

Compléments de Programmation

Benoit Donnet
Année Académique 2024 - 2025



Agenda

- Chapitre 1: Raisonnement Mathématique
- Chapitre 2: Construction de Programme
- Chapitre 3: Introduction à la Complexité
- **Chapitre 4: Récursivité**
- Chapitre 5: Types Abstraits de Données
- Chapitre 6: Listes
- Chapitre 7: Piles
- Chapitre 8: Files
- Chapitre 9: Elimination de la Récursivité

Agenda

- Chapitre 4: Récursivité
 - Principe
 - Algorithme Récursif
 - Types de Récursivité
 - Construction Recursive
 - Complexité
 - Contexte
 - Synthèse

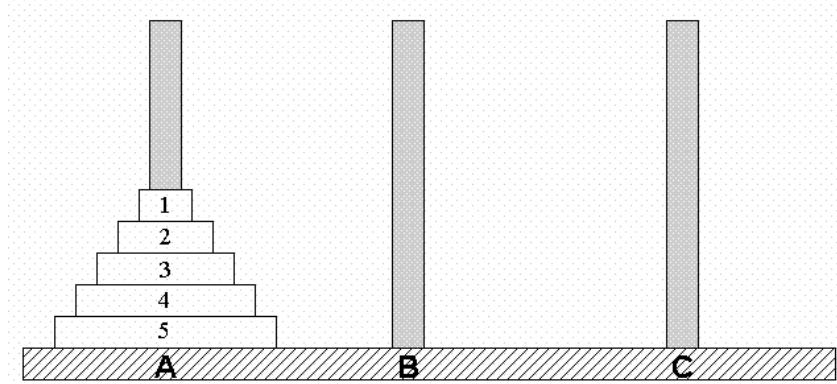
Agenda

- Chapitre 4: Récursivité
 - Principe
 - ✓ Généralités
 - ✓ Exemples
 - Algorithme Récursif
 - Types de Récursivité
 - Construction Recursive
 - Complexité
 - Contexte
 - Synthèse

Généralités

- En programmation, une majorité de problèmes sont résolus par répétition de tâches
- Certains langages (comme le C) sont munis de structures itératives
 - `for`
 - `while`
 - `do ... while`
- Cependant, certains problèmes se résolvent simplement en solutionnant des problèmes identiques
 - récursivité

Exemples



- Tours de Hanoï
 1. déplacer un disque à la fois d'une tour à l'autre
 2. ne jamais mettre un disque sur un plus petit
- But?
 - transférer la pile de disques de A vers B

Exemples (2)

- Les tours de Hanoï: une solution
 1. mettre tous les disques, sauf le plus grand, sur C
 2. déplacer le plus grand disque de A vers B
 3. mettre tous les disques de C sur B
- Les points 1 & 3 de la solution sont des problèmes de Hanoï avec un disque de moins
 - si on sait résoudre le problème avec $n-1$ disques, alors on sait le résoudre avec n disques
- Or, on sait résoudre le problème avec 1 disque
 - le problème est résolu pour tout nombre $n \geq 1$ de disques
 - ✓ principe de récurrence
 - la solution est *récursive*

Exemples (3)

- Calcul de dérivées
 - règles de dérivation
 - ✓ $(u + v)' = u' + v'$
 - ✓ $(u - v)' = u' - v'$
 - ✓ $(u \times v)' = u' \times v + u \times v'$
 - ✓ ...
- Pour dériver, il faut savoir dériver!
- Or, on sait dériver une fonction de base
 - on sait donc dériver toutes les fonctions (dérivables, bien entendu)
- Le calcul est *récursif*

Exemples (4)

- Calcul de factorielle
 - soit à calculer $1 \times 2 \times 3 \times \dots \times (n-1) \times n$
 - on sait que pour $n \geq 0$, $n! = n \times (n-1)!$
 - ✓ si on sait calculer $(n-1)!$, alors on sait calculer $n!$
 - on sait calculer $0! = 1$
 - ✓ on sait calculer $n! \forall n \geq 0$
- Le calcul est *récursif*

Exemples (5)

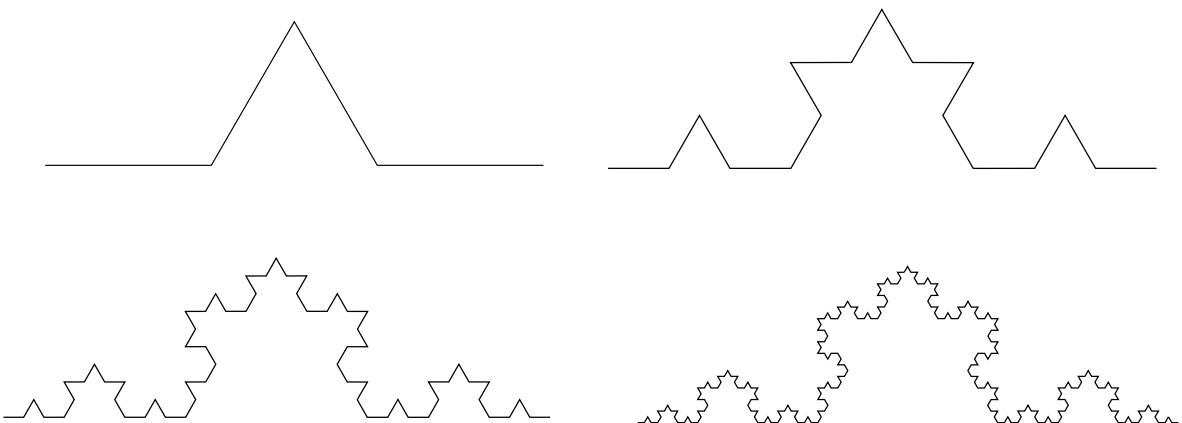
- Joseph McCarthy
 - Turing Award (1971)
 - inventeur du LISP
 - inventeur de l'expression "Intelligence Artificielle"
- McCarthy 91
 - $M(n) = \begin{cases} n - 10, & \text{si } n > 100 \\ M(M(n+1)), & \text{si } n \leq 100 \end{cases}$
- Propriétés intéressantes
 - $\forall n \leq 101, M(n) = 91$
 - $\forall n > 101, M(n) = n - 10$

Exemples (6)

- Fonction de Syracuse
 - Syracuse(n):
Si ($n == 0 \parallel n == 1$)
Alors 1
Sinon Si ($n \% 2 == 0$)
 Alors Syracuse($n/2$)
 Sinon Syracuse($3 \times n + 1$)
- Est-ce que cette fonction termine toujours?
 - difficile à dire, la suite n'étant pas monotone
- Conjecture de Collatz
 - $\forall n \in \mathbb{N}$, Syracuse(n) = 1
 - vérifié sur ordinateur $\forall n < 19 \times 2^{58} \sim 5 \times 10^{48}$
- Problème ouvert depuis 1937

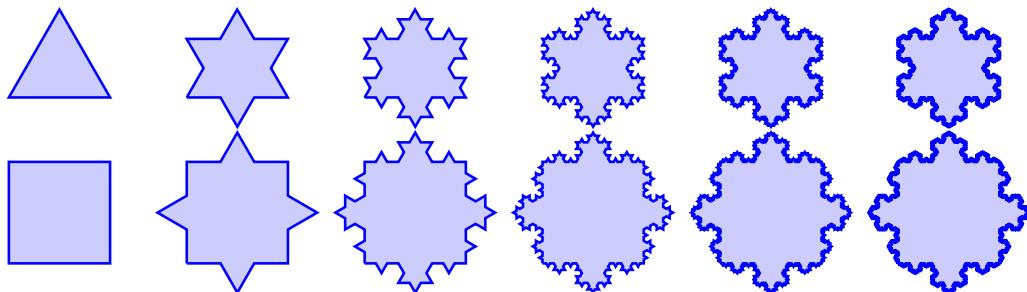
Exemples (7)

- Les fractales
 - néologisme créé par Mandelbrot en 1974
 - *fractus*
 - ✓ brisé, irrégulier
- Construction



