# Récursivité

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

# Multiplication du paysan russe

***function*** MULTIPLY$(x, y)$
  $p \leftarrow 0$
  ***while*** $x > 0$ ***do***
    ***if*** $x$ *est impair* ***then***
      $p \leftarrow p + y$
    $x \leftarrow \lfloor x/2 \rfloor$
    $y \leftarrow y + y$
  ***return*** $p$

| $x$ | $y$ | $p$ |
|---|---|---|
| 105 | 253 | 0 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# Multiplication du paysan russe

**function** MULTIPLY$(x, y)$
    $p \leftarrow 0$
    **while** $x > 0$ **do**
        **if** $x$ est impair **then**
            $p \leftarrow p + y$
        $x \leftarrow \lfloor x/2 \rfloor$
        $y \leftarrow y + y$
    **return** $p$

| $x$ | $y$ | $p$ |
|---|---|---|
| 105 | 253 | 0 |
| 52 | 506 | 253 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# Multiplication du paysan russe

**function** MULTIPLY$(x, y)$
  $p \leftarrow 0$
  **while** $x > 0$ **do**
    **if** $x$ *est impair* **then**
      $p \leftarrow p + y$
    $x \leftarrow \lfloor x/2 \rfloor$
    $y \leftarrow y + y$
  **return** $p$

| $x$ | $y$ | $p$ |
|---|---|---|
| 105 | 253 | 0 |
| 52 | 506 | 253 |
| 26 | 1012 | 253 |
| | | |
| | | |
| | | |
| | | |
| | | |

# Multiplication du paysan russe

***function*** MULTIPLY$(x, y)$
    $p \leftarrow 0$
    ***while*** $x > 0$ ***do***
        ***if*** $x$ *est impair* ***then***
            $p \leftarrow p + y$
        $x \leftarrow \lfloor x/2 \rfloor$
        $y \leftarrow y + y$
    ***return*** $p$

| $x$ | $y$ | $p$ |
|---|---|---|
| 105 | 253 | 0 |
| 52 | 506 | 253 |
| 26 | 1012 | 253 |
| 13 | 2024 | 253 |
| | | |
| | | |
| | | |
| | | |

# Multiplication du paysan russe

***function*** MULTIPLY$(x, y)$
  $p \leftarrow 0$
  ***while*** $x > 0$ ***do***
    ***if*** $x$ *est impair* ***then***
      $p \leftarrow p + y$
    $x \leftarrow \lfloor x/2 \rfloor$
    $y \leftarrow y + y$
  ***return*** $p$

| $x$ | $y$ | $p$ |
|---|---|---|
| 105 | 253 | 0 |
| 52 | 506 | 253 |
| 26 | 1012 | 253 |
| 13 | 2024 | 253 |
| 6 | 4048 | 2277 |
| | | |
| | | |
| | | |

# Multiplication du paysan russe

***function*** MULTIPLY$(x, y)$
  $p \leftarrow 0$
  ***while*** $x > 0$ ***do***
    ***if*** $x$ *est impair* ***then***
      $p \leftarrow p + y$
    $x \leftarrow \lfloor x/2 \rfloor$
    $y \leftarrow y + y$
  ***return*** $p$

| $x$ | $y$ | $p$ |
|---|---|---|
| 105 | 253 | 0 |
| 52 | 506 | 253 |
| 26 | 1012 | 253 |
| 13 | 2024 | 253 |
| 6 | 4048 | 2277 |
| 3 | 8096 | 2277 |
| | | |
| | | |

# Multiplication du paysan russe

**function** MULTIPLY$(x, y)$
  $p \leftarrow 0$
  **while** $x > 0$ **do**
    **if** $x$ est impair **then**
      $p \leftarrow p + y$
    $x \leftarrow \lfloor x/2 \rfloor$
    $y \leftarrow y + y$
  **return** $p$

| $x$ | $y$ | $p$ |
|---|---|---|
| 105 | 253 | 0 |
| 52 | 506 | 253 |
| 26 | 1012 | 253 |
| 13 | 2024 | 253 |
| 6 | 4048 | 2277 |
| 3 | 8096 | 2277 |
| 1 | 16192 | 10373 |
| | | |

# Multiplication du paysan russe

**function** MULTIPLY$(x, y)$
  $p \leftarrow 0$
  **while** $x > 0$ **do**
    **if** $x$ *est impair* **then**
      $p \leftarrow p + y$
    $x \leftarrow \lfloor x/2 \rfloor$
    $y \leftarrow y + y$
  **return** $p$

| $x$ | $y$ | $p$ |
|---|---|---|
| 105 | 253 | 0 |
| 52 | 506 | 253 |
| 26 | 1012 | 253 |
| 13 | 2024 | 253 |
| 6 | 4048 | 2277 |
| 3 | 8096 | 2277 |
| 1 | 16192 | 10373 |
| 0 | 32384 | 26565 |

# Multiplication du paysan russe

***function*** MULTIPLY$(x, y)$
  $p \leftarrow 0$
  ***while*** $x > 0$ ***do***
    ***if*** $x$ *est impair* ***then***
      $p \leftarrow p + y$
    $x \leftarrow \lfloor x/2 \rfloor$
    $y \leftarrow y + y$
  ***return*** $p$

| $x$ | $y$ | $p$ |
|---|---|---|
| 105 | 253 | 0 |
| 52 | 506 | 253 |
| 26 | 1012 | 253 |
| 13 | 2024 | 253 |
| 6 | 4048 | 2277 |
| 3 | 8096 | 2277 |
| 1 | 16192 | 10373 |
| 0 | 32384 | 26565 |

Cet algorithme réalise l'itération des suites $p$, $x$ et $y$ définies par :

$$p_0 = 0 \qquad x_0 = x \qquad y_0 = y$$

et :

$$p_{n+1} = \begin{cases} p_n + y_n & \text{si } x_n \text{ est impair} \\ p_n & \text{sinon} \end{cases} \qquad x_{n+1} = \lfloor x_n/2 \rfloor \qquad y_{n+1} = 2y_n.$$

# Multiplication du paysan russe

$$p_0 = 0 \qquad x_0 = x \qquad y_0 = y$$

$$p_{n+1} = \begin{cases} p_n + y_n & \text{si } x_n \text{ est impair} \\ p_n & \text{sinon} \end{cases} \qquad x_{n+1} = \lfloor x_n/2 \rfloor \qquad y_{n+1} = 2y_n.$$

