

Algorithmique avancée : introduction à la récursivité (partie 1)

Marin Bougeret
IUT Montpellier



Contenu

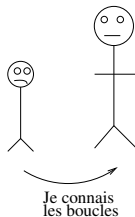
- récursivité (≈ 6 semaines)
- complexité (≈ 2 semaines)



"To iterate is human, to recurse divine." L. Peter Deutsch

Contenu

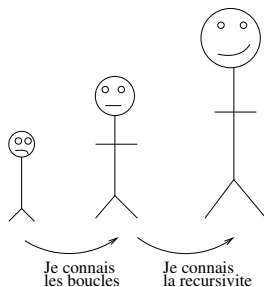
- récursivité (≈ 6 semaines)
- complexité (≈ 2 semaines)



"To iterate is human, to recurse divine." L. Peter Deutsch

Contenu

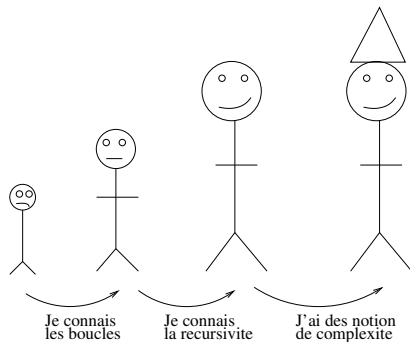
- récursivité (≈ 6 semaines)
- complexité (≈ 2 semaines)



"To iterate is human, to recurse divine." L. Peter Deutsch

Contenu

- récursivité (≈ 6 semaines)
- complexité (≈ 2 semaines)



"To iterate is human, to recurse divine." L. Peter Deutsch

Organisation

- langage en TP : java, langage en cours : pseudo java :)
 - 1 partiel final sur table (coeff 75%)
 - 1 note de TD (coeff 25%)
- slides/TP/TD disponibles après le cours sur moodle

Partie I : algorithmes récursifs

- **conception d'algorithmes récursifs (I) : exemples introductifs**
- conception d'algorithmes récursifs (II) : tests, idée de preuve, exemples
- conception d'algorithmes récursifs (III) : diviser pour régner

Partie II : structures récursives (listes, arbres..)

- ..

Partie III : complexité

- ..

Motivation pour la récursivité

Dans quels cas a-t-on besoin de la récursivité ?

- pour définir et manipuler simplement des structures complexes
- pour écrire des algorithmes
 - dont la spécification est elle même récursive (calculer une suite)
 - mais **bien plus** : la récursivité permet d'écrire facilement des algorithmes qui seraient extrêmement complexes en itératif (Hanoï, Saut de Puce, ..)

Motivation pour le récursivité

Dans quels cas a-t-on besoin de la récursivité ?

- pour définir et manipuler simplement des structures complexes
- pour écrire des algorithmes
 - dont la spécification est elle même récursive (calculer une suite)
 - mais **bien plus** : la récursivité permet d'écrire facilement des algorithmes qui seraient extrêmement complexes en itératif (Hanoï, Saut de Puce, ..)



Motivation pour le récursivité

Dans quels cas a-t-on besoin de la récursivité ?

- pour définir et manipuler simplement des structures complexes
- pour écrire des algorithmes
 - dont la spécification est elle même récursive (calculer une suite)
 - mais **bien plus** : la récursivité permet d'écrire facilement des algorithmes qui seraient extrêmement complexes en itératif (Hanoï, Saut de Puce, ..)



Motivation pour le récursivité

Dans quels cas a-t-on besoin de la récursivité ?

- pour définir et manipuler simplement des structures complexes
- pour écrire des algorithmes
 - dont la spécification est elle même récursive (calculer une suite)
 - mais **bien plus** : la récursivité permet d'écrire facilement des algorithmes qui seraient extrêmement complexes en itératif (Hanoï, Saut de Puce, ..)



Motivation pour le récursivité

Dans quels cas a-t-on besoin de la récursivité ?

- pour définir et manipuler simplement des structures complexes
- pour écrire des algorithmes
 - dont la spécification est elle même récursive (calculer une suite)
 - mais **bien plus** : la récursivité permet d'écrire facilement des algorithmes qui seraient extrêmement complexes en itératif (Hanoï, Saut de Puce, ..)



Définition

Un algorithme récursif est un algorithme qui s'appelle lui même :

```
...  A (...) {
```

```
    A (...) // "appel récursif"
```

```
}
```

Définition

Un algorithme récursif est un algorithme qui s'appelle lui même :

```
...  A(...){
```

```
    A(...) //"appel récursif"
```

```
}
```

Rmq : un algorithme récursif peut contenir plusieurs appels récursifs

Définition

Un algorithme récursif est un algorithme qui s'appelle lui même :

```
...  A (...) {  
  
      A (...) // "appel récursif"  
  
      A (...) // "appel récursif"  
}
```

