Fonctions et récursivité
STS SIO

Alex Esbelin
(Alex.Esbelin@math.univ-bpclermont.fr)
Malika More
(malika.more@u-clermont1.fr)

IREM Clermont-Ferrand

Stage du 29 Juin 2011

Contenu

Introduction

Fonction `Saut3Lignes()`

début
    Aller à la ligne
    Aller à la ligne
    Aller à la ligne
fin

- Éviter les redondances
  - Isoler une séquence d'instructions dans une fonction
  - Remplacer ses occurrences par des appels à la fonction
- Améliorer la clarté des algorithmes
  - Pas besoin de connaître l'intérieur de la fonction
- Permettre la modularité
  - J'écris une fonction aujourd'hui, une autre demain
  - J'écris une fonction et toi une autre

Paramètres d'entrée

Fonction `SautLignes(`*n : entier*`)`

début

  pour *i* **de** 1 **à** *n* **faire**

    Aller à la ligne

  fin

fin

- Permettre quelques variations
  - `SautLignes(3)`
  - `SautLignes(8)`
- Un paramètre peut être une variable
  - `Donner à` *p* `la valeur` 5
    `SautLignes(`*p*`)`
  - Le nom de la variable et le nom du paramètre sont différents

Valeur(s) retournée(s)

Fonction `Euclide(`*a,b : entiers*`)`

début
- Donner à *x* la valeur *a*
- Donner à *y* la valeur *b*
- **tant que** $y > 0$ **faire**
  - Donner à *temp* la valeur *y*
  - Donner à *y* la valeur $x \mod y$
  - Donner à *x* la valeur *temp*
- **fin**
- **retourner** : *x*

fin

- L'expression obtenue peut être
  - Assignée : `Donner à` *n* `la valeur Euclide(8,13)`
  - Affichée : `Afficher(Euclide(8,13))`
  - Etc.

Quelques points subtils

À propos des variables

- Variable locale : définie à l'intérieur d'une fonction, le programme principal et les autres fonctions ne la connaissent pas
  - Il est préconisé de déclarer les variables locales et leur type sur papier
  - Pas de déclaration en Python (ça dépend du langage)
- Variable globale : utilisée par le programme principal, à l'intérieur de la fonction on peut la lire, mais pas la modifier
  - Sauf si elle est déclarée globale à l'intérieur de la fonction (en Python)
  - Sauf si elle est d'un type "liste" (en Python)

Quelques points subtils

À propos des paramètres

- Passage par valeur : valeur du paramètre recopiée dans une nouvelle variable pour être utilisée dans la fonction
  - Consomme de la mémoire supplémentaire
  - La valeur du paramètre à l'extérieur ne peut pas être modifiée par la fonction
- Passage par référence : le paramètre lui-même est directement utilisé dans la fonction
  - Ne consomme pas de mémoire supplémentaire
  - La valeur du paramètre peut être modifiée par la fonction
- En Python, tous les paramètres sont passés par référence (pour la gestion de la mémoire) mais les types simples sont "immutables" (ne peuvent pas être modifiés par la fonction)

Commençons par un petit exercice de maths.
On considère la suite (u_n) définie pour $n \in \mathbb{N}$ par :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = 2 \times u_n + 3 \end{cases}$$

Sans utiliser l'expression de u_n en fonction de n, on se propose d'écrire un algorithme qui pour un entier n donné, retourne la valeur de u_n.

Ce que nous avons vu ce matin nous conduit à écrire :

Fonction `Uit(`*n : entier*`)`

Entrée : un entier n
Résultat : La valeur du terme u_n de la suite définie par : $u_0 = 1$ et $u_{n+1} = 2 \times u_n + 3$

```
début
    Donner à Utmp la valeur 1
    pour k de 1 à n faire
        Donner à Utmp la valeur 2 × Utmp + 3
    fin
    Retourner Utmp
fin
```

Remarque

On « monte » successivement les calculs des termes de la suite à partir de u_0 jusqu'à u_n.
$u_0 = 1$, $u_1 = 2 \times u_0 + 3 = 2 \times 1 + 3 = 5$ etc . . .
On fera référence à cet algorithme, sous le qualificatif de version itérative.

On peut aussi envisager de réaliser les calculs dans le sens contraire, ce qui donne :

$$u_n = \underbrace{2 \times (2 \times (\ldots (2 \times u_0 + 3) \ldots) + 3) + 3}_{n \textit{ groupes de } \text{parenthèses}}$$

On peut noter que cet ordre n'est pas le plus pratique pour mener les calculs, mais il est tout à fait concevable.

Nous allons voir dans la suite que cette façon de procéder est prévue dans l'algorithmique et peut apporter un grand confort dans certains cas.

On peut aussi envisager de réaliser les calculs dans le sens contraire, ce qui donne :

$$u_n = \underbrace{2 \times (2 \times (\ldots (2 \times u_0 + 3) \ldots) + 3) + 3}_{n \textit{ groupes de } \text{parenthèses}}$$

On peut noter que cet ordre n'est pas le plus pratique pour mener les calculs, mais il est tout à fait concevable.

Nous allons voir dans la suite que cette façon de procéder est prévue dans l'algorithmique et peut apporter un grand confort dans certains cas.

Définition

On dit d'une fonction qu'elle est récursive lorsqu'elle fait appel à elle-même.

Remarque

Le lien avec la récurrence en mathématiques n'est pas le fruit du hasard.

Revenons, à notre exemple :
l'écriture

$$u_n = 2 \times (2 \times (\ldots (2 \times u_0 + 3) \ldots) + 3) + 3$$

fait apparaitre un appel récursif à la séquence $2 \times \ldots + 3$

u_n est égal à $2 \times u_{n-1} + 3$
avec u_{n-1} lui-même égal à $2 \times u_{n-2} + 3$
...
la «descente» s'arrête avec la valeur de u_0 (égale à 1 ici).

Revenons, à notre exemple :
l'écriture

$$u_n = 2 \times (2 \times (\ldots (2 \times u_0 + 3) \ldots) + 3) + 3$$

fait apparaitre un appel récursif à la séquence $2 \times \ldots + 3$

u_n est égal à $2 \times u_{n-1} + 3$
avec u_{n-1} lui-même égal à $2 \times u_{n-2} + 3$
...
la «descente» s'arrête avec la valeur de u_0 (égale à 1 ici).

L'écriture de l'algorithme récursif donne :

Fonction `Urec(`*n : entier*`)`

Entrée : un entier n

Résultat : La valeur du terme u_n de la suite définie par : $u_0 = 1$ et $u_{n+1} = 2 \times u_n + 3$

début

  si *n = 0* **alors**

    retourner la valeur 1

    `% On traite le cas trivial correspondant au premier terme de la suite %`

  sinon

    retourner la valeur $2 \times \text{Urec}(n-1) + 3$

    `% ici,` $n \neq 0$ `donc on retourne` $2\times$ `la valeur du terme précédent` $+3$ `%`

  fin

fin

Remarques

- Le test $n = 0$ est essentiel puisqu'il assure l'arrêt des appels récursifs.
  ☞ *par sécurité, il peut-être remplacé par* $n \leq 0$
- De plus l'appel suivant se fait avec une valeur strictement inférieure à la valeur courante du paramètre.
  ☞ *assure la terminaison de l'algorithme.*

Remarques

- Le test $n = 0$ est essentiel puisqu'il assure l'arrêt des appels récursifs.
  ☞ *par sécurité, il peut-être remplacé par* $n \leq 0$
- De plus l'appel suivant se fait avec une valeur strictement inférieure à la valeur courante du paramètre.
  ☞ *assure la terminaison de l'algorithme.*

- L'écriture de l'algorithme sous sa version itérative a nécessité un travail de réflexion plus conséquent.
  ☞ *Mise en place d'une boucle, gestion du compteur de boucle, gestion d'un résultat intermédiaire etc. . .*
- La version récursive, elle, se trouve être la simple traduction de la définition mathématique de notre suite.

Dans toutes les situations, où la récurrence entre en jeu, les algorithmes récursifs vont procurer des avantages non négligeables dans leur écriture.

- L'écriture de l'algorithme sous sa version itérative a nécessité un travail de réflexion plus conséquent.
  ☞ *Mise en place d'une boucle, gestion du compteur de boucle, gestion d'un résultat intermédiaire etc. . .*
- La version récursive, elle, se trouve être la simple traduction de la définition mathématique de notre suite.

Dans toutes les situations, où la récurrence entre en jeu, les algorithmes récursifs vont procurer des avantages non négligeables dans leur écriture.

- L'écriture de l'algorithme sous sa version itérative a nécessité un travail de réflexion plus conséquent.
  ☞ *Mise en place d'une boucle, gestion du compteur de boucle, gestion d'un résultat intermédiaire etc. . .*
- La version récursive, elle, se trouve être la simple traduction de la définition mathématique de notre suite.

Dans toutes les situations, où la récurrence entre en jeu, les algorithmes récursifs vont procurer des avantages non négligeables dans leur écriture.

Le calcul de la factorielle sous forme itérative donne :

Fonction `Facto(`*n : entier*`)`

Entrée : un entier *n*
Résultat : Factorielle de *n*
début
    Donner à *Utmp* la valeur 1
    pour *k* **de** *1* **à** *n* **faire**
        Donner à *Utmp* la valeur $k \times Utmp$
    fin
    Retourner *Utmp*
fin

Question : Ecrire un algorithme récursif du calcul de $n!$ pour n donné, puis traduire cet algorithme sous Python.

Le calcul de la factorielle sous forme itérative donne :

Fonction `Facto(`*n : entier*`)`

Entrée : un entier *n*
Résultat : Factorielle de *n*
début
    Donner à *Utmp* la valeur 1
    pour *k* **de** *1* **à** *n* **faire**
        Donner à *Utmp* la valeur $k \times Utmp$
    fin
    Retourner *Utmp*
fin

Question : Ecrire un algorithme récursif du calcul de $n!$ pour n donné, puis traduire cet algorithme sous Python.

A travers les exemples précédents, on a pu constater qu'il est impératif :

- que l'appel récursif porte sur une valeur de paramètre inférieure à la valeur du paramètre précédent
- qu'un test de sortie soit réalisé sur une valeur précise du paramètre.

Ces points sont essentiels pour assurer la terminaison de l'algorithme récursif.
La preuve de correction, elle, est une preuve par récurrence.

Question : qu'en est-il de l'efficacité de ces algorithmes ?

A travers les exemples précédents, on a pu constater qu'il est impératif :

- que l'appel récursif porte sur une valeur de paramètre inférieure à la valeur du paramètre précédent
- qu'un test de sortie soit réalisé sur une valeur précise du paramètre.

Ces points sont essentiels pour assurer la terminaison de l'algorithme récursif.
La preuve de correction, elle, est une preuve par récurrence.

Question : qu'en est-il de l'efficacité de ces algorithmes ?

A travers les exemples précédents, on a pu constater qu'il est impératif :

- que l'appel récursif porte sur une valeur de paramètre inférieure à la valeur du paramètre précédent
- qu'un test de sortie soit réalisé sur une valeur précise du paramètre.

Ces points sont essentiels pour assurer la terminaison de l'algorithme récursif.
La preuve de correction, elle, est une preuve par récurrence.

Question : qu'en est-il de l'efficacité de ces algorithmes ?

