Complexité (1/2)

- Problématique : on veut pouvoir mesurer l'efficacité d'un algorithme, ce qu'on appelle sa complexité
  - pour pouvoir prévoir son temps d'exécution
  - pour pouvoir estimer les ressources qu'il va mobiliser dans une machine lors de son exécution (place occupée en mémoire en particulier)
  - pour pouvoir le comparer avec un autre qui fait le même traitement d'une autre façon, de manière à choisir le meilleur
- L'évaluation de la complexité peut se faire à plusieurs niveaux
  - au *niveau purement algorithmique*, par l'analyse et le calcul
  - au *niveau du programme*, par l'analyse et le calcul
  - au *niveau de l'exécution du programme* expérimentalement

Complexité (2/2)

- Jusqu'aux années 70, seule la mesure expérimentale de la complexité d'un algorithme était (parfois) effectuée
- Cette évaluation expérimentale dépendait énormément des machines mais permettait de comparer l'efficacité de différents algorithmes si on les écrivait dans un même langage et qu'on les faisait tourner sur une même machine
- Si on les faisait tourner sur des machines différentes, il fallait évaluer la puissance des machines
  - cette puissance dépend du matériel mais aussi du système d'exploitation
  - cette puissance varie en fonction des traitements effectués (calculs bruts, sur des entiers ou des réels, calculs liés à l'affichage, ...)

Benchmark

- Benchmark (point de référence) : batterie de tests consistant à faire tourner certains programmes sur une machine pour évaluer sa puissance
  - un benchmark est orienté vers certains types de calculs
  - la puissance d'une machine s'exprime généralement en flops (floating point operations per second)
- 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*)
- Exemples de benchmark :
  - Sandra : benchmark généraliste sous Windows (www.sisoftware.co.uk)
  - Sciencemark : benchmark orienté calculs scientifiques sous Windows
  - Benchmarks sous Linux : *lbs.sourceforge.net*

Echange de deux valeurs

- *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 en temps et en espace

- Plusieurs complexités peuvent être évaluées :
  - complexité en temps : il s'agit de savoir combien de temps prendra l'exécution d'un algorithme
  - complexité en espace : il s'agit de savoir combien d'espace mémoire occupera l'exécution de l'algorithme
- Souvent, si un algorithme permet de gagner du temps de calcul, il occupe davantage de place en mémoire (mais un peu d'astuce permet parfois de gagner sur les deux tableaux)
- Généralement, on s'intéresse essentiellement à la complexité en temps (ce qui n'était pas forcément le cas quand les mémoires coutaient cher)

Recherche séquentielle (1/2)

- *Exemple : recherche séquentielle d'un élément dans un tableau trié de n chaines de caractères*
  - *complexité en espace : la place d'un entier (en plus des paramètres)*
  - *complexité en temps : à chaque tour de boucle, on fait :*
    - *une comparaison d'entiers pour le test de boucle*
    - *2 comparaisons de chaines (1 seule si on est sur l'élément recherché)*
    - *une incrémentation d'entier (sauf si on est sur l'élément recherché)*

```
fonction avec retour booléen rechercheElement2(chaine[] tab, chaine x)
  entier i;
début
  i <- 0;
  tantque (i < tab.longueur) faire
      si (tab[i] = x) alors retourne VRAI;
      sinon
          si (tab[i] > x) alors retourne FAUX;
          sinon i <- i + 1;
          finsi
      finsi
  fintantque
  retourne FAUX;
fin
```

Complexité algorithmique (2/2)

