

Contrôle continu

Le contrôle continu aura lieu le

SAMEDI 19 MARS 2011 de 14h30 à 16h30

- 1/ Rappels et compléments Java.
- 2/ Tableaux, boucles et invariants.
- 3/ Notions élémentaires de complexité.
- 4/ Récursion. **ICI**
- 5/ Structures de données et introduction aux types abstraits de données.
- 6/ Quelques compléments Java.

4/ Récursion.

- ➊ Introduction & exemples.
- ➋ Algorithmes récursifs.
- ➌ Les différents types de récursivité.
- ➍ Récursif ou itératif.
- ➎ Complexités de nos exemples.

Programme récursif

Un **programme** est dit **récursif** s'il fait appel à lui-même (du latin *recurrere* ~ courir en arrière).

Programme récursif

Un **programme** est dit **récursif** s'il fait appel à lui-même (du latin *recurrere* ~ courir en arrière).

Conséquence.

Pour pouvoir s'appeler, un programme récursif est donc forcément une **fonction** !

Programme récursif

Un **programme** est dit **récursif** s'il fait appel à lui-même (du latin *recurrere* ~ courir en arrière).

Conséquence.

Pour pouvoir s'appeler, un programme récursif est donc forcément une **fonction** !

Un exemple de tous les jours: factoriel!

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

```
public static long factoriel (int n) {  
    if(n<2) return 1;  
    return n* factoriel (n-1);  
}
```

Un autre exemple : recherche d'un élément dans un tableau

On veut tester si un entier v est présent dans un tableau t .

Un autre exemple : recherche d'un élément dans un tableau

On veut tester si un entier v est présent dans un tableau t .

Propriétés

- $v \in t[1, r]$ si et seulement si $v \in t[1, m]$ **OU** $v \in t[m+1, r]$.
- $v \in t[\ell, \ell]$ si et seulement si $v = t[\ell]$.

En utilisant ces propriétés, on en déduit le programme récursif suivant :

Recherche d'un élément en Java

/★ recherche récursive de v dans t dans $[l, r]$ ★/

```
static boolean estPresent (int v, int[] t, int  $\ell$ , int r) {  
    if( $\ell == r$ ) return  $t[\ell] == v$ ;  
    int  $m = (\ell + r) / 2$ ;  
    return estPresent (v, t,  $\ell$ , m) | estPresent (v, t, m+1, r);  
}
```

On note le “|” qui permet de ne pas faire appel au second **estPresent** si le premier appel retourne déjà **true**.

Maximum/minimum d'un tableau

On veut calculer le maximum (resp. minimum) d'un tableau.

Maximum/minimum d'un tableau

On veut calculer le maximum (resp. minimum) d'un tableau.

Propriétés

- $\max_{i \in [1, r]} t[i] = \max(\max_{i \in [1, m]} t[i], \max_{i \in [m+1, r]} t[i])$
- $\max_{i \in [\ell, \ell]} t[i] = t[\ell].$

Maximum/minimum d'un tableau

On veut calculer le maximum (resp. minimum) d'un tableau.

Propriétés

- $\max_{i \in [1, r]} t[i] = \max(\max_{i \in [1, m]} t[i], \max_{i \in [m+1, r]} t[i])$
- $\max_{i \in [\ell, \ell]} t[i] = t[\ell].$

En utilisant ces propriétés, on en déduit le programme récursif suivant :

Recherche récursive du max

// recherche du max par diviser pour régner

```
static int max (int[] t, int  $\ell$ , int r) {  
    if( $\ell == r$ ) return t[ $\ell$ ];  
    int m=( $\ell+r$ )/2;  
    int a= max (t, $\ell$ ,m);  
    int b= max (t,m+1,r);  
    return (a>b)?a:b;  
}
```

L'exécution d'un algorithme récursif

Calcul de 4!:

factoriel(4)

L'exécution d'un algorithme récursif

Calcul de 4!:

$\text{factoriel}(4) \Rightarrow 4.\text{factoriel}(3)$

L'exécution d'un algorithme récursif

Calcul de 4!:

$\text{factoriel}(4) \Rightarrow 4.\text{factoriel}(3) \Rightarrow 4.3.\text{factoriel}(2)$

L'exécution d'un algorithme récursif

Calcul de 4!:

$\text{factoriel}(4) \Rightarrow 4.\text{factoriel}(3) \Rightarrow 4.3.\text{factoriel}(2) \Rightarrow 4.3.2.\text{factoriel}(1)$

L'exécution d'un algorithme récursif

Calcul de 4!:

$\text{factoriel}(4) \Rightarrow 4.\text{factoriel}(3) \Rightarrow 4.3.\text{factoriel}(2) \Rightarrow 4.3.2.\text{factoriel}(1) \Rightarrow 4.3.2.1 = 24$

Le principe de l'exécution

Le principe de l'exécution est simple : l'algorithme met de côté (**en fait dans une PILE**) ce qu'il ne sait pas faire quitte à les reprendre après!