Rmq : un algorithme récursif peut contenir plusieurs appels récursifs

Définition/Notation

- on notera E l'ensembles des entrées vérifiant les prérequis
 - on partitionne $E = E_r \cup E_b$ avec
 - E_b l'ensemble des entrées (de E) traitées **sans** appel récursif
 - E_r l'ensemble des entrées (de E) traitées **avec** un appel récursif
 - un $x \in E_b$ s'appelle un cas de base (E_b est donc l'ensemble des cas de base)
-
- $E = \{2, 4, 6, \dots\}$ (et pas $E = \text{"les ints"}$)
 - $E_b = \{2, 4\}$, $E_r = \{6, \dots\}$

```
int  A(int x){ //prerequis : x pair et x > 0
    if(x==2) {return 4;}
    else if(x==4){return 6;}
    else {
        int z = A(x-2); //"appel récursif"
        return z+2;
    }
}
```


Exemple 1 : `int somme(int x)`

But : `somme(int x)` qui calcule $1 + 2 + \dots + x$, prérequis $x \geq 1$

```
int somme(int x){  
  
    ...  
  
    int temp = somme(x-1); // temp = 1+2+...+x-1  
    return temp+x;  
}
```

- $E = \{1, 2, \dots\}$ avec
 - $E_b = \{1\}$
 - $E_r = \{2, 3, \dots\}$

Exemple 1 : int somme(int x)

But : somme(int x) qui calcule $1 + 2 + \dots + x$, prérequis $x \geq 1$

```
int somme(int x){
    if(x==1) // cas de base
        return 1;
    else{
        int temp = somme(x-1); // temp = 1+2+...+x-1
        return temp+x;
    }
}
```

- $E = \{1, 2, \dots\}$ avec
 - $E_b = \{1\}$
 - $E_r = \{2, 3, \dots\}$

Exemple 1 : int somme(int x)

But : somme(int x) qui calcule $1 + 2 + \dots + x$, prérequis $x \geq 1$

```
int somme(int x){  
    if(x==1) // cas de base  
        return 1;  
    else{  
        int temp = somme(x-1); // temp = 1+2+...+x-1  
        return temp+x;  
    }  
}
```

- $E = \{1, 2, \dots\}$ avec
 - $E_b = \{1\}$
 - $E_r = \{2, 3, \dots\}$

Exemple 1 : int somme(int x)

```
int somme(int x){  
    if(x==1) // cas de base  
        return 1;  
    else{  
        int temp = somme(x-1); // temp = 1+2+...+x-1  
        return temp+x;  
    }  
}
```

Pourquoi cela fonctionne ?

L'algorithme fonctionne car :

- la taille de l'entrée diminue dans l'appel récursif
- l'algorithme fonctionne pour $x = 1$

→ on peut prouver par récurrence que l'algo. termine et retourne un résultat correct

⇒ c'est pour cela que l'on peut supposer que "temp = $1+2+...+x-1$ "

Exemple 1 : int somme(int x)

```
int somme(int x){  
    if(x==1) // cas de base  
        return 1;  
    else{  
        int temp = somme(x-1); // temp = 1+2+...+x-1  
        return temp+x;  
    }  
}
```

Pourquoi cela fonctionne ?

L'algorithme fonctionne car :

- la taille de l'entrée diminue dans l'appel récursif
 - l'algorithme fonctionne pour $x = 1$
- on peut prouver par récurrence que l'algo. termine et retourne un résultat correct
- ⇒ c'est pour cela que l'on peut supposer que "temp = 1+2+...+x-1"

Exemple 1 : int somme(int x)

```
int somme(int x){  
    if(x==1) // cas de base  
        return 1;  
    else{  
        int temp = somme(x-1); // temp = 1+2+...+x-1  
        return temp+x;  
    }  
}
```

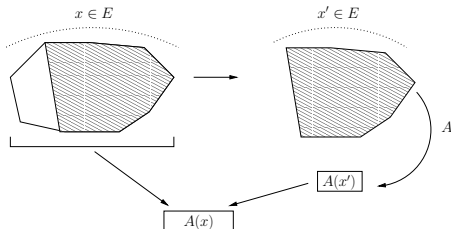
Pourquoi cela fonctionne ?

L'algorithme fonctionne car :

- la taille de l'entrée diminue dans l'appel récursif
 - l'algorithme fonctionne pour $x = 1$
- on peut prouver par récurrence que l'algo. termine et retourne un résultat correct
- ⇒ c'est pour cela que l'on peut supposer que "temp = $1+2+...+x-1$ "

Méthode pour concevoir un algorithme récursif A

- I) écrire A en supposant que A fonctionne déjà pour des entrées plus petites :
- penser à une grande entrée x
 - supposer que pour tout $x' \in E$ plus petite que x , $A(x')$ donnera la bonne réponse
 - en s'autorisant à faire de tel(s) appel(s) récursif(s), déduire la bonne réponse pour x
- II) pour les cas restants à traiter (où le code précédent n'a pas de sens): on ajoute un/des **cas de base(s)**



Exemple 2 : double sommeBiz(int x)

But

- prérequis : $x \geq 0$
- action:
 - $\text{sommeBiz}(0)=0$
 - $\text{sommeBiz}(1)=0$
 - $\text{sommeBiz}(2)=1$
 - $\text{sommeBiz}(3)=\frac{1}{2} + 1$
 - $\text{sommeBiz}(4)=\frac{1}{3} + \frac{1}{2} + 1$
 - $\text{sommeBiz}(x)=\frac{1}{(x-1)} + \frac{1}{x-2} + \dots + 1$