Un autre exemple : suite de Fibonacci

Considérons la suite de Fibonacci, dont on rappelle la définition ci-dessous :

$$\begin{cases} F_0 = F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \text{pour } n \geq 2 \end{cases}$$

Algorithmes itératif et récursif

On sait maintenant écrire deux algorithmes différents :

Fonction `Fib(`n`)` % *version itérative*

début
  si $n < 2$ **alors**
    retourner : 1
  sinon
    Donner à x la valeur 1
    Donner à y la valeur 1
    for i **de** 2 **à** n **do**
      Donner à *temp* la valeur $x + y$
      Donner à x la valeur y
      Donner à y la valeur *temp*
    end
    retourner : y
  fin
fin

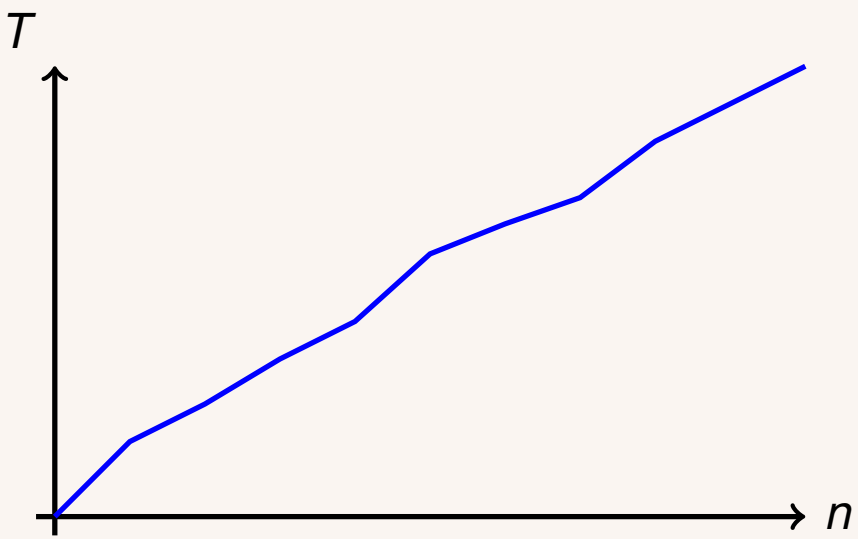
Fonction `Fib(`n`)` % *version récursive*

début
  si $n < 2$ **alors**
    retourner : 1
  fin
  retourner : Fib$(n-1)$+Fib$(n-2)$
fin

Quelques temps de calculs

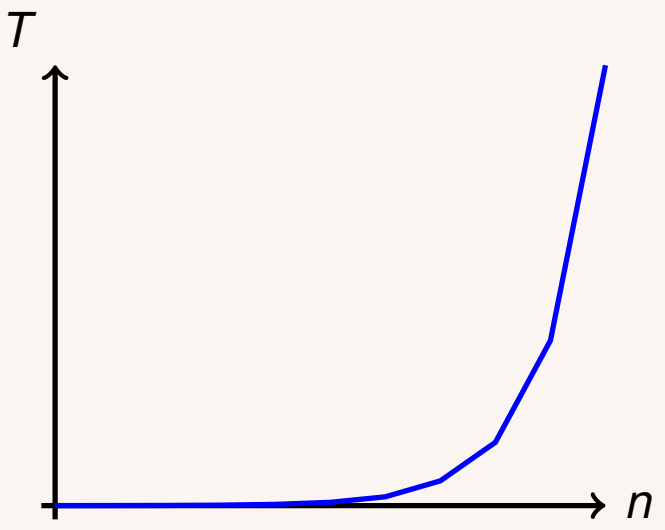
Itératif

n	T(n)
0	0.0e-05
2	2.0e-05
4	3.0e-05
6	4.2e-05
8	5.2e-05
10	7.0e-05
12	7.8e-05
14	8.5e-05
16	1.0e-04
18	1.1e-04
20	1.2e-04



Récursif

n	T(n)
0	0.0e-05
2	2.0e-05
4	6.0e-05
6	1.7e-04
8	4.6e-04
10	1.2e-03
12	3.2e-03
14	9.0e-03
16	2.3e-02
18	6.0e-02
20	1.6e-01



Quelques temps de calculs

Le constat est sans appel : l'algorithme itératif sort gagnant du duel.
Par exemple : le calcul de F_{20} nécessite 1.6e-01 s en version récursive contre 1.2e-04 s en version itérative, soit 1300 fois plus !

Au dela de cet aspect, les courbes de temps de ces deux versions contrastent énormément :

- La version itérative révèle une courbe T qui semble proportionnelle au temps.
- La version récursive révèle une courbe T présentant une croissance qui semble exponentielle. ☞ *Le calcul de F_{40}, par exemple, peut même se révéler plus rapide à la main.*

Quelques temps de calculs

Le constat est sans appel : l'algorithme itératif sort gagnant du duel.
Par exemple : le calcul de F_{20} nécessite 1.6e-01 s en version récursive contre 1.2e-04 s en version itérative, soit 1300 fois plus !

Au dela de cet aspect, les courbes de temps de ces deux versions contrastent énormément :

- La version itérative révèle une courbe T qui semble proportionnelle au temps.
- La version récursive révèle une courbe T présentant une croissance qui semble exponentielle. ☞ *Le calcul de F_{40}, par exemple, peut même se révéler plus rapide à la main.*

Une pile est une structure de donnée est dite de type LIFO [1] :

La dernière valeur entrée dans la pile est la première sortie, c'est-à-dire accessible.

Pour se représenter cette structure, il est intéressant de faire l'analogie avec une pile d'assiettes.

1. Last In First Out

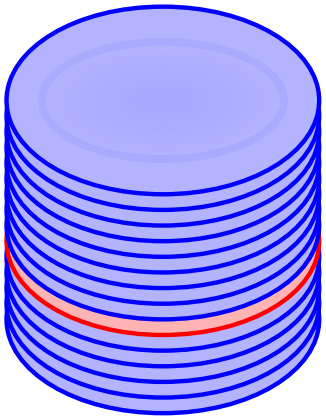
Une pile est une structure de donnée est dite de type LIFO [1] :

La dernière valeur entrée dans la pile est la première sortie, c'est-à-dire accessible.

Pour se représenter cette structure, il est intéressant de faire l'analogie avec une pile d'assiettes.

1. Last In First Out

Un exemple



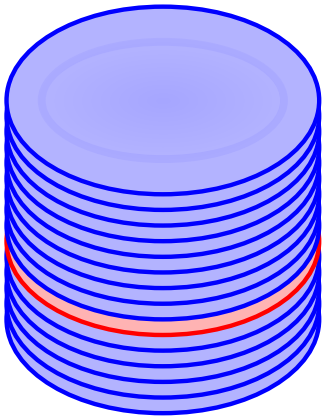
Présentation

On dispose d'une pile d'assiettes. Cette pile se compose d'assiettes bleues en dehors d'une assiette qui est rouge.

Objectif

Ecrire un algorithme, permettant de récupérer cette assiette rouge, afin de n'avoir plus que des assiettes bleues.

Un exemple



Présentation

On dispose d'une pile d'assiettes. Cette pile se compose d'assiettes bleues en dehors d'une assiette qui est rouge.

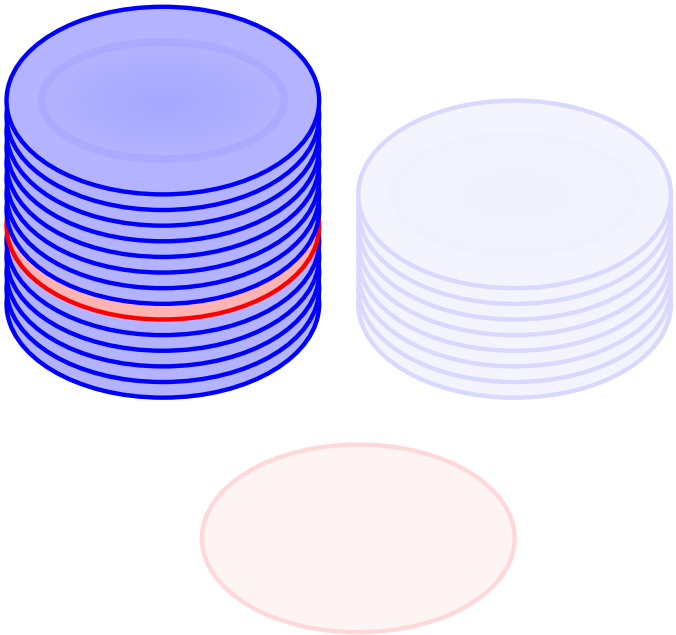
Objectif

Ecrire un algorithme, permettant de récupérer cette assiette rouge, afin de n'avoir plus que des assiettes bleues.

Pour cela, nous supposerons avoir à notre disposition, les fonctions suivantes :

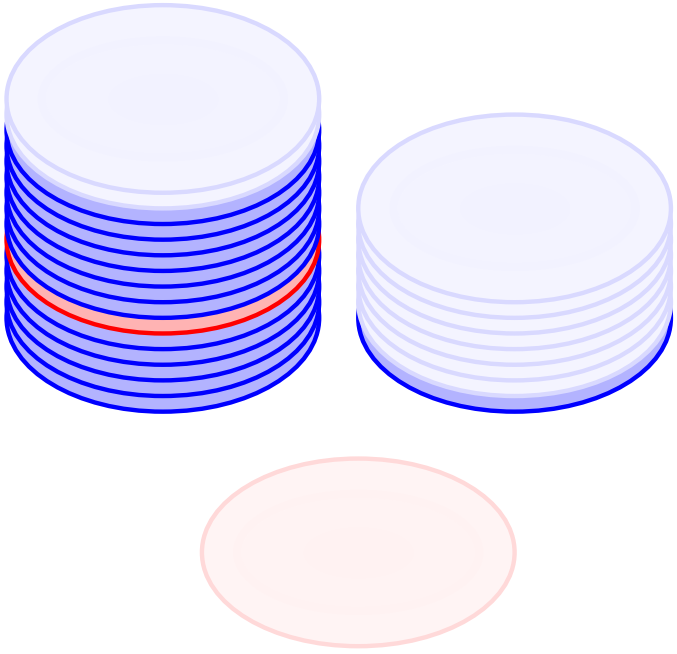
- *empiler(P,a)* ajoute l'élément *a* au sommet de la pile *P* et restitue la pile *P* ainsi modifiée.
- *depiler(P,a)* enlève l'élément supérieur de la pile *P*, restitue cet élément dans *a* et restitue la pile *P* sans cet élément.

En pratique



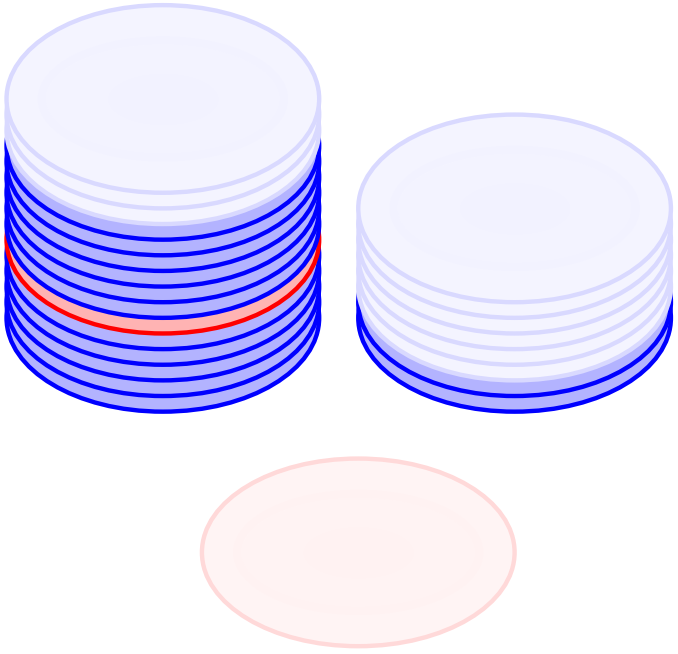
L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique



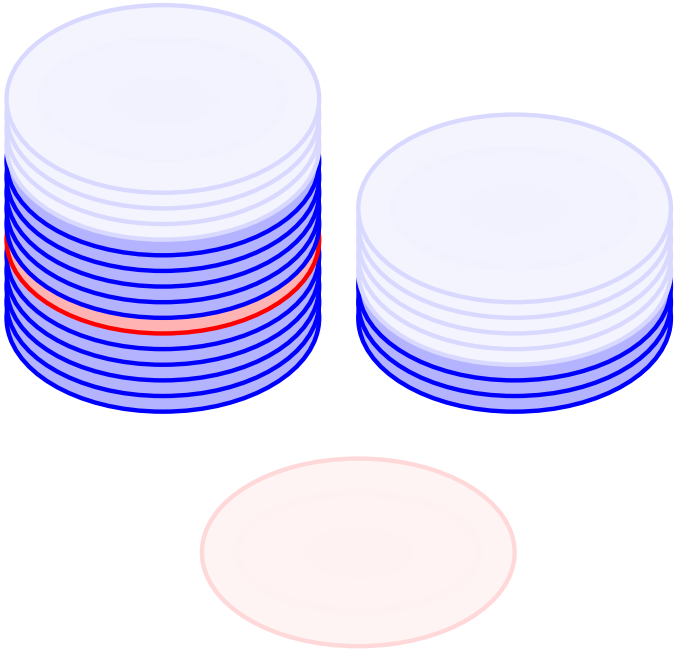
L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique



