Introduction à la complexité

Complexité : mesure des ressources nécessaires en temps ou en mémoire d'un calcul.

Calcul du temps d'exécution :
- matériel utilisé et
- donnée traitée.

Trop concret et sans information abstraite sur le comportement.

Détermination de l'information pertinente par rapport à la $\boxed{\text{taille}}$ de la donnée traitée.

3 cas d'analyse possible :
- complexité dans le meilleur des cas,
- complexité en moyenne,
- complexité dans le pire des cas.

Pour un algorithme donné et une représentation donnée, le temps sur les machines est identique à un facteur constant près.

Algorithme classique de recherche du maximum dans une liste

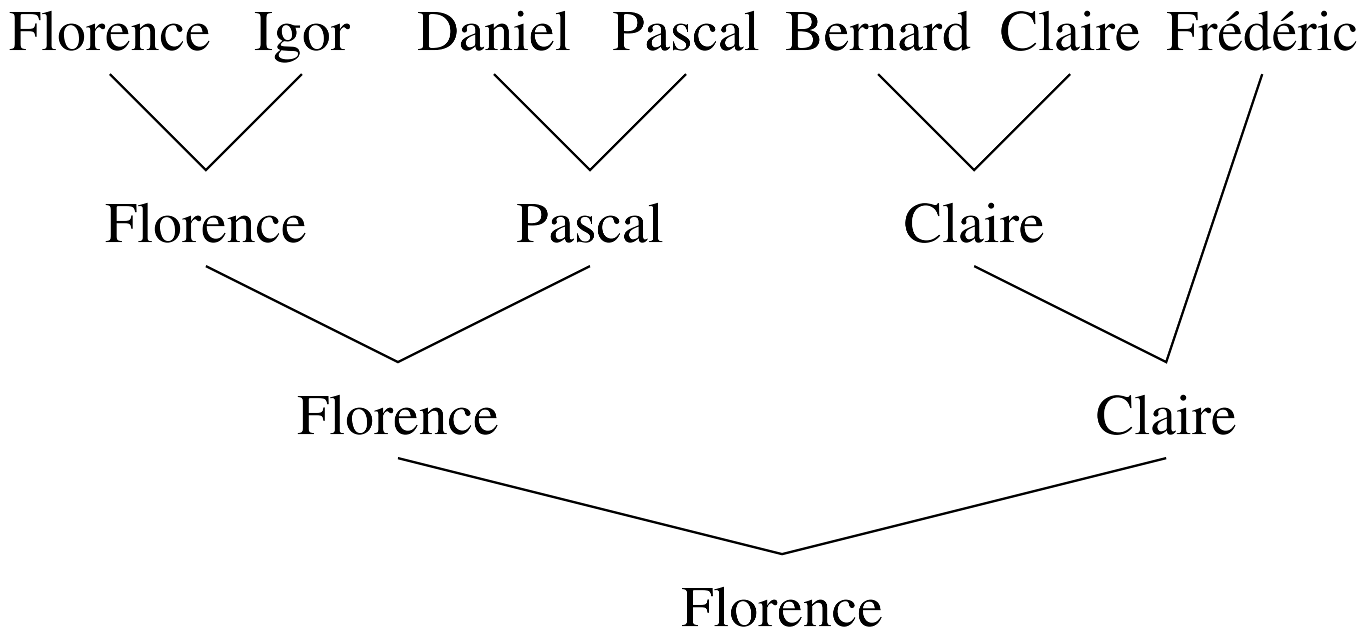
Algorithme de recherche séquentiel du maximum dans une liste d'éléments tous distincts :

```
spécification MAX LISTE(E, DOMAINE(max))
étend   LISTE RECURSIVE(E),
        ORDRE(E, DOMAINE(max))
opérations
max_liste : Liste(E) → E
préconditions {l : Liste(E)}
 pré max_liste(l) : ¬vide_?(l)
axiomes{l : Liste(E)}
 lr1(0) : max_liste(l) =
        si vide_?(fin(l)) alors premier(l)
        sinon max(premier(l), max_liste(fin(l)))
```

Le nombre de comparaisons entre éléments par rapport au nombre d'éléments dans la liste est déterminant pour la complexité.

