Langage et algorithmique

Rodéric Moitié

ENSTA Bretagne

Sommaire

Méthode fact

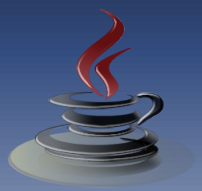
- Paramètres formels

```
public int fact(int i) {
      if(i == 0) {
      ...
}
public void cnp(int n, int p) {
      int num = fact(n);
      ...
}
```

Méthode fact

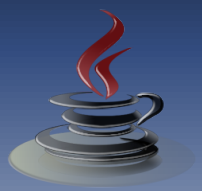
- Paramètres formels
- Paramètres effectifs

```
public int fact(int i) {
      if(i == 0) {
      ...
}
public void cnp(int n, int p) {
      int num = fact(n);
      ...
}
```



Sommaire

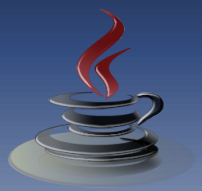
Sommaire



Définition

Définition (Fonction récursive)

Une fonction ou une méthode est dite récursive si elle se définit à partir d'elle même, c'est-à-dire si elle comporte un appel à elle même dans son corps.



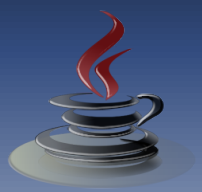
Définition

Définition (Fonction récursive)

Une fonction ou une méthode est dite récursive si elle se définit à partir d'elle même, c'est-à-dire si elle comporte un appel à elle même dans son corps.

Attention

Éviter les boucles infinies (appel systématique à la méthode).



Définition

Définition (Fonction récursive)

Une fonction ou une méthode est dite récursive si elle se définit à partir d'elle même, c'est-à-dire si elle comporte un appel à elle même dans son corps.

Attention

Éviter les boucles infinies (appel systématique à la méthode).

Exemple (Méthode infinie)

```
public int sansFin(int n) {
     return n+sansFin(n-1);
}
```

Exemple

Traduction immédiate des fonctions définies par récurrence.

Exemple (Factorielle)

$$\begin{cases} 0! = 1 \\ n! = n(n-1)! \end{cases}$$

Exemple

Traduction immédiate des fonctions définies par récurrence.

Exemple (Factorielle)

$$\begin{cases} 0! = 1 \\ n! = n(n-1)! \end{cases}$$

```
public int fact(int n) {
```

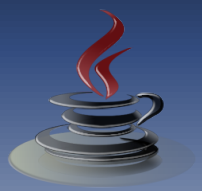
Exemple

Traduction immédiate des fonctions définies par récurrence.

Exemple (Factorielle)

$$\begin{cases} 0! = 1 \\ n! = n(n-1)! \end{cases}$$

```
public int fact(int n) {
     if (n==0)
          return 1;
```



Exemple

Traduction immédiate des fonctions définies par récurrence.

Exemple (Factorielle)

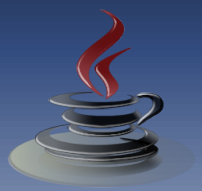
$$\begin{cases} 0! = 1 \\ n! = n(n-1)! \end{cases}$$

```
public int fact(int n) {
     if (n==0)
          return 1;
     else
          return n*fact(n-1);
}
```

Condition d'arrêt

Définition (Condition d'arrêt)

Condition pour laquelle il n'y a pas d'appel récursif.



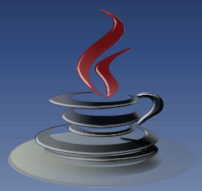
Condition d'arrêt

Définition (Condition d'arrêt)

Condition pour laquelle il n'y a pas d'appel récursif.

Important

Toute méthode récursive doit comporter au moins une condition d'arrêt. Sinon : boucle infinie.



Condition d'arrêt

Définition (Condition d'arrêt)

Condition pour laquelle il n'y a pas d'appel récursif.

Important

Toute méthode récursive doit comporter au moins une condition d'arrêt. Sinon : boucle infinie.

Remarque

Une méthode peut comporter plusieurs conditions d'arrêt.

