

Algorithmes et structures de données

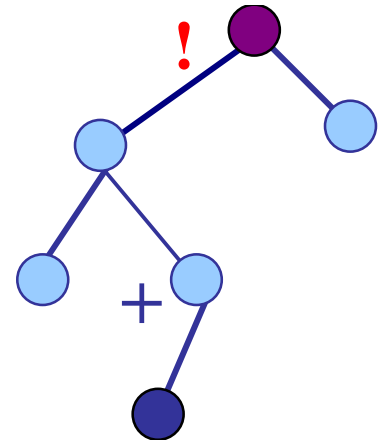
IFT-2008/GLO-2100

Mondher Bouden

Introduction à l'algorithmique

Édition ÉTÉ 2018

© Mario Marchand et Abder Alikacem



Plan

- Algorithme
- Analyse algorithmique
- Efficacité des algorithmes
- Notations asymptotiques
- Opération baromètre
- Comparaison entre les classes de complexité
- Analyse en pire cas, meilleur cas et cas moyen
- Résolution de récurrences
- Récursivité

Algorithme ?

- **Définition**

« *Enchaînement d'actions permettant l'accomplissement d'une tâche* »

- Le nom vient du Mathématicien perse (8^e siècle): ***al-Khwārizmī***
- C'est l'auteur des premiers algorithmes (sous forme écrite)
 - Description de procédures permettant de faire des opérations sur des nombres (addition, soustraction, multiplication, division)

Algorithmique (étude des algorithmes)

Concerne la **Conception** d'algorithmes pour la résolution de problèmes:

Déterminer la séquence d'opérations pour accomplir la tâche désirée.

Effectuer une preuve de bon fonctionnement (exactitude/validité):
démontrer que l'algorithme effectue ce qu'il est sensé faire.

Et l'**Analyse** de la Complexité/Efficacité des algorithmes:

Déterminer la quantité de ressources utilisées
pour exécuter l'algorithme.

Effacité des algorithmes

- L'efficacité d'un algorithme se mesure par la quantité de ressources utilisées. Ces ressources sont principalement:
 - **Le temps d'exécution:**
 - Combien de temps mets l'algorithme pour résoudre une instance de taille n
 - **L'espace mémoire utilisée:**
 - Combien d'octets l'algorithme utilise pour résoudre une instance de taille n
 - **La bande passante utilisée:**
 - Combien d'octets l'algorithme doit-il échanger avec une entité pour accomplir sa tâche. (nous ne considérerons pas cela dans ce cours)

L'approche empirique

- Consiste à essayer l'algorithme sur différents jeux de données bien choisis.
- Avantages:
 - Nécessite aucune connaissance en algorithmique
 - résultats réalistes pour les instances testées dans l'environnement de test.
- Inconvénients:
 - pas toujours généralisable aux instances non testées.
 - coûteux et long (nécessite beaucoup de tests)
 - **dépend de l'environnement d'exécution** (le processeur, l'OS, la charge, l'implémentation...)

L'approche algorithmique

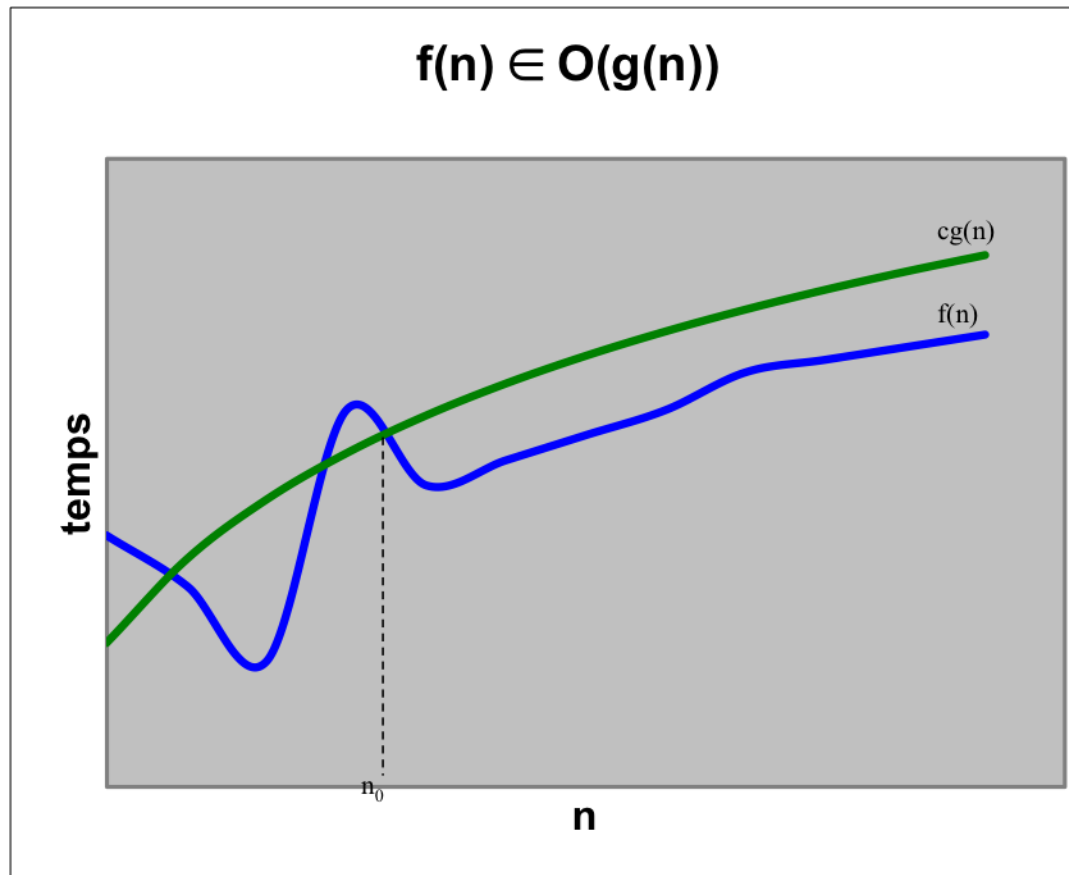
- Le temps d'exécution d'un algorithme est mesuré par le nombre d'opérations élémentaires effectuées durant son exécution
 - Une opération élémentaire est une opération non divisible.
 - Ex: comparaison, addition, multiplication, ... d'une donnée élémentaire (comme un int, float, char, double...).
 - Exemples d'opérations non élémentaires: le tri d'un tableau, la recherche d'un élément, l'exécution d'un autre algorithme...
- Avantages:
 - **résultats généraux**: ne dépend pas de l'environnement d'exécution
 - Estimation rapide et peu coûteuse
- Inconvénients:
 - Nécessite la compréhension de notions algorithmiques.

