Organisation du module + quelques rappels sur la complexité algorithmique

Lundi 26 Janvier 2015

Michael FRANÇOIS

michael.francois@esiea.fr

Organisation du module

- Coefficient du module : 3
- Cours $\Longrightarrow$ 1h30 / Cours (18h au total)
  - complexité algorithmique et programmation,
  - paradigmes de programmation (itératifs vs récursif),
  - Stratégies gauche-droite et droite-gauche,
  - méthode diviser pour régner,
  - programmation dynamique,
  - algorithmes voraces,
  - introduction aux algorithmes sur les graphes (si temps),
  - résolution de problèmes (TD/TP).
- TDs $\Longrightarrow$ 1h30 / TD (18h au total)
  - travail sur papier, application du cours, etc.
- TPs $\Longrightarrow$ $2 \times$ 1h30 $=$ 3h / TP (36h au total)
  - programmation C $+$ un mini-projet à la fin

L'équipe des intervenants

Intervenant	Encadrements habituels	Encadrement INF1032
N. IZRI	1A/2A	TDs + TPs
M. FRANÇOIS	1A/2A/3A/MS SIS	Cours + TDs + TPs

Séquencement des CMs/TDs/TPs

SEQUENCEMENT 1A-S2																			
	20	20	20	0	20	20	20	20	20	20	18	20	0	23	18	13	23	18	23
	15	13	18	0	14	17	20	18	18	20	16	16	0	13	3	7	57	3	0
1A Paris	s5	s6	s7	s8	s9	s10	s11	s12	s13	s14	s15	s16	s17	s18	s19	s20	s21	s22	s23
Intervenant	26/01/2015	02/02/2015	09/02/2015	16/02/2015	23/02/2015	02/03/2015	09/03/2015	16/03/2015	23/03/2015	30/03/2015	06/04/2015	13/04/2015	20/04/2015	27/04/2015	04/05/2015	11/05/2015	18/05/2015	25/05/2015	01/06/2015
																	fil rouge		
M. François (gpe 11+gpe 12)	C	C	C		C	C	C	C	C	C	C	C		C			fil rouge		
																	fil rouge		
gpe 11 François	td	tdo	td		td	td	td	td	td	td	td	td		td			fil rouge		
idem gpe 12 N. Izri																	fil rouge		
																	fil rouge		
gpe 11 François					tpi	tpi	tpi	tpi	tpi	tpi	tpi	tpi		tpi		tpi	fil rouge	sout	
idem gpe 12 N. Izri					tpi	tpi	tpi	tpi	tpi	tpi	tpi	tpi		tpi		tpi	fil rouge	sout	
																tpi	fil rouge	sout	

Quelques règles de "savoir-vivre"

- Cours : assiduité, application, et préparation avant le prochain TD.
- TDs : assiduité, anticipation, application.
- TPs : assiduité, application, autonomie, persévérance.
- Conseils : programmer régulièrement seul chez soi au moins 3h/semaine, cela permet non seulement de ne pas oublier les syntaxes d'utilisation mais aussi de s'améliorer.

Rôle des algorithmes en informatique

Rôle des algorithmes en informatique

• Qu'est-ce qu'un **algorithme** ? Quelle est l'utilité d'un algorithme ?

• **Algorithme :** procédure de calcul bien définie prenant en entrée un ensemble de valeurs (ou non), et donnant en sortie un résultat.

• On peut considérer un **algorithme** comme un outil de calcul permettant de résoudre un problème bien déterminé :

- Exemple : trier une suite de nombres donnés en entrée, dans l'ordre croissant.
  $(12, 57, 4, 48, 94, 79, 25) \Longrightarrow (4, 12, 25, 48, 57, 79, 94)$

• Un **algorithme** peut être spécifié en langage humain ou en langage informatique, mais peut aussi être basé sur un système matériel.

Qualités d'un algorithme

• **Correct :** un algorithme est dit correct si, pour chaque instance en entrée, il se termine en produisant la bonne sortie.

• **Complet :** si le programme considère tous les cas possibles et donne un résultat dans pour chacun des cas.

• **Efficace :** si le programme exécute sa tache avec efficacité, c'est-à-dire avec un coût minimal. le coût pour un ordinateur se mesure en termes de temps de calcul et d'espace mémoire nécessaire.

