Récursivité

Philippe Lac
(philippe.lac@ac-clermont.fr)
Malika More
(malika.more@u-clermont1.fr)

IREM Clermont-Ferrand

Stage Algorithmique

Année 2010-2011

Contenu

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples
2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme
3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner
4. Conclusion

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples
2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme
3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner
4. Conclusion

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples
2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme
3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner
4. Conclusion

Commençons par un petit exercice de maths du lycée.
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 la journée précédente, 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 $2 \times Utmp + 3$;
  fin
  Retourner *Utmp*;
fin

Remarque

On remonte 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 :

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

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.

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples
2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme
3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner
4. Conclusion

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.

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples
2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme
3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner
4. Conclusion

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
        on retourne la valeur 1;
        % On traite le cas trivial correspondant au
          premier terme de la suite  %
    sinon
        on retourne la valeur 2 × Urec(n − 1) + 3;
        % ici, n ≠ 0 donc on retourne 2× la valeur
          du terme précédent +3 %
    fin
fin
```

Remarque

- 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.*

Traduction de l'algorithme précédent :

SCILAB

```
function Urec=Urec(n)

   if n=0 then Urec=1
            else Urec=2*Urec(n-1)+3
   end

endfunction
```

XCAS

```
Urec(n) :={
si (n==0) alors
                    return 1
               sinon
                    return 2*Urec(n-1)+3 ;
fsi ; }
```

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 XCAS ou SCILAB.

Fonction FactoRec(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**
    on retourne la valeur 1;
    `% On traite le cas trivial correspondant à 0! %`
  sinon
    on retourne la valeur $n \times$ FactoRec($n - 1$);
    `% ici,` $n \neq 0$ `donc on retourne` $n\times$ `la valeur du terme précédent %`
  fin
fin

☞ *un grand classique !*

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
```

Il est à ce stade intéressant de traduire ces algorithmes,
☞ *les fichiers sont disponibles ?ici ?*.
et de comparer les temps d'exécution des versions itératives et récursives.

On pourra utiliser les instructions suivantes :

SCILAB

- la séquence d'instructions `tic();FibIter(10);toc()` renvoie le temps nécessaire à l'exécution de `FibIter(10)` ;
- la séquence `T=zeros(10,1);for i=2:2:20 do, tic();FibIter(i);T(i/2)=toc();end;` permet d'obtenir plusieurs temps d'exécution stockés dans le tableau `T` ;
- l'instruction `plot2d(T)` donne une représentation graphique des valeurs de `T`.

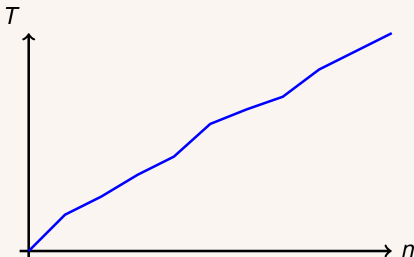
XCAS

- la séquence d'instructions `time(FibRec(10))[0]` renvoie le temps nécessaire à l'exécution de `FibIter(10)`
- la séquence `T=seq(time(FibRec(k))[0],k,2,20,2)` permet d'obtenir plusieurs temps d'exécution stockés dans la liste `T` ;
- l'instruction `plotlist(T)` donne une représentation graphique des valeurs de `T`.

Quelques temps de calculs

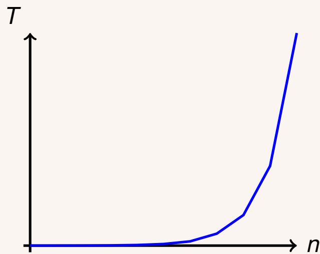
Itératif

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

- Dans l'exemple précédent, l'algorithme récursif s'est révélé clairement peu efficace par rapport à sa version itérative.
- De par sa conception, cet algorithme soulève d'importants problèmes de complexités qui seront développés dans le prochain chapitre.
- Mais une programmation récursive peut générer aussi d'autres problèmes et des effets de bord dont il vaut mieux avoir conscience.

Nous allons donner quelques éléments d'explication.