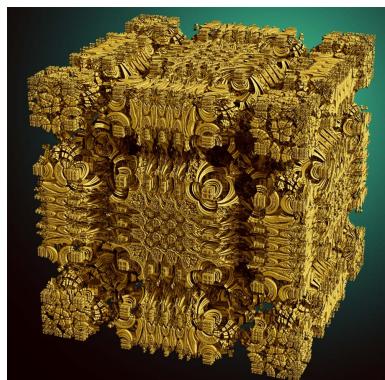
Exemples (8)

- Construction des fractales (suite)
 1. on définit ce qu'est la fractale de niveau 1
 2. la fractale de niveau $n+1$ s'obtient en appliquant la fractale de niveau 1 à chaque "segment" de la fractale de niveau n
- Définition récursive
 - une fractale est une ligne polygonale "fractalisée"



Exemples (9)

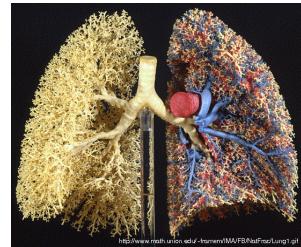
- Application des fractales
 1. populariser les maths par leur esthétique



(c)wikipedia

Exemples (10)

- Application des fractales (suite)
 2. générer des structures pour des applications graphiques



(c)wikipedia

Exemples (11)

- Application des fractales (suite)
 3. les fractales existent dans la nature



(c)wikipedia

Exemples (12)

- Certains acronymes sont récursifs
 - GNU
 - ✓ **Gnu is Not Unix**
 - PHP
 - ✓ **PHP: Hypertext Preprocessor**
 - PNG
 - ✓ **PNG's Not Gif**
 - Wine
 - ✓ **Wine Is Not an Emulator**
 - VISA
 - ✓ **Visa International Service Association**

Exemples (13)

- Certains objets informatiques sont eux-mêmes récursifs
 - liste chaînée
 - ✓ cfr. Chapitre 6
 - arbre binaire
 - ✓ cfr. INFO0902

Agenda

- Chapitre 4: Récursivité
 - Principe
 - Algorithme Récursif
 - ✓ Définition
 - ✓ Règles de Conception
 - ✓ Exemples
 - ✓ Exécution d'un Algorithme Récursif
 - Types de Récursivité
 - Construction Récursive
 - Complexité
 - Contexte
 - Synthèse

Définition

- Un algorithme de résolution d'un problème P sur une donnée a est dit **récursif** si parmi les opérations utilisées pour le résoudre, on trouve une résolution du même problème P sur une donnée b
- Dans un algorithme récursif, on nomme **appel récursif** toute étape de l'algorithme résolvant le même problème sur une autre donnée
- Une fonction/procédure est récursive
 - si son exécution peut conduire à sa propre invocation

Définition (2)

- Une fonction/procédure est donc récursive si son exécution peut conduire à sa propre invocation
- Une fonction/procédure récursive se présente donc comme suit

```
type_retour f(P){  
    //instructions  
  
    x = f(P');  
    //instructions  
  
    return r;  
} //fin f()
```

P = liste de paramètres

appel avec d'autres paramètres

renvoi non obligatoire

Conception

- Attention
 - il existe des algorithmes récursifs qui ne produisent aucun résultat

```
int f(int n){  
    return n * f(n-1);  
} //fin fact()
```

- Trace d'exécution

f(1)
1 × f(0)
1 × 0 × f(-1)
1 × 0 × -1 × f(-2)
...

Conception (2)

- 1ère règle
 - tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif
 - ✓ sinon risque de cercle vicieux et de calcul infini
- Les cas non-récurifs d'un algorithme récursif sont appelés **cas de base**
- Les conditions que doivent satisfaire les données dans ces cas de base sont appelées **conditions de terminaison**

Conception (3)

- Une fonction/procédure récursive avec cas de base devra donc avoir la forme suivante

```
type retour f(P){  
    if(test(P))  
        return un_resultat;          test du cas de base  
  
    x = f(P');  
  
    //instructions  
  
    return r;  
} //fin f()
```

Conception (4)

- Attention
 - même avec un cas de base, un algorithme récursif peut ne produire aucun résultat

```
int f(int n){  
    if(n==0)  
        return 1;  
    else  
        return f(n+1)/(n+1);  
} //fin f()
```

- Trace d'exécution

f(1)
f(2) / 2
f(3) / 3
...

Conception (5)

- 2ème règle
 - tout appel récursif doit se faire avec des données plus "proches" de données satisfaisant une condition de terminaison
- Théorème
 - il n'existe pas de suite infinie strictement décroissante d'entiers positifs ou nuls
 - ✓ permet de contrôler l'arrêt d'un calcul suivant un appel récursif

Conception (6)

- En résumé, le canevas d'une fonction/procédure récursive correcte se ramène à

```
type_retour f(P){  
    if(condition_de_terminaison)  
        //bloc sans appel à f  
    else  
        //bloc avec appel(s) de f(P'), où P' est "plus simple" que P  
    } //fin f()
```

- "Plus simple" varie selon les contextes
 - si P est un tableau, alors P' est un tableau plus petit
 - si $P \in \mathbb{N}$, alors $P' < P$
 - ...

Exemples

- Factorielle
 - $fact(n)$ = problème du calcul de $n!$

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

Exemples (2)

- Le nombre de Fibonacci

- $fib: \mathbb{N} \rightarrow \mathbb{N}$
 - ✓ $fib(0) = 0$
 - ✓ $fib(1) = 1$
 - ✓ $fib(n) = fib(n-1) + fib(n-2), n \geq 2$

- Code

```
int fib(int n){  
    if(n<=1)  
        return n;  
    else  
        return fib(n-1) + fib(n-2);  
} //fin fib()
```

Exemples (3)

- La tour de Hanoï

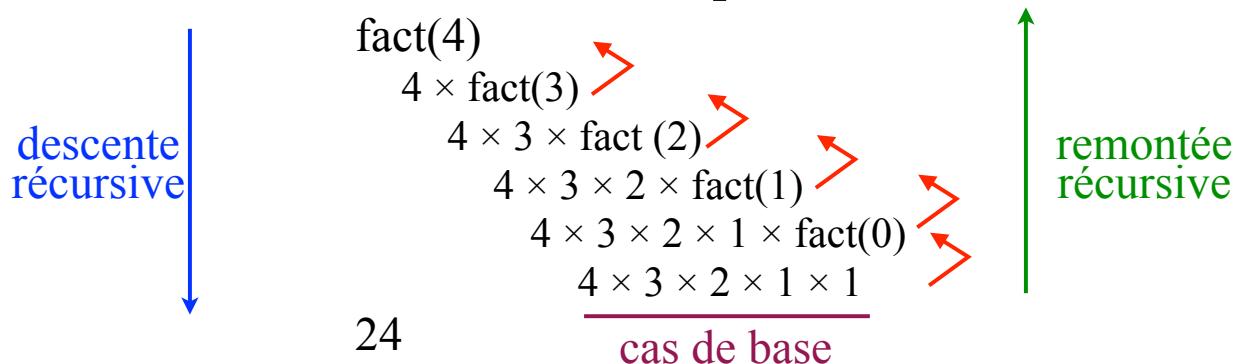
- $hanoi(n, d, a, i) =$ problème de déplacement de n disques depuis la tour d vers la tour a avec la tour intermédiaire i

```
void hanoi(int n, tour *d, tour *a, tour *i){  
    if(n==1)  
        deplacer_disque(d, a);  
    else{  
        hanoi(n-1, d, i, a);  
        deplace_disque(d, a);  
        hanoi(n-1, i, a, d);  
    }  
} //fin hanoi()
```