# Multiplication du paysan russe

$$p_0 = 0 \qquad x_0 = x \qquad y_0 = y$$

$$p_{n+1} = \begin{cases} p_n + y_n & \text{si } x_n \text{ est impair} \\ p_n & \text{sinon} \end{cases} \qquad x_{n+1} = \lfloor x_n/2 \rfloor \qquad y_{n+1} = 2y_n.$$

On pose $x = (b_k b_{k-1} \cdots b_1 b_0)_2$ avec $b_k = 1$. Alors :

$$x_n = (b_k b_{k-1} \cdots b_n)_2 \quad \text{et} \quad y_n = 2^n y.$$

# Multiplication du paysan russe

$$p_0 = 0 \qquad x_0 = x \qquad y_0 = y$$

$$p_{n+1} = \begin{cases} p_n + y_n & \text{si } x_n \text{ est impair} \\ p_n & \text{sinon} \end{cases} \qquad x_{n+1} = \lfloor x_n/2 \rfloor \qquad y_{n+1} = 2y_n.$$

On pose $x = (b_k b_{k-1} \cdots b_1 b_0)_2$ avec $b_k = 1$. Alors :

$$x_n = (b_k b_{k-1} \cdots b_n)_2 \quad \text{et} \quad y_n = 2^n y.$$

Nous avons $x_k = b_k = 1 \neq 0$ et $x_{k+1} = 0$, ce qui prouve la terminaison de l'algorithme.

# Multiplication du paysan russe

$$p_0 = 0 \qquad x_0 = x \qquad y_0 = y$$

$$p_{n+1} = \begin{cases} p_n + y_n & \text{si } x_n \text{ est impair} \\ p_n & \text{sinon} \end{cases} \qquad x_{n+1} = \lfloor x_n/2 \rfloor \qquad y_{n+1} = 2y_n.$$

On pose $x = (b_k b_{k-1} \cdots b_1 b_0)_2$ avec $b_k = 1$. Alors :

$$x_n = (b_k b_{k-1} \cdots b_n)_2 \quad \text{et} \quad y_n = 2^n y.$$

Nous avons $x_k = b_k = 1 \neq 0$ et $x_{k+1} = 0$, ce qui prouve la terminaison de l'algorithme.
Par ailleurs, $p_{n+1} = p_n + y_n \times (x_n \bmod 2) = p_n + b_n y_n$ donc :

$$p_{k+1} = p_0 + \sum_{n=0}^{k} b_k (2^k y) = \Big(\sum_{n=0}^{k} b_k 2^k\Big) y = xy$$

ce qui prouve sa validité.

# Multiplication du paysan russe

## Version récursive

Mise en œuvre pratique en PYTHON :

```python
def multiply(x, y):
    p = 0
    while x > 0:
        if x % 2 == 1:
            p += y
        x = x // 2
        y = y + y
    return p
```

# Multiplication du paysan russe

## Version récursive

Mise en œuvre pratique en PYTHON :

```
def multiply(x, y):
    p = 0
    while x > 0:
        if x % 2 == 1:
            p += y
        x = x // 2
        y = y + y
    return p
```

Cet algorithme repose sur les relations :

$$x \cdot y = \begin{cases} 0 & \text{si } x = 0 \\ \lfloor x/2 \rfloor \cdot (y+y) & \text{si } x \text{ est pair} \\ \lfloor x/2 \rfloor \cdot (y+y) + y & \text{si } x \text{ est impair} \end{cases}$$

→ le calcul du produit de $x$ par $y$ se ramène au produit de $\lfloor x/2 \rfloor$ par $2y$.

# Multiplication du paysan russe

## Version récursive

Version récursive du même algorithme :

```
def multiply(x, y):
    if x <= 0:
        return 0
    elif x % 2 == 0:
        return multiply(x//2, y+y)
    else:
        return multiply(x//2, y+y) + y
```

Cet algorithme repose sur les relations :

$$x \cdot y = \begin{cases} 0 & \text{si } x = 0 \\ \lfloor x/2 \rfloor \cdot (y+y) & \text{si } x \text{ est pair} \\ \lfloor x/2 \rfloor \cdot (y+y) + y & \text{si } x \text{ est impair} \end{cases}$$

$\rightarrow$ le calcul du produit de $x$ par $y$ se ramène au produit de $\lfloor x/2 \rfloor$ par $2y$. Python permet une telle réduction du problème.

# Le problème de la terminaison

Pour qu'une fonction récursive se termine :

- il est nécessaire qu'il y ait une condition d'arrêt ;
- il ne doit pas y avoir de suite infinie d'appels récursifs.

# Le problème de la terminaison

Pour qu'une fonction récursive se termine :

- il est nécessaire qu'il y ait une condition d'arrêt ;
- il ne doit pas y avoir de suite infinie d'appels récursifs.

Cas de la multiplication du paysan russe :

- condition d'arrêt : $x \leqslant 0$ ;
- nombre d'appels récursifs : il y en a $\lfloor \log x \rfloor + 1$.

On procède le plus souvent par récurrence pour prouver la terminaison et la validité.

# Le problème de la terminaison

Pour qu'une fonction récursive se termine :

- il est nécessaire qu'il y ait une condition d'arrêt ;
- il ne doit pas y avoir de suite infinie d'appels récursifs.

Cas de la multiplication du paysan russe :

- condition d'arrêt : $x \leqslant 0$ ;
- nombre d'appels récursifs : il y en a $\lfloor \log x \rfloor + 1$.

On procède le plus souvent par récurrence pour prouver la terminaison et la validité.
En PYTHON le nombre d'appels récursifs est majoré :

```
>>> def f(n):
        return 1+f(n+1)

>>> f(0)
RuntimeError: maximum recursion depth exceeded
```

# Le problème de la terminaison

Pour qu'une fonction récursive se termine :

- il est nécessaire qu'il y ait une condition d'arrêt ;
- il ne doit pas y avoir de suite infinie d'appels récursifs.

Cas de la multiplication du paysan russe :

- condition d'arrêt : $x \leqslant 0$ ;
- nombre d'appels récursifs : il y en a $\lfloor \log x \rfloor + 1$.

