

Définition

Définition

En informatique, on dit qu'une fonction est **récursive**,

Définition

En informatique, on dit qu'une fonction est **récursive**, lorsque cette fonction fait appel à elle-même.

Définition

En informatique, on dit qu'une fonction est **récursive**, lorsque cette fonction fait appel à elle-même.

Remarques

Définition

En informatique, on dit qu'une fonction est **récursive**, lorsque cette fonction fait appel à elle-même.

Remarques

- Une fonction récursive permet donc, *comme une boucle*, de répéter des instructions. Une même fonction peut donc souvent se programmer de façon **itérative** (avec des boucles) ou de façon **récursive** (en s'appelant elle-même).

Définition

En informatique, on dit qu'une fonction est **récursive**, lorsque cette fonction fait appel à elle-même.

Remarques

- Une fonction récursive permet donc, *comme une boucle*, de répéter des instructions. Une même fonction peut donc souvent se programmer de façon **itérative** (avec des boucles) ou de façon **récursive** (en s'appelant elle-même).
- Une fonction récursive doit toujours **contenir une condition d'arrêt**, dans le cas contraire elle s'appelle elle-même à l'infini et le programme ne se termine jamais.

Définition

En informatique, on dit qu'une fonction est **récursive**, lorsque cette fonction fait appel à elle-même.

Remarques

- Une fonction récursive permet donc, *comme une boucle*, de répéter des instructions. Une même fonction peut donc souvent se programmer de façon **itérative** (avec des boucles) ou de façon **récursive** (en s'appelant elle-même).
- Une fonction récursive doit toujours **contenir une condition d'arrêt**, dans le cas contraire elle s'appelle elle-même à l'infini et le programme ne se termine jamais.
- Les valeurs passées en paramètres lors des appels successifs doivent être différents, sinon la fonction s'exécute à l'identique à chaque appel et donc boucle à l'infini.

Exemple : les puissances positives

En mathématiques, pour un nombre quelconque a et un entier positif n , on définit a puissance n par :

$a^n = a \times a \times \cdots \times a$, et on convient que $a^0 = 1$

Exemple : les puissances positives

En mathématiques, pour un nombre quelconque a et un entier positif n , on définit a puissance n par :

$a^n = a \times a \times \cdots \times a$, et on convient que $a^0 = 1$

- Définir une fonction puissance qui prend en argument un flottant a et un entier n et renvoie a^n en effectuant ce calcul de façon itératif

Exemple : les puissances positives

En mathématiques, pour un nombre quelconque a et un entier positif n , on définit a puissance n par :

$a^n = a \times a \times \cdots \times a$, et on convient que $a^0 = 1$

- Définir une fonction **puissance** qui prend en argument un flottant a et un entier n et renvoie a^n en effectuant ce calcul de façon itératif
- Recopier et compléter : $a^n = \cdots \times a^{\cdots}$

Exemple : les puissances positives

En mathématiques, pour un nombre quelconque a et un entier positif n , on définit a puissance n par :

$a^n = a \times a \times \cdots \times a$, et on convient que $a^0 = 1$

- Définir une fonction puissance qui prend en argument un flottant a et un entier n et renvoie a^n en effectuant ce calcul de façon itératif
- Recopier et compléter : $a^n = \cdots \times a^{\cdots}$
- En déduire une version récursive de la fonction calculant les puissances

Exemple : les puissances positives

- Puissance : version itérative

```
1 float puissance_iteratif(float a, int n){  
2     float p = 1.0;  
3     for (int k = 1; k <= n; k = k + 1){  
4         p = p * a;}  
5     return p;}
```

Exemple : les puissances positives

- Puissance : version itérative

```
1  float puissance_iteratif(float a, int n){  
2      float p = 1.0;  
3      for (int k = 1; k <= n; k = k + 1){  
4          p = p * a;}  
5      return p;}
```

- $a^n = a \times a^{n-1}$

Exemple : les puissances positives

- Puissance : version itérative

```
1 float puissance_iteratif(float a, int n){  
2     float p = 1.0;  
3     for (int k = 1; k <= n; k = k + 1){  
4         p = p * a;  
5     return p;}
```

- $a^n = a \times a^{n-1}$

- Puissance : version récursive

```
1 float puissance_recuratif(float a, int n){  
2     if (n == 0){  
3         return 1;}  
4     else{  
5         return a * puissance_recuratif(a, n - 1);}}
```

Un petit détour par OCaml

La récursivité est très présente dans le paradigme de programmation fonctionnel que nous verrons en OCaml. On présente donc ici la version récursive des puissances en OCaml (même si nous n'avons pas encore commencé son apprentissage)

Un petit détour par OCaml

La récursivité est très présente dans le paradigme de programmation fonctionnel que nous verrons en OCaml. On présente donc ici la version récursive des puissances en OCaml (même si nous n'avons pas encore commencé son apprentissage)

```
1  let rec puissance x n =  
2    if n=0 then 1 else x * puissance x (n-1)
```


Un petit détour par OCaml

La récursivité est très présente dans le paradigme de programmation fonctionnel que nous verrons en OCaml. On présente donc ici la version récursive des puissances en OCaml (même si nous n'avons pas encore commencé son apprentissage)

```
1 let rec puissance x n =  
2   if n=0 then 1 else x * puissance x (n-1)
```

Un petit détour par OCaml

La récursivité est très présente dans le paradigme de programmation fonctionnel que nous verrons en OCaml. On présente donc ici la version récursive des puissances en OCaml (même si nous n'avons pas encore commencé son apprentissage)

```
1 let rec puissance x n =  
2   if n=0 then 1 else x * puissance x (n-1)
```

- La définition d'une fonction commence par **let**, suivi de **rec** si la fonction est récursive.

