Complexité

Objectifs des calculs de complexité :

- pouvoir prévoir le temps d'exécution d'un algorithme
- pouvoir comparer deux algorithmes réalisant le même traitement

Exemples :

- si on lance le calcul de la factorielle de 100, combien de temps faudra t-il attendre le résultat?

- quel algorithme de tri vaut-il mieux utiliser pour retrier un tableau où on vient de changer un élément?

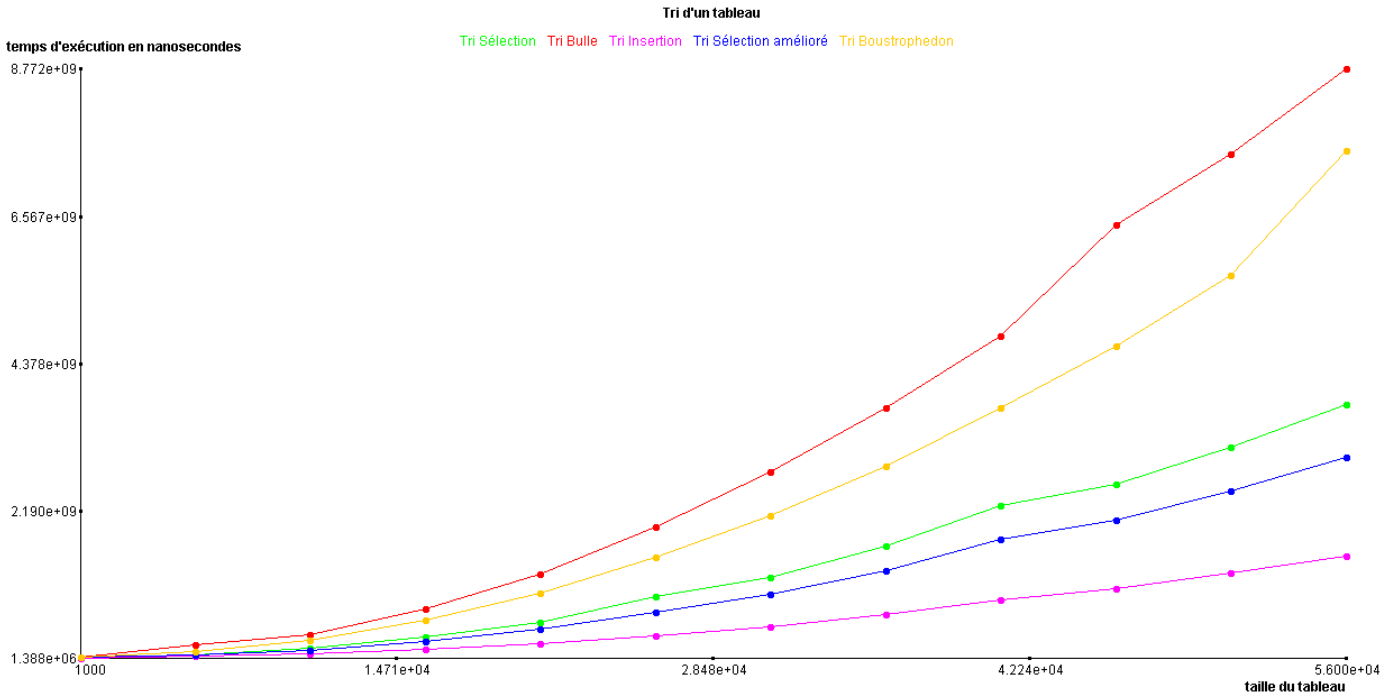
L'évaluation de la complexité peut se faire à plusieurs niveaux :

au *niveau purement algorithmique*, par l'analyse et le calcul

au *niveau de l'exécution du programme* expérimentalement

Complexité expérimentale

Il est possible d'évaluer de façon expérimentale le temps d'exécution des programmes.



Cette évaluation expérimentale dépend beaucoup des langages de programmation, ordinateurs et systèmes d'exploitation utilisés.

Pour avoir un sens, l'évaluation expérimentale requiert un étalonnage des ordinateurs utilisés.

Benchmarking

Un logiciel de benchmark (étalonnage) donne une mesure de puissance d'un ordinateur en flops (floating point operations per second).

Cette puissance varie en fonction des traitements effectués (calculs bruts, sur des entiers ou des réels, calculs liés à l'affichage, ...)