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples

2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme

3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner

4. Conclusion

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples
2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme
3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner
4. Conclusion

La 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

On dispose d'une pile d'assiettes.
Cette pile se compose d'assiettes bleues en dehors d'une assiette qui est rouge.
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 procédures[2] suivantes :

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

2. on utilise ici le terme procédure, car les paramètres peuvent être modifiés par la séquence d'instructions appelée

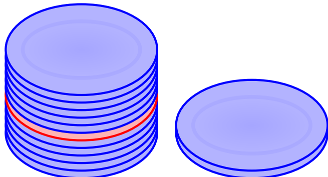
En pratique

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

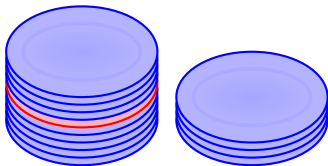


L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique

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

En pratique

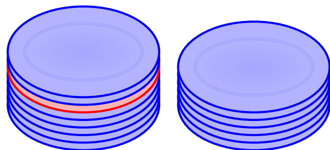


L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique

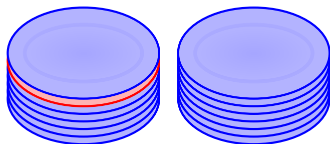
L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique



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

En pratique

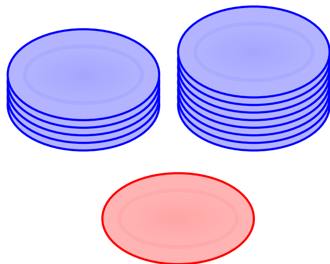


L'assiette, du sommet de la pile, est bleue : on l'enlève et on la place sur la pile voisine.

En pratique

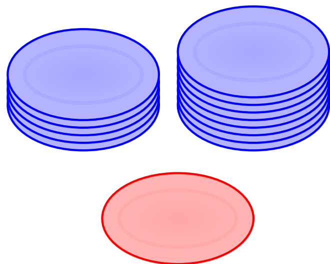
L'assiette, du sommet de la pile, est rouge.

En pratique



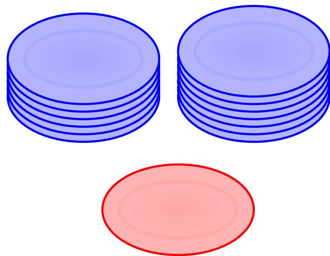
On la récupère.

En pratique



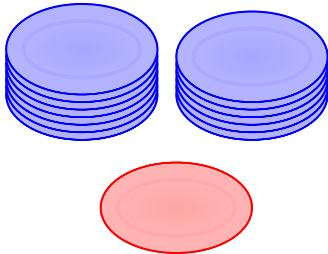
Et on reconstruit notre pile.

En pratique



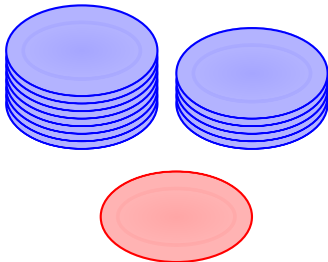
Et on reconstruit notre pile.

En pratique



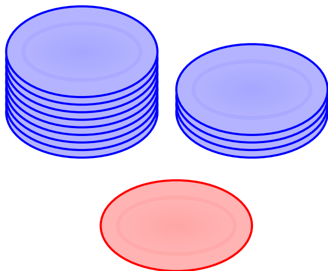
Et on reconstruit notre pile.

En pratique



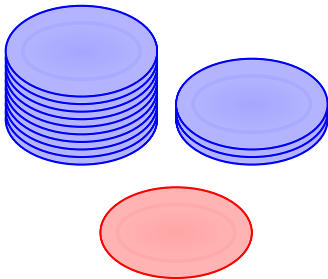
Et on reconstruit notre pile.

En pratique



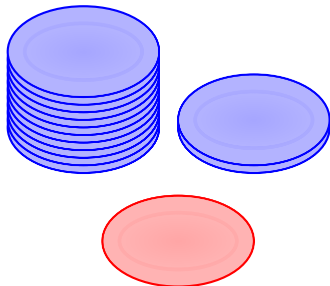
Et on reconstruit notre pile.

En pratique



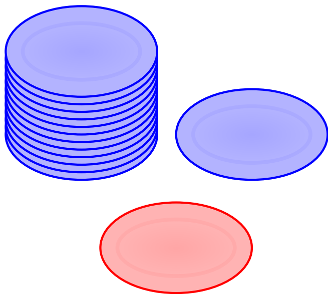
Et on reconstruit notre pile.

En pratique



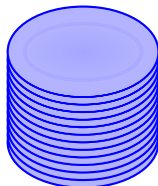
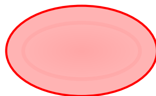
Et on reconstruit notre pile.

En pratique



Et on reconstruit notre pile.

En pratique

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

Algorithme

Algorithme 1: exemple d'une gestion de pile

Entrée :
début
  Q est une pile vide;
  Depile(P,a);
  tant que *a différent de rouge* **faire**
    Empile(Q,a);
    Depile(P,a)
  fin
  tant que *Q n'est pas vide* **faire**
    Depile(Q,b);
    Empile(P,b)
  fin
fin

L'algorithme, ci-contre, reçoit une pile P et la restitue débarassée de sa valeur rouge.
La pile est supposée n'être pas vide et possèder une valeur rouge.