Approche algorithmique et analyse asymptotique

- Nous utilisons l'approche algorithmique
 - Raison: L'approche empirique ne caractérise pas l'algorithme. Ça caractérise plutôt l'algorithme **et** son environnement d'exécution.
- L'approche algorithmique utilise **l'analyse asymptotique**: le comportement de l'algorithme lorsque la **taille n** de l'instance à traiter tends vers l'infini.
- **Temps d'exécution d'un algorithme = nombre d'opérations élémentaires pour traiter une instance de taille n .**
 - Analyse asymptotique: on s'intéresse uniquement à la croissance de ce nombre d'opérations en fonction de n lorsque n tends vers l'infini.
 - On utilise la **notation asymptotique** pour exprimer l'ordre de croissance de ce nombre d'opérations en fonction de n .
 - Il s'agit d'une mesure caractérisant uniquement l'algorithme (et donc indépendante de l'environnement d'exécution).

Notation O (*big-oh*)

- Détermine une **borne supérieure** éventuelle
- Définition formelle : $f(n) \in O(g(n))$ s'il existe deux constantes positives n_0 et c tel que $f(n) \leq cg(n)$ pour tout $n \geq n_0$



Exemple

- Considérons $f(n) = 12n^2 + 5n$
 - On a que $12n^2 + 5n \leq 17n^2$ pour tout $n \geq 1$
 - Donc $f(n) \in O(n^2)$
- Par contre il n'existe pas n_0 et c positifs tels que $12n^2 + 5n \leq c n$ pour tout $n \geq n_0$
- Donc $f(n) \notin O(n)$

Propriété de transitivité

Si

$$f(n) \in O(g(n))$$

et

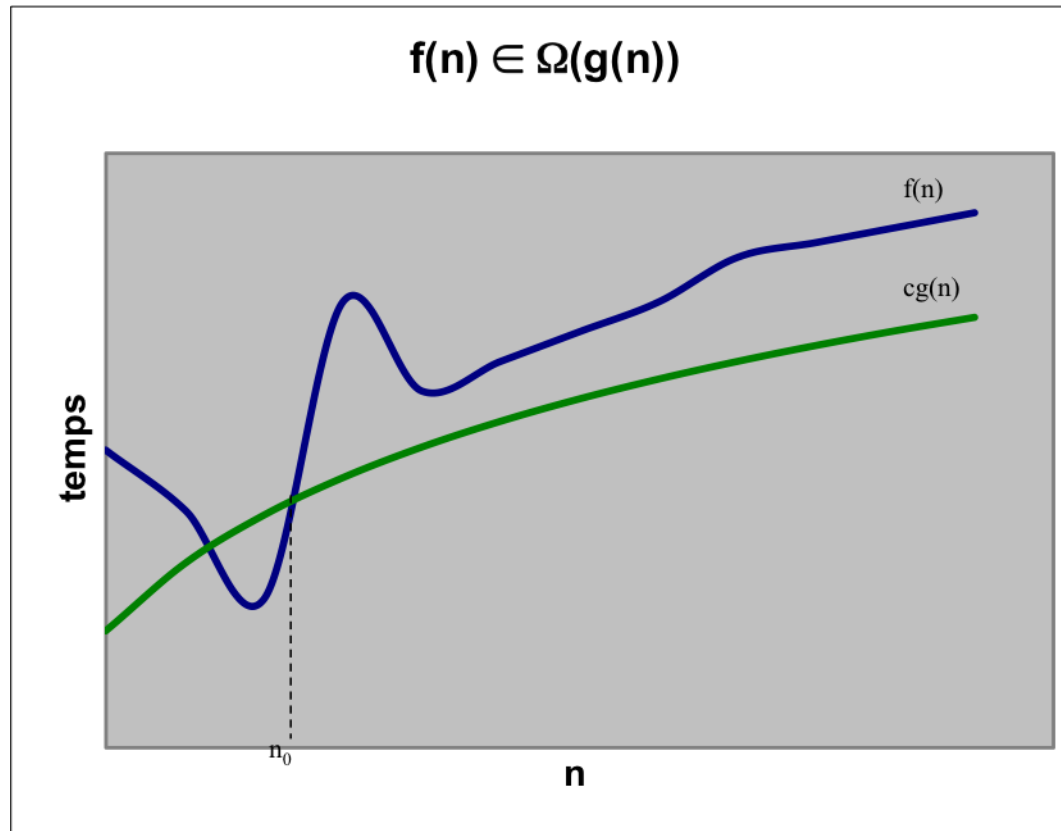
$$g(n) \in O(h(n)),$$

alors

$$f(n) \in O(h(n)).$$

Notation Ω (*big-omega*)

- Détermine **une borne inférieure** éventuelle
- Définition formelle : $f(n) \in \Omega(g(n))$ s'il existe deux constantes positives n_0 et c telles que $f(n) \geq cg(n)$ pour tout $n \geq n_0$