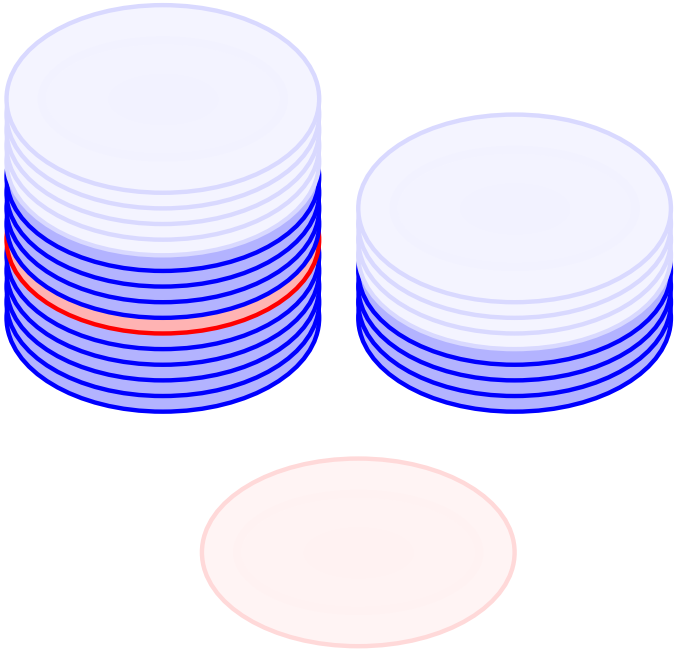
L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique



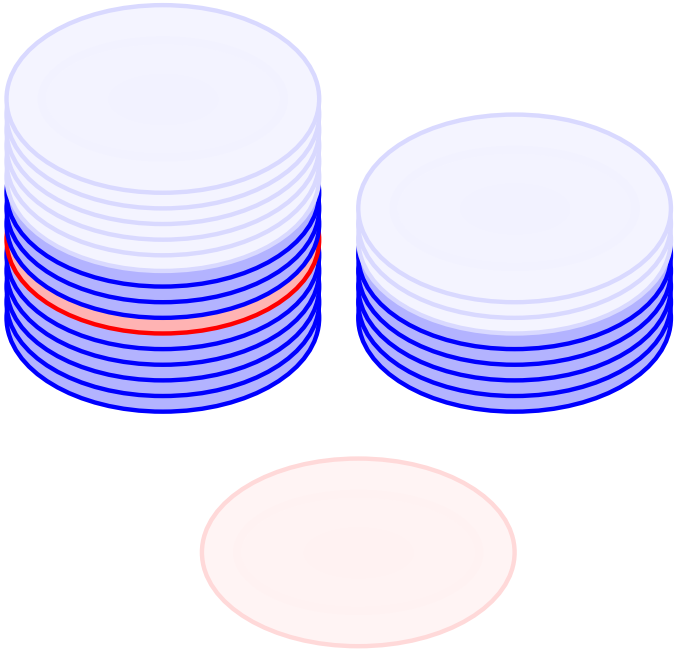
L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique



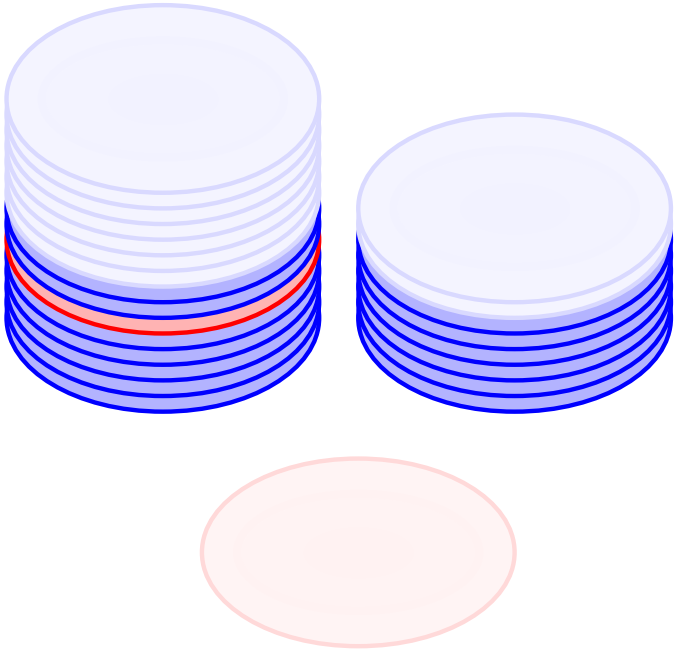
L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique



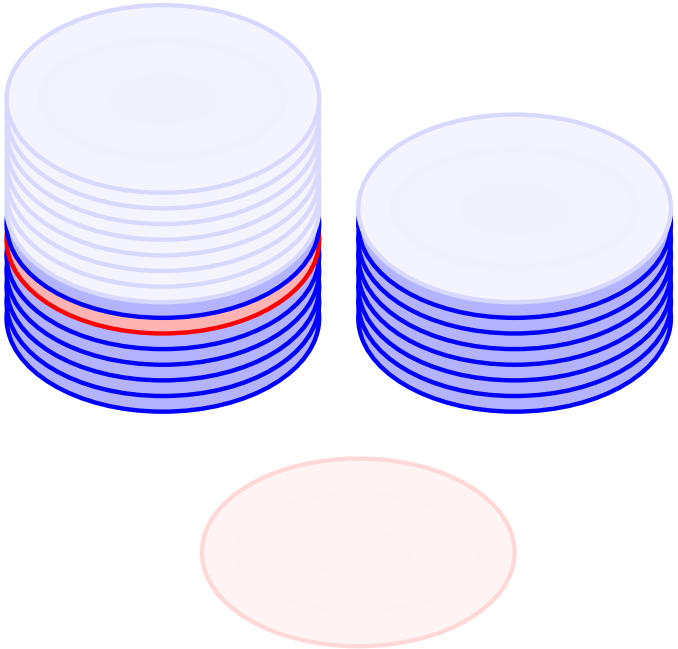
L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique



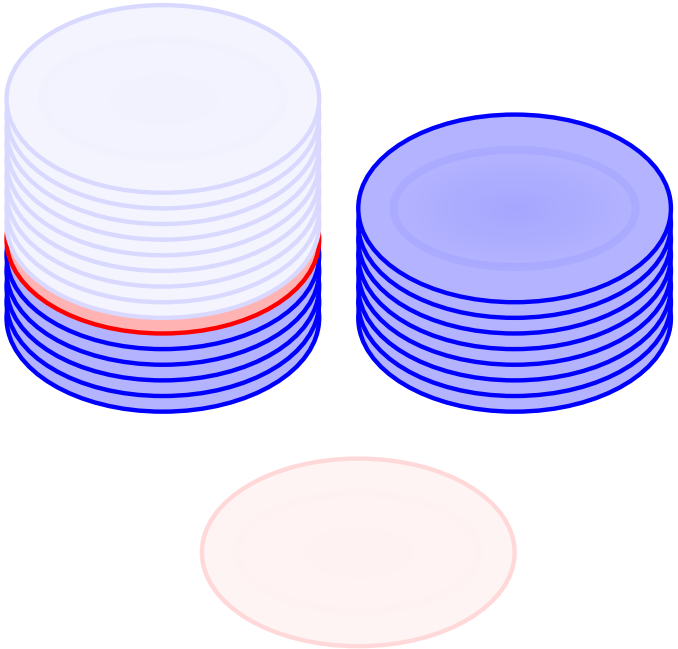
L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique



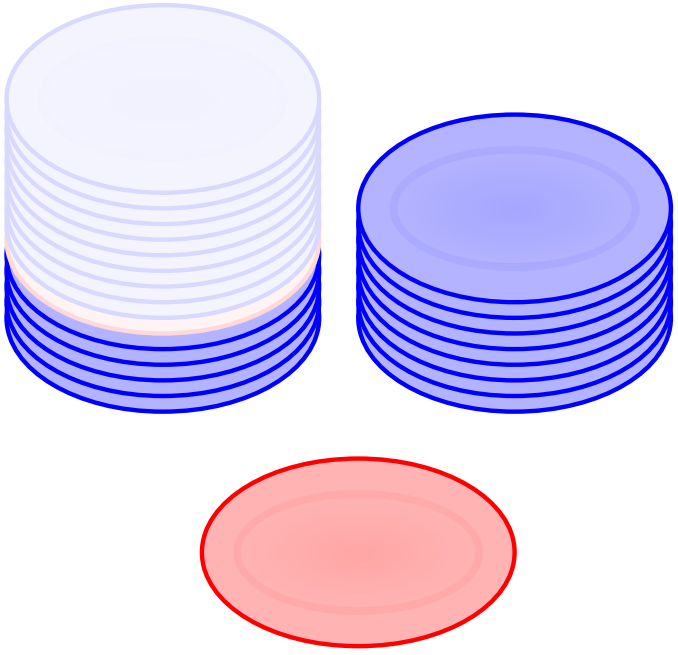
L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique



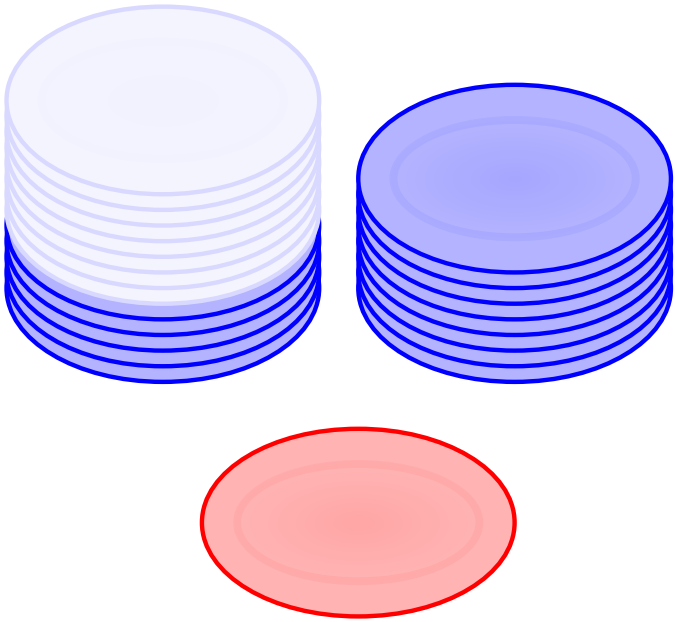
L'assiette, du sommet de la pile, est rouge.

En pratique



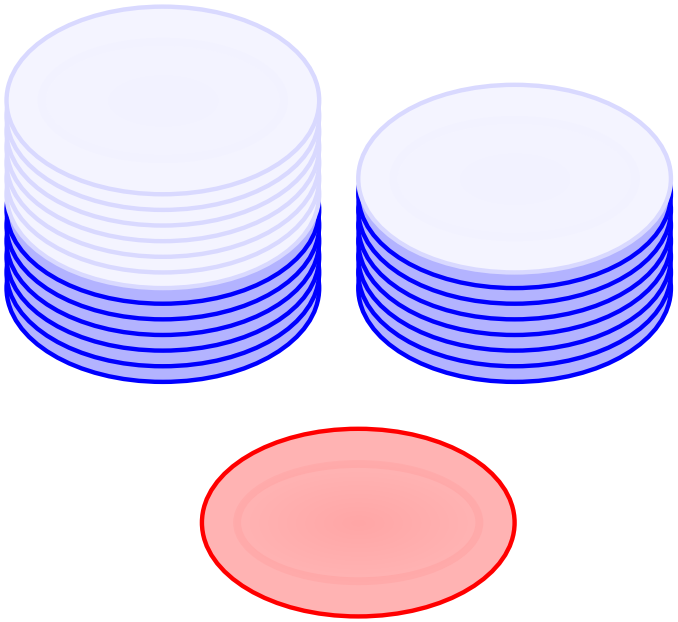
On la récupère.