Puissance des ordinateurs grand public actuels : quelques Gigaflops (10^6 flops)

Puissance des meilleurs super-ordinateurs actuels : environ 1000 Teraflops (10^{15} flops) (cf. *www.top500.org*)

Complexité temporelle vs spatiale (1/2)

Exemple : échange de deux valeurs entières

```
// échange des valeurs de deux variables
entier x, y, z;
... // initialisation de x et y
z <- x;
x <- y;
y <- z;
```

```
// échange des valeurs de deux variables
entier x, y;
... // initialisation de x et y
x <- y-x;
y <- y-x;
x <- y+x;
```

- la première méthode utilise une variable supplémentaire et réalise 3 affectations

- la deuxième méthode n'utilise que les deux variables dont on veut échanger les valeurs, mais réalise 3 affectations et 3 opérations

Complexité temporelle vs spatiale (2/2)

L'efficacité d'un algorithme peut être évalué en temps et en espace :

- complexité en temps : évaluation du temps d'exécution de l'algorithme

- complexité en espace : évaluation de l'espace mémoire occupé par l'exécution de l'algorithme

Règle (non officielle) de l'espace-temps informatique : pour gagner du temps de calcul, on doit utiliser davantage d'espace mémoire.

On s'intéresse essentiellement à la complexité en temps (ce qui n'était pas forcément le cas quand les mémoires coutaient cher)

Paramètre de complexité (1/3)

Le paramètre de la complexité est ce qui va faire varier le temps d'exécution de l'algorithme.

Exemple : la calcul de la factorielle

```
fonction avec retour entier factorielle1(entier n)
        entier i, resultat;
début
        resultat <- 1;
        pour (i allant de 2 à n pas 1) faire
            resultat <- resultat*i;
        finpour
        retourne resultat;
fin
```

Le paramètre de complexité est la valeur de n

Paramètre de complexité (2/3)

Exemple : multiplier tous les éléments d'un tableau d'entiers par un entier donné

```
fonction sans retour multiplie(entier[] tab, int x)
       entier i;
début
       pour (i allant de 0 à tab.longueur-1 pas de 1) faire
           tab[i] <- tab[i] * x;
       finpour
fin
```

Le paramètre de complexité est la longueur du tableau tab.

Paramètre de complexité (3/3)

Exemple : faire la somme des premiers éléments de chaque ligne d'un tableau à deux dimensions

```
fonction avec retour entier sommeTeteLigne(entier[][] tab)
        entier i,s;
début
        s <- 0;
        pour (i allant de 0 à tab[0].longueur-1 pas de 1) faire
            s <- s + tab[0][i];
        finpour
        retourne s;
fin
```

Le seul paramètre de complexité est la longueur de tab[0].

Quand l'algorithme opère sur une structure multidimensionnelle, il faut bien préciser le paramètre de complexité.

Un algorithme opérant sur de telles structures peut avoir des complexités différentes selon la dimension considérée.

Calcul de complexité (1/2)

Exemple : la factorielle

```
fonction avec retour entier factorielle1(entier n)
        entier i, resultat;
début
        resultat <- 1;
        pour (i allant de 2 à n pas 1) faire
            resultat <- resultat*i;
        finpour
        retourne resultat;
fin
```

On peut fixer des temps d'exécution constants à chaque type d'instruction :

- *affectation d'entier : ae*
- *comparaison d'entier : ce*
- *opération élémentaire sur des entiers : oe*

On néglige le coût des déclarations, des affectations et du retour.

Calcul de complexité (2/2)

L'exécution de la fonction va prendre :

```
resultat <- 1;
```

ae

+

```
pour (i allant de 2 à n pas 1) faire
       resultat <- resultat*i;
finpour
```

(n-1) * (ae + oe)

*Au total, le temps d'exécution sera de la forme a*n+b*

a et b dépendent du langage de programmation et de l'ordinateur utilisés.

Complexité au mieux et au pire (1/4)

Exemple : recherche séquentielle d'un élément dans un tableau de n chaines de caractères

```
fonction avec retour booléen rechercheElement(chaine[] tab, chaine x)
        entier i;
début
        i <- 0;
        tantque (i < tab.longueur) faire
            si (tab[i] = x) alors
                retourne VRAI;
            finsi
        fintantque
        retourne FAUX;
fin
```

Le paramètre de complexité est la taille du tableau d'entrée.

Le nombre de tours de boucles varie selon que x est dans le tableau ou pas, et selon l'endroit où x est présent.