Un petit détour par OCaml

La récursivité est très présente dans le paradigme de programmation fonctionnel que nous verrons en OCaml. On présente donc ici la version récursive des puissances en OCaml (même si nous n'avons pas encore commencé son apprentissage)

```
1 let rec puissance x n =  
2   if n=0 then 1 else x * puissance x (n-1)
```

- La définition d'une fonction commence par **let**, suivi de **rec** si la fonction est récursive.
- Remarquer l'absence des parenthèses autour des arguments, et surtout l'absence du type des variables. En effet, OCaml détecte automatiquement le type des variables utilisés (ici des entiers) par un procédé appelé **inférence de types**.

Un petit détour par OCaml

La récursivité est très présente dans le paradigme de programmation fonctionnel que nous verrons en OCaml. On présente donc ici la version récursive des puissances en OCaml (même si nous n'avons pas encore commencé son apprentissage)

```
1 let rec puissance x n =  
2   if n=0 then 1 else x * puissance x (n-1)
```

Un petit détour par OCaml

La récursivité est très présente dans le paradigme de programmation fonctionnel que nous verrons en OCaml. On présente donc ici la version récursive des puissances en OCaml (même si nous n'avons pas encore commencé son apprentissage)

```
1 let rec puissance x n =  
2   if n=0 then 1 else x * puissance x (n-1)
```

- Le test d'égalité est le `simple =`

Un petit détour par OCaml

La récursivité est très présente dans le paradigme de programmation fonctionnel que nous verrons en OCaml. On présente donc ici la version récursive des puissances en OCaml (même si nous n'avons pas encore commencé son apprentissage)

```
1 let rec puissance x n =  
2   if n=0 then 1 else x * puissance x (n-1)
```

- Le test d'égalité est le **simple** =
- Le mot clé **then** suit le test d'égalité.
- Remarquer l'absence de **return**.

Exemple : factorielle

- 1 Ecrire une fonction itérative en C permettant de calculer $n!$.

Exemple : dessin d'un triangle

Exemple : factorielle

- 1 Ecrire une fonction itérative en C permettant de calculer $n!$.
- 2 Ecrire la relation liant $n!$ et $(n - 1)!$.

Exemple : dessin d'un triangle

Exemple : factorielle

- 1 Ecrire une fonction itérative en C permettant de calculer $n!$.
- 2 Ecrire la relation liant $n!$ et $(n - 1)!$.
- 3 Donner une fonction récursive en C permettant de calculer $n!$.

Exemple : dessin d'un triangle

Exemple : factorielle

- 1 Ecrire une fonction itérative en C permettant de calculer $n!$.
- 2 Ecrire la relation liant $n!$ et $(n - 1)!$.
- 3 Donner une fonction récursive en C permettant de calculer $n!$.
- 4 Même question en Ocaml.

Exemple : dessin d'un triangle

Exemple : factorielle

- 1 Ecrire une fonction itérative en C permettant de calculer $n!$.
- 2 Ecrire la relation liant $n!$ et $(n - 1)!$.
- 3 Donner une fonction récursive en C permettant de calculer $n!$.
- 4 Même question en Ocaml.

Exemple : dessin d'un triangle

On suppose qu'on dispose déjà d'une fonction `ligne` qui ne renvoie rien prend en argument un entier `n` et affiche `n` caractères `*` suivi d'un saut de ligne.

Exemple : factorielle

- 1 Ecrire une fonction itérative en C permettant de calculer $n!$.
- 2 Ecrire la relation liant $n!$ et $(n - 1)!$.
- 3 Donner une fonction récursive en C permettant de calculer $n!$.
- 4 Même question en Ocaml.

Exemple : dessin d'un triangle

On suppose qu'on dispose déjà d'une fonction `ligne` qui ne renvoie rien prend en argument un entier n et affiche n caractères `*` suivi d'un saut de ligne.

- 1 Ecrire une fonction itérative en C qui prend en argument un entier n et dessine un triangle de caractères `*` comme ci-dessous ($n = 4$) :

```
****
```

```
***
```

```
**
```

```
*
```

Exemple : factorielle

- 1 Ecrire une fonction itérative en C permettant de calculer $n!$.
- 2 Ecrire la relation liant $n!$ et $(n - 1)!$.
- 3 Donner une fonction récursive en C permettant de calculer $n!$.
- 4 Même question en Ocaml.

Exemple : dessin d'un triangle

On suppose qu'on dispose déjà d'une fonction `ligne` qui ne renvoie rien prend en argument un entier n et affiche n caractères `*` suivi d'un saut de ligne.

- 1 Ecrire une fonction itérative en C qui prend en argument un entier n et dessine un triangle de caractères `*` comme ci-dessous ($n = 4$) :

```
****
```

```
***
```

```
**
```

```
*
```

- 2 Donner une version récursive de cette fonction.

