ALGORITHMIQUE AVANCEE

Bouchentouf Toumi
ENSAO

Informatique : Origines

Exemple 1

- Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on fera apparaître un message : « Plus petit ! », et inversement, « Plus grand ! » si le nombre est inférieur à 10.

- Variable N en Entier
- Debut
- N←0
- Ecrire "Entrez un nombre entre 10 et 20 »
- TantQue N < 10 ou N > 20 faire
  - Lire N
  - Si N < 10 Alors Ecrire "Plus grand ! »
  - SinonSi N > 20 Alors
  - Ecrire "Plus petit ! »
  - FinSi
- FinTantQue
- Fin

Exemple 2 : Algorithme nombre base 2

- Nombres en base 2. Tout nombre entier s'écrit de façon unique sous la forme d'une somme de puissances de 2 toutes différentes.
- Par exemple 2007 s'écrit 2007 = 1024 + 512 + 256 + 128 + 64 + 16 + 4 + 2 + 1
- Soit 2007 = 210 + 29 + 28 + 27 + +26 + 24 + 22 + 21 + 20
- On dit alors que l'écriture de 2007 en base 2 est 11111010111 plus généralement l'écriture de n en base 2 est égale à $c_k c_{k-1}$ . . . c_2c1c0 si les nombres ci sont des 0 ou des 1

- Algorithme Base2
- Variable a entier;
- Debut
  - tantque(a != 0) faire
    - ecrire(a%2);
    - a = a/2;
  - Fintantque
- Fin