L'exécution d'un algorithme récursif

Calcul de 4!:

$\text{factoriel}(4) \Rightarrow 4.\text{factoriel}(3) \Rightarrow 4.3.\text{factoriel}(2) \Rightarrow 4.3.2.\text{factoriel}(1) \Rightarrow 4.3.2.1 = 24$

Le principe de l'exécution

Le principe de l'exécution est simple : l'algorithme met de côté (**en fait dans une PILE**) ce qu'il ne sait pas faire quitte à les reprendre après!

Le principe de la programmation

Par conséquent, on doit dire à l'algorithme comment faire dans les cas les plus simples \Rightarrow **LES CONDITIONS D'ARRÊT** ! Dans les cas complexes, l'algorithme va faire appel à lui-même en **SIMPLIFIANT** ces cas jusqu'à tomber sur l'une des conditions d'arrêt.

L'exécution d'un algorithme récursif

Calcul de 4!:

$\text{factoriel}(4) \Rightarrow 4.\text{factoriel}(3) \Rightarrow 4.3.\text{factoriel}(2) \Rightarrow 4.3.2.\text{factoriel}(1) \Rightarrow 4.3.2.1 = 24$

Le principe de l'exécution

Le principe de l'exécution est simple : l'algorithme met de côté (**en fait dans une PILE**) ce qu'il ne sait pas faire quitte à les reprendre après!

Le principe de la programmation

Par conséquent, on doit dire à l'algorithme comment faire dans les cas les plus simples \Rightarrow **LES CONDITIONS D'ARRÊT** ! Dans les cas complexes, l'algorithme va faire appel à lui-même en **SIMPLIFIANT** ces cas jusqu'à tomber sur l'une des conditions d'arrêt.

Exemples de programmes FAUX!

Si ces principes ne sont pas appliqués, l'algorithme risque soit de ne pas **produire de résultat**, soit de **tourner à l'infini**.

Exemples de programmes FAUX!

Si ces principes ne sont pas appliqués, l'algorithme risque soit de ne pas **produire de résultat**, soit de **tourner à l'infini**.

Pas de conditions d'arrêt \Rightarrow CALCUL INFINI

```
public static long factoriel (int n) {  
    return n* factoriel (n-1);  
}
```

Exemples de programmes FAUX!

Si ces principes ne sont pas appliqués, l'algorithme risque soit de ne pas **produire de résultat**, soit de **tourner à l'infini**.

Pas de conditions d'arrêt \Rightarrow CALCUL INFINI

```
public static long factoriel (int n) {  
    return n* factoriel (n-1);  
}
```

Pas de simplification du cas à traiter

```
/* Dans le cas suivant,  
** on ne dirige pas le programme vers les conditions  
** d'arrêt */  
public static long factoriel (int n) {  
    if n<2 return(1);  
    return factoriel (n+1)/(n+1);  
}
```

Première règle.

Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif. Souvent ce sont les cas les plus simples.

Première règle.

Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif. Souvent ce sont les cas les plus simples.

Sans cela, on risque de tourner en rond et de faire des exécutions qui ne se finissent pas.

Première règle.

Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif. Souvent ce sont les cas les plus simples.

Sans cela, on risque de tourner en rond et de faire des exécutions qui ne se finissent pas.

Conditions d'arrêt.

Ces cas non récursifs sont appelés **cas de base** et les conditions que doivent satisfaire les données dans ces cas de base sont appelées **conditions d'arrêt** ou de **terminaison**.

Seconde règle.

On doit conduire le programme **vers** les cas de base : **tout appel récur­sif doit se faire avec des données plus proches des conditions de terminaison!**

Seconde règle.

On doit conduire le programme **vers** les cas de base : **tout appel récursif doit se faire avec des données plus proches des conditions de terminaison!**

Cette règle utilise le théorème suivant :

Théorème.

Il n'existe pas de suite infinie strictement décroissante d'entiers positifs ou nuls.

Seconde règle.

On doit conduire le programme **vers** les cas de base : **tout appel récursif doit se faire avec des données plus proches des conditions de terminaison!**

Cette règle utilise le théorème suivant :

Théorème.

