Développement efficace (R3.02)
Récursivité I : les bases

Marin Bougeret
LIRMM, IUT/Université de Montpellier

Outline

Admin

Les deux pages du module :

- sur moodle : R3.02 Développement efficace
- sur gitlab : `https://gitlabinfo.iutmontp.univ-montp2.fr/r3.02-dev-efficace/`

Sur moodle

- sallon BBB (et enregistrements de cours, disponibles environ 1H après la fin du cours)
- slides, sujets TD

Sur gitlab

- squelettes des sujets de TP (pas pour tous les TPs)

Admin

Organisation :

- cours de 7 semaines
- notation 30% CC, 70% exam final (sur papier, feuille A4 manuscrite recto verso autorisée)
- le CC est à la responsabilité de l'enseignant TD (mini(s) contrôle(s), noter les TP, ..)

Plan du module

1. récursivité : outil algorithmique puissant (porté dépasse module!)
   - d'abord sur des entiers, tableaux, ..
   - puis sur des listes, arbres : les premières structures de données que nous étudierons ici
2. complexité : modèle pour compter le nombre d'opérations d'un algorithme
   - pour donner un sens au terme "algorithme efficace"
   - pour se rendre compte que la puissance de la récursivité .. se paye parfois avec des algorithmes d'une grande complexité
   - nous donne le cadre nécessaire pour la partie III
3. structures de données : étude de classiques : arbres binaires de recherche, tas, arbres préfixes

Plan du module

Etude de structures de données : pourquoi ?

- pour des problèmes complexes, utiliser une bonne structure de donnée peut être critique pour avoir une bonne complexité
- exemple vu en 1A : TP taquin, on avait typiquement :
  - stockage des taquins déjà vus avec une `ArrayList` : résolution en 2 – 3 minutes
  - stockage des taquins déjà vus avec une `HashTable` : résolution en 2 – 3 secondes!

Etude de structures de données : but

But : avoir en tête un bilan des structures de donnée du type

opérations / structures	ajout	recherche	suppression	accès
tableau	..	O(n)	..	—
liste	O(1)	O(n)	...	—
arbre de recherche	...	O(log(n))	...	
⋮				

Plan du module

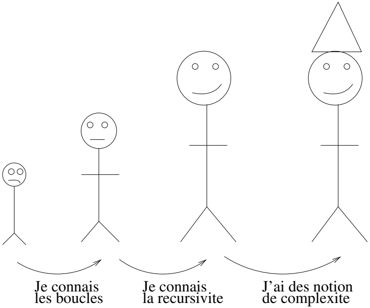
Etude de structures de données : comment ?

Pourquoi les coder soit même alors que dans la vraie vie on va les utiliser en boîte noire ? On pourrait juste les lister avec leur propriétés, et plus tard vous choisirez la bonne selon votre besoin, comme dans un catalogue ..

- ennui mortel
- il en existe des centaines, on ne peut pas toutes les connaître, mieux vaut connaître 4 OU 5 basiques et se dire "j'aurais besoin d'une qui combine un peu .. et .." (aide à exprimer son besoin pour chercher soit même)
- les recoder permet de voir quelques très jolies idées, bonnes à avoir pour sa culture
- et de plus, si votre besoin est très spécial et qu'aucune structure existante ne convient, vous pourrez recoder une version adaptée à votre problème

Outline

La récursivité : outil très puissant!



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

- tellement puissant qu'il permet d'écrire des algorithmes de quelques lignes très gourmands en calculs (typiquement un algorithme prenant en paramètre un tableau de n cases et faisant 2^n opérations.. quasi inutilisable en pratique!)
- d'où l'utilité de parler aussi un peu de complexité

Définition

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

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


      A(...) //"appel recursif"


}
```

Définition

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

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


       A(...) //"appel recursif"


}
```

Rmq : un algorithme récursif peut contenir plusieurs appels récursifs

Définition

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

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


         A(...) //"appel recursif"

         A(...) //"appel recursif"
}
```

Rmq : un algorithme récursif peut contenir plusieurs appels récursifs

Définition/Notation

- on notera E l'ensemble 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 $E =$ "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 recursif"
      return z+2;
    }
}
```

Introduction

Motivation pour le récursivité

Dans quels cas utilise-t-on la récursivité ?