En pratique



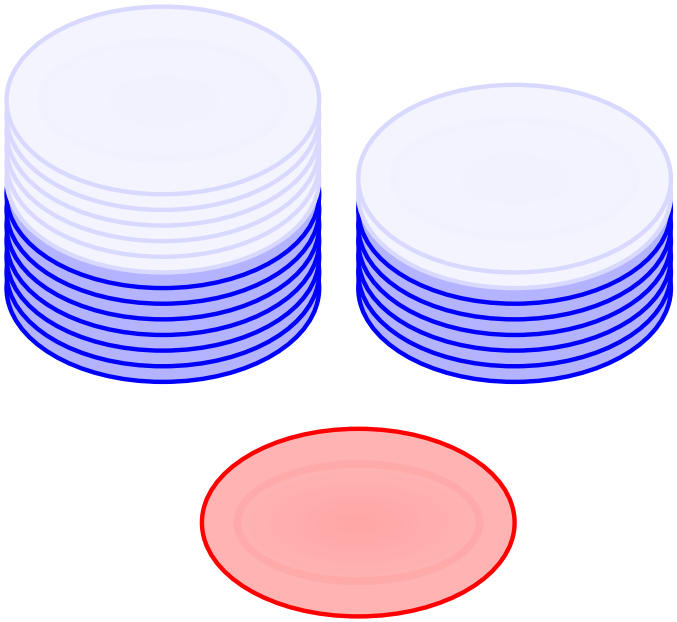
Et on reconstruit notre pile.

En pratique



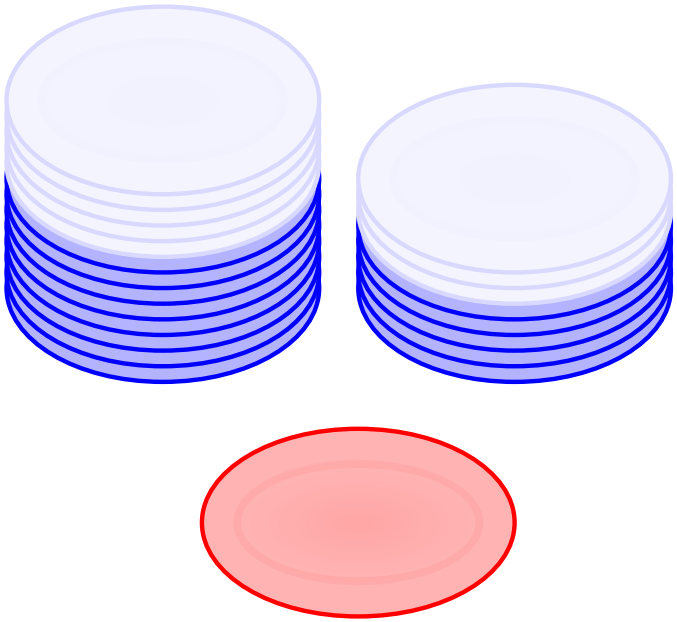
Et on reconstruit notre pile.

En pratique



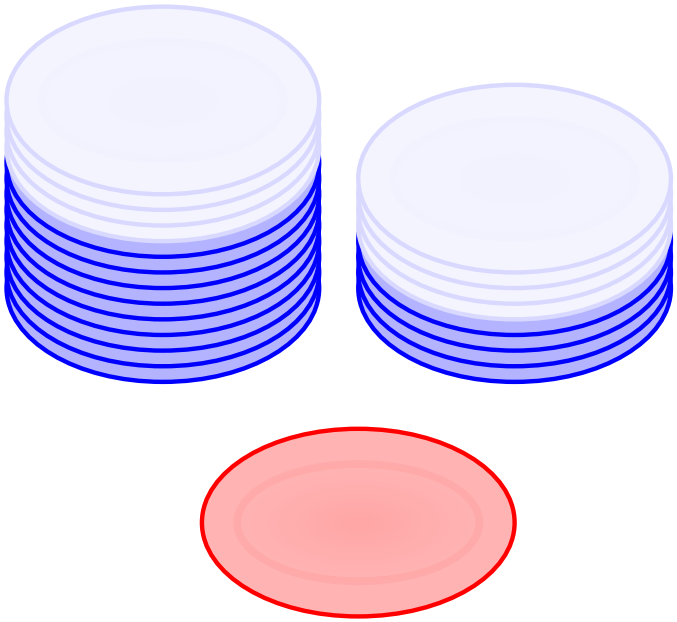
Et on reconstruit notre pile.

En pratique



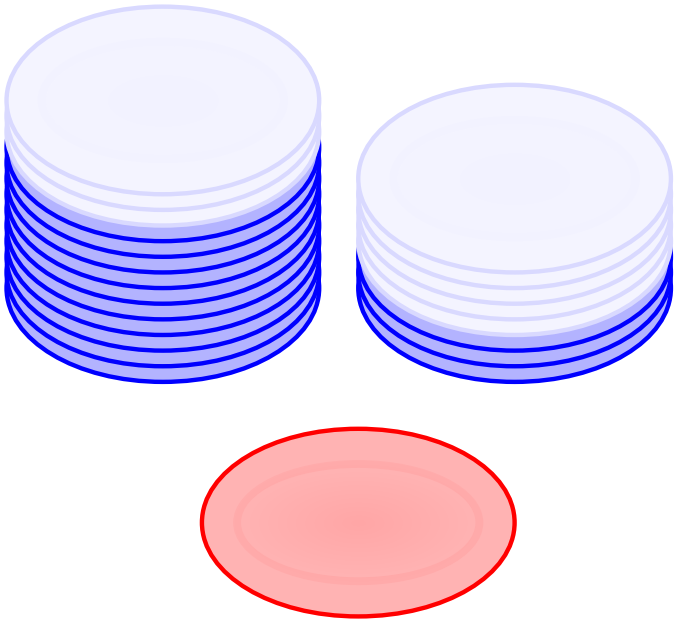
Et on reconstruit notre pile.

En pratique



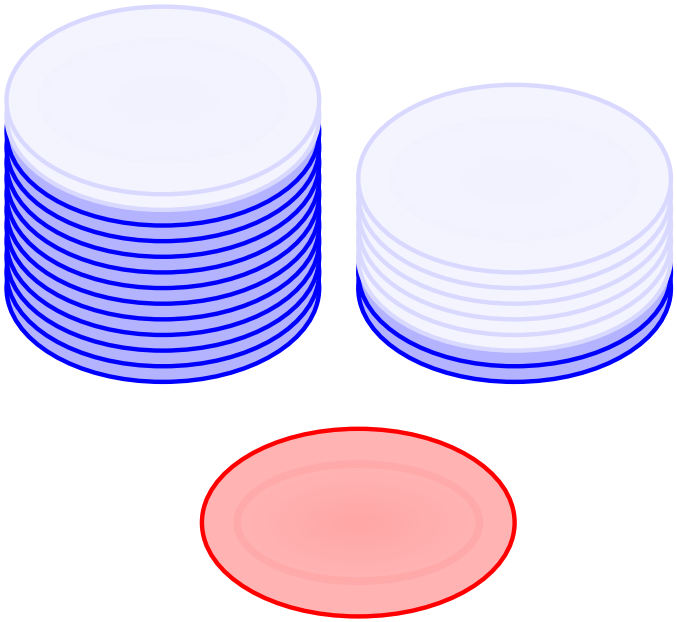
Et on reconstruit notre pile.

En pratique



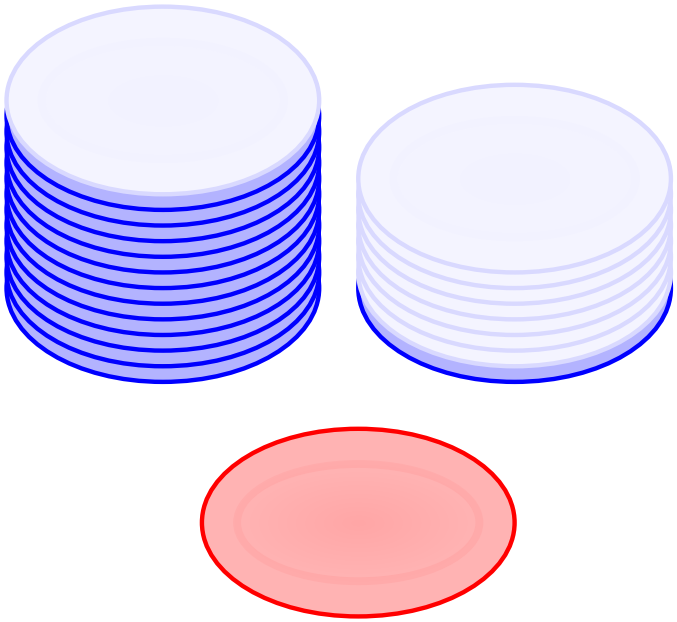
Et on reconstruit notre pile.

En pratique



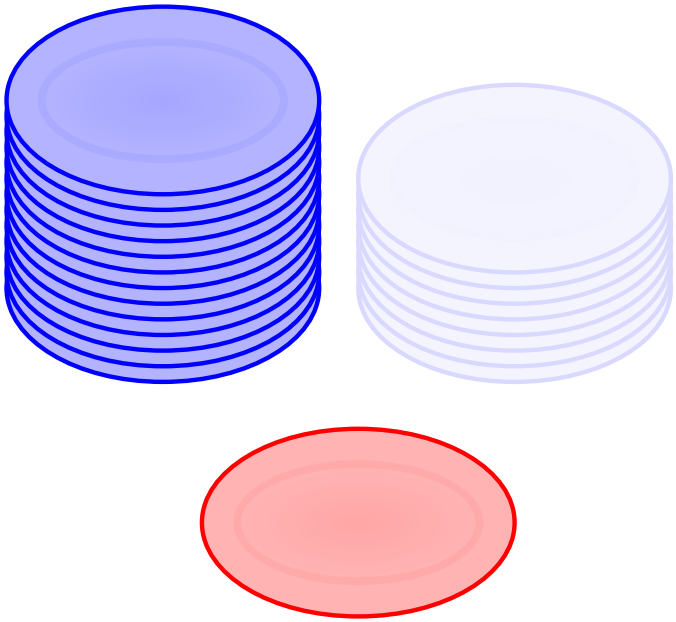
Et on reconstruit notre pile.

En pratique



Et on reconstruit notre pile.

En pratique



C’est terminé :
on a enlevé notre assiette de la pile.

Algorithme

Algorithme 10 : exemple d'une gestion de pile

```
Entrée :
début
    Q est une pile vide
    tant que P n'est pas vide faire
        Depiler(P,a)
        si a est différent de rouge
        alors
            Empiler(Q,a)
        fin
    fin
    tant que Q n'est pas vide faire
        Depiler(Q,b)
        Empiler(P,b)
    fin
fin
```

L'algorithme, ci-contre, reçoit une pile *P* et la retourne débarassée des valeurs rouges.

Remarque

L'algorithme utilise une pile auxiliaire Q permettant la reconstruction de la pile P et la restitution de ses données dans l'ordre initial.

Algorithme

Algorithme 11 : exemple d'une gestion de pile

```
Entrée :
début
    Q est une pile vide
    tant que P n'est pas vide faire
        Depiler(P,a)
        si a est différent de rouge
        alors
            Empiler(Q,a)
        fin
    fin
    tant que Q n'est pas vide faire
        Depiler(Q,b)
        Empiler(P,b)
    fin
fin
```

L'algorithme, ci-contre, reçoit une pile *P* et la retourne débarassée des valeurs rouges.

Remarque

L'algorithme utilise une pile auxiliaire Q permettant la reconstruction de la pile P et la restitution de ses données dans l'ordre initial.