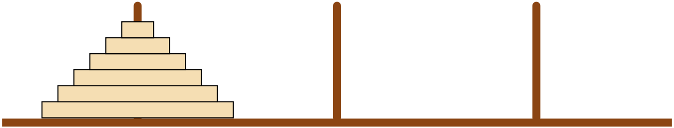
On procède le plus souvent par récurrence pour prouver la terminaison et la validité.
Il est parfois très difficile de prouver la terminaison :

```
def q(n):
    if n <= 2:
        return 1
    return q(n-q(n-1)) + q(n-q(n-2))
```

On ne sait pas si la fonction Q de Hofstadter se termine pour toute valeur de $n$.

# Les tours de Hanoï

## Un problème emblématique de la programmation récursive



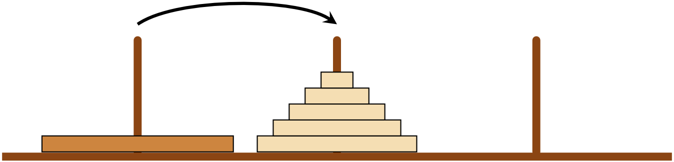
Trois tiges et $n$ disques doivent respecter les règles suivantes :

- un seul disque peut être déplacé à la fois ;
- on ne peut jamais poser un disque sur un disque de diamètre inférieur.

Comment déplacer les $n$ disques de la tige 1 à la tige 3 ?

# Les tours de Hanoï

## Un problème emblématique de la programmation récursive



Trois tiges et $n$ disques doivent respecter les règles suivantes :

- un seul disque peut être déplacé à la fois ;
- on ne peut jamais poser un disque sur un disque de diamètre inférieur.

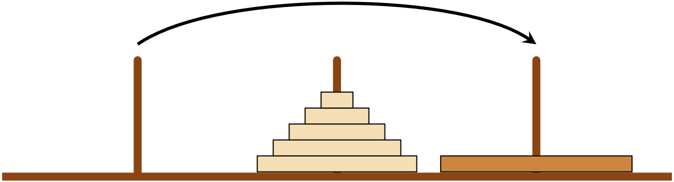
Comment déplacer les $n$ disques de la tige 1 à la tige 3 ?

On raisonne par récurrence :

- on déplace $n-1$ disques de la tige 1 à la tige 2 ;

# Les tours de HANOÏ

## Un problème emblématique de la programmation récursive



Trois tiges et $n$ disques doivent respecter les règles suivantes :

- un seul disque peut être déplacé à la fois ;
- on ne peut jamais poser un disque sur un disque de diamètre inférieur.

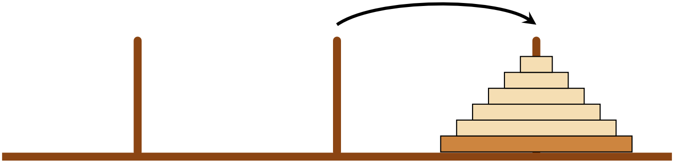
Comment déplacer les $n$ disques de la tige 1 à la tige 3 ?

On raisonne par récurrence :

- on déplace $n-1$ disques de la tige 1 à la tige 2 ;
- on déplace le plus gros disque de la tige 1 à la tige 3 ;

# Les tours de HANOÏ

## Un problème emblématique de la programmation récursive



Trois tiges et $n$ disques doivent respecter les règles suivantes :

- un seul disque peut être déplacé à la fois ;
- on ne peut jamais poser un disque sur un disque de diamètre inférieur.

Comment déplacer les $n$ disques de la tige 1 à la tige 3 ?

On raisonne par récurrence :

- on déplace $n-1$ disques de la tige 1 à la tige 2 ;
- on déplace le plus gros disque de la tige 1 à la tige 3 ;
- on déplace $n-1$ disques de la tige 2 à la tige 3.

# Les tours de Hanoï

Un problème emblématique de la programmation récursive

**Observation** : il faut généraliser le problème et savoir déplacer *n* disques de la tige *i* vers la tige *k*.

```python
def hanoi(n, i=1, j=2, k=3):
    if n == 0:
        return None
    hanoi(n-1, i, k, j)
    print("Déplacer le disque {} de la tige {} vers la tige {}."
                                                     .format(n, i, k))
    hanoi(n-1, j, i, k)
```

# Les tours de Hanoï

## Un problème emblématique de la programmation récursive

**Observation** : il faut généraliser le problème et savoir déplacer $n$ disques de la tige $i$ vers la tige $k$.

```python
def hanoi(n, i=1, j=2, k=3):
    if n == 0:
        return None
    hanoi(n-1, i, k, j)
    print("Déplacer le disque {} de la tige {} vers la tige {}."
                                                     .format(n, i, k))
    hanoi(n-1, j, i, k)
```

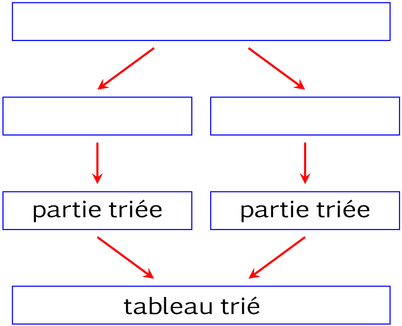
Condition d'arrêt : $n = 0$.
Nombre d'appels récursifs : $2^n$.

Avantage de la programmation récursive : notre unique tâche est de réduire le problème à un ou plusieurs sous-problèmes identiques et à veiller à respecter les deux conditions pour assurer la terminaison.

# Tri fusion

Algorithme pour trier un tableau de taille $n$ :

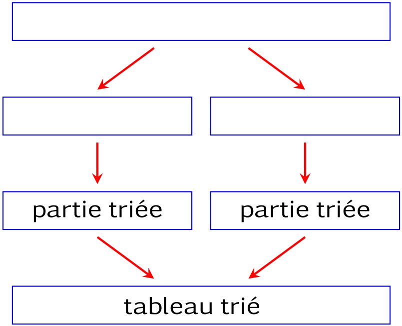
- diviser le tableau à trier en deux parties sensiblement égales ;
- trier récursivement chacune de ces deux parties ;
- fusionner les deux parties triées dans un seul tableau trié.



# Tri fusion

Algorithme pour trier un tableau de taille $n$ :

- diviser le tableau à trier en deux parties sensiblement égales ;
- trier récursivement chacune de ces deux parties ;
- fusionner les deux parties triées dans un seul tableau trié.




Condition d'arrêt : tableau de taille $< 2$ ;
Nombre d'appels récursifs : $\leqslant 2\lceil \log n \rceil$.

# Tri fusion

Scission du tableau : on utilise le slicing `t[:n//2]` et `t[n//2:]`.

