UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Programmation avancée et Complexité

Chapitre 3 - Complexité et Ordre de grandeur

Frédéric Flouvat

Université de la Nouvelle-Calédonie

Plan

Un problème, plusieurs solutions

Pour un problème, plusieurs méthodes de résolutions possibles

Ex. problème du calcule de x^n

- *Méthode 1* : $x^n = x^{n-1}.x$
- *Méthode 2* (méthode des facteurs) :

$$x^n = \begin{cases} x & \text{si } n = 1\,; \\ x^{n-1}.x & \text{si } n \text{ premier}\,; \\ (x^p)^{n'} & \text{si } n = p.n' \text{ avec } p \text{ plus petit diviseur premier de } n. \end{cases}$$

- ...

  (dans certains livres d'algorithmique, 26 pages sont consacrés à ce problème)

Un problème, plusieurs solutions

Pour une méthode de résolution, plusieurs algorithmes possibles

- p.ex. pour la méthode 1

Fonction Puissance(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: res ← 1
2: Pour i de 0 à n − 1 faire
3:     res ← res . x
4: Fin Pour
5: Retourner res
```

Fonction Puissance(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: Si n = 1 Alors
2:     Retourner x
3: Fin Si
4: res ← Puissance(x, n-1) . x
5: Retourner res
```

⇨ Quelle méthode choisir ? Quel algorithme ? Quel(s) critére(s) ?

⇨ analyser la complexité des algorithmes et choisir la solution la plus intéressante

- p.ex. simplicité, efficacité, quantité de données stockées sur le disque, quantité de trafic généré sur le réseau
- simplicité vs efficacité

Complexité d'un algorithme

Comment comparer des algorithmes ? Si deux algorithmes résolvent le même problème, quel est le meilleur ?

➪ *Intuition* : préférer celui qui utilise le moins de ressources machine

- ressources de calculs
- ressources d'espace de stockage

➪ Méthode 1 : Etudes expérimentales

- implémenter l'algorithme dans un langage de programmation (p.ex. C)
- faire fonctionner le programme avec des entrées de taille et de composition différentes
- mesurer le temps d'exécution et l'espace mémoire occupé

Complexité d'un algorithme

problème de la méthode expérimentale :

- nécessiter d'implémenter
- des résultats ne représentant pas toutes les données en entrée
- nécessités d'utiliser le même environnement (matériels et systéme d'exploitation) pour pouvoir comparer

⇨ Méthode 2 : Analyse théorique

- se fait à partir de l'algorithme, non de l'implémentation
- caractérise les performances comme une fonction de n, la taille de l'entrée
- prend en considération toutes les entrées
- indépendant de l'environnement utilisé

Complexité en temps et en espace

Complexité en temps

- l'algorithme s'exécute le plus rapidement possible

⇨ compter le nombre d'opérations élémentaires (car supposé en temps constant)

- opérations arithmétiques, affectation, instruction de contrôle, entrée-sortie ...
- p.ex. une boucle "Pour i de 1 à n faire"
  ⇨ n itérations $\times$ (1 affectation de i + 1 incrementation de i + 1 test de i) , i.e. $3n$ operations (ou $\sum_{i=1}^{n} 3$)

Complexité en espace

- l'algorithme occupe le moins d'espace mémoire possible

⇨ nombre de cellules mémoire utilisées (sauf les données en entrée)

- taille des entrées-sorties, taille des variables temporaires, ...
- p.ex. une variable entiére = 1 cellule

Complexité d'un algorithme dépend de la taille de son entrée

Exemple de complexité d'un algorithme

Procedure MinListe(L)

Entrée: L une liste d'entier
Sortie: affiche le nombre minimum contenu dans L

```
 1: min ← premier ( L )
 2: i ← 1
 3: Tant que  i < longueur( L ) faire
 4:     Si  iéme( L, i ) < min  Alors
 5:         min ← iéme( L, i )
 6:     Fin Si
 7:     i ← i + 1
 8: Fin Tant que
 9: écrire("le minimum est ")
10: écrire( min )
```

Nombre d'opérations élémentaires

Soit n la longueur de la liste L

1 affectation + 1 appel premier(L)
1 affectation
n comparaisons + n appels à longeur()
(n-1) comparaisons + (n-1) appels à iéme()

m affectations + m appels à iéme()

(n-1) affectations + (n-1) additions

1 écriture

1 écriture

Exemple de complexité d'un algorithme

Complexité en temps de l'algorithme

- en supposant que l'appel à chacune des fonctions *premier()*, *longueur()* et *iéme()* est équivalent à une opération élémentaire
- nombre total d'opérations élémentaires :
  $3 + 2n + 2(n-1) + 2m + 2(n-1) + 2 = 6n + 2m + 1$
- valeur de m ? ⇨ dépend de la liste et de son contenu
  - Meilleur cas : $m = 0$ si le premier de la liste est le minimum
  - Pire cas : $m = n - 1$ si les nombres de la liste sont rangé par ordre décroissant
  - Cas moyen : $m = \frac{n-1}{2}$ si les nombres respectent une distribution parfaitement aléatoire

⇨ Complexité des algorithmes peut aussi dépendre de la structure de données utilisée

Complexité en espace de l'algorithme

- un entier pour stocker le minimum min
- un entier i pour savoir où on est dans la liste

Complexité au pire, au meilleur et en moyenne

Dans certains cas, la complexité (en temps et/ou en espace) dépend d'autres paramètres que la taille des données en entrée de l'algorithme

- p.ex. afficher le minimum d'une liste de nombres

⇨ étude de la complexité au pire cas, au cas moyen et au meilleur cas

Complexité au pire : liée au plus grand nombre d'opérations qu'aura à exécuter l'algorithme pour une entrée de taille n

- avantage : donne une borne maximum, pas de mauvais surprise
- inconvénient : peut ne pas refléter le comportement de l'algorithme en général, peut être très rare