Condition d'arrêt

Exemple (Plusieurs conditions d'arrêt)

```
public int fact(int n) {
     if (n==0)
          return 1;
     else if (n==1)
          return 1;
     else
          return n*fact(n-1);
}
```

Condition d'arrêt

Exemple (Plusieurs conditions d'arrêt)

```
public int fact(int n) {
     if (n==0)
          return 1;
     else if (n==1)
          return 1;
     else
          return n*fact(n-1);
}
```

Sommaire

Pile d'appel

- Zone de mémoire

Pile d'appel

- Zone de mémoire
- Contient :

Pile d'appel

- Zone de mémoire
- Contient :
    - paramètres

Pile d'appel

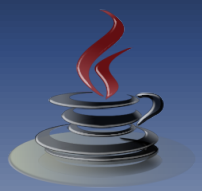
- Zone de mémoire
- Contient :
  - paramètres
  - adresse de retour

Pile d'appel

- Zone de mémoire
- Contient :
  - paramètres
  - adresse de retour
  - variables locales

Pile d'appel

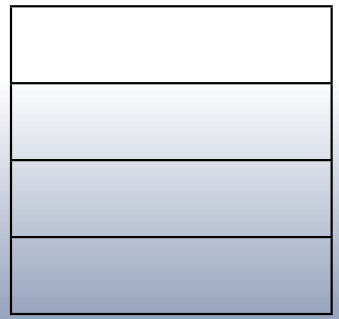
- Zone de mémoire
- Contient :
  - paramètres
  - adresse de retour
  - variables locales
- Fonctionnement lors de l'appel d'une méthode

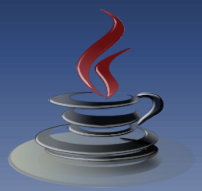


Pile d'appel

- Zone de mémoire
- Contient :
  - paramètres
  - adresse de retour
  - variables locales
- Fonctionnement lors de l'appel d'une méthode

```
public void methode(int a, double b) {
     int i;
}
methode(2, 1.5);
```

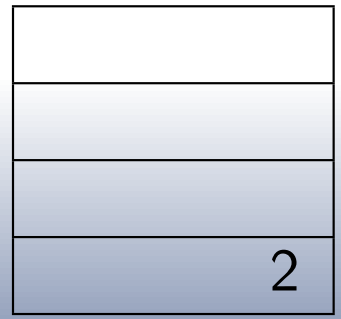


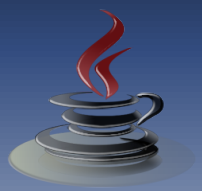


Pile d'appel

- Zone de mémoire
- Contient :
  - paramètres
  - adresse de retour
  - variables locales
- Fonctionnement lors de l'appel d'une méthode

```
public void methode(int a, double b) {
      int i;
}
methode(2, 1.5);
```





Pile d'appel

- Zone de mémoire
- Contient :
  - paramètres
  - adresse de retour
  - variables locales
- Fonctionnement lors de l'appel d'une méthode

```
public void methode(int a, double b) {
      int i;
}
methode(2, 1.5);
```

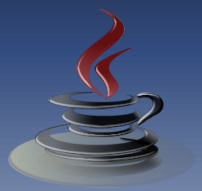
1.5
2

Pile d'appel

- Zone de mémoire
- Contient :
  - paramètres
  - adresse de retour
  - variables locales
- Fonctionnement lors de l'appel d'une méthode

```
public void methode(int a, double b) {
     int i;
}
methode(2, 1.5);
```

0x1E3C85
1.5
2



Pile d'appel

- Zone de mémoire
- Contient :
  - paramètres
  - adresse de retour
  - variables locales
- Fonctionnement lors de l'appel d'une méthode

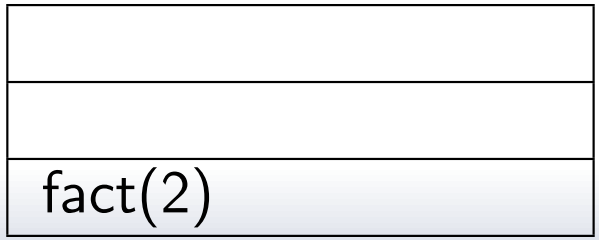
```
public void methode(int a, double b) {
      int i;
}
methode(2, 1.5);
```

i
0x1E3C85
1.5
2

Pile et récursivité

- Pile d'appel : notion fondamentale pour la récursivité.
- Principe : tous les appels sont empilés, traités, puis dépilés.

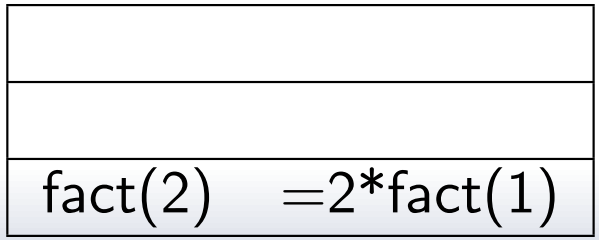
```
public int fact(int n) {
     if (n==0)
          return 1;
     else
          return n*fact(n-1);
}
```



Pile et récursivité

- Pile d'appel : notion fondamentale pour la récursivité.
- Principe : tous les appels sont empilés, traités, puis dépilés.

```
public int fact(int n) {
     if (n==0)
          return 1;
     else
          return n*fact(n-1);
}
```