Types de problèmes pouvant être résolus par des algorithmes

- **Génome humain :**
  - identification des 100 000 gènes de l'ADN humain,
  - détermination des 3 milliards de paires de bases chimiques constituant l'ADN humain,
  - stockage de ces informations dans des base de données,
  - développement des outils interroger ces bases de données,
  - etc.
- **Internet :** protection de données sensibles transitant sur internet exemples : cartes de crédit, mots de passe, etc. Toutes ces technologies s'appuient sur des algorithmes numériques très sophistiqués.
- **Carte routière :** déterminer le chemin le plus court entre deux points A et B, parmi un ensemble de trajets possibles. Ceci permettrait par exemple de minimiser les coûts pour un automobiliste.
- etc.

Programme informatique

• Un **programme informatique** c'est avant tout un ensemble d'algorithmes connectés entre eux et destiné à une tache donnée.

• [WIKI] Un **programme informatique** est la forme électronique et numérique d'un algorithme exprimé dans un langage de programmation. Il est exprimé sous une forme permettant de l'utiliser avec une machine (*i.e.* ordinateur) pour exécuter les instructions.

• [WIKI] C'est une séquence d'instructions spécifiant étape par étape les opérations à effectuer pour obtenir un résultat.

• **Programme source :** est généralement écrit dans un langage de programmation permettant ainsi une meilleure compréhension par des humains. Il se matérialise souvent sous la forme d'un ensemble de fichiers textes.

• **Programme objet ou binaire :** est la forme finale du programme, qui est générée à partir du programme source et exécutable par une machine (*i.e.* ordinateur).

Des programmes sont présents dans tous les appareils informatiques :

- ordinateur,
- console de jeu,
- distributeur automatique de billets,
- imprimante,
- téléphone portable,
- décodeur TV numérique,
- GPS,
- appareil photo numérique,
- ...

Analyse de complexité

Analyse de complexité

• La **complexité** d'un algorithme est la mesure du nombre d'opérations fondamentales effectuées sur un ensemble de données.

• La **complexité** est exprimée en fonction de la taille des données en entrée.

NB : généralement plus la taille de l'entrée est grande, plus il faudra du temps à l'algorithme pour trouver la solution. Par exemple, il est évident qu'on mettra plus de temps à trier une liste d'un million d'éléments plutôt qu'une liste de cent éléments. Il est alors naturel d'évaluer le temps de calcul d'un algorithme en fonction de la taille de son entrée.

- L'efficacité d'un algorithme peut être évaluée 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'algorithme pendant son exécution.

Complexité en temps

• Le temps d'exécution d'un algorithme varie effectivement selon la taille des données en entrée.

• On distingue généralement trois types de complexité :

1. **Complexité au meilleur** (*i.e.* complexité dans le meilleur des cas)
2. **Complexité au pire** (*i.e.* complexité dans le pire des cas)
3. **Complexité en moyenne**

On considère :

- D_n l'ensemble des données de taille n,
- $C(d)$ le coût d'exécution de l'algorithme sur la donnée d.

• **Complexité au meilleur :**
elle correspond à la complexité de l'algorithme dans le cas le plus favorable. Elle est donnée par le plus petit nombre d'opérations à effectuer sur un ensemble de données de taille fixée (ici n) :

$$T_{\min}(n) = \min_{d \in D_n} C(d)$$

Remarque : c'est en quelque sorte la borne inférieure de la complexité de l'algorithme pour un ensemble de données de taille n.

• **Complexité au pire :**
elle correspond à la complexité de l'algorithme dans le cas le plus défavorable. Elle est donnée par le plus grand nombre d'opérations à effectuer sur un ensemble de données de taille fixée (ici n) :

$$T_{\max}(n) = \max_{d \in D_n} C(d)$$

Remarque : il s'agit ici du maximum. Généralement, l'algorithme finira son exécution avant même d'atteindre $T_{\max}(n)$ opérations. Même si ce cas se produit rarement, il ne doit pas être négligé.

• **Complexité en moyenne :**
elle correspond à la moyenne des complexités de l'algorithme sur un ensemble de données de taille fixée (ici n) :

$$T_{\text{moy}}(n) = \sum_{d \in D_n} Pr(d) \cdot C(d),$$

où Pr(d) correspond à la probabilité d'avoir la donnée d comme entrée de l'algorithme.

Remarque : cette complexité indique le comportement "habituel" de l'algorithme, dans le cas où les cas extrêmes sont rares ou si la complexité varie peu en fonction des données.