Le nombre de comparaisons pour une liste de longueur n est exactement de $n - 1$.

Algorithme du Tournoi



Algorithme du tournoi : affrontement par paires de joueurs puis réalisation d'un tournoi avec les joueurs restants.

Recherche du maximum par l'algorithme du tournoi

spécification TOURNOI $\text{LISTE}_0(E, \text{DOMAINE}(max))$

étend LISTE RECURSIVE(E),
ORDRE$(E, \text{DOMAINE}(max))$

opérations
$vainqueur : Liste(E) \to E$
$tournoi : Liste(E) \to Liste(E)$
$élimination : Liste(E) \to Liste(E)$

L'opération *élimination* réalise un tour du tournoi.

suite de TOURNOI LISTE_0

préconditions $\{l : Liste(E)\}$
pré $élimination(l) : \neg vide\_?(l)$

axiomes $\{l : Liste(E)\}$
$tl_0(0) : élimination(l) =$
$si\ vide\_?(fin(l))\ alors\ premier(l)$
$sinon\ cons(\ max(premier(l), second(l)),$
$élimination(fin(fin(l))))$

L'opération ***tournoi*** enchaîne les tours d'élimination jusqu'à ce qu'il n'y ait plus qu'un seul joueur dans la liste ; l'opération ***vainqueur*** en extrait alors le vainqueur.

suite de TOURNOI LISTE_0

préconditions $\{l : Liste(E)\}$
pré $tournoi(l) : \neg vide\_?(l)$
pré $vainqueur(l) : \neg vide\_?(l)$

axiomes $\{l : Liste(E)\}$
$tl_0(1) : tournoi(l) =$
$\quad si\ vide\_?(fin(l))\ alors\ premier(l)$
$\quad sinon\ tournoi(élimination(l))$
$tl_0(2) : vainqueur(l) = premier(tournoi(l))$

Un arbre de tournoi est un arbre binaire localement complet et le nombre de nœuds d'un arbre binaire localement complet ayant n feuilles est $n-1$.

La complexité en nombre de comparaisons entre éléments de l'algorithme du tournoi pour la recherche du maximum dans une liste est exactement de $n-1$.

Optimalité des deux algorithmes de recherche

Propriété : La complexité dans le pire des cas d'un algorithme de recherche du maximum dans une liste de longueur n est au mieux en $n - 1$.

Lemme : Soit un algorithme quelconque de recherche du plus grand élément dans une liste alors nécessairement tout élément autre que le maximum est comparé au moins une fois avec un élément qui lui est plus grand.

Les algorithmes classique et du tournoi sont optimaux pour la recherche du maximum dans une liste.

Le deuxième plus grand élément

Première approche : Chercher le deuxième plus grand élément d'une liste, c'est cherché le plus grand élément de la liste privée de son maximum.

Complexité dans le pire des cas en nombre de comparaisons de la première approche est exactement $(n-1)+((n-1)-1)$.

Deuxième approche : Le deuxième meilleur joueur d'un tournoi, c'est un de ceux qui n'ont perdu que contre le vainqueur.

Si n est le nombre de joueurs alors le nombre des joueurs qui ont rencontré le vainqueur est $\lceil \log_2 n \rceil$.

Complexité dans le pire des cas en nombre de comparaisons de la deuxième approche est exactement $(n-1)+(\log_2 n-1)$.

Un peu de recul

Pour des données de grande taille les constantes sont négligeables.