Complexité au mieux et au pire (2/4)

```
fonction avec retour booléen rechercheElement(chaine[] tab, chaine x)
        entier i;
début
        i <- 0;
        tantque (i < tab.longueur) faire
            si (tab[i] = x) alors
                retourne VRAI;
            finsi
        fintantque
        retourne FAUX;
fin
```

Si x est dans la première case du tableau *: 1 tour de boucle avec la condition tab[i]=x vraie*

Si x est dans la deuxième case du tableau *: 1 tour de boucle avec la condition tab[i]=x fausse et 1 tour de boucle avec la condition tab[i]=x vraie*

...

Si x est dans dernière case du tableau *: tab.longueur-1 tours de boucle avec la condition tab[i]=x fausse et 1 tour de boucle avec la condition tab[i]=x vraie*

Si x n'est pas dans le tableau *: tab.longueur tours de boucle avec la condition tab[i]=x fausse*

Complexité au mieux et au pire (3/4)

Lorsque, pour une valeur donnée du paramètre de complexité, le temps d'exécution varie selon les données d'entrée, on peut distinguer :

- La complexité au pire : temps d'exécution maximum, dans le cas le plus défavorable.

- La complexité au mieux : temps d'exécution minimum, dans le cas le plus favorable (en pratique, cette complexité n'est pas très utile).

- La complexité moyenne : temps d'exécution dans un cas médian, ou moyenne des temps d'exécution.

Le plus souvent, on utilise la complexité au pire, car on veut borner le temps d'exécution.

Complexité au mieux et au pire (4/4)

```
fonction avec retour booléen rechercheElement(chaine[] tab, chaine x)
        entier i;
début
        i <- 0;
        tantque (i < tab.longueur) faire
            si (tab[i] = x) alors
                retourne VRAI;
            finsi
            i <- i + 1;
        fintantque
        retourne FAUX;
fin
```

n = la taille du tableau, ae = affectation d'entier, ce = comparaison d'entier, oe=opération sur les entiers.

Complexité au pire (x n'est pas dans le tableau) : ae+n(2*ce+oe+ae)*

*Complexité au mieux (x est dans la première case du tableau) : ae+2*ce*

Complexité en moyenne : considérons qu'on a 50% de chance que x soit dans le tableau, et 50% qu'il n'y soit pas, et, s'il y est sa position moyenne est au milieu. Le temps d'exécution est [(ae+n(2*ce+oe+ae)) + (ae+(n/2)*(2*ce+oe+ae))] /2, de la forme a*n+b (avec a et b constantes)*

Complexité asymptotique (1/4)

Calculer la complexité de façon exacte n'est pas raisonnable vu la quantité d'instructions de la plupart des programmes et n'est pas utile pour pouvoir comparer deux algorithmes.

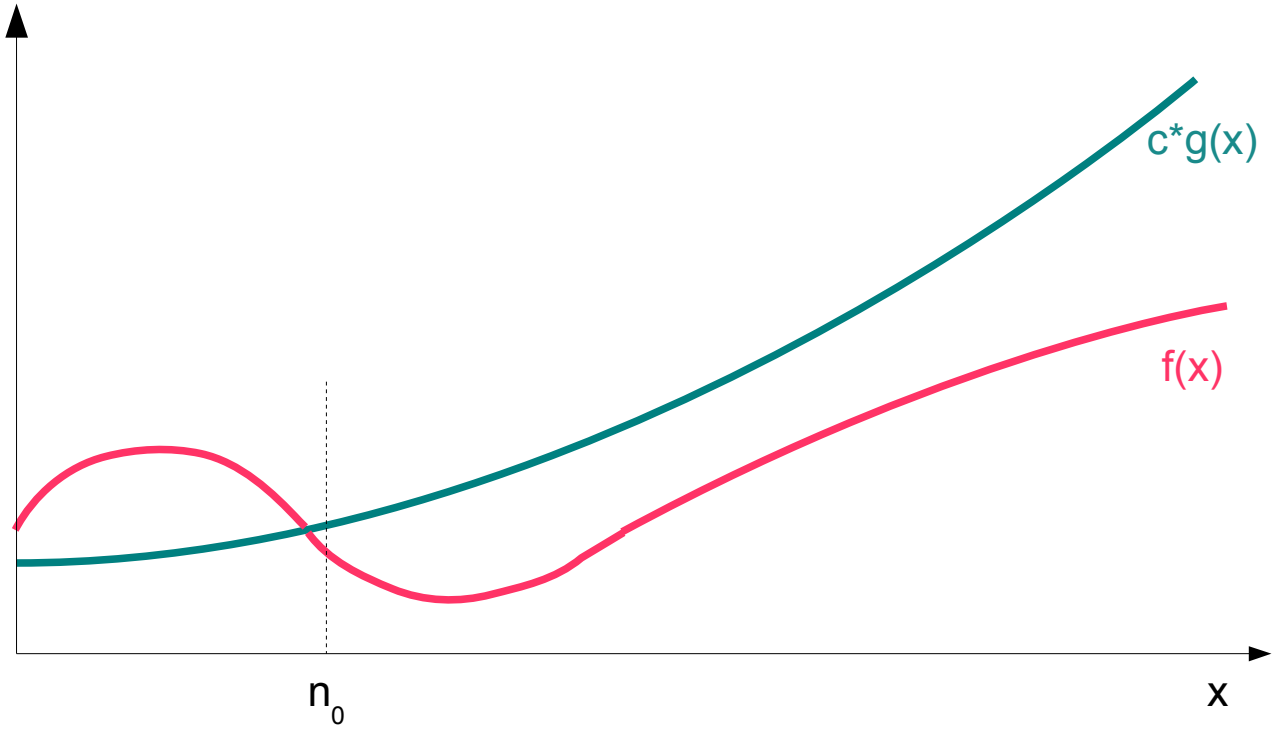
Première approximation : on ne considère souvent que la complexité au pire