Remarques :

• En pratique, on s'intéresse qu'à la complexité au pire et en moyenne. On a :

$$T_{\text{min}}(n) \leq T_{\text{moy}}(n) \leq T_{\text{max}}(n)$$

• L'analyse dans le pire des cas indique la limite supérieure de la performance et elle garantit qu'un algorithme ne fera jamais moins bien que ce qui a été fixé.

• Dans la pratique, la complexité en moyenne est beaucoup plus dure à déterminer que la complexité dans le pire cas, car l'analyse peut devenir en tout mathématiquement abstraite, et aussi parce qu'il n'est pas toujours facile de définir un modèle de probabilité convenable au problème.

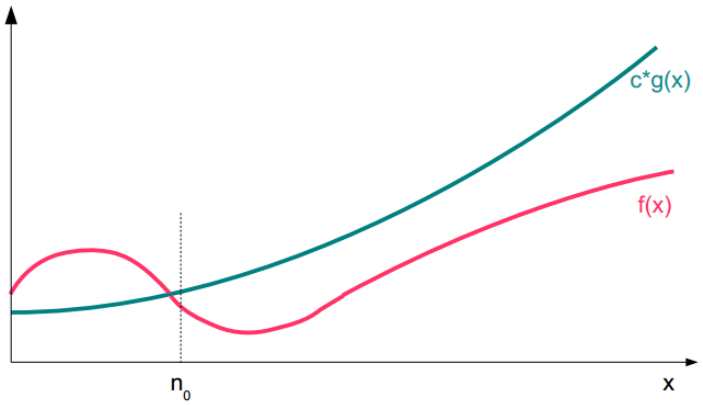
• Un algorithme est dit **optimal**, si sa complexité correspond à la complexité minimal parmi tous les algorithmes de sa classe.

Notations mathématique

• La notation O (dite également notation de Landau) est celle qui est le plus communément employée pour décrire les performances d'un algorithme.

• f et g étant des fonctions, $f = O(g)$ signifie que f est dominée asymptotiquement par g.
Autrement dit :

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



Quelques propriétés de base :

- Les termes constants : $O(c) = O(1)$
- Les constantes multiplicatives sont négligées : $O(cf) = cO(f) = O(f)$
- On prend le maximum pour l'addition : $O(f) + O(g) = O(f + g) = \max(O(f), O(g))$
- $O(f) \cdot O(g) = O(f \cdot g)$

Exemple :

• Considérons un algorithme dont le temps d'exécution est décrit par la fonction :

$$T(n) = 3n^2 + 15n + 9$$

En utilisant la notation O, on obtient :

$$O(T(n)) = O(3n^2 + 15n + 9) = O(3n^2) = 3O(n^2) = O(n^2)$$

• Pour $n = 25$, on obtient :

Fonction	Temps d'Exécution	TE par rapport à T(n)
T(n)	2259	100 %
$3n^2$	1875	83 %
$15n$	375	16.60 %
9	9	0.40 %

Remarque : on voit que le terme $3n^2$ l'emporte sur les autres pour $n = 25$. Mais le terme $15n$ n'est pas à négligé pour cette taille de n. Qu'est-ce qui se passe si n est beaucoup plus grand ? par exemple pour $n = 1000$?

• Pour $n = 1000$, on obtient :

Fonction	Temps d'Exécution	TE par rapport à T(n)
T(n)	3015009	100 %
$3n^2$	3000000	99.50 %
$15n$	15000	0.50 %
9	9	0.00 %

Remarque : on voit que le terme $3n^2$ l'emporte extrêmement sur les autres pour $n = 1000$. Ici, on peut même négliger les autres termes. Et dans le cas où n est encore plus grand, les termes $15n$ et 9 seront encore plus petits et donc encore plus négligeables. Ceci montre bien l'utilité de la notation O.

Classification des complexités

- La plupart des algorithmes admettent un paramètre de base, n, qui est habituellement la taille des données à traiter dont dépend très intimement la complexité en temps.
- Les complexités les plus utilisées sont :
  - $O(1)$
  - $O(\log n)$
  - $O(n)$
  - $O(n \log n)$
  - $O(n^2)$
  - $O(n^p)$
  - $O(p^n)$

$O(1)$

• Complexité (en temps) **constante** : pas d'augmentation du temps d'exécution quand le paramètre croit.

• Une complexité constante est la complexité algorithmique idéale, puisque peu importe la taille de l'échantillon à traiter, l'algorithme prendra toujours un nombre fixé à l'avance d'opérations pour réaliser sa tâche.