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples
2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme
3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner
4. Conclusion

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[3].
- Le processeur a alors besoin de stocker différentes informations (adresses mémoires, variables, paramètres etc . . .).

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.*

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

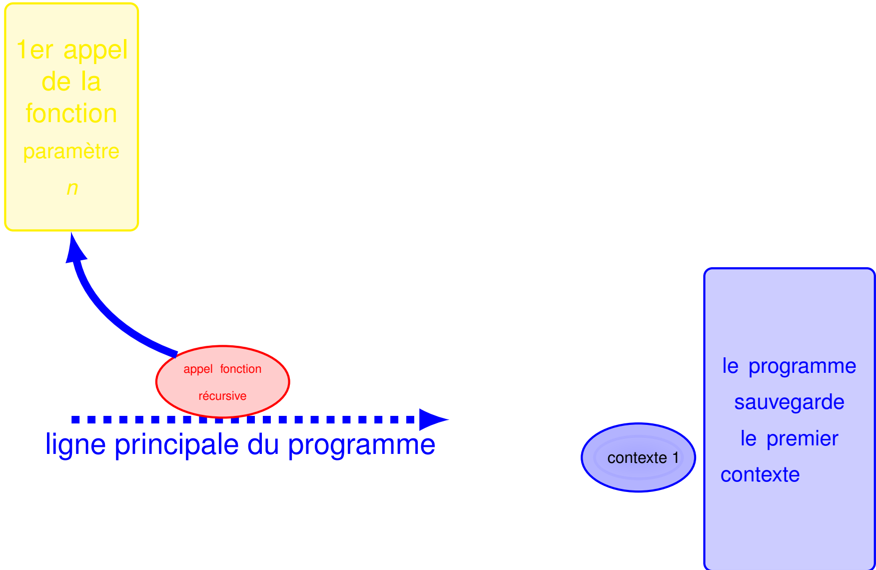
Récursivité et pile de programme

appel fonction récursive

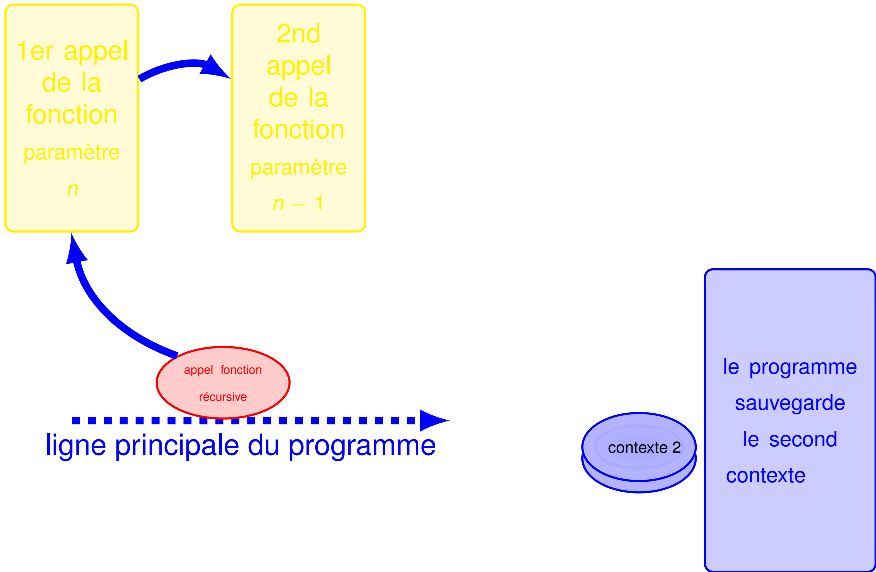
ligne principale du programme

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

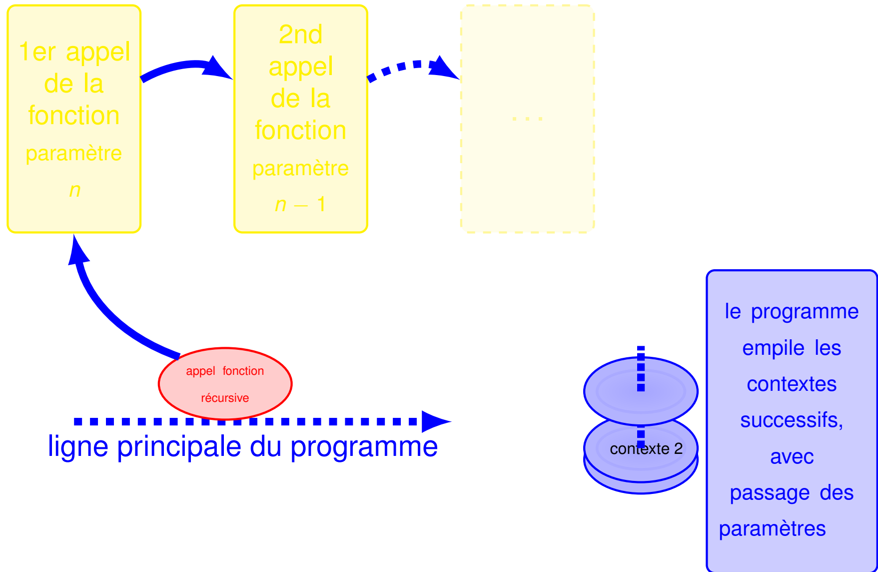
Récursivité et pile de programme



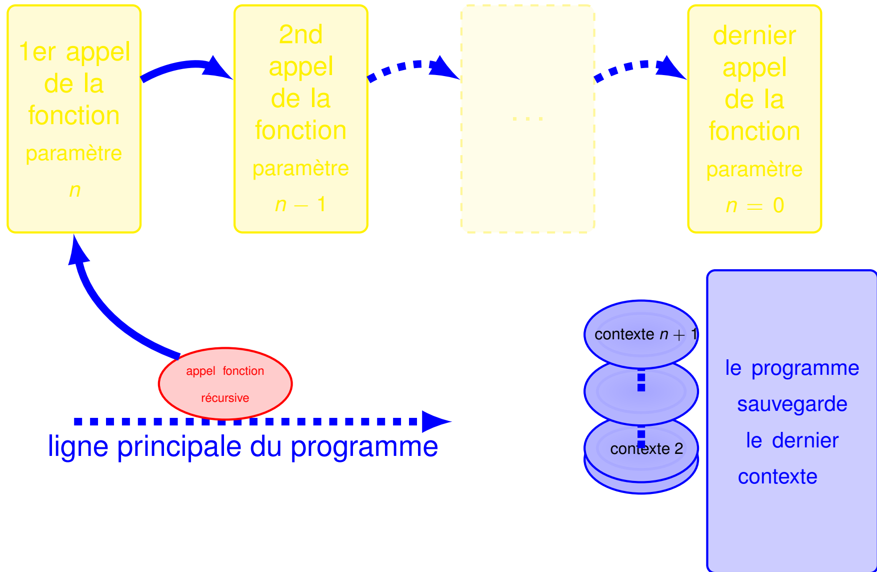
Récursivité et pile de programme



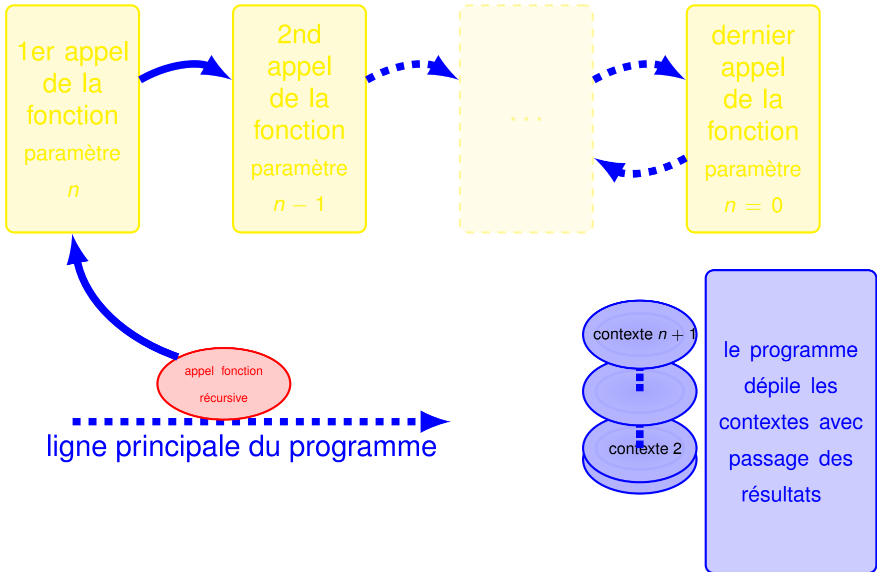
Récursivité et pile de programme



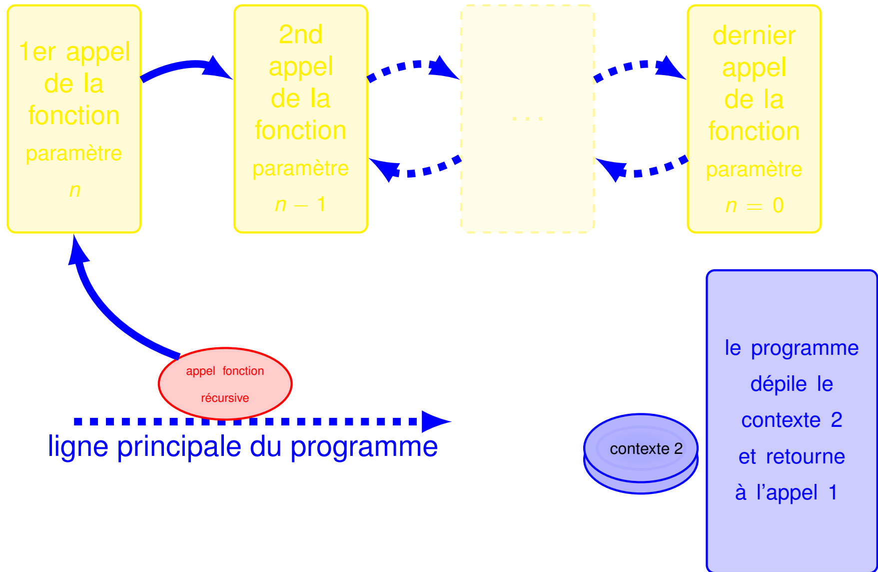
