جامعة بجاية
Tasdawit n Bgayet
Université de Béjaïa

Université Abderrahmane Mira de Béjaïa

Algorithmique Avancée

A. BELAÏD
Recherche Opérationnelle
Master 1 – 2ième semestre 2014

Bejaia.etudiant@gmail.com

Plateforme d'enseignement à distance

- http://elearning.univ-bejaia.dz/
- Mot de passe **ALGO2015**
- Vous trouverez : le cours avec ses deux versions polycopié et PowerPoint, énoncés de TD et TP + projet, Biblio, Forum…

Références

- Les meilleurs références se trouvent sur internet! Voir également sur le site E-learning
- Bibliothèque de notre Université !
  - Initiation à l'algorithmique et à la programmation en C. Cours avec 129 exercices corrigés. 2ième édition DUNOD
  - Initiation à l'algorithmique et aux structure de données. Volume 2 DUNOD

Objectifs pédagogiques

- À l'issue de ce cours, vous devriez être capables de :

✓ **Concevoir des algorithmes**
✓ **Démontrer et analyser leur efficacité**
✓ **Implémenter diverses structures de données classiques pour les manipuler**

Plan du cours

- **Introduction générale**
- **Complexité et optimalité**
- **La récursivité et le paradigme « diviser pour régner »**
- **Algorithmes de tri**
- **Structures de données élémentaires**
- **Programmation dynamique**
- **Algorithmes gloutons**
- **Graphes et arbres**
- **Arbres de recherche et arbres de recherche équilibrés**
- **NP-complétude**
- **Heuristiques**

Introduction générale

- **C'est quoi un algorithme?**
  - Une succession d'instructions
  - qui renvoie un résultat
  - exacte
  - en un nombre fini d'étapes
- **C'est quoi l'algorithmique?**
  - l'algorithmique est l'étude des algorithmes.
- **Attention : Un algorithme n'est pas un programme**

Temps d'exécution

- Rappel : on n'exige pas seulement d'un algorithme qu'il résolve un problème ; On veut également qu'il soit efficace :
  - rapide (en termes de temps d'exécution) ;
  - peu gourmand en ressources (espace de stockage, mémoire utilisée) ;
- On a donc besoin d'outils qui nous permettront d'évaluer la qualité théorique des algorithmes proposes ;
- On se focalisera dans un premier temps sur le temps d'exécution ;

Temps d'exécution

- On ne mesure pas la durée en heures, minutes, secondes, ... :
  1. cela impliquerait d'implémenter les algorithmes qu'on veut comparer ;
  2. de plus, ces mesures ne seraient pas pertinentes car le même algorithme sera plus rapide sur une machine plus puissante ;
- On utilise des unités de temps abstraites proportionnelles au nombre d'opérations effectuées
- Chaque instruction basique consomme une unité de temps (affectation d'une variable, comparaison, +, -,*, /, ...)

Temps d'exécution

- L'algorithme suivant calcule $n! = n * (n-1) * (n-2) * \cdots * 2 * 1$, avec $0! = 1$.

```
fonction factorielle(n)
    fact ← 1 ;                    initialisation : 1
    i ← 2 ;                       initialisation : 1
    tant que (i <= n)faire        itération : au plus n-1
        fact ← fact * i ;         multiplication + affectation : 2
        i ← i + 1 ;               addition + affectation : 2
    fin tant que
renvoyer fact;                    renvoi d'une valeur : 1
```

- Avec le test à faire à chaque itération, le nombre total d'opérations élémentaires est :
- $T(n) = 1 + 1 + (n-1) * 5 + 1 + 1 = 5n - 1$
- Cette algorithme consomme un temps linéaire

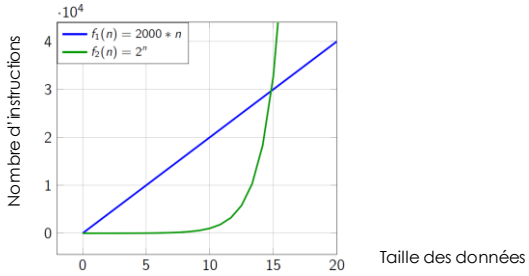
Complexité algorithmique

- La complexité d'un algorithme est une mesure de sa performance **asymptotique** dans le **pire cas** ;
  - Asymptotique : On s'intéresse a des données très grandes
    - les petites valeurs ne sont pas assez informatives ;
  - Pire cas : On s'intéresse a la performance de l'algorithme dans les situations ou le problème prend le plus de temps a résoudre ;
    - on veut être sûr que l'algorithme ne prendra jamais plus de temps que ce qu'on a estimé ;

Complexité algorithmique

Comportement asymptotique

- Soit deux algorithmes résolvant un problème (de taille de données n) en un temps f1(n) et f2(n), respectivement ;



- Quel algorithme préférez-vous ?
- La courbe verte semble correspondre a un algorithme beaucoup plus efficace ...
- ... mais seulement pour de très petites valeurs !