Il n'existe pas de suite infinie strictement décroissante d'entiers positifs ou nuls.

Ainsi, l'arrêt de l'exécution d'un programme récursif est **garanti** quand les deux règles sont appliquées!

Les différents types de récursivité

Dans ce cours, nous allons voir 3 types:

- ➊ récursivité **multiple** ,
- ➋ récursivité **croisée** ou **mutuelle** ,
- ➌ récursivité **imbriquée** .

Les différents types de récursivité

Dans ce cours, nous allons voir 3 types:

- ❶ récursivité **multiple** ,
- ❷ récursivité **croisée** ou **mutuelle** ,
- ❸ récursivité **imbriquée** .

Définition récursivité multiple

Un programme récursif est **multiple** si l'un des cas qu'il traite se résout avec plusieurs appels récursifs.

Les différents types de récursivité

Dans ce cours, nous allons voir 3 types:

- 1 récursivité **multiple** ,
- 2 récursivité **croisée** ou **mutuelle** ,
- 3 récursivité **imbriquée** .

Définition récursivité multiple

Un programme récursif est **multiple** si l'un des cas qu'il traite se résout avec plusieurs appels récursifs.

Définition récursivité croisée/mutuelle

Deux algorithmes sont **mutuellement récursifs** si l'un fait appel à l'autre et vice-versa.

Les différents types de récursivité

Dans ce cours, nous allons voir 3 types:

- 1 récursivité **multiple** ,
- 2 récursivité **croisée** ou **mutuelle** ,
- 3 récursivité **imbriquée** .

Définition récursivité multiple

Un programme récursif est **multiple** si l'un des cas qu'il traite se résout avec plusieurs appels récursifs.

Définition récursivité croisée/mutuelle

Deux algorithmes sont **mutuellement récursifs** si l'un fait appel à l'autre et vice-versa.

Définition récursivité imbriquée

Une fonction contient une **récursivité imbriquée** s'il contient comme paramètre un appel à lui-même

Une fonction imbriquée anecdotique : la fonction 91 de [MACCARTHY]

```
public static int f(int n) {  
    if (n>100) return (n-10);  
    else return f(f(n+11));  
}
```

Une fonction imbriquée anecdotique : la fonction 91 de [MACCARTHY]

```
public static int f(int n) {  
    if (n>100) return (n-10);  
    else return f(f(n+11));  
}
```

n-10 OU 91

Cette fonction retourne soit $n - 10$ si $n > 100$, soit 91! Par ailleurs, elle ne suit pas la seconde règle d'écriture des fonctions récursives.

Sa récursivité est à la fois **binaire** et **imbriquée** . Elle a le mérite d'être une fonction qui croît très très vite!

Sa récursivité est à la fois **binaire** et **imbriquée** . Elle a le mérite d'être une fonction qui croît très très vite!

Le code d'Ackermann

```
static int ack(int m, int n) {  
    if (m == 0) return n + 1;  
    else  
        if (n == 0) return ack (m - 1, 1);  
        else return ack (m - 1, ack (m, n - 1));  
}
```

La fonction d'Ackermann

Sa récursivité est à la fois **binaire** et **imbriquée** . Elle a le mérite d'être une fonction qui croît très très vite!

Le code d'Ackermann

```
static int ack(int m, int n) {  
    if (m == 0) return n + 1;  
    else  
        if (n == 0) return ack (m - 1, 1);  
        else return ack (m - 1, ack (m, n - 1));  
}
```

Une croissance phénoménale

On a $\text{ack}(0,n) = n+1$, $\text{ack}(1,n) = n+2$ mais $\text{ack}(5,1)$ est DÉJÀ TRÈS GRAND devant 2^{65536} qui est très grand par rapport à 10^{80} !

Remarque

Java passe les arguments des fonctions par valeur. Il calcule donc toujours l'argument avant de trouver le résultat d'une fonction.

Remarque

Java passe les arguments des fonctions par valeur. Il calcule donc toujours l'argument avant de trouver le résultat d'une fonction.

Dans le cas de la fonction de Morris, on a le code suivant.

Le code

```
static int g(int m, int n) {  
    if (m == 0) return 1;  
    else return g(m - 1, g(m, n));  
}
```

Remarque

Java passe les arguments des fonctions par valeur. Il calcule donc toujours l'argument avant de trouver le résultat d'une fonction.

Dans le cas de la fonction de Morris, on a le code suivant.