• En général, qu'un algorithme soit $O(2)$, $O(94)$ ou $O(60000)$, on dira de lui qu'il est en fait en $O(1)$, puisque la différence de performance entre deux algorithmes en temps constant peut être comblée par une simple amélioration matérielle (*i.e.* utilisation d'un processeur plus rapide).

• Exemple : accéder à un élément d'un tableau.

$O(\log n)$

• Complexité **logarithmique** : augmentation très faible du temps d'exécution quand le paramètre croit.

• Lorsqu'un algorithme possède une complexité en $\log n$, celui-ci devient légèrement plus lent quand n croit :

- Si $n = 100$ alors $\log_{10} n = log_{10} 10^2 = 2$
- Si $n = 1000$ alors $\log_{10} n = log_{10} 10^3 = 3$
- Si $n = 1000000$ alors $\log_{10} n = log_{10} 10^6 = 6$
- ...

• Exemple : recherche par dichotomie.

$O(n)$

• Complexité **linéaire** : augmentation linéaire du temps d'exécution.

• Ici, le temps de l'algorithme croît proportionnellement en fonction de la taille de la donnée en entrée.

• Si la taille croît par un facteur de 10, alors la complexité sera accrue elle aussi d'un facteur 10.

• Exemple : parcourir les éléments d'un tableau.

$O(n \log n)$

• Complexité **quasi-linéaire** : augmentation un peu supérieure à $O(n)$.

• Cette complexité concerne les algorithmes qui résolvent un problème en le décomposant en d'autres plus simples, traités indépendamment et qui combinent les solutions partielles pour calculer la solution générale correspondante.

$O(n^2)$

• Complexité **quadratique** : à chaque fois que la taille d'entrée double, le temps d'exécution de l'algorithme est multiplié par 4.

• Lorsqu'un algorithme possède une complexité quadratique, il n'est utilisable que pour des problèmes relativement petits.

• Ces complexités sont fréquentes dans les algorithmes traitant tous les couples de données, utilisant souvent deux boucles imbriquées.

• Exemple : parcourir les éléments d'une matrice.

$O(n^p)$

• Complexité **polynomiale** : quand le paramètre double, le temps d'exécution est multiplié par 2^p.

• Par exemple si $p = 3$, l'algorithme traite des données par triplets via trois boucles imbriquées, ce qui donne une complexité cubique.

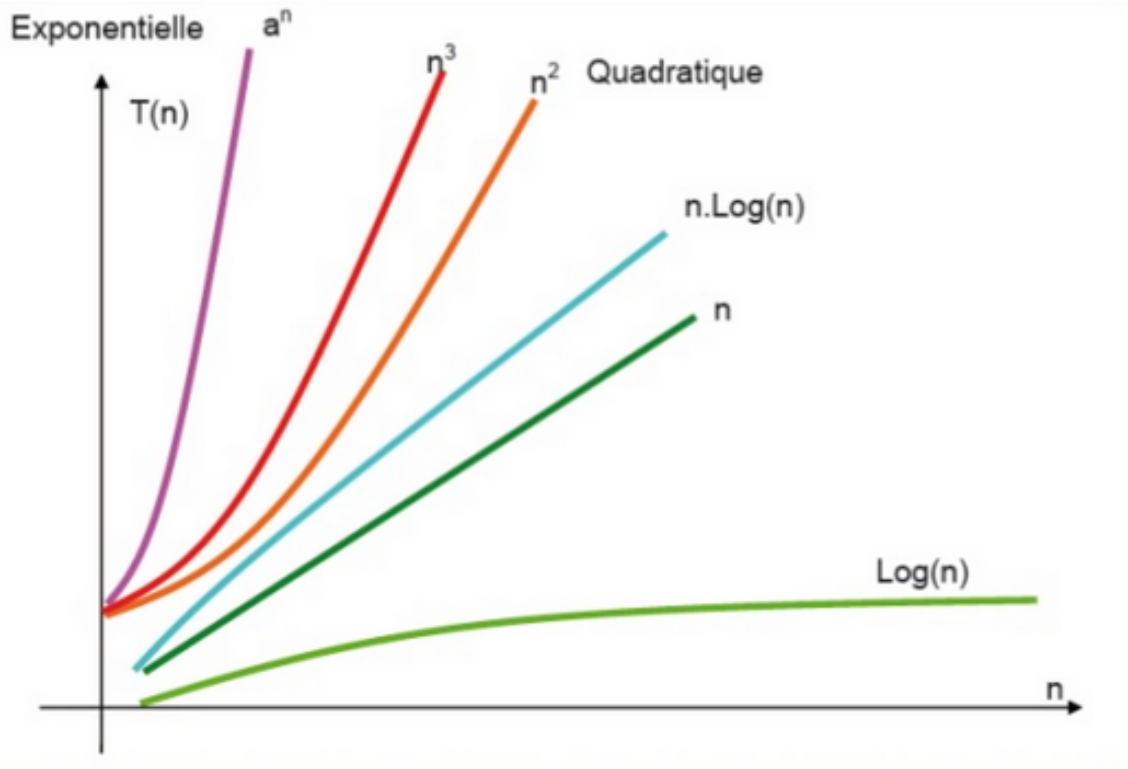
• Parcourir les éléments d'un tableau p-dimensionnel.