De façon (très) simplifiée :

- L'exécution d'un programme peut être représentée comme le parcours d'un chemin ayant une origine et une extrémité.
- L'appel d'une procédure (ou d'une fonction) se caractérise alors par un circuit [2].
- Le processeur a alors besoin de stocker différentes informations (adresses mémoires, variables, paramètres etc . . .).

Remarque

Pour réaliser tout cela, le processeur gère une ou plusieurs piles dans lesquelles il stocke les adresses de retour des procédures et les valeurs des différentes variables.
☞ *On appellera ces données contexte de la procédure.*

2. chemin dont l'origine et l'extrémité sont confondues

De façon (très) simplifiée :

- L'exécution d'un programme peut être représentée comme le parcours d'un chemin ayant une origine et une extrémité.
- L'appel d'une procédure (ou d'une fonction) se caractérise alors par un circuit [2].
- Le processeur a alors besoin de stocker différentes informations (adresses mémoires, variables, paramètres etc . . .).

Remarque

Pour réaliser tout cela, le processeur gère une ou plusieurs piles dans lesquelles il stocke les adresses de retour des procédures et les valeurs des différentes variables.
☞ *On appellera ces données contexte de la procédure.*

2. chemin dont l'origine et l'extrémité sont confondues

De façon (très) simplifiée :

- L'exécution d'un programme peut être représentée comme le parcours d'un chemin ayant une origine et une extrémité.
- L'appel d'une procédure (ou d'une fonction) se caractérise alors par un circuit [2].
- Le processeur a alors besoin de stocker différentes informations (adresses mémoires, variables, paramètres etc . . .).

Remarque

Pour réaliser tout cela, le processeur gère une ou plusieurs piles dans lesquelles il stocke les adresses de retour des procédures et les valeurs des différentes variables.
☞ *On appellera ces données contexte de la procédure.*

2. chemin dont l'origine et l'extrémité sont confondues

De façon (très) simplifiée :

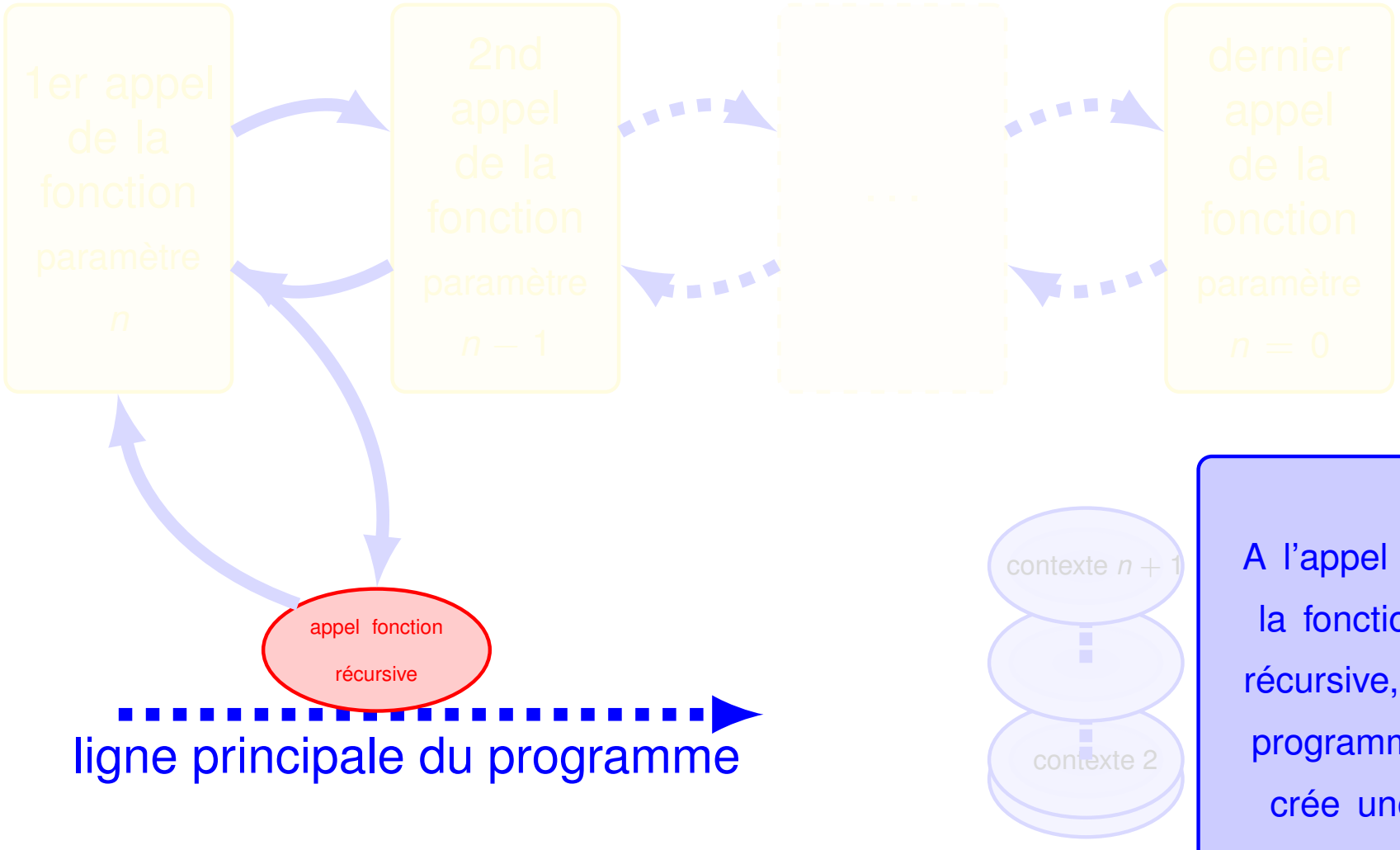
- L'exécution d'un programme peut être représentée comme le parcours d'un chemin ayant une origine et une extrémité.
- L'appel d'une procédure (ou d'une fonction) se caractérise alors par un circuit [2].
- Le processeur a alors besoin de stocker différentes informations (adresses mémoires, variables, paramètres etc . . .).

Remarque

Pour réaliser tout cela, le processeur gère une ou plusieurs piles dans lesquelles il stocke les adresses de retour des procédures et les valeurs des différentes variables.
☞ *On appellera ces données contexte de la procédure.*

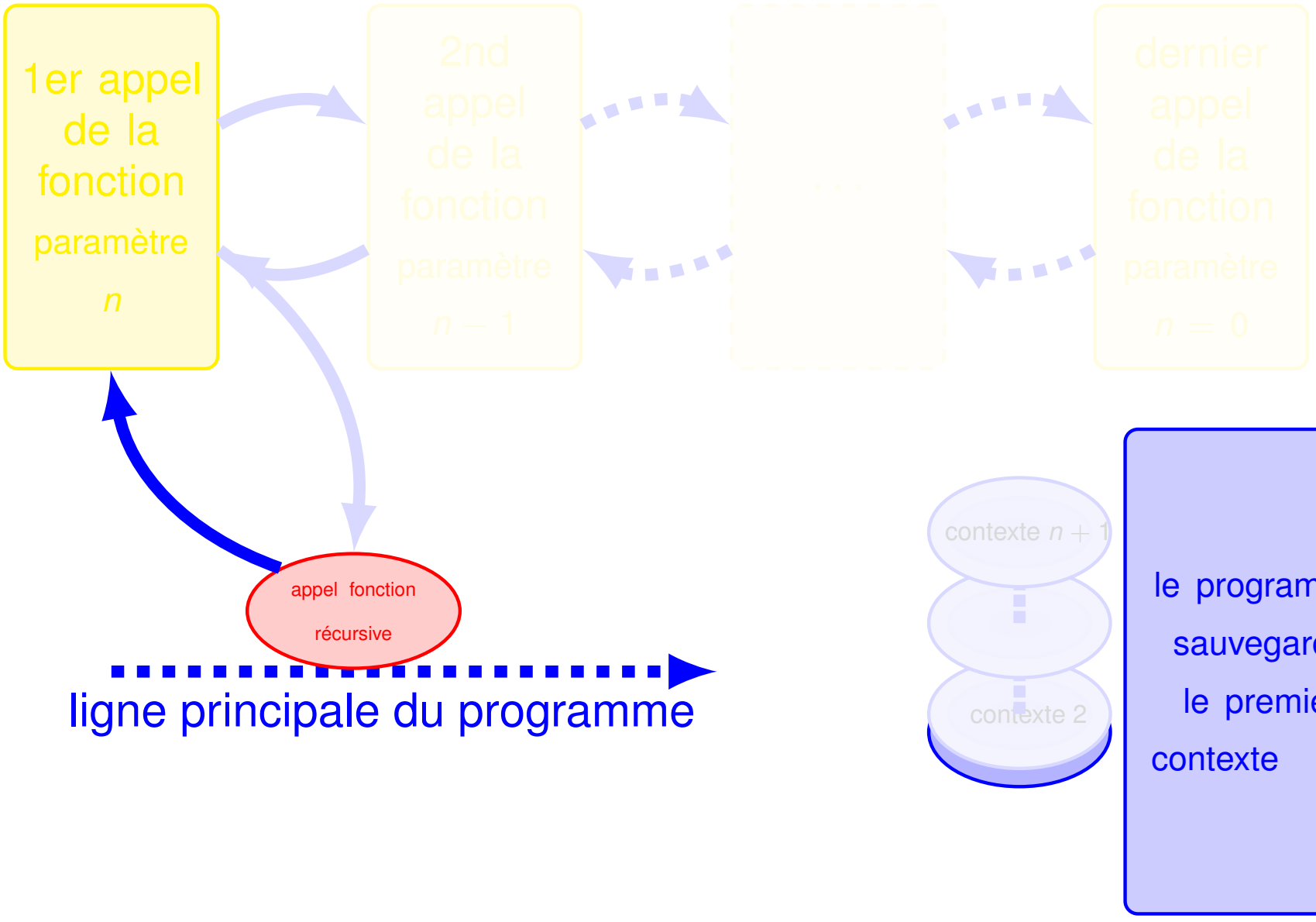
2. chemin dont l'origine et l'extrémité sont confondues