Exécution

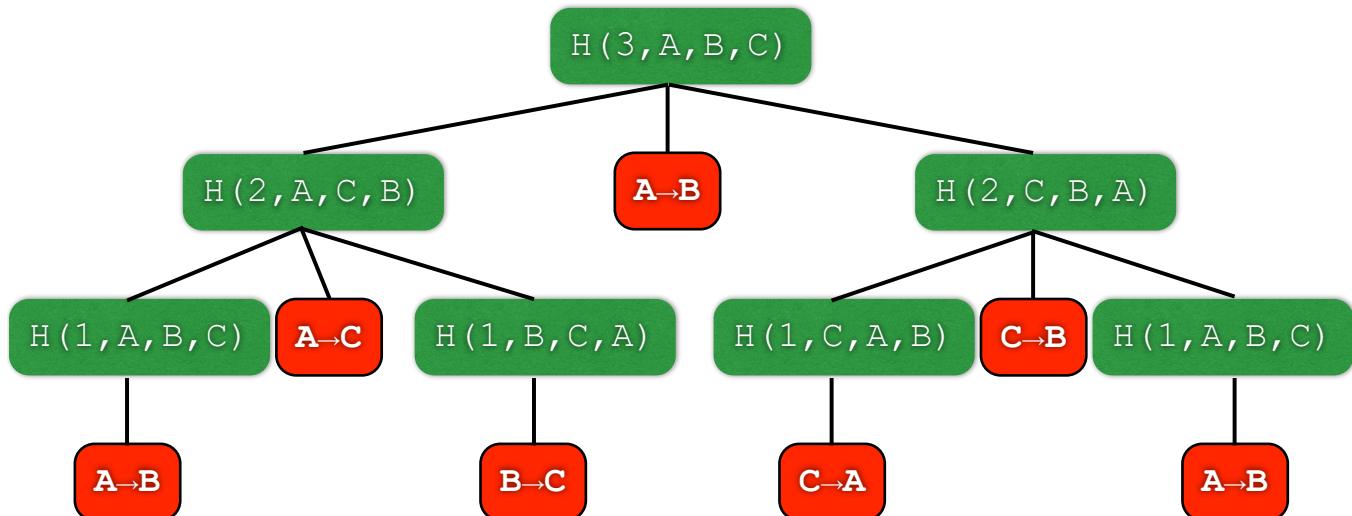
```
int fact(int n){  
    if(n==0)  
        return 1;  
    else  
        return n * fact(n-1);  
}//fin fact()
```

- Trace d'exécution pour `fact(4)`



Exécution (2)

- Tours de Hanoï pour déplacer 3 disques de la tour A à la tour B



Agenda

- Chapitre 4: Récursivité
 - Principe
 - Algorithme Récursif
 - Types de Récursivité
 - ✓ Récursivité Simple
 - ✓ Récursivité Multiple
 - ✓ Récursivité Croisée
 - ✓ Récursivité Terminale
 - ✓ Récursivité Imbriquée
 - Construction Récursive
 - Complexité
 - Contexte
 - Synthèse

Récursivité Simple

- Un algorithme récursif est simple ou linéaire si chaque cas se résout en au plus un appel récursif
- L'algorithme de calcul de $n!$ est récursif simple

```
int factorielle(int n){  
    if(n==0)  
        return 1;  
    else  
        return n * factorielle(n-1);  
} //fin factorielle()
```

Récursivité Multiple

- Un algorithme récursif est **multiple** si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs
 - si deux appels récursifs, on parle de **récursivité binaire**
- L'algorithme des tours de Hanoï est récursif binaire

```
void hanoi(int n, tour *d, tour *a, tour *i){  
    if(n==1)  
        deplacer_disque(d, a);  
    else{  
        hanoi(n-1, d, i, a);  
        deplace_disque(d, a);  
        hanoi(n-1, i, a, d);  
    }  
} //fin hanoi()
```

Récursivité Croisée

- Deux algorithmes sont **mutuellement** récursifs si l'un fait appel à l'autre et l'autre fait appel à l'un
- On parle aussi de récursivité **croisée**
- Exemple
 - parité d'un entier
 - ✓ $P(n)$ = prédicat de test de parité de l'entier n
 - ✓ $I(n)$ = prédicat de test d'imparité de l'entier n

Récursivité Croisée (2)

```
int I(int n);

/*
 * @pre: n ≥ 0
 * @post: 1 si n est pair
 *         0 sinon
 */
int P(int n){
    if(n==0)
        return 1;
    else
        return I(n-1);
}//fin P()

/*
 * @pre: n ≥ 0
 * @post: 1 si n est impair
 *         0 sinon
 */
int I(int n){
    if(n==0)
        return 0;
    else
        return P(n-1);
}//fin P()
```

Récursivité Croisée (3)

- Evaluation de $P(2)$
 - $P(2) \Rightarrow I(1) \Rightarrow P(0) \Rightarrow \text{True}$
- Evaluation de $P(3)$
 - $P(3) \Rightarrow I(2) \Rightarrow P(1) \Rightarrow I(0) \Rightarrow \text{False}$

Récursivité Terminale

- Un algorithme est **récursif terminal** si l'appel récursif est la dernière instruction de la fonction
 - `return f(...);`
- La valeur renvoyée est directement obtenue par un appel récursif
 - aucune opération sur cette valeur
 - rien à retenir sur la pile

```
int f(int n, int a){  
    if(n==0)  
        return a;  
    else  
        return f(n-1, n*a);  
} //fin f()
```

Récursivité Imbriquée

- Un algorithme est **récursif imbriqué** si l'appel récursif contient lui aussi un appel récursif
 - `f(..., f(...), ...);`

```
int ackermann(int m, int n){  
    if(m==0)  
        return n+1;  
    else{  
        if(n==0)  
            return ackermann(m-1, 1);  
        else  
            return ackermann(m-1, ackermann(m, n-1));  
    }  
} //fin ackermann()
```

Récursivité Imbriquée (2)

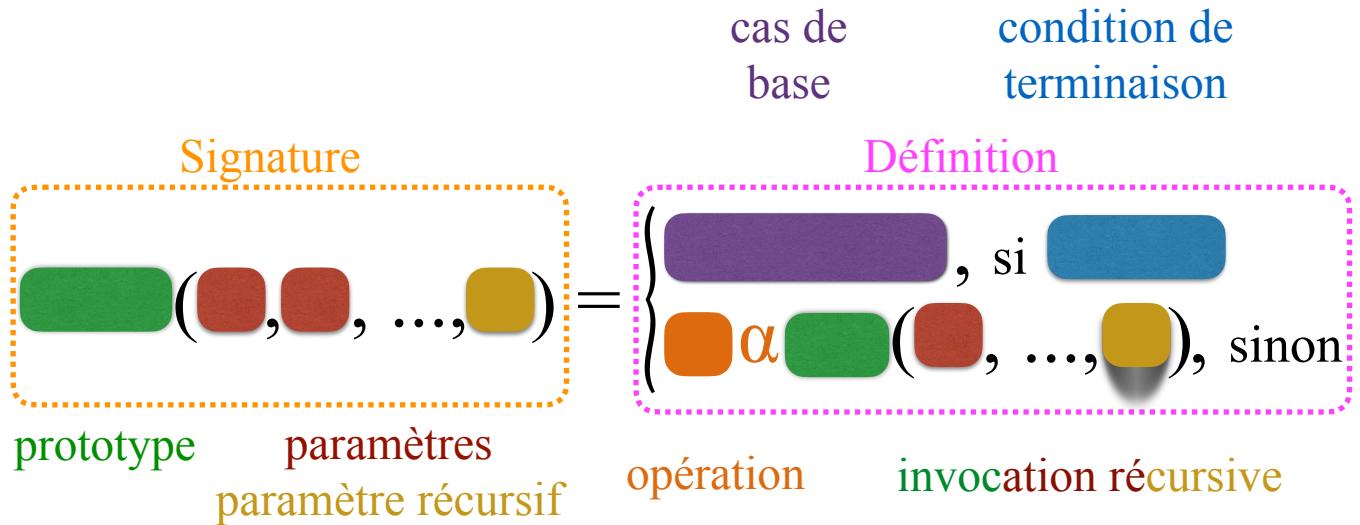
- Attention, la fonction d'Ackermann croît exponentiellement
 - $\text{Ack}(1, n) = n + 2$
 - $\text{Ack}(2, n) = 2 \times n + 3$
 - $\text{Ack}(3, n) = 8 \times 2^n - 3$
 - $\text{Ack}(4, n) = 2^{2^{2^{\dots^2}}} \text{ n fois}$
 - $\text{Ack}(4, 4) > 2^{65536} > 10^{80}$
 - ✓ quantité estimée de particules dans l'Univers

