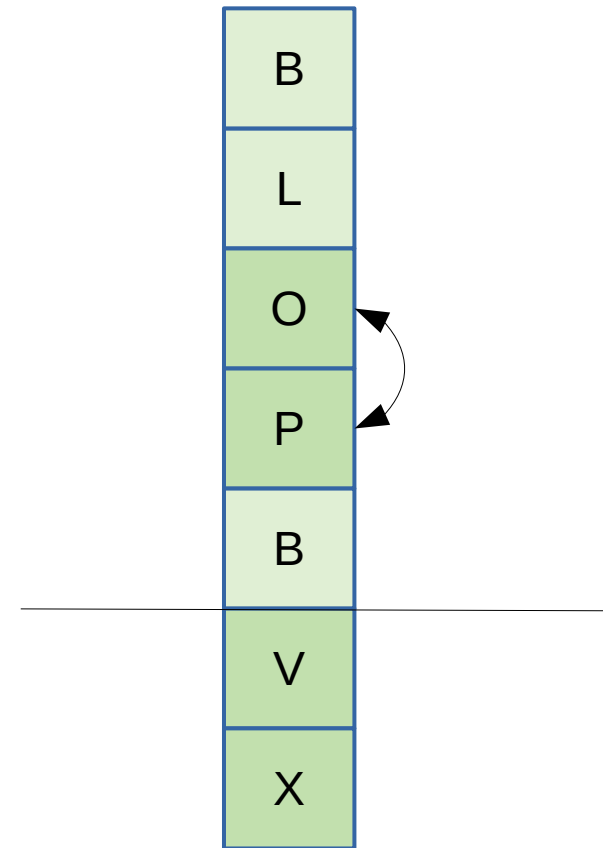
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

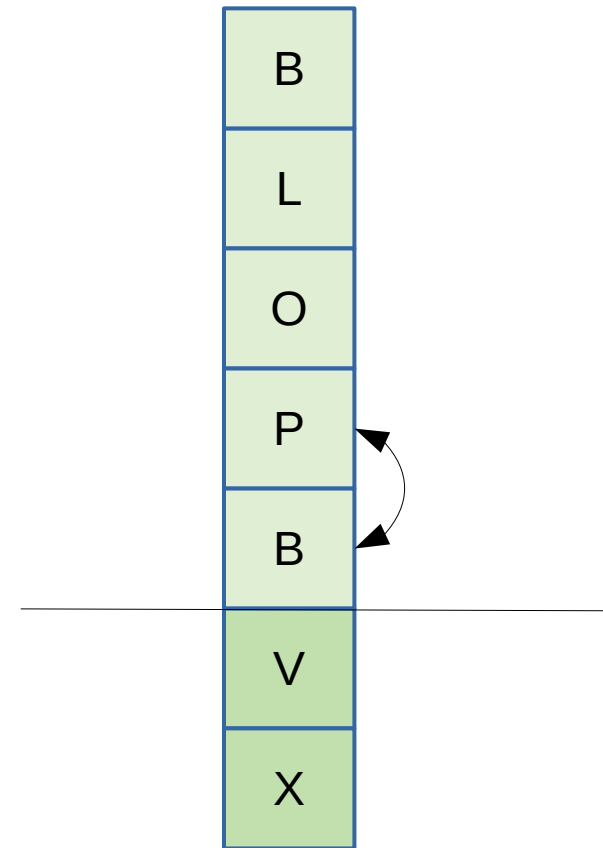
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

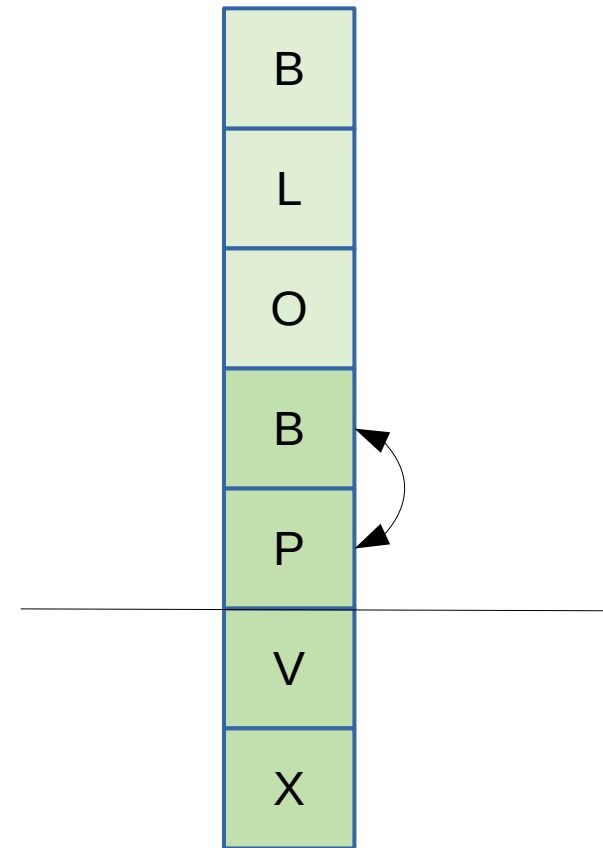
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

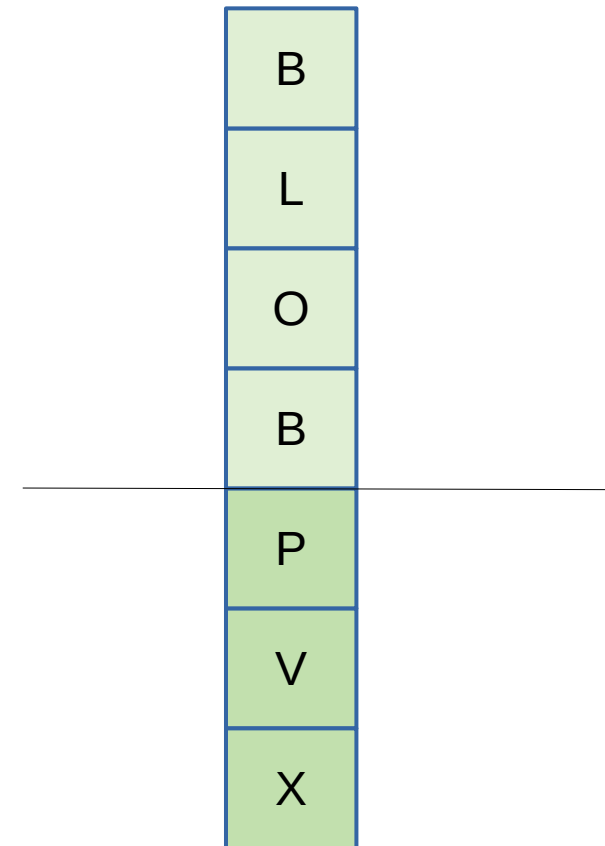


## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

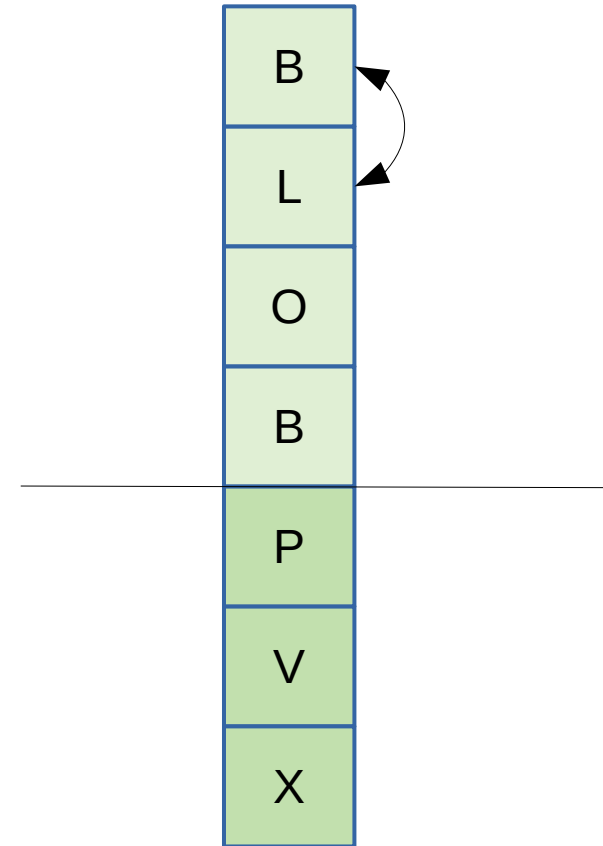
Fin du troisième tour



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

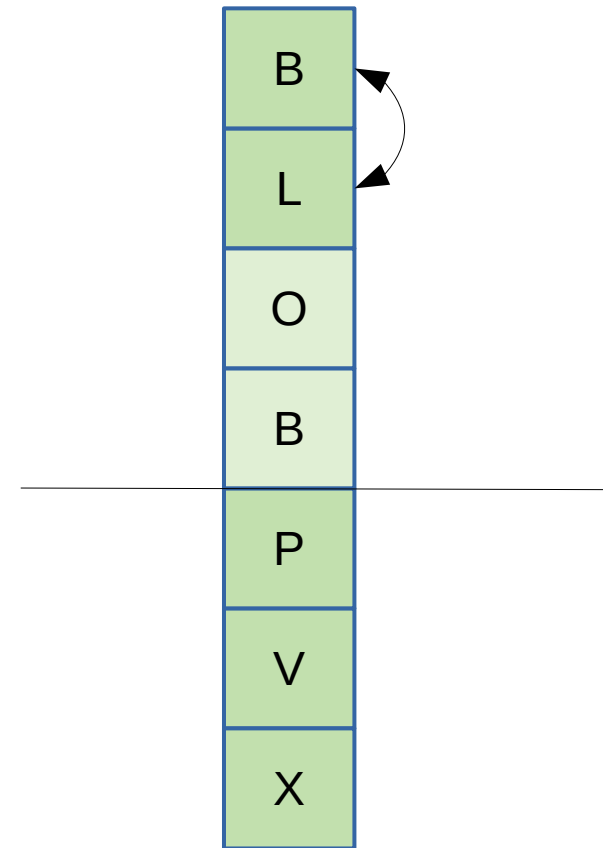
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

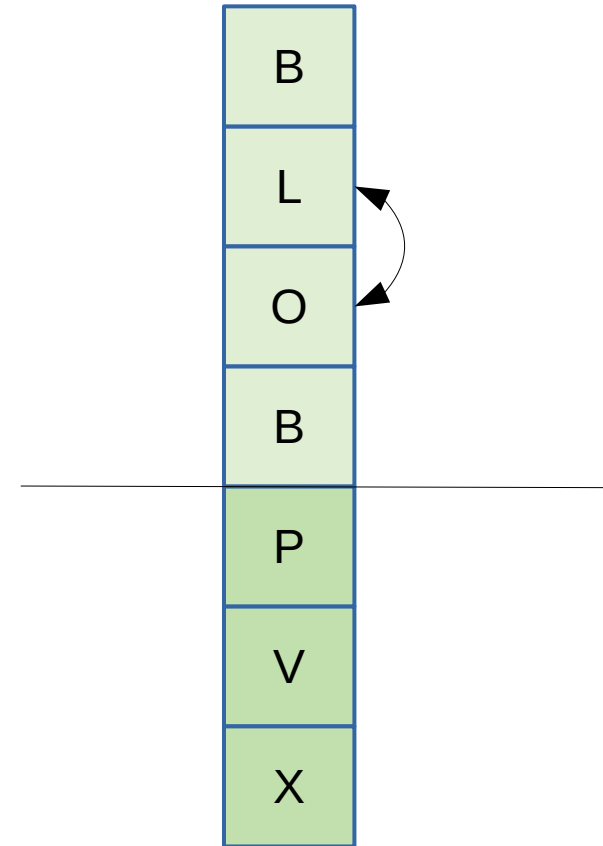
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

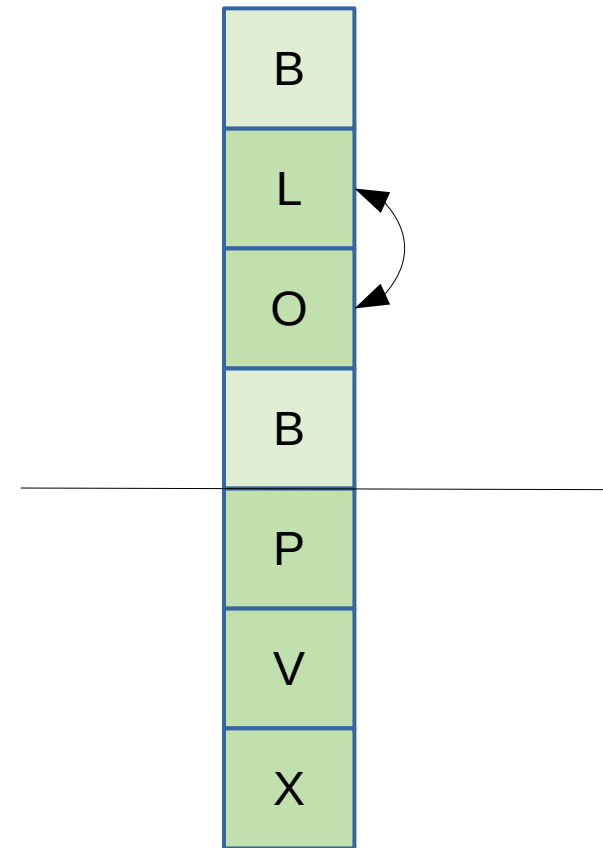




## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

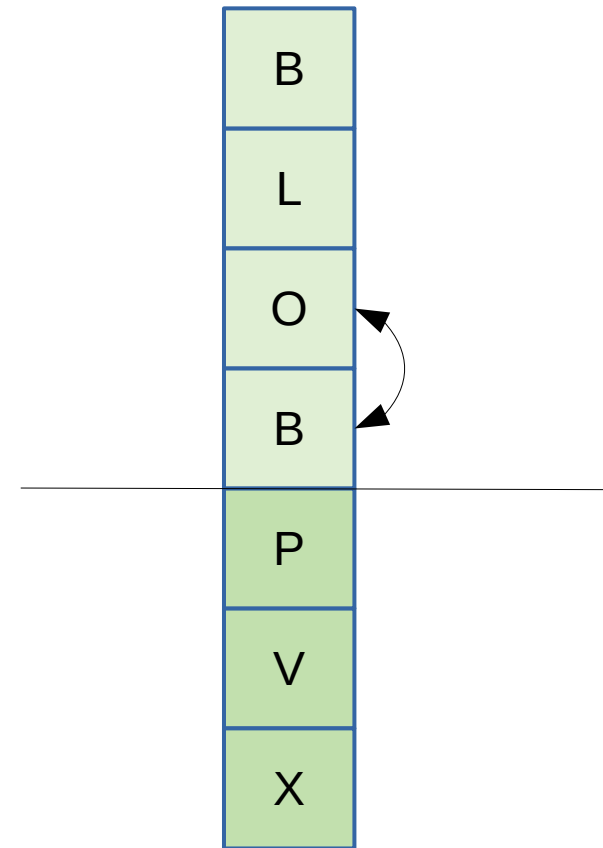
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

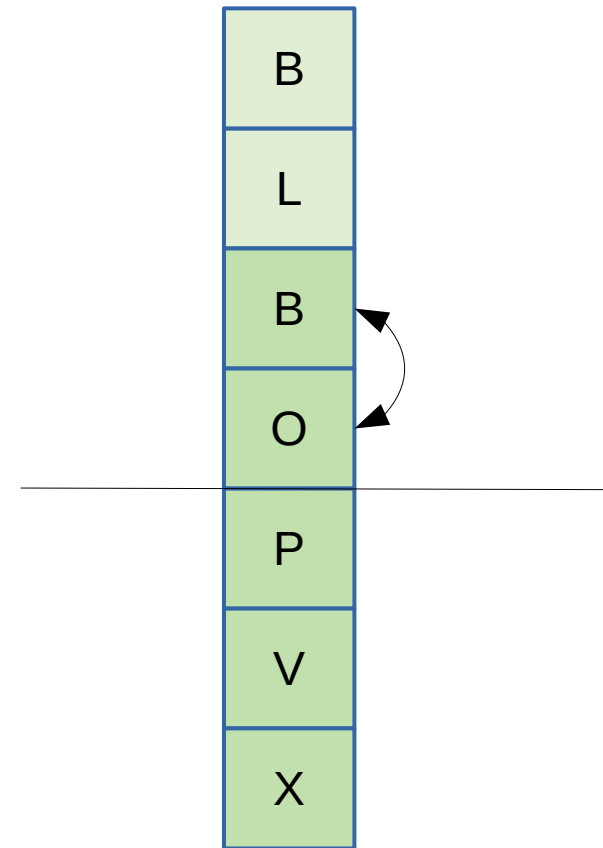
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

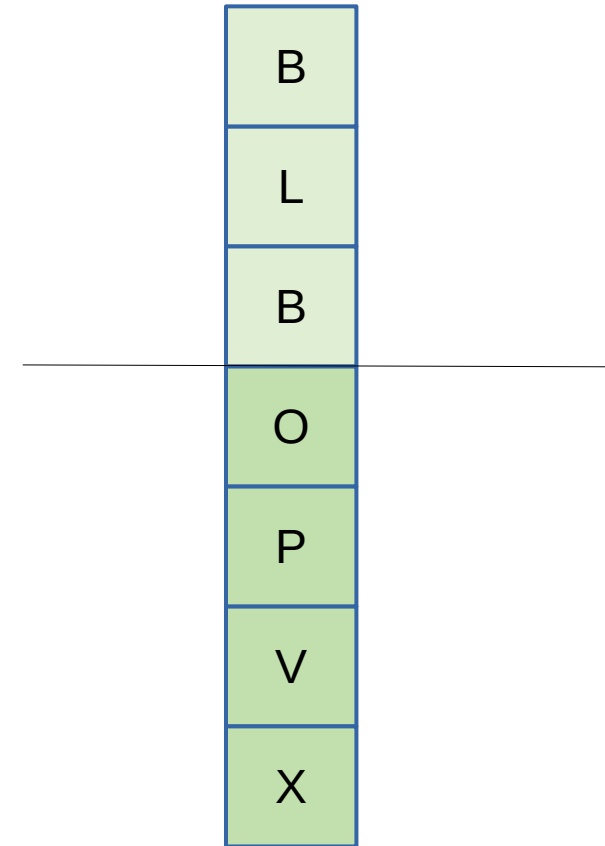


## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

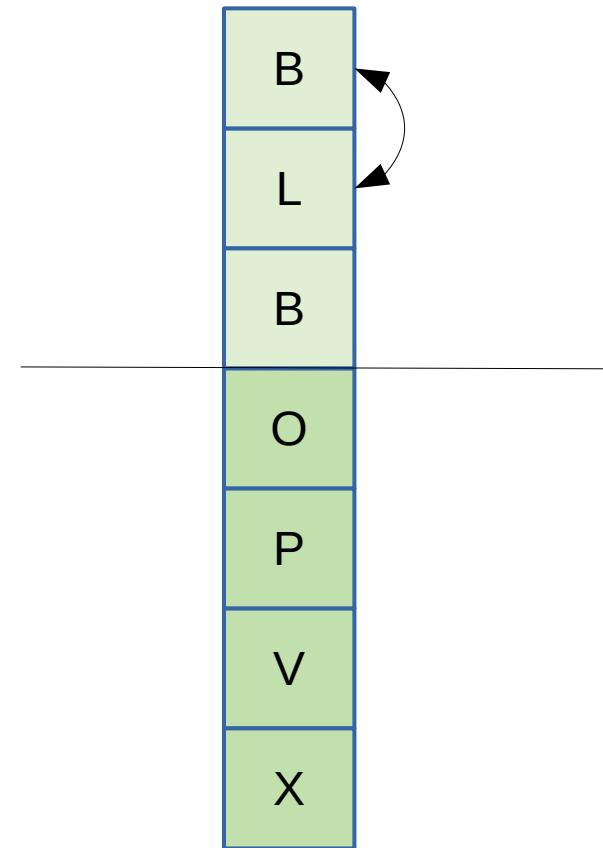
Fin du quatrième tour



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

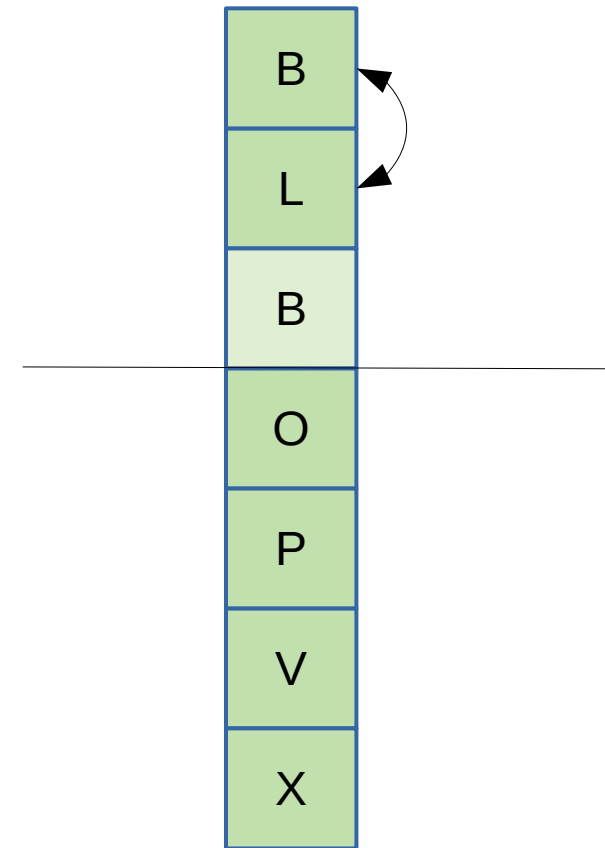
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