- *Combien fait-on de tours de boucle?*
  - *appelons cc le temps mis pour comparer deux chaines, ce le temps mis pour comparer deux entiers et ie le temps mis pour incrémenter un entier*
  - *si l'élément n'est pas dans le tableau, la recherche séquentielle va faire n tours de boucle et la complexité totale sera donc de (2*cc+ce+ie)*n*
  - *si l'élément est en ieme position, la recherche fera i tours de boucles, la complexité totale sera donc de (2*cc+ce+ie)*(i-1) + (cc+ce)*
  - *n et i peuvent varier, cc et ie sont des constantes*
  - *i peut varier entre 1 et n, en moyenne, il vaudra n/2. En moyenne, la complexité en cas de succès sera donc environ de (2*cc+ce+ie)*n/2*
  - *cc, ce et ei ne dépendent pas uniquement de l'algorithme mais également du langage utilisé, de la machine sur laquelle on fait tourner le programme*

Complexité au pire

- La complexité n'est pas la même selon les déroulements du traitement
  - complexité au pire : complexité maximum, dans le cas le plus défavorable
  - complexité en moyenne : il s'agit de la moyenne des complexités obtenues selon les issues du traitement
  - complexité au mieux : complexité minimum, dans le cas le plus favorable. En pratique, cette complexité n'est pas très utile
- Le plus souvent, on utilise la complexité au pire, car on veut borner le temps d'exécution
- Si on veut comparer les algorithmes indépendamment des implémentations et des machines, on ne peut comparer que la forme générale de la complexité, en particulier la façon dont elle évolue selon la taille des données

Calcul de la factorielle

- *Exemple : calcul de la factorielle d'un entier n*
  - *complexité en espace : la place de deux entiers (en plus du paramètre)*
  - *complexité en temps : à chaque tour de boucle, on fait :*
    - *une comparaison d'entiers pour le test de boucle et une incrémentation*
    - *une multiplication d'entiers et une affectation d'entier*
    - *il y aura n-1 tours de boucle*
    - *si ce est le temps mis pour comparer deux entiers, ie celui mis pour l'incrémentation, me celui mis par la multiplication et ae celui mis pour l'affectation, la complexité sera donc de (n-1)*(ce + ie + me + ae) + ae*

```
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
```


Paramètre de la complexité

- La complexité d'un algorithme est calculée en fonction d'un paramètre par rapport auquel on veut calculer cette complexité
- Pour un algorithme qui opère sur une structure de données (tableau, ...), la complexité est généralement exprimée en fonction d'une dimension de la structure
  - dans le cas où l'algorithme prend en entrée une structure linéaire, à une seule dimension, il n'y a pas d'ambiguité
  - dans le cas où l'algorithme prend en entrée une structure à plusieurs dimensions (tableau multidimensionnel, arbre, graphe, ...), il faut préciser en fonction de quelle dimension on calcule la complexité
  - dans le cas où l'algorithme prend en entrée une structure à plusieurs dimensions, l'algorithme peut avoir des complexités différentes selon la dimension considérée
- Pour un algorithme qui opère sur un nombre, la complexité est généralement exprimée en fonction de la valeur du nombre

Recherche dichotomique (1/3)

- *Exemple : recherche dichotomique dans un tableau trié de n chaines de caractères*
  - *on coupe le tableau en deux et on cherche l'élément dans une des deux parties en répétant le même traitement (on utilise donc deux entiers pour mémoriser les bornes inférieure et supérieur de l'intervalle d'indices considéré)*

```
fonction avec retour booléen rechercheElement3(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
```

  - *complexité en espace : la place de deux entiers*

Recherche dichotomique (2/3)

- complexité en temps : à chaque tour de boucle, on fait
  - une comparaison d'entiers pour le test de boucle
  - 2 comparaisons de chaines (1 seule si on est sur l'élément recherché)
  - 3 opérations élémentaires sur les entiers et une affectation (sauf si on est sur l'élément recherché)
  - au pire, 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 $t*\log(2) = \log(n)$ ou $t = \log_2(n)$
  - finalement, la complexité au pire de la recherche dichotomique sera $\log_2(n)*(ce + 2* cc + 3*oi + aff)$
- c'est plutôt compliqué, et la plupart du temps, pour comparer la complexité de deux algorithmes, on utilise une approximation de la complexité

Recherche dichotomique (3/3)