- celle généralement utilisée

Complexité au pire, au meilleur et en moyenne

Complexité en moyenne : moyenne des complexités pour un ensemble de jeux de données de taille n représentatifs (en tenant compte de la probabilité d'apparition de chacun)

- avantage : représente le comportement de l'algorithme en général
- inconvénient : pas une bonne indication pour certains jeux de données
- intéressante pour compléter l'information apportée par la complexité au pire cas

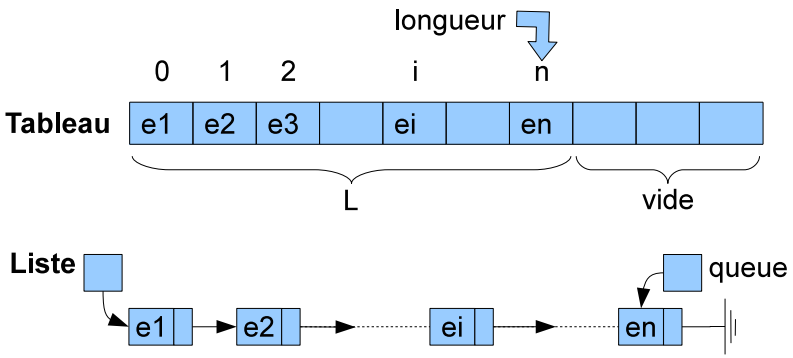
Complexité au meilleur : plus petit nombre d'opérations qu'aura à effectuer l'algorithme pour une entrée de taille n

- peut être un très mauvais indicateur si le meilleur cas est très rare
- peu utilisée pour choisir un algorithme

Attention à l'impact de la structure de données

Un même type abstrait de données peut être implémenté de plusieurs façons

- exemples d'implémentation des listes :
  - par un tableau
    - avantages : simple, économique en espace
    - inconvénients : modifications lentes
  - par une liste chaînée
    - avantages : opérations rapides, gestion souple
    - inconvénients : pas d'accès direct, consomme de l'espace mémoire



⇨ Souvent un fort impact sur l'efficacité des algorithmes

Exercice

Ecrire deux algorithmes permettant de calculer la somme des n premiers entiers et donner leur complexité (temps et espace)

Indication : utiliser pour l'un des algorithmes la propriété mathématique $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

En conclure que suivant l'algorithme, on arrive au même résultat mais pas nécessairement dans le même temps ni en utilisant la même quantité de mémoire.

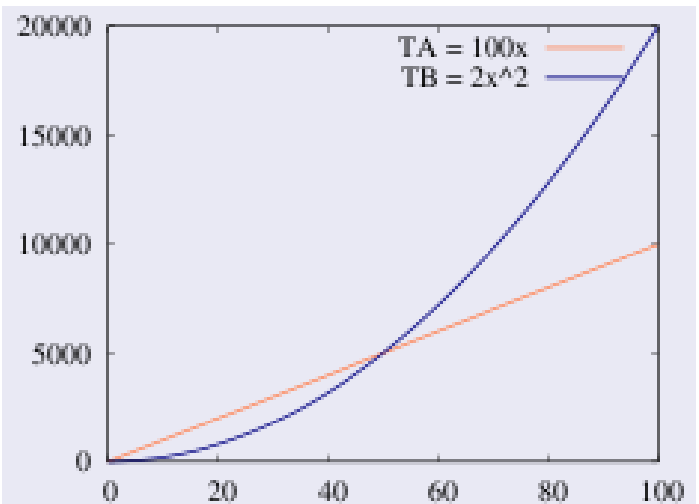
Ordre de grandeur

Jusqu'à présent, étude de la complexité ⇨ compter le nombre d'opérations élémentaires et l'espace occupé

Dans la majorité des cas, connaître le nombre exact d'opérations n'est pas nécessaire, un ordre de grandeur suffit

Illustration de l'inutilité des constantes

- comparaison du temps d'exécution de deux algorithmes A et B
- "temps d'exécution" de A (noté TA) : $T_A(n) = 100n$
- "temps d'exécution" de B (noté TB) : $T_B(n) = 2n^2$



Exemple de la recherche dans un tableau

Recherche dichotomique

- Hypothése : tableau trié
- Principe :
  - comparer la valeur recherchée avec l'élément au milieu du tableau
  - si c'est le même, retourner l'élément du milieu
  - sinon recommencer sur la premiére moitié (ou la deuxiéme) si la valeur recherchée est plus petite (ou plus grande) que l'élément rangé au milieu de la table

Fonction Dicho(chaine[] tab, chaine x)
Entrée: un tableau de chaînes de caractéres
Sortie: une chaîne de caractéres à trouver

```
 1: i ← 0 ; j ← tab.longueur - 1 ;
 2: Tant que i < j faire
 3:     Si tab[(j+i)/2] = x Alors
 4:         Retourner vrai
 5:     Sinon Si tab[(j+i)/2] > x Alors
 6:         j ← (j+i)/2 - 1
 7:     Sinon
 8:         i ← (j+i)/2 +1
 9:     Fin Si
10: Fin Tant que
11: Retourner faux
```

Exemple de la recherche dans un tableau trié

Recherche dichotomique

- Complexité en espace : la place de deux entiers
- Complexité en temps :
  - 2 affectations et 1 retourner + à chaque itération,
    - 1 comparaison d'entiers pour le test de boucle
    - 2 comparaisons de chaînes (1 si l'élément est trouvé)
    - 2 accès à une case d'un tableau
    - 7 opérations arithmétiques + / - (sauf si l'élément est trouvé)
    - 1 affectations $\leftarrow$
  - 1 test de condition d'arrêt du while
  - ⇨ au pire, la longueur du tableau entre i et j est n, puis $n/2$, puis $n/4$, ... , jusqu'à $n/2^t = 1$
  - ⇨ le nombre d'itérations est donc t tel que $n/2^t = 1$, i.e. $2^t = n$ soit $t * log(2) = log(n)$, $t = log_2(n)$
  - ⇨ complexité au pire cas : $13 \times log_2(n) + 4$