Agenda

- Chapitre 4: Récursivité
 - Principe
 - Algorithme Récursif
 - Types de Récursivité
 - Construction Récursive
 - ✓ Formulation Récursive
 - ✓ Approche Constructive
 - ✓ Exemple 1: Puissance
 - ✓ Exemple 2: Régionnement du Plan
 - ✓ Exemple 3: Recherche du Maximum
 - Complexité
 - Contexte
 - Synthèse

Formulation Récursive

- Forme générale d'une formulation (mathématique) récursive



Formulation Récursive (2)

- Comment construire la formulation récursive?
- Trois questions à se poser:
 1. est-ce que le problème dépend **d'un (ou plusieurs) paramètres?**
 - ✓ peut-on trouver un **paramètre récursif?**
 2. **peut-on résoudre le problème lorsque la (les) valeur(s) du paramètre est "petite(s)"?**
 - ✓ peut-on trouver une **condition de terminaison?**
 - ✓ peut-on trouver un **cas de base?**
 3. peut-on résoudre le problème à l'aide de **la résolution du problème partant sur une (des) "plus petite(s)" valeur(s) du (des) paramètre(s)**
 - ✓ peut-on exprimer récursivement le problème?

Généralisation

- Comment faire quand on ne manipule pas des valeurs naturelles?
- Méthode
 1. choisir une variable dans la postcondition
 - ✓ joue le rôle de *paramètre d'induction*
 - ✓ soit E , son *domaine de variation*
 2. définir une *relation bien fondée* $<$ sur E
 - ✓ permet de comparer les éléments de E deux à deux
 - ✓ permet de déterminer un *élément minimal*
 - ✓ souvent, pour une structure de données, la relation $<$ est définie sur sa taille
 3. construire une solution pour le cas où le paramètre d'induction prend une valeur minimale de E
 - ✓ *condition de terminaison* et *cas de base*
 4. construire une solution pour *le cas général*, i.e., le paramètre d'induction prend une valeur non-minimale

Approche Constructive

- Le fait d'avoir un algorithme récursif n'empêche pas d'utiliser l'approche constructive!
- Il faut donc continuer à construire son code en
 - fournissant une spécification
 - ✓ si possible, la post-condition doit être exprimée de manière récursive
 - en appliquant le triplet {P} Code {Q}
 - ✓ typiquement, on appliquera une structure conditionnelle
 - ✓ cfr. Chap. 2
 - en indiquant les différentes assertions intermédiaires
- En pratique
 - la précondition doit être vraie avant l'appel récursif
 - ✓ Précond_{REC}
 - la postcondition doit être vraie après l'appel récursif
 - ✓ Postcond_{REC}

Approche Constructive (2)

- Processus de construction du code
 - 1. formuler récursivement le problème
 - ✓ prototype pertinent
 - ✓ noms des paramètres pertinents
 - 2. interface de la fonction
 - ✓ prototype de la fonction différent de celui de la formulation mathématique (pour éviter toute confusion)
 - ✓ les paramètres formels de la fonction sont ceux de la formulation mathématique
 - contraintes dans la précondition
 - ✓ la postcondition s'exprime à l'aide de la formulation, en fonction des inputs
 - 3. construire le code en s'appuyant sur 1. et en appliquant l'approche constructive

Approche Constructive (3)

- Lien formulation mathématique -- code C
 - récursivité simple non terminale
 - cfr. Slide 43 pour le code couleur

```
type_retour [green]([red], [red], ..., [yellow]) {  
  
    if ([blue])  
        return [purple];  
  
    else  
  
        return [orange] α [green]([red], [red], ..., [yellow]);  
}
```

Puissance

- On désire avoir une solution permettant de calculer a^x , avec $x \geq 0$

1. Formulation récursive

- Trois questions
 - ✓ peut-on trouver un paramètre récursif?
 - › x
 - ✓ peut-on résoudre le problème pour la plus petite valeur de x ?
 - › condition de terminaison: $x = 0$
 - › cas de base: 1
 - ✓ peut-on exprimer le problème de manière récursive?
 - › $\forall x > 0$, on a $a^x = a \times a^{x-1}$

Puissance (2)

1. Formulation récursive (cont')

- il vient

$$power(a, x) = \begin{cases} 1 & , \text{ si } x = 0 \\ a \times power(a, x - 1) & , \text{ sinon} \end{cases}$$

Puissance (3)

2. Interface

- prototype + spécifications formelles

```
/*
 * @pre: a ≠ 0 ∧ x ≥ 0
 * @post: a = a₀ ∧ x = x₀ ∧ puissance = power(a, x)
 */
int puissance(int a, int x);
```

Puissance (4)

3. Construction du code

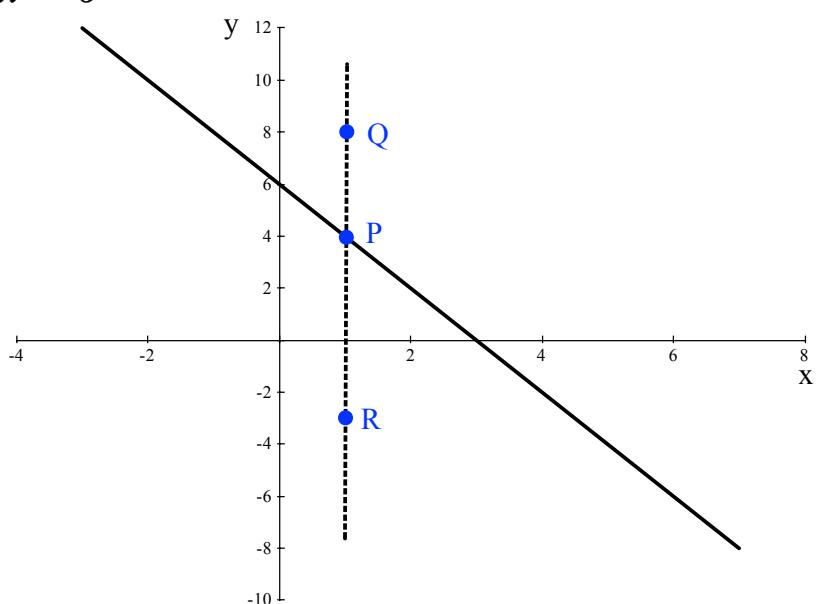
```
int puissance(int a, int x){
    {Pré ≡ a ≠ 0 ∧ x ≥ 0}
    if(x==0)
        {a ≠ 0 ∧ x = 0 ∧ x = x₀}
        return 1;
        {a = a₀ ∧ x = 0 ∧ x = x₀ ∧ puissance = a⁰}
        {a = a₀ ∧ x = x₀ ∧ puissance = power(a,x)}
    else
        {PrécondREC ≡ a ≠ 0 ∧ x > 0 ∧ x = x₀}
        return a * puissance(a, x-1);
        {PostcondREC ≡ a=a₀ ∧ x=x₀ ∧ puissance = power(a, x-1)}
        {a=a₀ ∧ puissance = a × power(a, x-1) ∧ x ≥ 0 ∧ x=x₀}
        {a = a₀ ∧ x = x₀ ∧ puissance = power(a, x)}
    {Post ≡ a = a₀ ∧ x = x₀ ∧ puissance = power(a, x)}
} //fin puissance()
```

Puissance (5)

```
int puissance(int a, int x){  
    if( x==0 )  
        return 1 ;  
  
    else  
        return a * puissance( a , x-1 );  
    } //fin puissance()
```

Réglonnemement du Plan

- Soit le graphique suivant
 - avec la droite D
 - d'équation $y = -2x + 6$



Régionnement du Plan (2)