$O(p^n)$

• complexité **exponentielle** : chaque fois que n double de valeur, le temps de calcul est multiplié par p^2.

• En pratique, peu d'algorithmes de complexité exponentielles sont utilisés, bien que de tels algorithmes se rencontrent naturellement dans les solutions dites naïves.

• Exemple : générer tous les sous-ensembles possibles, d'une suite binaire de taille n (exple n=10).


Exponentielle
a^n
n^3
n^2
Quadratique
T(n)
n.Log(n)
n
Log(n)
n

Exemple 1 : cas d'instructions simples

```
x : entier
y : entier
x ⟵ 94; /* instruction en O(1) */
y ⟵ x²; /* instruction en O(1) */
```

La complexité totale est de : $\max(O(1), O(1)) = O(1)$.

Exemple 2 : cas d'un traitement conditionnel

```
Si (condition) alors
   | Traitement 1
Sinon
   | Traitement 2
Fin si
```

La complexité totale est de :
$O(T_{\text{condition}}) + \max(O(T_{\text{Traitement1}}), O(T_{\text{Traitement2}}))$.

Exemple 3 : cas d'un traitement itératif "boucle tant que"

```
Tant que (condition) faire
   | Traitement
Fin Tant que
```

La complexité totale est de :
Nombre d'itérations $\times$ $[O(T_{\text{condition}}) + O(T_{\text{Traitement}})]$.

Exemple 4 : cas d'un traitement itératif "boucle pour"

```
Pour i de dep à fin faire
   | Traitement
Fin Pour
```

La complexité totale est de :

$$\sum_{i=dep}^{fin} O(T_{\text{Traitement}})$$

NP-complétude

NP-complétude

• Il est naturel de se demander si tous les problèmes peuvent être résolus en temps polynomial.

• Globalement, on considère que les problèmes résolubles par des algorithmes à temps polynomial sont "faciles" et que les problèmes nécessitant un temps suprapolynomial sont difficiles.

• Ici, on va parler d'une classe de problèmes, appelés problèmes "`NP-complets`".

• Pour le moment aucun algorithme à temps polynomial n'a encore été découvert pour un problème `NP-complet`. Alors P $\neq$ NP ???????????

• Depuis que ce problème a été posé en 1971, il continue toujours de secouer le monde de l'informatique théorique.

Les classes **P**, **NP** et **NPC**

• La classe **P** se compose des problèmes qui sont résolubles en temps polynomial ($O(n^k)$).

• La classe **NP** se compose des problèmes qui sont "vérifiables" en temps polynomial. C'est-à-dire pour une solution donnée, on peut vérifier en temps polynomial que cette solution est correcte par rapport à la taille des données. On a **P** $\subseteq$ **NP**