Exemple 2 : double sommeBiz(int x)

But : $\text{sommeBiz}(\text{int } x) = \frac{1}{x-1} + \frac{1}{x-2} + \dots + 1$ si $x \geq 2$, 0 sinon
(pré requis $x \geq 0$)

```
double sommeBiz(int x){  
  
    ...  
  
    int tmp = sommeBiz(x-1);  
    // tmp = 1/(x-2)+1/(x-3)+...+1  
    return tmp+1/(x-1);  
}
```

- $E = \mathbb{N}$
 - $E_b = \{0\}$
 - $E_r = \{1, 2, 3, \dots\}$

Exemple 2 : double sommeBiz(int x)

But : $\text{sommeBiz}(\text{int } x) = \frac{1}{x-1} + \frac{1}{x-2} + \dots + 1$ si $x \geq 2$, 0 sinon
(pré requis $x \geq 0$)

```
double sommeBiz(int x){  
    if(x==0) // cas de base  
        return 0;  
    else{  
        int tmp = sommeBiz(x-1);  
        // tmp = 1/(x-2)+1/(x-3)+..+1  
        return tmp+1/(x-1);  
    }  
}
```

- $E = \mathbb{N}$
 - $E_b = \{0\}$
 - $E_r = \{1, 2, 3, \dots\}$

Exemple 2 : double sommeBiz(int x)

But : $\text{sommeBiz}(\text{int } x) = \frac{1}{x-1} + \frac{1}{x-2} + \dots + 1$ si $x \geq 2$, 0 sinon
(pré requis $x \geq 0$)

```
double sommeBiz(int x){  
    if(x==0) // cas de base  
        return 0;  
    else{  
        int tmp = sommeBiz(x-1);  
        // tmp = 1/(x-2)+1/(x-3)+...+1  
        return tmp+1/(x-1);  
    }  
}
```

- $E = \mathbb{N}$
 - $E_b = \{0\}$
 - $E_r = \{1, 2, 3, \dots\}$

Exemple 2 : double sommeBiz(int x)

L'arnaque

- c'est faux!!
- que se passe-t-il avec $x = 1$?

Une correction:

Exemple 2 : double sommeBiz(int x)

But : $\text{sommeBiz}(\text{int } x) = \frac{1}{x-1} + \frac{1}{x-2} + \dots + 1$ si $x \geq 2$, 0 sinon
(prérequis $x \geq 0$)

```
double sommeBiz(int x){
    if(x == 0) return 0; // cas de base
    if(x == 1) return 0; // cas de base
    else{
        int tmp = sommeBiz(x-1);
        // tmp = 1/(x-2)+1/(x-3)+..+1
        return tmp+1/(x-1);
    }
}
```

- $E = \mathbb{N}$
 - $E_b = \{0, 1\}$
 - $E_r = \{2, 3, \dots\}$

Exemple 2 : double sommeBiz(int x)

But : $\text{sommeBiz}(\text{int } x) = \frac{1}{x-1} + \frac{1}{x-2} + \dots + 1$ si $x \geq 2$, 0 sinon
(prérequis $x \geq 0$)

```
double sommeBiz(int x){  
    if(x == 0) return 0; // cas de base  
    if(x == 1) return 0; // cas de base  
    else{  
        int tmp = sommeBiz(x-1);  
        // tmp = 1/(x-2)+1/(x-3)+...+1  
        return tmp+1/(x-1);  
    }  
}
```

- $E = \mathbb{N}$
 - $E_b = \{0, 1\}$
 - $E_r = \{2, 3, \dots\}$

Exemple 2 : double sommeBiz(int x)

- comment déterminer rigoureusement les cas de base à ajouter à E_b ?

Méthode pour concevoir un algorithme récursif A

- 1) écrire A en supposant que A fonctionne déjà pour des entrées plus petites :
- penser à une grande entrée x
 - supposer que pour tout $x' \in E$ plus petite que x , $A(x')$ donnera la bonne réponse
 - en s'autorisant à faire de tel(s) appel(s) récursif(s), déduire la bonne réponse pour x

On dit que $x \in E$ provoque une erreur dans le code 1) si au moins une des conditions suivantes est vraie

- 1 il y a une instruction incorrecte (division par 0, sortie de tableau, ..)
- 2 il y a un appel récursif $A(x')$ incorrect (x' ne vérifie pas les prérequis ($x' \notin E$), ou x' n'est pas plus petite que x)
- 3 tous les appels récursifs sont corrects, mais le calcul pour en déduire le résultat pour x est faux

Quelles entrées de E provoquent une erreur ?

```
double sommeBiz(int x){  
    //prerequis: x >= 0  
  
    int tmp = sommeBiz(x-1);  
    // tmp = 1/(x-2)+1/(x-3)+...+1  
    return tmp+1/(x-1);  
  
}
```

- $x = 0$ (sommeBiz(x-1): $(x-1) \notin E$)
- $x = 1$ ($1/(x-1)$ est une instruction incorrecte)
- $x \geq 2$ ok