Factorielle : correction

① Version itérative

```
1  int fact_iter( int n){  
2      int res = 1;  
3      for ( int i=1;i<=n;i++){  
4          res = res * i;}  
5      return res;}
```

Factorielle : correction

1 Version itérative

```
1  int fact_iter( int n){  
2      int res = 1;  
3      for ( int i=1;i<=n;i++){  
4          res = res * i;}  
5      return res;}
```

2 $n! = n \times (n - 1)!$

Factorielle : correction

① Version itérative

```
1  int fact_iter( int n){  
2      int res = 1;  
3      for ( int i=1;i<=n;i++){  
4          res = res * i;}  
5      return res;}
```

② $n! = n \times (n - 1)!$

③ Version récursive en C

```
1  int fact_rec( int n){  
2      if (n==0)  
3          {return 1;}  
4      return n*fact_rec(n-1);}
```


Factorielle : correction

① Version itérative

```
1  int fact_iter( int n){  
2      int res = 1;  
3      for ( int i=1;i<=n;i++){  
4          res = res * i;}  
5      return res;}
```

② $n! = n \times (n - 1)!$

③ Version récursive en C

```
1  int fact_rec( int n){  
2      if (n==0)  
3          {return 1;}  
4      return n*fact_rec(n-1);}
```

④ Version récursive en Ocaml

```
1  let rec fact n =  
2      if n=0 then 1 else n*fact (n-1)
```

Triangle : correction

1 Version itérative

```
1 void triangle_iter(int n){  
2     for (int i=n;i>=1;i--)  
3         {ligne(i);}   
4 }
```

Triangle : correction

① Version itérative

```
1 void triangle_iter(int n){  
2     for (int i=n;i>=1;i--)  
3         {ligne(i);}   
4 }
```

② Version récursive

```
1 void triangle_rec(int n) {  
2     if (n>0)  
3         {ligne(n);  
4         triangle_rec(n-1);}   
5 }
```

Définition

Une fonction est dite **récursive terminale** lorsque chaque appel récursif est la dernière instruction à être évaluée. C'est à dire qu'aucun calcul n'est effectué avec le résultat de l'appel récursif.

Exemples

Définition

Une fonction est dite **réursive terminale** lorsque chaque appel récursif est la dernière instruction à être évaluée. C'est à dire qu'aucun calcul n'est effectué avec le résultat de l'appel récursif.

Exemples

La fonction **puissance** vue en début de chapitre :

```
1 float puissance_recuratif(float a, int n){  
2     if (n == 0){  
3         return 1;}  
4     else{  
5         return a * puissance_recuratif(a, n - 1);}}
```

Définition

Une fonction est dite **réursive terminale** lorsque chaque appel récursif est la dernière instruction à être évaluée. C'est à dire qu'aucun calcul n'est effectué avec le résultat de l'appel récursif.

Exemples

La fonction **puissance** vue en début de chapitre :

```
1 float puissance_recurusif(float a, int n){  
2     if (n == 0){  
3         return 1;}  
4     else{  
5         return a * puissance_recurusif(a, n - 1);}}
```

n'est **pas** réursive terminale,

Définition

Une fonction est dite **récursive terminale** lorsque chaque appel récursif est la dernière instruction à être évaluée. C'est à dire qu'aucun calcul n'est effectué avec le résultat de l'appel récursif.

Exemples

La fonction **puissance** vue en début de chapitre :

```
1 float puissance_recurusif(float a, int n){  
2     if (n == 0){  
3         return 1;}  
4     else{  
5         return a * puissance_recurusif(a, n - 1);}}
```

n'est **pas** récursive terminale, en effet bien que l'appel récursif soit placé à la fin, le dernier calcul effectué par la fonction est la multiplication.

Exemples

- ① La fonction factorielle ci-dessous est-elle récursive terminale ?

```
1  int fact_rec( int n){  
2      if (n==0)  
3          {return 1;}  
4      return n*fact_rec(n-1);}
```


Exemples

- ① La fonction factorielle ci-dessous est-elle récursive terminale ?

```
1  int fact_rec( int n){  
2      if (n==0)  
3          {return 1;}  
4      return n*fact_rec(n-1);}
```

Non, car le dernier calcul effectué par la fonction est la multiplication.

Exemples

- ① La fonction factorielle ci-dessous est-elle récursive terminale ?

```
1  int fact_rec( int n){  
2      if (n==0)  
3          {return 1;}  
4      return n*fact_rec(n-1);}
```

Non, car le dernier calcul effectué par la fonction est la multiplication.

- ② Même question pour la fonction triangle

```
1  void triangle_rec(int n) {  
2      if (n>0)  
3          {ligne(n);  
4              triangle_rec(n-1);}  
5      }
```

Exemples

- ① La fonction factorielle ci-dessous est-elle récursive terminale ?

```
1  int fact_rec( int n){  
2      if (n==0)  
3          {return 1;}  
4      return n*fact_rec(n-1);}
```

Non, car le dernier calcul effectué par la fonction est la multiplication.

- ② Même question pour la fonction triangle

```
1  void triangle_rec(int n) {  
2      if (n>0)  
3          {ligne(n);  
4              triangle_rec(n-1);}  
5      }
```

Oui, l'appel récursif est la dernière instruction à être exécutée.

Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

Par exemple pour la fonction factorielle récursive :

Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

Par exemple pour la fonction factorielle récursive :

$\text{fact}(4) = 4 * \text{fact}(3)$

Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

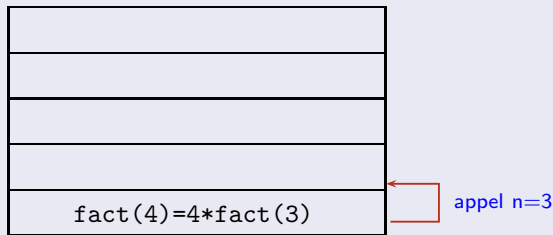
Par exemple pour la fonction factorielle récursive :

$\text{fact}(4) = 4 * \text{fact}(3)$

Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

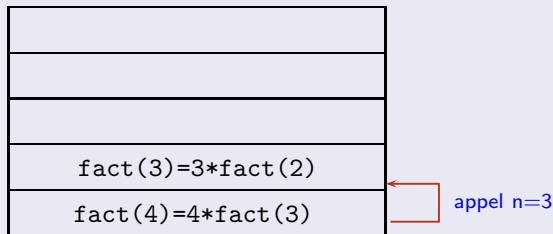
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

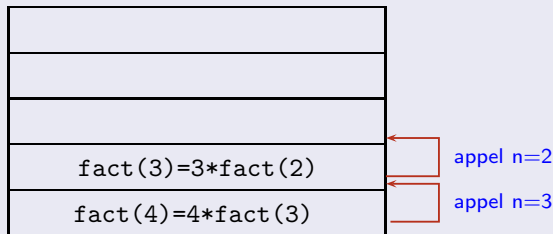
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

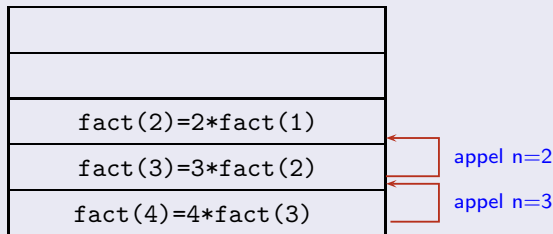
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

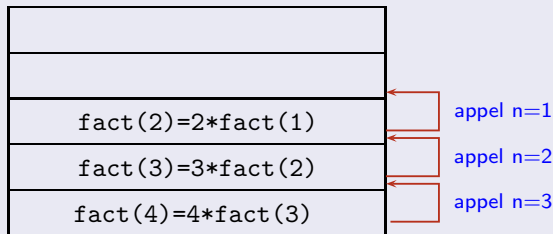
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

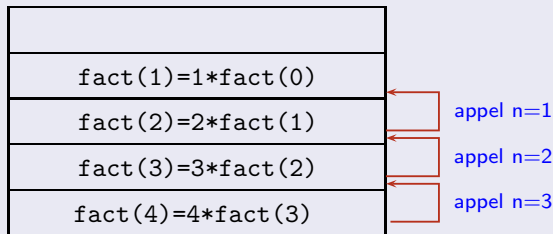
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

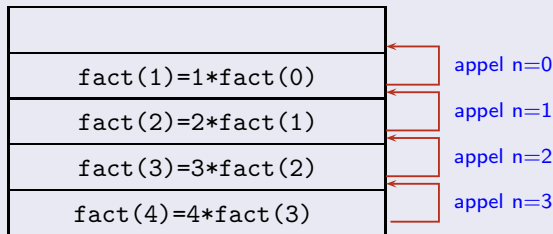
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

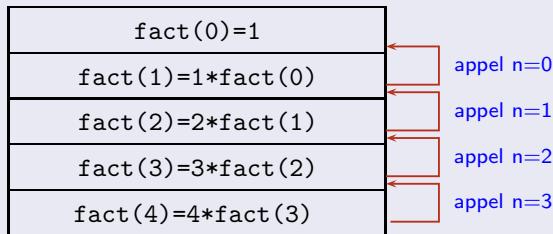
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

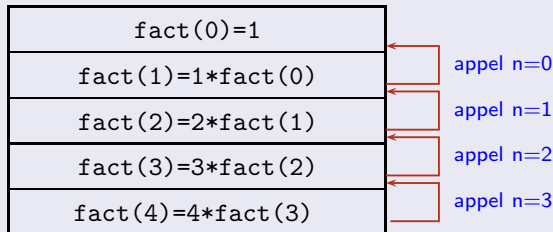
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

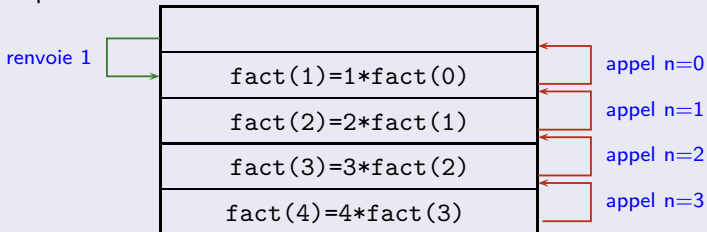
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

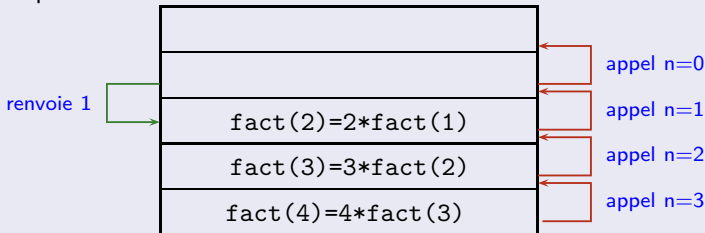
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

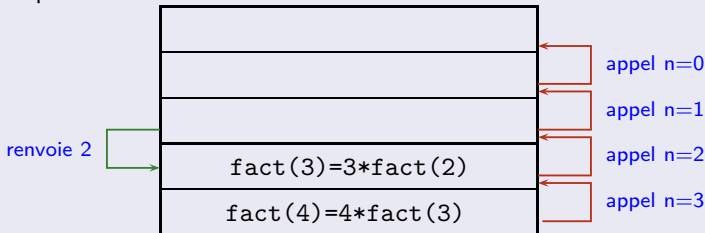
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