Pile et récursivité

- Pile d'appel : notion fondamentale pour la récursivité.
- Principe : tous les appels sont empilés, traités, puis dépilés.

```
public int fact(int n) {
      if (n==0)
            return 1;
      else
            return n*fact(n-1);
}
```

fact(1)
fact(2) =2*fact(1)

Pile et récursivité

- Pile d'appel : notion fondamentale pour la récursivité.
- Principe : tous les appels sont empilés, traités, puis dépilés.

```
public int fact(int n) {
     if (n==0)
          return 1;
     else
          return n*fact(n-1);
}
```

fact(1)	=1*fact(0)
fact(2)	=2*fact(1)

Pile et récursivité

- Pile d'appel : notion fondamentale pour la récursivité.
- Principe : tous les appels sont empilés, traités, puis dépilés.

```
public int fact(int n) {
      if (n==0)
            return 1;
      else
            return n*fact(n-1);
}
```

fact(0)	
fact(1)	=1*fact(0)
fact(2)	=2*fact(1)

Pile et récursivité

- Pile d'appel : notion fondamentale pour la récursivité.
- Principe : tous les appels sont empilés, traités, puis dépilés.

```
public int fact(int n) {
     if (n==0)
          return 1;
     else
          return n*fact(n-1);
}
```

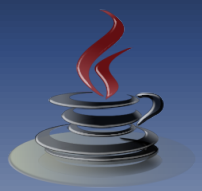
fact(0)	=1
fact(1)	=1*fact(0)
fact(2)	=2*fact(1)

Pile et récursivité

- Pile d'appel : notion fondamentale pour la récursivité.
- Principe : tous les appels sont empilés, traités, puis dépilés.

```
public int fact(int n) {
     if (n==0)
          return 1;
     else
          return n*fact(n-1);
}
```

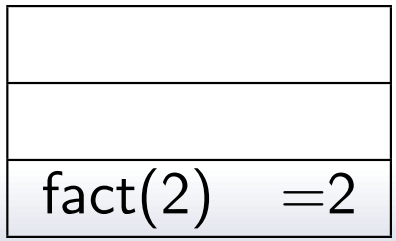
fact(1) =1
fact(2) =2*fact(1)



Pile et récursivité

- Pile d'appel : notion fondamentale pour la récursivité.
- Principe : tous les appels sont empilés, traités, puis dépilés.

```
public int fact(int n) {
     if (n==0)
          return 1;
     else
          return n*fact(n-1);
}
```



Sommaire

Suite de Fibonacci

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n \geqslant 2, F_n = F_{n-1} + F_{n-2} \end{cases}$$

Suite de Fibonacci

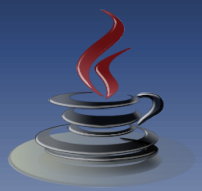
$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n \geqslant 2, F_n = F_{n-1} + F_{n-2} \end{cases}$$

- Deux conditions d'arrêt

Suite de Fibonacci

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n \geqslant 2, F_n = F_{n-1} + F_{n-2} \end{cases}$$

- Deux conditions d'arrêt
- Une condition de récurrence



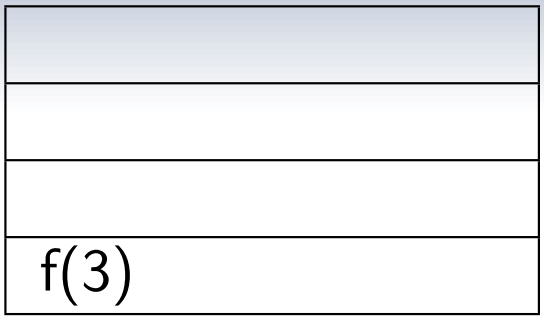
Suite de Fibonacci

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n \geqslant 2, F_n = F_{n-1} + F_{n-2} \end{cases}$$

```
public int fibo(int n) {
     if (n==0)
          return 0;
     else if (n==1)
          return 1;
     else
          return fibo(n-1) + fibo(n-2);
}
```

Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :



Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

f(3) = f(2)+f(1)

Explosion de la pile d'appel

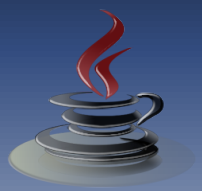
Évolution de la pile lors du calcul de F(3) :

f(2)
f(3) = f(2)+f(1)

Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

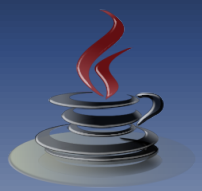
f(2) = f(1)+f(0)
f(3) = f(2)+f(1)



Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

f(1)
f(2) = f(1)+f(0)
f(3) = f(2)+f(1)



Explosion de la pile d'appel

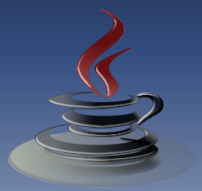
Évolution de la pile lors du calcul de F(3) :

f(1) = 1
f(2) = f(1)+f(0)
f(3) = f(2)+f(1)

Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

f(0)	
f(1)	$= 1$
f(2)	$=$ f(1)+f(0)
f(3)	$=$ f(2)+f(1)



Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

f(0)	= 0
f(1)	= 1
f(2)	= f(1)+f(0)
f(3)	= f(2)+f(1)

Explosion de la pile d'appel

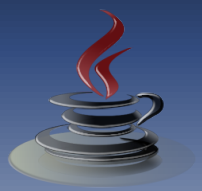
Évolution de la pile lors du calcul de F(3) :

f(2) $= 1$
f(3) $=$ f(2)+f(1)

Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

f(1)
f(2) = 1
f(3) = f(2)+f(1)



Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

f(1) = 1
f(2) = 1
f(3) = f(2)+f(1)

Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

f(3) = 2

Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

f(3) = 2

- Traduction immédiate de la formule de récurrence

Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

f(3) = 2

- Traduction immédiate de la formule de récurrence

$\Rightarrow$ peu efficace dans ce cas

Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

f(3) = 2

- Traduction immédiate de la formule de récurrence

$\Rightarrow$ peu efficace dans ce cas

- Nombre d'appel à F en $\Theta(2^n)$

Explosion de la pile d'appel

Évolution de la pile lors du calcul de F(3) :

f(3) = 2

- Traduction immédiate de la formule de récurrence

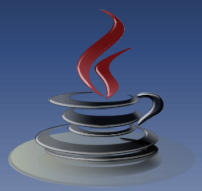
$\Rightarrow$ peu efficace dans ce cas

- Nombre d'appel à F en $\Theta(2^n)$
- Risque de **java.lang.StackOverflowError**

Sommaire

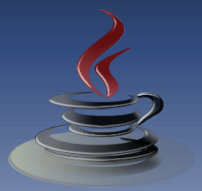
Conception d'algorithme récursif

- Découper l'algorithme en étapes



Conception d'algorithme récursif

- Découper l'algorithme en étapes
- Déterminer les règles de passage d'une étape à l'autre



Conception d'algorithme récursif

- Découper l'algorithme en étapes
- Déterminer les règles de passage d'une étape à l'autre

$\Rightarrow$ l'étape n dépend de l'étape $n-1$ (éventuellement $n-2$)

Conception d'algorithme récursif

- Découper l'algorithme en étapes
- Déterminer les règles de passage d'une étape à l'autre

$\Rightarrow$ l'étape n dépend de l'étape $n-1$ (éventuellement $n-2$)

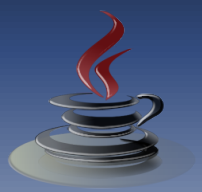
- Déterminer les conditions d'arrêt

Conception d'algorithme récursif

- Découper l'algorithme en étapes
- Déterminer les règles de passage d'une étape à l'autre

$\Rightarrow$ l'étape n dépend de l'étape $n-1$ (éventuellement $n-2$)

- Déterminer les conditions d'arrêt



Conception d'algorithme récursif

- Découper l'algorithme en étapes
- Déterminer les règles de passage d'une étape à l'autre

$\Rightarrow$ l'étape n dépend de l'étape $n-1$ (éventuellement $n-2$)

- Déterminer les conditions d'arrêt

Remarque

Tout algorithme récursif peut s'écrire de manière itérative, et réciproquement.

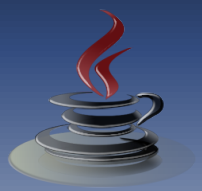
Sommaire

Recherche par dichotomie

- Recherche d'un élément dans un tableau trié

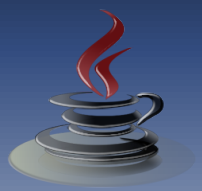
Recherche par dichotomie

- Recherche d'un élément dans un tableau trié
- Principe :



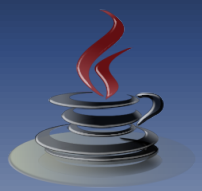
Recherche par dichotomie

- Recherche d'un élément dans un tableau trié
- Principe :
  - comparer l'élément recherché avec le milieu du tableau



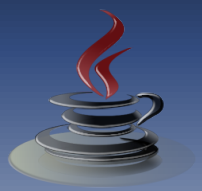
Recherche par dichotomie

- Recherche d'un élément dans un tableau trié
- Principe :
  - comparer l'élément recherché avec le milieu du tableau
  - si $>$: recherche dans le tableau $\Leftrightarrow$ recherche dans la partie droite