Entrées de E provoquant une erreur : $\{0, 1\}$

Quelles entrées de E provoquent une erreur ?

```
int somme(int x){  
    //prerequis: x >= 0  
  
    int temp = somme(x-1); // temp = 1+2+...+x-1  
    return temp+x;  
  
}
```

- $x = 0$ (somme($x-1$): $(x-1) \notin E$)
- $x \geq 1$ ok

Entrées de E provoquant une erreur : $\{0\}$

Quelles entrées de E provoquent une erreur ?

```
int A(int x){  
    //prerequis: x >= 0, x pair  
  
    int z = A(x-2); // "appel recursif"  
    return z+2;  
  
}
```

- $x = 0$ ($A(x-2)$: $(x-2) \notin E$)
- $x = 1$ non! pas dans E !
- $x \geq 2$, ok

Entrées de E provoquant une erreur : $\{0\}$

Méthode pour concevoir un algorithme récursif A

- I)** écrire A en supposant que A fonctionne déjà pour des entrées plus petites :
- penser à une grande entrée x
 - supposer que pour tout $x' \in E$ plus petite que x , $A(x')$ donnera la bonne réponse
 - en s'autorisant à faire de tel(s) appel(s) récursif(s), déduire la bonne réponse pour x
- II)** ajouter des cas de base:
- ajouter des cas de base pour traiter les entrées de E provoquant une erreur dans le code I)

conception d'algorithmes récur­sifs (I) : exemples introductifs

- exemple 2 : recherche
- exemple 3 : saut de puce
- exemple 4 : tri

Exemple 2 : boolean recherche(int[] t, int x)

But : recherche(t, x) qui retourne vrai ssi x est dans t (prérequis t non vide)

Méthode pour concevoir un algorithme récursif A

1) écrire A en supposant que A fonctionne déjà pour des entrées plus petites :

- penser à une grande entrée x
- supposer que pour tout $x' \in E$ plus petite que x , $A(x')$ donnera la bonne réponse
- en s'autorisant à faire de tel(s) appel(s) récursif(s), déduire la bonne réponse pour x

" A fonctionne déjà pour des entrées plus petites" \Leftrightarrow
 $recherche(t2, x)$ sait chercher x dans un tableau $t2$ plus petit

- que choisir pour x' ? $x' = t2$ où $t2 =$ les $n - 1$ dernières cases

Exemple 2 : boolean recherche(int[] t, int x)

But : recherche(t,x) qui retourne vrai ssi x est dans t (prérequis t non vide)

```
boolean recherche(int[] t, int x){
    ...

    int[] t2 = .. //t2 = t[1..(t.length-1)];
    boolean temp = recherche(t2,x);
    //vrai ssi x dans t[1..(t.length-1)];
    if(temp)
        return true
    else
        return (t[0]==x);
}
```

Exemple 2 : boolean recherche(int[] t, int x)

But : recherche(t, x) qui retourne vrai ssi x est dans t (prérequis t non vide)

Méthode pour concevoir un algorithme récursif A

//) ajouter les cas de base :

- ajouter des cas de base pour traiter les entrées de E provoquant une erreur dans le code I)

Exemple 2 : boolean recherche(int[] t, int x)

But : recherche(t,x) qui retourne vrai ssi x est dans t (prérequis t non vide)

```
boolean recherche(int[] t, int x){  
    ...  
  
    int[] t2 = .. //t2 = t[1..(t.length-1)];  
    boolean temp = recherche(t2,x);  
        //vrai ssi x dans t[1..(t.length-1)];  
    if(temp)  
        return true  
    else  
        return (t[0]==x);  
}
```

Quelles entrées de E provoquent une erreur dans I ?

les tableaux de longueur 1

Exemple 2 : boolean recherche(int[] t, int x)

But : recherche(t,x) qui retourne vrai ssi x est dans t (t non vide)

```
boolean recherche(int[] t, int x){
    if(t.length==1){return t[0]==x;}
    else{
        int[] t2 = .. //t2 = t[1..(t.length-1)];
        boolean temp = recherche(t2,x);
        //vrai ssi x dans t[1..(t.length-1)];
        if(temp)
            return true
        else
            return (t[0]==x);
    }
}
```

Rmq : parcours partiel ou total ?

Une astuce pour la récursivité sur les tableaux

- pour faire un appel récursif sur un tableau plus petit..
- .. on peut éviter de recopier le sous tableau correspondant !
- (cela a des conséquences sur le temps d'exécution de l'algorithme)

Pour cela, on va changer les spécifications de recherche, et résoudre un problème plus général.

rechercheAux(int []t, int x, int i):

- prérequis : $0 \leq i < t.length$, t non vide
- action : retourne vrai ssi x est dans $t[i..(t.length-1)]$

Exemple 2 : boolean rechercheAux(int[] t, int x, int i)

But : rechercheAux(t,x,i) qui retourne vrai ssi x dans t[i..(t.length-1)]

Méthode pour concevoir un algorithme récursif A

1) écrire A en supposant que A fonctionne déjà pour des entrées plus petites :