Récursivité et pile de programme

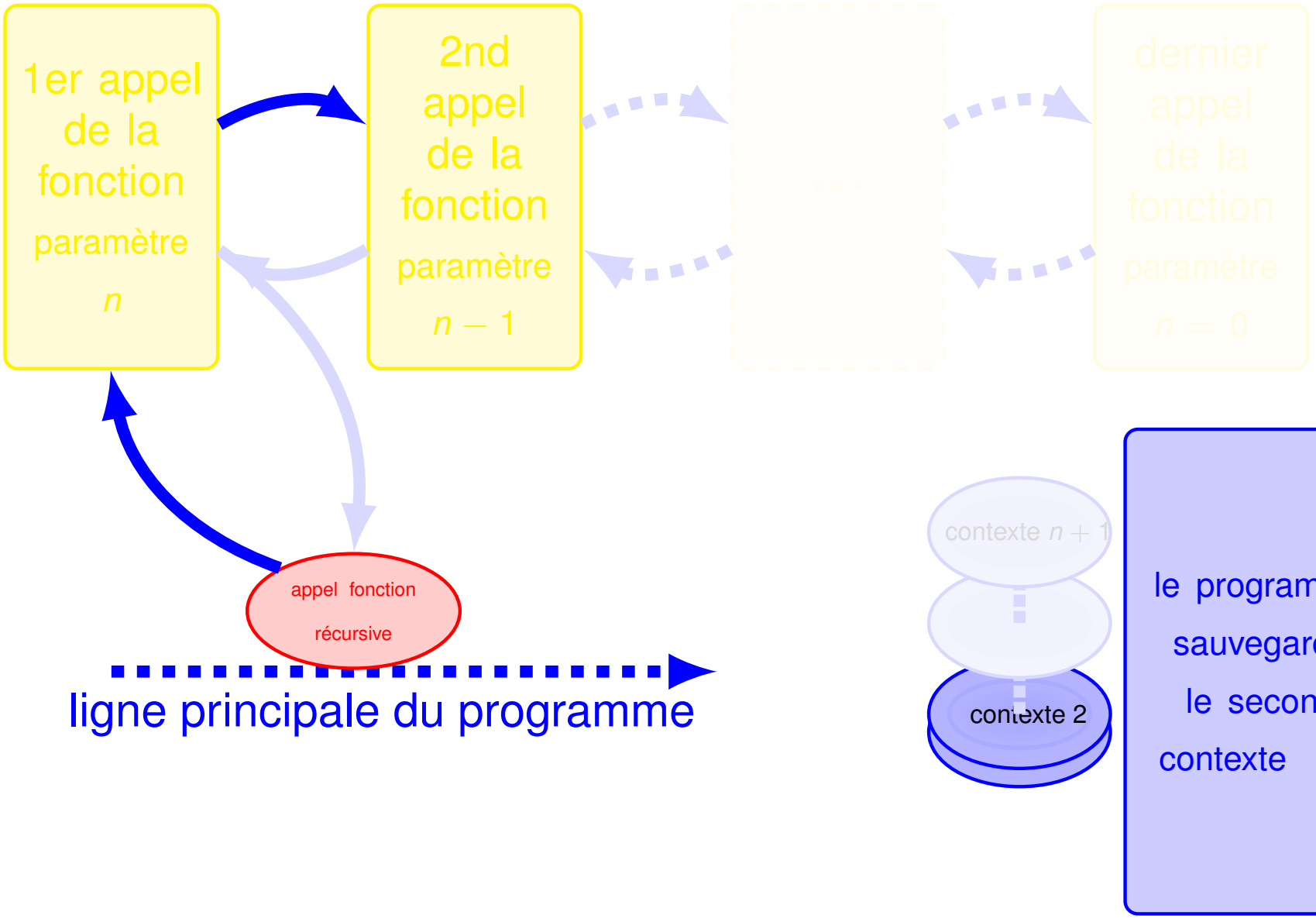


A l'appel de la fonction récursive, le programme crée une pile

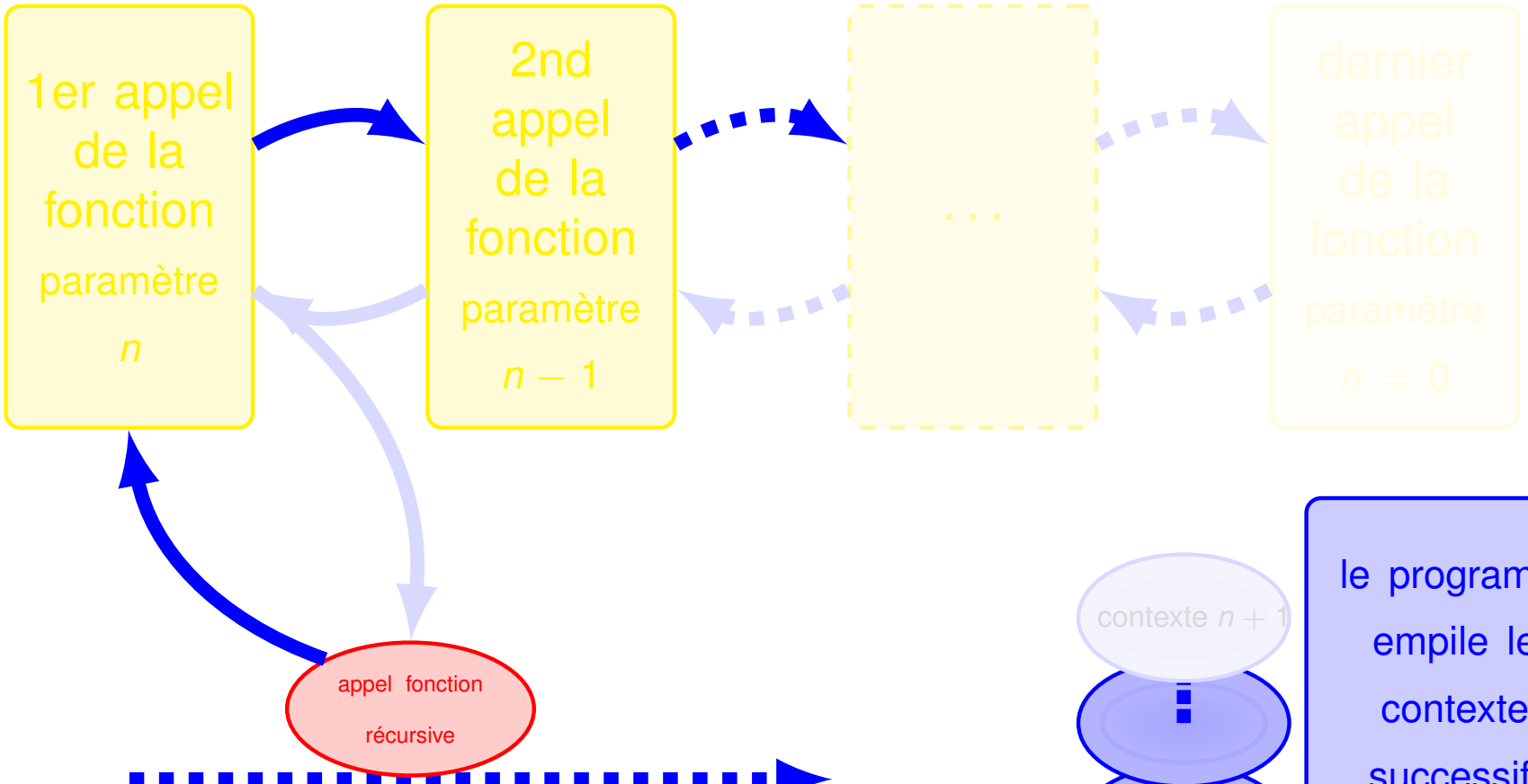
Récursivité et pile de programme



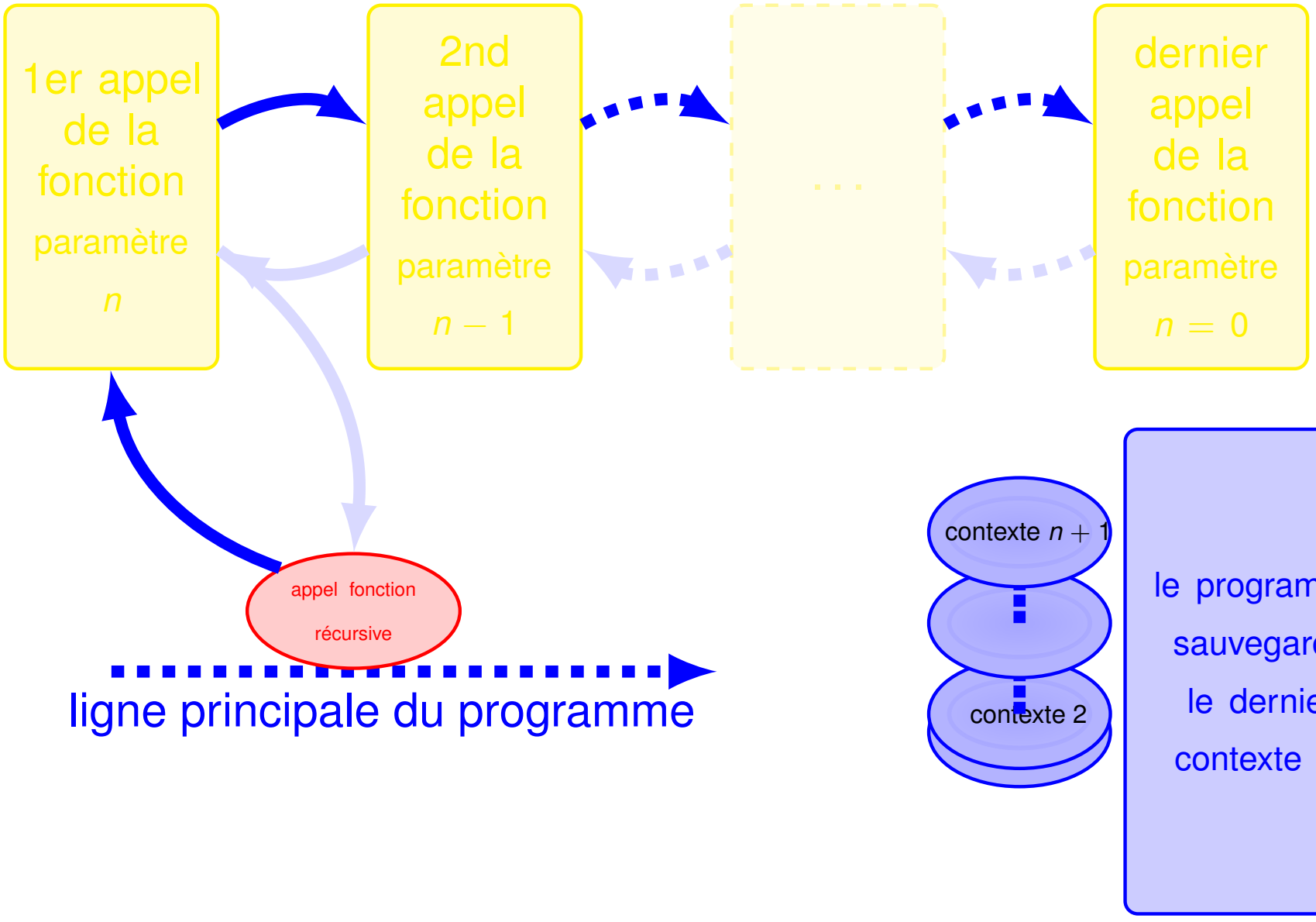
Récursivité et pile de programme



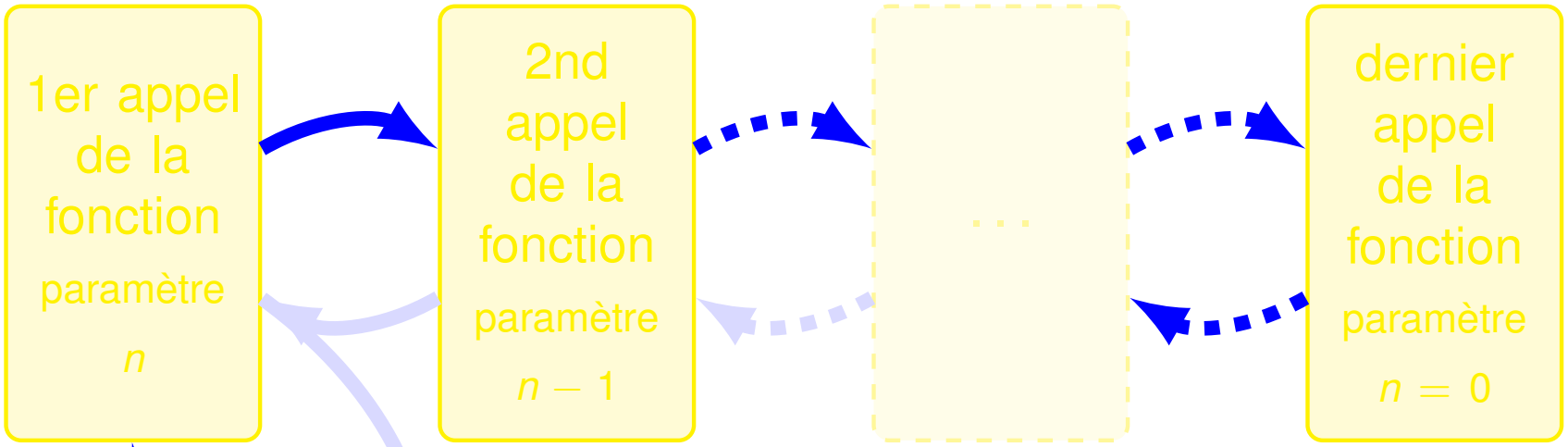
Récursivité et pile de programme

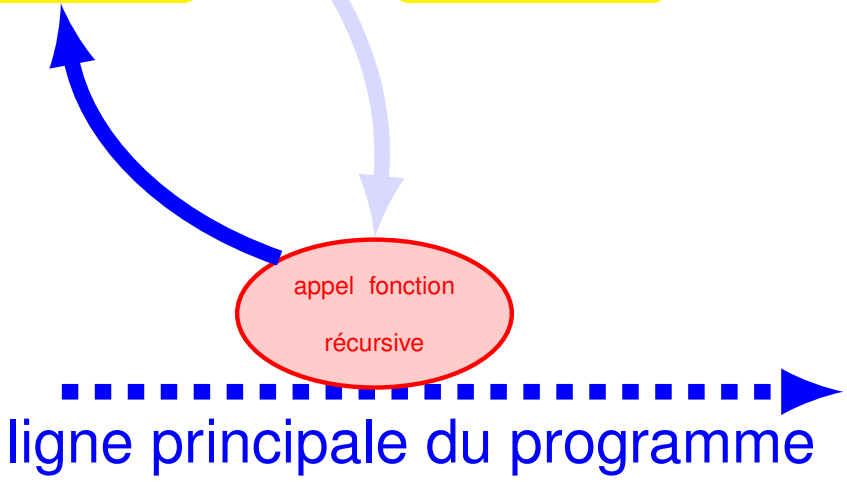


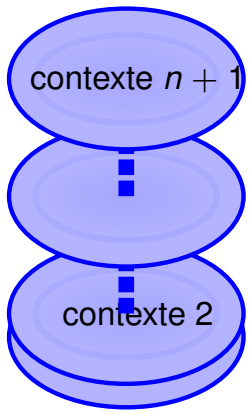
Récursivité et pile de programme



Récursivité et pile de programme

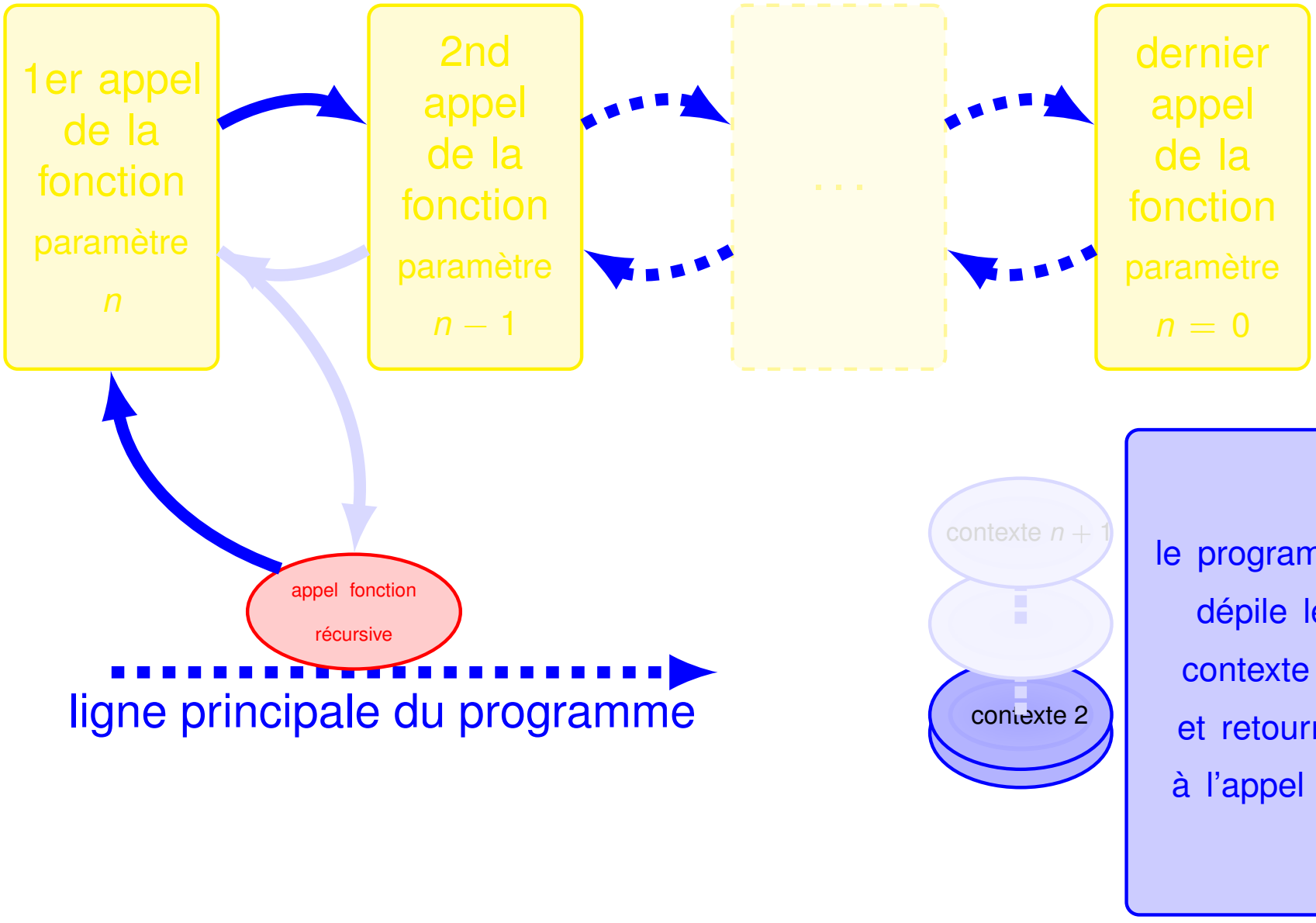




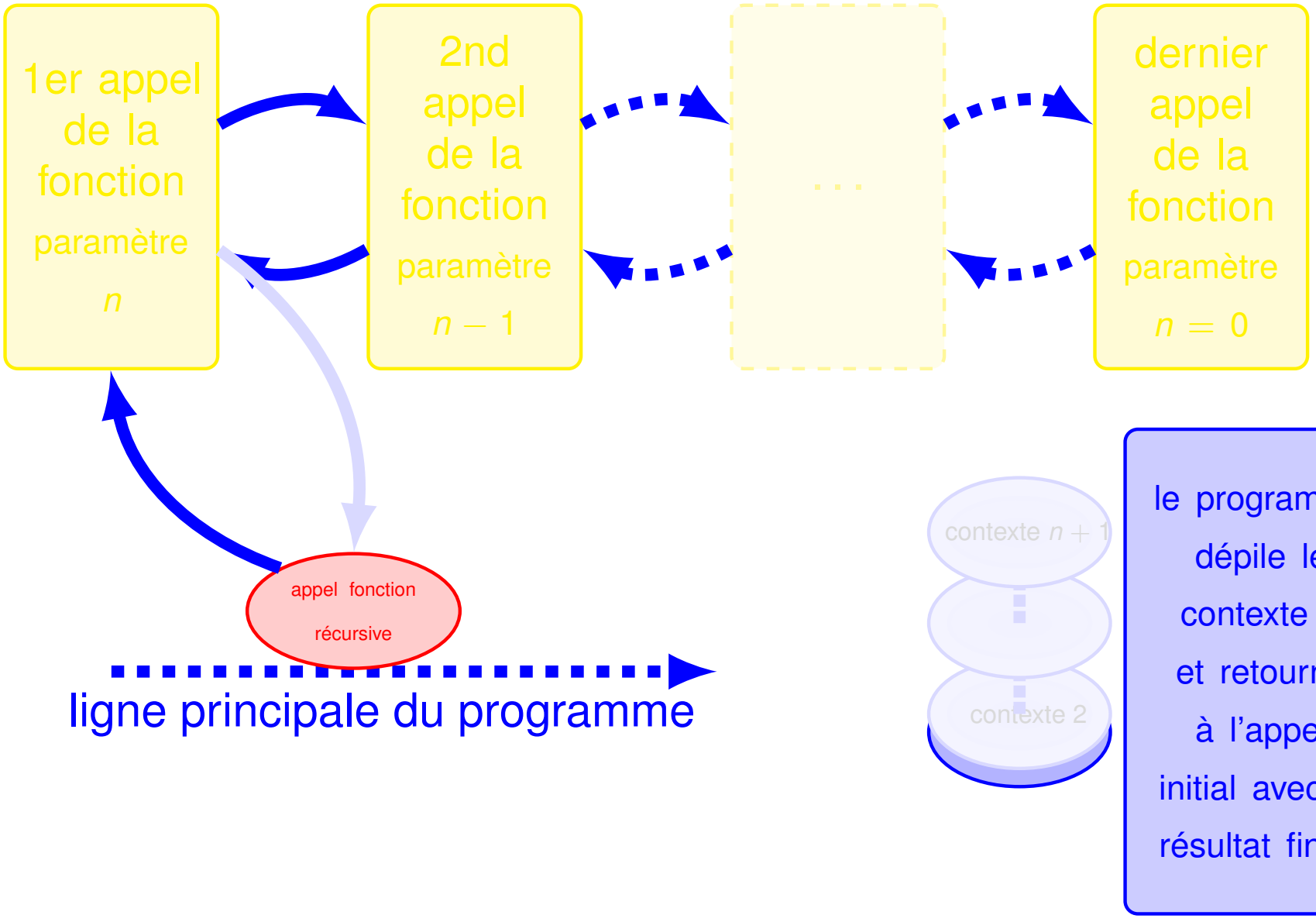


le programme dépile les contextes avec passage des résultats

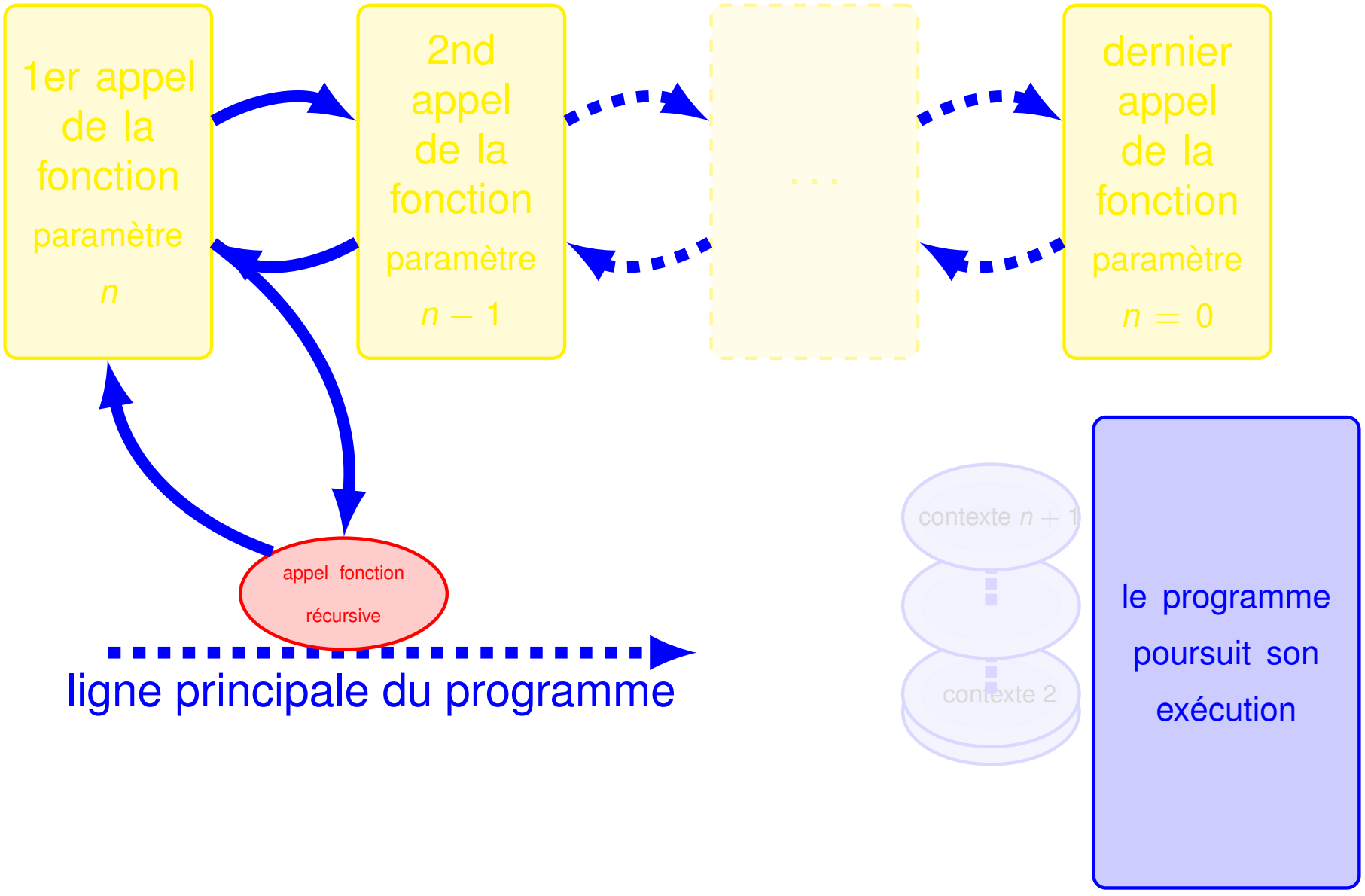
Récursivité et pile de programme



Récursivité et pile de programme

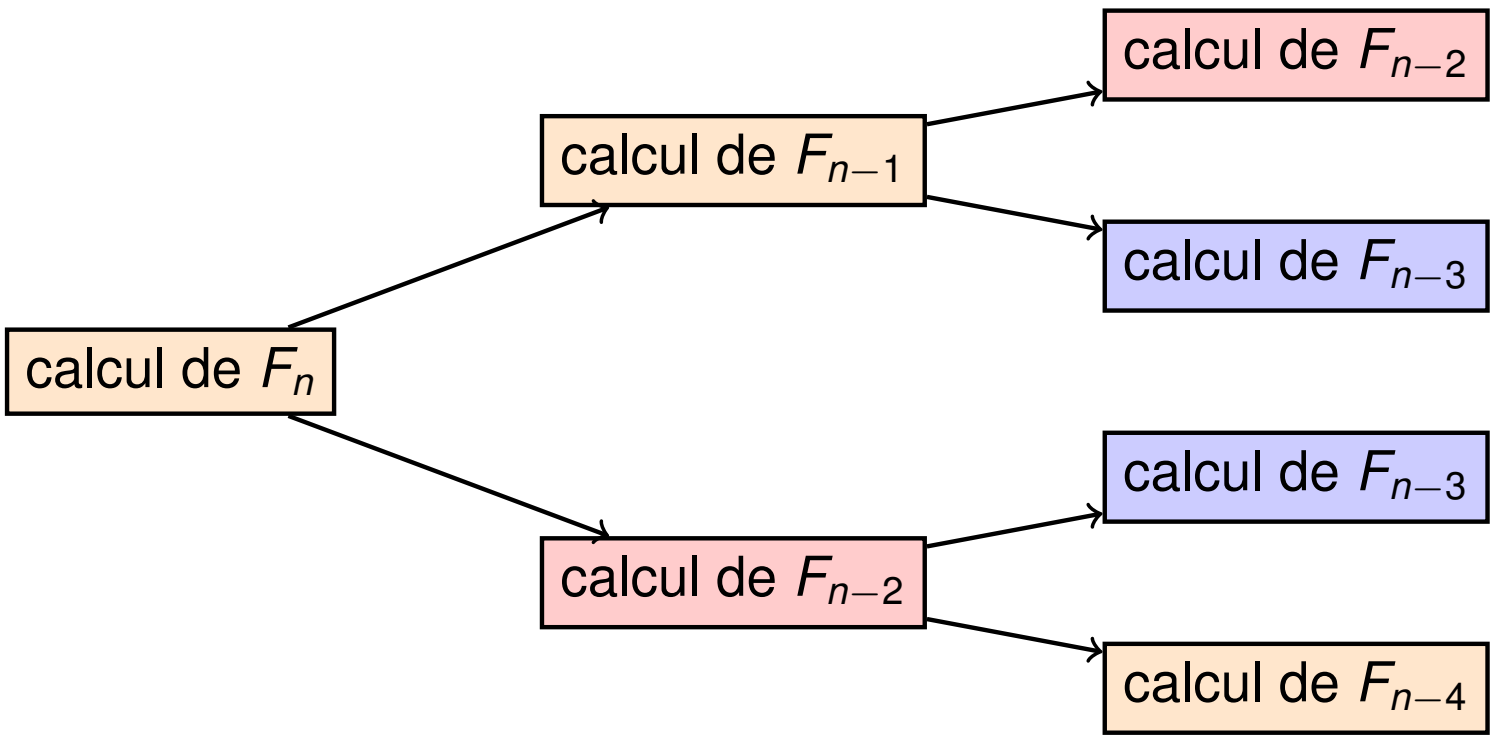


Récursivité et pile de programme



Retour à la site de Fibonacci

Dans notre algorithme récursif de Fibonacci, de nombreux appels récursifs sont redondants, comme l'illustre le schéma ci-dessous :



Un tel arbre donne un nombre de chemins de l'ordre de 2^n.
Par exemple, 2^{50} appels récursifs, nécessitent une pile d'au moins 10^{15} données.
☞ *pour information,* $1\,To \approx 10^{12}$ *octets.*

Remarque

La traduction de cet algorithme va ainsi générer de gros problèmes de temps de calcul et de mémoire, comme on l'a vu.

Un tel arbre donne un nombre de chemins de l'ordre de 2^n.
Par exemple, 2^{50} appels récursifs, nécessitent une pile d'au moins 10^{15} données.
☞ *pour information,* $1\,To \approx 10^{12}$ *octets.*

Remarque

La traduction de cet algorithme va ainsi générer de gros problèmes de temps de calcul et de mémoire, comme on l'a vu.

Euclide récursif

Fonction `EuclideRec(`a, b`)`

début
  si $b == 0$ **alors**
    retourner : a
  fin
  Donner à c la valeur $a \mod b$
  retourner : EuclideRec(b, c)
fin

- Cet algorihme est basé sur la propriété : $\text{pgcd}(a, b) = \text{pgcd}(b, a \mod b)$
- L'appel récursif est la dernière instruction de la fonction : on parle de récursivité terminale

Factorielle terminale

Fonction `FactoTerm(`n, acc`)`

début
  si $n <= 1$ **alors**
    retourner : acc
  fin
  retourner : FactoTerm($n-1, n \times acc$)
fin

- On appelle `FactoTerm(`n, 1`)`
- On voit ici l'utilisation d'une variable auxiliaire *acc* qui joue le rôle d'accumulateur des calculs intermédiaires

Encore un accumulateur

Fonction `UTerm(`n, acc`)`

début
  si $n <= 0$ **alors**
    retourner : acc
  fin
  retourner : UTerm$(n-1, 2 \times acc + 3)$
fin

- On appelle `UTerm(`n, 1`)`
- Tous les algorithmes récursifs peuvent être réécrits (plus ou moins facilement) sous une forme terminale
- Exercice : Écrire la fonction `FiboTerm(`n, $avtder$, der`)`