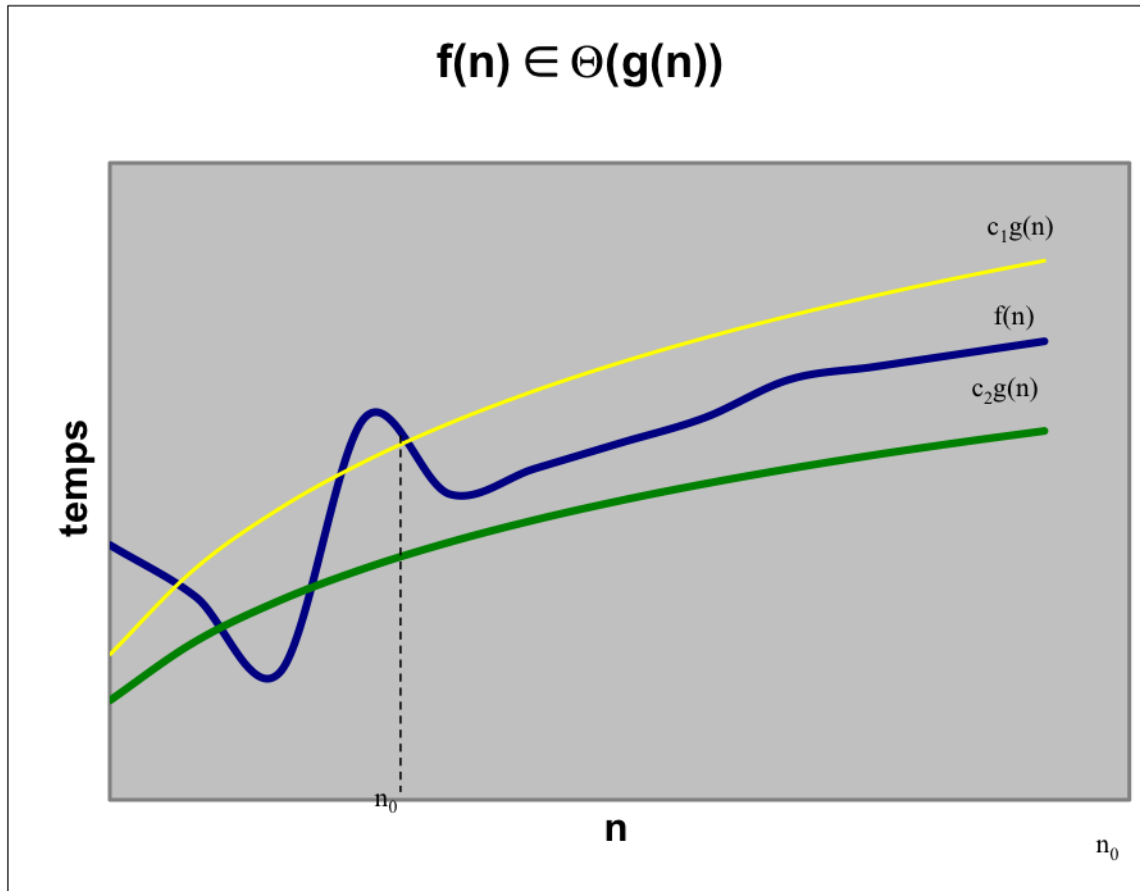
- Donc: $f(n) \in \Omega(g(n))$ ssi $g(n) \in O(f(n))$

Exemple

- Considérons $f(n) = 12n^2 + 5n$
 - On a que $12n^2 + 5n \geq 12n^2$ pour tout $n \geq 1$
 - Donc $f(n) \in \Omega(n^2)$
- Par contre il n'existe pas n_0 et c positifs tels que $12n^2 + 5n \geq c n^3$ pour tout $n \geq n_0$
- Donc $f(n) \notin \Omega(n^3)$

Notation Θ

Définition formelle : $f(n) \in \Theta(g(n))$ s'il existe trois constantes positives n_0 , c_1 et c_2 tel que $c_2 g(n) \leq f(n) \leq c_1 g(n)$ pour tout $n \geq n_0$



Donc $f(n) \in \Theta(g(n))$ ssi $f(n) \in \Omega(g(n))$ et $f(n) \in O(g(n))$

Exemple

- Considérons $f(n) = 12n^2 + 5n$
 - On a que $12n^2 \leq 12n^2 + 5n \leq 17n^2$ pour tout $n \geq 1$
 - Donc $f(n) \in \Theta(n^2)$
- Bien qu'il soit possible d'utiliser les définitions de O , Ω et Θ pour comparer les ordres de croissance de fonctions, il est plus simple d'utiliser le théorème sur les limites (à la page suivante).

Utilisation des limites pour comparer ordres de croissance

$$\text{si } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \begin{cases} = c \text{ (fini)} > 0 & \text{alors } f(n) \in \Theta(g(n)) \\ = 0 & \text{alors } f(n) \in O(g(n)) \text{ et } f(n) \notin \Theta(g(n)) \\ = +\infty & \text{alors } f(n) \in \Omega(g(n)) \text{ et } f(n) \notin \Theta(g(n)) \end{cases}$$

- Lorsque $f(n) \in \Theta(g(n))$ nous disons que $f(n)$ et $g(n)$ ont le même ordre de croissance.
- Lorsque $f(n) \in O(g(n))$ et $f(n) \notin \Theta(g(n))$ nous disons que l'ordre de croissance de $f(n)$ est strictement plus petit que celui de $g(n)$
- Lorsque $f(n) \in \Omega(g(n))$ et $f(n) \notin \Theta(g(n))$ nous disons que l'ordre de croissance de $f(n)$ est strictement plus grand que celui de $g(n)$

Remarques sur la notation asymptotique

- $O(g(n))$, $\Omega(g(n))$ et $\Theta(g(n))$ définissent des **ensembles de fonctions**.
 - C'est pour cette raison que nous utilisons le symbole d'appartenance pour indiquer qu'une fonction appartient à un ensemble. Ex: $f(n) \in O(g(n))$.
 - Certains auteurs utilisent le symbole d'égalité $=$ à la place du symbole d'appartenance \in , mais nous décourageons cette pratique!
- Notez que certains ensembles sont strictement plus grand que d'autres. Par exemple, si $f(n) \in O(n) \Rightarrow f(n) \in O(n^2)$ mais il existe des fonctions dans $O(n^2)$ qui ne sont pas dans $O(n)$.
 - On a donc que $O(n) \subset O(n^2)$.

Exemples d'utilisation des limites

Exemple 1: comparez l'ordre de croissance de $n(n-1)$ avec l'ordre de croissance de n^2

$$\lim_{n \rightarrow \infty} n(n-1) / n^2 = \lim_{n \rightarrow \infty} (1 - 1/n) = 1$$

Alors $n(n-1)$ et n^2 ont le même ordre de croissance.

Donc $n(n-1) \in \Theta(n^2)$. i.e., $O(n(n-1)) = O(n^2)$.

Exemple 2: comparez l'ordre de croissance de n^2 avec l'ordre de croissance de n^3

$$\lim_{n \rightarrow \infty} n^2 / n^3 = \lim_{n \rightarrow \infty} 1/n = 0$$

Alors n^2 a un ordre de croissance strictement plus petit que n^3 .

Donc $n^2 \in O(n^3)$ mais $n^2 \notin \Theta(n^3)$. i.e., $O(n^2) \subset O(n^3)$.

Exemples d'utilisation des limites (suite)

Exemple 3: comparez l'ordre de croissance de $\log_a(n)$ avec $\log_b(n)$

Notez que $x = \log_a(n)$ ssi $a^x = n$ ssi $x \ln(a) = \ln(n)$.

Donc $\log_a(n) = x = \ln(n) / \ln(a)$.