Recherche séquentielle

- Principe : examiner successivement tous les éléments de la table jusqu'à ce que l'élément voulu soit trouvé.
- Complexité au pire : $n \times c_s$ avec c_s une constante représentant le coût des opérations à chaque itération

Exemple de la recherche dans un tableau trié

Différence entre recherche dichotomique et séquentielle peut être simplifiée

- dans le pire cas, nombre d'opérations de la recherche dichotomique est proche de $log_2(n)$
- dans le pire cas, nombre d'opérations de la recherche séquentielle est proche de n

⇨ Approximation de la complexité suffisante pour comparer des algorithmes

- on sait qu'il existe une valeur v telle que pour tout $n > v$, $c_d \times log_2(n) < n \times c_s$

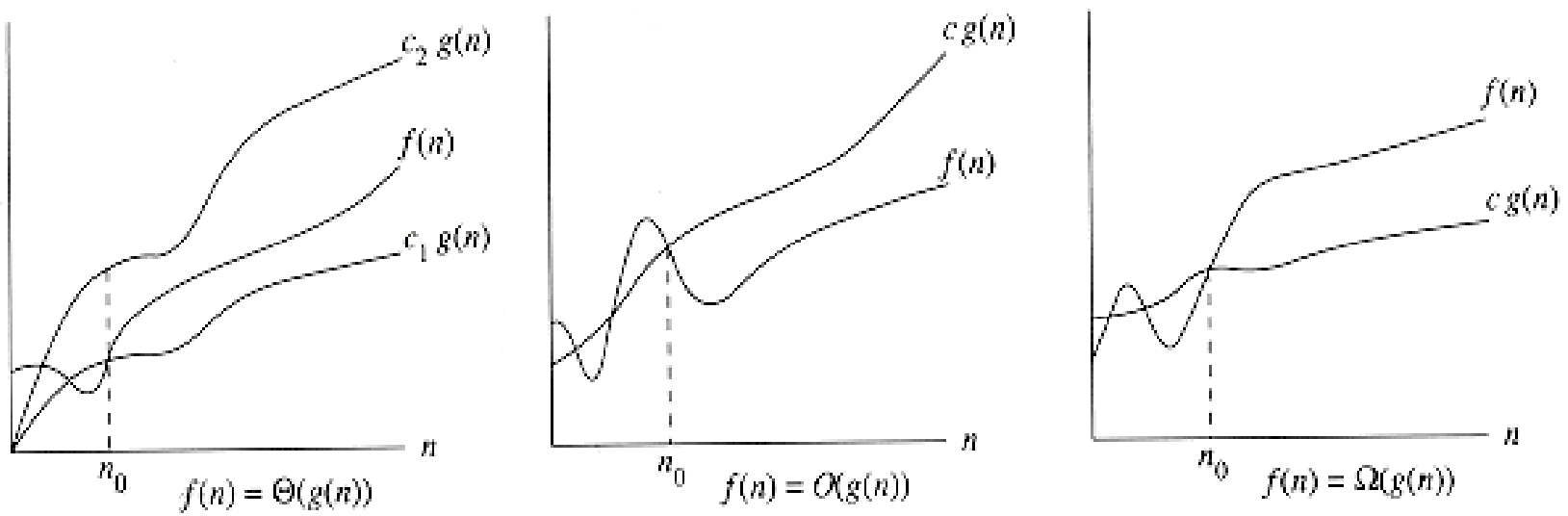
⇨ facilite la comparaison des algorithmes

Notation de Landau

Etude de l'ordre de grandeur

➪ besoin de notations asymptotiques, i.e. de bornes

- O ("grand O") : f en $O(g) \Leftrightarrow \exists n_0, \exists c > 0, \forall n \geq n_0, f(n) \leq c \times g(n)$
  - i.e. majoration
- Ω : f en $\Omega(g) \Leftrightarrow \exists n_0, \exists c > 0, \forall n \geq n_0, c \times g(n) \leq f(n)$
  - i.e. minoration
- Θ : f en $\Theta(g)$
  $\Leftrightarrow \exists n_0, \exists c_1, c_2 > 0, \forall n \geq n_0, c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$
  - i.e. équivalents à une constante multiplicative près



Notation de Landau : exemples

$2n$ en $O(?)$

- f(n) = 2n et g(n) = ?
- $\exists n_0, \exists c > 0, \forall n \geq n_0, 2n \leq c \times g(n)$
- prenons $g(n) = 3n$
- existe-t- il des valeurs pour n_0 et c telles que $2n \leq c \times 3n$?
- oui, si $n_0 = 1$ et $c = 2/3$

⇨ $2n$ en $O(3n)$

$\frac{1}{2}n^2 - 3n$ en $\Theta(?)$

- f(n) = $\frac{1}{2}n^2 - 3n$ et g(n) = ?
- $\exists n_{0_2}, \exists c_2 > 0, \forall n \geq n_{0_2}, f(n) \leq c_2 \times g(n)$ et $\exists n_{0_1}, \exists c_1 > 0, \forall n \geq n_{0_1}, c_1 \times g(n) \leq f(n)$
- prenons g(n) = n^2
- existe-t- il des valeurs pour n_{0_1}, n_{0_2}, c_1 et c_2 telles que $\frac{1}{2}n^2 - 3n \leq c_2 \times n^2$ et $c_1 \times n^2 \leq \frac{1}{2}n^2 - 3n$?
- oui si $n_{0_2} = 1$, $c_2 = 1/2$, $n_{0_1} = 7$, $c_1 = 1/14$

⇨ $\frac{1}{2}n^2 - 3n$ en $\Theta(n^2)$

Notation de Landau : exercices

n en $O(?)$

- $n_0 = ?$
- $c = ?$

$n^2 - n + 1$ en $\Theta(?)$

- $n_0 = ?$
- $c = ?$

$(n+1)^2$ en $O(?)$

- $n_0 = ?$
- $c = ?$

Comment "deviner" $g(n)$?