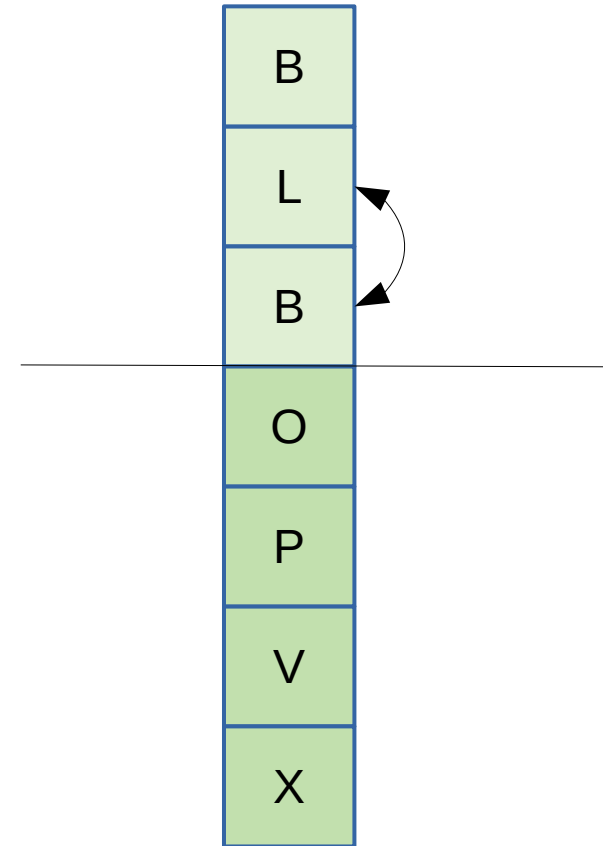
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

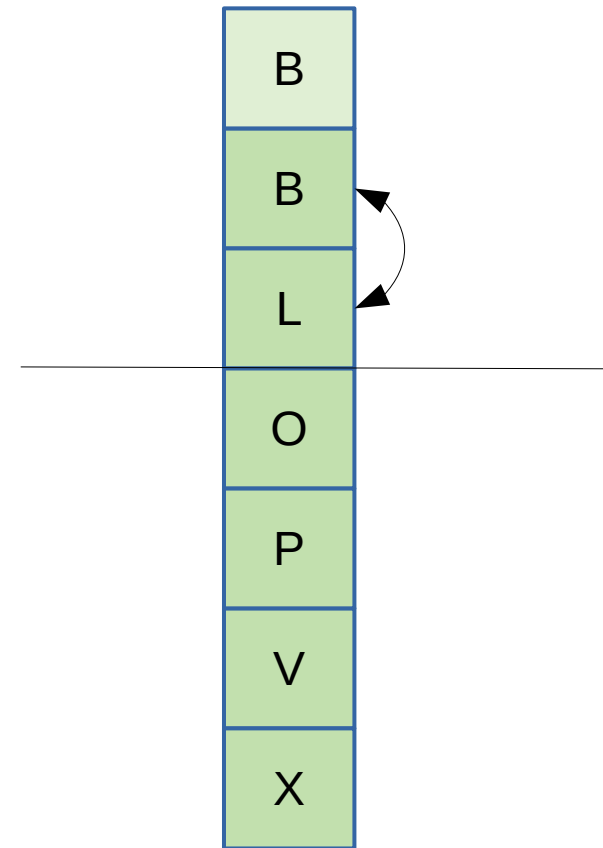
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.



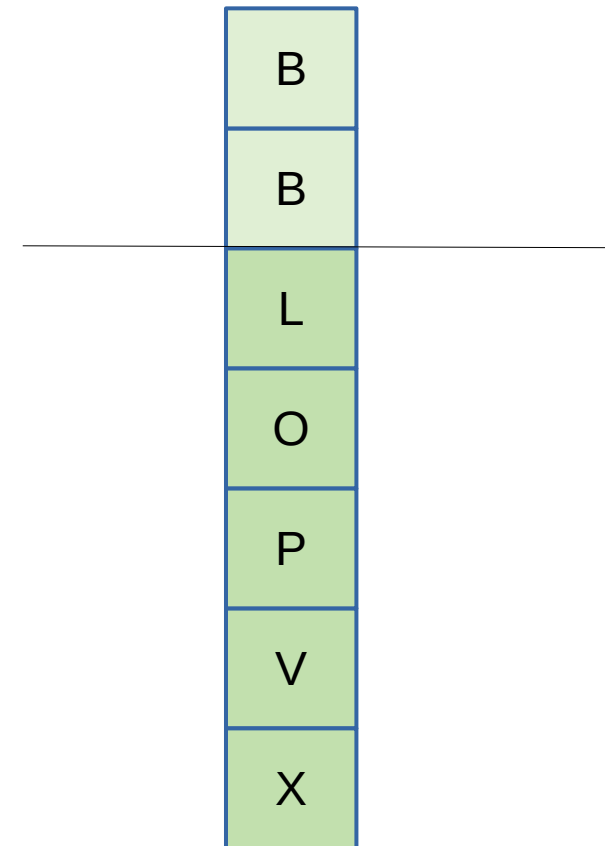


## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

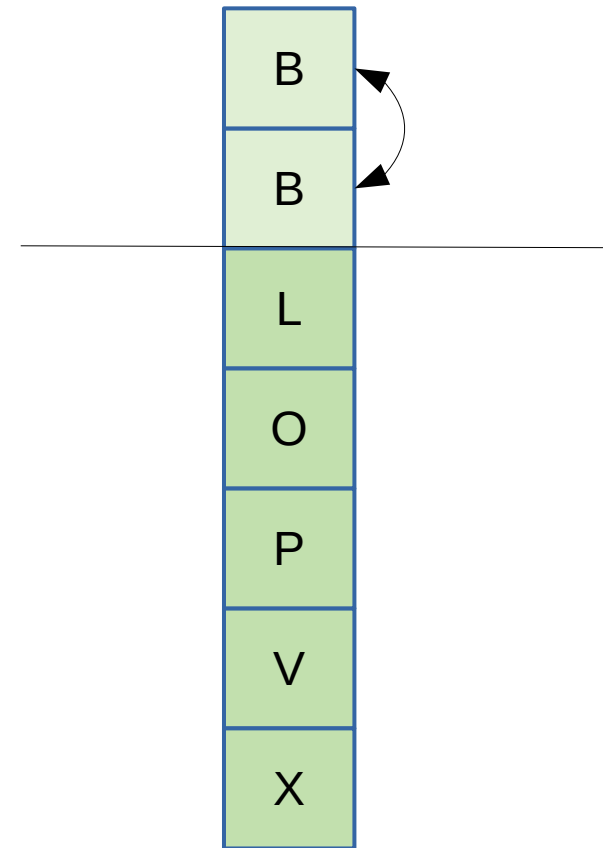
Fin du cinquième tour



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

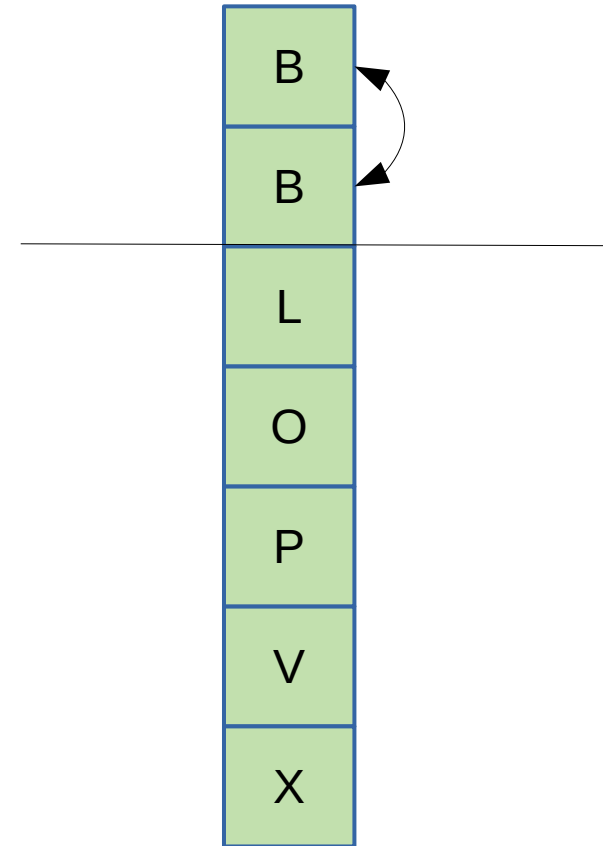
On arrête quand rien n'a bougé.



## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

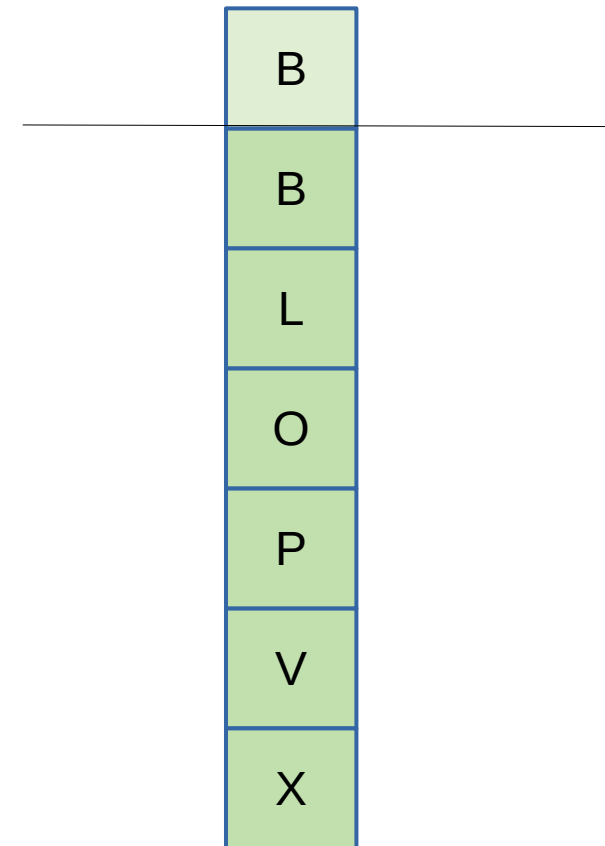


## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

Fin du sixième tour

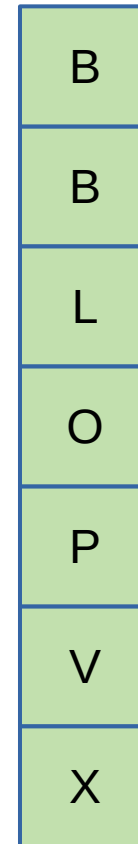


## Un dernier tri - Le tri à bulle (*bubble sort*)

Les petits éléments percolent,  
les gros éléments calent...

On arrête quand rien n'a bougé.

On arrête au sixième tour,  
rien n'a bougé.



# Un dernier tri - Le tri à bulle (*bubble sort*)

## Pseudo-code

Entrée :  $T$  un tableau de longueur  $n$

Sortie :  $T$  trié en ordre croissant

```
f ← n
témoin ← vrai
Tant que témoin est vrai, Faire
    témoin ← faux
    i ← 1
    Tant que i < f, Faire
        Si  $T[i - 1] > T[i]$ , Alors
            échanger  $T[i - 1]$  et  $T[i]$ 
            témoin ← vrai
        i ← i + 1
    f ← f - 1
```

Efficace pour les tableaux  
presque en ordre,  
mais en moyenne, très lent :(

$O(n^2)$

# Il y a d'autres façons de trier

Des meilleures ...

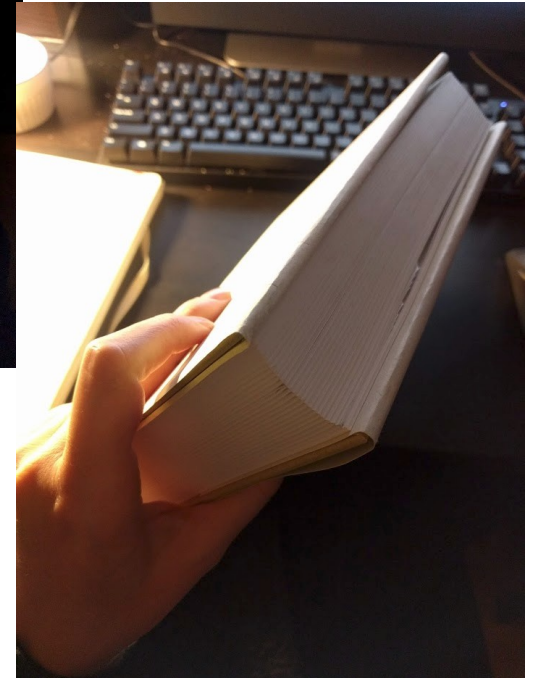
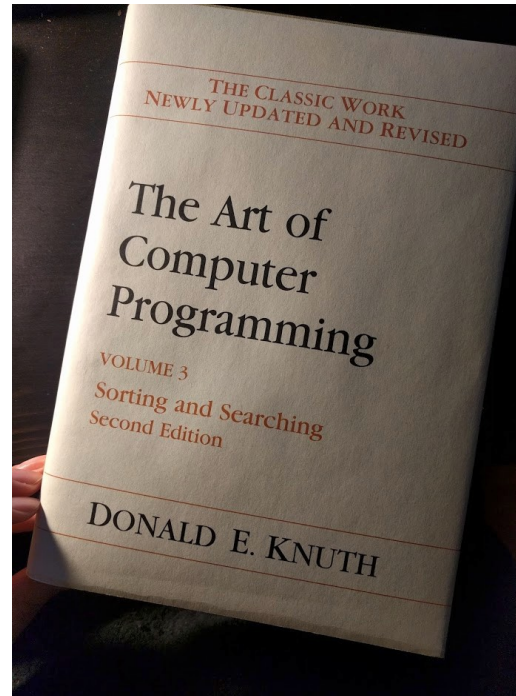
Tri rapide (*quick sort*)

Tri fusion (*merge sort*)

Tri par tas (*heap sort*)

Et des pires ...

Tri stupide (*bogo sort*)



# Il y a d'autres façons de trier

## Des meilleures ...

Tri rapide (*quick sort*)

Tri fusion (*merge sort*)

Tri par tas (*heap sort*)

## Et des pires ...

Tri stupide (*bogo sort*)

Un petit vidéo amusant  
qui montre les trois tris +  
le tri fusion :

[https://youtu.be/  
INHF\\_5RIxTE](https://youtu.be/INHF_5RIxTE)



# La récursivité

Programmation algorithmique (420-FC6-AG)

AUT2018

La récursion est reconnue comme un des grands concepts difficiles en programmation. Lisez ces notes quelques fois, répondez aux questions, essayez de dessiner la pile d'exécution et si vous rencontrez des difficultés, exprimez-les. Ces notes sont partielles, si vous remarquez des erreurs, mentionnez-les moi.

## 1 Qu'est-ce que la récursion

Une solution à un problème est récursive si elle le réduit à un problème plus court qui se résout de la même manière. Tous les langages modernes permettent à une fonction de contenir un appel à elle-même dans son corps. La fonction suivante est récursive :

---

```
1 void fonctionMystere (long n)
2 {
3     if (n == 0)
4         return;
5     fonctionMystere(n / 2);
6     printf("%ld", n % 2);
7 }
```

---

Elle fait un appel à elle-même à la ligne 5. Une fonction récursive peut toujours être modifiée pour ne pas qu'elle soit récursive, souvent, au coût de la lisibilité et de la concision. Le code de `fonctionMystere` est équivalent au code à l'annexe A.

Si vous exécutez cette fonction, vous verrez qu'elle transforme un nombre décimal en sa représentation binaire.

## 2 Comment utiliser la récursion

Pour utiliser la récursion une fonction doit au minimum avoir les trois éléments suivants :

1. Une condition de sortie,
2. Un appel récursif et
3. Un calcul de plus.

La condition de sortie sert à arrêter la récursion. L'appel récursif sert à répéter le corps de la fonction. On s'en sert pour réduire le problème à une instance plus petite. Le calcul de plus, c'est ce qui fait passer de la solution à l'instance plus petite à la solution voulue.

Dans `fonctionMystere`, la condition de sortie est `n == 0`. Quand `n` est nul, on a fini de traduire le nombre. L'appel récursif est `fonctionMystere(n / 2)` : Pour traduire `n` en binaire, on commence par traduire la moitié en binaire puis on ajoute 0 ou 1 à l'affichage pour écrire `n`. Dans `fonctionMystere`, le calcul de plus, c'est `printf("%d", n % 2)`.

Simplement dit, une fonction récursive résout un problème  $P$  en le décomposant en un problème plus petit  $P'$  qui se résout de la même manière. Une fois la solution à  $P'$  trouvée, on l'ajuste pour obtenir la solution recherchée au problème  $P$ .

## 3 Quelques fonctions intéressantes

Identifiez la condition de sortie, l'appel récursif et le calcul de plus dans les exemples suivants. Décrivez ce que fait le code. Transformez le code pour qu'il soit non-récursif.

### 3.1 Factorielle

---

```
1 int factorielle(int n)
2 {
3     if (n == 1)
4         return 1;
5     return n * factorielle(n - 1);
6 }
```

---

### 3.2 La suite de Fibonacci

---

```
1 int fibonacci(int n)
2 {
3     if (n == 1 || n == 0)
4         return 1;
5     return fibonacci(n - 1) + fibonacci(n - 2);
6 }
```

---

### 3.3 La suite de Collatz (Syracuse)

---

```
1 void collatz(int n)
2 {
3     printf("%d ", n);
4     if (n == 1)
5         return;
6     if (n % 2 == 0)
7         collatz(n / 2);
8     else
9         collatz(3 * n + 1);
10 }
```

---

### 3.4 Parenthésage préfixe

---

```
1 void parenthesePrefixe(char* c, int longueur)
2 {
3     if (longueur == 0)
4         return;
5     printf("(");
6     parenthesagePrefixe(c, longueur - 1);
7     printf("%c)", c[longueur - 1]);
8 }
```

---

### 3.5 Parenthésage suffixe

---

```
1 void parenthesageSuffixe(char* c, int longueur)
2 {
3     if (longueur == 0)
4         return;
5     printf("(%c", c[0]);
6     parenthesageSuffixe(c + 1, longueur - 1);
7     printf(")");
8 }
```

---

### 3.6 Palindrome

---

```
1 int palindrome(char* c, int longueur)
2 {
3     if (longueur <= 1)
4         return 1;
5     if (c[0] != c[longueur - 1])
6         return 0;
7     return palindrome(c + 1, longueur - 2);
8 }
```

---

## 4 Un tri récursif

Nous avons vu trois algorithmes de tri (insertion, sélection et à bulle) dont la complexité était quadratique ( $n^2$ ), même si certains (insertion et sélection) avaient des avantages d'efficacité sur la troisième. On peut trier une liste en faisant encore moins d'opérations ( $n \lg n$ ) en utilisant un algorithme récursif appelé "tri fusion".

Le tri fusion est un algorithme récursif qui décompose une liste en deux sous-listes, trie chacune d'elle et les fusionne.

### 4.1 La fusion de deux listes ordonnées

Soient deux listes ordonnées  $M$  et  $N$ , pour les fusionner dans une troisième liste ordonnée  $T$ , on effectue le pseudo-code suivant :

```
fusionner (M, N, T)
```

```
    i <- 0; // Servira a parcourir T
    m <- 0; // Servira a parcourir M
    n <- 0; // Servira a parcourir N

    Tant que i < longueur(M) + longueur(N)
        Si m < longueur(M) et (M[m] < N[n] ou n >= longueur(N))
            T[i] <- M[m]
            m <- m + 1
        Sinon
            T[i] <- N[n]
            n <- n + 1
        i <- i + 1
```

### 4.2 Le tri fusion

Une fois qu'on est capable de fusionner deux listes ordonnées, on peut exécuter le tri fusion. Soit  $T$  le tableau à ordonner.

```
triFusion(T)
```

```
    Si longueur(T) <= 1
        Retourner T
```

```
milieu <- longueur(T) / 2
M <- T[0] a T[milieu - 1]
N <- T[milieu] a T[longueur(T) - 1]

triFusion(M);
triFusion(N);
fusionner(M, N, T)
```

Trouvez les appels récursifs, les conditions d'arrêt et le calcul de plus. Appliquez l'algorithme sur le tableau suivant {L, 0, B, P, V, B, X}. Est-ce plus ou moins rapide que le tri par insertion?

