Introduction

Plan

- Eléments introductifs
  - Concepts, vocabulaire, rappels
  - Retour sur un exemple concret
    - Tri par fusion
  - Classes de complexité

Rappels

- Algorithme :
  - Suite (ordonnée) d'instructions permettant de réaliser une tâche
  - Vision de haut niveau masquant certains détails de mise en œuvre
  - Focalisation sur la suite logique (abstraction des contingences matérielles)

- Programme
  - Traduction dans un langage de programmation d'un algorithme
  - Syntaxe, sémantique à respecter
  - Le texte devient action

Construction d'un algorithme

- Etant donné un énoncé du problème
  - Données disponibles (d)
  - Résultats attendus (r)

- Fournir une suite d'instructions permettant étant donné d de construire r
  - Instructions =?

Construction de d'algorithme

- Résoudre un problème = trouver la suite logique de tous les ordres nécessaire à la solution ➔ algorithme
  - Décomposer l'énoncé en étapes
  - Définir les objets manipulés
  - Définir les opérations (actions)
- Exemple: convertir un nombre entier de secondes en j/h/m/s

Un premier exemple

- La suite de fibonacci

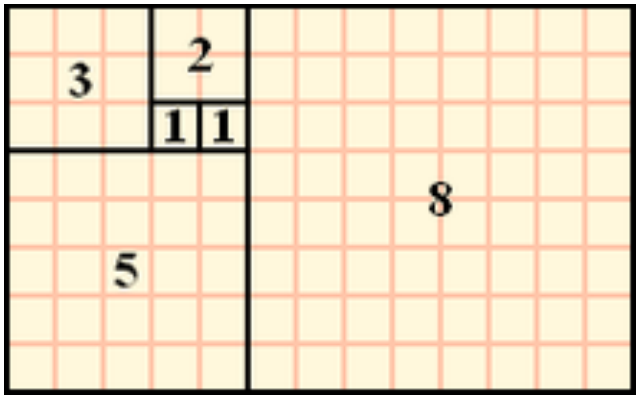
Suite de fibonacci – récursivité multiple

$$F_0 = 0, F_1 = 1, \ \forall\, n \geq 2, \quad F_n = F_{n-1} + F_{n-2}$$

```
static int fib(int n){
        if(n <= 1) return n; // cas de base
        else return fib(n-1)+fib(n-2);
    }
```

Carrés de Fibonacci en spirale:

(extrait de Wikipedia)



Un premier exemple

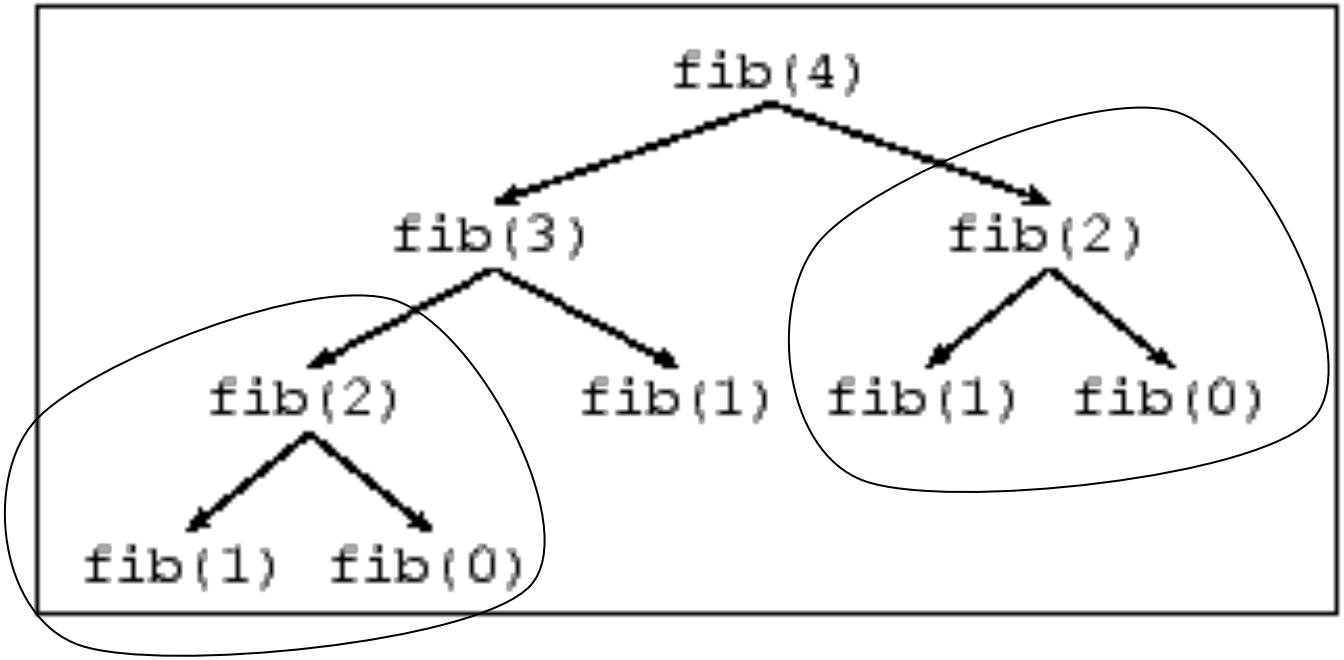
- La suite de fibonacci
- F(48) = ???
- Essayons
- Et F(65)