- *Exemple : recherche d'un élément dans un tableau*
  - *la complexité au pire de la recherche séquentielle est de la forme $n*k_s$ où k_s est une constante*
  - *la complexité au pire de la recherche dichotomique est de la forme $log(n)*k_d$ où k_d est une constante*
  - *on sait qu'il existe une valeur v telle que pour tout $x > v$, $log(x)*k_d < x*k_s$*
  - *la recherche dichotomique est donc préférable dès que la taille du tableau dépasse cette valeur v*
  - *en pratique, on ne se préocupe même pas de la valeur v, on privilégie toujours un algorithme dont la complexité évolue selon le logarithme du paramètre par rapport à un algorithme dont la complexité évolue selon le paramètre*

Complexité approchée

- Calculer la complexité de façon exacte
  - n'est pas raisonnable vu la quantité d'instructions de la plupart des programmes
  - 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, qui indique la façon dont elle évolue en fonction d'un paramètre
- Troisième approximation : on ne regarde que le **comportement asymptotique** de la complexité, quand la valeur du paramètre devient “assez” grande

Notation O (1/2)

- f et g étant des fonctions, g est dite en O(f) s'il existe des constantes c>0 et n_0 telles que g(x) < c*f(x) pour tout x > n_0
  - on note g = O(f) ou g(x) = O(f(x))
  - on dit que g est dominée asymptotiquement par f
  - cette notation est appelée notation de Landau

- Attention : il ne s'agit que d'une borne supérieure, et à partir d'un certain rang, cela indique juste que g ne croit pas plus vite que f à partir de ce rang (mais rien n'indique qu'elle croit moins vite, ni qu'elle croit aussi vite)

- *Exemples :*
  - $x = O(x^2)$ car pour x>1, $x<x^2$
  - $100*x = O(x^2)$ car pour x>100, $x<x^2$
  - $ln(x) = O(x)$ car pour x>0, $ln(x)<x$
  - $O(x) = O(x^2)$
  - $x^2 = O(x^3)$ car pour x>1, $x^2<x^3$
  - $\forall i>0$, $x^i = O(e^x)$ car pour x tel que $x/ln(x)>i$, $x^i<e^x$
  - $O(x^2) \neq O(x)$

Notation O (2/2)

- La notation de Landau vérifie les propriétés suivantes :
  - si g=O(f) et f=O(h) alors g=O(h)
  - si g=O(f) et k un nombre, alors k*g=O(f)
  - 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)

- Il existe d'autres notations pour décrire le comportement asymptotique des fonctions :
  - f domine asymptotiquement g : f(n) = Ω(g(n)) si et seulement s'il existe des constantes c>0 et n_0 telles que $f(n) \geq c*g(n)$ pour tout $n \geq n_0$
  - f est asymptotiquement équivalente à g : f(n) = Θ (g(n)) si et seulement s'il existe des constantes c_1, c_2 strictement positives et n_0 telles que $c_1*g(n) \leq f(n) \leq c_2*g(n)$ pour tout $n \geq n_0$

Types de complexité (1/3)

- Complexité constante (en O(1)) : pas d'augmentation du temps d'exécution quand le paramètre croit
- Complexité logarithmique (en O(log(n))) : augmentation très faible du temps d'exécution quand le paramètre croit. C'est typiquement la complexité des algorithmes qui décomposent un problème en un ensemble de problèmes plus petits (par exemple la dichotomie)
- Complexité linéaire (en O(n)) : augmentation linéraire du temps d'exécution quand le paramètre croit (si le paramètre double, le temps double). C'est typiquement le cas des algorithmes qui parcourent séquentiellement des structures linéaires (tableaux, listes, ...)
- Complexité quasi-linéaire (en O(nlog(n))) : augmentation un peu supérieure à O(n). C'est typiquement le cas des algorithmes qui décomposent un problème en d'autres plus simples, traités indépendaments et qui combinent les solutions partielles pour calculer la solution générale

Types de complexité (2/3)