Quelques principes généraux en pratique :

- les facteurs constants ne sont pas importants
- les termes d'ordre inférieur sont négligeables
  - $a_k n^k + a_{k-1} n^{k-1} + ... + a_2 n^2 + a_1 n + a_0$ est en $O(n^k)$

Exemples :

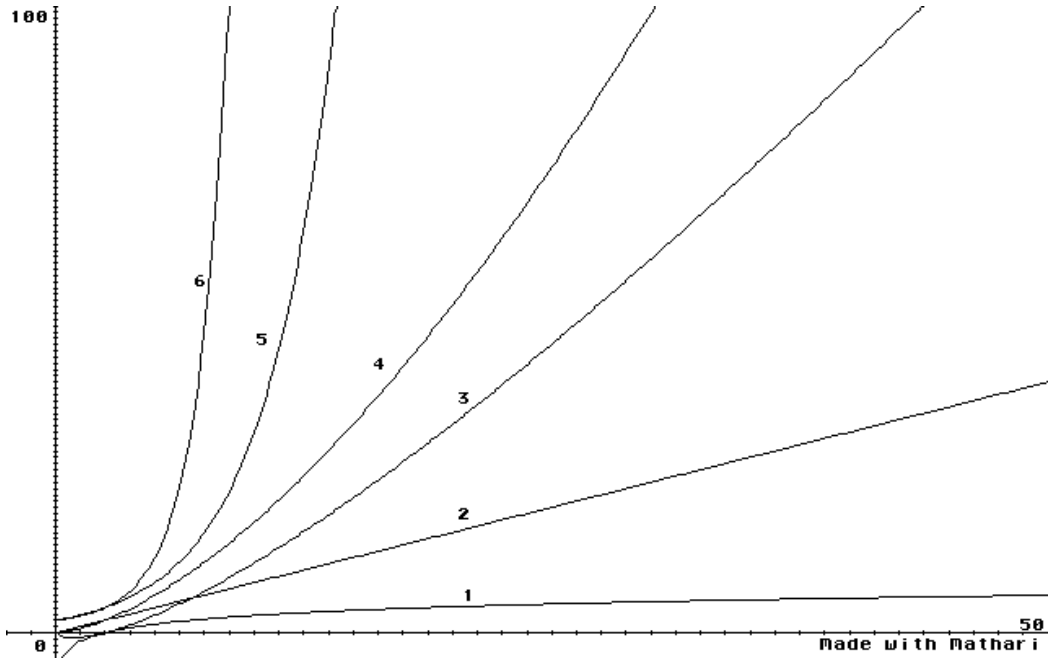
- $T(n) = (n+1)^2$ est en $O(n^2)$
- $Q(n) = 3n + 15$ est en ?
- $R(n) = n^2 + n + 250$ est en ?
- $S(n) = log_2(n) + 25$ est en ?
- $U(n) = 2^n + n^3$ est en ?

Classes de complexité

Plusieurs grandes classes de complexité

- les algorithmes sub-linéaires (temps logarithmique)
  - complexité en $O(log(n))$
  - les plus rapides
- les algorithmes linéaires
  - complexité en $O(n)$ et en $O(n\ log(n))$
  - des algorithmes rapides
- les algorithmes quadratiques
  - complexité en $O(n^2)$
  - des algorithmes avec des performances moyennes
- les algorithmes polynomiaux :
  - complexité en $O(n^k)$ (avec $k > 2$)
  - des algorithmes lents
- les algorithmes exponentiels :
  - complexité supérieur à tout polynôme en n
  - des algorithmes intraitables lorsque la taille des données est supérieur à quelques dizaines d'unités

Classes de complexité



1. complexité logarithmique en $O(ln(n))$ (ou $ln^k(n)$, $k > 1$)
2. complexité linéaire en $O(n)$
3. complexité quasi-linéaire en $O(n\ ln(n))$
4. complexité polynomiale en $O(n^k)$ $(k > 1)$
5. complexité exponentielle en $O(a^n)$ $(a > 1)$
6. complexité hyperexponentielle comme $O(n!)$ ou pire, $O(n^n)$

Comparaison des ordres de grandeur

$O(1) \subset O(log(n)) \subset O(ln(n)) \subset O(n) \subset O(n\ log(n)) \subset O(nln(n)) \subset O(n^k) \subset O(2^n) \subset O(n!) \subset O(n^n)$, avec $k \geq 2$

	2	2^4	2^6	2^8
log(log(n))	0	2	2.58	3
log(n)	1	4	6	8
n	2	16	64	256
n log(n)	2	64	384	2 048
n^2	4	256	4 096	65 536
2^n	4	65 536	1.84e+19	1.15e+77
n !	2	2.09e+13	1.26e+89	8.57e+506

Exemples de complexités et de tâches associées

Complexité	*Tâche*
O(1)	accès direct à un élément
O(log(n))	divisions successives par deux d'un ensemble
O(n)	parcours d'un ensemble
O(n log(n))	divisions successives par deux d'un ensemble et parcours de toutes les parties
O(n^2)	parcours d'une matrice carrée de taille n
O(2^n)	génération des parties d'un ensembles
O(n !)	génération des permutations d'un ensemble

Exercice complexité : égalité de deux matrices

Donner la complexité en temps et en espace de l'algorithme suivant vérifiant l'égalité entre deux matrices carrées de même taille

Fonction EGAL(A,B,l)

Entrée: A et B deux matrices carrées de même taille l
Sortie: vrai si A et B sont égaux, faux sinon

```
1: Pour i de 0 à l − 1 faire
2:     Pour j de 0 à l − 1 faire
3:         Si A[i][j] ≠ B[i][j] Alors
4:             Retourner Faux
5:         Fin Si
6:     Fin Pour
7: Fin Pour
8: Retourner Vrai
```