Complexité algorithmique

Notations de Landau

- Les calculs à effectuer pour évaluer le temps d'exécution d'un algorithme peuvent parfois être longs et pénibles ;
- De plus, le degré de précision qu'ils requièrent est souvent inutile ;
- On aura donc recours a une approximation de ce temps de calcul, représentée par la notation ***O(.)*** « grand O », ou « de l'ordre de »

Complexité algorithmique

Notations de Landau

- Soit n la taille des données a traiter; on dit qu'une fonction f(n) est en O(g(n)) (« en grand O de g(n) ») si :

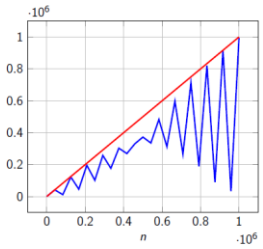
$$\exists c \in \mathbb{R}^{+*}, \exists n_0 \, , tels\ que\ \forall n > n_0, f(n) \leq c * g(n)$$

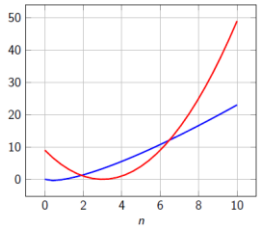
- On note $f = O(g)\ ou\ f \in O(g)$
- Autrement dit : f (n) est en O(g(n)) s'il existe un seuil a partir duquel la fonction f(.) est toujours dominée par la fonction g(.), a une constante multiplicative près

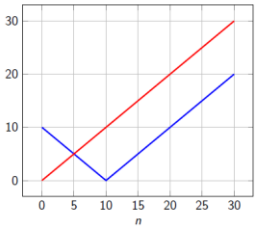
Complexité algorithmique

Notations de Landau

- Exemple : quelques cas ou **f(n)** = O(**g(n)**)







Complexité algorithmique

Notations de Landau

Exemple :

Prouvons que la fonction $f_1(n) = 5n + 37$est en $O(n)$:

Trouver une constante quelconque c>0 et un seuil n_0 à partir duquel $f_1(n) \leq c\, n$.

On en déduit que c=6 fonctionne à partir du seuil n_0=37.

Remarque : c=10 et n_0=8 est aussi acceptable

Prouvons que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $O(n^2)$

c=7 et $n_0 = 1$ nous donne le résultat voulu

Complexité algorithmique

Notations de Landau : Simplification sue le calcul du temps d'exécution

- On préfère avoir une idée du temps d'exécution de l'algorithme plutôt qu'une expression plus précise mais inutilement compliquée ;
- Retenir que les termes dominants et mettre à 1 les constantes multiplicatives,
- Exemple : $g(n) = 4n^3 - 5n^2 + 2n + 3$
  - Retenir le terme de plus haut degré : $\boldsymbol{4n^3}$
  - Mettre à un les coefficients : $\boldsymbol{n^3}$
  - Nous avons donc $\boldsymbol{g(n) = O(n^3)}$

Complexité algorithmique

Analyse du temps d'exécution d'un algorithme

- Opération élémentaire => temps constant: O(1)

- Bloc d'instructions : règle de sommation
  - **L'instruction Si** :
  - si $O(f_{si})$ *et* $O(f_{sinon})$ sont des bornes supérieurs des blocs « si » et « sinon » alors $O(\max\{f_{si}(n), f_{sinon}(n)\})$ est une borne supérieur du temps d'exécution de l'instruction « si »

  - **L'instruction : Pour, Tant que, repeter**
  - Si O(f(n)) est une borne supérieure du corps de la boucle et O(g(n)) est une borne supérieure du nombre d'itérations alors O(f(n)g(n)) est une borne supérieure de la boucle

Complexité algorithmique

Complexité au pire et au meilleur des cas

Soit d l'ensemble des données de taille n et coud(d) la complexité en temps sur la donnée d:

Complexité dans le meilleur des cas

$$Inf(n) = \inf\{coud(d) \ / \ d \ de \ taille \ n\}.$$

Complexité dans le pire des cas

$$Sup(n) = \sup\{coud(d) \ / \ d \ de \ taille \ n\}.$$

- Exemple : T[] tableau d'entiers, n la taille des données

```
Pour i=1 à n faire
   Si T[i]<0 alors
     Pour i=1 à n faire
       T[i]=T[i]-1;
     Fin Pour
   Fin si
Fin pour
```

inf(n) = $O(n)$ et sup(n) = $O(n^2)$

Complexité algorithmique

Quelques propriétés liées à la notation O(.) :