Une fonction de complexité g est en $\Theta(f)$, si il existe des constantes c, d, n_0 et n_1 telles que :

$$\forall n > max(n_0, n_1),$$
$$d * f(n) + n_0 \leq g(n) \leq c * f(n) + n_1$$

Comparaison de quelques complexités :

$$\Theta(\log_2 n) < \Theta(n) < \Theta(n \log_2 n) < \Theta(n^2) < \Theta(2^n)$$

n	$\log_2 n$	n	$n \log_2 n$	n^2	2^n
10	≈ 3.3	10	≈ 33	10^2	1024
10^2	≈ 6.6	10^2	$\approx 6.6 * 10^2$	10^4	$\approx 10^{30}$
10^3	≈ 10	10^3	$\approx 10^4$	10^6	$\approx 10^{301}$
10^6	≈ 20	10^6	$\approx 2 * 10^7$	10^{12}	∞

Influence du codage sur la complexité

Un codage est $\boxed{\text{raisonnable}}$ si il n'affecte la taille des données que de façon polynômiale.

Règles à respecter :
- pas de $\boxed{\text{bourrage}}$ du codage,
- décodage polynômial,
- représentation des chiffres autre qu'unaire.

En pratique : tous les codages "naturels" d'un même langage (instances d'un problème) sont raisonnables.

Exemple : Représentation d'un graphe (V, E) ($|V|= n$, $|E|= m$ et codage binaire des sommets sur $log_2(|V|)$ bits) par :
- Liste des sommets et liste des arcs : $O((n+m)log_2(n))$;
- Matrice d'incidence : $O(n^2)$.

Complexité et machines de Turing déterministes

Soit M une machine de Turing déterministe (sur un alphabet $\boldsymbol{\Sigma}$) qui s'arrête toujours. La $\boxed{\text{complexité en temps de } M}$ est la fonction :

$T_M(n) = max\{m/\exists x \in \Sigma^*, |x| = n$ et l'exécution de M sur x comporte m étapes$\}$.

Une machine de Turing déterministe M est $\boxed{\text{polynômiale}}$ si il existe un polynôme $p(n)$ tel que la fonction de complexité de T_M soit en $O(p(n))$.

Lemme : Si il existe un algorithme polynômial pour résoudre un problème alors il existe une machine de Turing déterministe polynômiale qui décide le langage des instances positives du problème.

La classe P

La $\boxed{\text{classe } \mathcal{P}}$ ($\boxed{\mathcal{P}\text{olynomial}}$) est la classe des langages décidés par une machine de Turing déterministe polynômiale.

classe $\mathcal{P} \cong$ algorithmes efficaces
(insensible au choix de la machine et
d'un codage raisonnable).

La classe $\mathcal{P}$ est close par composition.

Comment établir une hiérarchie de complexité des problèmes ?

Un outil : la $\boxed{\text{réduction}}$ (ou $\boxed{\text{transformation}}$) polynômiale.

La réduction polynômiale

Soient L_1 et L_2 deux langages respectivement sur Σ_1 et Σ_2. Une réduction polynômiale de L_1 vers L_2 (notée $L_1 \leq_P L_2$) est une fonction $f : \Sigma_1^* \to \Sigma_2^*$ telle que
- elle est calculable en temps polynômial,
- $f(x) \in L_2$ si et seulement si $x \in L_1$.

L_1 se réduit polynômialement à L_2 signifie que L_1 n'est pas plus difficile, à un facteur polynômial près, que L_2.

Exemple : Soient $L_1 = \{x/ax + b = 0\}$ et $L_2 = \{x/ax^2 + bx + c = 0\}$.
$L_1 \overset{?}{\leq_P} L_2$.
Soit $f : ax + b = 0 \to 0x^2 + ax + b = 0$,
- f est calculable en temps polynômial,
- $f(x) \in L_2$ si et seulement si $x \in L_1$.

La réduction polynômiale (suite)

Lemme : Si $L_1 \leq_P L_2$ alors
- si $L_2 \in \mathcal{P}$ alors $L_1 \in \mathcal{P}$,
- si $L_1 \notin \mathcal{P}$ alors $L_2 \notin \mathcal{P}$.