Autres écritures de l'algorithme ?

- Voir code
- Pourquoi est-ce lent (ou pas)?
- Pourrait on prédire le temps qui va être mis?

Problème de complexité

Deux critères de performances: temps d'exécution et place en mémoire



Nombre d'appels (temps d'exécution) …?

Représentation des données/résultats

- Sous la forme de type de « données »
  - Entier, réel, …
  - Classe d'objet
  - Et plus généralement
    - Enumération, (bleu, blanc, rouge)
    - Restriction 1..20
    - "renomage" : type indice = entier, type patronyme = chaine,
    - Etc.
- Type de données
  - Description des valeurs du type (intension/extension)
  - Opérations sur le type
  - Ex: booleen, liste, tableau, graphe

Représentation des données

- Coût en mémoire
  - Représentation des informations
- Coût d'utilisation
  - Ex: liste
    - Opération ajout en queue
    - Si liste chainée « classique » ➔ tout parcourir
    - Si liste dans un tableau ou liste chainée avec pointeur de queue
      ➔ 2 ou 3 opérations

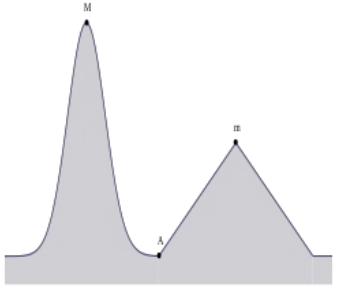
Instructions

- Affectation (et expression)
- Condition
- Itérations
- Procédure/fonction (appel)

Exécution d'un algorithme

- Programmation (***implantation***) dans un langage
- Exécution sur une machine
- Consommation de ressources
  - Mémoire
  - Processeur (temps de calcul)
- Comparer deux algorithmes devient dépendant
  - De la machine
  - De l'implantation

Stratégies de conception d'algo.



- Force brute
  - une approche directe, OK pour des pb simples
- Diviser pour régner
  - découper un problème en petite taille, typiquement à l'aide de la récursivité
- Approche gloutonne
  - espérer que l'optimalité locale conduit à l'optimalité globale, mais ne marche pas dans tous les cas, eg. rendu de monnaie
- Programmation dynamique
  - optimiser des sommes de fonctions monotones croissantes sous contrainte, par le principe : toute solution optimale s'appuie elle-même sur des sous-problèmes résolus localement de façon optimale
- Backtracking
  - pb d'optimisation et combinatoires, avec une approche sans exploiter tout espace d'état

Intérêts

- Gestion d'un annuaire
  - Opérations
    - Ajout, suppression et consultation d'un contact
  - Combien de temps pour faire une opération selon l'implantation et le nombre de contacts
    - Implantations possibles
      - Tableau (statique) = ???
      - Tableau dynamique
      - Liste
      - Arbre binaire de recherche
      - Table

Comparaison d'algorithmes

- Notion de complexité
  - Nombre d'opération fondamentales nécessaires à son exécution
- Abstraction dans un souci de généralité
  - Pas de considération de la machine, système, langage, compilateur,
- Modèle
  - On considère que toutes les opérations ont le même coût
  - On exprime le coût mémoire/exécution comme une fonction de la taille des données