- Règle du maximum :
  si $f, g: N \to R^+$, alors $O(f(n) + g(n)) = O(\max(f(n), g(n)))$
- La relation O(.) est réflexives $f(n) \in O(f(n))$
- La relation O(.) est transitive
- $si \lim_{n\to+\infty} \frac{f(n)}{g(n)} \in R^{*+}, alors\ f(n) \in O(g(n)) et\ g(n) \in O(f(n))$
- $si \lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0, alors\ f(n) \in O(g(n))\ mais\ \ g(n) \notin O(f(n))$
- $si \lim_{n\to+\infty} \frac{f(n)}{g(n)} = +\infty, alors\ g(n) \in O(f(n)), mais\ \ f(n) \notin O(g(n))$

Complexité algorithmique

Quelques propriétés liées à la notation O(.) :

La relation O(.) n'est pas symétrique :

$f(n) = O(g(n))$ n'implique pas $g(n) = O(f(n))$
Contre exemple : $5n + 43 = O(n^2)$ mais $n^2 \neq O(n)$

$f(n) \neq O(g(n))$ n'implique pas $g(n) \neq O(f(n))$
contre exemple $18n^3 - 35 \neq O(n)$ mais $n = O(n^3)$

Complexité algorithmique

Notations de Landau

- La notation **Ω** définit une borne asymptotique inférieure:
  $\exists c \in \mathbb{R}^{+*}, \exists n_0 , tels\ que\ \forall n > n_0, c * g(n) \leq f(n)$
  - Dans ce cas, on dit que f domine g, on a alors $g = O(f)$.
  - $f = \Omega(g) \Leftrightarrow\ g = O(f)$

- La notation **Θ** définit l'ordre exact d'une fonction.
  $\exists c_1, c_2 \in \mathbb{R}^{+*}, \exists n_0 , tels\ que\ \forall n > n_0, c_1 * g(n) \leq f(n) \leq c_2 * g(n)$
  - Dans ce cas, on dit que f et g sont de même ordre de grandeur asymptotique.
  - $f = \Theta(g) \Leftrightarrow f = O(g) et\ f = \Omega(g)$.
  - $f = \Theta(g) \Leftrightarrow f = O(g) et\ g = O(f)$.

- La notation $f = \boldsymbol{o}(g)$:
  $\forall\ c \in \mathbb{R}^{+*}, \exists n_0 , tels\ que\ \forall n > n_0, f(n) \leq c * g(n)$.
  - Dans ce cas, f est négligeable symptotiquement devant g.

Complexité algorithmique

Notations de Landau

- Exemples :

- $n * logn \in \Theta(n^2)$?
- Non
- $n * logn \in O(n^2)$?
- Oui
- $2^n \in \Theta(n^3)$?
- Non
- $n^3 \in \Theta(2^n)$?
- Non
- $n^3 \in O(2^n)$?
- Oui

Complexité algorithmique

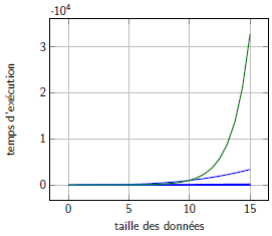
Notations de Landau

- Exemples :
- $n\,log(n) = O(n^2), O(n^3), etc.,$
- $n = O(n),$ $2n + 1 = O(3n),$ $\sqrt{n} = O(n),$ $\log(n) = O(n),$ $n = O(n^2),$
- $\sqrt{n} = o(n),$ $\log(n) = o(n),$ $n = o(n^2),$ $\log(n) = o(\sqrt{n}),$
- $n + \log(n) = \Theta(n + \sqrt{n}).$

Complexité algorithmique

Quelques classes de complexité

- On peut ranger les fonctions équivalentes dans la même classe
- En voici quelques classes fréquentes de complexité (par ordre croissant en termes de $O(.)$) :
- $O(1)$ constant
- $O(\log n)$ logarithmique
- $O(n)$ linéaire
- $O(n\,logn)$ sous-quadratique
- $O(n^2)$ quadratique
- $O(n^3)$ cubique
- $O(2^n)$ exponentiel



Complexité algorithmique

Quelques classes de complexité

- Exemples de temps d'exécution en fonction de la taille de la donnée et de la complexité de l'algorithme, si on suppose qu'une instruction est de l'ordre de la s;

T.\C.	log n	n	n log n	n^2	2^n
10	3 μs	10 μs	30 μs	100 μs	1000 μs
100	7 μs	100 μs	700 μs	1/100s	10^{14} siècles
1000	10 μs	1000 μs	1/100s	1s	astronomique
10000	13 μs	1/100s	1/7s	1,7mn	astronomique
100000	17 μs	1/10s	2s	2,8h	astronomique

Complexité algorithmique

Quelques classes de complexité : convention (hiérarchie)