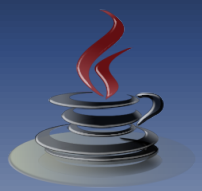
Recherche par dichotomie

- Recherche d'un élément dans un tableau trié
- Principe :
  - comparer l'élément recherché avec le milieu du tableau
  - si $>$: recherche dans le tableau $\Leftrightarrow$ recherche dans la partie droite
  - si $<$: recherche dans le tableau $\Leftrightarrow$ recherche dans la partie gauche



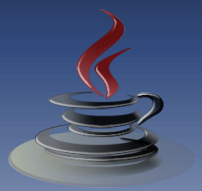
Recherche par dichotomie

- Recherche d'un élément dans un tableau trié
- Principe :
  - comparer l'élément recherché avec le milieu du tableau
  - si $>$: recherche dans le tableau $\Leftrightarrow$ recherche dans la partie droite
  - si $<$: recherche dans le tableau $\Leftrightarrow$ recherche dans la partie gauche
  - si $=$: élément trouvé (condition d'arrêt)



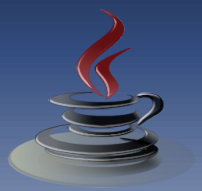
Recherche par dichotomie

- Recherche d'un élément dans un tableau trié
- Principe :
  - comparer l'élément recherché avec le milieu du tableau
  - si $>$: recherche dans le tableau $\Leftrightarrow$ recherche dans la partie droite
  - si $<$: recherche dans le tableau $\Leftrightarrow$ recherche dans la partie gauche
  - si $=$: élément trouvé (condition d'arrêt)
  - autre condition d'arrêt : taille du tableau $\leqslant 1$



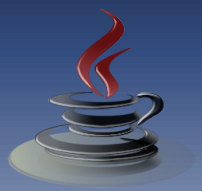
Recherche par dichotomie : code Java

```
public boolean rech(int tab[], int val, int deb, int fin)
{
      int n = (deb + fin)/2;
```



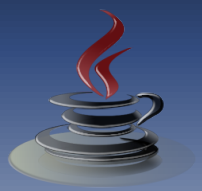
Recherche par dichotomie : code Java

```
public boolean rech(int tab[], int val, int deb, int fin)
{
      int n = (deb + fin)/2;
      if (tab[n] == val)
            return true;
```



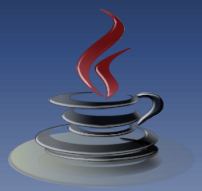
Recherche par dichotomie : code Java

```
public boolean rech(int tab[], int val, int deb, int fin)
{
      int n = (deb + fin)/2;
      if (tab[n] == val)
            return true;
      else if(deb >= fin)
            return false;
```



Recherche par dichotomie : code Java

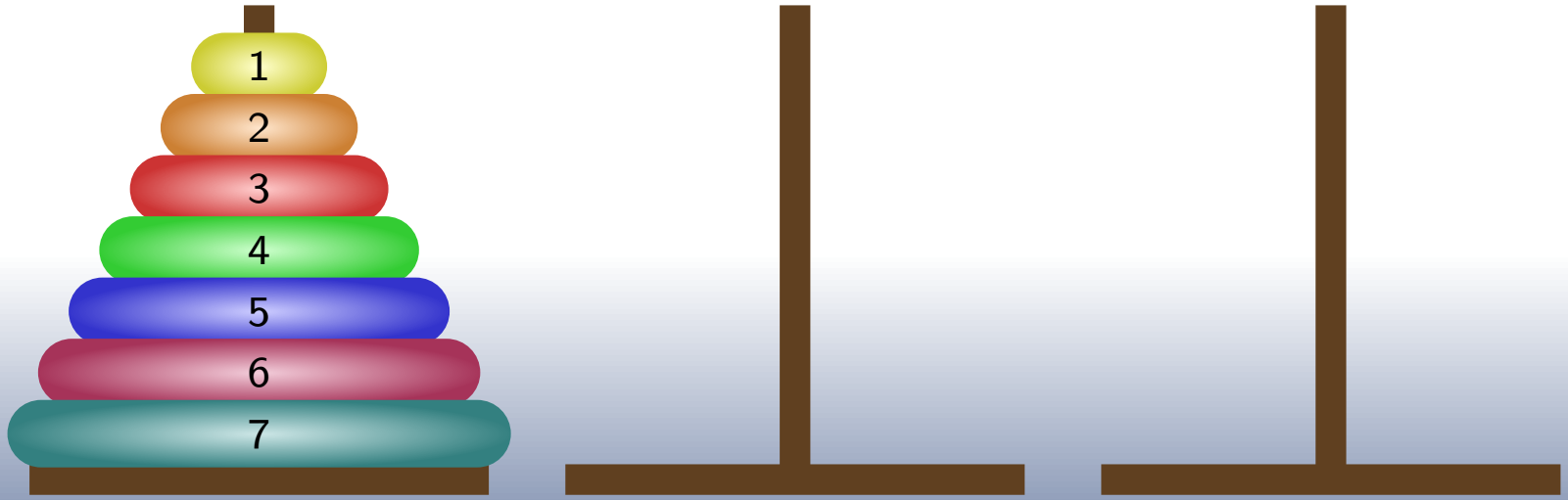
```
public boolean rech(int tab[], int val, int deb, int fin)
{
      int n = (deb + fin)/2;
      if (tab[n] == val)
            return true;
      else if(deb >= fin)
            return false;
      else if (tab[n] > val)
            return rech(tab, val, deb, n-1);
      else
            return rech(tab, val, n+1, fin);
}
```