Principes de calcul

- Séquences d'instruction
  - Cumul
  - P(i1;i2;i3) = P(i1)+P(i2)+P(i3)
- Condition
  - Majorant
  - P(si C alors A sinon S) ≤ P(C) + max (P(A), P(S))

- Itération
  - Cumul sur le nombre d'itération
  - Somme(P(i)) où
    - i = variable d'itération;
    - P(i) nombre d'opérations lors de l'exécution de *la ième itération*
- Fonction/procédure
  - Non récursif : reprendre les principes ci-dessus
  - Récursif = formule de récurrence

fonction FunctionRecursive (n)

```
1   si (n > 1) alors
2       FunctionRecursive(n/2),    coût T(n/2)
3       Traitement(n),             coût C(n)
4       FunctionRecursive(n/2),    coût T(n/2)
```

Equation récursive

$$T(n) = 2 * T(n/2) + C(n)$$

si $C(n) = 1$ alors $T(n) = K \times n$
si $C(n) = n$ alors $T(n) = K \times n \times \log n$

En pratique

L: tableau [1..n] d'element
X entier
j entier

j ← 1
Tant que (j<=n) et L[j]!= X
faire
    j ← j+1
ftq

Nombre d'itérations,
nombre d'opérations par itération

Qu'est-ce qui est significatif ?

En pratique

L: tableau [1..n] d'element
X entier
j entier

j ← 1
Tant que (j<=n) et L[j]!= X
faire
    j ← j+1
ftq

Qu'est-ce qui est significatif ?

j ← j+1 : disparaît avec un « for »
L[j] : structure de tableau (disparaît avec une liste
mais apparition de nouvelles instructions)

En pratique

L: tableau [1..n] d'element
X entier
j entier

```
j ← 1
Tant que (j<=n) et L[j]!= X faire
    j ← j+1
ftq
```

C'est une recherche séquentielle

significatif : comparaisons de X

Complexité :

meilleur cas 1

pire cas n O(n)

en moyenne ? Dépend des données

Compromis espace/temps

- Perdre de l'espace pour gagner du temps
- Perdre du temps pour gagner de l'espace
- Ex:
  - calculer une propriété quand nécessaire
  - La calculer une fois et la stocker

- Trouver un compromis
  - Efficacité demande souvent de l'espace (en plus)

Comparaison d'algorithme

- Ordre de grandeur
  - Soit deux algorithmes A et B et leurs complexités P(A) et P(B)
  - Comment les comparer ?

Ordre de grandeur

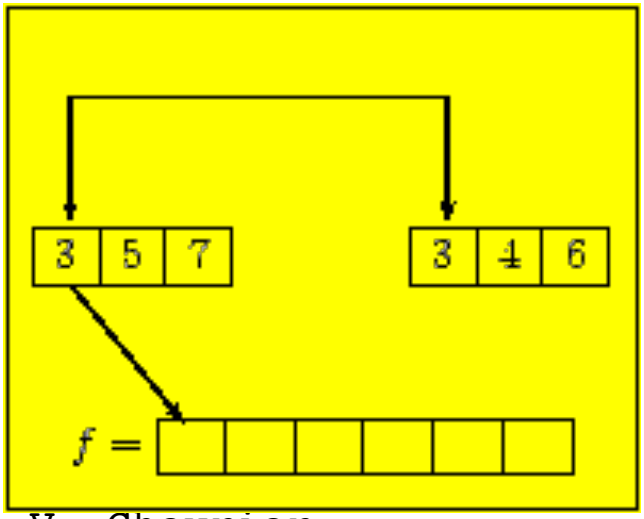
Complexité un exemple concret

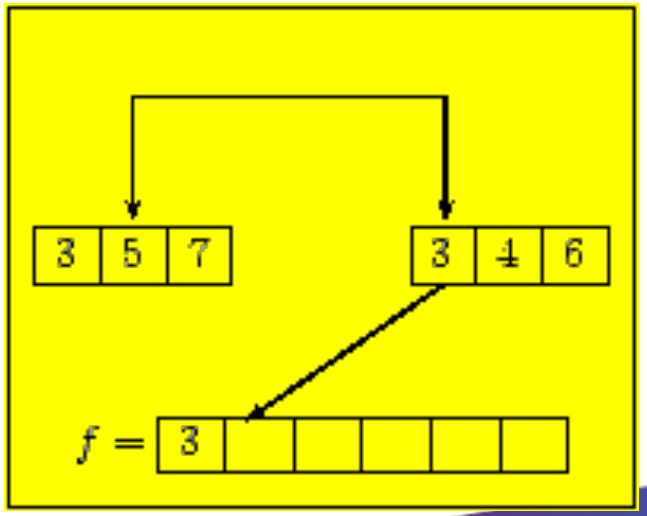
- Tri par fusion
  - Complexité en temps (nombre d'opérations)
  - Complexité en espace (mémoire nécessaire)