- Pour faire un choix éclaire entre plusieurs algorithmes, il faut être capable de situer leur complexité
- On fait une première distinction entre les deux classes suivantes :
  - 1. les algorithmes dits polynomiaux, dont la complexité est en $O(n^k)$ pour un certain k
  - 2. les algorithmes dits exponentiels, dont la complexité ne peut pas être majorée par une fonction polynomiale ;
- De même :
  - 1. un problème de complexité **polynomiale** est considéré **facile**
  - 2. sinon (complexité **non-polynomiale** ou inconnue (!)) il est considéré **difficile**

Complexité algorithmique

Quelques classes de complexité : convention (hiérarchie)

- En pratique, il s'avère que la plupart des algorithmes polynomiaux ont un comportement asymptotique équivalent à un polynôme de faible degré, 2 ou 3.
- De plus, la classe des algorithmes polynomiaux a de bonnes propriétés de clôture (par exemple, une "séquence" de deux algorithmes polynomiaux est polynomiale).
- Cependant cette convention à ses limites !

Complexité algorithmique

Classes de complexité : Les limites de la convention

- Si un algorithme est exponentiel dans le pire des cas, il peut être polynomial en moyenne;
  - si les pires cas sont exceptionnels, il peut être ***« facile ! » ...en pratique***:
  - certains algorithmes difficiles selon la définition précédente ... sont pratiqués tous les jours (comme le simplexe).
- L'exécution d'un algorithme dont la complexité moyenne est de l'ordre de grandeur de n^5 microsecondes prendrait 30 ans si n = 1000.

Complexité algorithmique

Exemple 1

```
// t tableau de n entiers
// n entier >0

        int max= t[0];
        for (i=1; i<=n-1; i++)
                if (max <t[i]) max=t[i];
```

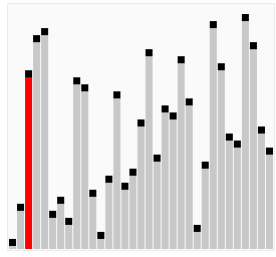
Algorithme en O(n), donc linéaire.

Complexité algorithmique

Exemple 2

```
// t tableau de n entiers
// n entier >0
        for (i=0; i<n-1; i++)
                for (j=0; j<n-1-i; j++)
                        if (t[j+1] <t[j])
                        echanger(t[j],t[j+1]);
```

Algorithme en $O(n^2)$, donc quadratique.



Complexité algorithmique

Exemple 3

```
// t tableau de n entiers
// n entier >0
        boolean permute=true;
        int last=n-1;
        while permute {
                permute=false;
                for (j=0; j<last; j++)
                        if (t[j+1] <t[j]){
                                permute=true;
                                echanger(t[j],t[j+1]);
                               }
                         last=last-1;

        }
```

- dans le meilleur des cas en O(n)
- dans le pire des cas en $O(n^2)$, donc quadratique

Complexité algorithmique

Exemple 4 Multiplication à la Russe ... ou à l'Egyptienne

- Quelle est la complexité de:

```
\\ x >=y >=0
        int Mult (int x, int y)
        int R=0;
        int a=x, b=y;
        while (b>0){
                if ( b %2 ==1)
                        R=R+a;
                a=2*a;
                b= b/2;
        }
\\ R= a*b
```

Le nombre d'opérations élémentaires est $O(\log_2 y)$.

Complexité algorithmique

Exemple 5

```
//n entier >1
      boolean Premier(int n){
      int rac =sqrt(n); \\racine carrée entière
      for (int i = 2; i<=rac; i++)
            if (n mod i=0) return false;
            return true;
      }
```

Le nombre d'opérations élémentaires $O(\sqrt{n})$

Récursivité et Paradigme Diviser pour régner

Quelques approches génériques pour aborder la résolution d'un problème :

- **Approche par force brute :** résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- **Diviser pour régner** : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original
- **Programmation dynamique** : obtenir la solution optimale à un problème en combinant des solutions optimales à des sous-problèmes similaires plus petits et se chevauchant
- **Approche gloutonne** : construire la solution incrémentalement, en optimisant de manière aveugle un critère local

Introduction

- **Approche par force brute**
- Consiste à appliquer la solution la plus directe à un problème Généralement obtenue en appliquant à la lettre la définition du problème
- Exemple simple :
  - Rechercher un élément dans un tableau (trié ou non) en le parcourant linéairement
  - Calculer a^n en multipliant a n fois avec lui-même
  - Implémentation récursive naïve du calcul des nombres de Fibonacci (voir plus loin)
  - . . .
- Souvent pas très efficace en terme de temps de calcul mais facile à implémenter et fonctionnel