Deuxième approximation : on ne calcule que la forme générale de la complexité

Troisième approximation : on ne regarde que le comportement asymptotique de la complexité

Domination asymptotique (1/2)

f et g étant des fonctions, f = O(g) s'il existe des constantes c>0 et n_0 telles que f(x) < c*g(x) pour tout x > n_0



f = O(g) signifie que f est dominée asymptotiquement par g.

Domination asymptotique (2/2)

La notation O, dite notation de Landau, vérifie les propriétés suivantes :

- si f=O(g) et g=O(h) alors f=O(h)
- si f=O(g) et k un nombre, alors k*f=O(g)
- si f1=O(g1) et f2=O(g2) alors f1+f2 = O(g2+g2)
- si f1=O(g1) et f2=O(g2) alors f1*f2 = O(g2*g2)

Exemples de domination asymptotique :

$x = O(x^2)$ car pour $x>1$, $x<x^2$ *$x^2 = O(x^3)$ car pour $x>1$, $x^2<x^3$*

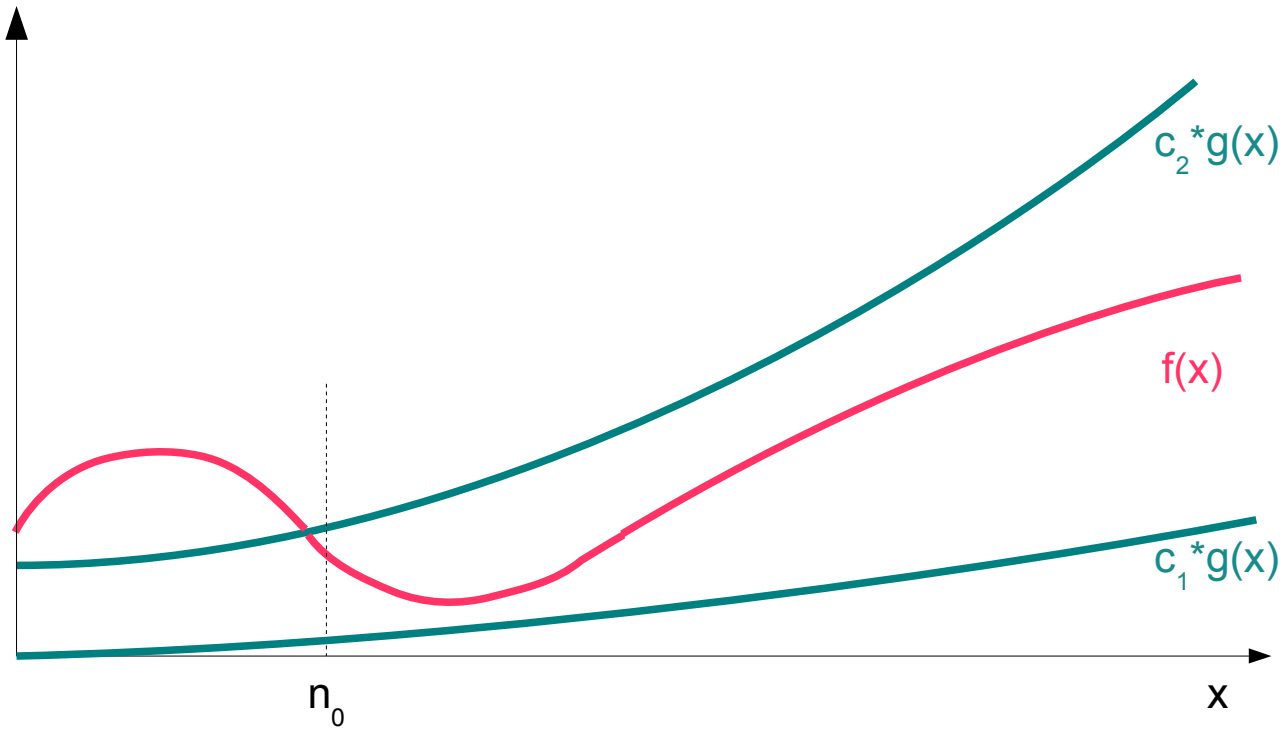
$100*x = O(x^2)$ car pour $x>100$, $x<x^2$ *$ln(x) = O(x)$ car pour $x>0$, $ln(x)<x$*