Tri par fusion: un algorithme à O(nlogn)

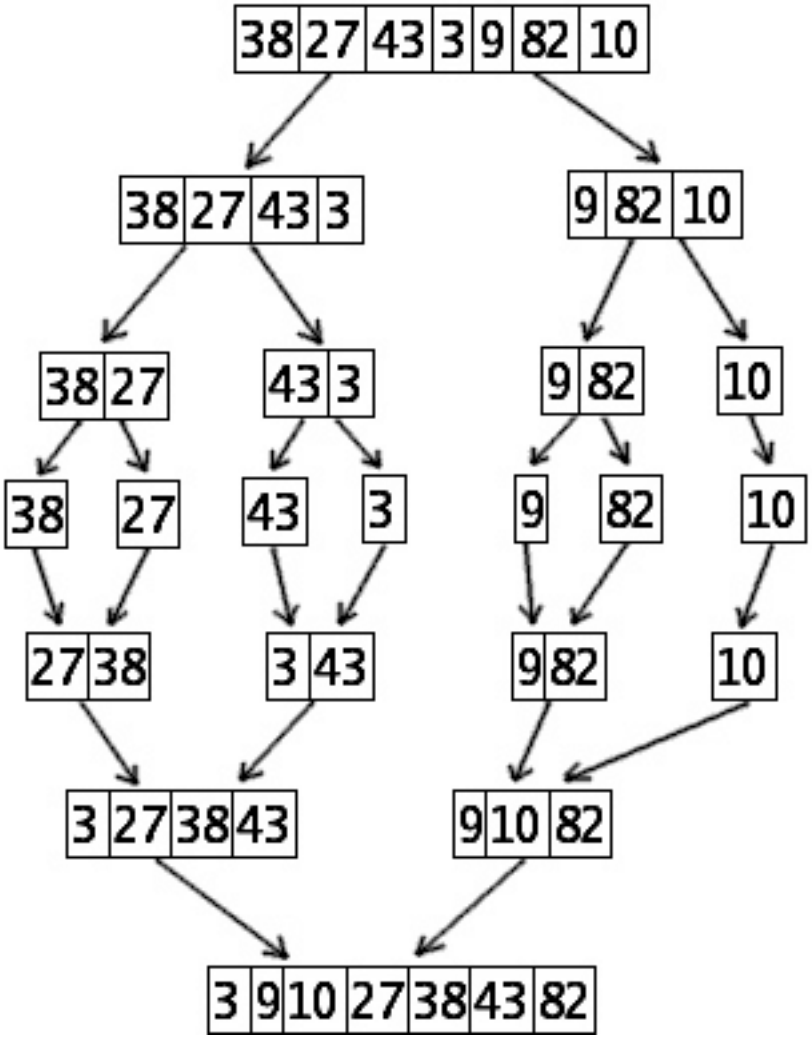
Tri par fusion est un exemple typique de « diviser pour résoudre »
Principe:
-couper en deux le tableau à trier
-trier chacune des deux moitiés en appelant récursivement la même méthode, ceci jusqu'au point où le tableau ne contient plus qu'un élément
-Interclasser les deux tableaux par la méthode de fusion

Exemple de fusion: `int[] t = {3, 5, 7, 3, 4, 6};`





Tri par fusion: un exemple



[source: wikipédia]

Tri par fusion: un algorithme à O(nlogn)

```
static int[] fusionner(int[] tg, int[] td){
        int[] f = new int[tg.length + td.length];
        int g = 0, d = 0;
        for(int k = 0; k < f.length; k++){
            // f[k] est la case à remplir
            if(g >= tg.length) // g est invalide
                f[k] = td[d++];
            else if(d >= td.length) // d est invalide
                f[k] = tg[g++];
            else // g et d sont valides
                if(tg[g] <= td[d])
                    f[k] = tg[g++];
                else // tg[g] > td[d]
                    f[k] = td[d++];
        }
        return f;
}
```