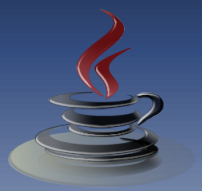


Tours de Hanoï

Principe :

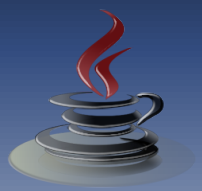
- Disques empilés : tour
- Ne jamais empiler un disque sur un disque plus petit
- Déplacer un seul disque à la fois





Tours de Hanoï : résolution

- Résolution par une méthode récursive
- Étapes de l'algorithme : hauteur de la tour
- Hauteur 0 : évident
- Récurrence :
  - on suppose savoir déplacer une tour de hauteur $h-1$
  - on veut déplacer une tour de hauteur h de A vers B
  - déplacer les $h-1$ premiers éléments de A vers C
  - déplacer l'élément restant de A vers B
  - déplacer les $h-1$ premiers éléments de C vers B



Tours de Hanoï : algorithme

Procédure hanoi(entier n, entier source, entier dest, entier tmp)

```
/* n : hauteur de la tour                                          */
/* source, dest, tmp : position d'origine, finale et
   intermédiaire de la tour à déplacer                             */
```

si $n > 0$ **alors**

```
    hanoi (n-1, source, tmp, dest) ;
    déplacer(source, dest) ;
    hanoi (n-1, tmp, dest, source) ;
```

fin

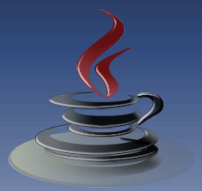
Étude de l'algorithme tours de Hanoï

$C(n)$: nombre d'opération pour déplacer une tour de hauteur n

Étude de l'algorithme tours de Hanoï

$C(n)$: nombre d'opération pour déplacer une tour de hauteur n

$$\begin{cases} C(n+1) = 2C(n) + 1 \\ C(0) = 0 \end{cases}$$

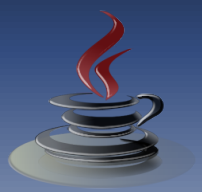


Étude de l'algorithme tours de Hanoï

$C(n)$: nombre d'opération pour déplacer une tour de hauteur n

$$\begin{cases} C(n+1) = 2C(n) + 1 \\ C(0) = 0 \end{cases}$$

Résolution : $C(n) = 2^n - 1$



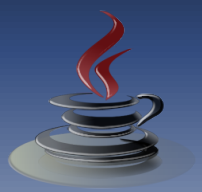
Étude de l'algorithme tours de Hanoï

$C(n)$: nombre d'opération pour déplacer une tour de hauteur n

$$\begin{cases} C(n+1) = 2C(n) + 1 \\ C(0) = 0 \end{cases}$$

Résolution : $C(n) = 2^n - 1$

Légende des tours de Hanoï : dans un temple Bouddhiste, des moines ont reçu pour mission de déplacer une tour de Hanoï de 64 disques. Lorsqu'ils l'auront déplacée, le monde tombera en poussière.



Étude de l'algorithme tours de Hanoï

$C(n)$: nombre d'opération pour déplacer une tour de hauteur n

$$\begin{cases} C(n+1) = 2C(n) + 1 \\ C(0) = 0 \end{cases}$$

Résolution : $C(n) = 2^n - 1$

Légende des tours de Hanoï : dans un temple Bouddhiste, des moines ont reçu pour mission de déplacer une tour de Hanoï de 64 disques. Lorsqu'ils l'auront déplacée, le monde tombera en poussière.
$2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$
Un déplacement par seconde $\rightsquigarrow$ 580 milliards d'années

Monnaie

Problème :

- Combien de manières de rendre la monnaie sur une somme s avec 1, 2 ou 5 € ?

Monnaie

Problème :

- Combien de manières de rendre la monnaie sur une somme s avec 1, 2 ou 5 € ?
- Résolution : méthodes récursives.

Monnaie

Problème :

- Combien de manières de rendre la monnaie sur une somme s avec 1, 2 ou 5 € ?
- Résolution : méthodes récursives.

Monnaie

Problème :

- Combien de manières de rendre la monnaie sur une somme s avec 1, 2 ou 5 € ?
- Résolution : méthodes récursives.