# Tri fusion

Scission du tableau : on utilise le slicing `t[:n//2]` et `t[n//2:]`.

Fusion de deux tableaux triés :

```
def merge(a, b):
    p, q = len(a), len(b)
    c = [None] * (p + q)
    i = j = 0
    for k in range(p+q):
        if j >= q:
            c[k:] = a[i:]
            break
        elif i >= p:
            c[k:] = b[j:]
            break
        elif a[i] < b[j]:
            c[k] = a[i]
            i += 1
        else:
            c[k] = b[j]
            j += 1
    return c
```

# Tri fusion

Scission du tableau : on utilise le slicing `t[:n//2]` et `t[n//2:]`.
Fusion de deux tableaux triés : `merge(a, b)`

```
def mergesort(t):
    n = len(t)
    if n < 2:
        return t
    a = mergesort(t[:n//2])
    b = mergesort(t[n//2:])
    return merge(a, b)
```

# Tri fusion

Scission du tableau : on utilise le slicing `t[:n//2]` et `t[n//2:]`.
Fusion de deux tableaux triés : `merge(a, b)`

```
def mergesort(t):
    n = len(t)
    if n < 2:
        return t
    a = mergesort(t[:n//2])
    b = mergesort(t[n//2:])
    return merge(a, b)
```

Coût du calcul `merge(a, b)` : $\Theta(|a|+|b|)$ (création et remplissage du tableau c).

# Tri fusion

Scission du tableau : on utilise le slicing `t[:n//2]` et `t[n//2:]`.
Fusion de deux tableaux triés : `merge(a, b)`

```
def mergesort(t):
    n = len(t)
    if n < 2:
        return t
    a = mergesort(t[:n//2])
    b = mergesort(t[n//2:])
    return merge(a, b)
```

Coût du calcul `merge(a, b)` : $\Theta(|a|+|b|)$ (création et remplissage du tableau c).
Coût de la fonction `mergesort` :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

# Tri fusion

Scission du tableau : on utilise le slicing `t[:n//2]` et `t[n//2:]`.
Fusion de deux tableaux triés : `merge(a, b)`

```
def mergesort(t):
    n = len(t)
    if n < 2:
        return t
    a = mergesort(t[:n//2])
    b = mergesort(t[n//2:])
    return merge(a, b)
```

Coût du calcul `merge(a, b)` : $\Theta(|a|+|b|)$ (création et remplissage du tableau c).
Coût de la fonction `mergesort` :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

On peut prouver que $C(n) = \Theta(n \log n)$.

# Tri fusion

## Résolution lorsque $n = 2^p$

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

On pose $u(p) = C(2^p)$. Alors $2u(p-1) + A2^p \leqslant u(p) \leqslant 2u(p-1) + B2^p$.

# Tri fusion

## Résolution lorsque $n = 2^p$

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

On pose $u(p) = C(2^p)$. Alors $2u(p-1) + A2^p \leqslant u(p) \leqslant 2u(p-1) + B2^p$.

$$\frac{u(p-1)}{2^{p-1}} + A \leqslant \frac{u(p)}{2^p} \leqslant \frac{u(p-1)}{2^{p-1}} + B$$

# Tri fusion

## Résolution lorsque $n = 2^p$

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

On pose $u(p) = C(2^p)$. Alors $2u(p-1) + A2^p \leqslant u(p) \leqslant 2u(p-1) + B2^p$.

$$\frac{u(p-1)}{2^{p-1}} + A \leqslant \frac{u(p)}{2^p} \leqslant \frac{u(p-1)}{2^{p-1}} + B$$

Par télescopage, $u(0) + Ap \leqslant \dfrac{u(p)}{2^p} \leqslant u(0) + Bp$ donc $u(p) = \Theta(p2^p)$.

# Tri fusion

## Résolution lorsque $n = 2^p$

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

On pose $u(p) = C(2^p)$. Alors $2u(p-1) + A\,2^p \leqslant u(p) \leqslant 2u(p-1) + B\,2^p$.

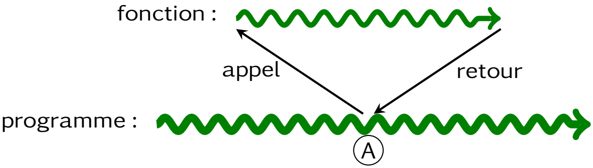
$$\frac{u(p-1)}{2^{p-1}} + A \leqslant \frac{u(p)}{2^p} \leqslant \frac{u(p-1)}{2^{p-1}} + B$$

Par télescopage, $u(0) + Ap \leqslant \dfrac{u(p)}{2^p} \leqslant u(0) + Bp$ donc $u(p) = \Theta(p2^p)$.

Avec $p = \log n$ on en déduit $C(n) = \Theta(n \log n)$.

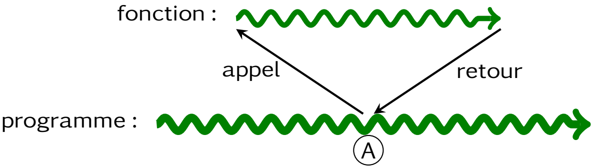
# Pile d'exécution d'un programme

Lors de l'appel d'une fonction au sein d'un programme, l'exécution de ce dernier est interrompu le temps de l'exécution de cette fonction, avant de reprendre à l'endroit où il s'est arrêté.

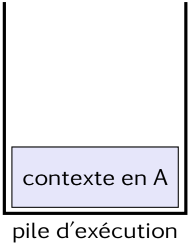


# Pile d’exécution d’un programme

Lors de l’appel d’une fonction au sein d’un programme, l’exécution de ce dernier est interrompu le temps de l’exécution de cette fonction, avant de reprendre à l’endroit où il s’est arrêté.




Il est nécessaire de sauvegarder l’état du contexte au moment de la bifurcation
⟶ dans la pile d’exécution du programme.