À quoi ça sert ?

À écrire du code plus efficace

- Un algorithme récursif est facile à écrire mais souvent peu efficace à l'exécution
- Un algorithme itératif est plus difficile à écrire mais souvent plus efficace à l'exécution
- Une récursion terminale est facile à dérécusiver
- Certains compilateurs sont capables de détecter et de dérécursiver automatiquement les récursions terminales

Dérécursiver

Fonction `FRec(`*param*`)`

début
  si *arret*(*param*) **alors**
    retourner : *finir*(*param*)
  fin
  Traiter(*param*)
  retourner : FRec(*changer*(*param*))
fin

Fonction `FIter(`*param*`)`

début
  tant que non *(arret*(*param*)*)*
  faire
    Traiter(*param*)
    Donner à *param* la valeur
    changer(*param*)
  fin
  retourner : *finir*(*param*)
fin

Dérécursiver

Fonction `FRec(`*param*`)`

début
  si *arret*(*param*) **alors**
    retourner : *finir*(*param*)
  fin
  Traiter(*param*)
  retourner : FRec(*changer*(*param*))
fin

Fonction `FIter(`*param*`)`

début
  tant que non *(arret*(*param*)*)*
  faire
    Traiter(*param*)
    Donner à *param* la valeur
    changer(*param*)
  fin
  retourner : *finir*(*param*)
fin

La récursivité est indispensable en algorithmique :

- traduction de la récurrence en mathématiques
- écritures simples des algorithmes
- solutions simples de problèmes complexes
- permet la mise en œuvre de méthodes très puissantes

On fera attention :

- aux conditions de terminaison
- à la gestion de la taille de la pile d'exécution
- à certains problèmes de complexités à surveiller
- à l'efficacité parfois moindre qu'une version itérative
  ☞ *on peut toujours transformer l'algorithme en version itérative (dérécursiver)*

La récursivité est indispensable en algorithmique :

- traduction de la récurrence en mathématiques
- écritures simples des algorithmes
- solutions simples de problèmes complexes
- permet la mise en œuvre de méthodes très puissantes

On fera attention :

- aux conditions de terminaison
- à la gestion de la taille de la pile d'exécution
- à certains problèmes de complexités à surveiller
- à l'efficacité parfois moindre qu'une version itérative
  ☞ *on peut toujours transformer l'algorithme en version itérative (dérécursiver)*

La récursivité est indispensable en algorithmique :

- traduction de la récurrence en mathématiques
- écritures simples des algorithmes
- solutions simples de problèmes complexes
- permet la mise en œuvre de méthodes très puissantes

On fera attention :

- aux conditions de terminaison
- à la gestion de la taille de la pile d'exécution
- à certains problèmes de complexités à surveiller
- à l'efficacité parfois moindre qu'une version itérative