si $i>0$, $x^i = O(e^x)$ car pour x tel que $x/ln(x)>i$, $x^i<e^x$

Notation Ω : f = Ω(g) s'il existe des constantes c>0 et n_0 telles que $f(x) \geq c*g(x)$ pour tout $x \geq n_0$

Equivalence asymptotique

f et g étant des fonctions, f = Θ(g) s'il existe des constantes c_1, c_2, strictement positives et n_0 telles que c1*g(x) ≤ f(x) ≤ c2*g(x) pour tout x ≥ n_0



Classes de complexité (1/3)

O(1) : complexité constante, pas d'augmentation du temps d'exécution quand le paramètre croit

O(log(n)) : complexité logarithmique, augmentation très faible du temps d'exécution quand le paramètre croit. *Exemple : algorithmes qui décomposent un problème en un ensemble de problèmes plus petits (dichotomie).*

O(n) : complexité linéaire, augmentation linéraire du temps d'exécution quand le paramètre croit (si le paramètre double, le temps double). *Exemple : algorithmes qui parcourent séquentiellement des structures linéaires.*

O(nlog(n)) : complexité quasi-linéaire, augmentation un peu supérieure à O(n). Exemple : algorithmes qui décomposent un problème en d'autres plus simples, traités indépendaments et qui combinent les solutions partielles pour calculer la solution générale.

Classes de complexité (2/3)

$O(n^2)$: complexité quadratique, quand le paramètre double, le temps d'exécution est multiplié par 4. *Exemple : algorithmes avec deux boucles imbriquées.*

$O(n^i)$: complexité polynomiale, quand le paramètre double, le temps d'exécution est multiplié par 2^i. *Exemple : algorithme utilisant i boucles imbriquées.*

$O(i^n)$: complexité exponentielle, quand le paramètre double, le temps d'exécution est élevé à la puissance 2.