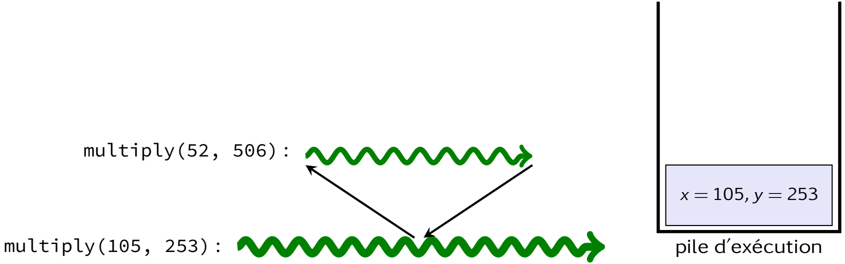


# Pile d'exécution d'un programme

Lors de l'exécution d'une fonction récursive, les appels récursifs conduisent à un empilement du contexte dans la pile d'exécution :

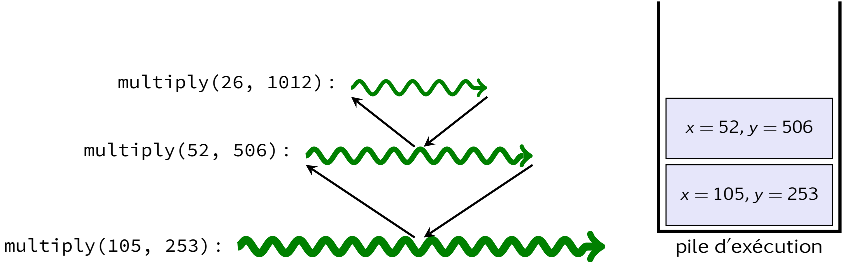
# Pile d'exécution d'un programme

Lors de l'exécution d'une fonction récursive, les appels récursifs conduisent à un empilement du contexte dans la pile d'exécution :



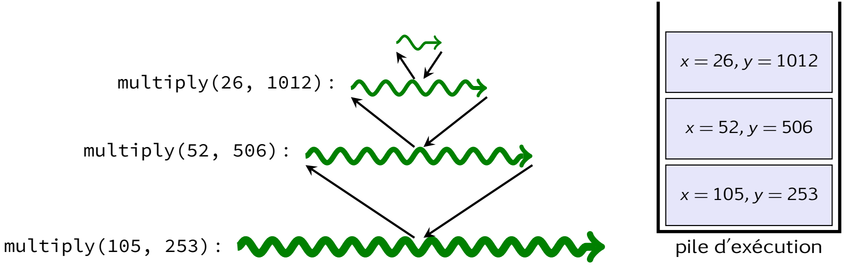
# Pile d'exécution d'un programme

Lors de l'exécution d'une fonction récursive, les appels récursifs conduisent à un empilement du contexte dans la pile d'exécution :



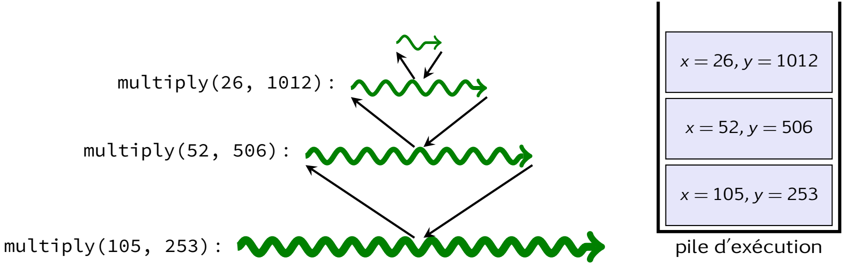
# Pile d'exécution d'un programme

Lors de l'exécution d'une fonction récursive, les appels récursifs conduisent à un empilement du contexte dans la pile d'exécution :



# Pile d'exécution d'un programme

Lors de l'exécution d'une fonction récursive, les appels récursifs conduisent à un empilement du contexte dans la pile d'exécution :




Une fonction récursive s'accompagne d'un coût spatial qui va croître avec le nombre d'appels récursifs (en général linéairement) ; ce coût ne doit pas être oublié lorsqu'on fait le bilan du coût d'une fonction récursive.

# Trace d'une fonction

Dans les langages de programmation de haut niveau, les spécificités de la pile d'exécution sont cachées au programmeur. Ce dernier a uniquement accès aux appels de fonctions et aux paramètres associés : la trace des appels récursifs.

# Trace d'une fonction

Dans les langages de programmation de haut niveau, les spécificités de la pile d'exécution sont cachées au programmeur. Ce dernier a uniquement accès aux appels de fonctions et aux paramètres associés : la trace des appels récursifs.

Un décorateur est une fonction qui à une fonction associe une autre fonction. Si `madeco` est un décorateur, le script suivant remplace la fonction `mafonction` par la fonction `madeco(mafonction)`.

```
@madeco
def mafonction(...):
    ....
```

# Trace d'une fonction

Dans les langages de programmation de haut niveau, les spécificités de la pile d'exécution sont cachées au programmeur. Ce dernier a uniquement accès aux appels de fonctions et aux paramètres associés : la trace des appels récursifs.

Un décorateur est une fonction qui à une fonction associe une autre fonction. Si madeco est un décorateur, le script suivant remplace la fonction mafonction par la fonction madeco(mafonction).

```
@madeco
def mafonction(...):
    ....
```

Souvent, un décorateur ajoute du code avant et après la fonction :

```
def madeco(func):
    def wrapper(args):
        # ici le code à exécuter avant la fonction
        func(args)
        # ici le code à exécuter après la fonction
    return wrapper
```

# Trace d'une fonction

Dans les langages de programmation de haut niveau, les spécificités de la pile d'exécution sont cachées au programmeur. Ce dernier a uniquement accès aux appels de fonctions et aux paramètres associés : la trace des appels récursifs.

Un décorateur est une fonction qui à une fonction associe une autre fonction. Si `madeco` est un décorateur, le script suivant remplace la fonction `mafonction` par la fonction `madeco(mafonction)`.