Récursivité et pile de programme



Récursivité et pile de programme



Récursivité et pile de programme



Récursivité et pile de programme

1er appel de la fonction paramètre n

2nd appel de la fonction paramètre $n-1$

...

dernier appel de la fonction paramètre $n=0$

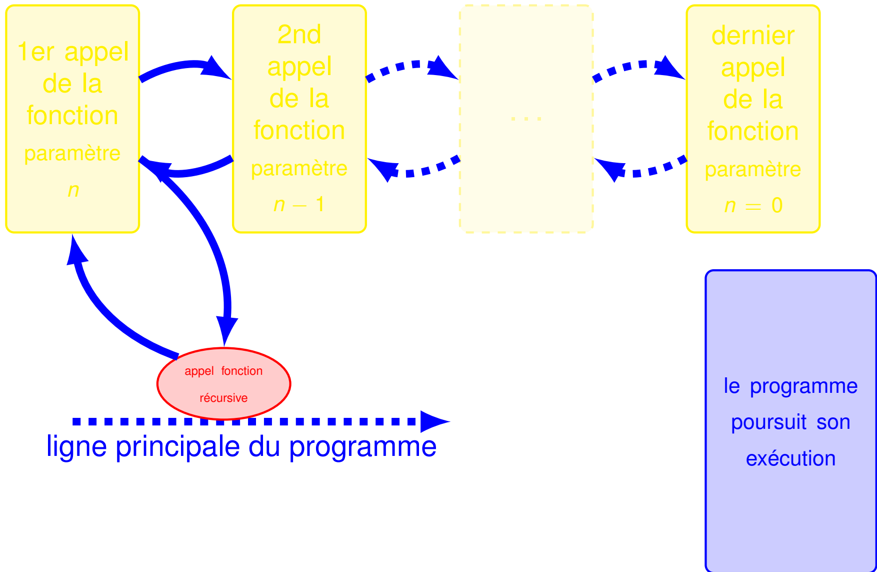
appel fonction récursive

ligne principale du programme

contexte 1

le programme dépile le contexte 1 et retourne à l'appel initial avec le résultat final

Récursivité et pile de programme



Retour à la site de Fibonacci

Nous avons maintenant un premier élément de réponse à la mauvaise efficacité de notre algorithme récursif de Fibonacci.
En effet plusieurs appels récursifs sont redondants, comme l'illustre le schéma ci-dessous :

calcul de F_n → calcul de F_{n-1} → calcul de F_{n-2}, calcul de F_{n-3}

calcul de F_n → calcul de F_{n-2} → calcul de F_{n-3}, calcul de F_{n-4}

Un tel arbre[4] 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.*
On comprend alors que la traduction de cet algorithme va générer de gros problèmes de mémoire.

4. Les calculs exacts seront donnés dans le prochain chapitre

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples
2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme
3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner
4. Conclusion

1 Définition et illustration
- Introduction
- Définition
- Premiers exemples

2 Une structure de données liée à la récursivité
- Notion de pile
- Récursivité et pile de programme

3 D'autres exemples
- Récursivité croisée
- Géométrie fractale
- Back tracking
- Diviser pour régner

4 Conclusion

un exercice du bac S

Pour illustrer cette notion partons d'un exercice du bac S :
On définit deux suites u et v par $u_0 = 1,\ v_0 = 12$ et pour tout entier naturel n :

$$\begin{cases} u_{n+1} = \frac{1}{3}(u_n + 2v_n) \\ v_{n+1} = \frac{1}{4}(u_n + 3v_n) \end{cases}$$

Il s'agit ici de définir deux fonctions récursives dépendant l'une de l'autre.