## 5 La récurrence structurelle

On peut définir des structures de données récursives. Une structure est récursive si un de ses membres est une référence vers ce type. Les algorithmes dans ce genre de structures sont souvent récursifs.

### 5.1 Liste chaînée

Une liste chaînée est une liste dont chaque élément pointe vers le suivant, sauf le dernier qui pointe vers nul. Trouvez les appels de fonction récursifs. Quels sont les conditions d'arrêt? Le calcul de plus?

Pouvez-vous facilement transformer les fonctions récursives en fonctions itératives (non-récursives)?

---

```
1 typedef struct Noeud Noeud;
2 struct Noeud{
3     int donnee;
4     Noeud* suivant;
5 };
6
7 void imprimerListe(Noeud* n)
8 {
9     if (n == NULL)
10        return;
11    printf("%d", n->donnee);
12    imprimerListe(n->suivant);
13 }
14
15 void imprimerEnvers(Noeud* n)
16 {
17     if (n == NULL)
18        return;
19    imprimerEnvers(n->suivant);
20    printf("%d", n->donnee);
21 }
22
23 int longueur(Noeud* n)
24 {
25     if (n == NULL)
26        return 0;
27    return 1 + longueur(n->suivant);
28 }
```

---

## A Annexe

---

```
1 void fonctionIterative(long n)
2 {
3     if (n == 0)
4         return;
5
6     const long N = n;
7
8     int longueur = 1;
9     while (n / 2 > 0)
10    {
11        longueur++;
12        n = n / 2;
13    }
14
15    n = N;
16    int envers[longueur];
17    int i = 0;
18    while (n / 2 > 0)
19    {
20        envers[i] = n % 2;
21        n = n / 2;
22        i++;
23    }
24    envers[i] = n % 2;
25
26    for (i = longueur - 1; i >= 0; i--)
27        printf("%d", envers[i]);
28 }
```