Introduction

- **Approche par force brute**
- Consiste à appliquer la solution la plus directe à un problème Généralement obtenue en appliquant à la lettre la définition du problème
- Exemple simple :
  - Rechercher un élément dans un tableau (trié ou non) en le parcourant linéairement
  - Calculer a^n en multipliant a n fois avec lui-même
  - Implémentation récursive naïve du calcul des nombres de Fibonacci
  - . . .
- Souvent pas très efficace en terme de temps de calcul mais facile à implémenter et fonctionnel

Introduction

- **Approches par force brute pour le problème de tri :**
- Définition : Un tableau est trié (en ordre croissant) si tout élément est plus petit que l'élément à sa droite
- ***tri à bulle :*** parcourir le tableau de gauche `a droite en échangeant toutes les paires d'éléments consécutifs ne respectant pas cette définition. Complexité : $O(n^2)$
- ***tri par sélection*** : trouver le minimum du tableau, l'échanger avec le premier élément, répéter pour trier le reste du tableau . Complexité : $\Theta(n^2)$

Introduction

- **Approches par force brute pour le problème de tri :**
- **Recherche exhaustive**
- Générer toutes les permutations du tableau de départ (une et une seule fois)
- Vérifier si chaque tableau permuté est trié. S'arrêter si c'est le cas. Complexité : ***O(n! n)***
- Généralement utilisable seulement pour des problèmes de petite taille
- Dans la plupart des cas, il existe une meilleure solution. Dans certain cas, c'est la seule solution possible
- **Inconvénients : Produit rarement des solutions efficaces**

Récursivité

- **Concept de récursivité**
- La récursivité est la propriété que possède un objet ou un concept de s'exprimer « en fonction de lui-même » ;
- En algorithmique et en programmation, on dit qu'une fonction est récursive si elle s'appelle elle-même ;
- Une telle fonction se présente donc comme suit :

```
fonction f(P) // P est la liste de paramètres
    // instructions (1)
    x = f(Q) // appel de f avec d'autres paramètres
    // instructions (2)
    return résultat
```


Récursivité

- **Concept de récursivité**
- Remarque : le renvoi de résultat n'est pas obligatoire ;
- Rappeler qu'un algorithme correct se termine ! Il faut donc toujours s'assurer que la fonction récursive qu'on écrit arrête a un moment de s'appeler ;
- Une fonction récursive doit donc posséder une condition d'arrêt :
- **Fonction récursive avec condition d'arrêt**

```
fonction f(P)
    si test(P) alors
       // bloc d'instructions sans appel récursif
       return resultat_1
    sinon
      // bloc avec appel(s) récursif(s) utilisant de
      //nouveaux paramètres
      return resultat_2
```

Récursivité

- **Conditions d'arrêt et appels récursifs**
- Pour que la condition d'arrêt soit vraie, il faut que les appels récursifs utilisent d'autres paramètres

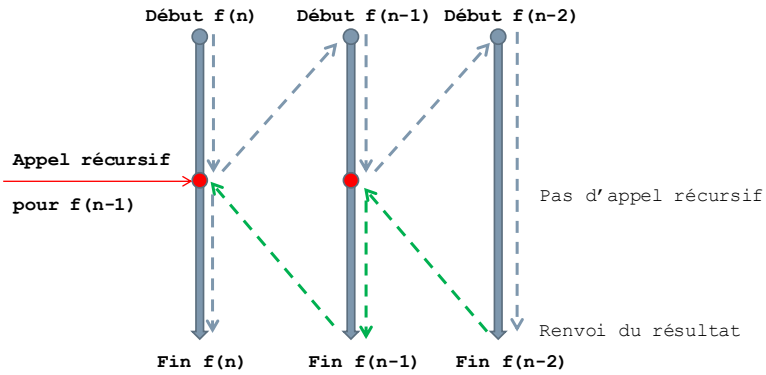
Mauvais test d'arrêt	Bon test d'arrêt

`void f(int n){ // n naturel`
`if (n == 0)`
`printf("!");`
`else{`
`printf("* ");`
`f(n);`
`}`
`}`
	`void f(int n){ // n naturel`
	`if (n == 0)`
	`printf("!");`
	`else{`
	`printf("* ");`
	`f(n-1);`
	`}`
	`}`

- La première fonction ne s'arrête jamais car si n ne vaut pas 0 au début, il ne vaudra jamais 0 ;

Récursivité

Conditions d'arrêt et appels récursifs (Problème d'espace mémoire):



Récursivité

- **Complexité des algorithmes récursifs**
- On a vu comment calculer la complexité des « algorithmes traditionnels »
- Les règles ne changent pas dans le cas des algorithmes récursifs
- Le nombre d'itérations est remplace par le nombre d'appels récursifs
- Les variantes récursives de la factorielle ou de la puissance ne sont donc pas plus rapides d'un point de vue O()

Récursivité

- **Exemple de fonctions récursives : Factorielle**

Factorielle Itérative	Factorielle Récursive

Définition :
$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$
0!=1
 	
Algorithme:
 	
`int factoI( int n){`
`  int fact=1, i=2;`
`  while (i <= n){`
`  fact = fact * i ;`
`  i=i+1 ;`
`  }`
`return fact;`
`}`
	Définition:
	$n! = 1\ si\ n = 0\ ou\ 1.$
	$n! = n \times (n-1)!\ sinon$
	
	Algorithme:
	
	`int factoR(int n){`
	`  if (n<=1)`
	`  return 1;`
	`  else`
	`  return n*factoR(n-1);`
	`}`

Complexité O(n)	Complexité O(n)!!!