- penser à une grande entrée x
- supposer que pour tout $x' \in E$ plus petite que x, A(x') donnera la bonne réponse
- en s'autorisant à faire de tel(s) appel(s) récursif(s), déduire la bonne réponse pour x

"A fonctionne déjà pour des entrées plus petites" \Leftrightarrow
rechercheAux(..) sait chercher x dans une zone plus petite

- que choisir pour x' ? $x' = (t, x, i + 1)$

Exemple 2 : boolean rechercheAux(int[] t, int x,int i)

But : rechercheAux(t,x,i) qui retourne vrai ssi x dans t[i..(t.length-1)]

```
boolean rechercheAux(int[] t, int x, int i){  
    ...  
  
    boolean temp = rechercheAux(t,x,i+1);  
    //vrai ssi x dans t[(i+1)..(t.length-1)];  
    if(temp)  
        return true  
    else  
        return (t[i]==x);  
}
```

Exemple 2 : boolean rechercheAux(int[] t, int x, int i)

But : rechercheAux(t,x,i) qui retourne vrai ssi x dans t[i..(t.length-1)]

Méthode pour concevoir un algorithme récursif A

//) ajouter les cas de base :

- ajouter des cas de base pour traiter les entrées de E provoquant une erreur dans le code I)

Exemple 2 : boolean rechercheAux(int[] t, int x, int i)

But : rechercheAux(t,x,i) qui retourne vrai ssi x dans t[i..(t.length-1)]

```
boolean rechercheAux(int[] t, int x, int i){  
    ..  
  
    boolean temp = rechercheAux(t,x,i+1);  
    //vrai ssi x dans t[(i+1)..(t.length-1)]  
    };  
    if(temp)  
        return true  
    else  
        return (t[i]==x);  
}
```

Quelles entrées de E provoquent une erreur dans I ?

les entrées où $i = t.length - 1$

Exemple 2 : boolean rechercheAux(int[] t, int x, int i)

But : rechercheAux(t,x,i) qui retourne vrai ssi x dans t[i..(t.length-1)]

```
boolean rechercheAux(int[] t, int x, int i){
    if(i==t.length-1)
        return t[i]==x;
    else{
        boolean temp = rechercheAux(t,x,i+1);
        //vrai ssi x dans t[(i+1)..(t.length-1)]
        ];
        if(temp)
            return true
        else
            return (t[i]==x);
    }
}
```


Exemple 2 : boolean rechercheAux(int[] t, int x, int i)

Fin de l'histoire

- il ne faut pas oublier de fournir ce qui nous intéressait au début:
- recherche(t,x) qui retourne vrai ssi x est dans t (t non vide)

```
boolean recherche(int[] t, int x){  
    return rechercheAux(t,x,0);  
}
```

Remarque

recherche n'est plus récursif.

Remarque

- on veut maintenant écrire `recherche2(t,x)` qui traite aussi les tableaux vide (et retourne faux)
- le code précédent ne fonctionne plus : l'appel à `rechercheAux` est impossible avec `t` vide
- deux solutions pour adapter:

Solution 1 (bof)

```
boolean recherche2(int[] t, int x){  
    if(t.length==0)  
        return false;  
    else  
        return rechercheAux(t,x,0);  
}
```

Remarque

- on veut maintenant écrire `recherche2(t,x)` qui traite aussi les tableaux vide (et retourne faux)
- le code précédent ne fonctionne plus : l'appel à `rechercheAux` est impossible avec `t` vide
- deux solutions pour adapter:

Solution 2 (bien)

```
boolean recherche2(int[] t, int x){  
    return rechercheAux2(t,x,0);  
}
```

Il faut donc écrire `rechercheAux2` qui traite les tableaux vides

Utilisation des prérequis du type $i \leq t.length$

rechercheAux2(int []t, int x, int i):

- prérequis : $0 \leq i < t.length$, ~~t non vide~~
- action : retourne vrai ssi x est dans $t[i..(t.length-1)]$

Ce prérequis est équivalent au précédent : ne supporte toujours pas les tableaux vides

Utilisation des prérequis du type $i \leq t.length$

rechercheAux2(int []t, int x, int i):

- prérequis : $0 \leq i < t.length$, ~~t non vide~~
- action : retourne vrai ssi x est dans $t[i..(t.length-1)]$

Ce prérequis est équivalent au précédent : ne supporte toujours pas les tableaux vides

rechercheAux2(int []t, int x, int i):

- prérequis : $0 \leq i \leq t.length$
- action : retourne vrai ssi x est dans $t[i..(t.length-1)]$

Bizarre à priori mais ..