Fonction U(n)

début
- **si** $n < 1$ **alors**
  - **retourner** : 1
- **sinon**
  - **retourner** : 1/3*(U(n-1)+2*V(n-1))
- **fin**

fin

Fonction V(n)

début
- **si** $n < 1$ **alors**
  - **retourner** : 1
- **sinon**
  - **retourner** : 1/4*(U(n-1)+3*V(n-1))
- **fin**

fin

- L'écriture de ces algorithmes ne pose pas de difficultés, elle apparait même beaucoup plus intuitive qu'une écriture itérative.
- On peut par contre s'attendre à retrouver les mêmes problèmes d'efficacité que pour la suite de Fibonacci.
- Cela est confirmé par l'exécution de ces algorithmes.

Récursivité croisée

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples
2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme
3. **D'autres exemples**
   - Récursivité croisée
   - **Géométrie fractale**
   - Back tracking
   - Diviser pour régner
4. Conclusion

Dragon de Heighway

La courbe du dragon [5], comme toutes les fractales, est un terrain privilégié pour la programmation récursive.
En voici le principe :

A partir d'un segment, on construit deux autres segments de façon à former un triangle rectangle isocèle. A chaque fois, on alterne le sens de construction des triangles.

5. ou courbe de Heighway

Pour l'écriture de cet algorithme, on suppose avoir à notre disposition les fonctions suivantes :

- `segment(A,B)` retourne le tracé du segment [AB]
- `rotation(A,B,theta)` retourne le point image du point B par la rotation de centre A et d'angle theta
- `homothetie(A,B,k)` retourne le point image du point B par l'homothétie de centre A et de rapport k.

Notre algorithme, devra retourner, pour n donné, la courbe du dragon tracée à l'ordre n.
L'écriture de cet algorithme sous forme récursive est bien plus simple que sous sa version itérative.
La fonction récursive nécessite les paramètres suivants :

- un entier n pour l'ordre de la courbe, qui servira d'arrêt pour les appels récursifs ;
- deux points P et Q, désignant le segment sur lequel va se faire le processus de construction.

Algorithme du dragon

Fonction dragon(n,P,Q)

Entrée : n entier, P et Q deux points
Résultat : courbe du dragon d'ordre n et d'origine P et Q

```
début
    variables locales : R un point
    si n = 0 alors
        retourner : segment(P,Q)
    sinon
        R=rotation(P,Q,π/4);
        R=homothetie(P,R,1/√2);
        dragon(n-1,P,R);
        dragon(n-1,R,Q);
    fin
fin
```

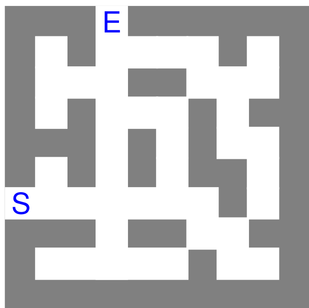
Un exemple de tracé (d’ordre 10) :

Remarque

Cet algorithme réalise le tracé, « en levant le crayon ».

Exercice : modifier l’algorithme pour obtenir un tracé continu.

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples
2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme
3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner
4. Conclusion



Pour illustrer cette notion, intéressons-nous à ce problème :
trouver le chemin de sortie du labyrinthe.

Pour cela, nous pouvons effectuer une exploration de tous les chemins possibles [6].

- la recherche doit être systématique, afin de ne pas oublier une possibilité
- lors d'une bifurcation, on doit faire un choix de chemin suivant toujours des règles précises
- en cas d'impasse, on remonte à la dernière bifurcation pour emprunter un autre chemin.

6. ces chemins sont nécessairement en nombre fini.

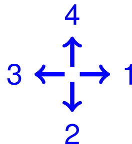
En utilisant l'ordre d'exploration ci-contre :

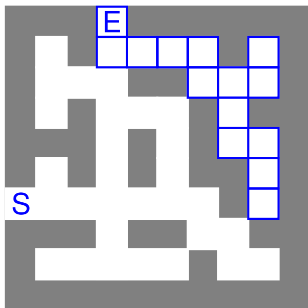
4
3 ← · → 1
2

E
S

La première exploration mène à une impasse.