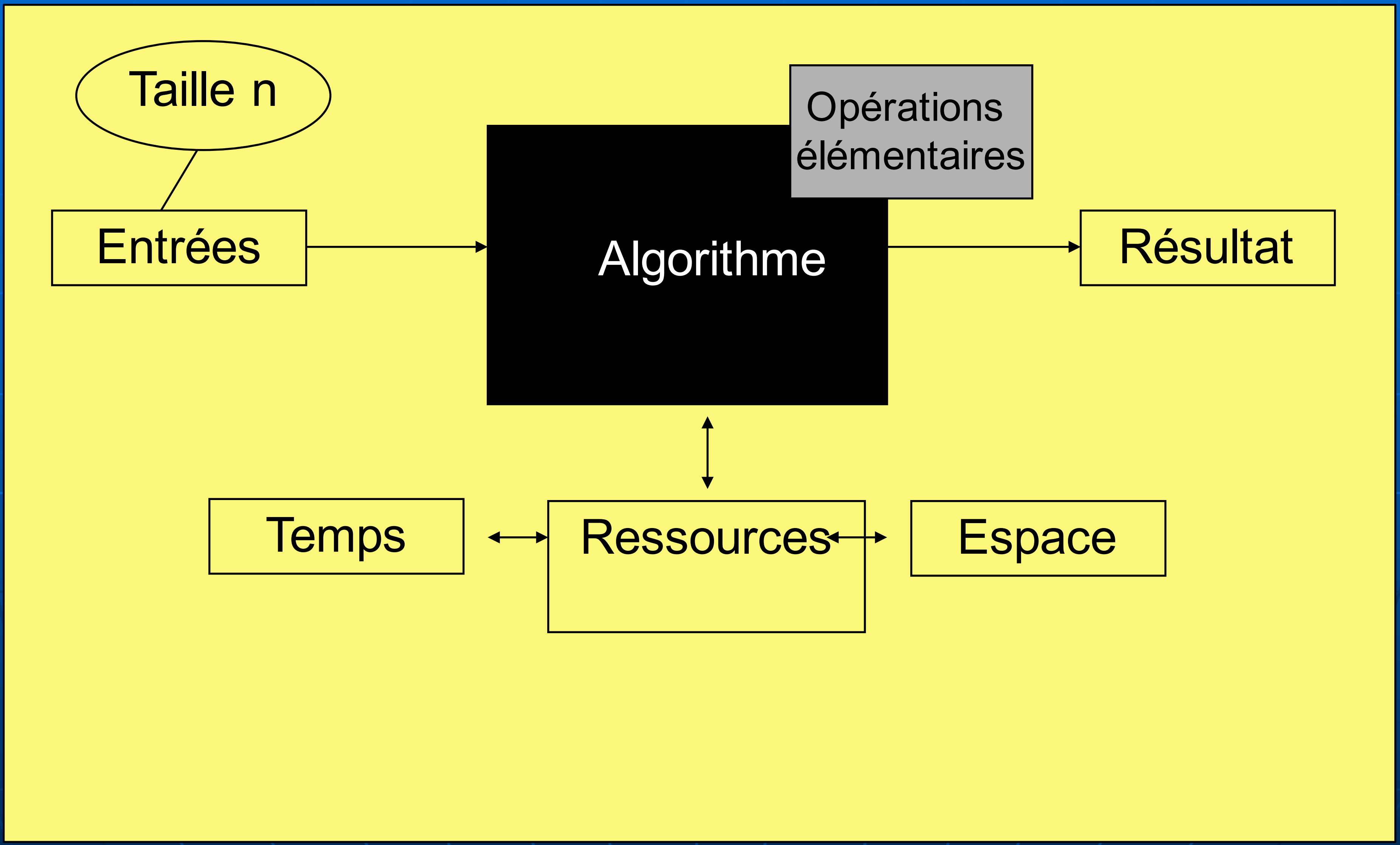
Histoire Bref de l'algorithme

- Existe-t-il un algorithme qui étant donné un énoncé logique permet de décider si cet énoncé est vrai ou pas.
- Gödel a démontré qu'il existe des vérités mathématiques qu'on ne peut pas démontrer: théorème d'incomplétude
- Alan Turing définit en 1934 un processus précis qui est assez général pour représenter toute méthode bien définie pour résoudre des problèmes de façon méthodique et automatisable. De là est né le concept de la machine de Turing.

- Machines de Turing ≈ formalisation minimaliste de la notion d'algorithme.

- Thèse de Turing ou Church-Turing : toute notion « raisonnable » d'algorithme est équivalente à la notion de machine de Turing.

Evaluation d'un algorithme



Notion de complexité d'un algorithme

- Pour évaluer l'**efficacité** d'un algorithme, on calcule sa **complexité**
- Mesurer la **complexité** revient à quantifier le **temps** d'exécution et l'espace **mémoire** nécessaire
- Le temps d'exécution est proportionnel au **nombre des opérations** effectuées. Pour mesurer la complexité en temps, on met en évidence certaines opérations fondamentales, puis on les compte
- Le nombre d'opérations dépend généralement du **nombre de données** à traiter. Ainsi, la complexité est une fonction de la taille des données. On s'intéresse souvent à son **ordre de grandeur** asymptotique
- En général, on s'intéresse à la **complexité** dans **le pire des cas** et à la **complexité moyenne**

Evaluation d'algorithmes

Question : étant donnés deux algorithmes qui calculent les solutions à un même problème. Comment comparer ces algorithmes? Autrement dit, quel est le meilleur?

- Intuition : On préférera celui qui nécessite le moins de ressources :
  - Ressource de calculs;
  - Ressource d'espace de stockage;

Evaluation des temps d'exécution

Problématique : Comment évaluer le temps d'exécution d'un algorithme donné?

- Idée : compter le nombre d'opérations élémentaires effectuées lors de l'exécution.
- Il varie avec les donnees d'entree de l'algorithme

Opération élémentaire

Définition : Une *opération élémentaire* est une opération qui s'effectue en temps constant sur tous les calculateurs usuels.

- On considérera les opérations suivantes comme élémentaires (sur des types simples):
  - Comparaisons;
  - Opérations arithmétiques et logiques;
  - Entrée-sortie;

Fonction de complexité

Définition : La *fonction de complexité temporelle* d'un algorithme exprime le temps requis, par l'algorithme pour calculer la solution correspondant à une instance en fonction de la taille de celle-ci.

Cette fonction de complexité dépend donc du codage retenu pour évaluer la taille de l'instance et du modèle de machine utilisé pour l'évaluation du temps d'exécution d'une opération élémentaire.

Hypothèses de simplification

Pour évaluer le nombres d’opérations élémentaires on s’appuie sur les paramètres de description de la donnée.

- nombre d’éléments d’un tableau;
- nombre de caractères d’une chaîne;
- nombre d’éléments d’un ensemble;
- profondeur et largeur d’un arbre;
- nombre de sommets et d’arêtes d’un graphe;
- dimension d’une relation d’ordre;
- taille ou valeur des nombres caractéristiques du problème;

Critères d'évaluation

- Pour une même taille de donnée, le nombre d'opérations élémentaires exécutées reste variable.
- On propose alors plusieurs critères d'évaluation :

Analyse dans le pire des cas : $t(n)$ = maximum des temps d'exécution de l'algorithme pour toutes les instances de taille n.

Analyse moyenne : $t_{moy}(n)$ = moyenne des temps d'exécution de l'algorithme pour toutes les instances de taille n.

Ordre de grandeur : Notation de Landau(grand O)

- Il reste fastidieux de compter toutes les opérations élémentaires d'une exécution.