Nb de comparaisons: O(n)

Tri par fusion: un algorithme à O(nlogn)

```
static int[] triFusion(int[] t){
    if(t.length == 1) return t;
    int m = t.length / 2;
    int[] tg = sousTableau(t, 0, m);
    int[] td = sousTableau(t, m, t.length);
 // on trie les deux moitiés
    tg = triFusion(tg);
    td = triFusion(td);
  // on fusionne
    return fusionner(tg, td);
}
```

```
// on crée un tableau contenant t[g..d[
    static int[] sousTableau(int[] t, int g, int d){
        int[] s = new int[d-g];

        for(int i = g; i < d; i++)
            s[i-g] = t[i];
        return s;
    }
```

Tri par fusion: un algorithme à O(nlogn)

La complexité spatiale (place mémoire nécessaire): *2n = O(n)*

La complexité temporelle: T(n) = 2T(n/2) + n
//correspond au tri de deux tableaux de n/2, plus copie d'un tableau n

Résolution de l'équation récurrente:
-diviser par n: T(n)/n = T(n/2)/(n/2) + 1
-poser n = 2^k, on a:

$$\frac{T(2^k)}{2^k} = \frac{T(2^{k-1})}{2^{k-1}} + 1 = \frac{T(2^{k-2})}{2^{k-2}} + 2 = \ldots = \frac{T(1)}{1} + k$$

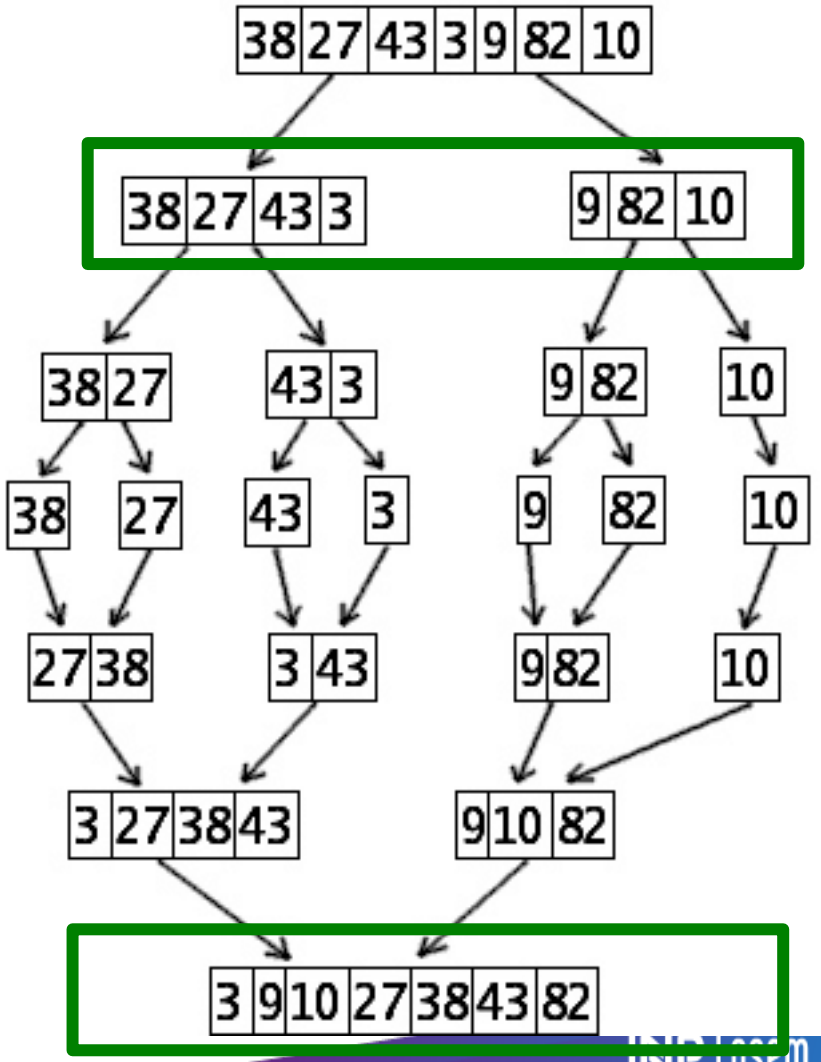
-comme k=$\log_2 n$, on a:

$$T(2^k) = T(n) = (k+1)2^k = (\log_2 n + 1)n = O(n \log_2 n)$$

Complexité spatiale

- Combien de tableaux sont créés à chaque étape ?

```
static int[] triFusion(int[] t){
    if(t.length == 1) return t;
    int m = t.length / 2;
    int[] tg = sousTableau(t, 0, m);
    int[] td = sousTableau(t, m, t.length);
 // on trie les deux moitiés
    tg = triFusion(tg);
    td = triFusion(td);
  // on fusionne
    return fusionner(tg, td);
}
```