- 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 utilise-t-on la récursivité ?

- 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

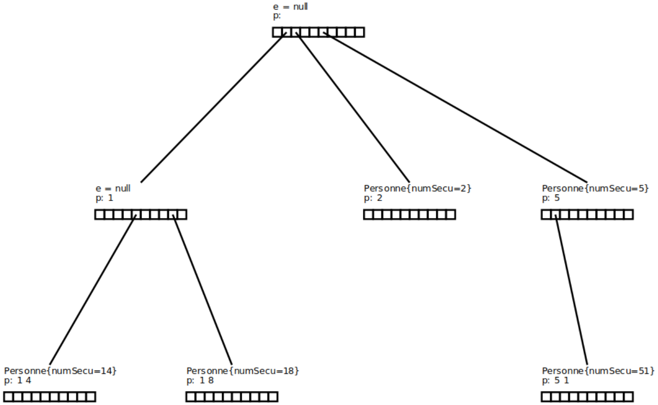
Dans quels cas utilise-t-on la récursivité ?

- 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 utilise-t-on la récursivité ?

- pour définir simplement des structures de données (listes, arbres, .. cf fin du cours) et écrire des algorithmes dessus



Exemple 0 : int u(int n)

On considère la suite u_n définie par

- $u_1 = 10,\ u_2 = 12$
- $\forall n \geq 3,\ u_n = u_{n-1} + 2u_{n-2} + (n+3)$

Solution 1

En itératif, on fait une boucle qui calcule u_1, puis u_2, ..

Exemple 0 : int u(int n)

On considère la suite u_n définie par

- $u_1 = 10$, $u_2 = 12$
- $\forall n \geq 3$, $u_n = u_{n-1} + 2u_{n-2} + (n+3)$

Solution 2

```
int u(int n){
  if(n==1) // cas de base
    return 10;
  if(n==2) // cas de base
    return 12;
  else{
    int temp = u(n-1);//  temp = u_{n-1}
    int temp2 = u(n-2);//  temp2 = u_{n-2}
    return temp+2*temp2+(n+3);
  }
}
```

Exemple 0 : int u(int n)

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

Solution 2

```
int u(int n){
  if(n==1) // cas de base
    return 10;
  if(n==2) // cas de base
    return 12;
  else{
    int temp = u(n-1);//  temp = u_{n-1}
    int temp2 = u(n-2);//  temp2 = u_{n-2}
    return temp+2*temp2+(n+3);
  }
}
```

Exemple 1 : int somme(int x)

But : `somme(int x)` qui calcule $1 + 2 + .. + x$, 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 ?

Se convaincre en exécutant à la main : bof :

- difficile à faire (imaginez quand il y a plusieurs appels récursifs)
- même quand on y arrive, n'aide pas à comprendre pourquoi l'algo est correct

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 ?

Plutôt voir les choses ainsi :

- l'algo est correct pour $x = 1$
- (P) j'ai écrit mon algo pour x en me disant "je suppose que ça marche pour $x - 1$", et j'écris un code qui marche pour x

Et donc

- comme il est correct pour $x = 1$, par (P) il est correct pour 2
- comme il est correct pour $x = 2$, par (P) il est correct pour 3
- ..

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 ?

Conclusion : si

- l'algo est correct pour les cas de base ($x = 1$), et que
- (P) j'ai écrit mon algo pour x en me disant "je suppose que ça marche pour $x - 1$", et j'écris un code qui marche pour x

Alors je pourrai prouver par récurrence qu'il est correct pour tout x

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 ?

Si je veux exécuter mon algo à la main pour $x = 5$,

- le faire pour $x = 1$, et noter le résultat (pour ne plus jamais le refaire)
- le faire pour $x = 2$, et noter le résultat (pour ne plus jamais le refaire)
- ..

Conclusions

Pour réussir à produire un code récursif $A(n)$, il faut écrire en supposant que les appels récursifs $A(n')$ fonctionnent dès lors que n' est plus petit que n.