Donc $\log_b(n) = \ln(n) / \ln(b)$

Donc $\log_a(n) / \log_b(n) = \ln(b) / \ln(a)$ (pour tout n)

Donc $\log_a(n) \in \Theta(\log_b(n))$. Donc $O(\log_a(n)) = O(\log_b(n))$ (même ordre)

La base d'un logarithme ne change donc pas l'ordre de croissance

On utilise alors $O(\log n)$ pour désigner la classe de complexité (ordre de croissance) logarithmique

La règle de l'Hôpital

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)} \quad (\text{si cette limite existe})$$

Exemple: comparez l'ordre de croissance de $\ln(n)$ avec l'ordre de croissance de n

$$\lim_{n \rightarrow \infty} \ln(n) / n = \lim_{n \rightarrow \infty} (1/n) / 1 = 0$$

Alors $\ln(n)$ a un ordre de croissance strictement plus petit que n .
Donc $O(\log n) \subset O(n)$.

Utilisation de la notation asymptotique pour exprimer le temps d'exécution des algorithmes

- Le temps d'exécution d'un algorithme = nombre d'opérations élémentaires pour traiter une instance de taille n .
- TRUC: Puisque nous nous intéressons uniquement à son **ordre de croissance**, nous allons voir qu'il suffit d'identifier une **opération baromètre** et de compter le nombre de fois que cette opération est effectuée sur une instance de taille n .
- **Opération baromètre** = opération élémentaire qui, à une constante près, est effectuée au moins aussi souvent que n'importe quelle autre opération élémentaire de l'algorithme.
 - Nous ne sommes pas obligé de choisir une opération qui est exécutée au moins aussi souvent que n'importe quelle autre. Il suffit qu'elle le soit à une constante près (qui ne croît pas avec la taille de l'instance).
- Il est généralement assez simple d'identifier une opération baromètre
 - Ex1: pour un algorithme de tri: la comparaison entre deux valeurs (ou clés) est une opération baromètre.
 - Ex2: pour la multiplication de matrices: la multiplication de deux nombres est une opération baromètre.

Exemple du tri sélection:

```
void triSelection(std::vector<int> & x)
{
    for (int i = x.size()-1; i > 0; i--)
    {
        int pMax = i;
        for (int j = 0; j < i; j++)
        {
            if (x[j] > x[pMax])
                pMax = j;
        }
        int temp = x[i];
        x[i] = x[pMax];
        x[pMax] = temp;
    }
}
```

Baromètres
possibles

Baromètres
invalides

Exemple (suite) : calcul du temps d'exécution

Opération baromètre: la comparaison $j < i$

$$t(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$t(n) = \sum_{i=1}^{n-1} i$$

$$t(n) = 1 + 2 + \dots + (n-1)$$

$$t(n) = n(n-1) / 2$$

*Alors $t(n) \in \Theta(n^2)$
L'algorithme est
en $\Theta(n^2)$*

Pause math

- Pourquoi avons-nous $\sum_{i=0}^{n-1} i = n(n-1)/2$
- Preuve (de Carl Friedrich Gauss alors qu'il était âgé de 9 années):
 - $S = 1 + 2 + 3 + \dots + n-1$
 - $S = n-1 + n-2 + n-3 + \dots + 1$
 - Donc $2S = n + n + n + \dots + n$ (n-1 fois)
 - Donc $2S = (n-1)n$ **CQFD**.
- À l'aide de ce truc, vous pouvez trouver la valeur de n'importe quelle série arithmétique!
- (Pour les autres séries, vous pouvez consulter l'aide mémoire).

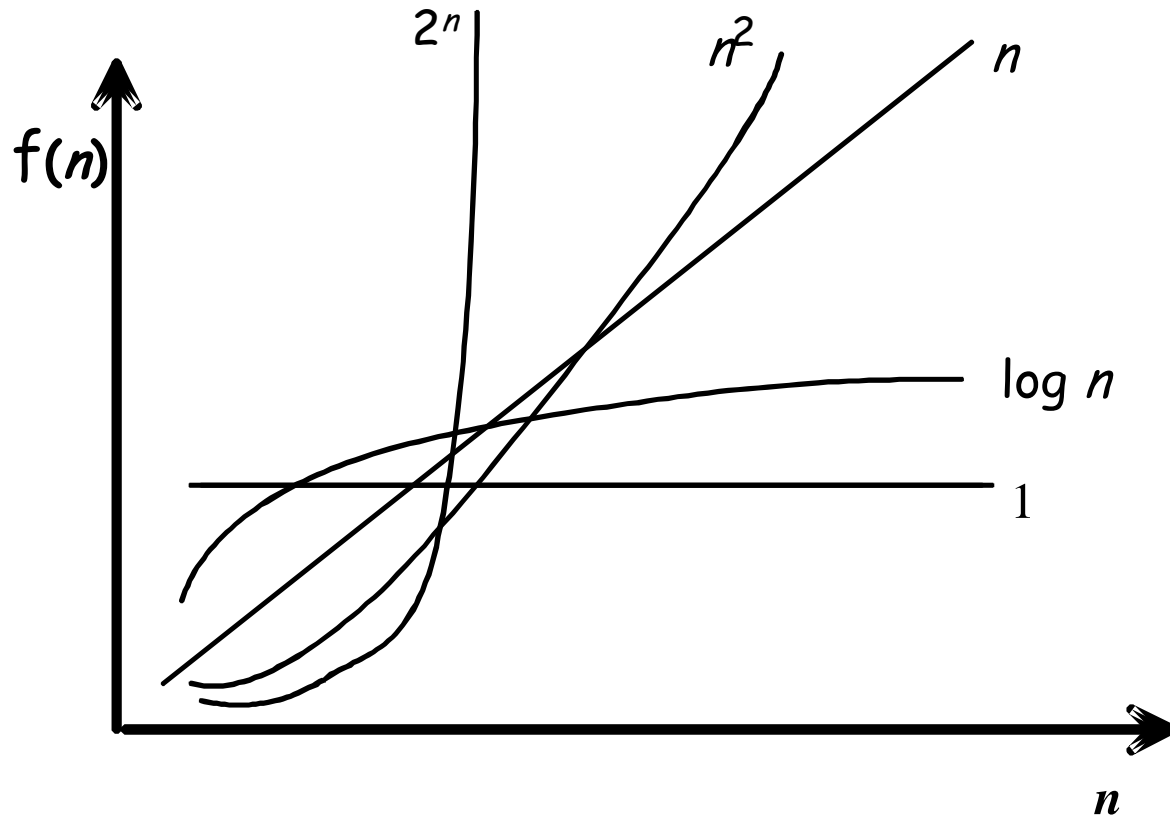
Pourquoi est-ce suffisant de compter le nombre de fois que le baromètre est exécuté?