Tri par fusion: un algorithme à O(nlogn)

```
static int[] fusionner(int[] tg, int[] td){
        int[] f = new int[tg.length + td.length];
        int g = 0, d = 0;
        for(int k = 0; k < f.length; k++){
            // f[k] est la case à remplir
            if(g >= tg.length) // g est invalide
                f[k] = td[d++];
            else if(d >= td.length) // d est invalide
                f[k] = tg[g++];
            else // g et d sont valides
                if(tg[g] <= td[d])
                    f[k] = tg[g++];
                else // tg[g] > td[d]
                    f[k] = td[d++];
        }
        return f;
}
```

Nb de comparaisons: O(n)

Complexité spatiale

- Étape 1: N données
  - N/2 +N/2 + N =2N
- Etape 2 N/2 données
  - …. = N
- Etape 3 N/4 données
  - ….. N/2

Combien d'étapes ?

2(N+N/2+N/4+ ....) =2N (1+1/2+1/4+1/8+..) <4 N

Complexité spatiale

- Comment la réduire ?

- Idée:
  - Gérer un seul tableau pour stocker
  - Les sous tableaux sont désignés par indices de début et de fin

Complexité spatiale

```
static int[] triFusion(int[] t, int gauche, int droite ){
    if(t.length == 1) return t;
          int m = t.length / 2;
          int[] tg = sousTableau(t, 0, m);
           int[] td = sousTableau(t, m, t.length);
 // on trie les deux moitiés
    tg = triFusion(tg……);
    td = triFusion(td,…..);
  // on fusionne
    return fusionner(tg, td, …….);
}
```

Retour (comparaison d'algorithmes)