- cette idée est valable aussi pour tout type de paramètres (pas que entier!) : tableaux, graphes, ... ! D'où la puissance de l'outil !
- il faut alors donner un sens à plus petit (taille du tableau, nb de sommets ..)

Conclusions

Autrement dit : pour réussir à produire un code récursif $A(n)$, il faut faire comme si les appels à $A(n')$ (avec n' plus petit que n) étaient des appels à une libraire déjà écrite

- cette idée est valable aussi pour tout type de paramètres (pas que entier!) : tableaux, graphes, ... ! D'où la puissance de l'outil !
- il faut alors donner un sens à plus petit (taille du tableau, nb de sommets ..)

Conclusions

Pour réussir à produire un code récursif $A(n)$, il faut écrire en supposant que les appels récursifs $A(n')$ fonctionnent dès lors que n' est plus petit que n.

- cette idée est valable aussi pour tout type de paramètres (pas que entier!) : tableaux, graphes, ... ! D'où la puissance de l'outil !
- il faut alors donner un sens à plus petit (taille du tableau, nb de sommets ..)

Choix des cas de base

Code typique

```
.. A(int x){
  //cas de base où x est petit (et qu'on sait
      traiter facilement)
  if(x est petit)
    return ..;


  else{
    //résolution avec un/des appels récursifs
    .. A(x2) ; // x2 plus petit que x
    return ..
  }
}
```

Choix des cas de base

Code typique

```
.. A(int x){
  //cas de base où x est petit (et qu'on sait
     traiter facilement)
  if(x == 1)
    return ..;
  if(x == 2)
    return ..;
  else{
    //résolution avec un/des appels récursifs
    .. A(x2) ; // x2 plus petit que x
    return ..
  }
}
```

Choix des cas de base = comment décider ce qui est petit

Retour Exemple 1 : int somme(int x)

On avait :

```
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;
  }
}
```

Retour Exemple 1 : int somme(int x)

Cette version est aussi correcte :

```
int somme(int x){
  if(x==1) // cas de base
    return 1;
  if(x==2) // un autre cas de base
    return 3;
  else{
    int temp = somme(x-1);// temp = 1+2+..+x-1
    return temp+x;
  }
}
```

Et donc, comment choisir ?

Wait .., nous allons répondre, mais avant, un autre exemple

Exemple 2 : double sommeBiz(int x)

But

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

Exemple 2 : double sommeBiz(int x)

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

```
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);
  }
}
```

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

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 : `sommeBiz(int x)` $= \frac{1}{x-1} + \frac{1}{x-2} + .. + 1$ 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);
  }
}
```

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

Comment choisir ses cas de base

Code typique

```
.. A(int x)
  if(x est petit) //cas de base
    return ..;
  else{ //cas traités par récurrence
    .. A(x2) ; // x2 plus petit que x
    return ..
  }
```

Pour les $x \in E$ traités par récurrence, il faut s'assurer que

1. toutes les instructions sont correctes (pas de division par 0, sortie de tableau, ..)
2. les appels récursifs $A(x')$ sont corrects (x' vérifie les prérequis ($x' \in E$), et x' plus petit que x)
3. le calcul qui déduit le résultat des appels récursifs est correct

Comment choisir ses cas de base

Principes pour choisir ses cas de base

- on choisit donc ses cas de base tels que, si un $x \in E$ ne rentre pas dedans (et qu'il est donc traité par récurrence), alors on ait les propriétés précédentes 1, 2 et 3
- on évite les cas de base inutiles (ex : pour `somme`, `x==1` suffit)
- une technique : écrire d'abord les cas traités par récurrence (en pensant à des x très grands), voir pour quels x elle n'est pas correct, et ajouter des cas de base pour ceux là

Outline

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

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

Rappel

Pour réussir à produire un code récursif $A(n)$, il faut écrire en supposant que les appels récursifs $A(n')$ fonctionnent dès lors que n' est plus petit que n.

"$A(n')$ fonctionne déjà pour n' plus petit" $\Leftrightarrow$ $\mathit{recherche}(t2, x)$ sait chercher x dans un tableau $t2$ plus petit

Que choisir pour n' ici ?

- par exemple $n' = t2$ où $t2$ est n'importe quel tableau à $n - 1$ cases
- disons $t2 =$ 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)

Principes pour choisir ses cas de base

Une technique : écrire d'abord les cas traités par récurrence (en pensant à des x très grands), voir pour quels x elle n'est pas correct, et ajouter des cas de base pour ceux là

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 $\mathcal{O}(n^2)$ vs $\mathcal{O}(n)$)

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)]

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

Rappel

Pour réussir à produire un code récursif $A(n)$, il faut écrire en supposant que les appels récursifs $A(n')$ fonctionnent dès lors que n' est plus petit que n.

"$A(n')$ fonctionne déjà pour n' plus petit" $\Leftrightarrow$ *rechercheAux*(...) sait chercher x dans une zone plus petite

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

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)]