- **Réponse:** c'est suffisant parce que la taille d'un algorithme ne dépend pas de la taille de l'instance à traiter!
- **Preuve:**
 - Soit: $N_{\text{baromètre}}(S)$ = le nombre de fois que le baromètre est exécuté sur l'instance S .
 - Soit \hat{t} = une instruction de l'algorithme qui est exécutée le plus souvent, et $N_{\hat{t}}(S)$ le nombre de fois que \hat{t} est exécutée.
 - Par définition, il existe des constantes C_1, C_2, C_3 et C_4 telles que
 - $C_1 N_{\text{baromètre}}(S) + C_2 \leq N_{\hat{t}}(S) \leq C_3 N_{\text{baromètre}}(S) + C_4$
 - Soit K = le nombre d'opérations élémentaires contenu dans l'algorithme (i.e., la «taille» de l'algorithme).
 - Soit $T(S)$ le temps d'exécution de l'algorithme sur l'instance S
 - Puisque l'instruction \hat{t} est exécutée le plus souvent, on a que
 - $N_{\hat{t}}(S) \leq T(S) \leq K N_{\hat{t}}(S)$
 - **Alors:** $C_1 N_{\text{baromètre}}(S) + C_2 \leq T(S) \leq K (C_3 N_{\text{baromètre}}(S) + C_4)$
 - Puisque K est indépendant de $n = |S|$ (taille de l'instance S) on a que $T(S) \in \Theta(N_{\text{baromètre}}(S))$. **CQFD.**

Classes de complexité pour temps d'exécution

- $O(1)$ indique un temps d'exécution constant, indépendant de n
 - $O(n)$ indique essentiellement que l'on doit effectuer un nombre constant d'opérations sur chaque donnée d'entrée
 - $O(\log n)$ implique que toutes les données n'ont pas à être traitées
 - $O(n^2)$ implique un nombre d'opérations proportionnel à n , pour chaque donnée
 - $O(n \log n)$ est typique des algorithmes diviser-pour-régner (ex: tri fusion)
 - $O(n^a)$ pour $a \geq 1$: algorithme à temps polynomial
 - $O(a^n)$ est un algorithme à temps exponentiel. Donc algorithme très lent!
 - Certains problèmes, comme celui du voyageur de commerce, n'admettent toujours pas d'algorithmes à temps polynomial..
 - $O(n!)$: algorithme plus lent qu'exponentiel...
- $O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(2^n) \subset O(10^n) \subset O(n!)$

Classes de complexité pour temps d'exécution



Classes de complexité pour temps d'exécution (suite)

- **Preuve de $O(2^n) \subset O(10^n)$:**

- $\lim_{n \rightarrow \infty} 2^n / 10^n = \lim_{n \rightarrow \infty} (2/10)^n = 0$

- **Preuve de $O(10^n) \subset O(n!)$:**

- Utilisons la formule de Stirling (fourni sur aide-mémoire):

$$\lim_{n \rightarrow \infty} n! = (2\pi n)^{1/2} (n/e)^n$$

- Donc, $\lim_{n \rightarrow \infty} (n!/10^n) = \lim_{n \rightarrow \infty} (2\pi n)^{1/2} (n/10e)^n = \infty$.

- **Preuve de $O(\log^k(n)) \subset O(n)$:**

- Par la règle de l'Hôpital:

- $\lim_{n \rightarrow \infty} (\ln^k(n)/n) = \lim_{n \rightarrow \infty} (k \ln^{k-1}(n) (1/n) / 1) = \lim_{n \rightarrow \infty} (k \ln^{k-1}(n) / n)$

- $= \lim_{n \rightarrow \infty} (k(k-1) \ln^{k-2}(n) / n) = \dots = \lim_{n \rightarrow \infty} (k! \ln^0(n) / n) = 0$.

La règle du maximum

Si

$$f_1(n) \in \Theta(g_1(n))$$

et

$$f_2(n) \in \Theta(g_2(n)),$$

alors

$$f_1(n) + f_2(n) \in \Theta(\max(g_1(n), g_2(n))).$$

Exemple: $n + n \log(n) \in \Theta(n \log n)$

Donc, lorsqu'un algorithme est constitué de deux parties, l'une s'exécutant après l'autre, le temps d'exécution est déterminé uniquement par la partie la plus lente.

Cette règle s'applique également pour O et Ω .

Exemples sur les boucles

```
void f(int n)
{
    for(int i=0; i<n; i++) cout <<"Allo\n";
}
```



Baromètre

Cet algorithme est en $\Theta(n)$

```
void f(int n)
{
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            cout << i << j << endl;
}
```



Baromètre

Cet algorithme-ci est en $\Theta(n^2)$

Autres exemples sur les boucles

```
void f(int n)
{
```

```
    for(int i=0; i<n; i++)
        for(int j=0; j<n*n; j++)
            cout << i << endl;
}
```

Baromètre

Cet algorithme est en $\Theta(n^3)$

```
void f(int n)
{
```

```
    for(int i=0; i<100000000; i++)
        cout<<"Allo\n";
}
```

Baromètre

Puisque son temps d'exécution ne dépend pas de n , même si la boucle est très longue, l'algorithme est en $O(1)$

Autres exemples sur les boucles

```
void f(int n)
{
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            cout << i << j << endl;
    for(int i=0; i<n; i++)
        cout << i << endl;
}
```

Baromètre partie 1

Baromètre partie 2

On a deux parties indépendantes: une boucle double suivie d'une boucle simple

Utilisons alors un baromètre pour chacune des parties


La double boucle s'exécute en $\Theta(n^2)$

La boucle simple s'exécute en $\Theta(n)$

Selon la règle du maximum, cet algorithme s'exécute donc en $\Theta(n^2)$

Autres exemples sur les boucles


```
void f(int n)
{
    for(int i=0; i<5340*n; i++)
        cout <<"Allo\n";
}
```



Baromètre

Puisque qu'il faut faire abstraction de toute constante multiplicative, cet algorithme est en $\Theta(n)$

```
void f(int n)
{
    for(int i=0; i<n; i++)
        for(int j=0; j<i; j++)
            cout <<"Allo\n";
}
```




Baromètre

La boucle interne s'exécute en $\Theta(i)$.
On a donc $T(n) = 0 + 1 + 2 + \dots + n-1 \in \Theta(n^2)$.