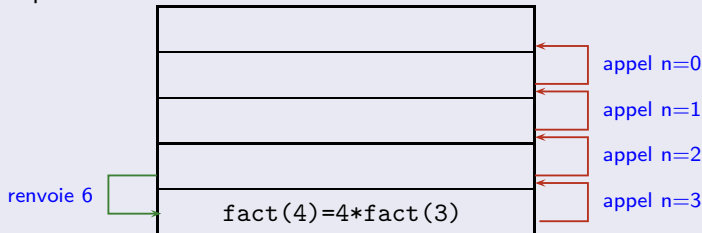
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

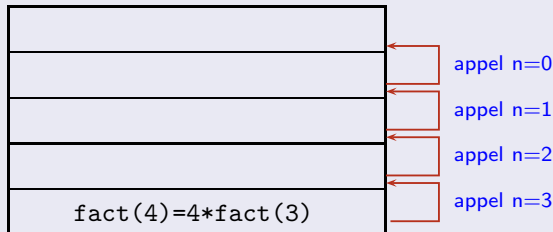
Par exemple pour la fonction factorielle récursive :



Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

Par exemple pour la fonction factorielle récursive :

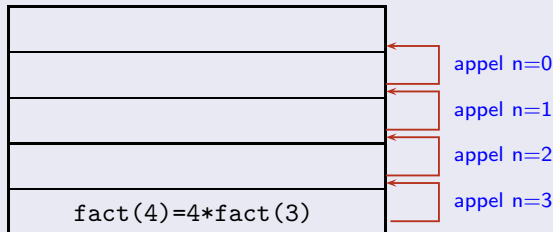


Le dépassement de capacité de la pile d'appel est le fameux *stackoverflow* (débordement de pile).

Remarque

Dans le cas d'une récursivité non terminale, les appels successifs sont empilés dans la **pile d'exécution**, en effet le résultat de chaque appel doit être renvoyé vers l'appel précédent. Puis être dépilés une fois le cas de base atteint.

Par exemple pour la fonction factorielle récursive :



Le dépassement de capacité de la pile d'appel est le fameux *stackoverflow* (débordement de pile). Pour l'éviter, on peut transformer une fonction récursive non terminale en récursive terminale ce qui évite l'empilement des appels successifs.

Exemple : factorielle récursive terminale

L'idée est d'utiliser une variable auxiliaire (un *accumulateur*) qui est passé en argument à l'appel récursif et dans lequel les calculs sont effectués au fur et à mesure des appels.

Exemple : factorielle récursive terminale

L'idée est d'utiliser une variable auxiliaire (un *accumulateur*) qui est passé en argument à l'appel récursif et dans lequel les calculs sont effectués au fur et à mesure des appels.

- En C :

```
1  int fact_acc( int n,  int acc){
2      if (n==0)
3          {return acc;}
4      return fact_acc(n-1,n*acc);}
```

- En OCaml :

```
1  let rec fact_acc  n acc =
2      if n=0 then acc else fact_acc (n-1) (n*acc)
```


Exemple : puissance

Donner une version récursive terminale de la fonction puissance :

Exemple : puissance

Donner une version récursive terminale de la fonction puissance :

- En C

Exemple : puissance

Donner une version récursive terminale de la fonction puissance :

- En C

```
1  float puissance_acc(float a, int n, float acc){  
2      if (n == 0) {  
3          return acc;}  
4      else {  
5          return puissance_acc(a, n - 1, acc * a);}}
```

Exemple : puissance

Donner une version récursive terminale de la fonction puissance :

- En C

```
1  float puissance_acc(float a, int n, float acc){  
2      if (n == 0) {  
3          return acc;}  
4      else {  
5          return puissance_acc(a, n - 1, acc * a);}}
```

- En OCaml

```
1  let rec puissance_acc x n acc =  
2      if n=0 then acc else puissance_acc x (n-1) (x*acc)
```

Remarque

- La version récursive terminale ne provoquera pas de débordement de pile en OCaml car elle sera automatiquement optimisée par le compilateur.

Remarque

- La version récursive terminale ne provoquera pas de débordement de pile en OCaml car elle sera automatiquement optimisée par le compilateur.
- En C, on utilisera l'option d'optimisation `-O2` pour que cela soit le cas.

Remarque

- La version récursive terminale ne provoquera pas de débordement de pile en OCaml car elle sera automatiquement optimisée par le compilateur.
- En C, on utilisera l'option d'optimisation `-O2` pour que cela soit le cas.
- Le langage Ocaml est plus "adapté" à une programmation fonctionnel (et donc à la récursivité) que le C qui permet la récursivité mais est avant tout suit le paradigme impératif.

Définition

Lorsqu'on définit simultanément plusieurs fonctions qui s'appellent mutuellement on parle de **récursivité croisée** (ou de récursivité mutuelle).

Définition

Lorsqu'on définit simultanément plusieurs fonctions qui s'appellent mutuellement on parle de **récursivité croisée** (ou de récursivité mutuelle).