On définit le décorateur suivant :

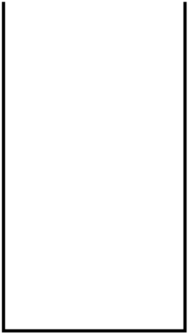
```python
def trace(func):
    def wrapper(*args):
        print('{} <- {}'.format(func.__name__, str(args)))
        val = func(*args)
        print('{} -> {}'.format(func.__name__, str(val)))
        return val
    return wrapper
```

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



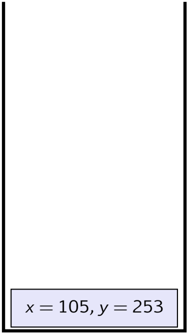
pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



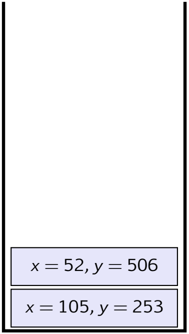

pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```python
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



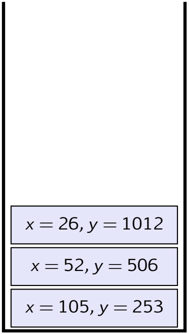

pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



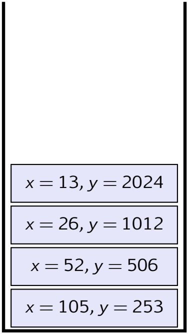

pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```




pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```

| |
|---|
| $x = 6, y = 4048$ |
| $x = 13, y = 2024$ |
| $x = 26, y = 1012$ |
| $x = 52, y = 506$ |
| $x = 105, y = 253$ |

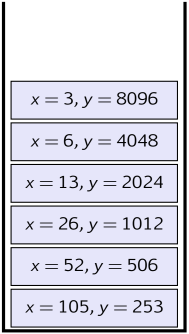
pile d'exécution

# Trace d’une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



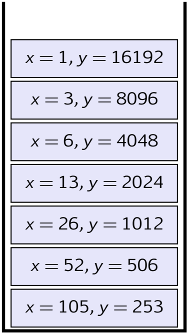

pile d’exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



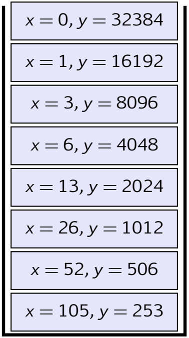

pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```




pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```

| |
|---|
| 0 |
| $x = 1, y = 16192$ |
| $x = 3, y = 8096$ |
| $x = 6, y = 4048$ |
| $x = 13, y = 2024$ |
| $x = 26, y = 1012$ |
| $x = 52, y = 506$ |
| $x = 105, y = 253$ |

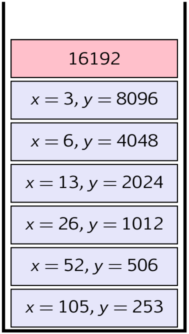
pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



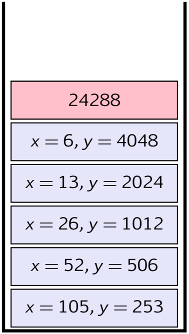

pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



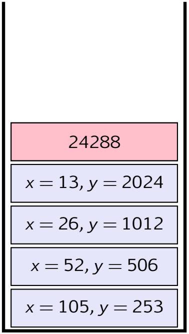

pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



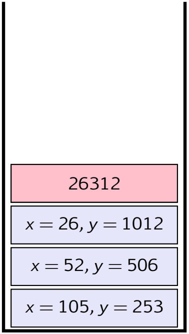

pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



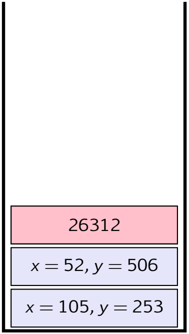

pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



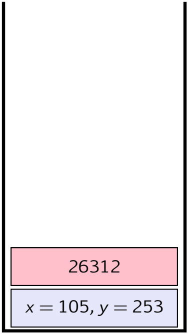

pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```



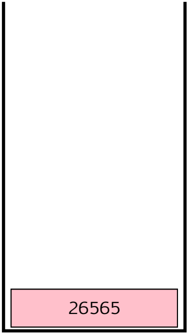

pile d'exécution

# Trace d'une fonction

## Exemple de multiplication du paysan russe

```python
@trace
def multiply(x, y):
    ......
```

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```




pile d'exécution

# Trace d'une fonction

## Exemple d'un tri fusion

La trace de la fonction `mergesort` appliquée à la liste $[6, 2, 4, 3, 5, 1]$ met en évidence les deux appels récursifs utilisés :

```
mergesort <- [6, 2, 4, 3, 5, 1]
```

Tri du demi-tableau $[6, 2, 4]$ :

```
mergesort <- [6, 2, 4]
 mergesort <- [6]
 mergesort -> [6]
 mergesort <- [2, 4]
  mergesort <- [2]
  mergesort -> [2]
  mergesort <- [4]
  mergesort -> [4]
 mergesort -> [2, 4]
mergesort -> [2, 4, 6]
```

Tri du demi-tableau $[3, 5, 1]$ :

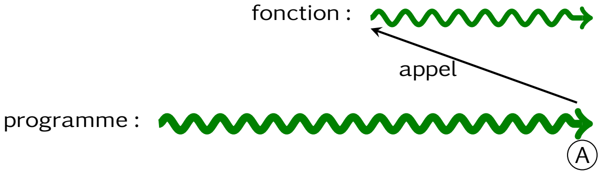
```
mergesort <- [3, 5, 1]
 mergesort <- [3]
 mergesort -> [3]
 mergesort <- [5, 1]
  mergesort <- [5]
  mergesort -> [5]
  mergesort <- [1]
  mergesort -> [1]
 mergesort -> [1, 5]
mergesort -> [1, 3, 5]
```

Fusion des deux demi-tableaux triés :

```
mergesort -> [1, 2, 3, 4, 5, 6]
```

# Récursivité terminale

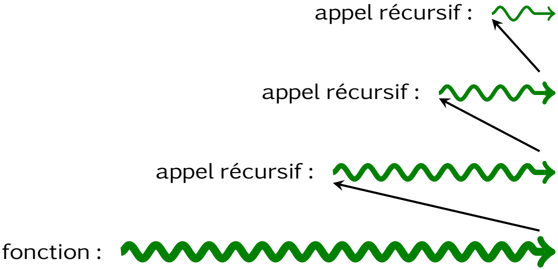
Lorsque l'appel à une fonction est la dernière instruction du programme, il n'est *a priori* pas nécessaire de mémoriser le contexte.



# Récursivité terminale

Lorsque l'appel à une fonction est la dernière instruction du programme, il n'est *a priori* pas nécessaire de mémoriser le contexte.

Dans une fonction récursive terminale, l'appel récursif est la *dernière* instruction à être évaluée : le contexte n'est pas rangé dans la pile d'exécution, il n'y a pas de coût spatial.



# Récursivité terminale

Lorsque l'appel à une fonction est la dernière instruction du programme, il n'est *a priori* pas nécessaire de mémoriser le contexte.

Dans une fonction récursive terminale, l'appel récursif est la *dernière* instruction à être évaluée : le contexte n'est pas rangé dans la pile d'exécution, il n'y a pas de coût spatial.