Ordre de grandeur : On dit qu'une fonction $f(n)$ est en $O(g(n))$ s'il existe une constance c, positive et non nulle, telle que $|f(n)| \leq c\,|g(n)| \quad n \geq 0$

- $3\,n + 15$ est en $O(n)$;
- $n^2 + n + 250$ est en $O(n^2)$;
- $2\,n + n \log_2 n$ est en $O(n \log n)$;
- $\log_2 n + 25$ est en $O(\log_2 n)$;

Ordre de grandeur

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 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 ensemble
$O(n!)$	Génération des permutations d'un ensemble

Recherche séquentielle : complexité

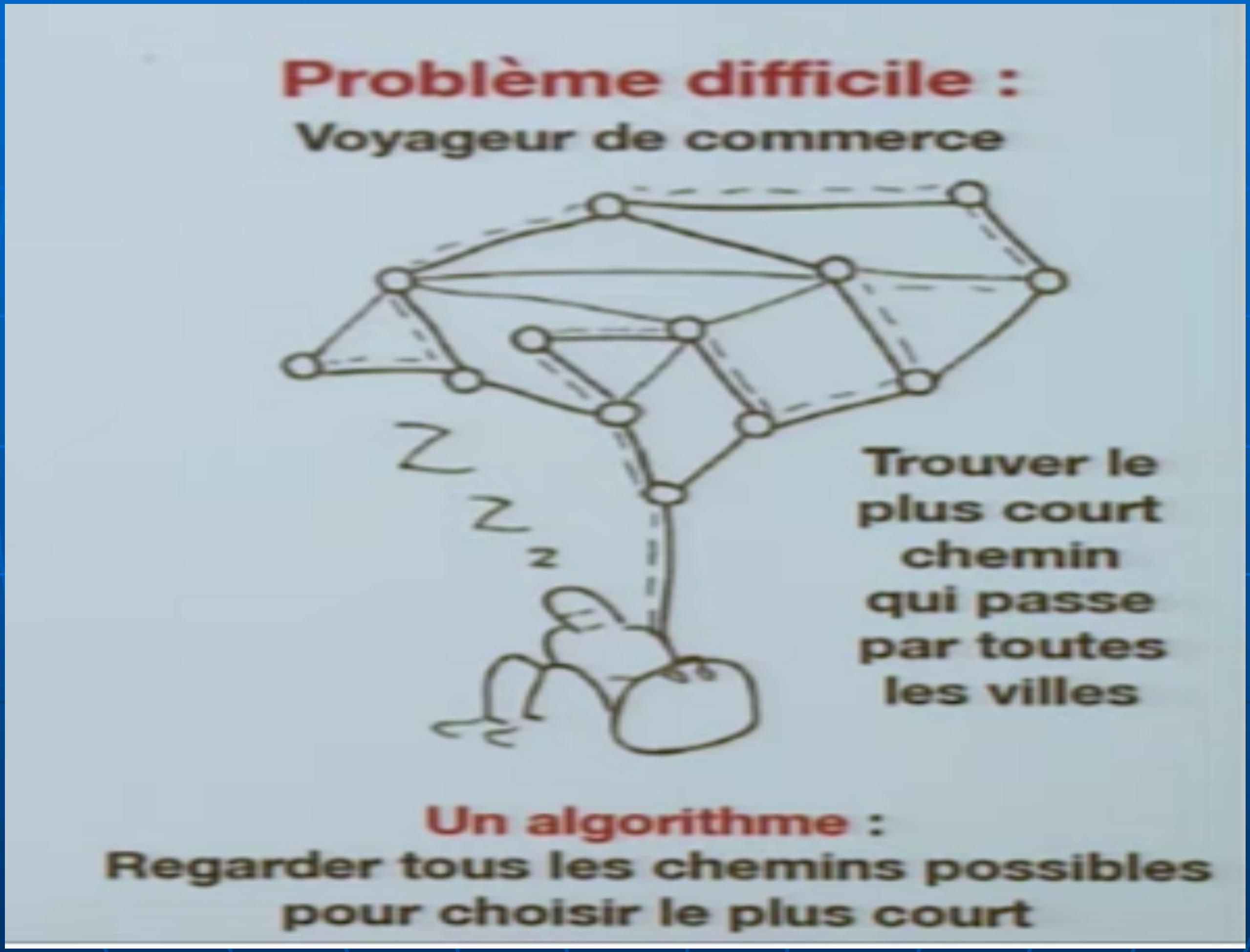
- Pour évaluer l'efficacité de l'algorithme de recherche séquentielle, on va calculer sa complexité dans le pire des cas. Pour cela on va compter le nombre de tests effectués
- Le pire des cas pour cet algorithme correspond au cas où x n'est pas dans le tableau T
- Si x n'est pas dans le tableau, on effectue 3N tests : on répète N fois les tests (i < N), (Trouvé=Faux) et (T[i]=x)
- La **complexité** dans le pire des cas est **d'ordre N**, Notation de Landau (on note **O(N)**)
- Pour un ordinateur qui effectue 10^6 tests par seconde on a :

N	10^3	10^6	10^9
temps	1ms	1s	16mn40s

Exemple de problème difficile

- on doit accomplir des tâches T1, T2, . . . , Tn le coût de la réalisation de l'ensemble de ces tâches dépend de l'ordre dans lequel elles sont accomplies ;
- pour déterminer le meilleur ordre il faut considérer tous ceux possibles et pour chacun d'entre eux déterminer son coût,
- Il y a **n! ordres** possibles ce qui fait beaucoup : c'est un nombre qui croît de manière *exponentielle* avec n.
- Si la détermination du coût d'un ordre prend par exemple **10^{-8} secondes pour 20 tâches** (ce qui est une performace à peine atteignable par les ordinateurs actuels , le temps de calcul pour évaleur tous les ordres sera de l'ordre de **$(20!)10^{-8}$** secondes (soit de l' ordre de 2.10^{10} secondes), c'est à dire plusieurs milliers d'années.
- On remarque ainsi les limites de la puissance de l'informatique.