Utilisation des prérequis du type $i \leq t.length$

But : `rechercheAux2(t,x,i)` qui retourne vrai ssi x dans $t[i..(t.length-1)]$

Méthode pour concevoir un algorithme récursif A

1) écrire A en supposant que A fonctionne déjà pour des entrées plus petites :

- penser à une grande entrée x
- supposer que pour tout $x' \in E$ plus petite que x , $A(x')$ donnera la bonne réponse
- en s'autorisant à faire de tel(s) appel(s) récursif(s), déduire la bonne réponse pour x

" A fonctionne déjà pour des entrées plus petites" \Leftrightarrow
`rechercheAux2(..)` sait chercher x dans une zone plus petite

- que choisir pour x' ? $x' = (t, x, i + 1)$

But : rechercheAux2(t,x,i) qui retourne vrai ssi x dans t[i..(t.length-1)]

```
boolean rechercheAux2(int[] t, int x, int i){
    ...

    boolean temp = rechercheAux2(t,x,i+1);
    //vrai ssi x dans t[(i+1)..(t.length-1)];
    if(temp)
        return true
    else
        return (t[i]==x);
}
```

But : rechercheAux2(t, x, i) qui retourne vrai ssi x dans $t[i..(t.length-1)]$

Méthode pour concevoir un algorithme récursif A

//) ajouter les cas de base :

- ajouter des cas de base pour traiter les entrées de E provoquant une erreur dans le code I)

Utilisation des prérequis du type $i \leq t.length$

But : rechercheAux2(t, x, i) qui retourne vrai ssi x dans $t[i..(t.length-1)]$

```
boolean rechercheAux2(int[] t, int x, int i){  
    ..  
  
    boolean temp = rechercheAux2(t2,x,i+1);  
    //vrai ssi x dans t[(i+1)..(t.length-1)]  
    };  
    if(temp)  
        return true  
    else  
        return (t[i]==x);  
}
```

Quelles entrées de E provoquent une erreur dans I ?

les entrées où $i = t.length$

Utilisation des prérequis du type $i \leq t.length$

But : rechercheAux2(t, x, i) qui retourne vrai ssi x dans $t[i..(t.length-1)]$

```
boolean rechercheAux2(int[] t, int x, int i){
    if(i==t.length)
        return false;
    else{
        boolean temp = rechercheAux2(t2,x,i+1);
        //vrai ssi x dans t[(i+1)..(t.length-1)]
        ];
        if(temp)
            return true
        else
            return (t[i]==x);
    }
}
```

Remarque : si t vide, alors forcément $i = 0$, et tout va bien.

Fin de l'histoire

- il ne faut pas oublier de fournir ce qui nous intéressait au début:
- `recherche2(t,x)` qui retourne vrai ssi x est dans t (pas de prérequis)

```
boolean recherche2(int[] t, int x){  
    return rechercheAux2(t,x,0);  
}
```

Conclusion

Des prérequis du type $i \leq t.length$ mènent à des algorithmes plus facile à écrire, et qui traitent plus de cas!

Exemple 3 : int sautPuce (int n)

- On considère une puce située à une altitude $n \geq 0$ cm
- A chaque étape, la puce peut faire un saut (vers le bas) de 2 cm, ou de 1 cm
- Question : étant donné une altitude de départ $n \geq 0$, combien de trajectoires possibles amènent la puce à l'altitude 0?

Par exemple, pour $n = 5$, on compte 8 trajectoires

- 5-> 3-> 1-> 0
- 5-> 3-> 2-> 0
- 5-> 3-> 2-> 1-> 0
- 5-> 4-> 2-> 0
- 5-> 4-> 2-> 1-> 0
- 5-> 4-> 3-> 1-> 0
- 5-> 4-> 3-> 2-> 0
- 5-> 4-> 3-> 2-> 1-> 0

Exemple 3 : `int sautPuce (int n)`

But : `sautPuce(n)` qui calcule le nb de trajectoires en partant de n

Méthode pour concevoir un algorithme récursif A

1) écrire A en supposant que A fonctionne déjà pour des entrées plus petites :

- penser à une grande entrée x
- supposer que pour tout $x' \in E$ plus petite que x , $A(x')$ donnera la bonne réponse
- en s'autorisant à faire de tel(s) appel(s) récursif(s), déduire la bonne réponse pour x

" A fonctionne déjà pour des entrées plus petites" $\Leftrightarrow \text{sautPuce}(n')$ sait calculer le nombre de trajectoires en partant de n' , pour tout $n' < n$

- que choisir pour x' ? $x' = ?$
- quel(s) appel(s) récursif(s) faire ?

Exemple 3 : int sautPuce (int n)

Reprenons le cas $n = 5$.

Supposons que l'on nous dise qu'il y a

- 3 trajectoires en partant de $n' = 3$
- 5 trajectoires en partant de $n' = 4$

Comment utiliser ces informations pour en déduire la réponse pour $n = 5$?

- 5-> 3-> ..
- 5-> 3-> ..
- 5-> 3-> ..

- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..

D'où $\text{sautPuce}(5) = \text{sautPuce}(3) + \text{sautPuce}(4)$

Exemple 3 : int sautPuce (int n)

Reprenons le cas $n = 5$.

Supposons que l'on nous dise qu'il y a

- 3 trajectoires en partant de $n' = 3$
- 5 trajectoires en partant de $n' = 4$

Comment utiliser ces informations pour en déduire la réponse pour $n = 5$?

- 5-> 3-> ..
- 5-> 3-> ..
- 5-> 3-> ..

- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..