- Complexité quadratique (en $O(n^2)$) : quand le paramètre double, le temps d'exécution est multiplié par 4. C'est typiquement la complexité des algorithmes avec deux boucles imbriquées. Cette complexité est acceptable uniquement pour des données de petite taille.
- Complexité polynomiale (en $O(n^i)$) : quand le paramètre double, le temps d'exécution est multiplié par 2^i. Un algorithme utilisant i boucles imbriquées est polynomial. Cette complexité est inacceptable pour des algorithmes utilisés intensivement.
- Complexité exponentielle (en $O(i^n)$) : quand le paramètre double, le temps d'exécution est élevé à la puissance 2. Cette complexité est inacceptable pour des algorithmes utilisés intensivement.
- Complexité factorielle (en O(n!)) : cette complexité est asymptotiquement équivalente à n^n (formule de Stirling). Un algorithme de complexité factorielle ne sert à rien.

Types de complexité (3/3)

- Relations asymptotiques entre les complexités :
  $O(1) = O(log) = O(n) = O(n*log(n)) = O(n^2) = O(n^3) = O(2^n) = O(n!)$

- On peut imaginer encore pire que la factorielle : la fonction d'Ackerman est une de celles qui croissent le plus vite

  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}

- Cette fonction est utilisée comme benchmark pour tester la puissance des super ordinateurs ou des grilles de calcul

Complexité et flops

- Temps d'exécution d'un algorithme en fonction de sa complexité, de la taille des données et de la puissance de la machine utilisée

flops	Taille des données : 1 million				Taille des données : 1 milliard			
	n	n log(n)	n^2	2^n	n	n log(n)	n^2	2^n
10^6	secondes	secondes	semaines	10000 ans	heures	heures	10000 ans	une éternité
10^9	millisecondes	millisecondes	heures	10 ans	secondes	secondes	décennies	10 billions d'années
10^{12}	microsecondes	microsecondes	secondes	semaines	millisecondes	millisecondes	semaines	10 milliards d'années

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

- Conclusion :
  - il vaut mieux optimiser ses algorithmes qu'attendre des années qu'un processeur surpuissant soit inventé
  - c'est d'autant plus indispensable que le volume des données stockées dans les systèmes d'informations augmente de façon exponentielle

Complexité et récursivité (1/3)

- *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
```

- *Si n ≠ 1, le calcul de la factorielle de n demande :*
  - *une comparaison d'entiers*
  - *le calcul de la factorielle de n-1*
  - *une multiplication d'entiers*

- *Si n = 1, le calcul de la factorielle demande :*
  - *une comparaison d'entiers*

Complexité et récursivité (2/3)

- *Si c(n) est la complexité recherchée, on a :*
  - *c(n) = ce + c(n-1) + me si n ≠ 1*
  - *c(1) = ce*

- *On résoud cette équation récursive : c(n) = n*ce + (n-1)*me*

- *La complexité théorique de la factorielle récursive est donc linéaire, comme celle de la factorielle itérative*

- *Par contre, à l'exécution, la fonction récursive est un peu moins rapide (pente de la droite plus forte) du fait des appels récursifs*

Complexité et récursivité (3/3)

- 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 (cf. UE Outils pour l'informatique 1 et 2)
  - méthode des fonctions génératrices et décomposition des fractions rationnelles
  - transformée en Z
  - ...

Résolution des récurrences (1/3)

- c(n) = c(n-1) + b
  - solution : c(n) = c(0) + b*n = O(n)
  - *exemples : factorielle, recherche séquentielle récursive dans un tableau*
- c(n) = a*c(n-1) + b, a ≠ 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*
- 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*
- 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 (2/3)

- c(n) = a*c(n/2) + b, a ≠ 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*
- c(n) = c(n/2) + n
  - solution : c(n) = O(n)
  - *exemples : traitement linéaire avant l'appel récursif dichotomique*
- 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*

Résolution des récurrences (3/3)

- 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 :

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

La relation entre a, b et k détermine la fonction T(n) comme suit :

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 $T(n) \Theta (n^k)$