Exemple célèbre



Exemple 1 : calculabilité des algorithmes

- Supposons que l'on dispose de deux ordinateurs.
- L'ordinateur A est capable d'effectuer 10^9 instructions par seconde.
- L'ordinateur B est capable d'effectuer 10^7 instructions par seconde.
- Considérons un même problème (de tri par exemple) dont la taille des données d'entrées est n.
- Pour l'ordinateur A, on utilise un algorithme qui réalise $2n^2$ instructions.
- Pour l'ordinateur B, on utilise un algorithme qui réalise $50n\log_2(n)$ instructions.
- Pour traiter une entrée de taille 10^6 : l'ordinateur A prendra 2000 s et l'ordinateur B prendra 100 s. Ainsi, même si la machine B est médiocre, elle résoudra le problème 20 fois plus vite que l'ordinateur A.

Exemple 2 : calculabilité des algorithmes

- On suppose qu'une vieille machine peut exécuter 10000 opérations en une seconde, et que la nouvelle machine est dix fois plus rapide.
- La première colonne donne la complexité de l'algorithme
- la deuxième la plus grande taille de problème soluble en une seconde sur la vieille machine
- la suivante donne la plus grande taille de problème soluble en une seconde sur la nouvelle machine.
- Il est clair que si l'algorithme utilisé est de complexité exponentielle, peu importe les progrès en architecture des machines : le problème restera de toute façon au-delà des capacités du programme

$f(n)$	n	n'	Gain nouv machine	n'/n
$10n$	1000	10000	$n' = 10n$	10
$20n$	500	5000	$n' = 10n$	10
$5n \log n$	250	1842	$\sqrt{10}n \leq n' \leq 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
n^3	23	48	$n' = \sqrt[3]{10}n$	2.08
2^n	13	16	$n' = n + 3$	–

Définitions

- L'**efficacité d'un algorithme est mesurée par son coût (complexité) en temps et en mémoire**
- La **complexité d'un algorithme est donc :**
  - en temps, le nombre d'opérations élémentaires effectuées pour traiter une donnée de taille n ;
  - en mémoire, l'espace mémoire nécessaire pour traiter une donnée de taille n.

Définitions

- Un problème **NP-complet est un problème pour lequel on ne connaît pas d'algorithme correct efficace, c'est-à-dire** réalisable en temps et en mémoire.
- Le problème le plus célèbre est le **problème du voyageur de commerce.**
- L'ensemble des problèmes NP-complets ont les propriétés suivantes :
  - si on trouve un algorithme efficace pour un problème NP complet alors il existe des algorithmes efficaces pour tous ;
  - personne n'a jamais trouvé un algorithme efficace pour un problème NP-complet ;
- Une **heuristique est une procédure de calcul correcte pour certaines instances du problème (c'est-à-**dire se termine **ou produit une sortie correcte).**

Exemple de complexité

- Complexité en n

- Complexité en $2n^2$

- Ordre de grandeur pour n grand
  - N=> si n*10, temps *10
  - N^2=> si n*10, temps *100

Efficacité

- Simuler une partie d'échec:
  - Trop d'états à mettre en mémoire
  - On choisit des choses non-optimales mais efficaces
- Voyageur de commerce
  - Comment minimiser le trajet du voyageur de commerce allant de villes en villes
    - Enoncé simple, mais solution très difficile si nombre de villes est grand (n! possibilités)
      - N=5 =>120, n=15 => 1300 G

- Le temps d’exécution d’un algorithme dépend des facteurs suivants :
    - Les données du programme,
    - La qualité du compilateur (langage utilisé),
    - La machine utilisée (vitesse, mémoire,),
    - La complexité de l’algorithme lui-même,
- On cherche à mesurer la complexité d’un algorithme indépendamment de la machine et du langage utilisés, c.- à-d. uniquement en fonction de la taille des données que l’algorithme doit traiter.