D'où $\text{sautPuce}(5) = \text{sautPuce}(3) + \text{sautPuce}(4)$

Exemple 3 : int sautPuce (int n)

Reprenons le cas $n = 5$.

Supposons que l'on nous dise qu'il y a

- 3 trajectoires en partant de $n' = 3$
- 5 trajectoires en partant de $n' = 4$

Comment utiliser ces informations pour en déduire la réponse pour $n = 5$?

- 5-> 3-> ..
- 5-> 3-> ..
- 5-> 3-> ..

- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..

D'où $\text{sautPuce}(5) = \text{sautPuce}(3) + \text{sautPuce}(4)$

Exemple 3 : int sautPuce (int n)

Reprenons le cas $n = 5$.

Supposons que l'on nous dise qu'il y a

- 3 trajectoires en partant de $n' = 3$
- 5 trajectoires en partant de $n' = 4$

Comment utiliser ces informations pour en déduire la réponse pour $n = 5$?

- 5-> 3-> ..
- 5-> 3-> ..
- 5-> 3-> ..

- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..
- 5-> 4-> ..

D'où $\text{sautPuce}(5) = \text{sautPuce}(3) + \text{sautPuce}(4)$

Exemple 3 : int sautPuce (int n)

On en déduit le code suivant

```
int sautPuce(int n){  
    //n >= 0  
    //calcule le nb de traj. en partant de n  
  
    int temp1 = sautPuce(n-2);  
    int temp2 = sautPuce(n-1);  
    return temp1+temp2;  
  
}
```

Méthode pour concevoir un algorithme récursif A

//) ajouter les cas de base :

- ajouter des cas de base pour traiter les entrées de E provoquant une erreur dans le code I)

Exemple 3 : int sautPuce (int n)

```
int sautPuce(int n){  
    //n >= 0  
    //calcule le nb de traj. en partant de n  
  
    int temp1 = sautPuce(n-2);  
    int temp2 = sautPuce(n-1);  
    return temp1+temp2;  
  
}
```

Quelles entrées de E provoquent une erreur dans I ?

$n = 0$ et $n = 1$

Exemple 3 : int sautPuce (int n)

On en déduit le code suivant

```
int sautPuce(int n){  
    //n >= 0  
    //calcule le nb de traj. en partant de n  
    if(n==0)  
        return 1;  
    if(n==1)  
        return 1;  
    else{  
        int temp1 = sautPuce(n-2);  
        int temp2 = sautPuce(n-1);  
        return temp1+temp2;  
    }  
}
```

Que se passe-t-il si l'on oublie le cas de base $n == 1$?

Exemple 4 : void triInser (int []t)

But : triInser(*t*) qui trie *t* par ordre croissant

Méthode pour concevoir un algorithme récursif *A*

1) écrire *A* en supposant que *A* fonctionne déjà pour des entrées plus petites :

- penser à une grande entrée *x*
- supposer que pour tout $x' \in E$ plus petite que *x*, *A*(*x'*) donnera la bonne réponse
- en s'autorisant à faire de tel(*s*) appel(*s*) récursif(*s*), déduire la bonne réponse pour *x*

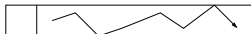
"*A* fonctionne déjà pour des entrées plus petites" \Leftrightarrow triInser(*t2*) sait trier un tableau *t2* plus petit

- que choisir pour x' ? $x' = ?$
- quel(*s*) appel(*s*) récursif(*s*) faire ?

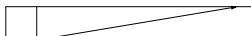
Exemple 4 : void triInser (int []t)

But : triInser(*t*) qui trie *t* par ordre croissant

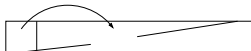
Idée:



Trier recursivement
les $n - 1$ derniers elements



Insérer $t[0]$ dans
le sous tableau trié



Exemple 4 : void triInsér (int []t)

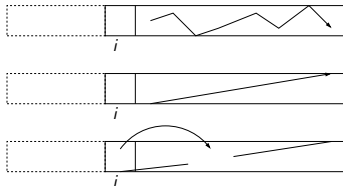
Astuce de l'ajout d'indice

Nous allons donc écrire void triInsérAux(int[] t, int i) qui pour $0 \leq i \leq t.length$ trie le sous tab. $t[i..(t.length - 1)]$ par ordre croissant. **On se fiche des éléments avant i !**

Attention, le dessin précédent devient :

Trier récursivement
le sous tableau $t[i+1]..t[t.length-1]$

Insérer $t[i]$ dans
le sous tableau trié



Exemple 4 : void triInser (int []t)

On en déduit le code suivant

```
void triInserAux(int[] t, int i){  
    // 0 <= i <= t.length  
    // trie le sous tableau t[i..(t.length-1)] par  
    // ordre croissant  
  
    triInserAux(t,i+1);  
    insere(t,i); //TODO  
}
```

Méthode pour concevoir un algorithme récursif A

//) ajouter les cas de base :

- ajouter des cas de base pour traiter les entrées de E provoquant une erreur dans le code I)

Exemple 4 : void triInser (int []t)

```
void triInserAux(int[] t, int i){  
    // 0 <= i <= t.length  
    // trie le sous tableau t[i..(t.length-1)] par  
    // ordre croissant  
  
    triInserAux(t,i+1);  
    insere(t,i); //TODO  
}
```