Exemple de récursivité terminale :

```python
def pgcd(a, b):
    if b == 0:
        return a
    return pgcd(b, a % b)
```

```
pgcd <- (132, 48)
 pgcd <- (48, 36)
  pgcd <- (36, 12)
   pgcd <- (12, 0)
   pgcd -> 12
  pgcd -> 12
 pgcd -> 12
pgcd -> 12
```

# Récursivité terminale

Lorsque l'appel à une fonction est la dernière instruction du programme, il n'est *a priori* pas nécessaire de mémoriser le contexte.

Dans une fonction récursive terminale, l'appel récursif est la *dernière* instruction à être évaluée : le contexte n'est pas rangé dans la pile d'exécution, il n'y a pas de coût spatial.

Exemple de récursivité terminale :

```python
def pgcd(a, b):
    if b == 0:
        return a
    return pgcd(b, a % b)
```

```
pgcd <- (132, 48)
 pgcd <- (48, 36)
  pgcd <- (36, 12)
   pgcd <- (12, 0)
   pgcd -> 12
  pgcd -> 12
 pgcd -> 12
pgcd -> 12
```

Attention : la récursivité terminale n'est pas gérée en PYTHON : il n'y a donc aucun intérêt à chercher à écrire une version terminale des algorithmes.

# Les écueils de la programmation récursive

## Attention aux coûts cachés

Principe de la recherche dichotomique dans un tableau trié : on compare $x$ et $t_k$ avec $k = \lfloor n/2 \rfloor$.

- si $x = t_k$, la recherche est terminée ;
- si $x < t_k$ la recherche se poursuit dans $t[0 \dots k-1]$ ;
- si $x > t_k$ la recherche se poursuit dans $t[k+1 \dots n-1]$.

# Les écueils de la programmation récursive

## Attention aux coûts cachés

Principe de la recherche dichotomique dans un tableau trié : on compare $x$ et $t_k$ avec $k = \lfloor n/2 \rfloor$.

- si $x = t_k$, la recherche est terminée ;
- si $x < t_k$ la recherche se poursuit dans $t[0 \dots k-1]$ ;
- si $x > t_k$ la recherche se poursuit dans $t[k+1 \dots n-1]$.

```python
def dicho(x, t):
    if len(t) == 0:
        return False
    k = len(t) // 2
    if x == t[k]:
        return True
    elif x < t[k]:
        return dicho(x, t[:k])
    else:
        return dicho(x, t[k+1:])
```

# Les écueils de la programmation récursive

## Attention aux coûts cachés

Principe de la recherche dichotomique dans un tableau trié : on compare $x$ et $t_k$ avec $k = \lfloor n/2 \rfloor$.

- si $x = t_k$, la recherche est terminée ;
- si $x < t_k$ la recherche se poursuit dans $t[0 \dots k-1]$ ;
- si $x > t_k$ la recherche se poursuit dans $t[k+1 \dots n-1]$.

```python
def dicho(x, t):
    if len(t) == 0:
        return False
    k = len(t) // 2
    if x == t[k]:
        return True
    elif x < t[k]:
        return dicho(x, t[:k])
    else:
        return dicho(x, t[k+1:])
```

Cette fonction n'est par de coût logarithmique mais de coût linéaire ! (en cause : la recopie en mémoire d'un demi-tableau à chaque étape).

# Les écueils de la programmation récursive

## Attention aux coûts cachés

La bonne version récursive : généraliser en cherchant $x$ dans $t[i...j-1]$ :

```
def dicho(x, t, *args):
    if len(args) == 0:
        i, j = 0, len(t)
    else:
        i, j = args
    if j <= i:
        return False
    k = (i + j) // 2
    if x == t[k]:
        return True
    elif x < t[k]:
        return dicho(x, t, i, k)
    else:
        return dicho(x, t, k+1, j)
```

# Les écueils de la programmation récursive

## Attention aux coûts cachés

La bonne version récursive : généraliser en cherchant $x$ dans $t[i \dots j-1]$ :

```python
def dicho(x, t, *args):
    if len(args) == 0:
        i, j = 0, len(t)
    else:
        i, j = args
    if j <= i:
        return False
    k = (i + j) // 2
    if x == t[k]:
        return True
    elif x < t[k]:
        return dicho(x, t, i, k)
    else:
        return dicho(x, t, k+1, j)
```

$C(n) = C(n/2) + \mathrm{O}(1)$ donc $C(n) = \mathrm{O}(\log n)$.

# Les écueils de la programmation récursive

## Attention aux coûts cachés

La bonne version récursive : généraliser en cherchant $x$ dans $t[i...j-1]$ :

```
def dicho(x, t, *args):
    if len(args) == 0:
        i, j = 0, len(t)
    else:
        i, j = args
    if j <= i:
        return False
    k = (i + j) // 2
    if x == t[k]:
        return True
    elif x < t[k]:
        return dicho(x, t, i, k)
    else:
        return dicho(x, t, k+1, j)
```

$C(n) = C(n/2) + \mathrm{O}(1)$ donc $C(n) = \mathrm{O}(\log n)$.

Lorsque $n = 2^p$ et $u_p = C(n)$ on a $u_p = u_{p-1} + \mathrm{O}(1)$ donc $u_p = \mathrm{O}(p)$, soit $C(n) = \mathrm{O}(\log n)$.

# Appels récursifs multiples

Calcul du $n^e$ terme $f_n$ de la suite de FIBONACCI :

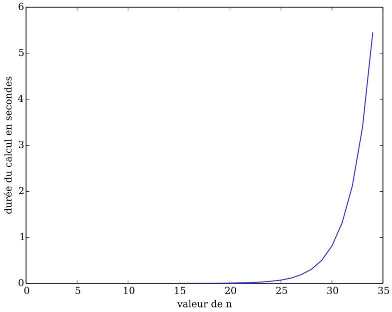
```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

# Appels récursifs multiples

Calcul du $n^{e}$ terme $f_n$ de la suite de FIBONACCI :

```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Le coût de cette fonction est exponentiel :



# Appels récursifs multiples

Calcul du $n^e$ terme $f_n$ de la suite de FIBONACCI :

```python
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Trace des appels pour calculer $f_6$ : $f_2$ est calculé cinq fois !