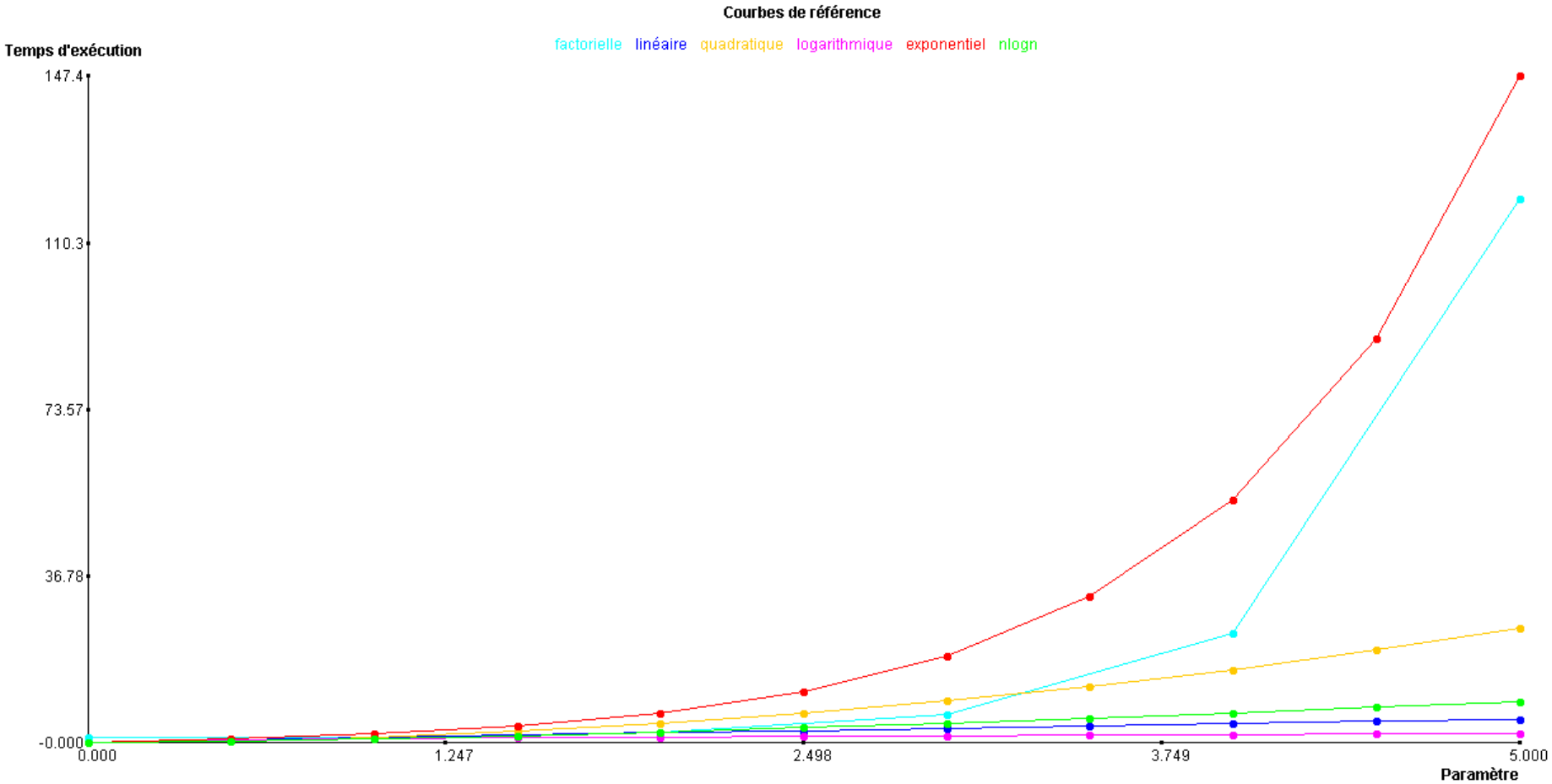
$O(n!)$: complexité factorielle, asymptotiquement équivalente à n^n

Encore pire : la fonction d'Ackerman

Ack(m,n) = n+1 si m = 0
Ack(m,n) = Ack(m-1,1) si n=0 et m > 0
Ack(m,n) = Ack(m-1, Ack(m,n-1)) sinon

Ack(0,0) = 1, Ack(1,1) = 3, Ack(2,2) = 7, Ack(3,3) = 61, Ack(4,4) est supérieur à 10^{80}

Classes de complexité (3/3)



Les algorithmes de complexité polynomiale ne sont utilisables que sur des données réduites, ou pour des traitements ponctuels

Les algorithmes exponentiels ou au delà ne sont pas utilisables en pratique

Complexité et temps d'exécution

Temps de calcul pour des données de taille 1 million

flops \ complexité	ln(n)	n	n^2	2^n
10^6	0,013ms	1s	278 heures	10000 ans
10^9	0,013µs	1ms	¼ heure	10 ans
10^{12}	0,013ns	1µs	1s	1 semaine

Loi de Moore (empirique) : à coût constant, la rapidité des processeurs double tous les 18 mois (les capacités de stockage suivent la même loi).

Constat : le volume des données stockées dans les systèmes d'informations augmente de façon exponentielle.

Conclusion : il vaut mieux optimiser ses algorithmes qu'attendre des années qu'un processeur surpuissant soit inventé.