Le code

```
static int g(int m, int n) {  
    if (m == 0) return 1;  
    else return g(m - 1, g(m, n));  
}
```

On peut se poser la question: **que vaut $g(1, 0)$?**

Remarque

Java passe les arguments des fonctions par valeur. Il calcule donc toujours l'argument avant de trouver le résultat d'une fonction.

Dans le cas de la fonction de Morris, on a le code suivant.

Le code

```
static int g(int m, int n) {  
    if (m == 0) return 1;  
    else return g(m - 1, g(m, n));  
}
```

On peut se poser la question: **que vaut $g(1, 0)$?**

$$g(1, 0) = g(0, g(1, 0)).$$

Le code

```
static int s(int n) {  
    if (n==1) return 1;  
    else {  
        if (n % 2 == 0) return s(n/2);  
        else return s(3*n+1);  
    }  
}
```

Le code

```
static int s(int n) {  
    if (n==1) return 1;  
    else {  
        if (n % 2 == 0) return s(n/2);  
        else return s(3*n+1);  
    }  
}
```

Problème ouvert

Sur plein d'exemples, on trouve $s(n) = 1$ mais personne ne sait le prouver mathématiquement!

Dérécursion

Théoriquement on peut toujours **dérécursiver** un algorithme récursif, c'est-à-dire le transformer en **algorithme itératif**. En pratique, ce n'est pas toujours facile!

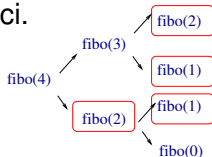
Dérécursion

Théoriquement on peut toujours **dérécursiver** un algorithme récursif, c'est-à-dire le transformer en **algorithme itératif**. En pratique, ce n'est pas toujours facile!

Factoriel itératif

```
public static int fact(int n) {  
    int res=1;  
    int i=1;  
    for (i=1;i<= n;i++) {  
        res=res*i;  
    }  
    return res;  
}
```

- L'exécution d'une version récursive d'un programme est moins rapide que celle de la version itérative même quand le nombre d'instructions restera le même!
- La version récursive d'un algorithme peut parfois conduire à exécuter bien plus d'instructions que la version itérative : exemple Fibonacci.



On va revenir sur les complexités de deux exemples (les fonctions max et la recherche d'un élément).

On va revenir sur les complexités de deux exemples (les fonctions max et la recherche d'un élément).

Max

Que ce soit dans le cas pire/ en moyenne ou dans le cas meilleur, la fonction **max** nécessite **$O(n)$** itérations.

On va revenir sur les complexités de deux exemples (les fonctions max et la recherche d'un élément).

Max

Que ce soit dans le cas pire/ en moyenne ou dans le cas meilleur, la fonction **max** nécessite **$O(n)$** itérations.

Idée de la preuve.

Si $T(N)$ est le nombre d'invocations pour une entrée de taille N , et si un appel de taille N réalise un appel pour une entrée de taille K ($K < N$) et une entrée de taille $N-K$ on obtient l'équation: $T(N) = T(N-K) + T(K)$ avec $T(1) = 0$. On montre par induction que $T(N) = N-1$ est solution des équations précédentes. Si la somme des tailles des entrées pour un appel sur une entrée de taille N est inférieur à N , $T(N)$ sera majorée par $N-1$.

Complexité de la recherche dichotomique d'un élément

- Dans le meilleur des cas, on va de suite trouver l'élément recherché. L'algorithme nécessite alors $O(1)$ itération.
- Dans le cas pire, on va rechercher la présence de l'élément dans tous les **sous-tableaux**, on fera alors $O(\log n)$ itérations (**si le tableau est trié**). Sinon on fera $O(\log n)$ itérations
- En moyenne (hypothèses : p est la probabilité de présence de l'élément dans le tableau), on fera aussi $O(\log n)$ itérations (**si trié**). Sinon on fera $O(\log n)$ itérations.

Complexité de la recherche dichotomique d'un élément

- Dans le meilleur des cas, on va de suite trouver l'élément recherché. L'algorithme nécessite alors $O(1)$ itération.
- Dans le cas pire, on va rechercher la présence de l'élément dans tous les **sous-tableaux**, on fera alors $O(\log n)$ itérations (**si le tableau est trié**). Sinon on fera $O(\log n)$ itérations
- En moyenne (hypothèses : p est la probabilité de présence de l'élément dans le tableau), on fera aussi $O(\log n)$ itérations (**si trié**). Sinon on fera $O(\log n)$ itérations. **Explications :**

$$(1 - p) \log_2(n) + p \dots$$

Ici, il n'y a point besoin de trop de calculs pour m.q. la moyenne est aussi $O(\log n)$ (resp. $O(n)$).