Récursivité

- Exemple de fonctions récursives : Puissance

Puissance Itérative	Puissance Récursive
Définition: $a^n = a \times a \times \cdots \times a$ $a^0 = 1$ ***Algorithme:*** `int puissI(int a, int n){ int p = 1,i; for (i=1;i<=n;i++){ p=p*a; } return p; }`	*Définition:* $a^n = 1 \; si \; n = 0;$ $a^n = a \times a^{n-1} \; sinon$ ***Algorithme:*** `int puissR(int a, int n){ if (n==0) return 1; else return a*puissR(a,n-1); }`
Complexité O(n)	Complexité O(n)!!!

Récursivité

- Exemple de complexité sur une fonctions récursives implémentée de façon naïve : Fibonacci

Fibonacci Itérative	Fibonacci Récursive
$F_0 = 0, F_1 = 1;$ $F_n = F_{n-1} + F_{n-2}, n \geq 2$ `int fiboI(int n){ int v,w,F = n; // permet de //traiter les cas n=0 et n=1 if (n > 1){ v = 0; // valeur de F(0) w = 1; // valeur de F(1) } for (int i=2;i<=n;i++){ F = w + v;//F(i)=F(i-1)+F(i-2) v = w;//v vaut maintenant F(i-1) w = F; //w vaut maintenant F(i) } return F; }`	$F_0 = 0, F_1 = 1;$ $F_n = F_{n-1} + F_{n-2}, n \geq 2$ `int fiboR(int n){ if (n < 2) return n; else return fiboR(n-1)+fiboR(n-2); }`
Complexité O(n)	Complexité $O(2^n)$!!!

Récursivité

- **Bilan sur la récursivité**
- *Ecrire en récursif n'implique pas forcément un algorithme plus efficace!*
- **Avantages**
- Code plus court et plus lisible
- Permet d'implémenter le paradigme de **Diviser pour Régner**
- **Inconvénients**
- Compliqué à analyser
- Il peut y avoir des problèmes de mémoire
- Tous les langage ne sont pas récursifs
- Il n'existe pas d'algorithme récursifs pour tout

Paradigme Diviser pour régner

- **Principe général de l'approche diviser-pour-régner (Divide and conquer)**
- **Si** le problème est trivial, on le résout directement
- **Sinon** :

1) Diviser le problème en sous-problèmes de taille inférieure (Diviser)

2) Résoudre **récursivement** ces sous-problèmes (Régner)

3) Combiner les solutions aux sous-problèmes pour produire une solution au problème original

Paradigme Diviser pour régner

- **Modélisation du paradigme diviser pour régner :** La forme générale considérée est :
- **Diviser** : on découpe le problème en a sous-problèmes de tailles $\frac{n}{b}$, qui sont de même nature, avec *a>1 et b>1.*
- **Régner** : les sous-problèmes sont résolus récursivement.
- **Recombiner** : on utilise les solutions aux sous-problèmes pour reconstruire la solution au problème initial en temps $O(n^d), d > 0$.
- L'équation sur la complexité qu'on aura à résoudre quand on traduit le programme est :

$$T(1) = O(1),$$
$$T(n) = a\,T\left(\frac{n}{b}\right) + O(n^d).$$

Paradigme Diviser pour régner

- **Modélisation du paradigme diviser pour régner** : Le théorème maître permet de résoudre ce type d'équations.
- **Théorème (Théorème Maître) :**

On considère l'équation $T(n) = a\,T\left(\frac{n}{b}\right) + O(n^d)$. Soit $\lambda = \log_b(a)$, alors on a les trois cas suivants:

- Si λ>d, alors $T(n) = O(n^\lambda)$,
- Si λ<d, alors $T(n) = O(n^d)$,
- Si λ=d, alors $T(n) = O(n^d \log n)$.

- **Remarque** : En pratique, seuls les cas 1 et 3 peuvent mener à des solutions algorithmiques intéressantes. Dans le cas 2, tout le coût est concentré dans la phase "recombiner", ce qui signifie souvent qu'il y a des solutions plus efficaces.