Recherche séquentielle

```
fonction avec retour booléen rechercheElement(chaine[] tab, chaine x)
        entier i;
début
        i <- 0;
        tantque (i < tab.longueur) faire
            si (tab[i] = x) alors
                retourne VRAI;
            finsi
            i <- i + 1;
        fintantque
        retourne FAUX;
fin
```

n = la taille du tableau, ae = affectation d'entier, ce = comparaison d'entier, oe = opération sur les entiers.

Complexité au pire (x n'est pas dans le tableau) : ae+n(2*ce+ae+oe) = O(n)*

*Complexité au mieux (x est dans la première case du tableau) : ae+2*ce = O(1)*

Complexité en moyenne : considérons qu'on a 50% de chance que x soit dans le tableau, et 50% qu'il n'y soit pas, et, s'il y est sa position moyenne est au milieu. Le temps d'exécution est [(ae+n(2*ce+oe+ae)) + (ae+(n/2)*(2*ce+oe+ae))] /2, de la forme a*n+b (avec a et b constantes) = O(n)*

Recherche dichotomique (1/2)

```
fonction avec retour booléen rechercheDicho(chaine[] tab, chaine x)
        entier i, j;
début
        i <- 0;
        j <- tab.longueur-1;
        tantque (i <= j) faire
            si (tab[(j+i)/2] = x) alors retourne VRAI;
            sinon
                si (tab[(j+i)/2] > x) alors j <- (j+i)/2 - 1;
                sinon i <- (j+i)/2 + 1;
                finsi
            finsi
        fintantque
        retourne FAUX;
fin
```

Le paramètre de complexité est la longueur du tableau, qu'on appelle n.

Cas au pire : x n'est pas dans le tableau.

La longueur de la partie du tableau comprise entre i et j est d'abord n, puis n/2, puis n/4, ..., jusqu'à ce que $n/2^t = 1$. Le nombre de tours de boucles est donc un entier t tel que $n/2^t = 1$ soit $2^t = n$ soit $t*\log(2) = \log(n)$ soit $t = \log_2(n)$

Recherche dichotomique (2/2)

```
fonction avec retour booléen rechercheDicho(chaine[] tab, chaine x)
        entier i, j;
début
        i <- 0;
        j <- tab.longueur-1;
        tantque (i <= j) faire
            si (tab[(j+i)/2] = x) alors retourne VRAI;
            sinon
                si (tab[(j+i)/2] > x) alors j <- (j+i)/2 - 1;
                sinon i <- (j+i)/2 + 1;
                finsi
            finsi
        fintantque
        retourne FAUX;
fin
```

La complexité au pire de la recherche dichotomique est (ce + 2 cc + 3*oe)*$log_2(n)$ avec ce = comparaison d'entier, cc = comparaison de chaine, oe = opération élémentaire.*

Complexité asymptotique : O(log(n))

Conclusion : pour des données de grande taille, si le tableau est trié, il vaut mieux utiliser la recherche dichotomique

Factorielle récursive (1/2)

Exemple : calcul récursif de la factorielle

```
// cette fonction renvoie n! (n est supposé supérieur ou égal à 1)
fonction avec retour entier factorielle2(entier n)
début
       si (n = 1) alors
           retourne 1;
        sinon
           retourne n*factorielle2(n-1);
        finsi
fin
```

Paramètre de complexité : la valeur de n

Il n'y a qu'un seul cas d'exécution (pas de cas au pire ou au mieux)

Si n ≠ 1, le calcul de la factorielle de n coûte une comparaison d'entiers, le calcul de la factorielle de n-1 et une multiplication d'entiers

Si n = 1, le calcul de la factorielle coûte une comparaison d'entiers

Factorielle récursive (2/2)

On pose une équation de récurrence : appelons c(n) la complexité

$$c(n) = ce + c(n-1) + oe \quad si\ n \neq 1$$

$$c(1) = ce$$

*On résoud cette équation de récurrence : c(n) = n*ce + (n-1)*oe = O(n)*

La complexité de la factorielle récursive est donc linéaire, comme celle de la factorielle itérative.

A l'exécution, la fonction récursive est un peu moins rapide (pente de la droite plus forte) du fait des appels récursifs.

Complexité et récursivité

En général, dérécursiver un algorithme ne change pas la forme de sa complexité, pas plus que passer en récursivité terminale!

Il existe diverses techniques pour la résolution des équations de récurrence (méthode des fonctions génératrices et décomposition des fractions rationnelles, transformée en Z, …).