Calculer la complexité en temps d'un algorithme itératif

1. Compter le nombre d'opérations élémentaires en fonction de la taille de l'entrée, noté $f(n)$

une affectation, une opération arithmétique, un accès à une case d'un tableau, etc	1 opération élémentaire
un test d'égalité, une comparaison (pour les types primitifs)	1 opération élémentaire
une boucle `Pour X de i à j` avec $X \leq j$	3 opérations élémentaires $\times\ (j - i + 1)$ itérations
une boucle `Tant que ... faire`	nb d'op. élém. du test $\times$ nb itérations + nb d'op. élém. du test d'arrêt
à l'intérieure d'une boucle	multiplier chaque opérations par le nombre d'itérations de la boucle

- Si le nombre d'itérations d'une boucle dépend des données, introduire une nouvelle variable et étudier la complexité au pire et/ou au meilleur et/ou en moyenne

Calculer la complexité en temps d'un algorithme itératif

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau)

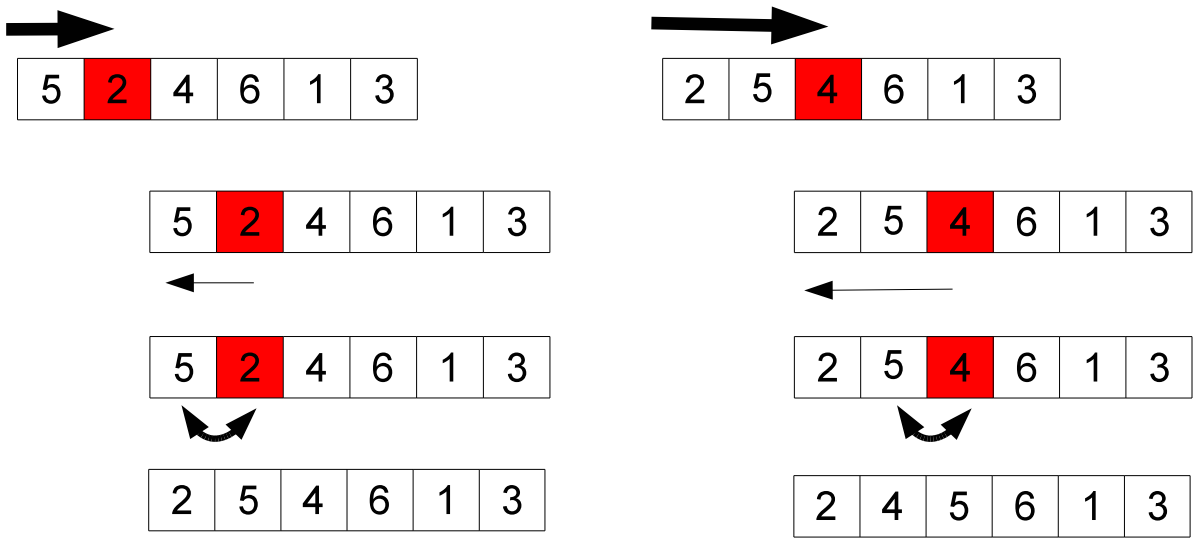
1. Choisir la notation de Landau en fonction de l'approximation souhaitée : O (borne sup.), Ω (borne inf.) et Θ (équivalents)
2. Deviner $g(n)$ à partir des termes d'ordre supérieur
3. Démontrer que $f(n)$ vérifie la notation de Landau en trouvant un n_o et un c

Exemple du tri par insertion

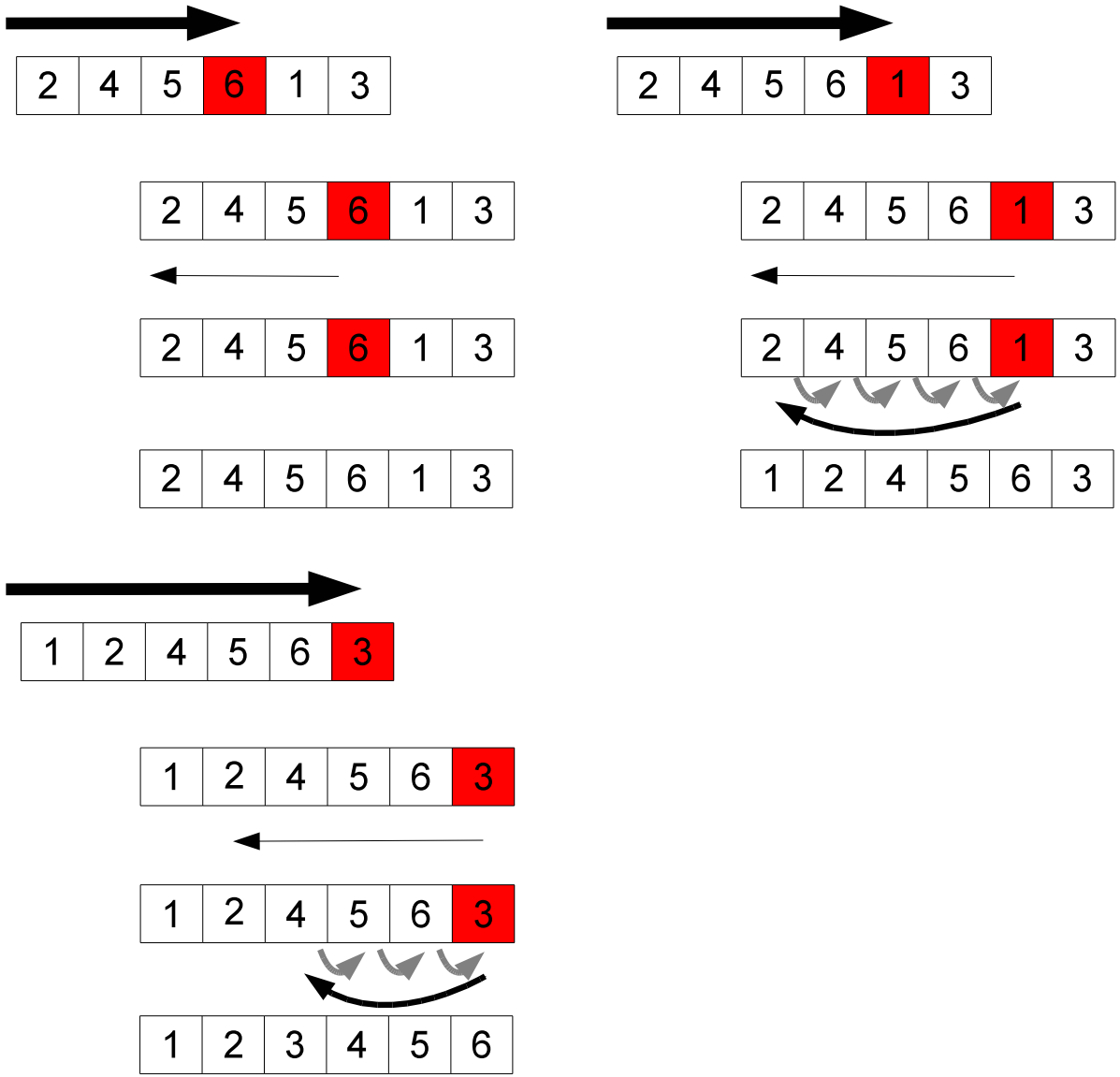
Principe :