Paradigme Diviser pour régner

- **Exemple d'application 1:** Revenons au calcul de l'exponentielle: ***Exponentiation rapide !***
- Une nouvelle définition du calcul de l'exponentielle basé sur le paradigme diviser pour régner:
- L'idée est de calculer d'abord x^n, puis par une simple opération de multiplication déduire x^{2n}
- Autrement dit: Pour calculer x^n, on divise notre problème en un sous problème de taille moitié qui est : le calcul de $x^{\frac{n}{2}}$. Avec une simple multiplication du résultat $x^{\frac{n}{2}}$ par lui-même nous pouvons reconstruire x^n.

Paradigme Diviser pour régner

- **Exemple d'application 1:** Application du théorème maître :
- **a = 1** car on appelle une seule foi la fonction soit avec n pair ou impaire,
- **b= 2** car les sous-problèmes sont de taille n/2 et
- **d = 0** car on se contente de renvoyer la solution en utilisant une multiplication, en temps constant. La complexité est donc **O(log n)**

Définition	Programme

• $x^0 = 1$,
• $x^n = (x^2)^{\frac{n}{2}}$ si n pair,
• $x^n = x(x^2)^{\frac{n-1}{2}}$ si n impair.
	`int PuissRapid(int x, int n)`
	`{`
	`if(n==0)`
	`  return 1;`
	`else`
	`  if (n%2==0)`
	`  return PuissRapid(x*x,n/2);`
	`  else`
	`  return x*PuissRapid(x*x,(n-1)/2);`
	`}`

Paradigme Diviser pour régner

- **Exemple d'application 2:** Recherche dichotomique
- T tableau trié de taille n. Rechercher si x est dans T.
- Spécifier un indice de début d et de fin f, et rechercher x entre la position f et d. L'appel initial se fait avec d=1 et f=n.

```
fonction Recherche(T,x,d,f)
si f<d alors
        renvoyer faux
sinon
        m = (d+f)/2; //milieu du tableau avec arrondi
        si T[m] = x alors
                renvoyer vrai;
        sinon
                si T[m]<x alors
                        renvoyer Recherche(T,x,m+1,f);
                sinon
                        renvoyer Recherche(T,x,d,m-1);
                finsi
        finsi
finsi
```


Paradigme Diviser pour régner

- **Exemple d'application 2:** Recherche dichotomique
- Application du théorème maître pour le calcul de la complexité
- **a=1**, on appelle soit à gauche soit à droite.
- **b=2**, les sous problèmes sont de taille n/2
- **d=0**, on se contente de renvoyer la solution, donc en temps constant,
- La complexité est **O(log n)**

Paradigme diviser pour régner

- **Exemple d'application 3: tri par fusion**
- Le tri par fusion est un exemple d'algorithme de tri base sur le principe de "Diviser pour régner".

1) Décomposer le tableau en deux sous-tableaux de même longueur (±1)
2)Appliquer l'algorithme récursivement sur chacun des 2 sous-tableaux
3) Puis fusionner les 2 sous-tableaux tries en un tableau trie.

Paradigme diviser pour régner

- **Exemple d'application 3: tri par fusion**

```
T tableau à trier
si n=1 alors
        renvoyer T;
sinon
        n = |T|;        //taille du tableau
        T1 = TriFusion(T[0...n/2]);
        T2 = TriFusion(T[(n/2)+1...n-1]);
        renvoyer  Fusion(T1,T2);
Finsi
```

- Complexité :
- La profondeur de l'arbre est $O(\log_2(n))$. A chaque étage $O(n)$ opérations de fusion: Complexité optimale : $O(n \log_2(n))$

Paradigme diviser pour régner

- **Exemple d'application 3: tri par fusion**
- Le tri par fusion est base sur le fait que fusionner deux tableaux tries en un seul tableau trie T1, T2 se fait en temps linéaire (sur le nombre total d'éléments, dans le pire et le meilleur des cas) :

```
i1, i2 = 0, et T = tableau vide
TANT QUE l'on n'a pas atteint la fin d'un des tableaux:
        SI T1[i1] <= T2[i2] ALORS
                Insérer T1[i1] a la fin de T
                incrémenter i1
        SINON
                Insérer T2[i2] a la fin de T
                incrémenter i2
        FINSI
FIN TANT QUE
Insérer les éléments restants du tableau non vide en fin de
T
```


Paradigme diviser pour régner

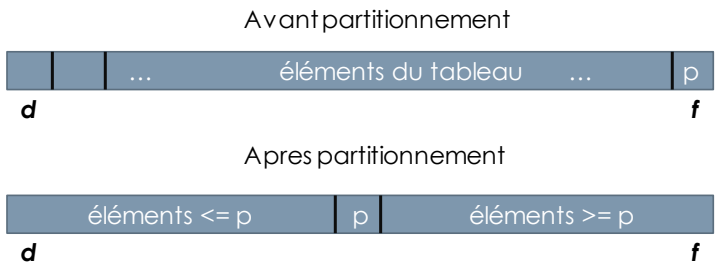
- **Exemple d'application 4: tri rapide (quiksort)**
- **Principe** :
- La méthode consiste à placer un élément du tableau (appelé ***pivot***) à sa place définitive, en effectuant des permutations des éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite.
- Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

Paradigme diviser pour régner

- **Exemple d'application 4: tri rapide**
- **Principe** :
- Pour partitionner un sous-tableau :
- on choisie (***arbitrairement***), le dernier élément du sous-tableau comme pivot;
- on place tous les éléments inférieurs au pivot en début du sous-tableau ;
- on place le pivot à la fin des éléments déplacés.