Quelles entrées de E provoquent une erreur dans I ?

les entrées où $i = t.length$

Exemple 4 : void triInser (int []t)

```
void triInserAux(int[] t, int i){  
    // 0 <= i <= t.length  
    // trie le sous tableau t[i..(t.length-1)] par  
    // ordre croissant  
    if( i >= t.length ){rien!}  
    else{  
        triInserAux(t,i+1);  
        insere(t,i); //TODO  
    }  
}
```

- bien écrire les spécifications de `insere(t,i)` :
 - étant donné $0 \leq i \leq t.length - 1$, et t tel que $t[(i+1)..(t.length-1)]$ est trié par ordre croissant
 - `insere(t,i)` modifie $t[i..(t.length-1)]$ tel que $t[i..(t.length-1)]$ soit trié par ordre croissant

Exemple 4 : void triInser (int []t)

```
void triInserAux(int[] t, int i){  
    // 0 <= i <= t.length  
    // trie le sous tableau t[i..(t.length-1)] par  
    // ordre croissant  
    if( i >= t.length ){rien!}  
    else{  
        triInserAux(t,i+1);  
        insere(t,i); //TODO  
    }  
}
```

- bien écrire les spécifications de `insere(t,i)` :
 - étant donné $0 \leq i \leq t.length - 1$, et t tel que $t[(i+1)..(t.length-1)]$ est trié par ordre croissant
 - `insere(t,i)` modifie $t[i..(t.length-1)]$ tel que $t[i..(t.length-1)]$ soit trié par ordre croissant

Exemple 4 : void triInser (int []t)

```
void triInserAux(int[] t, int i){  
    // 0 <= i <= t.length  
    // trie le sous tableau t[i..(t.length-1)] par  
    // ordre croissant  
    if( i >= t.length ){rien!}  
    else{  
        triInserAux(t,i+1);  
        insere(t,i); //TODO  
    }  
}
```

- bien écrire les spécifications de `insere(t,i)` :
 - étant donné $0 \leq i \leq t.length - 1$, et t tel que $t[(i+1)..(t.length-1)]$ est trié par ordre croissant
 - `insere(t,i)` modifie $t[i..(t.length-1)]$ tel que $t[i..(t.length-1)]$ soit trié par ordre croissant

Exemple 4 : void trlnser (int []t)

```
void insere(t,i){
    // 0 <= i <= t.length-1, t tel que
    // t[(i+1)..(t.length-1)] est trie croissant
    // modifie t[i..(t.length-1)] tel que
    // t[i..(t.length-1)] soit trie croissant
    int x = t[i];
    int j=i+1;
    boolean trouve = false;
    while(!trouve && (j < t.length)){
        if(t[j]<x){
            t[j-1]=t[j];
            j++;
        }
        else{
            trouve = true;
        }
    }
    //2 cas possibles, soit t[j]>=x,
    //soit j=t.length. Dans les 2 cas :
    t[j-1]=x;
}
```

Exemple 4 : void triInser (int []t)

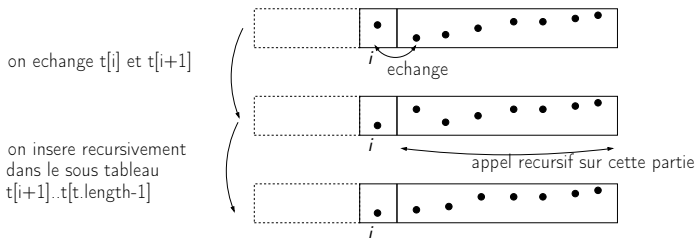
C'est fini ! Le code final est

```
void triInser(int [] t){  
    // trie t par ordre croissant  
    triInserAux(t,0);  
}
```

Pour s'amuser, on pourrait aussi écrire void insere(t,i) récursivement!

void insere(t, i) en récursif !

- si $t[i] \leq t[i + 1]$, on ne fait rien
- sinon :



Pour la prochaine fois : implémentez `trilInsert` avec `insere` en récursif.

Bilan : une remarque

- on parle partout de " x' plus petit que x " ..
- dans chaque cas, on a réussi à définir ce que veut dire "plus petit" ..
- .. de telle sorte qu'il n'y ait pas de "descente infinie" (on retombe toujours sur les cas de base après un nombre fini d'étapes)
 - (ici on arrive toujours à définir une taille qui associe à une entrée $x \in E$ un entier $t(x) \in \mathbb{N}$, on définit alors "plus petit" grâce à t)

Mais sur certains exemples il peut être compliqué de trouver une bonne définition de "plus petit" (discussion TD: ordre lexicographique, syracuse, pb avec les réels ..)

Mais ce n'est pas l'objet de ce cours : dans tous nos exemples on trouvera facilement un sens à "plus petit" qui empêche les descentes infinies

A retenir

- la méthode de conception : points I) et II)
- ne pas trouver les cas de base "par tâtonnement"
- astuce de l'ajout d'indice i pour les tableaux
 - utilisation prérequis $i \leq t.length$ souvent mieux