Revisions :

Voir les videos suivantes :

https://www.youtube.com/watch?v=clZ4q5zPBlE

Et

https://www.youtube.com/watch?v=exaHKrP6RsA

Exemple : Calcul de la valeur d'un polynôme

- Soit P(X) un polynôme de degré n
- $P(X) = a_nX^n + a_{n-1}X^{n-1} + \ldots + a_1X + a_0$
- Où,
  - n : entier naturel
  - $a_n, a_{n-1}, \ldots, a_1, a_0$: les coefficients du polynôme
- 1ère variante :

```
début
P=0
Pour i de 0 à n faire
P = P+ ai*X^i
finpour
fin
```

Coût de l'algorithme :
-(n+1) additions
-(n+1) multiplications
-(n+1) puissances

Exemple : Calcul de la valeur d'un polynôme

- 2ème variante :

debut

Inter=1   P =0

Pour i de 0 à N faire

  $P = P + Inter * a_i$

  Inter = Inter * X

finpour

Fin

Coût de l'algorithme :

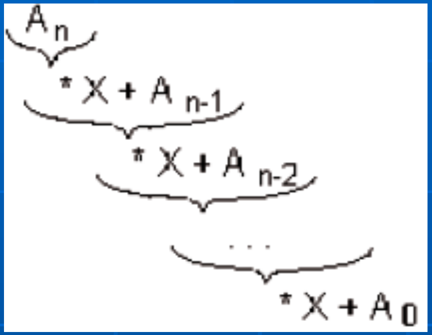
-(n+1) additions

-2(n+1) multiplications

Exemple : Calcul de la valeur d'un polynôme

3ème variante : Schéma de Horner

$P(x) = (\ldots.(((a_n x+a_{n-1})x+a_{n-2})x+a_{n-3})\ldots..)x+a_0$



début

P = a_n

Pour i de n-1 à 0 (pas = –1) faire

  P = P*X + a_i

finpour Fin

Coût de l'algorithme :

-n additions

-n multiplications

➔ Nécessité d'estimer le coût d'un algorithme avant de l'écrire et l'implémenter

TYPE DE LA COMPLEXITÉ

Complexité au meilleur

- C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille n.
- $\mathbf{T_{min}(n) = min_{d \in D_n} T(d)}$

Complexité en moyenne

- C'est la moyenne des complexités de l'algorithme sur des jeux de données de taille n
- $\mathbf{T_{moy}(n) = \Sigma_{d \in D_n} T(d) \,/\, |D_n|}$

Complexité au pire

- C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille n
- $\mathbf{T_{max}(n) = max_{d \in D_n} T(d)}$

- **Notations:**
  - $\mathbf{D_n}$ l'ensemble des données de taille **n**
  - **T(n)** le nombre d'opération sur un jeu donnée de taille **n**

NOTATION DE LANDAU

- **La notation de Landau « O »** est celle qui est le plus communément utilisée pour expliquer formellement les performances d'un algorithme.
  - Cette notation exprime **la limite supérieure d'une** fonction dans un facteur constant.

$$f = O(g) \quad \Leftrightarrow \quad \exists n_0, \exists c \geq 0, \forall n \geq n_0, f(n) \leq c \times g(n)$$

- **Exemple:** $T(n) = O(n^2)$ veut dire qu'il existe une constante $c > 0$ et une constante $n_0 > 0$ tel que pour tout $n > n_0$ $T(n) <= c\, n^2$

NOTATION DE LANDAU

- Les règles de la notation O sont les suivantes :
  - Les termes constants : **O(c) = O(1)**
  - Les constantes multiplicatives sont omises :

$$\mathbf{O(cT) = c\,O(T) = O(T)}$$

  - L'addition est réalisée en prenant le maximum :

$$\mathbf{O(T1) + O(T2) = O(T1 + T2) = max(O(T1);O(T2))}$$

  - La multiplication reste inchangée

$$\mathbf{O(T1)O(T2) = O(T1T2)}$$

NOTATION DE LANDAU

- Supposant que le temps d'exécution d'un algorithme est décrit par la fonction $T(n) = 3n^2+10n+10$, **Calculer $O(T(n))$?**

$O(T(n)) = O(3n^2 + 10n + 10)$

$= O(\max(3n^2, 10n, 10))$

$= O(3n^2)$

$= O(n^2)$

- **Remarque:**

Pour n = 10 nous avons :

- Temps d'exécution de $3n^2$: 3(10)2 / 3(10)2+10(10)+10 = 73,2%
- Temps d'exécution de 10n : 10(10) / 3(10)2+10(10)+10 = 24,4%
- Temps d'exécution de 10 : 10 / 3(10)2+10(10)+10 = 2,4%