- de manière répétée, on choisit un nombre de la séquence d'entrée et on le déplace à la bonne position dans la séquence des nombres déjà triés
- même principe que celui utilisé pour trier une poignée de cartes

Exemple :



Exemple du tri par insertion



Exemple du tri par insertion

Procedure TriInsertion(element[] A)

Entrée: A un tableau d'entier
Sortie: le tableau A trié dans l'ordre croissant

		Nombre d'opérations élémentaires
		Soit n la longueur du tableau A
1:	**Pour** j de 1 à $longueur(A) - 1$ **faire**	4 (n-1) (si longueur(A) $\rightarrow$ 0 op.)
2:	clé$\leftarrow A[j]$	2 (n-1)
3:	$i \leftarrow j - 1$	2 (n-1)
4:	**Tant que** $i >= 0$ et $A[i] >$clé **faire**	$\sum_{j=1}^{n-1}(3t_j + 3) = 3(n-1) + \sum_{j=1}^{n-1} 3t_j$
5:	$A[i+1] \leftarrow A[i]$	$\sum_{j=1}^{n-1} 4t_j$
6:	$i \leftarrow i - 1$	$\sum_{j=1}^{n-1} 2t_j$
7:	**Fin Tant que**	
8:	$A[i+1] \leftarrow$ clé	3(n-1)
9:	**Fin Pour**	

t_j : nombre d'itérations de la boucle **Tant que** pour une valeur de j donnée $(j \in [1..n-1])$

Exemple du tri par insertion

1. Compter le nombre d'opérations élémentaires en fonction de la taille de l'entrée, noté $f(n)$

- Nombre d'opérations élémentaires du tri pas insertion : $f(n) = 14n - 14 + \sum_{j=1}^{n-1} 9t_j$, avec valeur de t_j dépendant du contenu du tableau

- Complexité (en temps) au pire :
  - tableau trié dans l'ordre inverse $\rightarrow t_j = j$
  - ⇨ $f(n) = \frac{9}{2}n^2 + \frac{19}{2}n - 14$

$$f(n) = 14n - 14 + 9\sum_{j=1}^{n-1} j$$

rappel : $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$

or $\sum_{j=1}^{n} j = n + \sum_{j=1}^{n-1} j$

donc $\sum_{j=1}^{n-1} j = \frac{n(n+1)}{2} - n = \frac{n^2-n}{2}$

$$f(n) = 14n - 14 + \frac{9}{2}n^2 - \frac{9}{2}n$$

$$f(n) = \frac{9}{2}n^2 - \frac{19}{2}n - 14$$

Exemple du tri par insertion

1. Compter le nombre d'opérations élémentaires en fonction de la taille de l'entrée, noté $f(n)$ (Suite)

- Complexité (en temps) en moyenne :
  - si les nombres sont choisis au hasard, quelle sera la valeur de t_j ? , i.e. où devra-t-on insérer $A[j]$ dans le sous-tableau $A[1..j-1]$?

  ⇨ $t_j = j/2$ en moyenne

  ⇨ $f(n) = \frac{9}{4}n^2 + \frac{47}{4}n - 14$

$$f(n) = 14n - 14 + 9(\frac{n(n-1)}{4})$$

- Complexité (en temps) au meilleur :
  - tableau déjà trié $\rightarrow t_j = 0$

  ⇨ $f(n) = 14n - 14$

Exemple du tri par insertion

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau)

- Complexité (en temps) au pire :
  1. approximation souhaitée : O
     - rappel : $f = O(g) \Leftrightarrow \exists n_0, \exists c > 0, \forall n \geq n_0, f(n) \leq c \times g(n)$
  2. supposons que $g(n) = n^2$
  3. existe-t-il n_o et c vérifiant la définition ?
     oui, si $n_o = 1$ et $c = 14$, alors $\forall n \geq n_0, \frac{9}{2}n^2 + \frac{19}{2}n - 14 \leq 14 \times n^2$

  ⇨ l'algorithme est en $O(n^2)$ (i.e. quadratique) dans le pire des cas

- Complexité (en temps) en moyenne :
  1. approximation souhaitée : O
  2. supposons que $g(n) = n^2$
  3. existe-t-il n_o et c vérifiant la définition ?
     oui, si $n_o = 1$ et $c = 14$, alors $\forall n \geq n_0, \frac{9}{4}n^2 + \frac{47}{4}n - 14 \leq 14 \times n^2$

  ⇨ l'algorithme est en $O(n^2)$ (i.e. quadratique) en moyenne

Exemple du tri par insertion

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau) (Suite)

- Complexité (en temps) au meilleur :
  1. approximation souhaitée : O
  2. supposons que $g(n) = n$
  3. existe-t-il n_o et c vérifiant la définition ?
     oui, si $n_o = 1$ et $c = 14$, alors $\forall n \geq n_0, 14n - 14 \leq 14 \times n$
  - ⇨ l'algorithme est en $O(n)$ (i.e. linéaire) dans le meilleur cas

Exercice : donner une approximation en Θ de la complexité (en temps) de l'algorithme de tri par insertion.