Exemple : moyenne arithmético-géométrique

Etant donné $a \in \mathbb{R}^+, b \in \mathbb{R}^+$, on définit les suites :

$$\begin{cases} u_0 &= a \\ u_{n+1} &= \sqrt{u_n v_n} \end{cases} \quad \begin{cases} v_0 &= b \\ v_{n+1} &= \frac{u_n + v_n}{2} \end{cases}$$

Ecrire deux fonctions mutuellement récursives `un` et `vn` permettant de calculer les termes de ces deux suites.

s deux suites.

Définition

Lorsqu'on définit simultanément plusieurs fonctions qui s'appellent mutuellement on parle de **récursivité croisée** (ou de récursivité mutuelle).

Exemple : moyenne arithmético-géométrique

Etant donné $a \in \mathbb{R}^+, b \in \mathbb{R}^+$, on définit les suites :

$$\begin{cases} u_0 &= a \\ u_{n+1} &= \sqrt{u_n v_n} \end{cases} \quad \begin{cases} v_0 &= b \\ v_{n+1} &= \frac{u_n + v_n}{2} \end{cases}$$

Ecrire deux fonctions mutuellement récursives `un` et `vn` permettant de calculer les termes de ces deux suites.

⚠ En C, une fonction doit être définie avant d'y faire appel. On doit donc dans le cas d'une récursivité croisée donner la signature d'une des deux fonctions avant la définition complète de l'autre. s deux suites.

Définition

Lorsqu'on définit simultanément plusieurs fonctions qui s'appellent mutuellement on parle de **récursivité croisée** (ou de récursivité mutuelle).

Exemple : moyenne arithmético-géométrique

Etant donné $a \in \mathbb{R}^+, b \in \mathbb{R}^+$, on définit les suites :

$$\begin{cases} u_0 &= a \\ u_{n+1} &= \sqrt{u_n v_n} \end{cases} \qquad \begin{cases} v_0 &= b \\ v_{n+1} &= \frac{u_n + v_n}{2} \end{cases}$$

Ecrire deux fonctions mutuellement récursives `un` et `vn` permettant de calculer les termes de ces deux suites.

⚠ En C, une fonction doit être définie avant d'y faire appel. On doit donc dans le cas d'une récursivité croisée donner la signature d'une des deux fonctions avant la définition complète de l'autre. s deux suites.

En OCaml, on définira les deux fonctions dans le même "let" en les séparant par `and`.

Correction : En C

```
1 double vn(int n, double a, double b);  
2  
3 double un(int n, double a, double b){  
4     if (n==0)  
5     {return a;}  
6     return sqrt(un(n-1,a,b)*vn(n-1,a,b));}  
7  
8 double vn(int n,double a, double b){  
9     if (n==0)  
10    {return b;}  
11    return (un(n-1,a,b)+vn(n-1,a,b))/2;}
```

Correction : En OCaml

```
1 let rec un n a b=  
2   if n=0 then a else sqrt((un (n-1) a b) *. vn (n-1) a b)  
3   and  
4   vn n a b =  
5   if n=0 then b else ((un (n-1) a b) +. (vn (n-1) a b)) /. 2.0
```

Correction : En OCaml

```
1 let rec un n a b=  
2   if n=0 then a else sqrt((un (n-1) a b) *. vn (n-1) a b)  
3   and  
4   vn n a b =  
5   if n=0 then b else ((un (n-1) a b) +. (vn (n-1) a b)) /. 2.0
```

On remarquera `+. , *. , et /.` qui correspondent aux opérations sur les flottants (inférence de type)

Exemple introductif

- ① Ecrire une fonction récursive qui prend en argument un entier n et renvoie le n ème terme de la suite de Fibonacci défini par :

$$\begin{cases} f_0 &= 0, \\ f_1 &= 1, \\ f_n &= f_{n-1} + f_{n-2} \text{ pour tout } n \geq 2. \end{cases}$$

Exemple introductif

- 1 Ecrire une fonction récursive qui prend en argument un entier n et renvoie le n ième terme de la suite de Fibonacci défini par :
$$\begin{cases} f_0 &= 0, \\ f_1 &= 1, \\ f_n &= f_{n-1} + f_{n-2} \text{ pour tout } n \geq 2. \end{cases}$$
- 2 Tracer le graphe des appels récursifs de cette fonction pour $n = 5$

Exemple introductif

- 1 Ecrire une fonction récursive qui prend en argument un entier n et renvoie le n ième terme de la suite de Fibonacci défini par :
$$\begin{cases} f_0 &= 0, \\ f_1 &= 1, \\ f_n &= f_{n-1} + f_{n-2} \text{ pour tout } n \geq 2. \end{cases}$$
- 2 Tracer le graphe des appels récursifs de cette fonction pour $n = 5$
- 3 Commenter

Correction question 1

- En C

```
1  int fibo_rec(int n){  
2      if (n<2)  
3          {return 1;}  
4      return fibo_rec(n-1) + fibo_rec(n-2);}
```

Correction question 1

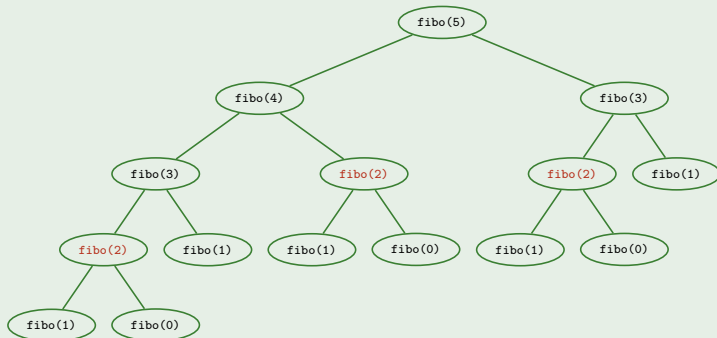
- En C

```
1  int fibo_rec(int n){  
2      if (n<2)  
3          {return 1;}  
4      return fibo_rec(n-1) + fibo_rec(n-2);}
```