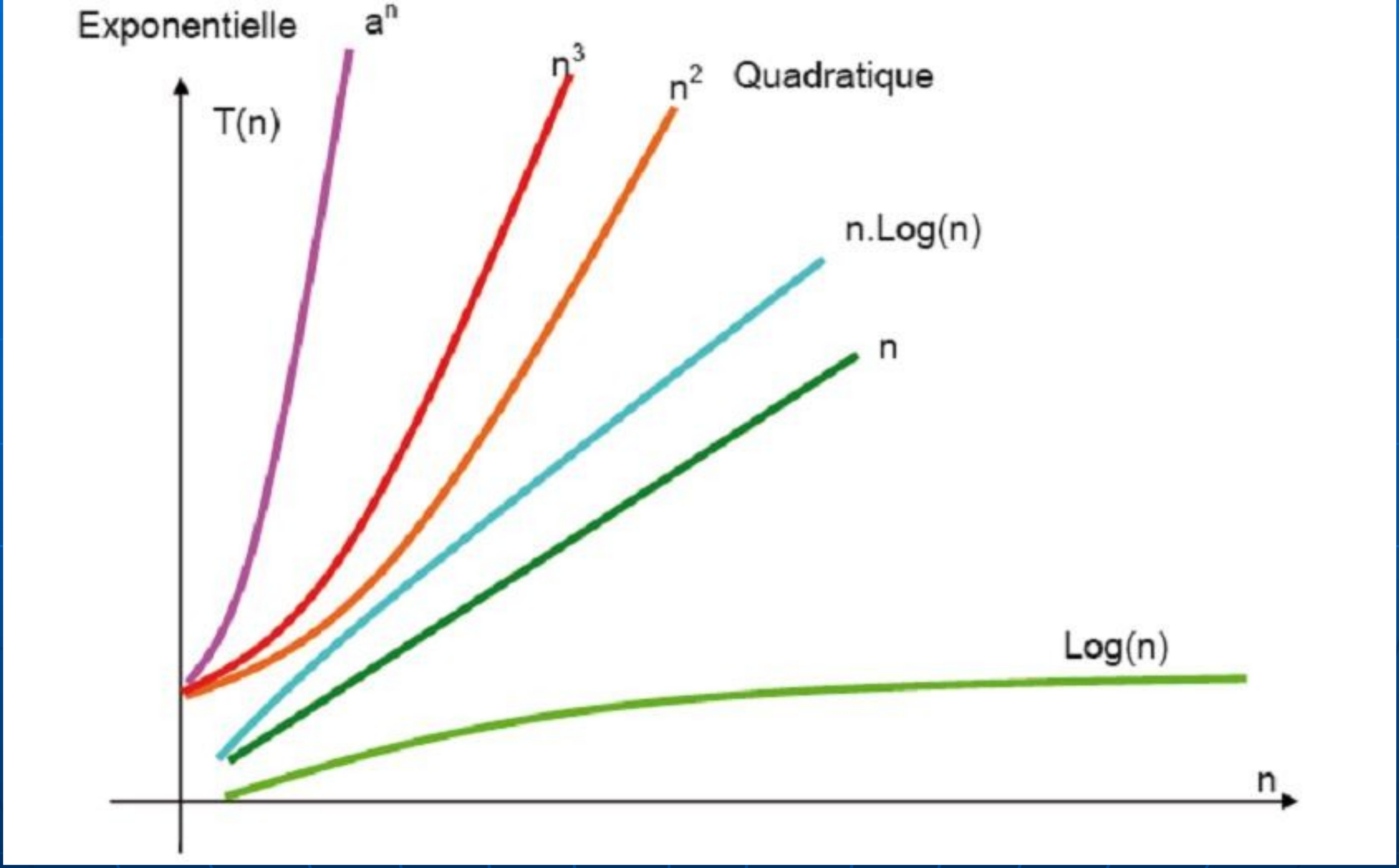
Le poids de $3n^2$ devient encore plus grand quand n = 100, soit 96,7%

On peut négliger les quantités 10n et 10. **Ceci explique les règles de la notation O.**

CLASSES DE COMPLEXITÉ

Classe	Notation O	Exemple
Constante	O(1)	Accéder au premier élément d'un ensemble de données
Linéaire	**O(n)**	**Parcourir un ensemble de données**
Logarithmique	O(log(n))	Couper un ensemble de données en deux parties égales, puis couper ces moitiés en deux parties égales, etc.
Quasi-linéaire	**O(n log(n))**	**Couper répétitivement un ensemble de données en deux et combiner les solutions partielles pour calculer la solution générale**
Quadratique	**$O(n^2)$**	**Parcourir un ensemble de données en utilisant deux boucles imbriquées**
Polynomiale	**$O(n^P)$**	**Parcourir un ensemble de données en utilisant P boucles imbriquées**
Exponentielle	**$O(a^n)$**	**Générer tous les sous-ensembles possibles d'un ensemble de données**

cLASSES DE COMPLEXITÉ



CALCUL DE LA cOMPLEXITÉ

1. **Cas d'une instruction simple (écriture, lecture, affectation) :**

Le temps d'exécution de chaque instruction simple est O(1).

1. **Cas d'une suite d'instructions simples**: Le temps d 'exécution d'une séquence d'instruction est déterminée par la règle de la somme. C'est donc le temps de la séquence qui a le plus grand temps d 'exécution: O(T) = O (T1 + T2) = max(O(T1);O(T2)) .

$$\left.\begin{array}{ll} \text{Traitement1} & T_1(n) \\ & \\ \text{Traitement2} & T_2(n) \end{array}\right\} \quad T(n) = T_1(n) + T_2(n)$$

CALCUL DE LA cOMPLEXITÉ

- **Exemple 2:**

Permutation (Var S: tableau [0..n-1] d'entier, i, j: entier)

tmp$\leftarrow$S[i]	**O(T1) = O(1)**
S[i]$\leftarrow$S[j]	**O(T2) = O(1)**
S[j]$\leftarrow$tmp	**O(T3) = O(1)**

$$O(T) = O(T1 + T2 + T3) = O(1)$$

CALCUL DE LA COMPLEXITÉ

3. **Cas d'un traitement conditionnel:**

Le temps d'exécution d'une instruction SI est le temps d 'exécution des instructions exécutées sous condition, plus le temps pour évaluer la condition. Pour une alternative, on se place dans le cas le plus défavorable.