---



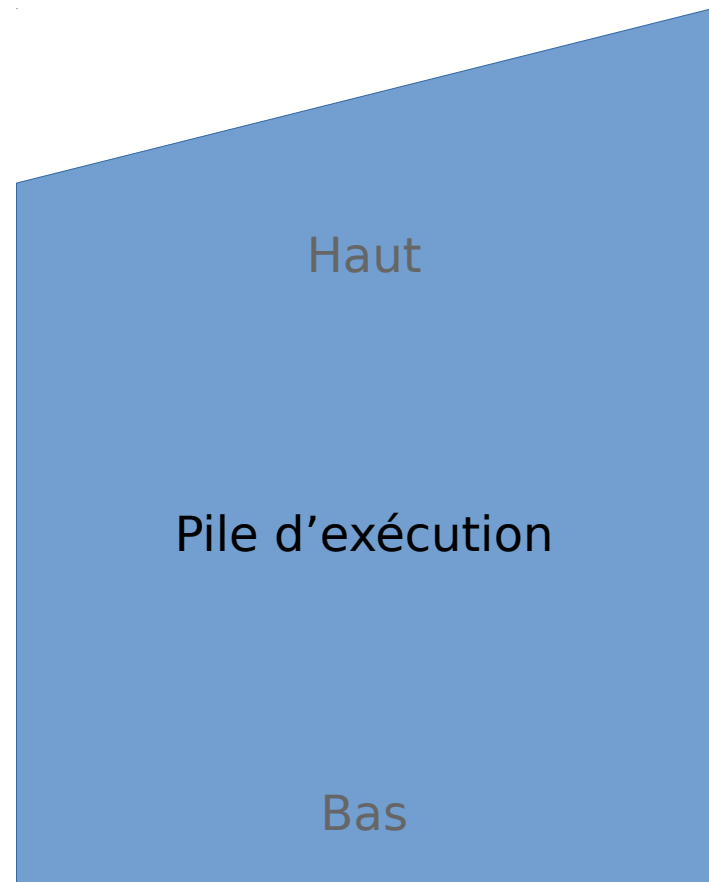
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4);
}
```



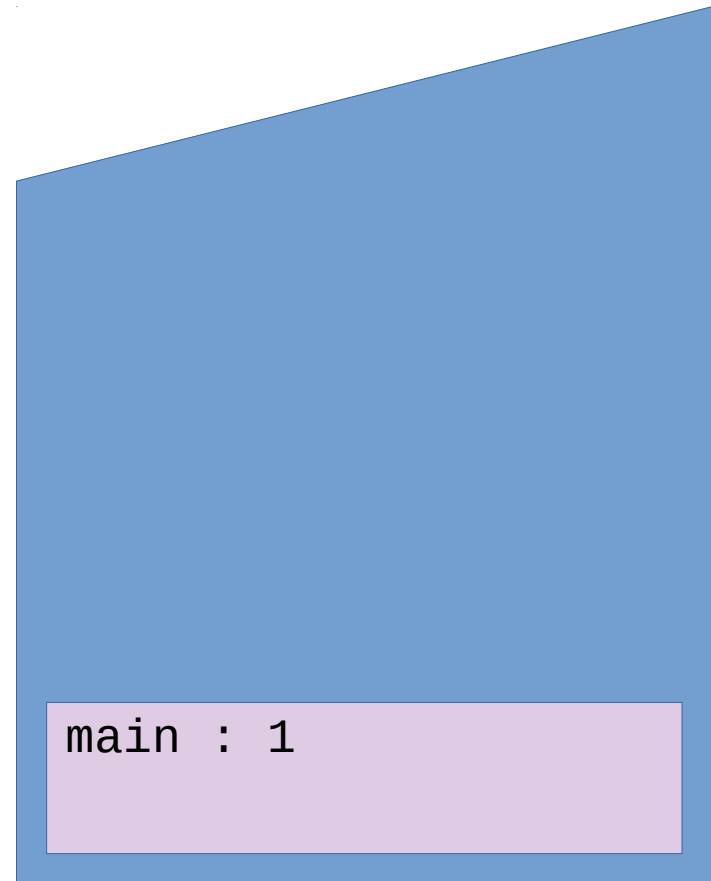
# Pile d'exécution pour des appels récur­sifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



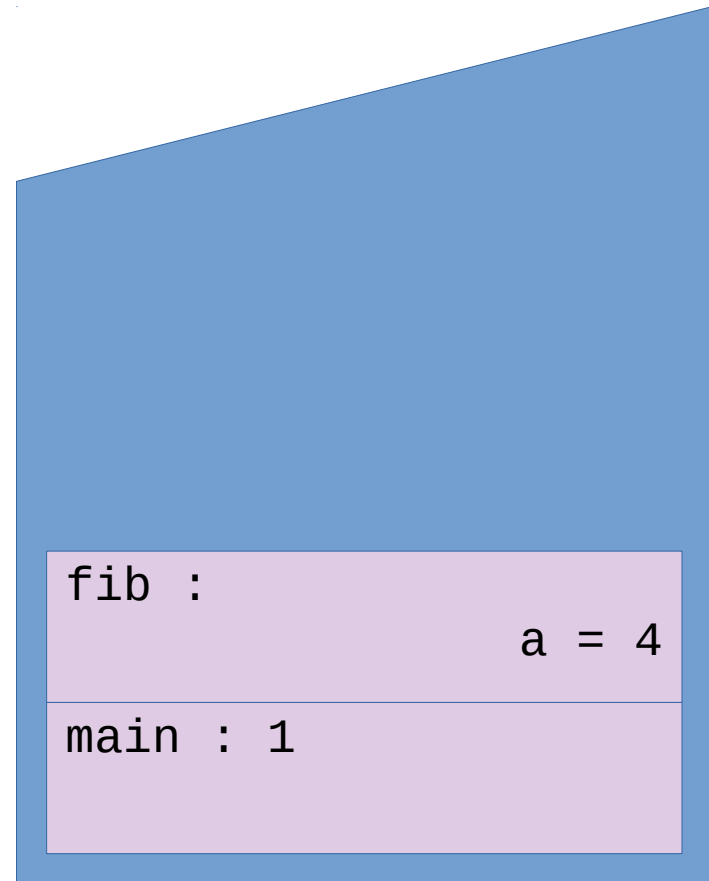
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



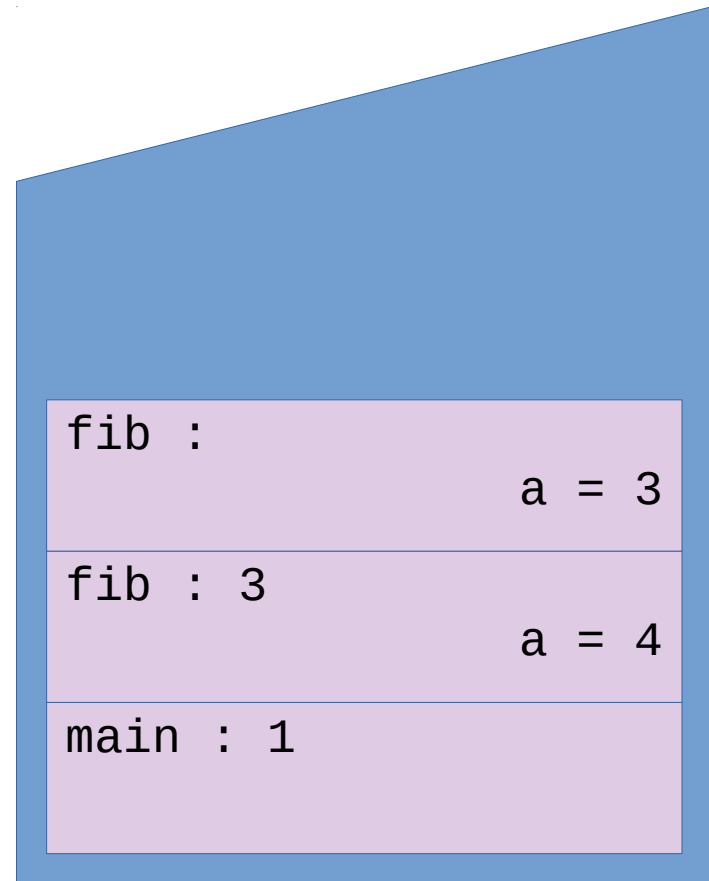
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



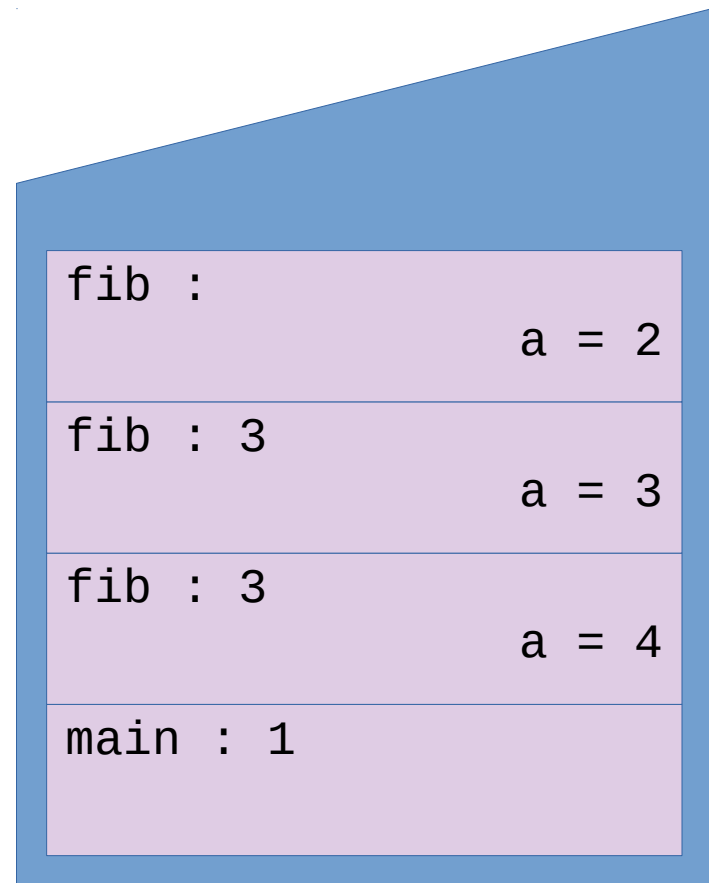
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



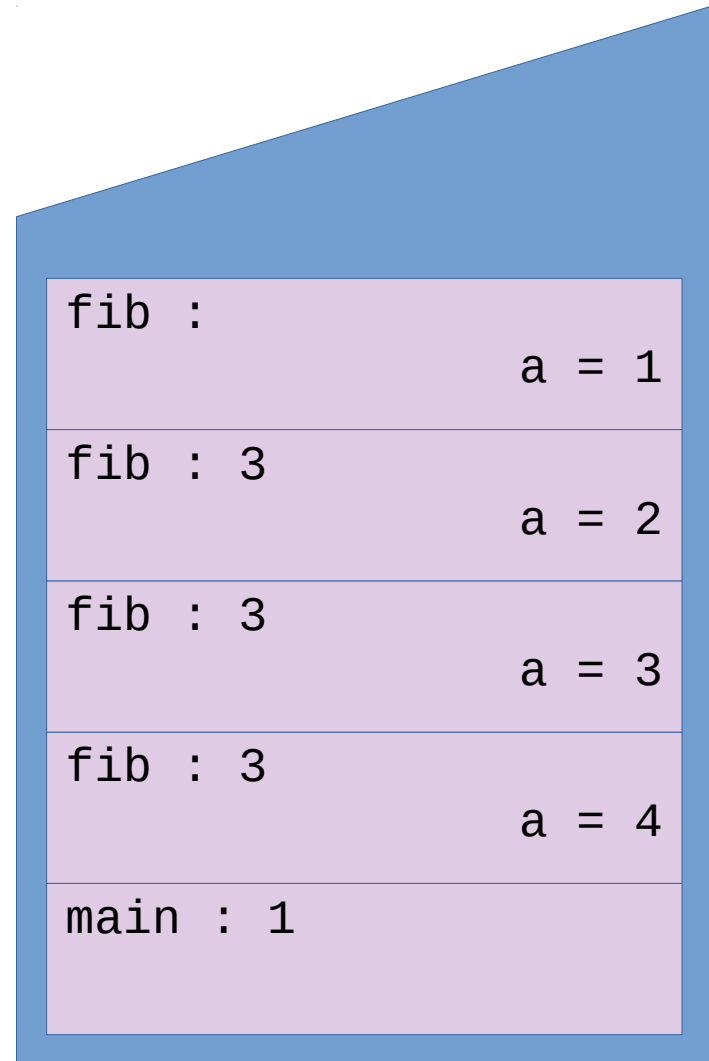
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



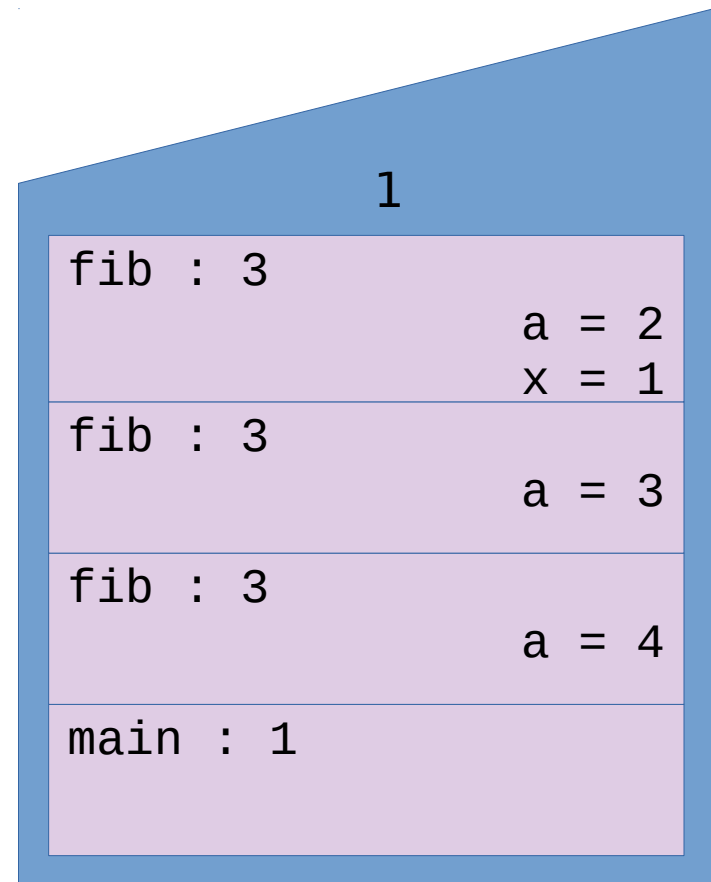
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



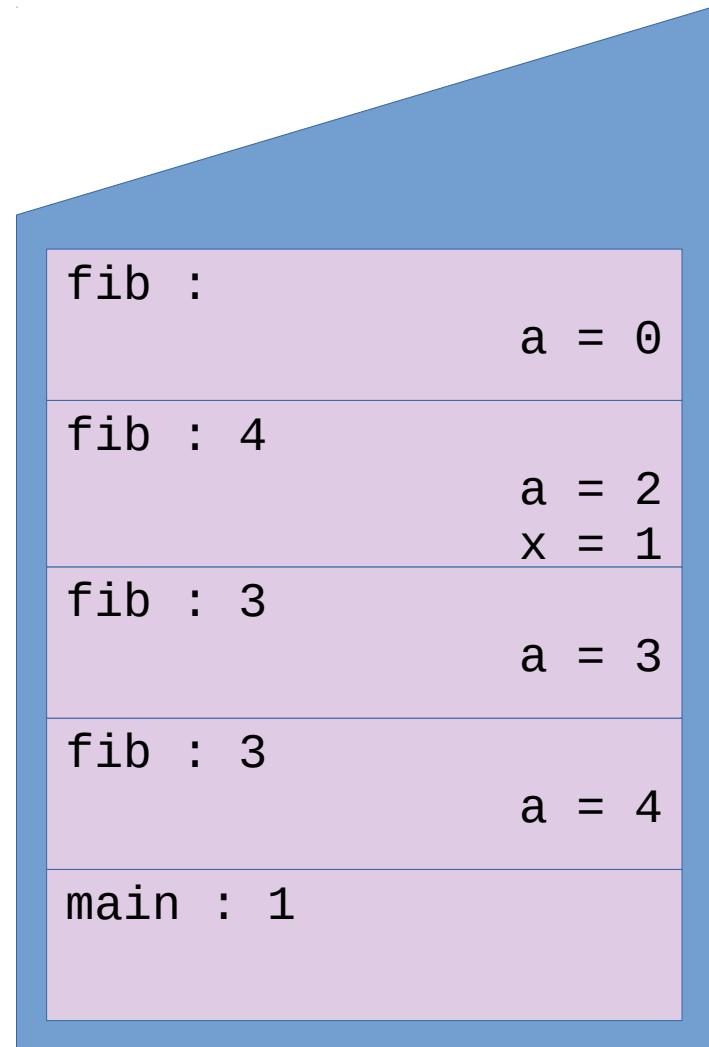
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```





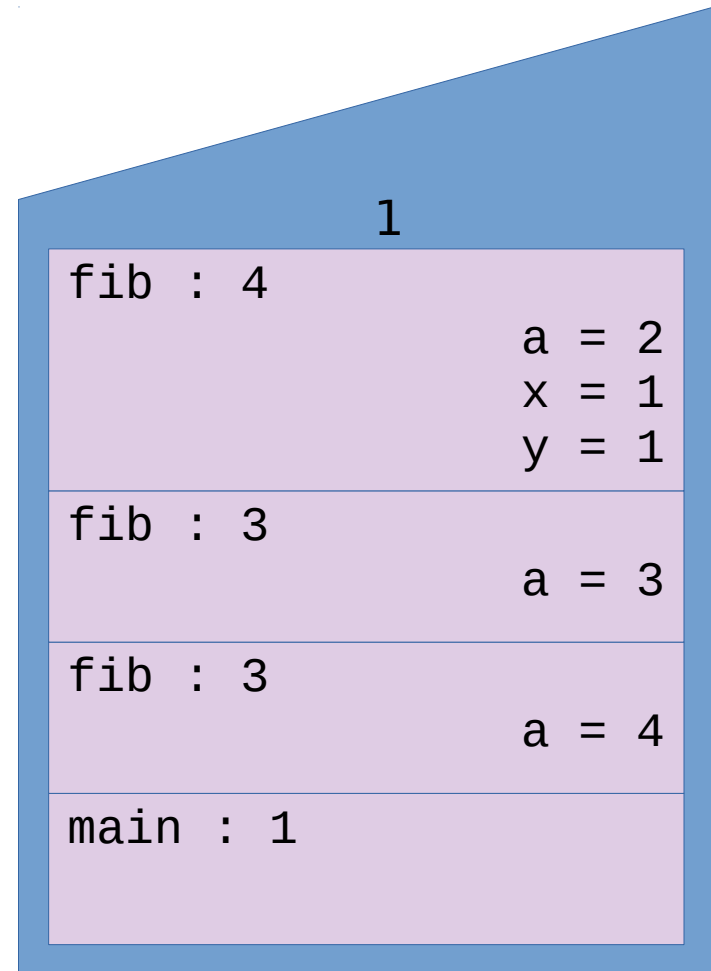
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



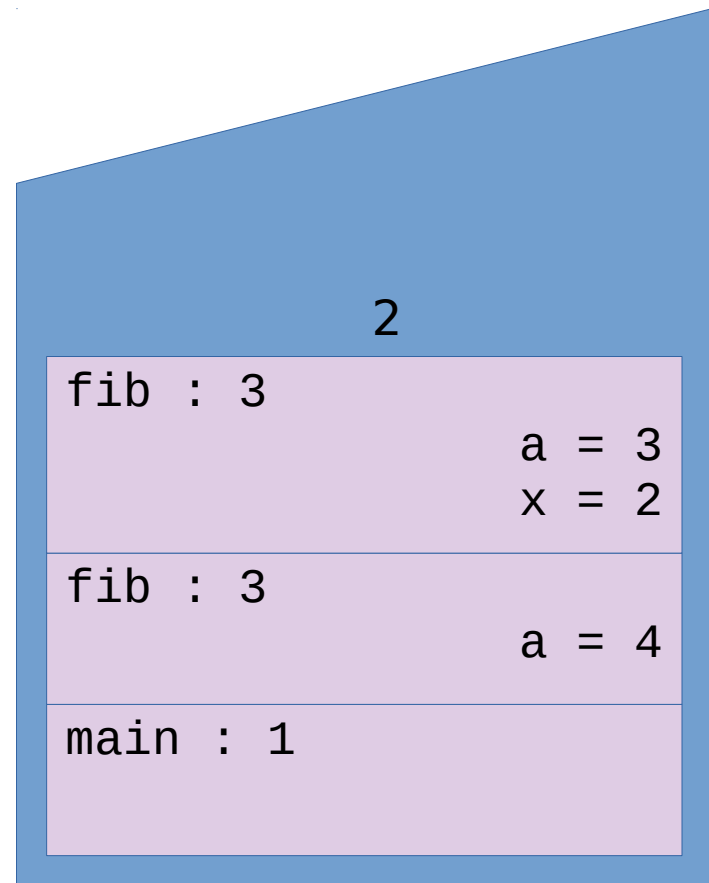
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



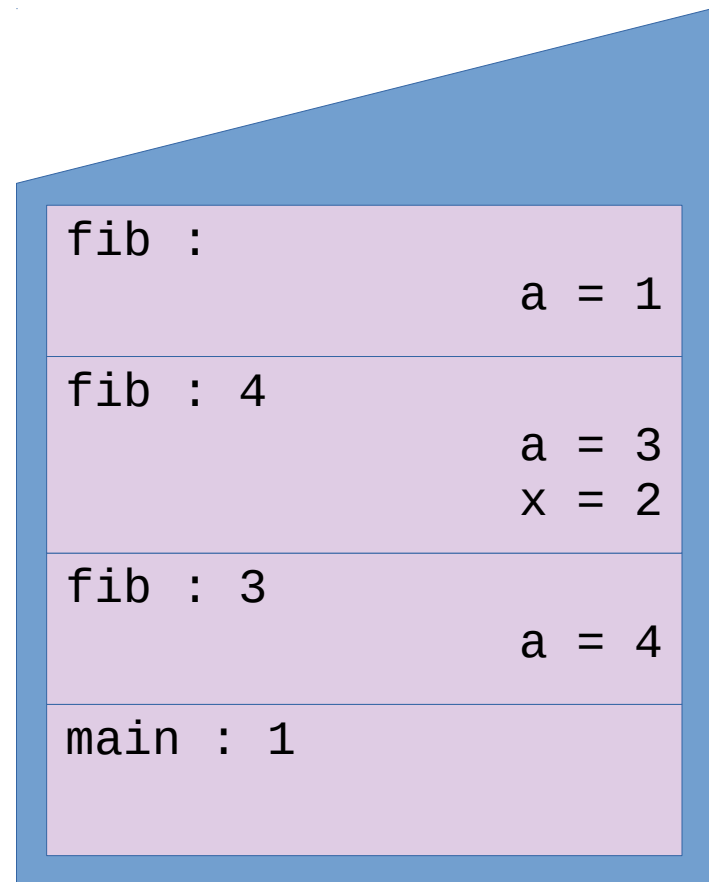
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



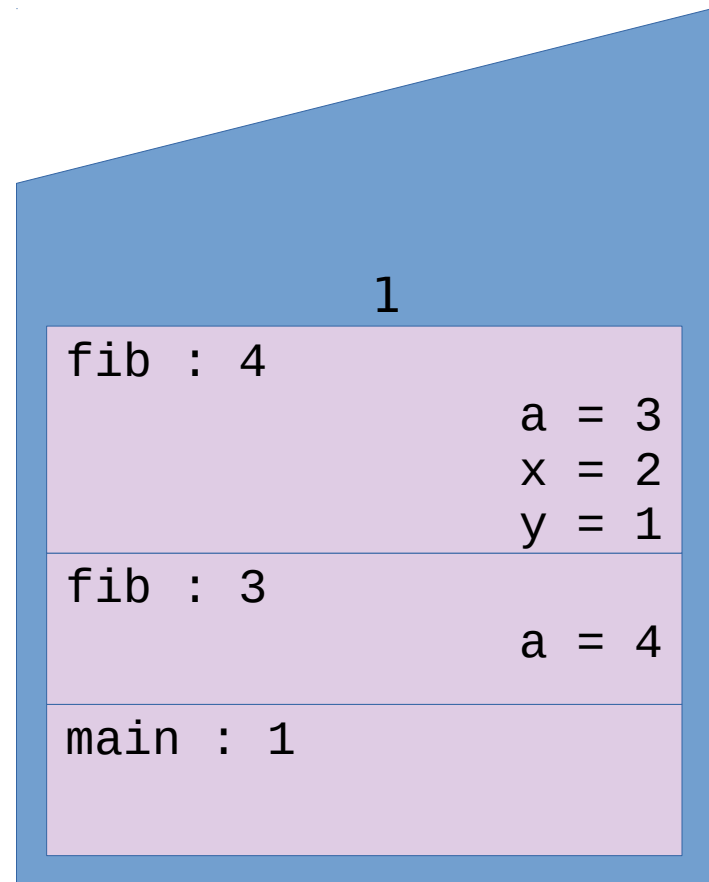
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



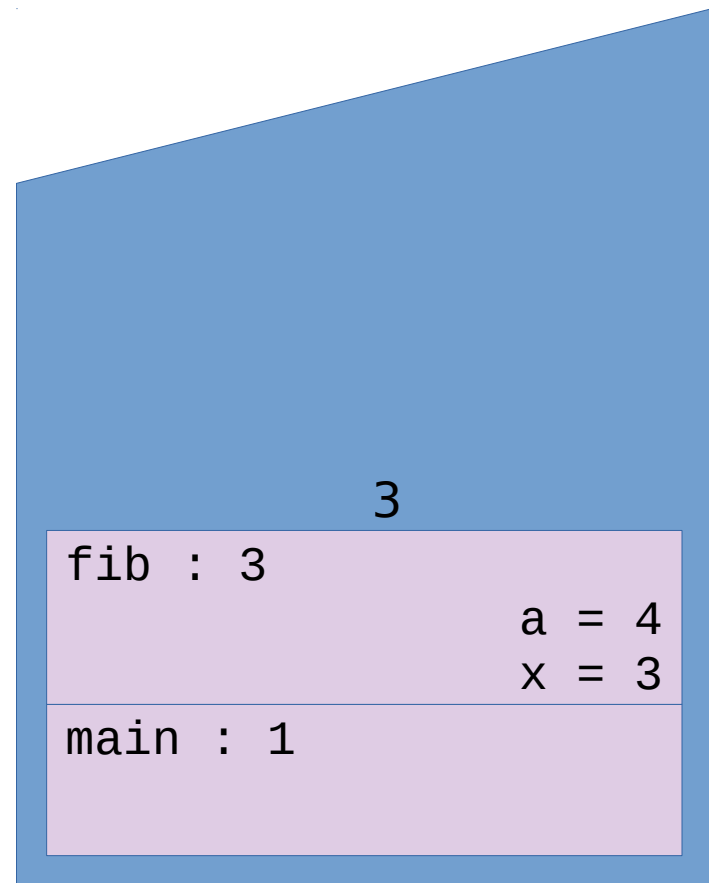
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



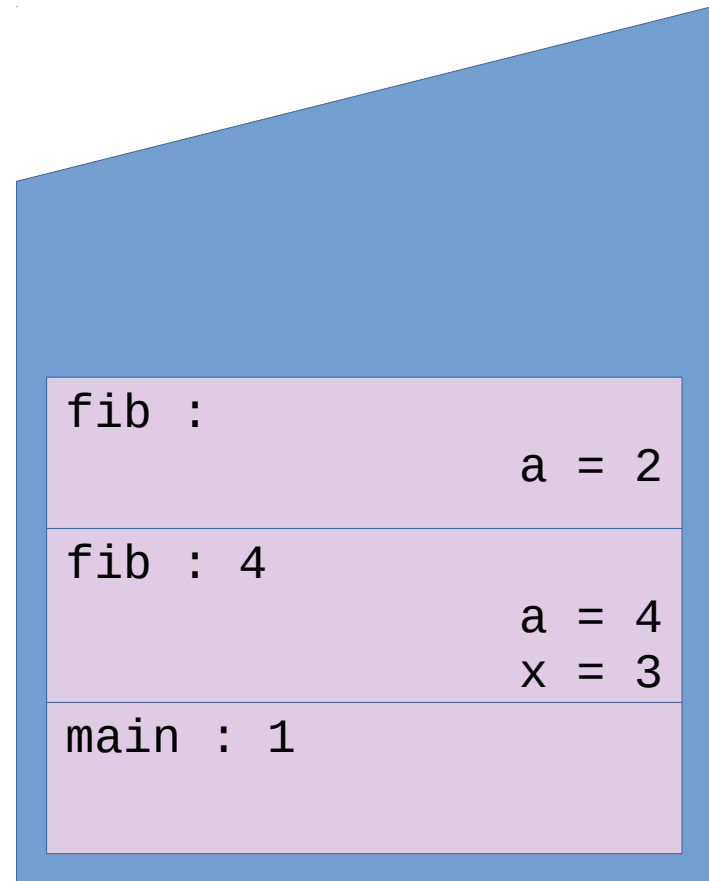
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



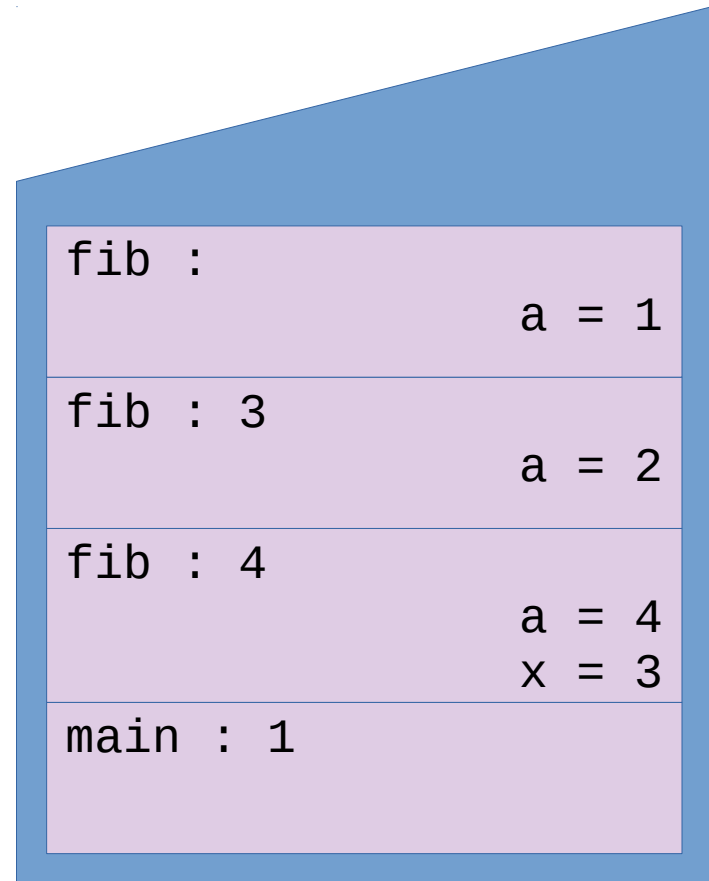
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



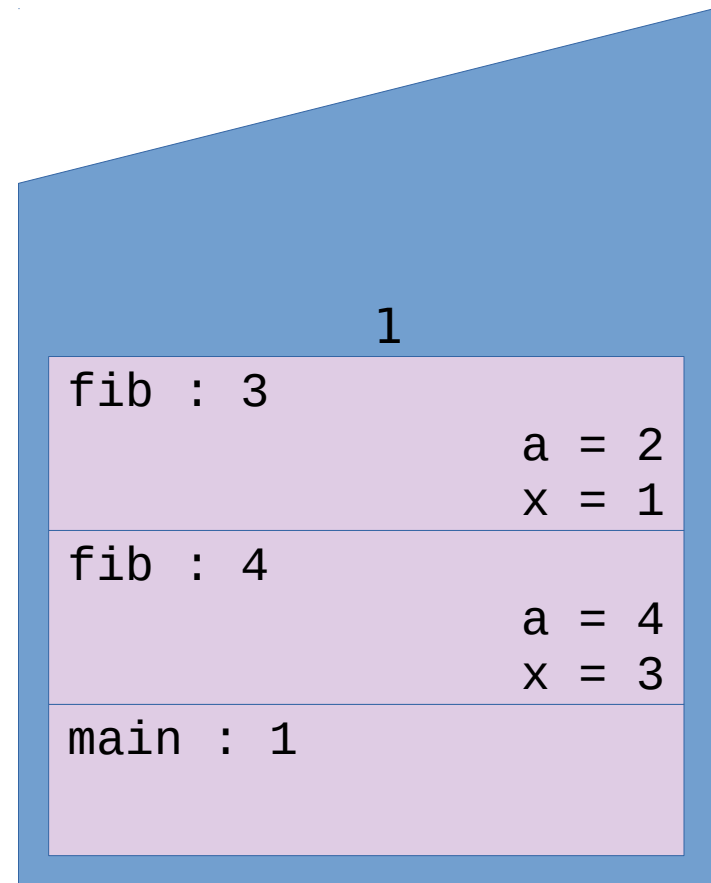
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```





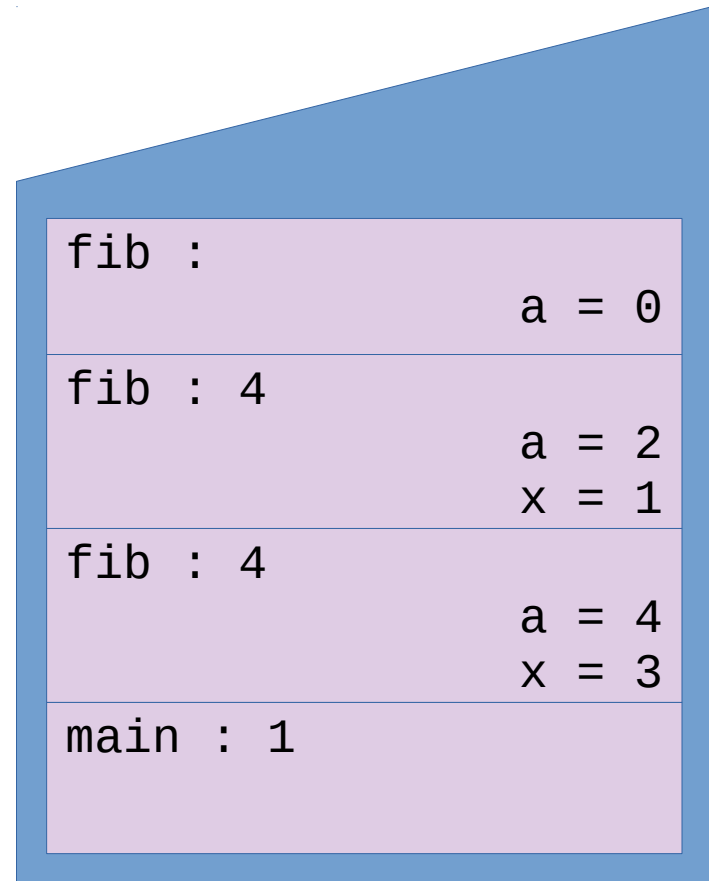
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



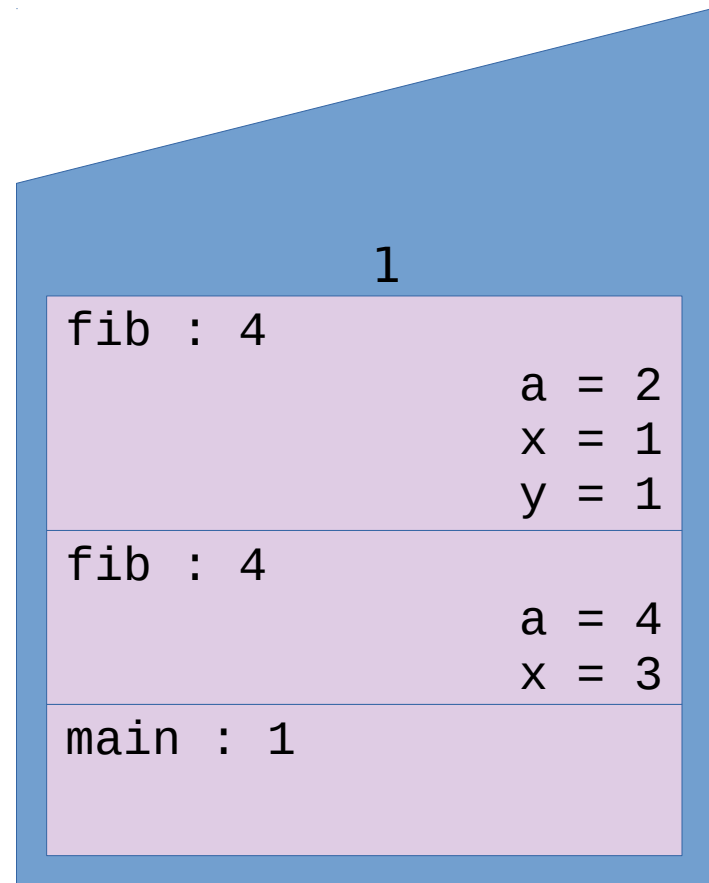
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



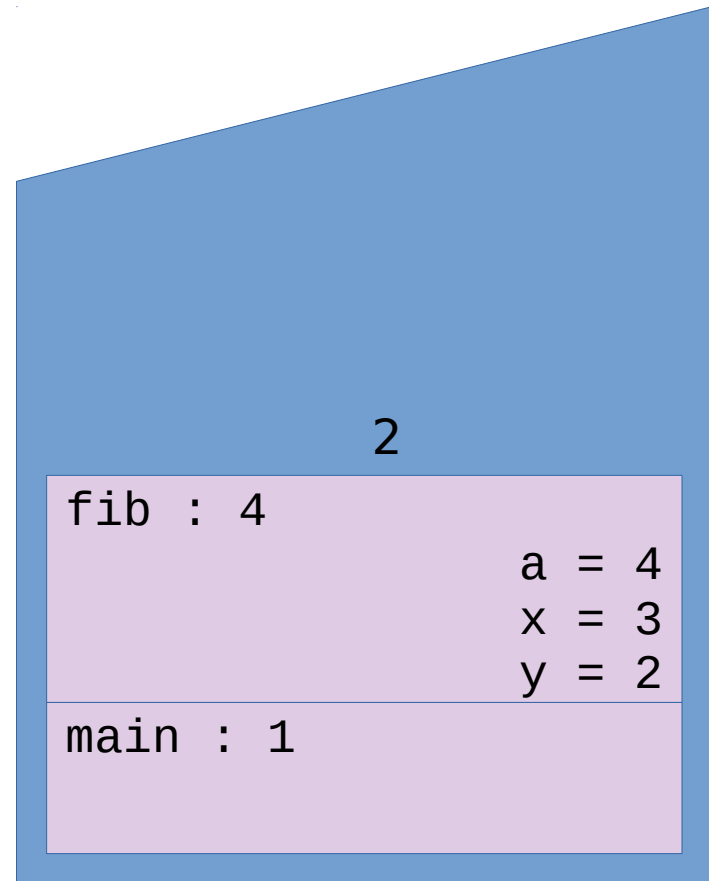
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



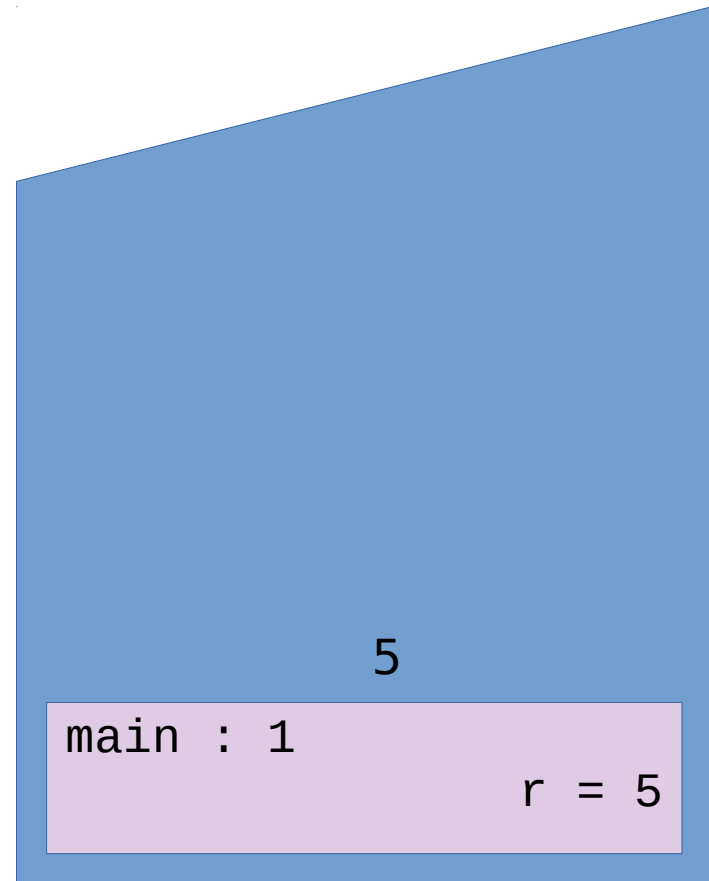
# Pile d'exécution pour des appels récursifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

int main()
{
1   int r = fib(4));
}
```