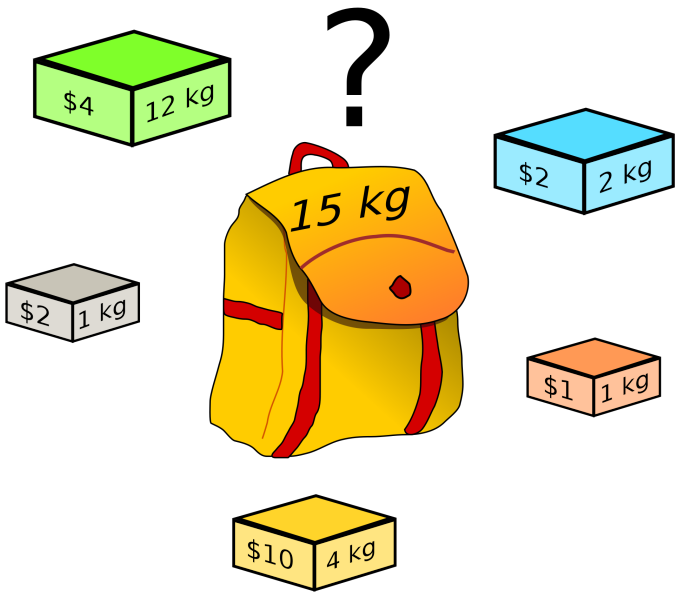
• Un problème est dit **NPC** (NP-complet) s'il appartient à **NP** et s'il est au moins aussi "difficile" que n'importe quel problème de **NP**. S'il existe un quelconque problème **NPC** résoluble en temps polynomial, alors tout problème **NPC** admet un algorithme à temps polynomial.

• On peut rencontrer également des problèmes dits **NP-difficile**. Un problème est dit **NP-difficile** s'il est au moins aussi difficile que tout problème de **NP**. Ce problème n'a pas besoin d'appartenir à la classe **NP**. On a **NP-complet** $\subseteq$ **NP-difficile**

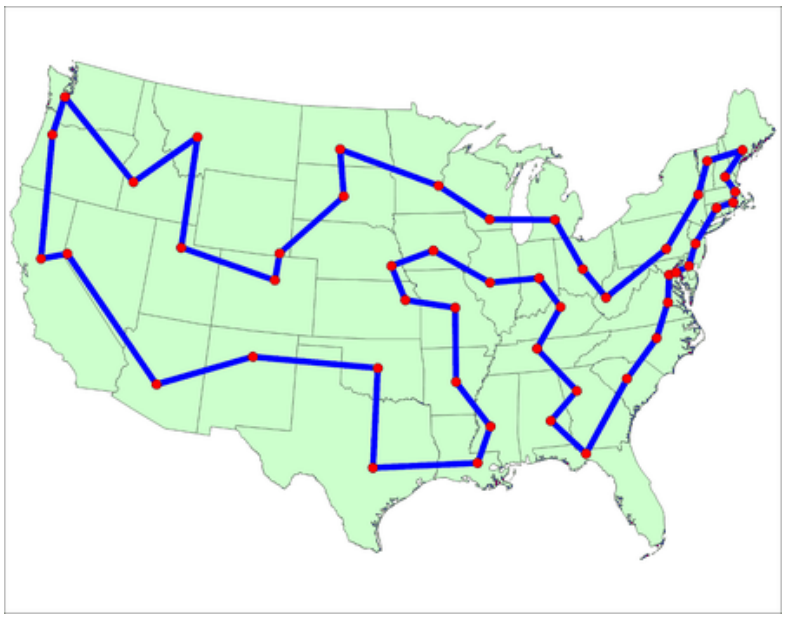
- Les problèmes **NP-complets** apparaissent dans les domaines variés :
  - logique booléenne,
  - graphes,
  - programmation mathématique,
  - conception de réseau,
  - jeux et puzzles,
  - biologie,
  - physique,
  - chimie,
  - etc.

Exemples de problèmes NP-complets

• **Problème du sac à dos (Knapsack problem) :** est un problème d'optimisation combinatoire. Il consiste à remplir un sac à dos, ne pouvant supporter plus d'un certain poids, avec tout ou partie d'un ensemble donné d'objets ayant chacun un poids et une valeur. Le but est de maximiser la valeur totale sans dépasser le poids maximum autorisé.



• **Problème du voyageur de commerce (Traveling Salesman Problem) :** est un problème mathématique qui consiste à trouver le plus court chemin reliant un ensemble de villes séparées par des distances connues. Pour le moment on ne connaît pas d'algorithme permettant de trouver une solution exacte en temps polynomial.



Bibliographie

- R. ERRA, "Cours 1 d'informatique", 1A-S2 2013-2014 ESIEA.
- Cormen, Leiserson, Rivest, Stein "Algorithmique", 3ème éd. DUNOD.
- S. MESFAR, "Algorithmique et complexité", 2011-2012.

- http://ressources.unisciel.fr/algoprog/s34plexite/emodules/cx00macours1/co/cx00acours2.html
- https://www.u-picardie.fr/~furst/docs/4-Complexite.pdf