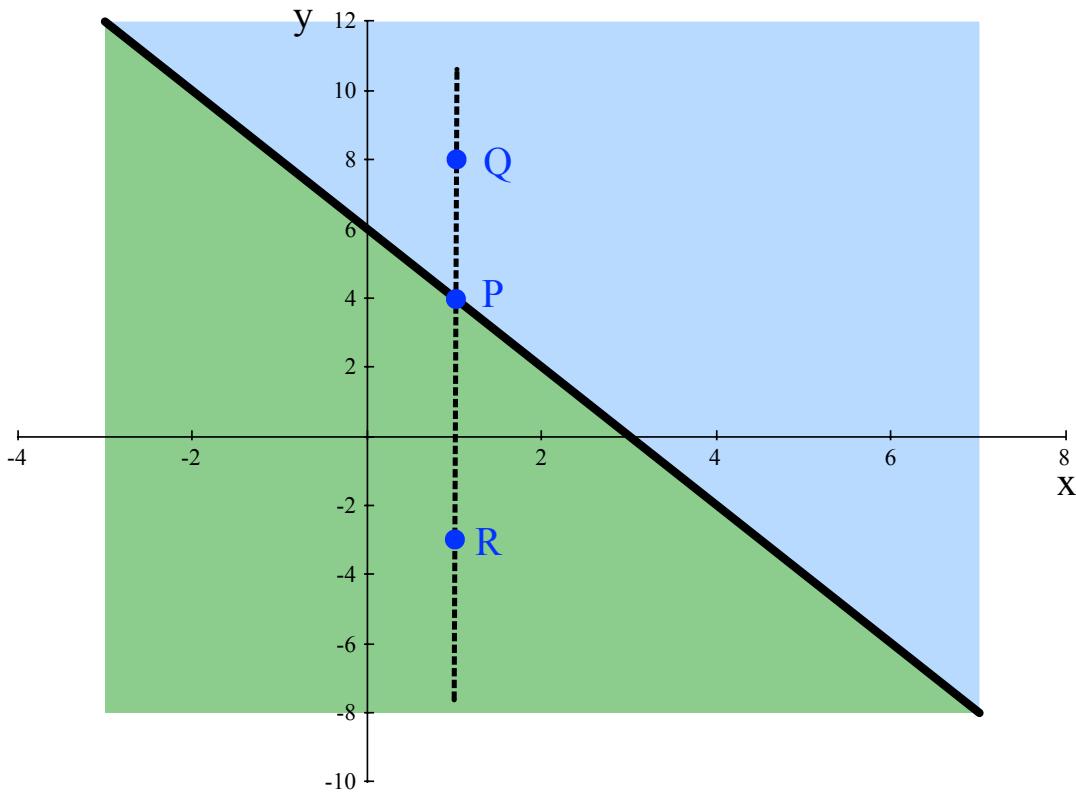
- Pour un point $P(x, y)$ situé sur D
 - $y = -2x + 6$
- Pour un point $M(x, y)$ non situé sur D
 - $y \neq -2x + 6$
- Prenons les points Q et R , de même abscisse que P
 - l'ordonnée de Q est supérieure à celle de P
 - ✓ ordonnée y_Q de Q est telle que $y_Q > -2x + 6$
 - l'ordonnée de R est inférieure à celle de P
 - ✓ ordonnée y_R de R est telle que $y_R < -2x + 6$

Régionnement du Plan (3)

- La droite D , d'équation $y = -2x+6$, délimite le plan en 3 régions

Type de Région	Relation entre x et y
points situés <u>sur</u> la droite	$y = -2x + 6$
points situés <u>sous</u> la droite	$y < -2x + 6$
points situés <u>au dessus</u> de la droite	$y > -2x + 6$

Régionnement du Plan (4)

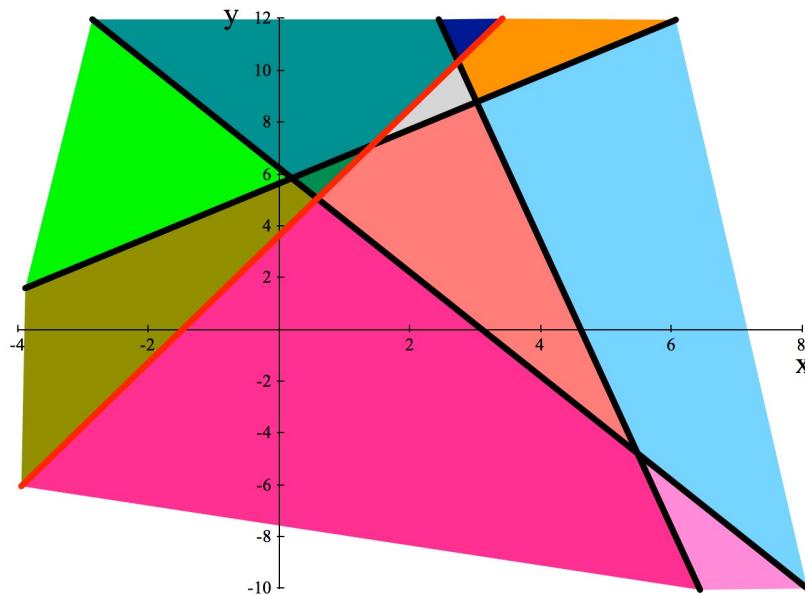


Régionnement du Plan (4)

- Problème
 - étant donné un nombre n de droites, calculer le nombre R_n maximum de régions du plan obtenus
1. Formulation récursive
 - peut-on trouver un paramètre récursif?
 - ✓ n , le nombre de droites
 - peut-on résoudre le problème pour la plus petite valeur de n ?
 - ✓ condition de terminaison: $n=0$
 - ✓ cas de base: 1
 - peut-on exprimer le problème de manière récursive?

Réglonnemment du Plan (5)

- Formulation récursive?



n	R _n
0	1
1	3
2	6
3	10
4	15

Réglonnemment du Plan (6)

- Formulation récursive (cont.)?

n	R _n		
0	1		
1	3	2 + 1	1 + 1 + 1
2	6	3 + 3	2 + 1 + 3
3	10	4 + 6	3 + 1 + 6
4	15	5 + 10	4 + 1 + 10

R_{n-1}

n

la droite qu'on ajoute

Régionnement du Plan (7)

1. Formulation récursive (cont')

- il vient

$$\text{regionnement}(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 + n + \text{regionnement}(n - 1) & \text{sinon} \end{cases}$$

Régionnement du Plan (8)

2. Interface

- prototype + spécifications formelles

```
/*
 * @pre: n ≥ 0
 * @post: n = n₀ ∧ region = regionnement(n)
 */
int region(int n);
```

Réglonnement du Plan (9)

3. Construction du code

```
int region(int n){  
    {Pré ≡ n ≥ 0}  
    if(n==0)  
        {n = 0 ∧ n = n₀}  
        return 1;  
        {n = 0 ∧ region = 1 ∧ n = n₀}  
        {n = n₀ ∧ region = regionnement(n)}  
    else  
        {n ≥ 0 ∧ n ≠ 0 ∧ n = n₀}  
        {n > 0 ⇒ PrécondREC}  
        return (1 + n) + region(n-1);  
        {PostcondREC ≡ n = n₀ ∧ region = regionnement(n-1)}  
        {n = n₀ ∧ region = (1 + n) + regionnement(n-1)}  
        {n = n₀ ∧ region = regionnement(n)}  
        {Post ≡ n = n₀ ∧ region = regionnement(n)}  
    } //fin region()
```

Réglonnement du Plan (10)

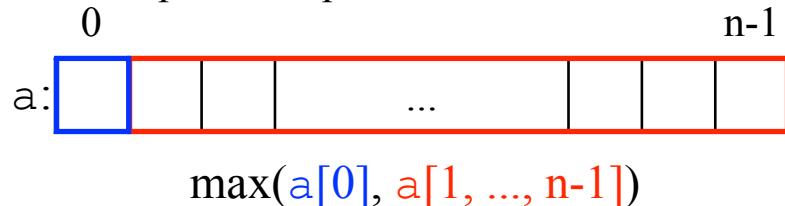
```
int region (int n) {  
    if(n==0)  
        return 1 ;  
  
    else  
  
        return (1 + n) + region (n-1);  
    } //fin region()
```

Maximum

- On désire un programme retournant le maximum d'un tableau a de n valeurs entières

1. Formulation récursive

- Trois questions
 - ✓ peut-on trouver un paramètre récursif?
 - › n , la taille du tableau
 - ✓ peut-on résoudre le problème pour la plus petite valeur de n ?
 - › condition de terminaison: $n=1$
 - › cas de base: $a[0]$
 - ✓ peut-on exprimer le problème de manière récursive?