Notion de classes de complexité

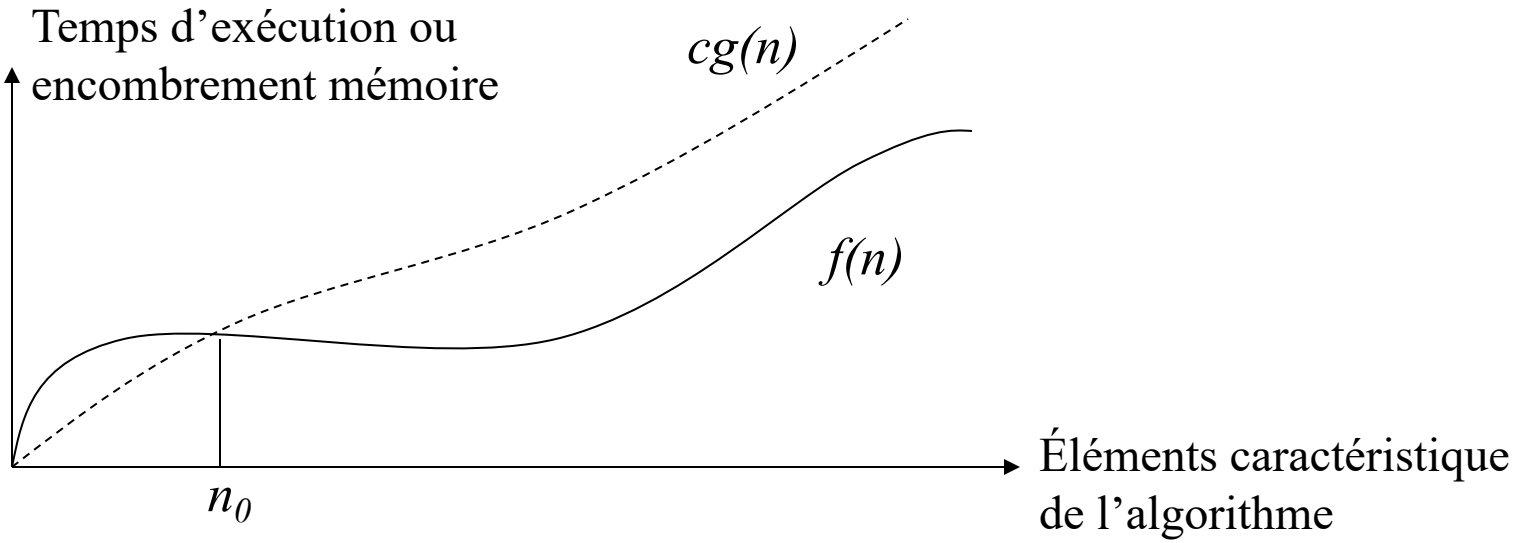
Ou

Pourquoi utiliser $O(n^2)$ au lieu du nombre exact $(n^2 – n)/2$?

Complexité des algorithmes

Soient deux fonctions positives f et g, on dit que $f(n)$ est $O(g(n))$ s'il existe deux constantes positives c et n_0 telle que $f(n) \leq cg(n)$ pour tout $n \geq n_0$.

L'idée est donc d'établir un ordre de comparaison relatif entre les fonctions f et g. Cette définition indique qu'il existe une valeur n_0 à partir de laquelle $f(n)$ est toujours inférieure ou égale à $cg(n)$. On dit alors que $f(n)$ est de *l'ordre de* $g(n)$.



Complexité des algorithmes

Nous pouvons montrer facilement que la fonction $(n^2 – n)/2$ est de l'ordre de $O(n^2)$.

Pour que $(n^2 – n)/2 \leq c\, n^2$, en prenant $c = ½$, il est évident que n'importe quel $n_0 > 0$ vérifie l'inégalité.

Remarque: on aurait pu aussi bien écrire que $(n^2 – n)/2$ est $O(n^3)$ ou $O(n^5)$, mais $O(n^2)$ est plus précis.

D'une façon générale, on choisira une fonction g la plus proche possible de f, en respectant les règles suivantes:

- $cf(n) = O(f(n))$ pour tout facteur constant c
- $f(n) + c = O(f(n))$ pour tout facteur constant c
- $f(n)+g(n) = O(max(f(n), g(n)))$
- $f(n)\ x\ g(n) = O(f(n)\ x\ g(n))$
- Si $f(n)$ est un polynôme de degré m,
  $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_m n^m$, alors $f(n)$ est de $O(n^m)$.
- $c^n = O(c^n)$ pour tout $c > 1$
- $\log n^m = O(\log n)$ pour tout $m > 0$
- $\log n = O(\log n)$

Complexité des algorithmes

- Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité
  - Temps constant O(1)
  - Sous-linéaire/logarithmique: O(log n)
  - Linéaire: O(n) ou O(n log n)
  - Polynômiale: $O(n^k)$
  - Exponentielle: $O(c^n)$, $c > 1$
  - Non exponentielle (Nexp) …

Complexité des algorithmes

complexité \ n	10	20	30	40	50	60
n	0,00001 secondes	0,00002 secondes	0,00003 secondes	0,00004 secondes	0,00005 secondes	0,00006 secondes
n^2	0,0001 secondes	0,0004 secondes	0,0009 secondes	0,0016 secondes	0,0025 secondes	0,0036 secondes
n^3	0,001 secondes	0,008 secondes	0,027 secondes	0,064 secondes	0,125 secondes	0,216 secondes
n^5	0,1 secondes	3,2 secondes	24,3 secondes	1,7 minutes	5,2 minutes	13,0 minutes
2^n	0,001 secondes	1,0 secondes	17,9 minutes	12,7 jours	35,7 ans	366 siècles
3^n	0,059 secondes	58 minutes	6,5 ans	3855 siècles	$2x10^8$ siècles	$1,3x10^{13}$ siècles

Les problèmes classiques

- Tri
- Recherche
- Traitement des chaînes de caractères
- Graphes (le plus court chemin, coloriage,…)
- Problèmes combinatoires
- Calcul numérique (intégrale, systèmes d'équations, …)

En bref

- ***Concevoir*** et ***analyser*** des algorithmes résolvant ces familles problèmes

- Concevoir
  - Comment construire l'algorithme ?
- Analyser
  - Estimer les ressources nécessaires à son utilisation

- Relations entre complexité et méthode de conception