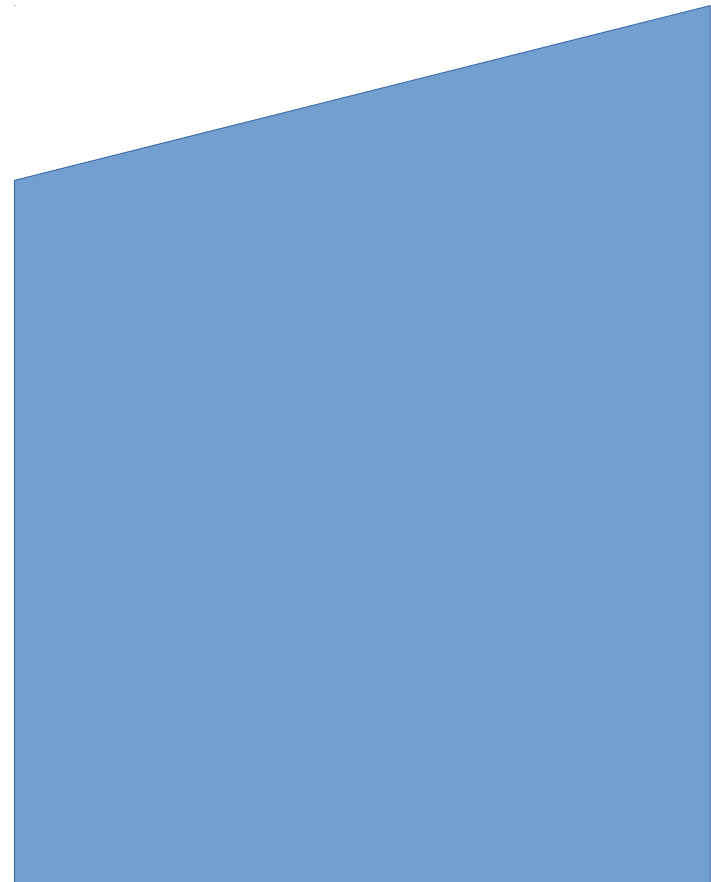
# Pile d'exécution pour des appels récur­sifs

```
int fib(int a)
{
1   if (a < 2)
2       return 1;

3   int x = fib(a - 1);
4   int y = fib(a - 2);

5   return x + y;
}

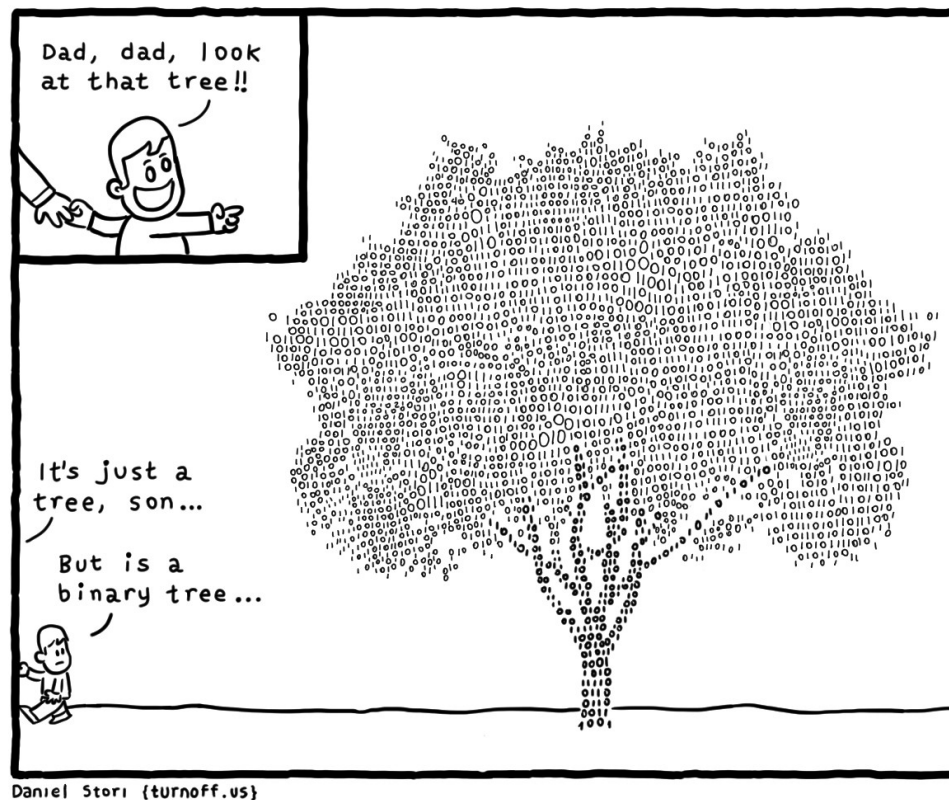
int main()
{
1   int r = fib(4));
}
```





# Programmation algorithmique

## Leçon 5 Les arbres



# Les arbres

## Aperçu

À quoi ça sert

Représentation

Le parcours

Les arbres de recherche binaire

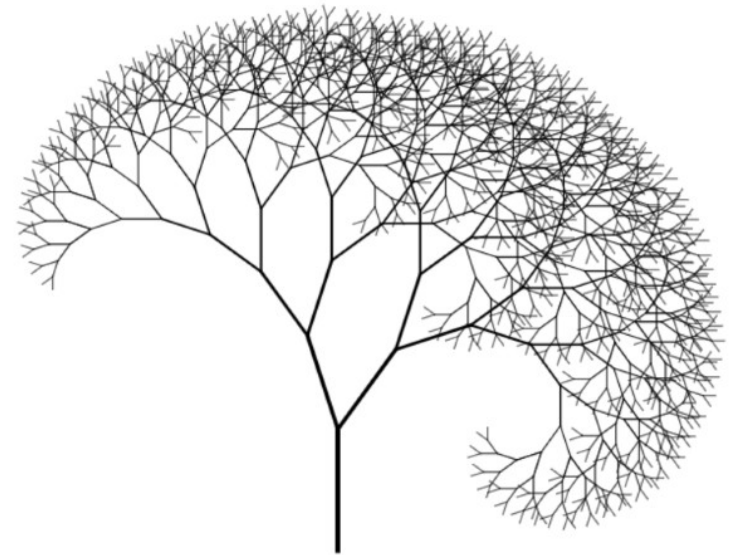
# Les arbres - Un aperçu

Un arbre est un ensemble de nœuds avec les propriétés suivantes :

Un nœud spécial est appelé la racine,

Chaque nœud a 0, 1 ou plusieurs nœuds enfants,

Chaque nœud a exactement un parent, sauf la racine qui en a aucun.



<http://blog.ploeh.dk/2017/06/06/fractal-trees-with-purescript/>



# Les arbres - Un aperçu

Un arbre est un ensemble de nœuds avec les propriétés suivantes :

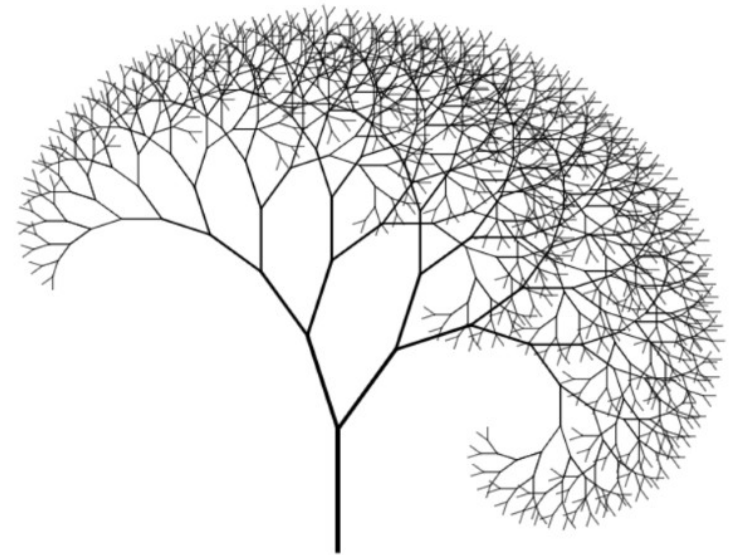
Un nœud spécial est appelé la racine,

Chaque nœud a 0, 1 ou plusieurs nœuds enfants,

Chaque nœud a exactement un parent, sauf la racine qui en a aucun.

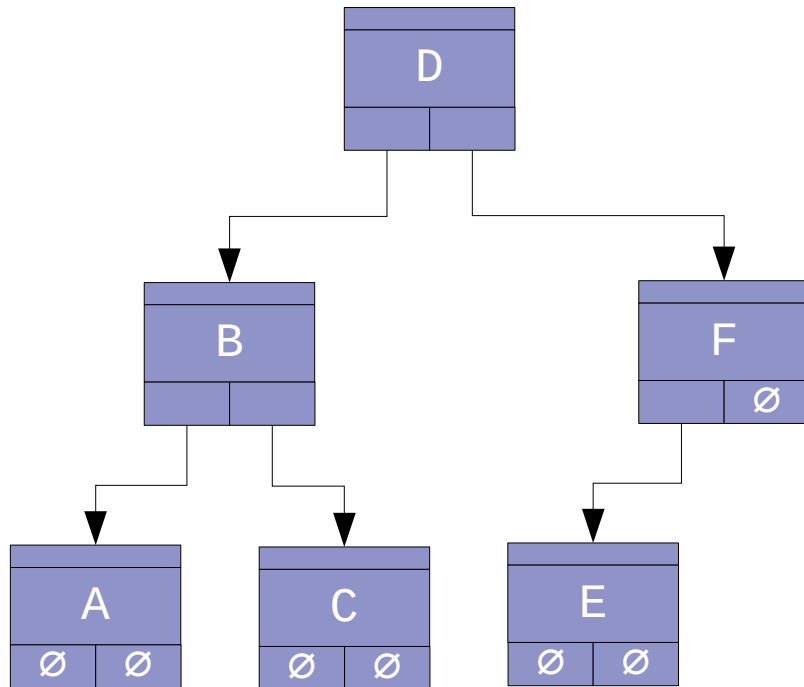
Arbre binaire = 0, 1 ou deux enfants.

Feuille = Un nœud sans enfants.



<http://blog.ploeh.dk/2017/06/06/fractal-trees-with-purescript/>

# Les arbres - Un aperçu



```
struct Noeud{  
    char donnee;  
    Noeud* g;  
    Noeud* d;  
};
```

```
int main(){  
    Noeud a = {'A', NULL, NULL};  
    Noeud c = {'C', NULL, NULL};  
    Noeud b = {'B', &a, &c};  
    Noeud e = {'E', NULL, NULL};  
    Noeud f = {'F', &e, NULL};  
    Noeud d = {'D', &b, f};  
}
```

Quel nœud est la racine, quelles sont les feuilles?

# Les arbres

Aperçu

**À quoi ça sert**

Représentation

Le parcours

Les arbres de recherche binaire

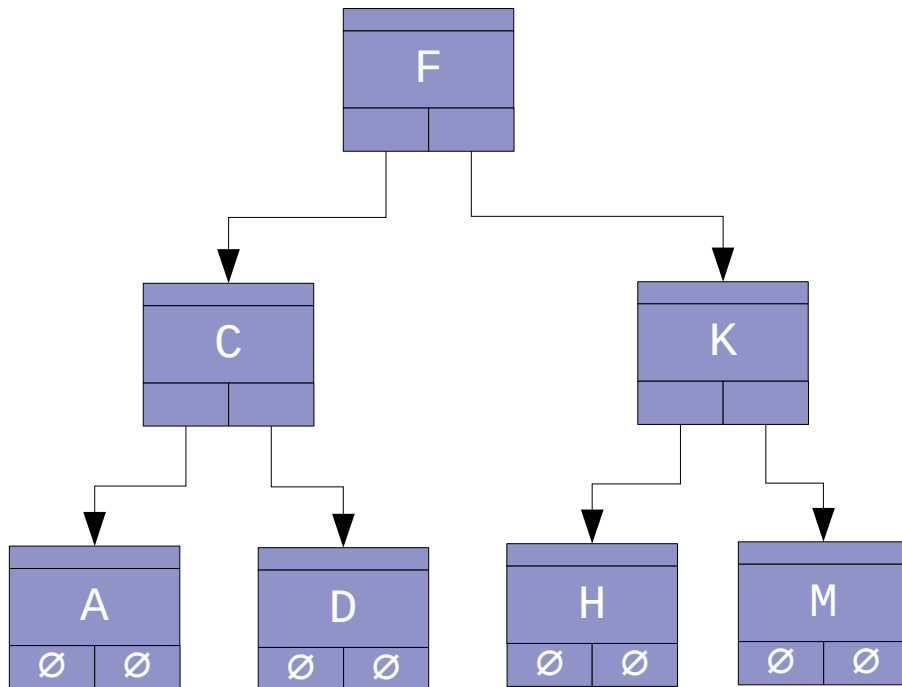
# Les arbres - À quoi ça sert

Les arbres (binaires) sont des structures fondamentales dans de nombreux algorithmes ...

- Opérations CRUD en bases de donnée,
- Compilateurs et interpréteurs,
- Compression d'image et de texte,
- 3D,
- Hachage,
- Partage de fichier,
- Chaîne de blocs,
- Chemin de décisions en IA, ...

# Les arbres - À quoi ça sert

Dans un arbre binaire de recherche, il n'est pas nécessaire de parcourir tous les nœuds pour fouiller.

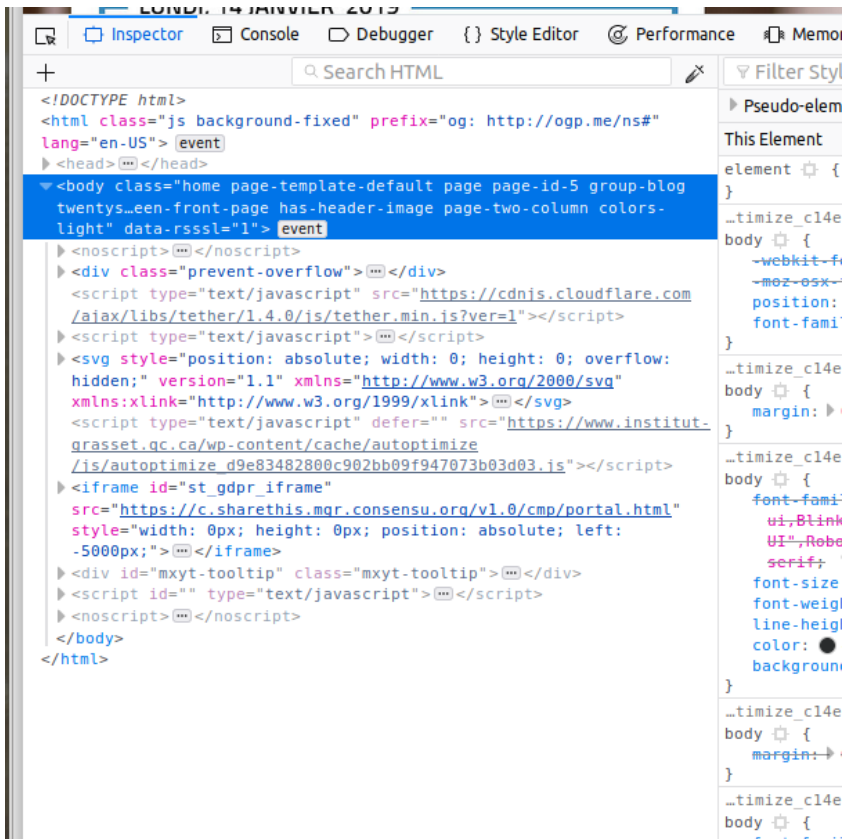


Est-ce que 'K' est dans l'arbre?

Est-ce que 'E' est dans l'arbre?

# Les arbres - À quoi ça sert

Beaucoup de données sont organisées sous forme d'arbre.



```
<!DOCTYPE html>
<html class="js background-fixed" prefix="og: http://ogp.me/ns#"
lang="en-US">
  <head>
  </head>
  <body class="home page-template-default page page-id-5 group-blog
  twentys...een-front-page has-header-image page-two-column colors-
  light" data-rsssl="1">
    <noscript>
    </noscript>
    <div class="prevent-overflow">
    </div>
    <script type="text/javascript" src="https://cdnjs.cloudflare.com
    /ajax/libs/tether/1.4.0/js/tether.min.js?ver=1">
    </script>
    <script type="text/javascript">
    </script>
    <svg style="position: absolute; width: 0; height: 0; overflow:
    hidden;" version="1.1" xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
    </svg>
    <script type="text/javascript" defer="" src="https://www.institut-
    grasset.qc.ca/wp-content/cache/autoptimize
    /is/autoptimize_d9e83482800c902bb09f947073b03d03.js">
    </script>
    <iframe id="st_gdpr_iframe"
    src="https://c.sharethis.mgr.consensu.org/v1.0/cmp/portal.html"
    style="width: 0px; height: 0px; position: absolute; left:
    -5000px;">
    </iframe>
    <div id="mxyt-tooltip" class="mxyt-tooltip">
    </div>
    <script id="" type="text/javascript">
    </script>
    <noscript>
    </noscript>
  </body>
</html>
```

Quel est le nœud racine dans une page html?

Identifiez les deux enfants de la racine.

