Intelligence Artificielle 1

Cours 11 - Glouton et Grand O

Table des matières

Algorithme glouton - Définition

Un algorithme glouton “force brute” évalue naïvement toutes les solutions locales (sous-problèmes) sans retour en arrière pour trouver la solution globale (le problème lui-même). En force brute, il faut comparer tous les résultats des sous-problèmes et prendre le meilleur (global optimal)

Un algorithme glouton peut avoir une règle de décision au niveau des solutions locales (sous-problèmes) pour trouver une solution globale (problème) de manière plus optimale.

Il prend en général beaucoup de mémoire et de complexité (notation Grand O), mais celui-ci est relativement simple à programmer et il est optimal (si on a le temps d'attendre).

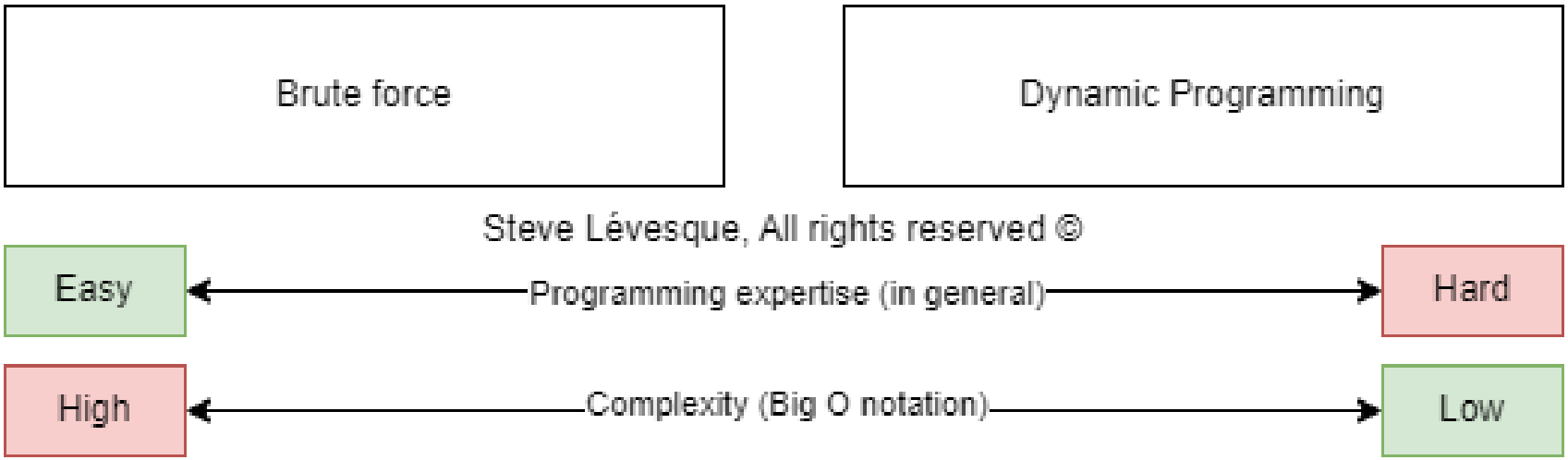
Un exemple de problème est celui de **l'optimisation du sac à dos**.

Comparaison asymptotique / Complexité de temps (grand O)

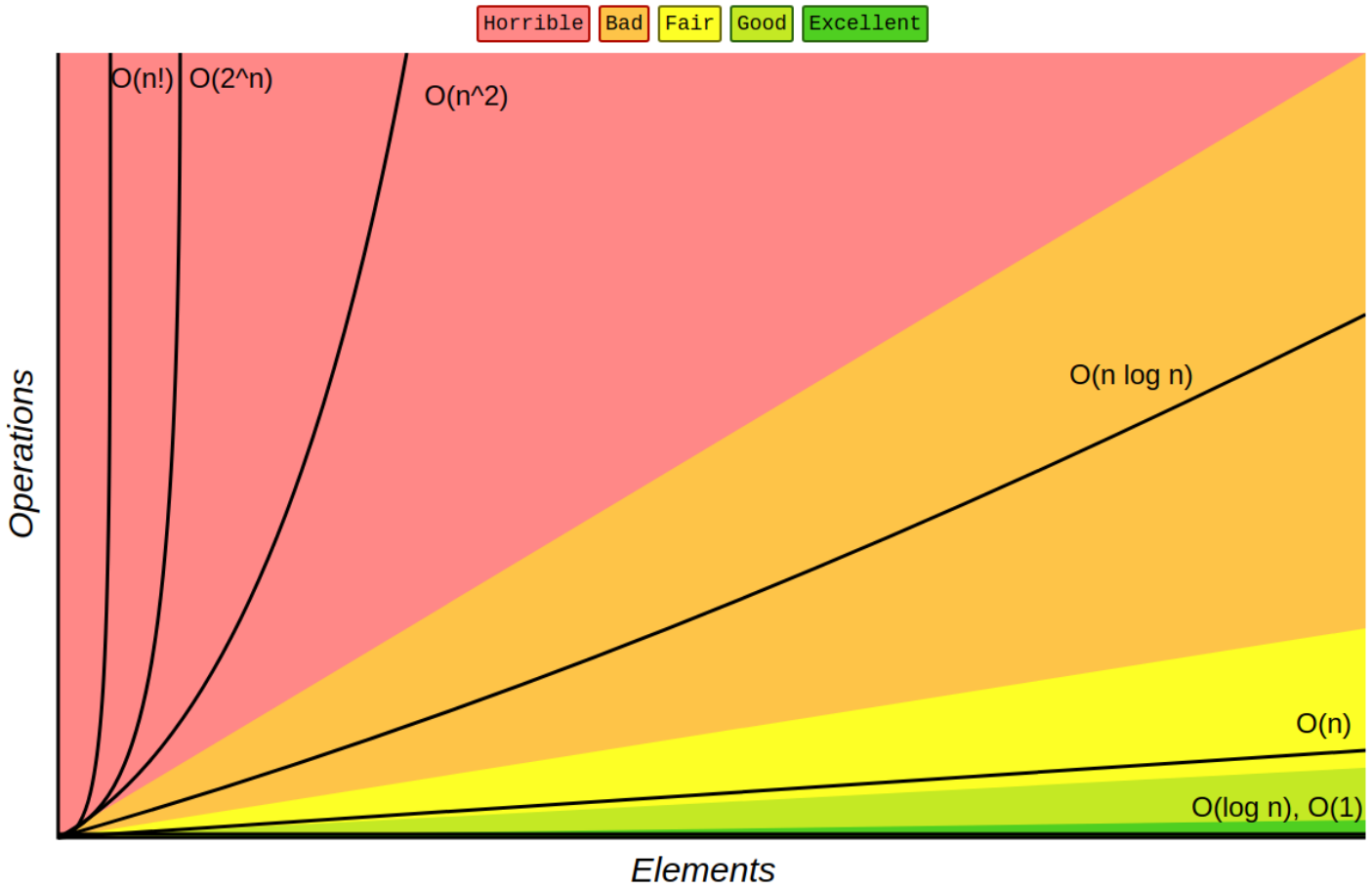
La complexité de temps ("time complexity") est en lien avec le temps qu'un programme (algorithme) prends pour s'exécuter.