Si (condition) **Alors**
  | Traitement1
Sinon
  | Traitement2
Fin Si
/* $O(\mathrm{T}_{condition}) + \max(O(\mathrm{T}_{Traitement1}), O(\mathrm{T}_{Traitement2}))$ */

CALCUL DE LA COMPLEXITÉ

4. **Cas d'un traitement itératif :**

Le temps d 'exécution d'une boucle est la somme du temps pour évaluer le corps et du temps pour évaluer la condition.

Remarque : Souvent ce temps est le produit du nombre d'itérations de la boucle par le plus grand temps possible pour une exécution du corps.

Boucle Pour

Pour i **de** indDeb **à** indFin **faire**
| Traitement
Fin Pour

$$\sum_{i=indDeb}^{indFin} O(T_{Traitement})$$

Boule Tant que

Tant que (condition) faire
| Traitement
Fait

$/^{*}$ nombre d'itérations $^{*}(O(T_{condition})+O(T_{Traitement}))$

CALCUL DE LA COMPLEXITÉ

- **Exemple 2:**

Recherche séquentielle (S: tableau [0..n-1] d'entier, x: entier): booléen

i ← 0	O(1)
Trouve ← faux	O(1)
Tant que ((i<n) et (non trouve)) faire	Condition = O(1); nombre d'itération = n
Si (S[i] = x) alors	
Trouve ← vrai	
i ← i + 1	
FinTantQue	
Retourner trouve	

O(T) = max(O (1) + O (n *1) + O(1)) = O (n)

Complexité du Tri par Insertion

- Exemple 1 : Tri par insertion
  - Principe : Cette méthode de tri s'apparente à celle utilisée pour trier ses cartes dans un jeu : on prend une carte, tab[1], puis la deuxième, tab[2], que l'on place en fonction de la première, ensuite la troisième tab[3] que l'on insère à sa place en fonction des deux premières et ainsi de suite.
  - Le principe général est donc de considérer que les (i-1) premières cartes, tab[1],..., tab[i-1] sont triées et de placer la i^e carte, tab[i], à sa place parmi les

(i-1) déjà triées, et ce jusqu'à ce que i = N.

Complexité du Tri par Insertion

Exemple 1 : Tri par insertion

```
Procédure tri_Insertion (var tab : tableau entier [N])
i, j :entier ;  x: entier ;
Pour i de 2 à N faire
    x← tab[i];
    k ← i - 1;
    Tant que j > 0 ET tab[j] > x faire
            tab[j+1] ← tab[j];
            j ← j - 1;
    Fin Tant que
tab[j+1] ← x;
Fin pour
Fin
```

Complexité du Tri par Insertion

- Exemple 1 : Tri par insertion

Faire la trace pour

T = [3,1,4,0,5], U = [0,3,4,6,9] et V = [9,6,4,3,0]

Complexité du Tri par Insertion

Trace de insert(T)

i	x	j	j > 0 et T[j] > x	T
-	-	-	-	3 1 4 0 5
2	1	1	oui	
		0	non	**1 3** 4 0 5
3	4	2	non	
4	0	3	oui	
		2	oui	1 3 **0 4** 5
		1	oui	1 **0 3** 4 5
		0	non	**0 1** 3 4 5
5	5	4	non	0 1 3 4 5

Trace de insert(U) et insert(V)

i	x	j	U
-	-	-	0 3 4 6 9
2	3	1	
3	4	2	
4	6	3	
5	9	4	

i	x	j	V
-	-	-	9 6 4 3 0
2	6	1	
		0	**6 9** 4 3 0
3	4	2	
		1	6 **4 9** 3 0
		0	**4 6** 9 3 0
4	3	3	
		2	4 6 **3 9** 0
		1	4 **3 6** 9 0
		0	**3 4** 6 9 0
5	0	4	
		3	3 4 6 **0 9**
		2	3 4 **0 6** 9
		1	3 **0 4** 6 9
		0	**0 3** 4 6 9

Complexité du Tri par Insertion

Exemple 1 : Tri par insertion

- Calcul de la complexité:
  - la taille du tableau à trier est *n.*
  - *On a deux boucles* imbriquées :
    - La première indique l'élément suivant à insérer dans la partie triée du tableau.
    - Elle effectuera *n - 1 itérations puisque le premier élément est déjà* trié.
    - Pour chaque élément donné par la première boucle, on fait un parcourt dans la partie triée pour déterminer son emplacement.