L'algorithme est donc en $\Theta(n^2)$

Autres exemples sur les boucles

```
void f(int n)
{
    for(int i=1; i<=n; i*=2)
        for(int j=1; j<=n; j++)
            cout << "Allo\n";
}
```

 Baromètre

La boucle interne s'exécute en $\Theta(n)$.

La valeur de i dans la boucle externe est multipliée par 2 à chaque itération.

Après k itérations, on aura $i = 2^k$.

Le nombre k d'itérations est donc donné par le plus petit k tel que $2^k > n$.

C'est donc le plus petit k tel que $k > \log_2(n)$.

La boucle externe fera donc $\Theta(\log n)$ itérations.

Chacune de ces itérations s'effectuent en $\Theta(n)$ à cause de la boucle interne.

L'algorithme s'exécute donc en $\Theta(n \log(n))$.

Pire cas, meilleur cas, cas moyen

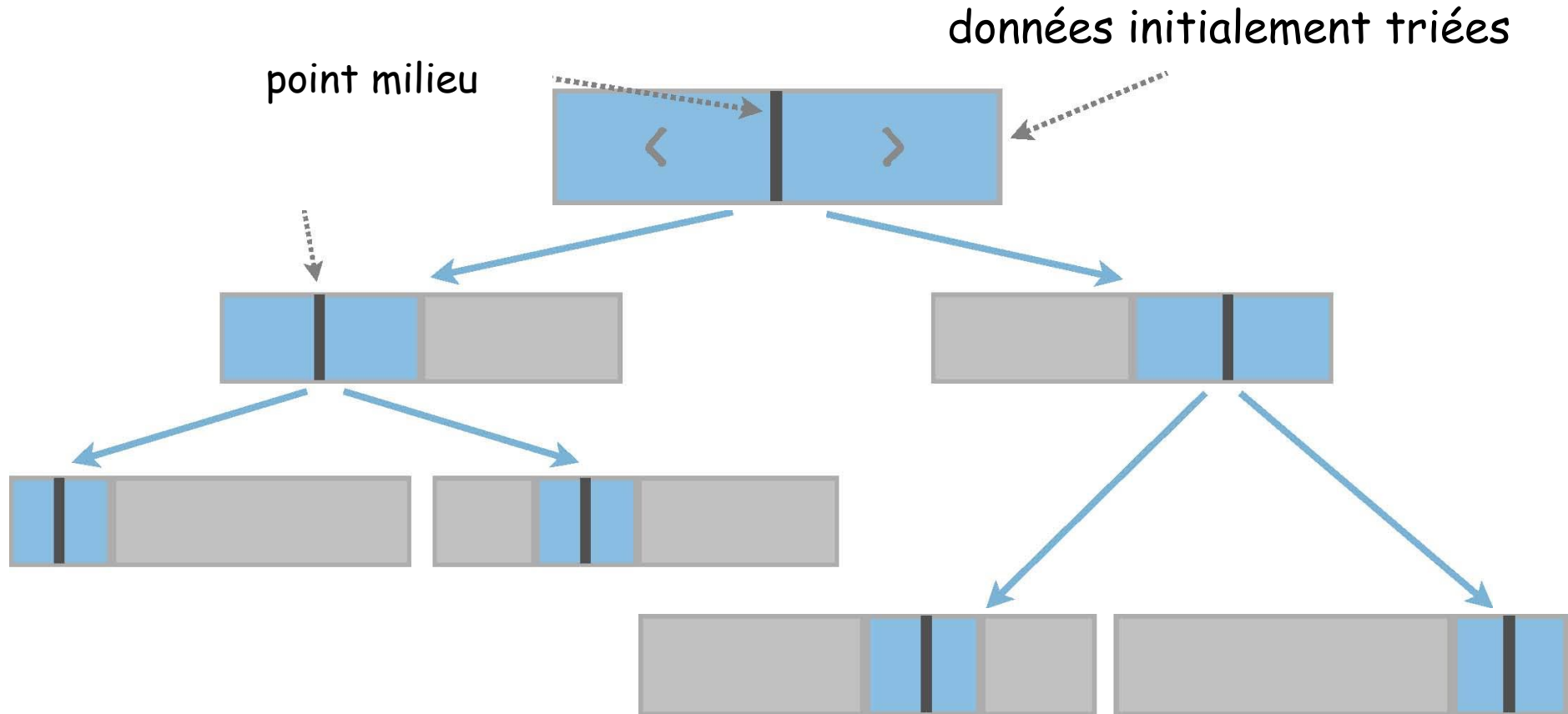
- Jusqu'à maintenant, on a supposé que le temps d'exécution est exactement le même pour toutes les instances S ayant la même taille n .
- **Dans le cas général, le temps d'exécution d'un algorithme dépend non seulement de la taille de l'instance mais également de sa nature**
 - Ex: certains algorithmes de tri sont beaucoup plus rapides lorsque les données sont quasiment triés (le tri par insertion par exemple)
- On distingue alors le temps d'exécution en **pire cas**, en **meilleur cas** et en **cas moyen** qui, eux, ne dépendent que de la taille n de S .
- **Le temps d'exécution en pire cas $T_{\text{worst}}(n)$** est le temps d'exécution maximal parmi toutes les instances S de taille n .
- **Le temps d'exécution en meilleur cas $T_{\text{best}}(n)$** est le plus petit temps d'exécution parmi toutes les instances S de taille n .
- **Le temps d'exécution moyen $T_{\text{avg}}(n)$** est la moyenne des temps d'exécution parmi toutes les instances S de taille n .
- Lorsque le temps d'exécution ne dépends que de la taille n de l'instance **(et seulement dans ces cas)**, nous avons $T_{\text{worst}}(n) = T_{\text{best}}(n) = T_{\text{avg}}(n)$.

Pire cas et meilleur cas: exemple

- Donc lorsque $|S| = n$, nous avons que $T_{\text{best}}(n) \leq T(S) \leq T_{\text{worst}}(n)$.
- **Ex: recherche séquentielle d'un élément X dans un tableau S de n éléments**
 - Puisque le tableau n'est pas trié, on n'a pas le choix d'examiner séquentiellement (du début à la fin) chaque élément du tableau.
 - Si $X =$ à l'élément examiné, alors on retourne l'index (position) de cet élément dans le tableau. Sinon, on passe à l'élément suivant.
 - Si on n'a pas trouvé X après avoir examiné tous les éléments du tableau, alors on retourne -1 (index non valide).
 - En **meilleur cas**, X se trouve à la première position $\Rightarrow T_{\text{best}}(n) \in O(1)$.
 - En **pire cas**, il faut examiner les n éléments $\Rightarrow T_{\text{worst}}(n) \in \Theta(n)$.
 - Donc, **dans tous les cas**, $T(S) \in \Omega(1)$ et $T(S) \in O(n)$ avec $n=|S|$.