Théorème : soit T(n) une fonction définie par l'équation de récurrence suivante, où b ≥ 2, k ≥ 0, a > 0, c > 0 et d >0 :

$$T(n) = a*T(n/b) + c*n^k$$

si $a > b^k$ alors $T(n) \Theta (n^{\log_b(a)})$

si $a = b^k$ alors $T(n) \Theta (n^k * \log(n))$

si $a < b^k$ alors $T(n) \Theta n^k$

Résolution des récurrences (1/2)

Equation : $c(n) = c(n-1) + b$

Solution : $c(n) = c(0) + b*n = O(n)$

Exemples : factorielle, recherche séquentielle récursive dans un tableau

Equation : $c(n) = a*c(n-1) + b$, $a \neq 1$

Solution : $c(n) = a^n*(c(0) – b/(1-a)) + b/(1-a) = O(a^n)$

Exemples : répétition a fois d'un traitement sur le résultat de l'appel récursif

Equation : $c(n) = c(n-1) + a*n + b$

Solution : $c(n) = c(0) + a*n*(n+1)/2 + n*b = O(n^2)$.

Exemples : traitement en coût linéaire avant l'appel récursif, tri à bulle

Equation : $c(n) = c(n/2) + b$

Solution : $c(n) = c(1) + b*\log_2(n) = O(\log(n))$

Exemples : élimination de la moitié des éléments en temps constant avant l'appel récursif, recherche dichotomique récursive

Résolution des récurrences (2/2)

Equation : $c(n) = a*c(n/2) + b$, $a \neq 1$

Solution : $c(n) = n^{\log2(a)} *(c(1) – b/(1-a)) + b/(1-a) = O(n^{\log2(a)})$

Exemples : répétition a fois d'un traitement sur le résultat de l'appel récursif dichotomique

Equation : $c(n) = c(n/2) + a*n +b$

Solution : $c(n) = O(n)$

Exemples : traitement linéaire avant l'appel récursif dichotomique

Equation : $c(n) = 2*c(n/2) + a*n + b$

Solution : $c(n) = O(n*\log(n))$

Exemples : traitement linéaire avant double appel récursif dichotomique, tri fusion

Le piège de Fibonacci (1/2)

```
fonction avec retour entier fibo(entier n)
début
        si ((n = 0) OU (n = 1)) alors
            retourne 1;
        sinon
            retourne fibo(n-1) + fibo(n-2);
        finsi
fin
```

Paramètre de complexité : la valeur de n

Un seul cas d'exécution

La complexité c(n) vérifie l'équation : $c(n) = a + c(n-1) + c(n-2)$ si $n>1$

$c(1) = 1$

$c(0) = 1$

Solution de l'équation : $c(n) = \Theta(\Phi^n)$ où $\Phi = (1+\sqrt{5})/2$

Le piège de Fibonacci (2/2)

```
fonction avec retour entier fibo(entier n)
        entier a, b, c, i;
début
        si ((n = 0) ou (n = 1)) alors retourne 1;
        sinon
            a <- 1; b <- 1;
            pour (i allant de 2 à n pas 1) faire
                c <- b; b <- a + b; a <- c;
            finpour
            retourne b;
        finsi
fin
```

Paramètre de complexité : la valeur de n

Un seul cas d'exécution

Pour n≠0, il y a (n-1) tours de boucles, donc la complexité vaut a*(n-1)+b avec a et b constantes donc la complexité est en O(n).

Complexité du tri à bulle

```
fonction sans retour triBulle(entier[] tab){
     entier i,j,temp;
début
     pour (i allant de tab.longueur-2 à 1 pas -1) faire
          pour (j allant de 0 à i pas 1) faire
               si (tab[j] > tab[j+1]) alors
                    temp <- tab[j]; tab[j] <- tab[j+1]; tab[j+1] <- temp;
               finsi
          finpour
     finpour
fin
```

Paramètre de complexité : la taille du tableau

Cas au pire : les éléments sont triés dans l'ordre inverse de celui voulu

Complexité au pire : remonter le premier élément nécessite (n-1) tours de la boucle imbriquée, remonter le deuxième nécessite (n-2) tours, etc. Donc le nombre de tours total est (n-1) + (n-2) + … + 1 = n*(n-1)/2 soit $O(n^2)$.

Remarque : les tris par sélection et par insertion sont aussi quadratiques. Mais il existe des algorithmes de tri quasi-linéaires.