Lemme : Si $L_1 \leq_P L_2$ et $L_2 \leq_P L_3$ alors $L_1 \leq_P L_3$.

Deux langages L_1 et L_2 sont $\boxed{\text{polynômialement équivalents}}$ (noté $L_1 \equiv_P L_2$) si $L_1 \leq_P L_2$ et $L_2 \leq_P L_1$.

La classe $\mathcal{P}$ est une classe d'équivalence pour $\equiv_P$.

Une classe d'équivalence polynômiale C_1 est inférieure à une classe d'équivalence polynômiale C_2 (noté $C_1[\leq_P]C_2$) si

$$\forall L_1 \in C_1, \forall L_2 \in C_2, L_1 \leq_P L_2.$$

Voyageur de commerce et circuit hamiltonien

$\boxed{\text{Problème du voyageur de commerce } (\boldsymbol{TS}) :}$ Soit un ensemble de ville $\boldsymbol{V}$ et pour chaque paire de villes, une distance. Soit une distance $\boldsymbol{m}$. Existe-t-il un parcours fermé passant par toutes les villes et qui revient à son point de départ ?

$\boxed{\text{Problème du circuit hamiltonien } (\boldsymbol{HC}) :}$ Pour un graphe donné, existe-t-il un parcours fermé du graphe contenant chaque sommet une et une seule fois ?

Lemme : $\boldsymbol{HC} \equiv_{\boldsymbol{P}} \boldsymbol{TS}$.

Algorithme de résolution : génération d'une permutation puis vérification.

Problème : exploration de l'espace des permutations.

Machines de Turing non-déterministes

Soit $M = (Q, \Gamma, \Sigma, \delta, s, \#, F)$
- Q est un ensemble fini d'états,
- Γ est un alphabet de ruban,
- Σ est un alphabet d'entrée,
- $\Delta \subset ((Q \times \Gamma) \times (Q \times \Gamma \times \{g, d\}))$ la relation de transition,
- $s \in Q$ l'état initial,
- $\# \in \Gamma - \Sigma$ le caractère "blanc",
- $F \subseteq Q$ l'ensemble des états accepteurs.

M est une $\boxed{\text{machine de Turing non-déterministe}}$.

Un mot est accepté par une machine de Turing non-déterministe si il existe une exécution qui contienne une configuration dont l'état est accepteur.

Machines de Turing non-déterministes et complexité

Le $\boxed{\text{temps de calcul}}$ d'une machine de Turing sur un mot est donné par
- la longueur de la plus petite exécution en nombre d'étapes acceptant le mot,
- 1 si le mot n'est pas accepté.

Soit M une machine de Turing (sur un alphabet Σ). La $\boxed{\text{complexité en temps de } M}$ est la fonction :

$T_M(n) = max\{m/\exists x \in \Sigma^*, |x| = n$ et le temps de calcul de M sur x est $m\}$.

Une machine de Turing non-déterministe M est $\boxed{\text{polynômiale}}$ si il existe un polynôme $p(n)$ tel que la fonction de complexité de T_M soit en $O(p(n))$.

La classe $\mathcal{NP}$

$1^{ère}$ caractérisation :

La $\boxed{\text{classe } \mathcal{NP}}$ ($\boxed{\mathcal{N}\text{on déterministe } \mathcal{P}\text{olynômial}}$) est la classe des langages acceptés par une machine de Turing non-déterministe polynômiale.

$2^{ème}$ caractérisation :

La $\boxed{\text{classe } \mathcal{NP}}$ est la classe des langages telle que la $\boxed{\text{vérification}}$ soit décidable par une machine de Turing polynômiale.

Th.: Soit $L \in \mathcal{NP}$, alors il existe une machine de Turing déterministe M et un polynôme $p(n)$ tel que M décide L en $O(2^{p(n)})$.