L'unité (n) est le nombre d'entrée traité par l'algorithme.

Comparaison asymptotique / Complexité de temps (grand O)



Comparaison asymptotique / Complexité de temps (grand O)



Comparaison asymptotique / Complexité de temps (grand O)

Data Structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartresian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Exemple - Cambrioleur (Problème du Sac à dos)

Un cambrioleur peut voler des objets en fonction de la capacité de son sac (la masse en Kg).

Les objets ont une différente valeur et son but est de pouvoir obtenir le plus de profits ($) possible.

objet	A	B	C	D
masse (Kg)	13	12	8	10
valeur ($)	700	400	300	300

- **Variable de décision :** binaire, dans le sac (1) ou non (0)
- **Contrainte :** capacité du sac est de 30 (Kg)
- **Objectif :** maximiser le profit ($)

Exemple - Cambrioleur (Problème du Sac à dos)

Une solution optimale se calcule soit par un algorithme de type gloutonne “force brute” (en énumérant toutes les solutions possibles) ou par programmation linéaire (pulp) ou programmation dynamique (algorithme récursif).

Un choix judicieux permet soit de programmer plus facilement, soit de sauver du temps et de la mémoire.

À évaluer selon la situation.

Exemple - Cambrioleur (Problème du Sac à dos) - Complexité

Complexité exponentielle en $O(e^n)$:

- e est une constante
- n est la "taille" du problème (ici le nombre d'objets à mettre dans le sac à dos)

Le temps de calcul pour résoudre ce type de problème en énumérant toutes les solutions (2^n) croît exponentiellement avec sa taille.

Exemple - Cambrioleur (Problème du Sac à dos) - Algorithme glouton "force brute"

Rappel (pour un sac avec une capacité de 30 Kg) :

objet	**A**	B	C	D
masse (Kg)	13	12	8	10
valeur ($)	700	400	300	300

On commence avec **A** (solution locale) et on calcule la meilleure combinaison possible avec B, C et D.

Ensuite, a-t-on besoin d'évaluer les autres possibilités : B, C et D (solutions locales) pour obtenir notre solution globale ?

Exemple - Cambrioleur (Problème du Sac à dos) - Algorithme glouton "force brute"

$$A \implies 13 + 12 \textit{ ou } 13 + 8 \textit{ ou } 13 + 10$$
$$\implies 1100 \textit{ ou } 1000 \textit{ ou } 1000$$
$$\implies 1100$$

$$B \implies 12 + 13 \textit{ ou } 12 + 8 + 10 \textit{ ou } 12 + 10 + 8$$
$$\implies 1100 \textit{ ou } 1000 \textit{ ou } 1000$$
$$\implies 1100$$

Exemple - Cambrioleur (Problème du Sac à dos) - Algorithme glouton "force brute"

Même principe ... calcul fait directement

$$C \implies 1100$$
$$D \implies 1100$$

$$Sol \implies A(B) = 1100 \; ou \; B(A) = 1100 \; ou$$
$$C(A) = 1100 \; ou \; D(A) = 1100$$
$$\implies 1100$$

Exemple - Cambrioleur (Problème du Sac à dos) - Algorithme glouton "force brute"

Peut-on faire mieux au niveau de la logique/programmation et décision de la solution globale vis-à-vis les résultats des solutions locales ? Oui, tout à fait.

Cependant, tant qu'il n'y a pas de retour en arrière, cela est considéré comme glouton puisqu'on doit évaluer les solutions locales optimales.

Par exemple, lorsqu'on se rend compte que B, C, D donne 1000 et que ce n'est pas optimal localement, on ne peut pas retourner en arrière après avoir pris B et C. On est obligé de prendre D et de continuer à "perdre notre temps".

L'algorithme glouton est fatal si l'arbre de décision est profond et large...

Bibliographie

- https://coin-or.github.io/pulp/main/index.html