En utilisant l'ordre d'exploration ci-contre :





La première exploration mène à une impasse.
On revient à la bifurcation précédente pour explorer le chemin suivant.

En utilisant l'ordre d'exploration ci-contre :

4
3 ← → 1
2

E

S

La première exploration mène à une impasse.
On revient à la bifurcation précédente pour explorer le chemin suivant.
Après plusieurs essais, on sort !

«Backtracking» se traduit littéralement par «Marche arrière».
Ce principe consiste à explorer les chemins possibles d'un problème afin de trouver une solution.
En cas d'échec, on remonte au dernier choix effectué, pour explorer un autre chemin.

Remarque

La récursivité se prête tout à fait bien à cette façon de procéder permettant un retour sur le dernier choix effectué en cas d'impasse.

Algorithme du labyrinthe

Fonction depl(fi,fj,di,dj : entiers)

Entrée : di,dj désignent la position de départ, fi,fj celle d'arrivée
Résultat : vraie si la sortie est atteinte, faux sinon
variables locales : sortie : un booléen
début

```
    Marquer la case (fi,fj) comme étant explorée;
    si (fi,fj) désigne la sortie alors
        sortie=vraie
    sinon
        sortie=faux
    fin
    si sortie=faux et le deplacement de sens 1 est valide alors
        sortie=depl(fi+1,fj,fi,fj)
    fin
    si sortie=faux et le deplacement de sens 2 est valide alors
        sortie=depl(fi,fj-1,fi,fj)
    fin
    si sortie=faux et le deplacement de sens 3 est valide alors
        sortie=depl(fi-1,fj,fi,fj)
    fin
    si sortie=faux et le deplacement de sens 4 est valide alors
        sortie=depl(fi,fj+1,fi,fj)
    fin
    si sortie=faux alors
        marquer la case (fi,fj) comme explorée mais non retenue
    fin
    retourner : sortie
```

fin

L'algorithme peut ne pas se terminer dans certaines conditions.
Lesquelles, et comment remédier de façon simple à ce problème ?

Remarque

On retiendra que :

- l'algorithme peut poser des problèmes de terminaison, en cas de cycle ;
- l'algorithme fournit une seule solution ;
- l'algorithme ne fournit pas la meilleure solution.

☞ *On peut remédier à tous ces problèmes, mais, de façon plus générale, cette théorie sera développée au cours de la troisième journée. . .*

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples

2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme

3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner

4. Conclusion

Le principe du diviser pour régner s'applique à un problème qui peut être séparé en plusieurs sous-problèmes, de même nature, mais de taille moindre, qui seront résolus de façon récursive.

Trois étapes interviennent à chaque niveau de la récursivité :

- **Diviser :** le problème en sous-problèmes semblables, de taille moindre
- **Régner :** sur les sous-problèmes en les résolvant de façon récursive
  ☞ *si la taille du problème est suffisament petite, la résolution est alors directe.*
- **Combiner :** les solutions des sous-problèmes pour produire la solution du problème initial.

Encore un tri

Ce principe précédent appliqué à un problème de tri, donne ceci :

- **Diviser :** le tableau de n éléments à trier en deux tableaux de $n/2$ éléments
- **Régner :** trier les deux tableaux obtenus de façon récursive
  ☞ *la récursivité s'arrête à un tableau contenant un seul élément, donc déjà trié.*
- **Combiner :** fusionner les deux tableaux obtenus pour obtenir la réponse triée
  ☞ *d'où le nom de tri fusion.*

La partie essentielle de l'algorithme est l'écriture de la fonction permettant de fusionner deux tableaux.
Supposons que l'on ait cette fonction, que nous appellerons `Interclassement`, à notre disposition :

Fonction Interclassement(T,n,debut,milieu,fin)

Entrée : Un tableau *T* de *n* entiers, des entiers *debut*, *milieu* et *fin*
Résultat : Le tableau *T* interclassé entre *debut* et *fin*

Notre algorithme, devient alors facile à écrire :

Tri fusion

Fonction TriFusion(T,n,debut,fin)

Entrée : Un tableau *T* de *n* entiers et des entiers *debut* et *fin*
Résultat : Le tableau *T* trié entre *debut* et *fin*

début
  variables locales : Un entier *milieu*, un tableau *temp* de *n* entiers
  ;
  si $debut < fin$ **alors**
    Donner à *milieu* la valeur $\lfloor \frac{debut+fin}{2} \rfloor$;
    `% calcul d'un entier proche du milieu du tableau` *T* `%`
    `TriFusion(`*T,n,debut,milieu*`);`
    `% tri de la première moitié du tableau` *T* `%`
    `TriFusion(`*T,n,milieu+1,fin*`);`
    `% tri de la seconde moitié du tableau` *T* `%`
    `Interclassement(`*T,n,debut,milieu,fin*`);`
    `% fusion des deux sous tableaux (triés) précédents %`
  fin
fin