# Les arbres

Aperçu

À quoi ça sert

**Représentation**

Le parcours

Les arbres de recherche binaire

# Les arbres - Représentation

On peut encoder des arbres de plusieurs façons. Les deux principales sont :

- Avec des structs
- Sous forme de tableau



# Les arbres - Représentation

On peut encoder des arbres de plusieurs façons. Les deux principales sont :

- Avec des structs
- Sous forme de tableau

```
char arbre[] = "FCKADHM";  
// Pour un noeud à la position i :  
//     Son enfant gauche est à la position  $2 * i + 1$   
//     Son enfant droite est à la position  $2 * i + 2$   
//     Son parent est à la position  $i / 2$ 
```

# Les arbres - Représentation

On peut encoder des arbres de plusieurs façons. Les deux principales sont :

- Avec des structs
- Sous forme de tableau

Taille dynamique

```
char arbre[] = "FCKADHM";  
// Pour un noeud à la position i :  
//     Son enfant gauche est à la position  $2 * i + 1$   
//     Son enfant droite est à la position  $2 * i + 2$   
//     Son parent est à la position  $i / 2$ 
```

# Les arbres

Aperçu

À quoi ça sert

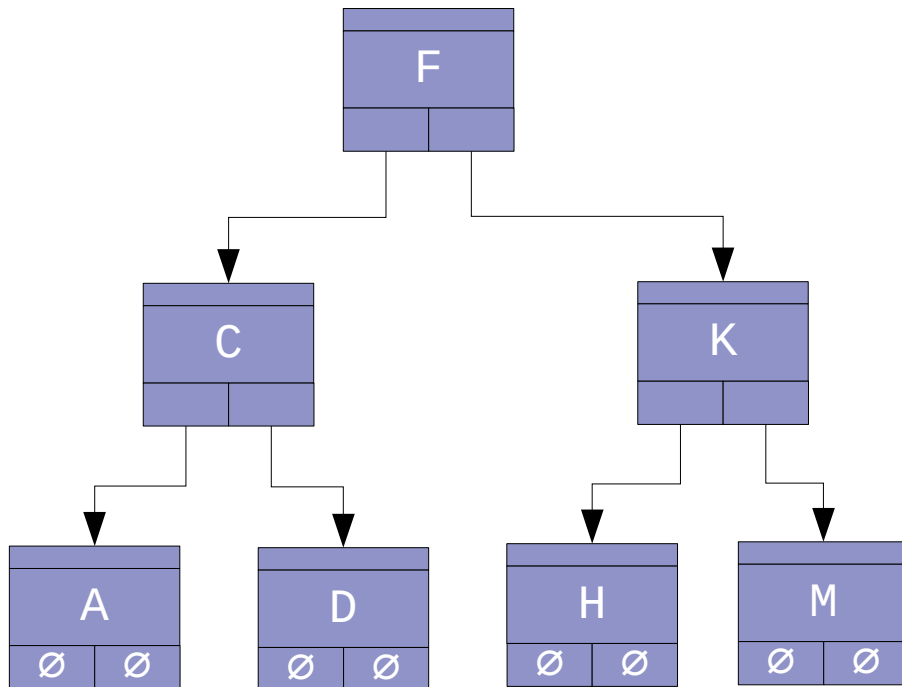
Représentation

**Le parcours**

Les arbres de recherche binaire

# Les arbres - les parcours

Comment parcourir un arbre?

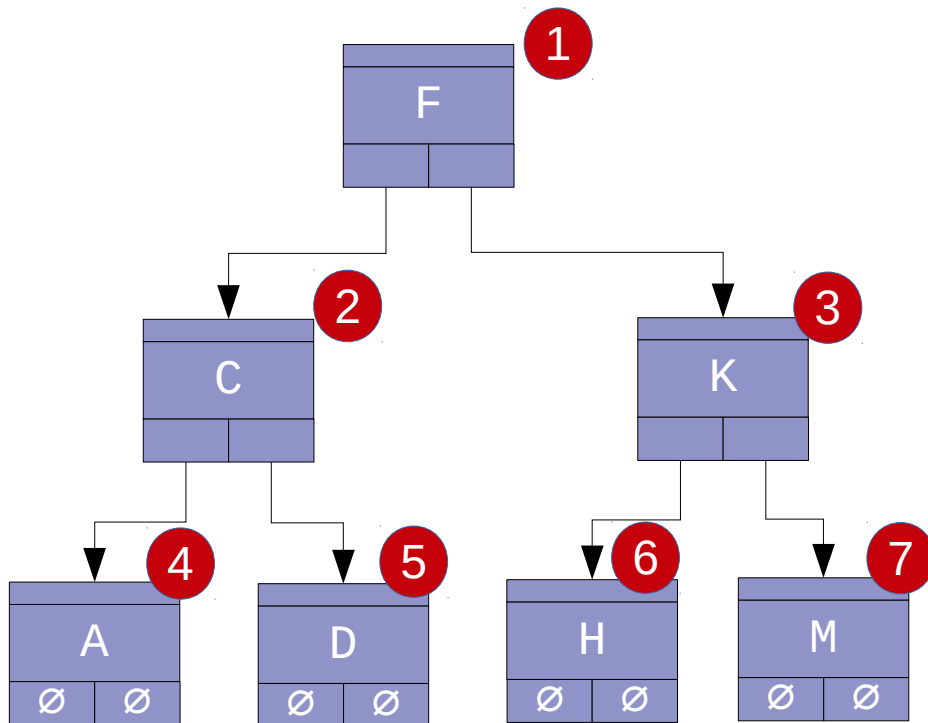


# Les arbres - les parcours

Comment parcourir un arbre?

## Le parcours en largeur

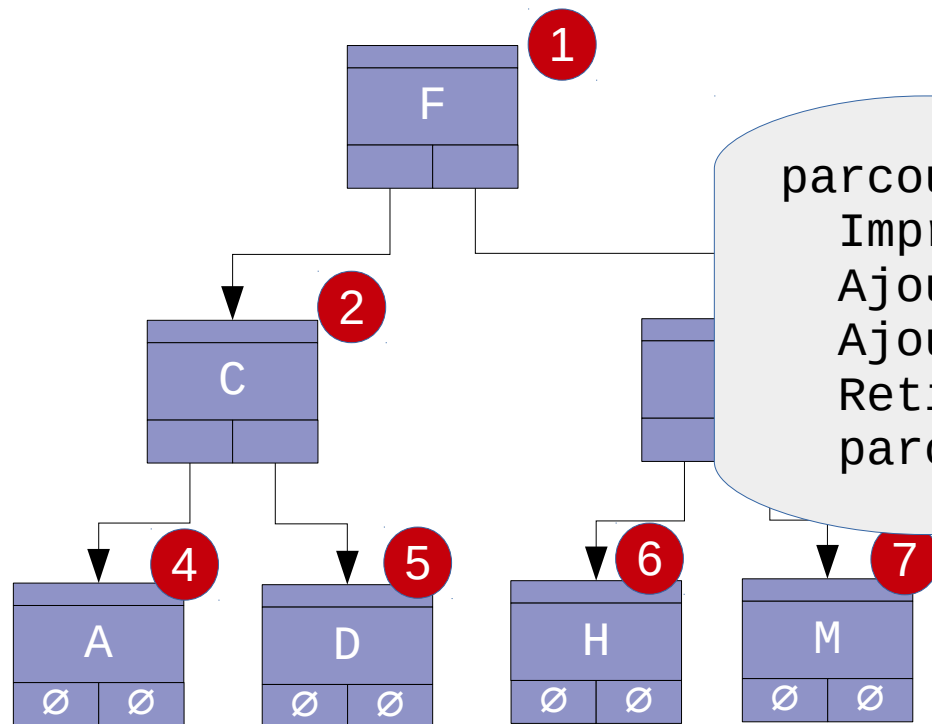
On imprime un nœud,  
puis ses enfants,  
puis leurs enfants,  
puis leurs enfants,  
etc.



# Les arbres - les parcours

Comment parcourir un arbre?

## Le parcours en largeur



`parcoursLargeur(queue)`

Imprimer le premier élément de la queue

Ajouter son enfant gauche à la queue

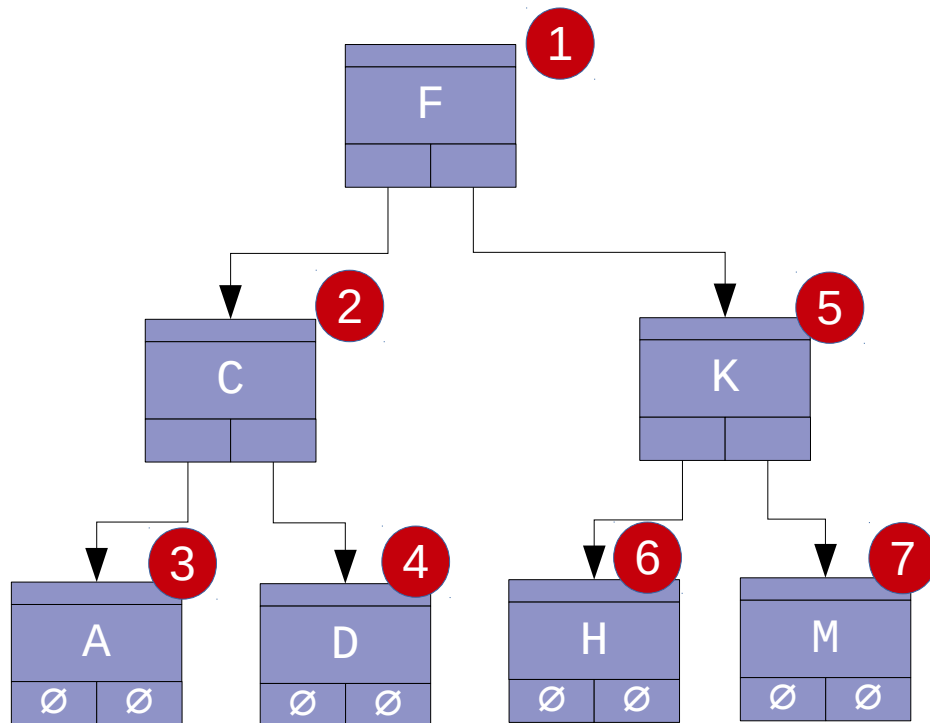
Ajouter son enfant droit à la queue

Retirer le premier élément de la queue

`parcoursLargeur(queue)`

# Les arbres - les parcours

Comment parcourir un arbre?



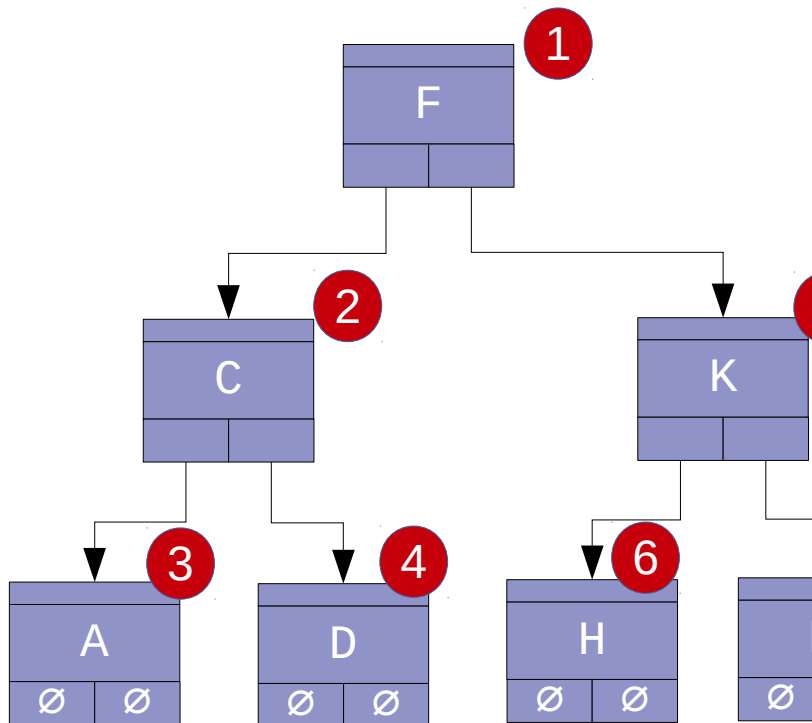
## Le parcours en profondeur

On imprime un nœud,  
puis son enfant gauche,  
puis son enfant droit,  
Récursivement

# Les arbres - les parcours

Comment parcourir un arbre?

## Le parcours en profondeur



```
void parcoursProfondeur(Noeud* racine)
{
    if (racine != NULL)
    {
        printf("%c", racine->donnee);
        parcoursProfondeur(racine->g);
        parcoursProfondeur(racine->d);
    }
}
```

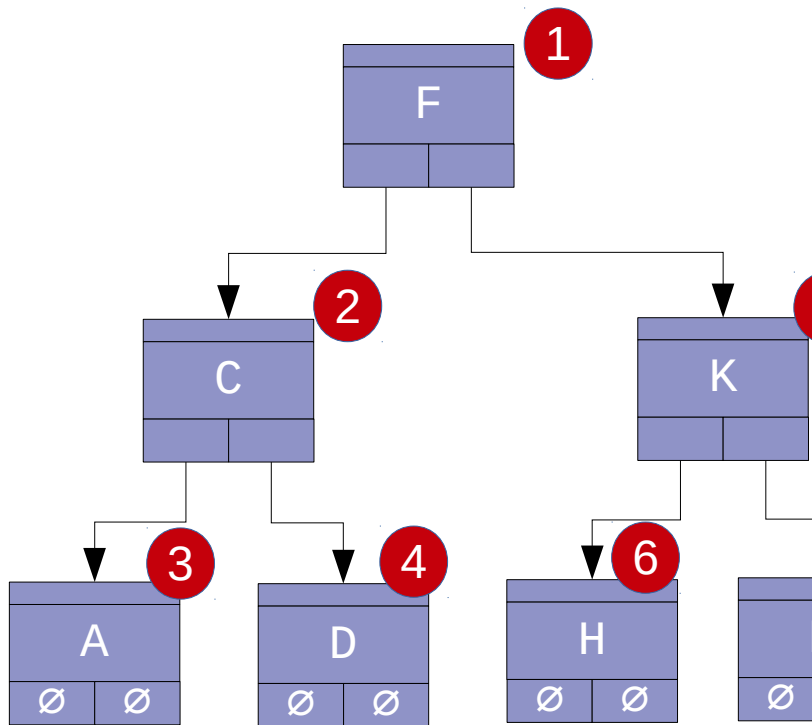
Bâissez cet arbre, et essayez le parcours en profondeur  
Changer l'ordre des lignes en italique.  
Que remarquez-vous?



# Les arbres - les parcours

Comment parcourir un arbre?

## Le parcours en profondeur



```
void parcoursProfondeur(Noeud* racine)
{
    if (racine != NULL)
    {
        printf("%c", racine->donnee);
        parcoursProfondeur(racine->g);
        parcoursProfondeur(racine->d);
    }
}
```

Un arbre de fréquence peut servir à l'encodage de mots en leur représentation binaire la plus succincte.

# Les arbres

Aperçu

À quoi ça sert

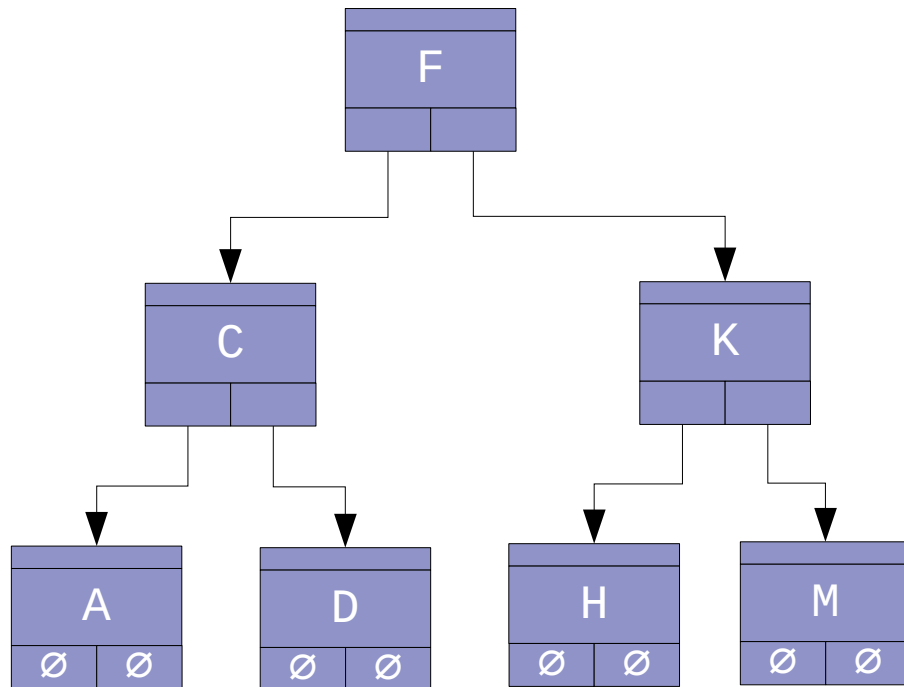
Représentation

Le parcours

**Les arbres de recherche binaire**

# Les arbres - Les arbres de recherche

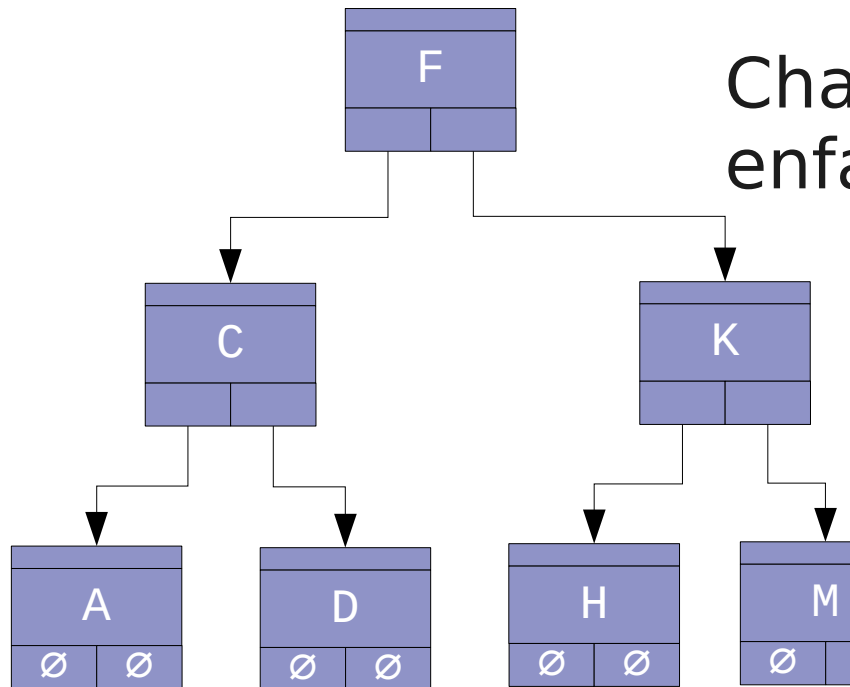
Les arbres de recherche sont des structures dynamique optimisées pour la recherche (!).



**Chaque nœud est entre son enfant gauche et son enfant droit.**

# Les arbres - Les arbres de recherche

Les arbres de recherche sont des structures dynamique optimisées pour la recherche (!).

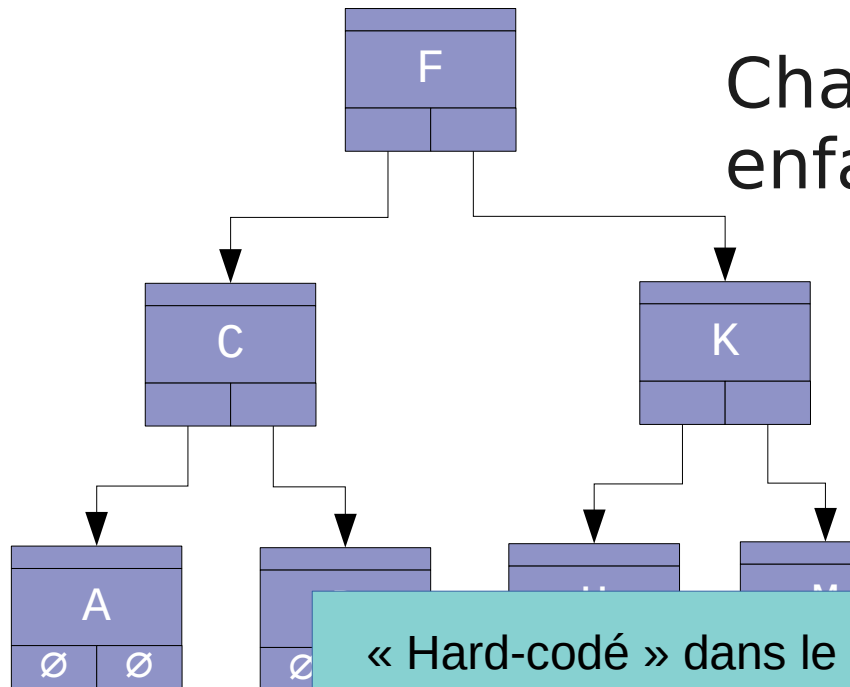


Chaque nœud est entre son enfant gauche et son enfant droit.

```
fouille(racine, c)
Si racine = nul
    Retourner faux
Si racine = c
    Retourner vrai
Si racine > c
    Retourner fouille(racine.g, c)
Sinon
    Retourner fouille(racine.d, c)
```

# Les arbres - Les arbres de recherche

Les arbres de recherche sont des structures dynamique optimisées pour la recherche (!).



Chaque nœud est entre son enfant gauche et son enfant droit.

```
fouille(racine, c)
Si racine = nul
    Retourner faux
Si racine = c
    Retourner vrai
Si racine > c
    Retourner fouille(racine.g, c)
Si racine < c
    Retourner fouille(racine.d, c)
```

« Hard-codé » dans le main, bâtissez un arbre de recherche pour les nombres {1, 2, 6, 8, 9, 11, 12, 15, 20}. Traduisez le pseudo-code en C et cherchez 16. Imprimez le parcours effectué pour trouver 16.