Recherche binaire dans un tableau trié

- On compare X avec l'élément M au milieu du tableau trié.
- On cherche à gauche si $X < M$. On cherche à droite si $X > M$. Autrement $X = M$.



Code source

```
template <typename Comparable>
int binarySearch( const vector<Comparable> & a, const Comparable & x )
{
    int low = 0;
    int high = a.size( ) - 1;
    int mid;
    while( low <= high )
    {
        mid = ( low + high ) / 2;
        if( a[ mid ] < x ) low = mid + 1;
        else if( a[ mid ] > x ) high = mid-1;
        else return mid;
    }
    return NOT_FOUND;
}
```

Détermination du temps d'exécution

Opération baromètre: la comparaison $\text{low} \leq \text{high}$

En pire cas, le nombre $T_{\text{worst}}(n)$ d'exécutions de cette comparaison est donnée par la **relation de récurrence**:

$$T_{\text{worst}}(n) = T_{\text{worst}}(n/2) + 1$$

Avec la condition initiale: $T_{\text{worst}}(1) = \text{cste}$ (pouvant être choisie =1)

Il suffit de résoudre pour $n = 2^k$

Utilisons $T(n)$ pour désigner $T_{\text{worst}}(n)$. On a donc ici:

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

Avec la condition initiale: $T(2^0) = 1$

Méthode par substitutions à rebours

On a donc:

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + 1 \\ &= [T(2^{k-2}) + 1] + 1 \\ &= T(2^{k-2}) + 2 \\ &= T(2^{k-3}) + 3 \\ &= \dots \\ &= T(2^{k-i}) + i \quad (\text{pour tout } 0 \leq i \leq k) \\ &= T(2^0) + k \quad (\text{pour } i = k) \\ &= 1 + k \end{aligned}$$

Puisque $n = 2^k$, on a: $k = \log_2 n$

Alors:

$$T(n) = \log_2 n + 1;$$

Donc $T_{\text{worst}}(n) \in \Theta(\log n)$

Relations de récurrence

- Soit: \mathbb{N} = les entiers naturels et \mathbb{R}^+ = les réels non négatifs.
- Une **relation de récurrence** est une relation entre une fonction f
: $\mathbb{N} \rightarrow \mathbb{R}^+$ et elle-même, évaluée sur une entrée plus petite.
 - Ex: $f(n) = f(n-1) + 1$.
- Pour résoudre une récurrence, il faut spécifier des **conditions initiales**.
 - Nous en verrons plusieurs autres exemples dans ce cours.
- **Pourquoi, dans l'exemple précédent, était-il suffisant de résoudre uniquement pour $n = 2^k$?**
 - Réponse: cela vient du **théorème sur les fonctions harmonieuses**

Fonctions harmonieuses

- **Définition:** Une fonction non négative $f : \mathbb{N} \rightarrow \mathbb{R}^+$ est dite **éventuellement non décroissante** s'il existe n_0 telle que $f(n) \leq f(n+1)$ pour tout $n \geq n_0$.
- **Définition:** $f : \mathbb{N} \rightarrow \mathbb{R}^+$ est dite **harmonieuse** lorsqu'elle est éventuellement non décroissante et que $f(2n) \in \Theta(f(n))$.
 - Ex1: n^3 est harmonieuse car $(2n)^3 = 8n^3 \in \Theta(n^3)$.
 - Ex2: $\log(n)$ est harmonieuse car $\log(2n) = \log(2) + \log(n) \in \Theta(\log n)$
 - Ex3: 2^n n'est pas harmonieuse car $2^{2n} = (2^n)^2 \notin \Theta(2^n)$.
- **Théorème:** Soit $f(n)$ une fonction harmonieuse. Si $T(n)$ est éventuellement non décroissante et que $T(n) \in \Theta(f(n))$ pour $n = b^k$ avec $b \geq 2$, alors $T(n) \in \Theta(f(n))$ (pour toutes les autres valeurs de n infiniment grandes).
 - Preuve: voir manuels de cours pour IFT-3001.
 - Le théorème est également valable pour O et Ω .

Espace mémoire

La quantité d'espace mémoire utilisé est également déterminée à l'aide de l'analyse asymptotique.

On utilise donc les même techniques que celles utilisées pour déterminer le temps d'exécution

Récurtivité

Définition

Un **objet** est récursif s'il est défini à partir de lui-même.

Objets possibles:

- **Algorithmes.** Un algorithme qui est récursif est défini à partir de lui-même. C'est donc un algorithme qui s'exprime à partir de lui-même.
- **Fonctions.** Une fonction récursive, définie à partir d'elle-même, doit donc s'appeler elle-même (directement ou indirectement)
 - Ex: fonction factorielle: $n! = n \times (n-1)!$
- **Types.**
 - Liste** non vide = premier élément suivi d'une **liste**
 - Arbre** binaire = racine + sous-**arbre** droit + sous-**arbre** gauche

Algorithme Récursif

Idée générale :

La réalisation d'une tâche par un algorithme récursif correct repose sur deux éléments:

- L'algorithme résout un (ou des) cas particulier(s) du problème d'une façon simple et non récursive.
- Le cas général est solutionné par des appels récursifs successifs qui doivent converger au(x) cas particulier(s).
 - Il y a donc une **condition d'arrêt** qui permet de toujours arrêter la cascade d'appels récursifs afin de résoudre un cas particulier simple de manière non récursive.

Récurtivité

Règles qui régissent la récursivité:

- Trouver la relation de récurrence définissant la fonction en terme d'elle-même.
- Assurer-vous d'avoir identifié les cas de base pouvant être résolus sans récursivité.
- Assurez-vous de toujours progresser vers le cas de base à chaque appel récursif.
 - « Ayez la Foi » mais vérifiez que les appels récursifs progressent réellement toujours vers les cas de base.
- **Danger:** Ne jamais refaire le même travail dans des appels récursifs différents (voir la fonction récursive de Fibonacci).
 - Cela peut entraîner un temps d'exécution prohibitif.