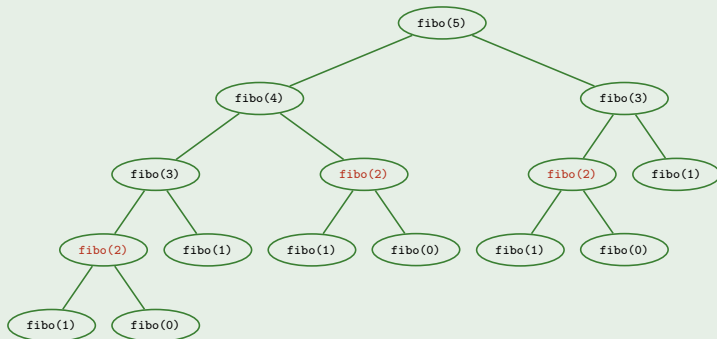
- En OCaml

```
1  let rec fibo n =  
2      if n<2 then 1 else fibo (n-1) + fibo (n-2)
```

Correction questions 2



Correction questions 3



On calcule à plusieurs reprises les *mêmes valeurs*, ici par exemple `fibo(2)` est calculé à trois reprises.

Remarque

- La solution récursive est ici simple à mettre en oeuvre (on traduit directement la définition mathématique de la suite)

Remarque

- La solution récursive est ici simple à mettre en oeuvre (on traduit directement la définition mathématique de la suite)
- L'exécution est problématique car on effectue plusieurs fois les mêmes appels récursifs.

Remarque

- La solution récursive est ici simple à mettre en oeuvre (on traduit directement la définition mathématique de la suite)
- L'exécution est problématique car on effectue plusieurs fois les mêmes appels récursifs.
- Pour pallier ce problème :

Remarque

- La solution récursive est ici simple à mettre en oeuvre (on traduit directement la définition mathématique de la suite)
- L'exécution est problématique car on effectue plusieurs fois les mêmes appels récursifs.
- Pour pallier ce problème :
 - On peut stocker les résultats des appels antérieurs dans une structure de données adaptées (c'est le principe de [mémoïsation](#))

Remarque

- La solution récursive est ici simple à mettre en oeuvre (on traduit directement la définition mathématique de la suite)
- L'exécution est problématique car on effectue plusieurs fois les mêmes appels récursifs.
- Pour pallier ce problème :
 - On peut stocker les résultats des appels antérieurs dans une structure de données adaptées (c'est le principe de **mémoïsation**)
 - On peut utiliser une formulation itérative.

Remarques

- Bien que la programmation récursive soit parfois gourmande en ressource (débordement de pile, chevauchement d'appels récursifs). Certains problèmes ont une solution récursive très lisible et rapide à programmer.

Remarques

- Bien que la programmation récursive soit parfois gourmande en ressource (débordement de pile, chevauchement d'appels récursifs). Certains problèmes ont une solution récursive très lisible et rapide à programmer.
- La formulation récursive est donc parfois « plus adaptée » à un problème et constitue une façon élégante et concise de programmer une résolution.

Remarques

- Bien que la programmation récursive soit parfois gourmande en ressource (débordement de pile, chevauchement d'appels récursifs). Certains problèmes ont une solution récursive très lisible et rapide à programmer.
- La formulation récursive est donc parfois « plus adaptée » à un problème et constitue une façon élégante et concise de programmer une résolution.
- On présente ici un exemple simple mais d'autres seront vus en TP (tours de Hanoï, dessins récursifs)

Exemple : exponentiation rapide

- ① Combien faut-il faire de multiplications pour calculer a^{13} avec la fonction

```
1  float puissance_iteratif(float a, int n){  
2      float p = 1.0;  
3      for (int k = 1; k <= n; k = k + 1){  
4          p = p * a;}  
5      return p;}  

```

Exemple : exponentiation rapide

- ① Combien faut-il faire de multiplications pour calculer a^{13} avec la fonction

```
1  float puissance_iteratif(float a, int n){  
2      float p = 1.0;  
3      for (int k = 1; k <= n; k = k + 1){  
4          p = p * a;}  
5      return p;}  

```

- ② Combien en faut-il si on procède de la façon suivante :

Exemple : exponentiation rapide

- ① Combien faut-il faire de multiplications pour calculer a^{13} avec la fonction

```
1  float puissance_iteratif(float a, int n){  
2      float p = 1.0;  
3      for (int k = 1; k <= n; k = k + 1){  
4          p = p * a;}  
5      return p;}  

```

- ② Combien en faut-il si on procède de la façon suivante :

- Calculer a^6 , l'élever au carré et le multiplier par a .

Exemple : exponentiation rapide

- ① Combien faut-il faire de multiplications pour calculer a^{13} avec la fonction

```
1  float puissance_iteratif(float a, int n){  
2      float p = 1.0;  
3      for (int k = 1; k <= n; k = k + 1){  
4          p = p * a;}  
5      return p;}  

```

- ② Combien en faut-il si on procède de la façon suivante :

- Calculer a^6 , l'élever au carré et le multiplier par a .
- Pour calculer a^6 , calculer a^3 et l'élever au carré.