Montreal, 26/10/2018.

## Algorithme

Définition -> liste d'instructions qui résolvent un problème.

### Pseudo code

Variable  
Affectation  $\leftarrow$   
Si...alors  
Tant que...faire

Si  $x < 0$ ,  
Alors  $x \leftarrow x * (-1)$   
Retourner  $x$

If ( $x < 0$ )  
 $x = x * -1$ ;  
return  $x$ ;

### Les Chiffres Romain

M = 1000 | D = 500 | C = 100 | L = 50 | X = 10 | V = 5 | I = 1

CM = 900 (100 – 1000 = 900)

CD = 400 (100 – 500 = 400)

XC = 90 (10 – 100 = 90)

XL = 40 (10 – 50 = 40)

IX = 9 (1 – 10 = 9)

IV = 4 (1 – 5 = 4)

### Exercice

En utilisant

L'affectation ( $\leftarrow$ ),

Si... alors

Tant que... faire

Les opérateurs arithmétiques de base (+, -, \*, /, et %)

***Développez un algorithme en pseudo-code qui traduit un nombre écrit en nombre romain en nombre écrit en décimal.***

$d \leftarrow 0$

$i \leftarrow 1$

tant que  $i \leq \text{longueur de } r$

$c \leftarrow i^{\text{e}}$  caractère de  $r$

Si  $c = X$  alors

$d \leftarrow d + 10$

Si  $c = V$  alors

$d \leftarrow d + 5$

Si  $c = I$  alors

Si le caractère à la position  $i + 1$  est  $X$

$d \leftarrow d + 9$

$i \leftarrow i + 1$

Si le caractère à la position  $i + 1$  est  $V$

$d \leftarrow d + 4$

$i \leftarrow i + 1$

Si le caractère à la position  $i + 1$  n'est ni  $X$  ni  $V$

$d \leftarrow d + 1$

$i \leftarrow i + 1$

retourne  $d$

### Operation de base

- Calculez la negation

De 88916  $\rightarrow$  - 88916

De 7  $\rightarrow$  -7

- Calculez le produit

De 5 et 6  $\rightarrow$  30

De 88416 et 7  $\rightarrow$  622412

Montreal, 31/10/2018

**Pile d'exécution** : Contient l'état de l'exécution du programme.

**Au début** : 1 assiette le Main.

**À chaque appel de fonction** : une empile (push)

**Une nouvelle assiette** : nom de la fonction – ligne où de pointeur d'exécution est rendu.  
Variables locales et leur valeur.

**À chaque return** : un dépile (pop)  $\rightarrow$  ou fermeture de l'accolade finale de la fonction. Le contenu de L'assiette est perdu.

Le pointeur d'exécution se retrouve à l'assiette d'en-dessous on continue l'exécution à la ligne indiquée.

À la fin du main, la pile est vide, l'exécution arrête. Chaque assiette à sa portée.

```
=====
                                Code
=====
void inverser (int*a, int* b)
    int t = *b ;
    *b = *a ;
    *a = t ;
=====
void ordonner (Jet j) {
    if (jet.des[0] > jet.des[1])
        inverser(&jet.des[0], &jet.des[1]) ;
    if (jet.des[1] > jet.des[2])
        inverser(&jet.des[1], &jet.des[2]);
    if (jet.des[0] > jet.des[1])
        inverser(&jet.des[0], &jet.des[1]) ;
}
=====
int main() {
    Jet j ;
    j.des = {1,2,3} ;
    ordonner(j) ;
}
=====
```

**Pile** = Stack

**Tas** = Heap

Montreal, 05/11/2018

**La Pile** : L'ordre d'exécution des fonctions.

- Les variables locales sont formelles.
- `int* p = &TAILLE ;`
- Mémoire libérée au return



**Le Tas** : mémoire dynamique c'est là que vont les valeurs des variables créées avec malloc.

- Memoire liberee au free
- Un peu plus lent

```
***** Un tableau de 8 float *****
- Pile : float tab[8] ;
        float tab[] = {0,0,0,0,0,0,0,0} ;

- Tas : float* tab = (float*) ;
        malloc (sizeof(float)*8) ;

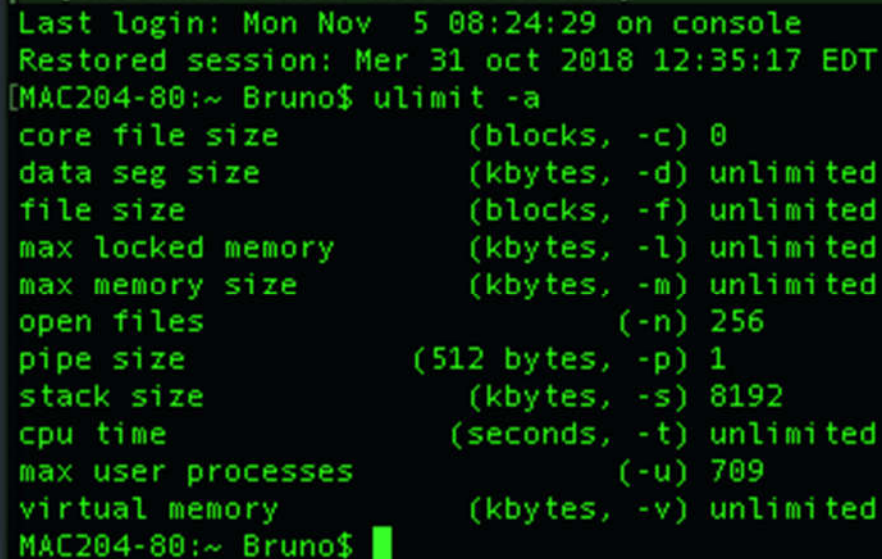
        float* tab = (float*)
        calloc(8, sizeof(float)) ;

        float* t2 = (float*) ;
        realloc (tab, 12*sizeof(float) ;

        free (t2) ;
*****
```

**La mémoire statique** : Le code les globales. Sa taille est fixée du début à la fin.

**Comando para verificar memoria** : ulimit -a

A terminal window with a black background and green text. The text shows the output of the 'ulimit -a' command. At the top, it says 'Last login: Mon Nov 5 08:24:29 on console' and 'Restored session: Mer 31 oct 2018 12:35:17 EDT'. Below that, the prompt is '[MAC204-80:~ Bruno\$ ulimit -a'. The output lists various system limits: core file size (0), data seg size (unlimited), file size (unlimited), max locked memory (unlimited), max memory size (unlimited), open files (256), pipe size (1), stack size (8192), cpu time (unlimited), max user processes (709), and virtual memory (unlimited). The prompt ends with 'MAC204-80:~ Bruno\$' followed by a green cursor.

```
Last login: Mon Nov 5 08:24:29 on console
Restored session: Mer 31 oct 2018 12:35:17 EDT
[MAC204-80:~ Bruno$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files              (-n) 256
pipe size               (512 bytes, -p) 1
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 709
virtual memory           (kbytes, -v) unlimited
MAC204-80:~ Bruno$
```

Montreal, 09/11/2018.

```
=====
Double* tab = (double*) malloc (sizeof(double)* 10) ;
For(int i = 0 ; i<10 ; i++)
    Tab[i] = 0 ;
//...
//...
Tab = (double*) realloc(tab, sizeof(double)*15) ;
//...
Free (tab) ;
=====
```

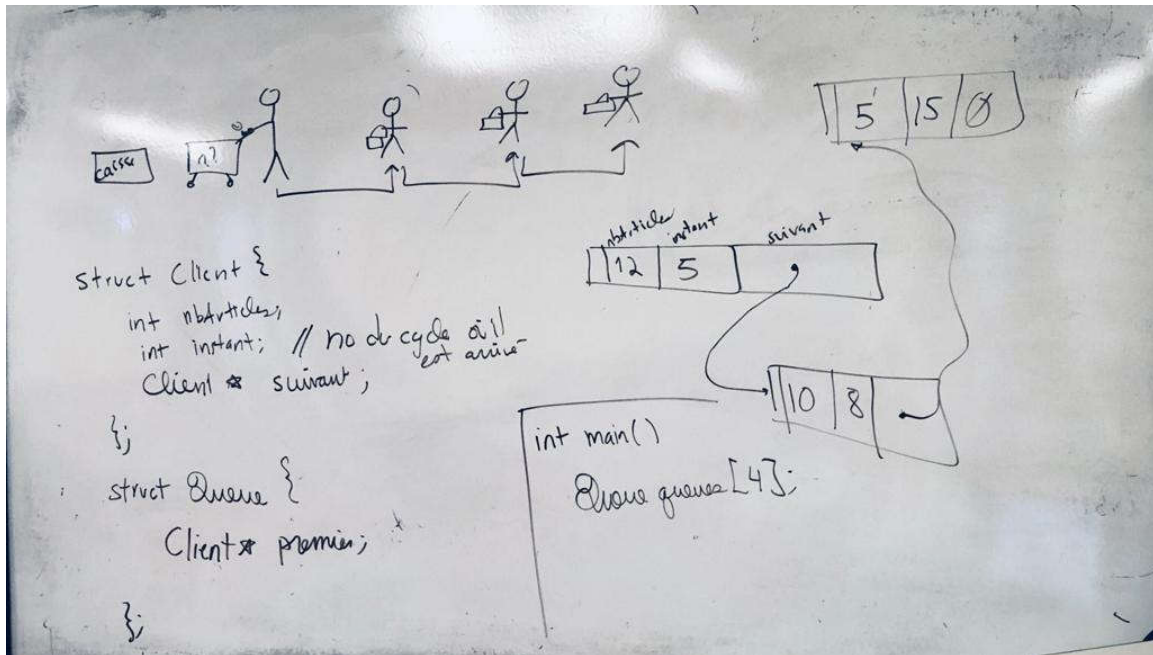
**Obs : dans le Java :**

```
Int[] tab = new int [10]
String a = new String()
```

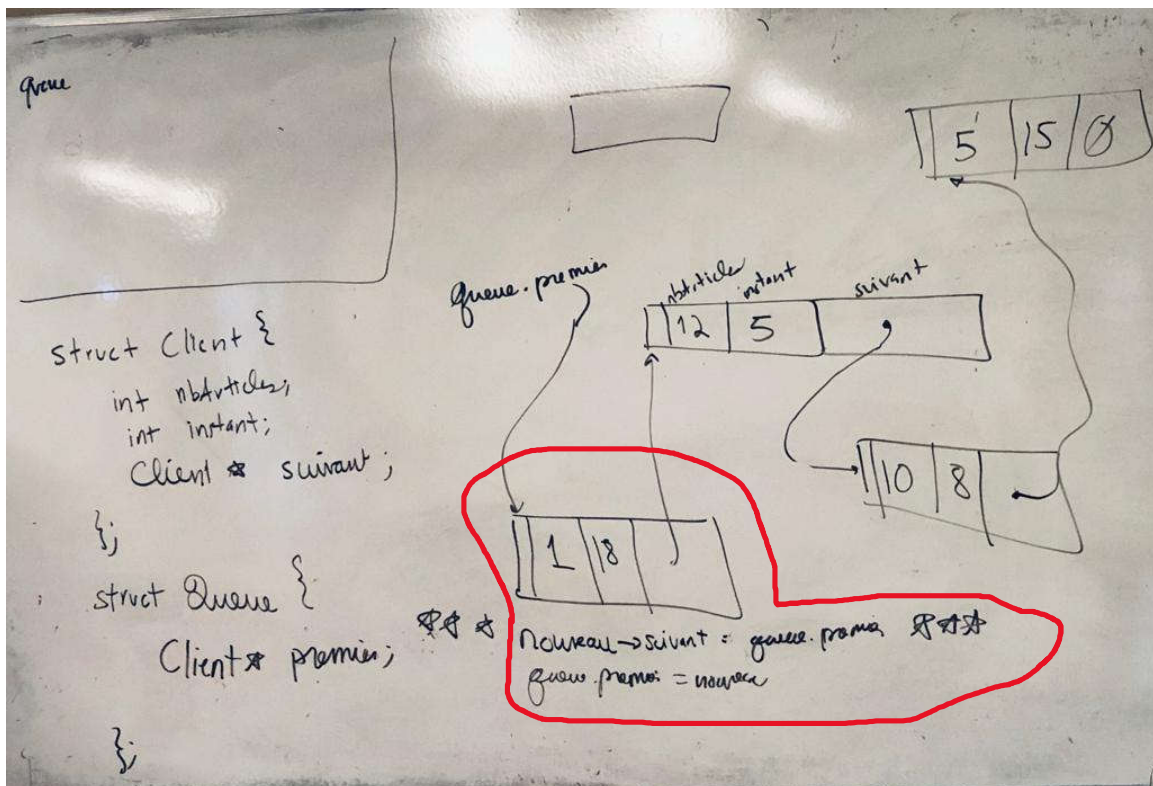
**Code par TP1**

```
Struct client {
    Int nbArticles ;
    Int instant ;
    Client* suivant ;
};
```

```
Struct Queue {
    Client* premier;
};
```



Une observation.



```

queue
Client* c = queue.premier;
while (c != NULL) {
    printf("%d", c->nbArticles);
    c = c->suivant;
}

```

```

struct Client {

```

```

for (Client* c = queue.premier; c != NULL; c = c->suivant)
    printf("%d", c->nbArticles, c->instant);

```

premier

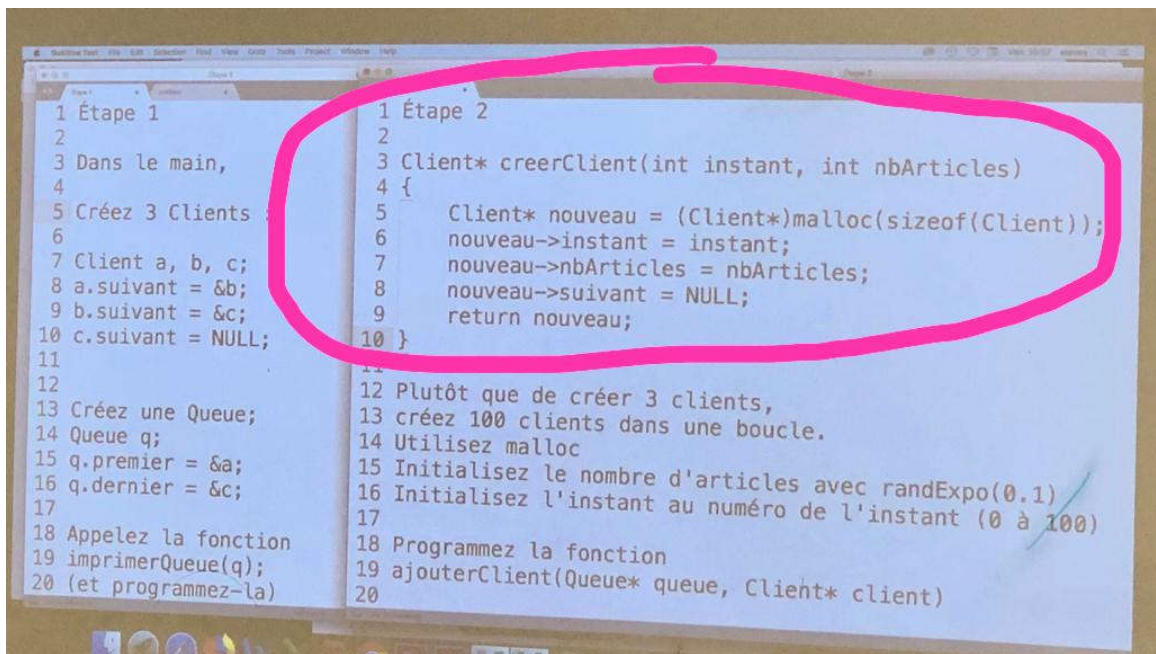
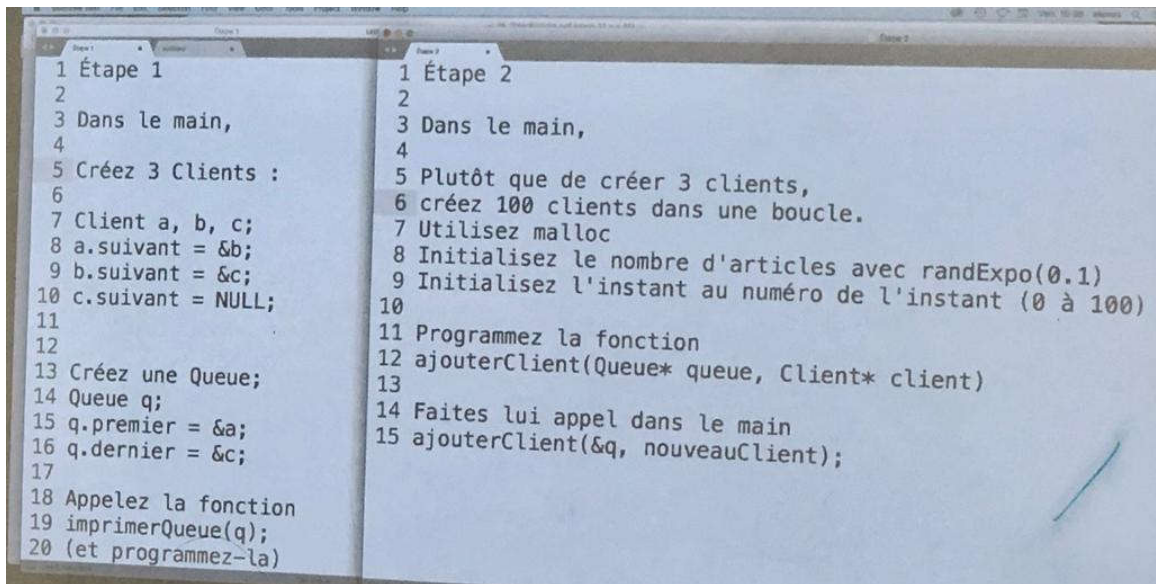
1	18
---	----

```

Client* nouveau = malloc(sizeof(Client));
nouveau->suivant = NULL;
queue.dernier->suivant = nouveau;
queue.dernier = nouveau;

```





## Important

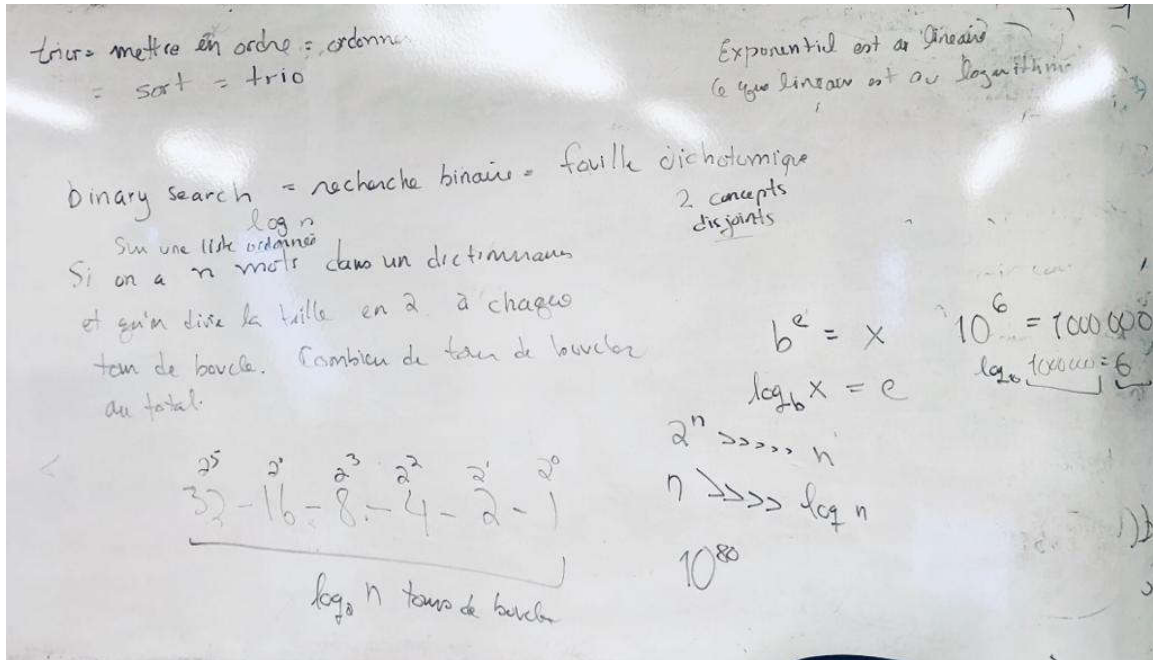
```
instant ← 0
tant que instant < 200
  de ← tirage au hasard entre 0 et 5
  Si de = 0
    client ← creerClient(instant, randExpo(0.1))
    Soit q = la queue la plus courte dans les 4 queues
    offrirClient(&q, client)
  Pour chaque queue dans queues
    client ← coupDeilTete(queue)
    client → nbArticles --
    Si client → nbArticles = 0
      obtenirTete(queue)
      libérer la machine utilisée par client
  imprimer queues
  sleep(1);

instant ← instant + 1
imprimer les statistiques
```

Montreal, 13/11/2018.