Paradigme diviser pour régner

- **Exemple d'application 4: tri rapide**
- **Principe** :
- Pour partitionner un sous-tableau :



Paradigme diviser pour régner

- **Exemple d'application 4: tri rapide**

- **Algorithme du partitionnement**:
- Introduire les indices ***i*** et ***j*** initialisés respectivement au début et à l'avant dernier élément du tableau
- Remonter ***i*** jusqu'au premier élément supérieur au pivot ***p***; redescendre ***j*** jusqu'au premier élément inférieur à ***p***
- On échange les éléments d'indice ***i*** et ***j***
- On itère ce procédé tant que ***i<=j***
- A la fin, on place la valeur pivot en position ***i***

Paradigme diviser pour régner

- **Exemple d'application 4: tri rapide**

- **Algorithme de tri rapide :**

```
SI d<f ALORS
      i = partitionnement(T, d, f)
      TriRapide(T, d, i-1)
      TriRapide(T,  i+1, f)
Fin SI
```

Paradigme diviser pour régner

- **Exemple d'application 4: tri rapide**
- **Algorithme de partitionnement**:

```
I = d et j = f
p = T[f]                    //p valeur pivot
TANT QUE i<=j faire
        TANT QUE i<f et T[i]<= p FAIRE
                i = i + 1
        FIN TQ
        TANT QUE j>=d et T[i]>= p FAIRE
                j = j - 1
        FIN TQ
        SI i < j ALORS
                echanger(T[i],T[j])
        FINSI
FIN TANT QUE
T[f]=T[i] et T[i]= p        //on place la valeur pivot en i
renvoyer i                  //renvoyer l'endroit de la séparation
```


Paradigme diviser pour régner

- **Exemple d'application 4: tri rapide**
- **Complexité**
- $T(n) = O(n) + T(p) + T(n - p)$ avec $T(1) = O(1)$.
- faire des hypothèses sur la valeur de l'indice de partitionnement p.
- Dans le cas où le partitionnement est systématiquement égal à 1, les sous-tableaux gauche et droit sont totalement déséquilibrés, avec respectivement $n - 1$ termes et un seul terme. Cette situation correspond aux tableaux *triés*! Dans ce cas, la récurrence devient: $T(n) = O(n) + T(1) + T(n - 1)$
- soit une complexité dans le pire des cas $T(n) = O(n^2)$

Paradigme diviser pour régner

- **Exemple d'application 4: tri rapide**
- **Complexité**
- $T(n) = O(n) + T(p) + T(n - p)$ avec $T(1) = O(1)$.
- faire des hypothèses sur la valeur de l'indice de partitionnement p.
- Le minimum d'appels récursifs est atteint dans le cas d'un tableaux T où le pivot est toujours la valeur moyenne de celles du tableau, ce qui correspond intuitivement à des tableaux bien "mélangés".
- La formule de récurrence devient dans ce cas $T(n) = O(n) + 2\,T(n/2)$; soit une complexité dans le meilleur des cas $O(n \log n)$

Paradigme diviser pour régner

- **Exemple d'application 4: tri rapide**
- **Remarques**
- La complexité en moyenne du tri rapide pour n éléments est $O(n \log n)$, mais la complexité dans le pire des cas est quadratique. Malgré ce désavantage théorique, c'est en pratique un des tris les plus rapides, et donc un des plus utilisés. Le pire cas est en effet peu probable lorsque l'algorithme est correctement mis en œuvre,

Paradigme diviser pour régner

- **Des tris encore plus rapides ?**

- **On a vu**
- Des algorithmes de tri dont la complexité dans le pire des cas est en $O(n^2)$ (tri par sélection, tri à bulles).
- Le tri fusion, dont la complexité dans le pire des cas est en O(n log n)

- **Peut-on faire mieux ?**
- Existe-t-il des algorithmes de tri (tri par comparaison) asymptotiquement plus rapides que le tri fusion de plus d'un facteur constant ?

- **Corollaire**
- Aucun tri par comparaisons ne possède une complexité meilleure que O(n log n). Le tri fusion a une complexité optimale.

- **Une précision importante:**
- Ces O sont en fait des Θ.