Lemme : $\mathcal{P} \subseteq \mathcal{NP}$

Lemme : $\forall L_1 \in \mathcal{P}, \forall L_2 \in \mathcal{NP}, L_1 \leq_P L_2$

Question ouverte : $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$

La classe $\mathcal{NP}-complet$

La classe des problèmes $\mathcal{NP}-complet$ est la classe des problèmes les plus durs de $\mathcal{NP}$.

Un langage L est $\mathcal{NP}-complet$ si
- $L \in \mathcal{NP}$ et
- $\forall L' \in \mathcal{NP}, L' \leq_P L$.

Un langage L est $\mathcal{NP}-dur$ si
- $L \notin \mathcal{NP}$ et
- $\forall L' \in \mathcal{NP}, L' \leq_P L$.

Lemme : $\mathcal{NP}-complet$ est une classe d'équivalence pour $\equiv_P$ et $\mathcal{P} \leq_P \mathcal{NP}-complet$.

cor.: si il existe un langage $L \in \mathcal{NP}-complet$ décidé par une machine de Turing déterministe polynômiale alors $\mathcal{P} = \mathcal{NP}$.

cor.: Pour démontrer qu'un langage L est $\mathcal{NP}-complet$, il suffit de montrer que
- $L \in \mathcal{NP}$ et
- $\exists L' \in \mathcal{NP}-complet, L' \leq_P L$.

Un problème $\mathcal{NP}-complet$: SAT

Le problème SAT : Existe-t-il une interprétation qui satisfasse une formule propositionnelle en forme normale conjonctive?

Exemple : La formule $(p \vee \neg q) \wedge r \wedge (s \vee \neg p)$ est en forme normale conjonctive.

Th. de Cook : Le problème SAT est $\mathcal{NP}-complet$.

Idée Démo : - Montrer que $SAT \in \mathcal{NP}$.
- Construire une fonction f de l'ensemble des couples (Machine de Turing polynômiale, mot) vers l'ensemble des formules propositionnelles sous forme normale conjonctive telle pour toute machine de Turing polynômiale M et tout mot w, M accepte w si et seulement si $f(< M, w >)$ est satisfaisable.

D'autres problèmes $\mathcal{NP}-\boldsymbol{complet}$

Lemme : Le problème $3-SAT$ est $\mathcal{NP}-\boldsymbol{complet}$.

Lemme : Pour tout $k \leq 3$, $k-SAT$ est $\mathcal{NP}-\boldsymbol{complet}$.

Lemme : Le problème de couverture de sommets dans un graphe est $\mathcal{NP}-\boldsymbol{complet}$.

Lemme : HC est $\mathcal{NP}-\boldsymbol{complet}$.

Lemme : TS est $\mathcal{NP}-\boldsymbol{complet}$.

Lemme : Le problème du nombre chromatique d'un graphe est $\mathcal{NP}-\boldsymbol{complet}$.

Lemme : Le problème de la programmation entière est $\mathcal{NP}-\boldsymbol{complet}$.

Au delà de $\mathcal{NP}$

La classe $co-\mathcal{NP}$ est la classe des langages dont le complément est dans $\mathcal{NP}$.

La classe $EXPTIME$ est la classe des langages décidés par une machine déterministe dont la complexité est en $O(2^{p(n)})$, $p(n)$ un polynôme.

La classe $PSPACE$ est la classe des langages décidés par une machine de Turing déterministe dont la complexité en espace est polynômiale.

La classe $NPSPACE$ est la classe des langages décidés par une machine de Turing non-déterministe dont la complexité en espace est polynômiale.

Lemme : $\mathcal{P} \subseteq \begin{array}{c} \mathcal{NP} \\ co-\mathcal{NP} \end{array} \subseteq PSPACE \subseteq EXPTIME$.

Lemme : $PSPACE = NPSPACE$.

Lemme : $\mathcal{P} \subset EXPTIME$.