Maximum (2)

1. Formulation récursive (cont')

- il vient

$$\max_tab(a, n) = \begin{cases} a[0] & \text{si } n = 1 \\ \max(a[0], \max_tab(a + 1, n - 1)) & \text{sinon} \end{cases}$$

Maximum (3)

2. Interface

- prototype + spécifications formelles

```
/*
 * @pre: a initialisé ∧ x > 0
 * @post: a = a0 ∧ x = x0 ∧ maximum = max_tab(a, n)
 */
int maximum(int *a, int n);
```

Maximum (4)

3. Construction du code

- On peut envisager un module
 - ✓ fonction qui retourne le maximum de deux valeurs entières

```
/*
 * @pré: /
 * @post: x = x0 ∧ y = y0 ∧ max = max(x, y)
 */
int max(int x, int y){
    if(x>y) return x;
    else return y;
}//fin max()
```

Maximum (5)

3. Construction du code (cont')

```
int maximum(int *a, int n){  
    {Pré ≡ n > 0}  
    if(n==1)  
        {n = 1 ∧ a = a0}  
    return a[0];  
    {n = 1 ∧ a = a0 ∧ maximum = a[0]}  
    {a = a0 ∧ n = n0 ∧ maximum = max_tab(a, n)}  
else  
    {n > 1 ∧ a = a0 ⇒ PrécondREC}  
    return max(a[0], maximum(a+1, n-1));  
    {PostcondREC ≡ a = a0 ∧ n = n0 ∧ maximum = max_tab(a+1,n-1)}  
    {n ≥ 0 ∧ n = n0 ∧ a = a0 ∧ maximum = max(a[0],  
                                                max_tab(a+1,n-1)}  
    {a = a0 ∧ n = n0 ∧ maximum = max_tab(a, n)}  
    {Post ≡ a = a0 ∧ n = n0 ∧ maximum = max_tab(a, n)}  
}//fin maximum()
```

Maximum (6)

```
int maximum(int *a, int n){  
    if(n==1)  
        return a[0];  
    else  
        return max(a[0], maximum(a+1, n-1));  
}//fin maximum()
```

Agenda

- Chapitre 4: Récursivité
 - Principe
 - Algorithme Récursif
 - Types de Récursivité
 - Construction Récursive
 - Complexité
 - ✓ Principe
 - ✓ Equations de Réurrence
 - ✓ Factorielle
 - ✓ Régionnement du Plan
 - ✓ Fibonacci
 - ✓ Tours de Hanoï
 - Contexte
 - Synthèse

Principe

- On a vu comment évaluer la complexité des programmes "classiques"
 - évaluer le nombre d'itérations
 - cfr. INFO0946, Chap. 5
- La situation ne change pas vraiment avec la récursivité
 - évaluer le nombre d'appels récursifs
- Mais cela implique, la plupart du temps, de résoudre une **équation de récurrence**

Principe (2)

- Etant donné une suite de nombres ($t(1), t(2), \dots, t(n), \dots$), une équation reliant le $n^{\text{ème}}$ terme à ses prédecesseurs est appelée **équation de récurrence**
 - on parle aussi d'**équation aux différences**
- La résolution d'une équation de récurrence consiste à trouver une expression du $n^{\text{ème}}$ en fonction du paramètre n
- Exemple
 - soit la suite géométrique (1, 2, 2^2 , ..., 2^n , ...)
 - on peut écrire
 - ✓ $t(0) = 1$
 - ✓ $t(n) = 2t(n-1)$, $n \geq 1$

Equation de Récurrence

- 3 approches pour résoudre une équation de récurrence
 1. éliminer la récurrence par substitution de proche en proche
 2. deviner une solution et la démontrer par récurrence
 - ✓ cfr. Chap. 1
 3. utiliser la solution de certaines équations connues

Equation de Récurrence (2)

- Solutions d'équations de récurrence

- $T(n) = T(n-1) + b$
 - ✓ solution: $T(n) = T(0) + b \times n$
 - ✓ complexité: $O(n)$
 - ✓ factorielle, recherche séquentielle récursive dans un tableau
- $T(n) = a \times T(n-1) + b, a \neq 1$
 - ✓ solution: $T(n) = a^n \times \left(T(0) - \frac{b}{1-a} \right) + \frac{b}{1-a}$
 - ✓ complexité: $O(a^n)$
 - ✓ répéter a fois le traitement sur un appel récursif
- $T(n) = T(n-1) + a \times n + b$
 - ✓ solution: $T(n) = T(0) + a \times n \times \frac{n+1}{2} + n \times b$
 - ✓ complexité: $O(n^2)$
 - ✓ traitement en coût linéaire avant l'appel récursif, bubble sort

Equation de Récurrence (3)

- Solutions d'équations de récurrence (cont.)

- $T(n) = T(n/2) + b$
 - ✓ solution: $T(n) = T(1) + b \times \log_2(n)$
 - ✓ complexité: $O(\log(n))$
 - ✓ élimination de la moitié des éléments en temps constant avant l'appel récursif, recherche dichotomique
- $T(n) = a \times T(n/2) + b, a \neq 1$
 - ✓ solution: $T(n) = n^{\log_2(a)} \times \left(T(1) - \frac{b}{1-a} \right) + \frac{b}{1-a}$
 - ✓ complexité: $O(n^{\log_2(a)})$
 - ✓ répétition n fois d'un traitement sur le résultat de l'appel récursif dichotomique

Factorielle

- Exemple 1: calcul de la factorielle
 - approche par substitution

```
int factorielle(int n){  
    if(n==1)          O(1)  
        return 1;      O(1)  
    else  
        return n * factorielle(n-1);  
} //fin factorielle()
```

a
b

dépend de factorielle(n-1)

- Soit $T(n)$, temps d'exécution nécessaire pour un appel à `factorielle(n)`
- On peut écrire
 - $T(n) = a$ si $n = 1$
 - $T(n) = b + T(n-1)$ sinon

Factorielle (2)

- Solution générale de l'équation

$$\begin{aligned} T(n) &= b + T(n-1) \\ &= b + [b + T(n-2)] \\ &= 2 \times b + T(n-2) \\ &= 2 \times b + [b + T(n-3)] \\ &= \dots \\ &= i \times b + T(n-i) \\ &= \dots \\ &= (n-1) \times b + T(n-(n-1)) \\ &= n \times b - b + T(1) \\ &= (n-1) \times b + a \end{aligned}$$