Fonction récursive

Structure générale d'une fonction récursive

{

if(/* condition de convergence non respecté */)

Lancer une exception;

if(/*condition d'arrêt réalisé*/)

return(/*Ce qu'elle doit retourné*/);

else

appel récursif



Traitement

}

Exemple. La fonction fact() effectuant l'opération $n!$

Règle récursive : $\text{fact}(n) = n * \text{fact}(n-1)$

Condition d'arrêt : $\text{fact}(0) = 1$

Condition de convergence: $n \geq 0$



```
int fact(int n){  
    if (n < 0) throw logic_error("fact:argument < 0 ");  
    if (n == 0) return 1;  
    else return (n * fact(n-1));  
}
```

Avantages et inconvénients d'une solution récursive

Avantages

- Formulation compacte, claire et élégante.
- Solution naturelle et facile à concevoir.
- Maîtrise des problèmes dont la nature même est récursive.

Désavantages

- Possibilité de grande occupation de la mémoire.
- Temps d'exécution peut être plus long.
- Estimation parfois difficile de la profondeur maximale de la récursivité.

Danger des solutions récursives: exemple

Considérez la suite de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13 ...

Le nième nombre de Fibonacci $F(n)$ est donné par la récurrence:

$$F(n) = F(n-1) + F(n-2)$$

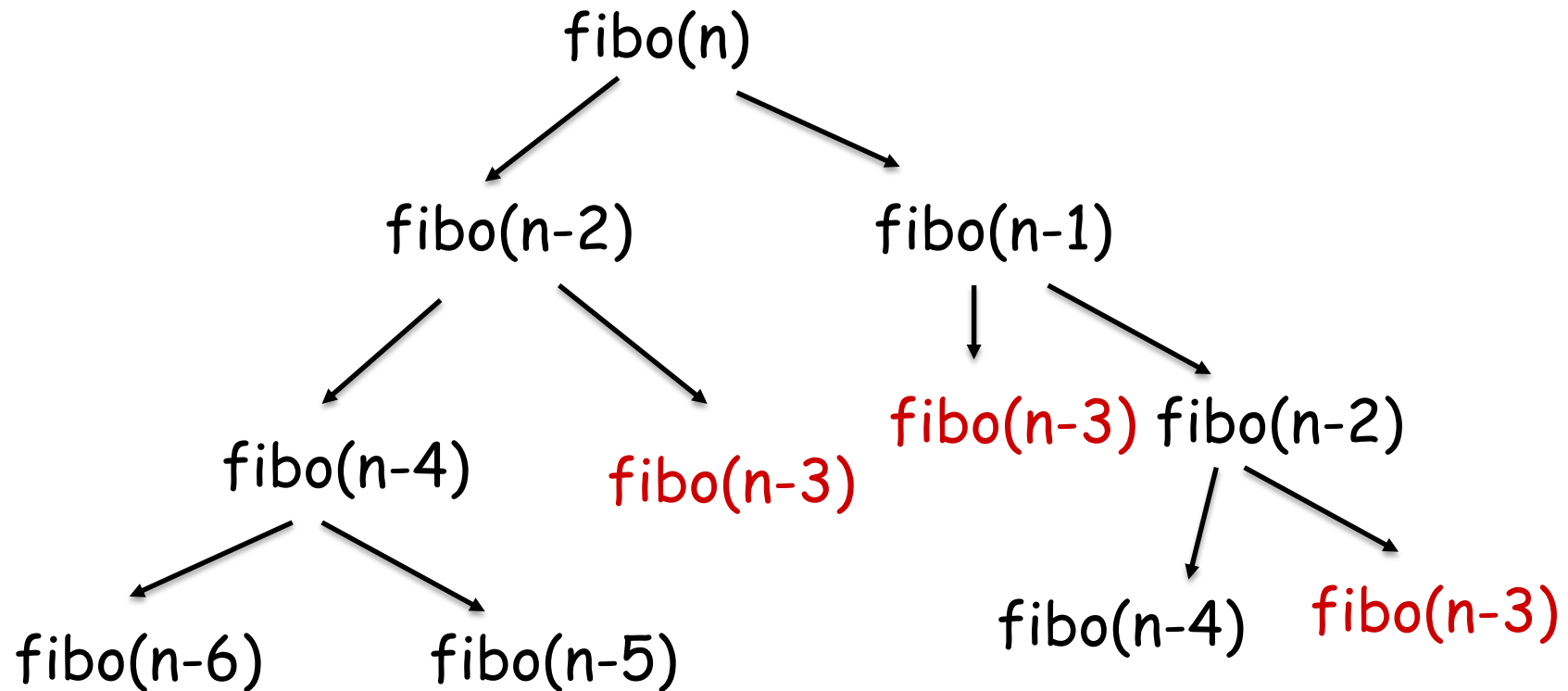
avec la conditions initiales: $F(0) = 0$ et $F(1) = 1$;

La fonction suivante permet de calculer $F(n)$ en utilisant sa définition récursive:

```
int fibo (int n)
{
    if (n<0) throw logic_error("On doit avoir n >= 0");
    if (n <= 1)
        return n;
    else
        return(fibo(n-1) + fibo(n-2));
}
```

Exécution de Fibonacci

Soit les appels effectués pour $\text{fibonacci}(n)$:



- **Règle:** *Ne jamais dupliquer le travail par des appels récurifs différents.*

Danger d'une solution récursive

- `fibonacci(n)` effectue plusieurs appels au même calcul. Par exemple pour obtenir `fibonacci(5)`, on calculera 5 fois `fibonacci(1)` et 3 fois `fibonacci(0)`. Puisque `fibonacci(n)` est exponentiel en n , le nombre de fois que `fibonacci(1)` et `fibonacci(0)` seront calculés pour un appel à `fibonacci(n)` augmente **exponentiellement** avec n .
- Le temps d'exécution de cet algorithme est donc exponentiel en n .
- Par contre, la solution non-récursive suivante est **obtenue en $\Theta(n)$** .

```
int fibonacci(int n)
{
    if (n < 0) throw logic_error("On doit avoir n >= 0");
    int* fib = new int[n+1];
    fib[0] = 0;
    if(n>0) fib[1] = 1;
    for (int i=2; i<=n; ++i)
        fib[i] = fib[i-1] + fib[i-2];
    int r = fib[n];
    delete[] fib;
    return r;
}
```

Utilisation des algorithmes récursifs

- Nous utiliserons fréquemment la récursivité dans ce cours.
- Particulièrement pour les arbres et les graphes
- Nous nous assurerons de ne pas dédoubler le travail dans les appels récursifs successifs.