Calculer la complexité en temps des algorithmes récursifs

1. Exprimer la complexité de l'algorithme sous la forme d'une équation de récurrence

- Pourquoi ?
  - impossible de calculer directement le nombre d'opérations à cause des appels récursifs
- Comment ?
  - principe de la récursivité : problème initial divisé en a sous-problèmes chacun de taille $1/b$ de la taille du problème initial

⇨

$$f(n) = \begin{cases} \Theta(1), & \text{si } n \leq c \\ af(n/b) + D(n) + C(n), & \text{sinon} \end{cases}$$

- $af(n/b)$: temps de résolution des a sous-problèmes (avec $n/b = \lfloor n/b \rfloor$ ou $\lceil n/b \rceil$)
- $D(n)$: temps nécessaire à la division du problème en sous-problèmes
- $C(n)$: temps pour construire la solution finale à partir des solutions des sous-problèmes

Calculer la complexité en temps des algorithmes récursifs

1. Exprimer la complexité de l'algorithme sous la forme d'une équation de récurrence (Suite)

- Exemple :

Fonction Fact(n)	**Nombre d'opérations élémentaires**
Entrée: un entier $n \geq 0$	*Soit f(n) le nombre d'opérations pour calculer Fact(n)*
Sortie: l'entier $n!$	
1: **Si** n = 0 **Alors**	1
2: **Retourner** 1	1
3: **Fin Si**	
4: res ← Fact(n-1) . n	1 + f(n-1) + 2
5: **Retourner** res	1

- $f(n) = \begin{cases} 2, & \text{si } n \leq 0 \\ f(n-1) + 5, & \text{sinon} \end{cases}$
  - décomposition en $a = 1$ sous-problème de taille $n - 1$
  - temps nécessaire à la division du problème en sous-problèmes : $D(n) = 1$
  - temps pour construire la solution finale à partir des solutions des sous-problèmes : $C(n) = 3$

Calculer la complexité en temps des algorithmes récursifs

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau)

1. Choisir la notation de Landau en fonction de l'approximation souhaitée : O (borne sup.), Ω (borne inf.) et Θ (équivalents)
2. Deviner $g(n)$
   - attention : impossible d'utiliser les termes d'ordre supérieur
   - ⇨ s'inspirer des récurrences similaires, et dont la solution est connue
   - ⇨ trouver une borne supérieure (et/ou inférieure) très large, puis réduire
     - p.ex. étudier $O(n^2)$, puis $O(n\ log_2(n))$, ...
3. Démontrer que $f(n)$ vérifie la notation de Landau en faisant une démonstration par récurrence
   - 3.1 vérifier que f satisfait la notation de Landau choisie pour un n_0
   - 3.2 faire l'hypothèse que $f(n/b)$ vérifie la notation de Landau
   - 3.3 démontrer que $f(n)$ vérifie la notation de Landau en substituant $f(n/b)$ (cf hypothèse)

Calculer la complexité en temps des algorithmes récursifs

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau) (Suite)

Exemple avec l'algorithme récursif $Fact(n)$:

1. approximation souhaitée : O
2. supposons que $g(n) = n$
3. démontrer par récurrence que $f(n) = f(n-1) + 5$ est en $O(n)$, cad $\exists n_0, \exists c > 0, \forall n \geq n_0, f(n) \leq c \times n$
   - 3.1 vérifier que f satisfait la notation de Landau choisie pour un n_0, cad $\exists c > 0, f(n_0) \leq c \times n_0$
     - oui, si $n_0 = 1$ alors $f(1) = f(0) + 5 = 7$ et $f(1) \leq c \times 1, \forall c \geq 7$
   - 3.2 hypothèse de récurrence : $f(n-1)$ en $O(n-1)$, cad $\exists c > 0, f(n-1) \leq c \times (n-1)$
   - 3.3 démontrer que $f(n)$ en $O(n)$
     - $f(n) = f(n-1) + 5$
     - donc $f(n) \leq c \times (n-1) + 5$
     - $f(n) \leq c \times n - c + 5 \leq c \times n$, si $c \geq 5$
     - CQFD

Exemple du tri fusion

1. Exprimer la complexité de l'algorithme sous la forme d'une équation de récurrence

Procedure mergeSort(element[] tab, entier i, entier j)

Entrée: une partition du tableau tab située entre les indices i et j
Sortie: la partition triée

		Nombre d'opérations élémentaires
		Soit n la longueur du tableau T et f(n) la fonction représentant le nombre d'opérations
1:	**Si** $i < j$ **Alors**	1
2:	mergeSort(tab, i, (i+j)/2)	f(n/2) + 2
3:	mergeSort(tab, (i+j)/2+1, j)	f(n/2) + 3
4:	fusion(tab, i, (i+j)/2, j)	2 + fusion(n) ?
5:	**Fin Si**	

➪ $f(n) = 2f(n/2) + 8 + fusion(n)$

- mais que vaut $fusion(n)$?

Exemple du tri fusion

```
Procedure fusion(element[] T, entier deb, entier mid, entier fin)
  Entrée: T le tableau initial triés entre deb et mid, et entre
          mid+1 et fin
  Sortie: le tableau T trié entre deb et fin
 1: i ← 0 ; i_1 ← deb ; i_2 ← mid + 1                                  | 4
 2: Tant que i_1 ≤ mid et i_2 ≤ fin faire                              | 2 × (n-1) + 1 (condition d'arrêt sur le premier sous tableau)
 3:     Si T[i_1] < T[i_2] Alors                                       | 3 × (n-1)
 4:         temp[i] ← T[i_1]                                           | 3 × (n-1)
 5:         i_1 ← i_1 + 1                                              | 2 × (n-1)
 6:     Sinon
 7:         temp[i] ← T[i_2]
 8:         i_2 ← i_2 + 1
 9:     Fin Si
10:     i ← i + 1                                                     | 2 × (n-1)
11: Fin Tant que
12: Si i_1 < mid + 1 Alors                                            | 2
13:     Pour j de i_1 à mid faire
14:         temp[i] ← T[j]
15:         i ← i + 1
16:     Fin Pour
17: Sinon Si i_2 < fin + 1 Alors                                      | 2
18:     Pour j de i_2 à fin faire                                     | 3 × 1 (car 1 itération dans le pire des cas)
19:         temp[i] ← T[j]                                            | 3
20:         i ← i + 1                                                 | 2
21:     Fin Pour
22: Fin Si
23: k ← 0                                                             | 1
24: Pour i de deb à fin faire                                         | 3 × n
25:     T[i] ← temp[k] ; k ← k + 1                                    | 5 × n
26: Fin Pour
```