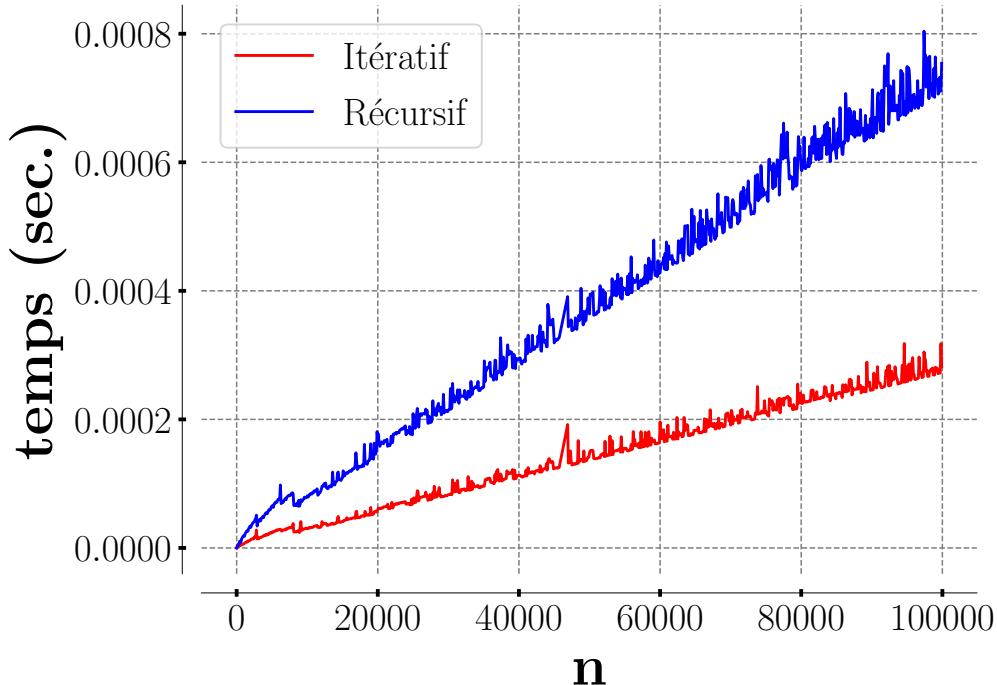
or $T(n-1) = b + T(n-2)$

or $T(n-2) = b + T(n-3)$

- Complexité
 - $O(n)$

Factorielle (3)

- Temps d'exécution pour factorielle



Réglonnement du Plan

- Exemple 2: régloonnement du plan

```
int region(int n){  
    if(n==0)  
        return 1;  
    else  
        return (1 + n) + region(n-1);  
} //fin region()
```

a
b

- Soit $T(n)$, temps d'exécution nécessaire pour un appel à `region(n)`
- On peut écrire
 - $T(n) = a$ si $n = 0$
 - $T(n) = b + T(n-1)$ sinon

Régionnement du Plan (2)

- Solution générale de l'équation

$$\begin{aligned} T(n) &= b + T(n-1) \\ &= b + [b + T(n-2)] \\ &= 2 \times b + T(n-2) \\ &= 2 \times b + [b + T(n-3)] \\ &= \dots \\ &= i \times b + T(n-i) \\ &= \dots \\ &= n \times b + T(n-n) \\ &= n \times b + T(0) \\ &= n \times b + a \end{aligned}$$

- Complexité

- $O(n)$

Fibonacci

- Exemple 4: le nombre de Fibonacci

- $fib: \mathbb{N} \rightarrow \mathbb{N}$
 - ✓ $fib(0) = 0$
 - ✓ $fib(1) = 1$
 - ✓ $fib(n) = fib(n-1) + fib(n-2), n \geq 2$

- Code

```
int fib(int n){  
    if(n<=1)  
        return n;  
    else  
        return fib(n-1) + fib(n-2);  
} //fin fib()
```

Fibonacci (2)

- Résultat
 - rapide pour $n = 5$
 - lent pour $n = 30$
- Opérations significatives?
 - le nombre d'appel de la fonction $fib()$
- Le nombre d'appels croît exponentiellement avec n
 - exemple: $fib(5)$
 - ✓ $fib(0) \Rightarrow 1$ appel
 - ✓ $fib(1) \Rightarrow 1$ appel
 - ✓ $fib(2) \Rightarrow 3$ appels
 - ✓ $fib(3) \Rightarrow 5$ appels
 - ✓ $fib(4) \Rightarrow 9$ appels
 - ✓ $fib(5) \Rightarrow 15$ appels

Fibonacci (3)

- Soit $Ap(fib(n))$, une fonction déterminant le nombre d'appels (récursifs) à $fib(n)$
- Définition inductive, $\forall n \in \mathbb{N}$

$$Ap(fib(n)) = \begin{cases} 1 & \text{si } n \leq 1 \\ 1 + Ap(fib(n-1)) + Ap(fib(n-2)) & \text{sinon} \end{cases}$$

- Comment évaluer $Ap(fib(n))$ (et donc la complexité de $fib(n)$)?
 - borner supérieurement et inférieurement $Ap(fib(n))$

Fibonacci (4)

- Borne supérieure

$$\begin{aligned} \text{- } \text{Ap}(fib(n)) &= 1 + \text{Ap}(fib(n-1)) + \text{Ap}(fib(n-2)) \\ &\leq 1 + 2 \times \text{Ap}(fib(n-1)) \\ &\leq 1 + 2 \times [1 + \text{Ap}(fib(n-2)) + \text{Ap}(fib(n-3))] \\ &\quad = 1 + 2 + 2 \times \text{Ap}(fib(n-2)) + 2 \times \text{Ap}(fib(n-3)) \\ &\leq 1 + 2 + (2 \times 2 \times \text{Ap}(fib(n-2))) \\ &\leq 1 + 2 + [2^2 \times (1 + \text{Ap}(fib(n-3)) + \text{Ap}(fib(n-4)))] \\ &\dots \\ &\leq 1 + 2 + 2^2 + 2^3 + \dots + 2^n \end{aligned}$$

- Formulation générale

$$\begin{aligned} \text{Ap}(fib(n)) &\leq \sum_{i=0}^n 2^i \\ &\leq 2^{n+1} - 1 \end{aligned}$$

Fibonacci (5)

- Borne inférieure

$$\begin{aligned} \text{- } \text{Ap}(fib(n)) &= 1 + \text{Ap}(fib(n-1)) + \text{Ap}(fib(n-2)) \\ &\geq 1 + 2 \times \text{Ap}(fib(n-2)) \\ &\geq 1 + 2 \times [1 + \text{Ap}(fib(n-3)) + \text{Ap}(fib(n-4))] \\ &\quad = 1 + 2 + 2 \times \text{Ap}(fib(n-3)) + 2 \times \text{Ap}(fib(n-4)) \\ &\geq 1 + 2 + (2 \times 2 \times \text{Ap}(fib(n-4))) \\ &\geq 1 + 2 + [2^2 \times (1 + \text{Ap}(fib(n-5)) + \text{Ap}(fib(n-6)))] \\ &\dots \\ &\geq 1 + 2 + 2^2 + 2^3 + \dots + 2^{n/2} \end{aligned}$$

- Formulation générale

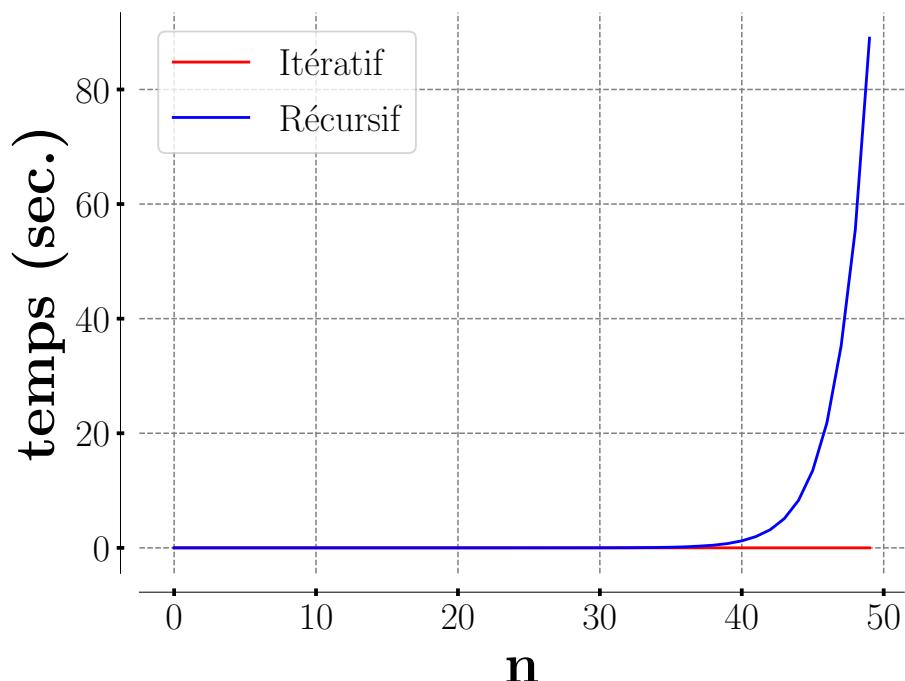
$$\begin{aligned} \text{Ap}(fib(n)) &\geq \sum_{i=0}^{n/2} 2^i \\ &\geq 2^{n/2+1} - 1 \end{aligned}$$