Trier = mettre en ordre = ordonner = sort = trio

Binary search = recherche binaire = fouille dichotomique (2 concepts, disjoints)



Montreal, 14/11/2018

\*\*\*\*\*

Int factorielle (int n)

```
{  
    If (n == 1)  
        Return 1;  
    Return n * factorielle (n - 1);  
}
```

\*\*\*\*\*

Int pgcd (int a, int b)

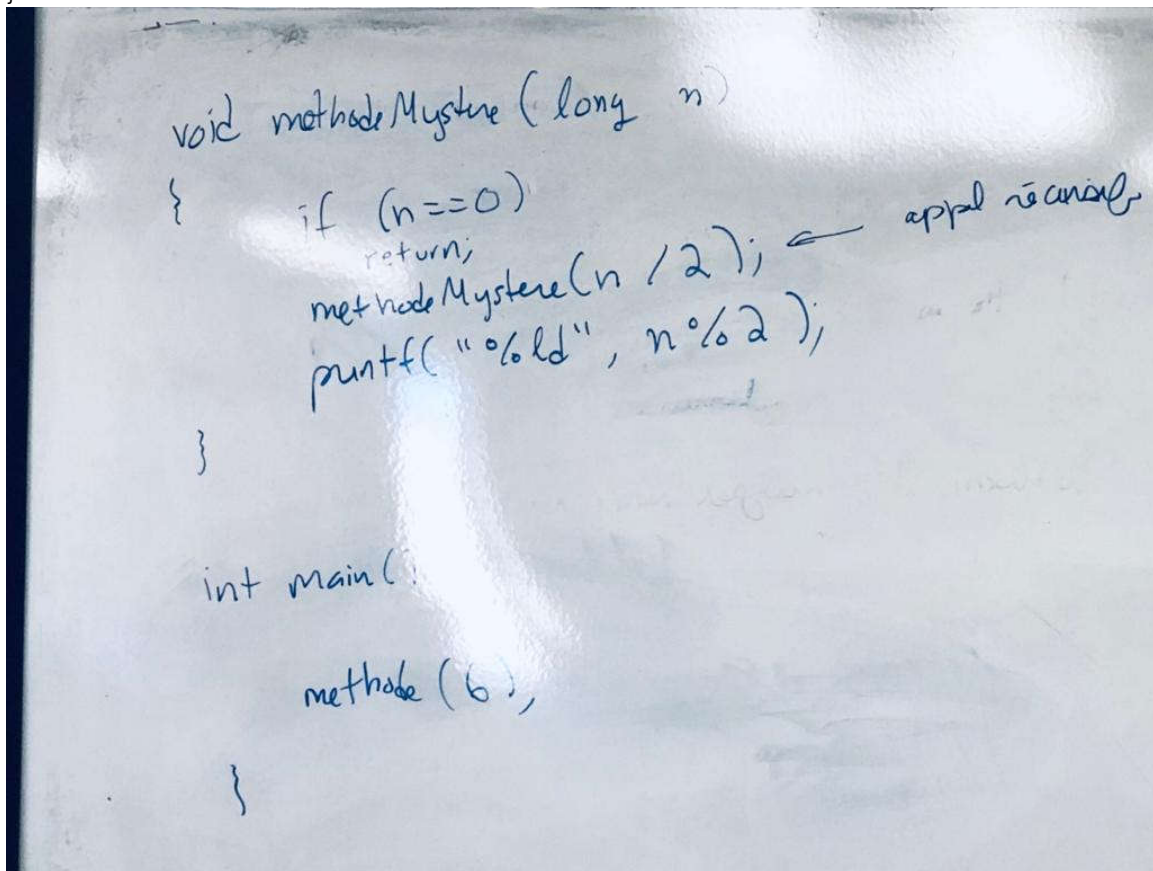
```
{  
    Return (b == 0 ? a : pgcd(  
}
```

.....

```
Void methodeMystere (long n)
{
    If (n ==0)
        Return;
    methodeMystere(n/2);
    printf(« %ld « , n%2);
}
```

---

```
Int main ()
{
    Methode (9);
}
```



The image shows a handwritten version of the C code from the previous blocks. The function 'methodeMystere' is defined with a 'void' return type and a 'long n' parameter. It uses an 'if' statement to check for the base case 'n == 0', followed by a 'return;' statement. The recursive call 'methodeMystere(n / 2);' is annotated with an arrow pointing to it and the text 'appel récursif'. The 'printf' statement uses the format string '"%ld"' and the expression 'n % 2'. The 'main' function is declared as 'int main()' and calls 'methode (9);'.

```
void methodeMystere (long n)
{
    if (n == 0)
        return;
    methodeMystere(n / 2); ← appel récursif
    printf( "%ld", n % 2 );
}

int main()
{
    methode (9);
}
```



```

void methodeMystere (long n)
{
    if (n == 0) → condition de sortie
        return;
    methodeMystere(n / 2); ← appel récursif
    printf("%ld", n % 2); ← fait ce qui manque pour
                          passer d'une solution à n/2
                          à n
}

```

Les trois éléments qu'une fonction récursive doit avoir

1. Une condition de sortie (cas de base pour lequel il n'y a pas de récursion)
2. Un appel récursif (on appelle la fonction sur une instance plus petite / sur un sous-problème)
3. Un calcul de plus (ce qu'il faut effectuer comme calcul pour passer de la solution au sous-problème à la solution générale.)

```

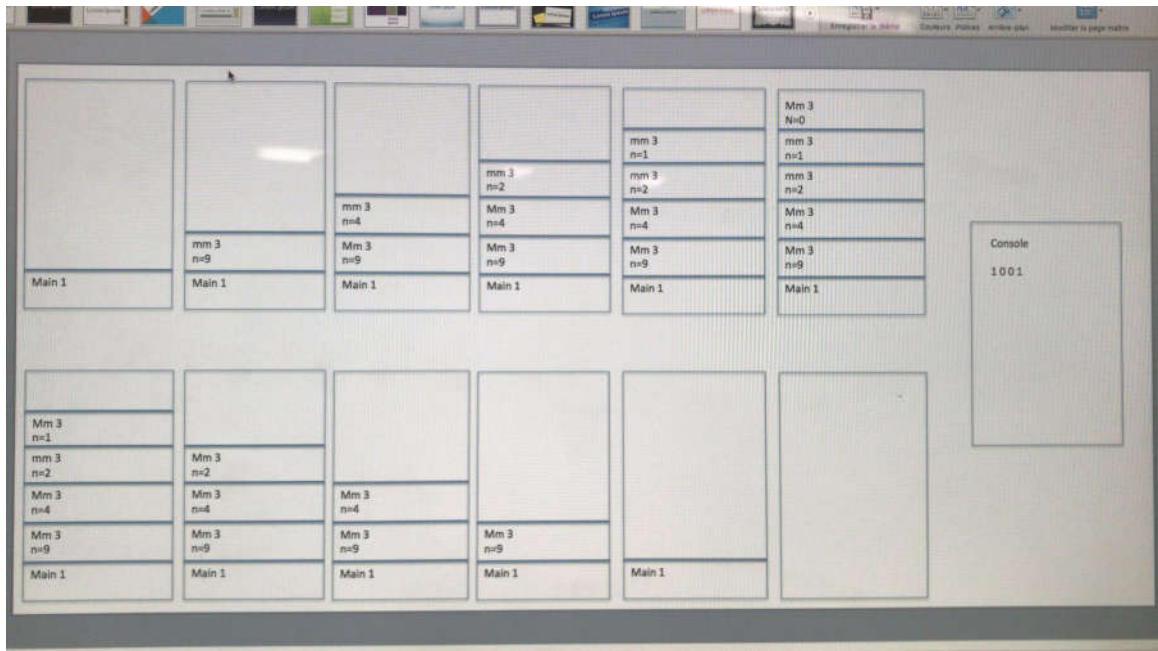
void f (int n)
{
    if (n == 1)
        return 1;
    return n * f(n-1);
}

```

calcul de plus

// Factorielle  
 // condition de sortie  $n=1$   
 Pour calculer factorielle de  $n$ ,  
 on commence par trouver factorielle de  $n-1$   
 et on le multiplie par  $n$ .  

$$n! = n * (n-1)!$$



## Survol de la matière

### Qu'est-ce qu'une opération de base?

*C'est une opération qui prend un temps constant, comme diviser et multiplier par 10, par exemple.*

### Qu'est-ce qu'un algorithme correct? Efficace?

*Un algorithme est correct s'il fait ce qu'il est censé de faire, il est exact s'il donne la bonne sortie pour toute les entrées possibles (il ne se trompe jamais!). Il est efficace s'exécute en peu d'opérations basiques.*

### Quelles sont les trois sections de mémoire pour les exécutables? Quels sont leur rôle, leur contenu? Quand sont-elles libérées? Déclarez des variables dans chaque section.

**La pile d'exécution** : c'est une structure de données qui suit le principe dernier arrivé premier servi. L'exécution se fait à la fois avec un pointeur d'exécution. La pile contient l'état d'exécution du programme. Les variables dans la pile sont libérées soit au return ou à la fin du programme.  
Ex de déclaration : `int i = 0;` (variables locales/formelles)

**Le tas** : n'a pas une structure particulière, sa taille s'ajuste au cours de l'exécution et la mémoire est dynamique. Il permet au programmeur de contrôler la mémoire. Les variables dans le tas sont libérées quand on appelle le `free()` pour la variable ou à la fin du programme.  
Ex de déclaration : `float* tab = (float*) malloc(sizeof(float) * 8);` pour créer un tableau de 8 cases  
`float* tab = (float*) calloc(8, sizeof(float));` pour créer un tableau de 8 cases initialisé à 0; (malloc/calloc)

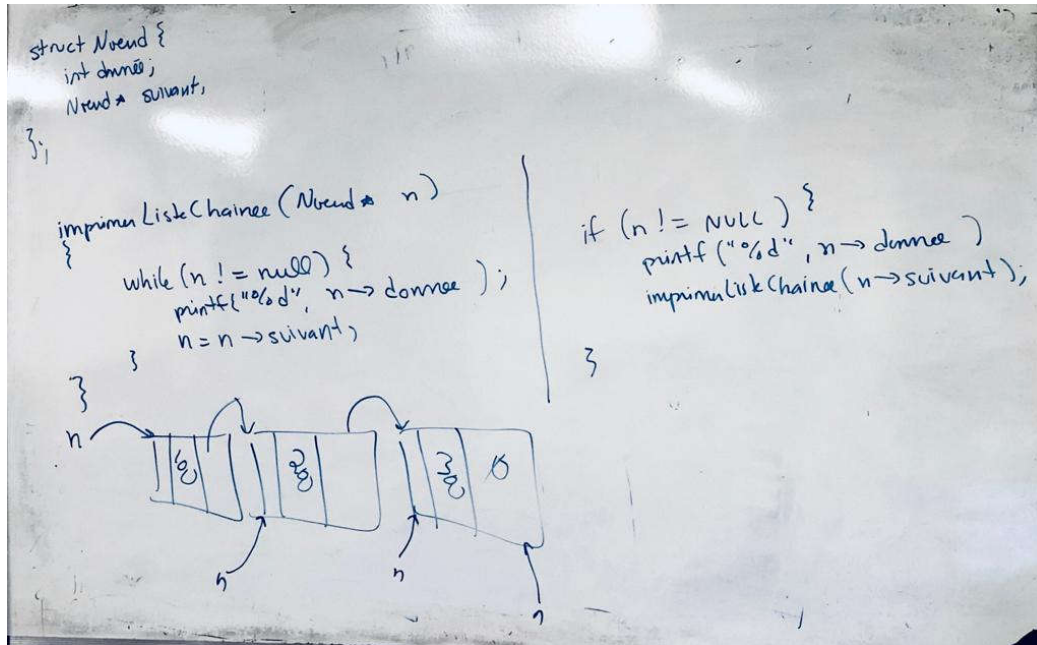
**Le data** : est rapide, compacte, statique et sa taille est fixée à la compilation. Contient, entre autres, le code, les instructions et les variables globales. La mémoire est libérée à la fin du programme.

### Qu'est-ce qu'une pile? Une file? Une liste chaînée? Un tableau dynamique?

**Une pile (stack)** c'est une structure de données qui suit le principe premier arrivé, premier servi. L'exécution se fait à la fois avec un pointeur d'exécution. La pile contient l'état d'exécution du programme.

**En C, une file est une liste chaînée** où chaque élément pointe vers le suivant. Le dernier élément de la file pointe vers NULL (il n'a pas de suivant).

Une liste chaînée est une structure ordonnée qui représente une collection ordonnée de taille variable d'éléments de même type, dont chaque élément pointe vers son suivant. Si la liste est doublement chaînée, chaque élément pointe vers son suivant et son précédent.



**Un tableau dynamique** est un tableau qui on déclare avec un malloc/calloc dans le tas et on peut changer sa taille pendant l'exécution du programme avec un realloc pour qu'il soit plus grand ou plus petit au besoin.

### Comment ajoute/retire-t-on un élément dans un tableau dynamique? Une liste chaînée?

Pour ajouter ou retirer des éléments dans un tableau dynamique, on utilise realloc.

Dans une liste chaînée, on peut ajouter, lire, modifier et supprimer des nœuds à condition qu'on connaisse l'adresse du premier nœud. L'ajoute et la suppression autour d'une adresse connue sont optimales, il faut juste changer la référence pour le suivant->suivant ou NULL.

### Appliquez les quatre algorithmes de tri (sélection/insertion/bulle/fusion) sur un tableau. Quelles sont les forces et les faiblesses de chacun?

```

void echanger (int*a, int* b)
{
    int temporaire;
    temporaire = *a;
    *a = *b;
    *b = temporaire;
}

tab[] = {'P', 'L', 'O', 'B', 'V', 'X', 'B'};
    
```

**Tri sélection:** on trouve le plus petit, puis le deuxième plus petit et ainsi de suite. Pas beaucoup d'échange, pratique sur les listes chaînées.

```
void triSelection (int* tab, int n)
{
    int min;
    for (int i = 0; i < n - 1; i++)
    {
        min = i;
        for (int j = i + 1; j < n; j++)
        {
            if (tab[min] > tab[j])
            {
                min = j;
            }
        }
        echanger (&tab[i], &tab[min]);
    }
}
```

**Tri insertion :** on prendre un élément à la fois et on l'insère au bon endroit dans le tableau. Efficace sur les petits tableaux, les tableaux presque en ordre, en ligne.

```
void triInsertion (int* tab, int n)
{
    for (int i = 1; i < n; i++)
    {
        for (int j = i; j > 0 && tab[j - 1] > tab [j]; j--)
        {
            echanger (&tab[j], &tab[j - 1]);
        }
    }
}
```

**Tri bulle** : les petites éléments percolent, les gros éléments calent. On arrête quand rien n'a bougé.  
Efficace pour les tableaux presque en ordre, mais en moyenne très lent.

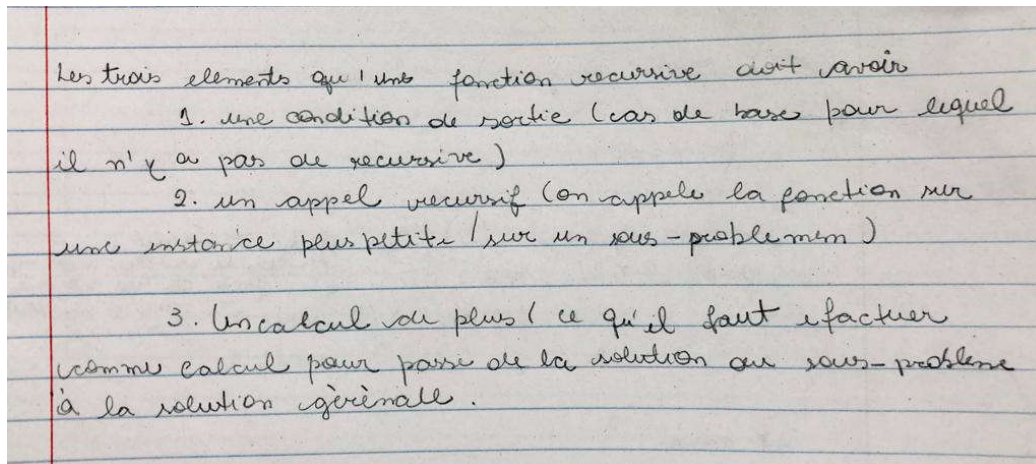
```
void triBulle (int* tab, int n)
{
    int temoin = 1;
    int f = n;
    while (temoin)
    {
        temoin = 0;
        for (int i = 1; i < f; i++)
        {
            if (tab[i - 1] > tab[i])
            {
                echanger (&tab[i - 1], &tab[i]);
                temoin = 1;
            }
        }
        f--;
    }
}
```

**Tri fusion** : fait encore moins d'opérations en utilisant un algorithme récursif. Le tri fusion est un algorithme récursif qui décompose une liste en deux sous-listes, trie chacune d'elle et les fusionne.

```
int triFusion (int tab, int n)
{
    if (n <= 1)
    {
        return n;
    }
    int millieu = n / 2;
    int* tabM[millieu - 1];
    int* tabN[n - 1];
    return tab;
}
```







#### Algorithme récursif :

void fonctionMystere (long n)

```
{  
    if (n == 0) // n == 0 est l'element Condition de Sortie  
        return;  
    fonctionMystere (n / 2); // fonctionMystere (n/2) est l'element Appel Récursif  
    printf ("%ld", n % 2); // est l'element Calcul de Plus  
}
```

#### Algorithme itératif (non récursif) :

void fonctionIterative (long n)

```
{  
    if (n == 0)  
        return;  
    const long N = n;  
    int longueur = 1;  
    while (n / 2 > 0)  
    {  
        longueur++;  
        n = n / 2;  
    }  
  
    n = N;  
    int envers[longueur];  
    int i = 0;  
  
    while (n / 2 > 0)  
    {  
        envers[i] = n % 2;  
        n = n / 2;
```



```

        i++;
    }
    envers[i] = n % 2;

    for ( i = longueur - 1; i >= 0; i--)
        printf("%d", envers[i]);
}

```

**Qu'est-ce qu'un arbre, une racine, une feuille, un nœud? Qu'est-ce la profondeur? Qu'est-ce qu'un arbre binaire? De recherche? Appliquez le parcours en profondeur sur un arbre.**

**Un arbre est** un ensemble de nœuds avec un nœud spécial appelé racine, chaque nœud a 0, 1 ou plusieurs enfants et chaque nœud a exactement un parent (sauf la racine, qui en a aucun).

La racine est un nœud spécial, avec aucun parent et 0, 1 ou plusieurs enfants qui est à la origine de l'arbre.

**Une feuille** est un nœud qui n'a pas d'enfants.

**Un nœud** est un élément de l'arbre (soit la racine, soit un enfant/parent).

**Le parcours en profondeur** est parcourir l'arbre la racine en premier, puis son enfant gauche, puis son enfant droit, récursivement.

**Un arbre binaire est** un arbre où chaque parent a 0, 1 ou 2 enfants.

**Les arbres de recherche sont** des structures dynamiques optimisées pour la recherche, chaque nœud est entre son enfant gauche et son enfant droit.

**void parcoursProfondeur(Noeud\* racine)**

```

{
    if (racine != NULL)
    {
        printf("%c", racine->donnee);
        parcoursProfondeur (racine->g);
        parcoursProfondeur(racine->d);
    }
}

```

## nbNoeuds

```

int nbNoeudsVal (Noeud * n, int val)
{
    if (n == NULL) ← condition de sortie
        return 0;
    return nbNoeudsVal(n->suivant, val) + (n->donnee == val);
    // nb de noeuds qui valent val après
    // 1 si n vaut val, 0 sinon
}

```

11	1	15	6
5	7	9	10
8	3	6	12
2	4	0	13

→ 1/6 fa

```

int nbNoeudsValen (Noeud * n, int val)
{
    int compteur = 0;
    while (n != NULL) {
        compteur += (n->donnee == val);
        n = n->suivant;
    }
    return compteur;
}

```

```

int nbNoeuds (Noeud * n)
{
    if (n == NULL) ← condition de sortie
        return 0;
    return nbNoeuds(n->g) + nbNoeuds(n->d) + 1;
    // nb de noeud dans le sous arbre gauche
    // droit
    // compte n
}

```

11	1	15	6
5	7	9	10
8	3	6	12
2	4	0	13

→ 1/6 fa

```

int nbNoeudsValen (Noeud * n, int val)
{
    int compteur = 0;
    while (n != NULL) {
        compteur += (n->donnee == val);
        n = n->suivant;
    }
    return compteur;
}

```