Nombre d'opérations élémentaires (dans le pire cas)
Soit n la longueur du tableau T entre les cases deb **et** fin

Exemple du tri fusion

1. Exprimer la complexité de l'algorithme sous la forme d'une équation de récurrence (Suite)

- Dans le pire cas, $fusion(n) = 20n + 6$
- Donc $f(n) = 2f(n/2) + 20n + 14$ avec $f(1) = 1$

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau)

1. approximation souhaitée : O
2. supposons que $g(n) = n\ log_2(n)$
3. démontrer par récurrence que $f(n) = 2f(n/2) + 20n + 14$ est en $O(n\ log_2(n))$, cad $\exists n_0, \exists c > 0, \forall n \geq n_0, f(n) \leq c \times n \times log_2(n)$

Exemple du tri fusion

2. Afin de faciliter les comparaisons avec d'autres algorithmes, approximer cette complexité en donnant un ordre de grandeur (utiliser la notation de Landau) (Suite)

3 démontrer par récurrence que $f(n) = 2f(n/2) + 20n + 14$ est en $O(n\ log_2(n))$, cad $\exists n_0, \exists c > 0, \forall n \geq n_0, f(n) \leq c \times n \times log_2(n)$

3.1 vérifier que f satisfait la notation de Landau choisie pour un n_0, cad $\exists c > 0, f(n_0) \leq c \times n_0 \times log_2(n_0)$

- oui, si $n_0 = 2$ alors $f(2) = 2f(1) + 20 \times 2 + 14 = 56$ et $f(2) \leq c \times 2\log_2(2)$ $(\log_2(2) = 1)$, $\forall c \geq 28$

3.2 hypothèse de récurrence : $f(n/2)$ en $O(\ n/2\ log_2(n/2)\ )$, cad $\exists c > 0, f(n/2) \leq c \times n/2 \times log_2(n/2)$

3.3 démontrer que $f(n)$ en $O(n\ log_2(n))$

- $f(n) = 2f(n/2) + 20n + 14$
- donc $f(n) \leq 2 \times (\ c \times n/2 \times log_2(n/2)\ ) + 20n + 14$
  $\leq c \times n \times (\ log_2(n) - log_2(2)\ ) + 20n + 14$
  $\leq c \times n \times log_2(n) - c \times n + 20n + 14$
- Or $c \times n \times log_2(n) - c \times n + 20n + 14 \leq c \times n \times log_2(n)$, si $c \times n \geq 20n + 14$, cad si $c \geq 34$ $(n \geq 1)$
- CQFD

Exercices : démonstration par récurrence

En supposant que $f(1) = 1$,

1. montrer que $f(n) = 2f(\lfloor n/2 \rfloor) + n$ est en $\Theta(n\ log_2(n))$

2. montrer que $f(n) = f(\lfloor n/2 \rfloor) + 1$ est en $O(log_2(n))$

3. montrer que $f(n) = 2f(\lfloor n/2 \rfloor + 2) + n$ est en $O(n\ log_2(n))$
   - avec $f(n) = 1$, tout $0 < n < 5$
   - indication : démontrer d'abord que $f(n) \leq c(n-4)log_2(n-4) - n$ (car $c(n-4)log_2(n-4) - n \leq c \times n \times log_2(n)$, $c > 0$ et $n > 0$)

Pour résumer

Analyse de la complexité des algorithmes

➪ Objectif : choisir le meilleur algorithme en fonction de ses besoins

- en général, un algorithme est plus efficace qu'un autre si sa complexité dans le pire cas a un ordre de grandeur inférieur
- compromis espace-temps

Techniques :

- complexité en temps :
  - complexité des algorithmes itératifs
    - compter les opérations élémentaires et exprimer la complexité sous la forme d'un fonction dépendant de la taille de l'entrée (n)
    - estimer l'ordre de grandeur (O, Θ) en fonction de la taille de l'entrée
    - trouver des valeurs de n_0 et de c vérifiant la définition
  - complexité des algorithmes récursifs
    - compter les opérations élémentaires et exprimer la complexité sous la forme d'une équation de récurrence
    - estimer l'ordre de grandeur (O, Θ) en fonction de la taille de l'entrée
    - démontrer par récurrence que l'ordre de grandeur est vérifié
- complexité en espace
  - étudier la taille (en mémoire généralement) des structures de données et variables utilisées

Rq : si nécessaire, étudier le pire cas et/ou le cas moyen

Exercice problème de puissance

Donner la complexité (dans le pire cas) en temps et en espace des trois algorithmes suivants permettant de calculer x^k

Fonction Puissance1(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: res ← 1
2: Pour i de 0 à n − 1 faire
3:     res ← res . x
4: Fin Pour
5: Retourner res
```

Fonction Puissance2(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: Si n = 0 Alors
2:     Retourner 1
3: Sinon
4:     Retourner x.Puissance2( x, n-1 )
5: Fin Si
```

Exercice problème de puissance

Fonction Puissance3(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

```
1: Si n = 0 Alors
2:     Retourner 1
3: Sinon
4:     Si estPair( n ) Alors
5:         Retourner Puissance3( x.x, n/2 )
6:     Sinon
7:         Retourner x.Puissance3( x.x, (n-1)/2 )
8:     Fin Si
9: Fin Si
```