Exemple : exponentiation rapide

- ① Combien faut-il faire de multiplications pour calculer a^{13} avec la fonction

```
1  float puissance_iteratif(float a, int n){  
2      float p = 1.0;  
3      for (int k = 1; k <= n; k = k + 1){  
4          p = p * a;}  
5      return p;}  

```

- ② Combien en faut-il si on procède de la façon suivante :

- Calculer a^6 , l'élever au carré et le multiplier par a .
- Pour calculer a^6 , calculer a^3 et l'élever au carré.
- Pour calculer a^3 , élever a au carré et multiplier par a .

Exemple : exponentiation rapide

- ① Combien faut-il faire de multiplications pour calculer a^{13} avec la fonction

```
1  float puissance_iteratif(float a, int n){  
2      float p = 1.0;  
3      for (int k = 1; k <= n; k = k + 1){  
4          p = p * a;}  
5      return p;}  

```

- ② Combien en faut-il si on procède de la façon suivante :

- Calculer a^6 , l'élever au carré et le multiplier par a .
- Pour calculer a^6 , calculer a^3 et l'élever au carré.
- Pour calculer a^3 , élever a au carré et multiplier par a .

- ③ Généraliser la méthode précédente au cas d'un exposant quelconque et en déduire une relation de récurrence entre a^n et $a^{\frac{n}{2}}$ si n est pair et $a^{\frac{n-1}{2}}$ sinon.

Exemple : exponentiation rapide

- ① Combien faut-il faire de multiplications pour calculer a^{13} avec la fonction

```
1 float puissance_iteratif(float a, int n){  
2     float p = 1.0;  
3     for (int k = 1; k <= n; k = k + 1){  
4         p = p * a;}  
5     return p;}  

```

- ② Combien en faut-il si on procède de la façon suivante :

- Calculer a^6 , l'élever au carré et le multiplier par a .
- Pour calculer a^6 , calculer a^3 et l'élever au carré.
- Pour calculer a^3 , élever a au carré et multiplier par a .

- ③ Généraliser la méthode précédente au cas d'un exposant quelconque et en déduire une relation de récurrence entre a^n et $a^{\frac{n-1}{2}}$ si n est pair et $a^{\frac{n-1}{2}}$ sinon.
- ④ Proposer une implémentation récursive en C puis en Ocaml.

Exemple : exponentiation rapide

- ① Combien faut-il faire de multiplications pour calculer a^{13} avec la fonction

```
1 float puissance_iteratif(float a, int n){  
2     float p = 1.0;  
3     for (int k = 1; k <= n; k = k + 1){  
4         p = p * a;}  
5     return p;}  

```

- ② Combien en faut-il si on procède de la façon suivante :

- Calculer a^6 , l'élever au carré et le multiplier par a .
- Pour calculer a^6 , calculer a^3 et l'élever au carré.
- Pour calculer a^3 , élever a au carré et multiplier par a .

- ③ Généraliser la méthode précédente au cas d'un exposant quelconque et en déduire une relation de récurrence entre a^n et $a^{\frac{n-1}{2}}$ si n est pair et $a^{\frac{n-1}{2}}$ sinon.
- ④ Proposer une implémentation récursive en C puis en Ocaml.
- ⑤ Que pensez-vous d'une implémentation itérative ?

Exemple : exponentiation rapide

① Il faut faire 13 multiplications, puisque a^{13} est calculé avec :

$$a^{13} = 1 \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a$$

Exemple : exponentiation rapide

- ❶ Il faut faire 13 multiplications, puisque a^{13} est calculé avec :
$$a^{13} = 1 \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a$$
- ❷ Dans ce cas, il ne faut que 5 multiplications en effet, on calcul a^{13} avec :
$$a^{13} = \left((a^2 \times a)^2 \right)^2 \times a$$

Exemple : exponentiation rapide

- ❶ Il faut faire 13 multiplications, puisque a^{13} est calculé avec :

$$a^{13} = 1 \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a$$

- ❷ Dans ce cas, il ne faut que 5 multiplications en effet, on calcul a^{13} avec :

$$a^{13} = \left((a^2 \times a)^2 \right)^2 \times a$$

- ❸
$$\begin{cases} a^n = \left(a^{\frac{n}{2}} \right)^2, & \text{si } n \text{ est paire} \\ a^n = \left(a^{\frac{n-1}{2}} \right)^2 \times a, & \text{sinon} \end{cases}$$

Exponentiation rapide

4 Implémentation en C :

```
1  float puissance_rapide(float a, int n){
2      if (n == 0){
3          return 1;}
4      else {
5          float pr = puissance_rapide(a, n / 2);
6          if (n % 2 == 0){
7              return pr * pr;}
8          else{
9              return pr * pr * a;}}}
```

Exponentiation rapide

4 Implémentation en C :

```
1  float puissance_rapide(float a, int n){
2      if (n == 0){
3          return 1;}
4      else {
5          float pr = puissance_rapide(a, n / 2);
6          if (n % 2 == 0){
7              return pr * pr;}
8          else{
9              return pr * pr * a;}}}
```

5 Implémentation en OCaml :

```
1  let rec puissance_rapide x n =
2      if n=0 then 1 else
3          if n mod 2 = 0 then
4              (puissance_rapide x (n/2) * puissance_rapide x (n/2))
5          else x * (puissance_rapide x (n/2) * puissance_rapide x (n/2))
```