Essai sur 4 ; 1 ; 5 ; 7 ; 8 ; 2 ; 3 ; 6

Les récursions découpent notre tableau jusqu'à l'obtention de tableaux unitaires :

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

4 ;1 ; 5 ; 7 | 8 ; 2 ; 3 ; 6

4 ; 1 | 5 ; 7 | 8 ; 2 | 3 ; 6

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

Essai sur 4 ; 1 ; 5 ; 7 ; 8 ; 2 ; 3 ; 6

Puis à la remontée des récursions, les tableaux sont fusionnés :

Ecriture de la fonction interclassement

Positionnement du problème :

- Cette fonction reçoit un tableau, dont les éléments d'indices allant de `debut` à `milieu` sont triés, et les éléments d'indices allant de `milieu`+1 à `fin` sont triés.
- Objectif : recombiner ces éléments pour former un tableau dont les éléments d'indice allant de `debut` à `fin` sont triés.

Ecriture de la fonction interclassement

Pour ce faire, nous utiliserons :

- deux variables *i* et *j* permettant de gérer les indices de chacune des deux parties du tableau.
  ☞ *i désigne l'indice courant de la première partie de tableau et j l'indice courant de la seconde partie de tableau.*
- un tableau auxiliaire, noté `temp`, permettant de ranger tour à tour les éléments de chacune des parties du tableau en fonction de leur ordre de grandeur ;

Ecriture de la fonction interclassement

2 affectations sont possibles :

- temp[k] reçoit $T[i]$, $T[i]$ est traité, donc on incrémente i
- temp[k] reçoit $T[j]$, $T[j]$ est traité, donc on incrémente j

Ecriture de la fonction interclassement

2 affectations sont possibles :

- temp[k] reçoit $T[i]$, $T[i]$ est traité, donc on incrémente i
  ☞ *se produit lorsqu'il n'y a plus d'éléments à traiter dans la 2nde partie du tableau (j dépasse la fin du tableau) ou lorsqu'il reste des éléments à combiner dans les deux parties de tableaux et que T[i]<T[j]*
- temp[k] reçoit $T[j]$, $T[j]$ est traité, donc on incrémente j

Ecriture de la fonction interclassement

2 affectations sont possibles :

- temp[k] reçoit $T[i]$, $T[i]$ est traité, donc on incrémente i
  ☞ *se produit lorsqu'il n'y a plus d'éléments à traiter dans la 2nde partie du tableau (j dépasse la fin du tableau) ou lorsqu'il reste des éléments à combiner dans les deux parties de tableaux et que T[i]<T[j]*
- temp[k] reçoit $T[j]$, $T[j]$ est traité, donc on incrémente j
  ☞ *situation contraire à la précédente.*

Ecriture de la fonction interclassement

Fonction Interclassement(T,n,debut,milieu,fin)

Entrée : Un tableau *T* de *n* entiers, des entiers *debut*, *milieu* et *fin*
Résultat : Le tableau *T* interclassé entre *debut* et *fin*

début
    variables locales : Des entiers *i*, *j*, *k*, un tableau *temp* de *n* entiers
    ;
    Donner à *i* la valeur *debut*;
    Donner à *j* la valeur $milieu + 1$;
    pour *k* **de** *debut* **a** *fin* **par pas de** 1 **faire**
        si $(j > fin$ **ou** $(i \leq milieu$ **et** $T[i] < T[j]))$ **alors**
            Donner à $temp[k]$ la valeur $T[i]$;
            Donner à *i* la valeur $i + 1$;
        sinon
            Donner à $temp[k]$ la valeur $T[j]$;
            Donner à *j* la valeur $j + 1$;
        fin
        ;
    fin
    ;
    % copier le tableau résultat *temp* à sa place dans le tableau *T* %
    ;
    pour *k* **de** *debut* **a** *fin* **par pas de** 1 **faire**
        Donner à $T[k]$ la valeur $temp[k]$;
    fin
fin

La récursivité permet ainsi l'écriture simple d'une solution puissante de tri. La complexité de cet algorithme sera étudiée dans le prochain chapitre.

1. Définition et illustration
   - Introduction
   - Définition
   - Premiers exemples
2. Une structure de données liée à la récursivité
   - Notion de pile
   - Récursivité et pile de programme
3. D'autres exemples
   - Récursivité croisée
   - Géométrie fractale
   - Back tracking
   - Diviser pour régner
4. Conclusion

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

FIN