Algorithme :

- Rendre la monnaie sur s avec des pièces de 1 € : 1 seule possibilité

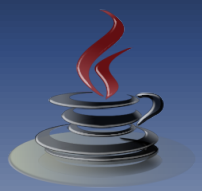
Monnaie

Problème :

- Combien de manières de rendre la monnaie sur une somme s avec 1, 2 ou 5 € ?
- Résolution : méthodes récursives.

Algorithme :

- Rendre la monnaie sur s avec des pièces de 1 € : 1 seule possibilité
- Rendre la monnaie sur s avec 1 ou 2 € :



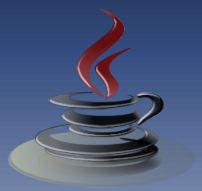
Monnaie

Problème :

- Combien de manières de rendre la monnaie sur une somme s avec 1, 2 ou 5 € ?
- Résolution : méthodes récursives.

Algorithme :

- Rendre la monnaie sur s avec des pièces de 1 € : 1 seule possibilité
- Rendre la monnaie sur s avec 1 ou 2 € :
  - Donner une pièce de 2€ et rendre la monnaie sur $s - 2$ avec 1 ou 2 €



Monnaie

Problème :

- Combien de manières de rendre la monnaie sur une somme s avec 1, 2 ou 5 € ?
- Résolution : méthodes récursives.

Algorithme :

- Rendre la monnaie sur s avec des pièces de 1 € : 1 seule possibilité
- Rendre la monnaie sur s avec 1 ou 2 € :
  - Donner une pièce de 2€ et rendre la monnaie sur $s - 2$ avec 1 ou 2 €
  - Ou rendre la monnaie uniquement avec 1€

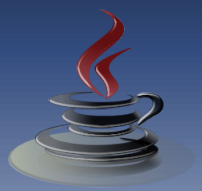
Monnaie

Problème :

- Combien de manières de rendre la monnaie sur une somme s avec 1, 2 ou 5 € ?
- Résolution : méthodes récursives.

Algorithme :

- Rendre la monnaie sur s avec des pièces de 1 € : 1 seule possibilité
- Rendre la monnaie sur s avec 1 ou 2 € :
  - Donner une pièce de 2€ et rendre la monnaie sur $s - 2$ avec 1 ou 2 €
  - Ou rendre la monnaie uniquement avec 1€
- Rendre la monnaie sur s avec 1, 2 ou 5 € : même principe



Algorithme

Procédure monnaie1(somme s)

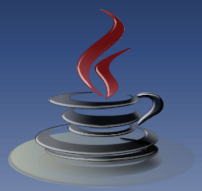
retourner 1;

Procédure monnaie1-2(somme s)

retourner **monnaie1-2** (s-2) + **monnaie1** (s);

Procédure monnaie1-2-5(somme s)

retourner **monnaie1-2-5** (s-5) + **monnaie1-2** (s) + **monnaie1** (s);



Algorithme : condition d'arrêt

- Conditions d'arrêt de l'algorithme ?
- **monnaie1** non récursive : pas de condition d'arrêt
- Pour **monnaie1-2** : s'arrêter si $s < 2$
- Pour **monnaie1-2-5** : s'arrêter si $s < 5$

Quicksort

Procédure triSeg(tableau_entier tab, entier debut, entier fin)

si *debut* $<$ *fin* **alors**

Quicksort

Procédure triSeg(tableau_entier tab, entier debut, entier fin)

si *debut* $<$ *fin* **alors**

choisir élément pivot p entre debut et fin ;

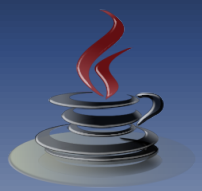
Quicksort

Procédure triSeg(tableau_entier tab, entier debut, entier fin)

si *debut* $<$ *fin* **alors**

choisir élément pivot p entre debut et fin ;

placer pivot en i, $\leqslant$ pivot avant, $\geqslant$ pivot après ;



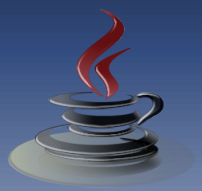
Quicksort

Procédure triSeg(tableau_entier tab, entier debut, entier fin)

si *debut* $<$ *fin* **alors**

choisir élément pivot p entre debut et fin ;
placer pivot en i, $\leqslant$ pivot avant, $\geqslant$ pivot après ;
`triSeg` (tab, deb, i-1) ;
`triSeg` (tab, i+1, fin) ;

fin



Quicksort

Procédure triSeg(tableau_entier tab, entier debut, entier fin)

```
si debut < fin alors
    si fin − debut ⩽ 1 // cas particulier : 2 éléments à trier
    alors
        si tab[debut] > tab[fin] alors
            inverser(tab, debut, fin) ;
        fin
    sinon
        choisir élément pivot p entre debut et fin ;
        placer pivot en i, ⩽ pivot avant, ⩾ pivot après ;
        triSeg (tab, deb, i-1) ;
        triSeg (tab, i+1, fin) ;
    fin
fin
```