  ☞ *on peut toujours transformer l'algorithme en version itérative (dérécursiver)*

La récursivité est indispensable en algorithmique :

- traduction de la récurrence en mathématiques
- écritures simples des algorithmes
- solutions simples de problèmes complexes
- permet la mise en œuvre de méthodes très puissantes

On fera attention :

- aux conditions de terminaison
- à la gestion de la taille de la pile d'exécution
- à certains problèmes de complexités à surveiller
- à l'efficacité parfois moindre qu'une version itérative
  ☞ *on peut toujours transformer l'algorithme en version itérative (dérécursiver)*

La récursivité est indispensable en algorithmique :

- traduction de la récurrence en mathématiques
- écritures simples des algorithmes
- solutions simples de problèmes complexes
- permet la mise en œuvre de méthodes très puissantes

On fera attention :

- aux conditions de terminaison
- à la gestion de la taille de la pile d'exécution
- à certains problèmes de complexités à surveiller
- à l'efficacité parfois moindre qu'une version itérative

  ☞ *on peut toujours transformer l'algorithme en version itérative (dérécursiver)*

La récursivité est indispensable en algorithmique :

- traduction de la récurrence en mathématiques
- écritures simples des algorithmes
- solutions simples de problèmes complexes
- permet la mise en œuvre de méthodes très puissantes

On fera attention :

- aux conditions de terminaison
- à la gestion de la taille de la pile d'exécution
- à certains problèmes de complexités à surveiller
- à l'efficacité parfois moindre qu'une version itérative
  ☞ *on peut toujours transformer l'algorithme en version itérative (dérécursiver)*

La récursivité est indispensable en algorithmique :

- traduction de la récurrence en mathématiques
- écritures simples des algorithmes
- solutions simples de problèmes complexes
- permet la mise en œuvre de méthodes très puissantes

On fera attention :

- aux conditions de terminaison
- à la gestion de la taille de la pile d'exécution
- à certains problèmes de complexités à surveiller
- à l'efficacité parfois moindre qu'une version itérative
  ☞ *on peut toujours transformer l'algorithme en version itérative (dérécursiver)*

La récursivité est indispensable en algorithmique :

- traduction de la récurrence en mathématiques
- écritures simples des algorithmes
- solutions simples de problèmes complexes
- permet la mise en œuvre de méthodes très puissantes

On fera attention :

- aux conditions de terminaison
- à la gestion de la taille de la pile d'exécution
- à certains problèmes de complexités à surveiller
- à l'efficacité parfois moindre qu'une version itérative
  ☞ *on peut toujours transformer l'algorithme en version itérative (dérécursiver)*

FIN