```
fib <- 6
 fib <- 5
  fib <- 4
   fib <- 3
    fib <- 2  ***
     fib <- 1
     fib -> 1
     fib <- 0
     fib -> 0
    fib -> 1
    fib <- 1
    fib -> 1
   fib -> 2
   fib <- 2   ***
    fib <- 1
    fib -> 1
    fib <- 0
```

```
    fib -> 0
   fib -> 1
  fib -> 3
  fib <- 3
   fib <- 2   ***
    fib <- 1
    fib -> 1
    fib <- 0
    fib -> 0
   fib -> 1
   fib <- 1
   fib -> 1
  fib -> 2
 fib -> 5
 fib <- 4
  fib <- 3
   fib <- 2   ***
```

```
    fib <- 1
    fib -> 1
    fib <- 0
    fib -> 0
   fib -> 1
   fib <- 1
   fib -> 1
  fib -> 2
  fib <- 2   ***
   fib <- 1
   fib -> 1
   fib <- 0
   fib -> 0
  fib -> 1
 fib -> 3
fib -> 8
```

# Appels récursifs multiples

Calcul du $n^e$ terme $f_n$ de la suite de Fibonacci :

```python
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Le nombre $a_n$ d'appels à la fonction fib pour calculer $f_n$ vérifie les relations :

$$a_0 = a_1 = 1 \qquad \text{et} \qquad \forall n \geqslant 2,\ a_n = a_{n-1} + a_{n-2} + 1.$$

Cette suite se résout en $a_n = 2f_{n+1} - 1 = \Theta(\varphi^n)$ ; le coût de cette fonction, tant temporel que spatial, est exponentiel !

# Appels récursifs multiples

## Mémoïsation

Une solution mathématique : itérer la suite de vecteurs $(f_n, f_{n+1})$ pour ne plus avoir qu'un seul appel récursif :

$$(f_0, f_1) = (0, 1) \qquad \text{et} \qquad (f_n, f_{n+1}) = (f_n, f_{n-1} + f_n) = \varphi(f_{n-1}, f_n)$$

```python
def fib(n):
    def aux(n):
        if n == 0:
            return (0, 1)
        else:
            (x, y) = aux(n-1)
            return (y, x + y)
    return aux(n)[0]
```

mais cette solution nous éloigne de la simplicité de la première version récursive.

# Appels récursifs multiples

## Mémoïsation

Autre solution : mémoriser le résultat des calculs dans un dictionnaire.

# Appels récursifs multiples

## Mémoïsation

Autre solution : mémoriser le résultat des calculs dans un dictionnaire.

- `d = {c1: v1, c2: v2, c3: v3}` crée un dictionnaire d comportant trois paires d'association ;

# Appels récursifs multiples

## Mémoïsation

Autre solution : mémoriser le résultat des calculs dans un dictionnaire.

- `d = {c1: v1, c2: v2, c3: v3}` crée un dictionnaire d comportant trois paires d'association ;
- `d[c2]` renvoie la valeur $v_2$ associée à la clé $c_2$ ou déclenche l'exception `KeyError` si l'association n'existe pas ;

# Appels récursifs multiples

## Mémoïsation

Autre solution : mémoriser le résultat des calculs dans un dictionnaire.

- `d = {c1: v1, c2: v2, c3: v3}` crée un dictionnaire d comportant trois paires d'association ;
- `d[c2]` renvoie la valeur $v_2$ associée à la clé $c_2$ ou déclenche l'exception `KeyError` si l'association n'existe pas ;
- `d[c4] = v4` ajoute une nouvelle paire d'association.

# Appels récursifs multiples

## Mémoïsation

Autre solution : mémoriser le résultat des calculs dans un dictionnaire.

- `d = {c1: v1, c2: v2, c3: v3}` crée un dictionnaire d comportant trois paires d'association ;
- `d[c2]` renvoie la valeur $v_2$ associée à la clé $c_2$ ou déclenche l'exception `KeyError` si l'association n'existe pas ;
- `d[c4] = v4` ajoute une nouvelle paire d'association.

```python
d_fib = {0: 0, 1: 1}

def fib(n):
    if n not in d_fib:
        d_fib[n] = fib(n−1) + fib(n−2)
    return d_fib[n]
```

# Appels récursifs multiples

## Mémoïsation

Autre solution : mémoriser le résultat des calculs dans un dictionnaire.

- `d = {c1: v1, c2: v2, c3: v3}` crée un dictionnaire d comportant trois paires d'association ;
- `d[c2]` renvoie la valeur $v_2$ associée à la clé $c_2$ ou déclenche l'exception `KeyError` si l'association n'existe pas ;
- `d[c4] = v4` ajoute une nouvelle paire d'association.

```
d_fib = {0: 0, 1: 1}

def fib(n):
    if n not in d_fib:
        d_fib[n] = fib(n-1) + fib(n-2)
    return d_fib[n]
```

```
>>> fib(10)
55
>>> d_fib
{0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34, 10: 55}
```

# Décorateur et mémoïsation

On peut définir un décorateur pour mémoïser automatiquement une fonction récursive :

```
def memoise(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper
```

# Décorateur et mémoïsation

On peut définir un décorateur pour mémoïser automatiquement une fonction récursive :

```
def memoise(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper
```

Mémoïsation de la fonction fib :

```
@memoise
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

# Décorateur et mémoïsation

On peut définir un décorateur pour mémoïser automatiquement une fonction récursive :

```python
def memoise(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper
```

Calcul d'un coefficient binomial par mémoïsation :

```python
@memoise
def binom(n, p):
    if p == 0 or n == p:
        return 1
    return binom(n-1, p-1) + binom(n-1, p)
```

# Décorateur et mémoïsation

On peut définir un décorateur pour mémoïser automatiquement une fonction récursive :

```
def memoise(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper
```

Calcul d'un coefficient binomial par mémoïsation :

```
@memoise
def binom(n, p):
    if p == 0 or n == p:
        return 1
    return binom(n-1, p-1) + binom(n-1, p)
```

Coût spatial et temporel du calcul de $\binom{n}{p}$ :

# Décorateur et mémoïsation

On peut définir un décorateur pour mémoïser automatiquement une fonction récursive :

```
def memoise(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper
```

Calcul d'un coefficient binomial par mémoïsation :

```
@memoise
def binom(n, p):
    if p == 0 or n == p:
        return 1
    return binom(n-1, p-1) + binom(n-1, p)
```

Coût spatial et temporel du calcul de $\binom{n}{p}$ : $\mathrm{O}(np)$ (la taille du dictionnaire nécessaire).