Fibonacci (6)

- Le nombre d'appels récursifs est donc
 - $2^{n/2 + 1} - 1 \leq A_p(fib(n)) \leq 2^{n+1} - 1$
- Complexité exponentielle
- La version itérative de Fibonacci est plus efficace mais
 - écriture non immédiate
 - moins élégante
 - demande plus de réflexion
- Il peut être parfois intéressant de transformer un algorithme récursif en algorithme itératif
 - cfr. Chap. 9

Fibonacci (7)

- Temps d'exécution



Tours de Hanoï

- Exemple 5: Tours de Hanoï

```
void hanoi(int n, tour *d, tour *a, tour *i){  
    if(n==1)  
        deplacer_disque(d, a);  
    else{  
        hanoi(n-1, d, i, a);  
        deplace_disque(d, a);  
        hanoi(n-1, i, a, d);  
    }  
}//fin hanoi()
```

Tours de Hanoï (2)

- Soit $T(n)$, le nombre de déplacements pour n disques
- On a
 - $T(n) = 1$ si $n=1$
 - $T(n) = T(n-1) + 1 + T(n-1)$ sinon
 - ✓ $T(n) = 2 \times T(n-1) + 1$
- Solution
 - $T(n) = 2^n - 1$

Agenda

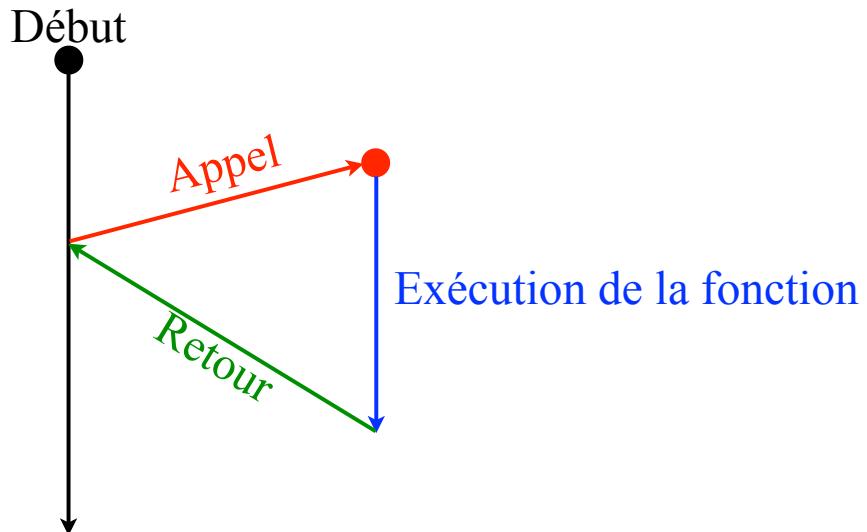
- Chapitre 4: Récursivité
 - Principe
 - Algorithme Récursif
 - Types de Récursivité
 - Construction Récursive
 - Complexité
 - Contexte
 - Synthèse

Contexte

- Il existe des coûts cachés liés à l'utilisation de la récursivité
- A chaque appel, il faut sauver le **contexte**
- Contexte?
 - le contexte d'une fonction est l'ensemble des variables (et leurs valeurs), sur la pile, qu'elle utilise et l'adresse de retour
- Conséquence?
 - on consomme des ressources supplémentaires!

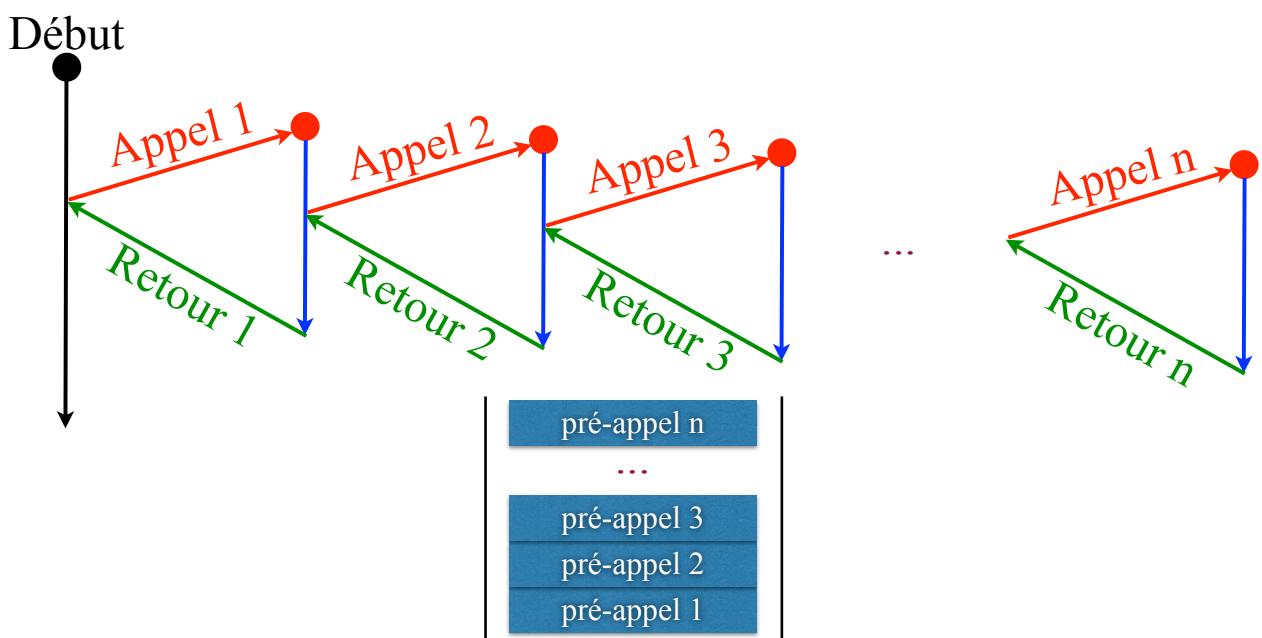
Contexte (2)

- Rappel
 - lorsqu'on exécute une fonction, on interrompt le flux du programme



Contexte (3)

- Ainsi, si une fonction s'appelle n fois, on va devoir sauvegarder $n-1$ contextes



Agenda

- Chapitre 4: Récursivité
 - Principe
 - Algorithme Récursif
 - Types de Récursivité
 - Construction Récursive
 - Complexité
 - Contexte
 - Synthèse

Synthèse

- La récursivité est un moyen naturel de résolution de certains problèmes
- Tout algorithme peut s'exprimer de manière récursive
- Mais on ne se lance pas tête baissée dans l'écriture d'une fonction/procédure récursive
 1. on cherche une définition récursive
 2. on s'assure que la définition comporte une condition de terminaison (i.e., cas de base)
 3. on écrit le code correspondant

Synthèse (2)

- Avantages

- simplifie la vie quand on a compris le truc
- beaucoup d'algorithme sont basés sur la récursivité
 - ✓ *décomposition*
 - décomposer une action répétitive en sous-actions "identiques" de petites tailles
 - *divide and conquer*
 - ✓ *exploration*
 - explorer un ensemble de possibilités
 - 1 possibilité = 1 appel récursif
 - *branch and bound*
 - *backtracking*
 - ✓ cfr. INFO0902 et INFO0027
- code souvent
 - ✓ plus court
 - ✓ plus élégant
 - ✓ plus lisible

Synthèse (3)

- Inconvénients

- plus compliqué à analyser
- tous les langages ne sont pas récursifs
 - ✓ COBOL, Fortran, Basic
- souvent moins efficace qu'un code itératif
 - ✓ **dérécursification**
 - ✓ cfr. Chap. 9