Sommaire

Récursivité terminale

- A chaque appel récursif : empilement de données

Récursivité terminale

- A chaque appel récursif : empilement de données

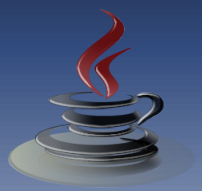
$\Rightarrow$ limite au nombre d'appels possibles

Récursivité terminale

- A chaque appel récursif : empilement de données

$\Rightarrow$ limite au nombre d'appels possibles

- Solution : récursivité terminale



Récursivité terminale

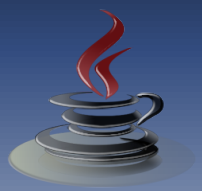
- A chaque appel récursif : empilement de données

$\Rightarrow$ limite au nombre d'appels possibles

- Solution : récursivité terminale

Définition (Récursivité terminale, tail recursion)

Une méthode est dite récursive terminale s'il n'y a aucune opération sur ses appels récursifs.



Récursivité terminale

- A chaque appel récursif : empilement de données

$\Rightarrow$ limite au nombre d'appels possibles

- Solution : récursivité terminale

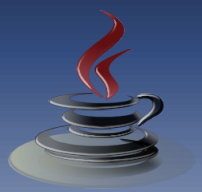
Définition (Récursivité terminale, tail recursion)

Une méthode est dite récursive terminale s'il n'y a aucune opération sur ses appels récursifs.

Exemple (Méthode récursive non terminale)

Factorielle :

```
return n*fact(n-1);
```



Écriture de méthode récursive terminale

- Idée générale : utiliser un accumulateur
- Paramètre qui contient le résultat intermédiaire

Exemple (Factorielle récursive terminale)

```
public int fact(int n, int acc) {
     if (n==0)
          return acc;
     else
          return fact(n-1, n*acc);
}
```

Fonctionnement :

Appel :

```
fact(5, 1);
```

Si le langage sait optimiser la récursivité terminale, pas d'empilement d'appels. Modification des valeurs dans la pile.

```
fact(4, 1);
```

1
4
0x1E3C85

Fonctionnement :

Appel :

```
fact(5, 1);
```

Si le langage sait optimiser la récursivité terminale, pas d'empilement d'appels. Modification des valeurs dans la pile.

```
fact(3, 4);
```

4
3
0x1E3C85

Fonctionnement :

Appel :

```
fact(5, 1);
```

Si le langage sait optimiser la récursivité terminale, pas d'empilement d'appels. Modification des valeurs dans la pile.

```
fact(2, 12);
```

12
2
0x1E3C85

Fonctionnement :

Appel :

```
fact(5, 1);
```

Si le langage sait optimiser la récursivité terminale, pas d'empilement d'appels. Modification des valeurs dans la pile.

```
fact(1, 24);
```

24
1
0x1E3C85

Fonctionnement :

Appel :

```
fact(5, 1);
```

Si le langage sait optimiser la récursivité terminale, pas d'empilement d'appels. Modification des valeurs dans la pile.

```
fact(0, 24);
```

24
0
0x1E3C85

Fonctionnement :

Appel :

```
fact(5, 1);
```

Si le langage sait optimiser la récursivité terminale, pas d'empilement d'appels. Modification des valeurs dans la pile.

```
fact(0, 24);
```

24
0
0x1E3C85

Remarque

Java n'optimise pas la récursivité terminale

Sommaire

Eclipse

- Disponible sur `http://www.eclipse.org`

Eclipse

- Disponible sur `http://www.eclipse.org`
- Principe d'un IDE

Eclipse

- Disponible sur `http://www.eclipse.org`
- Principe d'un IDE
- Avantages / inconvénients

Eclipse

- Disponible sur `http://www.eclipse.org`
- Principe d'un IDE
- Avantages / inconvénients
  - Debugger

Eclipse

- Disponible sur `http://www.eclipse.org`
- Principe d'un IDE
- Avantages / inconvénients
  - Debugger
  - Analyse syntaxique à la volée

Eclipse

- Disponible sur `http://www.eclipse.org`
- Principe d'un IDE
- Avantages / inconvénients
  - Debugger
  - Analyse syntaxique à la volée
  - Lourdeur

Eclipse

- Disponible sur `http://www.eclipse.org`
- Principe d'un IDE
- Avantages / inconvénients
  - Debugger
  - Analyse syntaxique à la volée
  - Lourdeur
- Installation / utilisation : voir didacticiels sur **moodle**