```
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.

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

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);
}
```

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

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)

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

Bizarre à priori mais .. on obtient le code suivant, qui est donc plus général (puisque ok avec tableaux vides), et en bonus, dont le cas de base est encore plus facile à écrire

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

But : `rechercheAux2(t,x,i)` qui pour $0 \leq i \leq t.legnth$ 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(t,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!

Outline

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

int sautPuce (int n)

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

Rappel

Pour réussir à produire un code récursif $A(n)$, il faut écrire en supposant que les appels récursifs $A(n')$ fonctionnent dès lors que n' est plus petit que n.

"$A(n')$ fonctionne déjà pour n' plus petit" $\Leftrightarrow$ $sautPuce(n')$ sait calculer le nombre de trajectoires en partant de n', pour tout $n' < n$

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

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 sautPuce(5) = sautPuce(3)+sautPuce(4)

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 sautPuce(5) = sautPuce(3)+sautPuce(4)

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 sautPuce(5) = sautPuce(3)+sautPuce(4)

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 sautPuce(5) = sautPuce(3)+sautPuce(4)

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;

}
```

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 $n = 1$

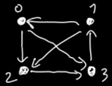
int sautPuce (int n)

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 $n == 1$?

Def : Un <u>tournoi</u> T est une orientation d'un graphe complet.

Ex : T_1 = 

est un tournoi (à 4 sommets)

idée : 

signifie "le joueur 3 a gagné le joueur 0"

Def : Un <u>chemin hamiltonien</u> dans un tournoi à n sommets est une liste $L = (v_0, \dots, v_{n-1})$ telle que $\forall i$, $\exists$ arc $v_i \longrightarrow v_{i+1}$ (et $v_i \in [0, n-1]$)

Ex $L = (0, 3, 1, 2)$ est un chemin hamiltonien de T_1

Théorème : Tout tournoi admet un chemin hamiltonien (et on va même écrire un algo qui en construit un !)

Petits tests :

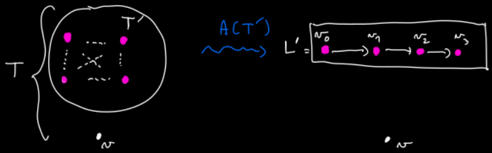
Algo ...

On cherche A(T){

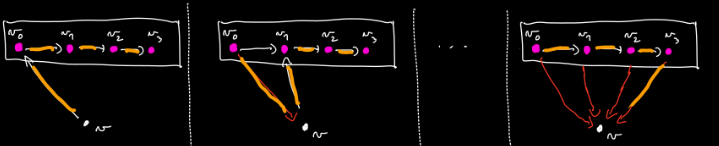
...

. utilisons la récurrence :

supposons que $\forall T'$ à $n-1$ sommets, $A(T')$ nous construit un chemin hamiltonien L' de T'



• Essayons de compléter L' en un chemin hamiltonien L de T :



cas 0 : si arc $v \to v_0$: ok

cas 1 : si pas cas 0, et $v \to v_1$: ok

cas $n-1$: si pas cas 0
et pas cas 1
et pas cas $n-2$: ok!

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

Bilan

A retenir

- la méthode de conception : écrire en supposant que les appels $A(n')$ fonctionnent pour n' plus petit
- 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

Whiteboard

Whiteboard