Complexité du Tri par Insertion

Exemple 1 : Tri par insertion

- Calcul de la complexité:
  - **Au meilleur des cas :** le cas le plus favorable pour cet algorithme est quand le tableau est déjà trié (de taille n) ➔ O(n)
  - **Au pire des cas :** Le cas le plus défavorable pour cet algorithme est quand le tableau est inversement trié ➔ on fera une itération pour le 1er élément, deux itérations pour le 2ème et ainsi de suite pour les autres éléments.

Soit 1+2+3+4+…+*(n-1)* = $\frac{n(n+1)}{2} - n$

➔ sa complexité $O(n^2)$

Complexité du Tri par Insertion

Exemple 1 : Tri par insertion

- Calcul de la complexité:
  - **En moyenne des cas :** En moyenne, la moitié des éléments du tableau sont triés, et sur l'autre moitié ils sont inversement triés.

➔ $O(n^2)$

Complexité du Tri par sélection

- ***Principe :***
- Le principe est que pour classer **n** valeurs, il faut rechercher la plus petite valeur (resp. la plus grande) et la placer au début du tableau (resp. à la fin du tableau),
- puis la plus petite (resp. plus grande) valeur dans les valeurs restantes et la placer à la deuxième position (resp. en avant dernière position) et ainsi de suite...

Complexité du Tri par sélection

```
Procédure select (T[1..n])
    pour i ← 1 jusqu'à n-1 faire
    minj ← i,
    minx ← T[i]
    pour j ← i + 1 jusqu'à n  faire
        si T[j] < minx alors
            minj ← j
            minx ← T[j]
        FinSi
        T[minj] ← T[i]
            T[i] ← minx
    FinPour
FinPour
```

Exercice : Faire la trace pour
T = [3,1,4,0,5], U = [0,3,4,6,9] et V = [9,6,4,3,0]

Complexité du Tri par sélection

Trace de select(T)

i	j	minj	minx	T
-	-	-	-	3 1 4 0 5
1	-	1	3	
	2	2	1	
	3	2	1	
	4	**4**	**0**	
	5	4	0	**0** 1 4 **3** 5
2	-	2	1	
	3	2	1	
	4	2	1	
	5	2	1	0 1 4 3 5
3	-	3	4	
	4	**4**	**3**	
	5	4	3	0 1 **3 4** 5
4	-	4	4	
	5	4	4	0 1 3 4 5

Trace de select(U) et select(V)

i	j	minj	minx	U
-	-	-	-	0 3 4 6 9
1	-	1	0	
	2	1	0	
	3	1	0	
	4	1	0	
	5	1	0	0 3 4 6 9
2	-	2	1	
	3	2	3	
	4	2	3	
	5	2	3	0 3 4 6 9
3	-	3	4	
	4	3	4	
	5	3	4	0 3 4 6 9
4	-	4	6	
	5	4	6	0 3 4 6 9

i	j	minj	minx	V
-	-	-	-	9 6 4 3 0
1	-	1	9	
	2	**2**	**6**	
	3	**3**	**4**	
	4	**4**	**3**	
	5	**4**	**0**	**0** 6 4 3 **9**
2	-	2	6	
	3	**3**	**4**	
	4	**4**	**3**	
	5	4	3	0 **3** 4 **6** 9
3	-	3	4	
	4	3	4	
	5	3	4	9 6 4 3 0
4	-	4	6	
	5	4	6	9 6 4 3 0

Complexité du Tri par selection

Procédure select (T[1...n])
pour i ← 1 **jusqu'à** n-1 **faire**
minj ← i, minx ← T[i]
pour j ← i+1 **jusqu'à** n **faire**
si T[j] < minx **alors**
minj ← j
minx ← T[j]
T[minj] ← T[i]
T[i] ← minx

a
b c
e
d

$t(n) = \Sigma_{1 \le i \le n-1} [\, a + \Sigma_{i+1 \le j \le n} (b) + d \,] = (a + d + bn)(n - 1) - bn(n-1)/2$
$\Rightarrow t(n) \in O(n^2)$

Complexité du Tri par propagation / à bulles

- ***Principe :***
- Il consiste à parcourir le tableau tab en permutant toute paire d'éléments consécutifs (tab[k],tab[k+1]) non ordonnés - ce qui est un échange et nécessite donc encore une variable intermédiaire de type entier.
- Après le premier parcours, le plus grand élément se retrouve dans la dernière case du tableau, en tab[N], et il reste donc à appliquer la même procédure sur le tableau composé des éléments tab[1], ..., tab[N-1].

Complexité du Tri par propagation / à bulles

Algorithme :

Procédure **tri_Bulle** (tab : tableau entier [N]) i, k :entier ;tmp : entier ;

```
Pour i de N à 2 faire
    Pour k de 1 à i-1 faire
            Si (tab[k] > tab[k+1]) alors
                tmp ← tab[k];
                tab[k] ← tab[k+1];
                tab[k+1] ← tmp;
            Fin si
     Fin pour
Fin pour
```

➔ $T(n) = O(n^2)$