Qu'est-ce qu'un problème ? C'est une question paramétrée dans un ensemble infini.

Instance : Une instance d'un problème est obtenue en spécifiant la valeur du paramètre.

Résolution : Un algorithme résoud un problème s'il trouve la solution à toutes ses instances.

Heuristique : Ressemble à un algorithme mais ne trouve pas toujours la même solution.

Encodage d'une instance : On représente les instances d'un problème par une suite de symboles répondant à un schéma d'encodage.

Taille d'une instance : Nombre de symboles utilisés pour la représenter.

Remarque : En général, on ne s'intéresse pas au schéma d'encodage précisément mais à un critère proportionnel à la longueur d'un schéma d'encodage raisonnable.

Complexité temporelle d'un algorithme :
$f : n \rightarrow$ nb_max d'étapes élémentaires pour résoudre une instance de taille n

O : Une fonction $f(n)$ est dans $O(g(n))$ si il existe deux constantes strictement positives K et C telles que $\forall n > k, f(n) \leq g(n)$
Version équivalente : $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

Ω : $f(n) \in \Omega(g(n))$ si $g(n) \in O(f(n))$.

θ : $f(n) \in \theta(g(n))$ si $g(n) \in O(f(n))$ et $f(n) \in O(g(n))$

Propriétés du O : Soit f et g tq $f(n) \in O(g(n))$. Soient a, b, c des constantes et x une variable.

1. Si $a \geq 0$ et $\forall n \geq a, f(n) < g(n)$ alors $f(n) \in O(g(n))$
2. Si $a \geq 0$ alors $af(n) \in O(f(n))$
3. Si $a \geq 1$ et $b \geq 0$ alors $a^{n+b} \in O(a^n)$
4. Si $f(n) \in O(g(n))$ et $f_1(n) \in O(g_1(n))$ alors $f(n) + f_1(n) \in O(\max(g(n), g_1(n)))$
5. Si $f(n) \in O(g(n))$ et $f_1(n) \in O(g_1(n))$ alors $f(n)f_1(n) \in O(g(n)g_1(n))$
6. Si $a \geq 2, b \geq 1, c > 0$ alors $\log_a(n^b) \in O(n^c)$
7. Si $a > 0$ et $b > 1$ alors $n^a \in O(b^n)$

preuves

1. On pose $k = a$ et $C = 1$.

2. On pose $k = 1$ et $C = a$.

3. $a^{n+b} = a^b a^n$, on applique le cas 2 avec $f(n) = a^n$

4. On prend (k, C) pour $f(n) \in O(g(n))$, (k_1, C_1) pour $f_1(n) \in O(g_1(n))$
   On prend (k_2, C_2) avec $k_2 = max(k, k_2)$ et $C_2 = 2\max(C, C_1)$
   On pose $h(n) = \max(g(n), g_1(n))$
   On a pour $n > k_2$: $f(n) + f_1(n) \leq Cg(n) + C_1 g_1(n) \leq \max(C, C_1)h(n) + \max(C, C_1)h(n) \leq C_2 h(n)$

5. On prend (k, C) pour $f(n) \in O(g(n))$, (k_1, C_1) pour $f_1(n) \in O(g_1(n))$ On prend (k_2, C_2) avec $k_2 = max(k, k_2)$ et $C_2 = CC_1$
   On pose $h(n) = g(n)g_1(n))$
   On a pour $n > k_2$: $f(n)f_1(n) \leq Cg(n)C_1 g_1(n) \leq \mathbb{C}_2 h(n)$

6. $\log_a(n^b) = b\log_a(n)$ donc il reste à montrer que $\log_a(n) \in O(n^c)$
   Il suffit de faire lim $\frac{\ln n}{n^c}$, qui vaut 0 par le lemme de l'hôpital.

7. Il suffit de faire lim $\frac{n^a}{b^n}$ avec le lemme de l'hôpital.

Idée : On cherche à exprimer la complexité d'un algo en fonction de la taille n de l'instance en entrée.

3 étapes :

1. Diviser le problème initial en plusieurs sous problèmes.
2. Régner : Si la taille d'un sous problème est assez petite, on le résoud. Sinon récursion.
3. Combiner : On forme la solution au problème initial en combinant les solutions aux sous problèmes.

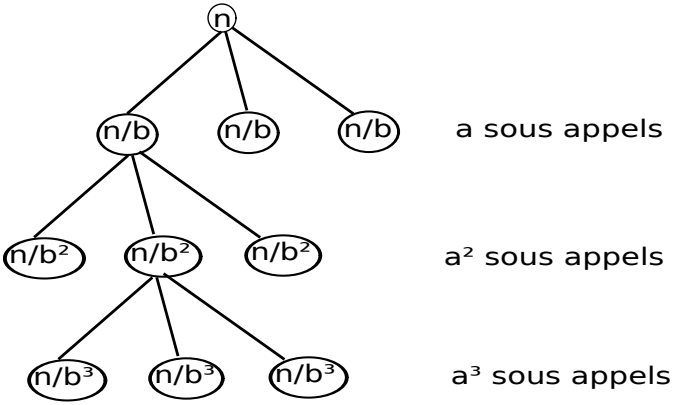
Schéma :

Fonction DPR(Probleme P)
Debut
    Si taille(P) petite alors Resoudre(P)
    Sinon
        Diviser P en P_1, P_2, ..., P_k
        Pour i de 1 à k S_i = DPR(P_i) FinPour
        Combiner(S_1,...,S_n)
Fin

Théorème Maître:

- Si $f(n) = O(n^{\log_b a - \varepsilon})$ $(\varepsilon > 0)$ alors $T(n) \in O(n^{\log_b a})$
- Si $f(n) = \theta(n^{\log_b a})$ alors $T(n) = O(n^{\log_b a} \ln n)$
- Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ $(\varepsilon > 0)$. Alors $T(n) \in O(f(n))$

Représentation des appels récursifs par un arbre :



Niveau	Nb Noeuds	Temps pour ce niveau
0	1	$f(n)$
1	a	$af(\frac{n}{b})$
2	a^2	$a^2 f(\frac{n}{b^2})$
3	a^3	$a^3 f(\frac{n}{b^3})$
$\vdots$	$\vdots$	$\vdots$

1. hauteur de l'arbre : $\frac{n}{b^h} = 1 \equiv h = \log_b n$

2. Nb de noeuds dans l'arbre : $a^0 + a^1 + a^2 + \ldots + a^h = \frac{1+a^{h+1}}{1-a} = C + C2a^h$. Or $a^h = a^{\log_b n} = (b^{\log_b a})^{\log_b n} = (b^{\log_b n})^{\log_b a} = n^{\log_b a}$. Donc le nombre de noeuds dans l'arbre est en $O(n^{\log_b a})$.

3. La complexité de l'algorithme dépend ce qui compte le plus de $f(n)$ ou du nombre de noeuds dans l'arbre. Il y a donc trois cas.

Sous structure optimale : Une solution à une instance contient en elle la solution de plusieurs sous problèmes.

Mémoriser les solutions aux sous problèmes pour pouvoir les réutiliser dans le calcul de la solution.

Cas d'utilisation : La programmation dynamique est souvent utile lorsqu'on a une équation de la forme $T(n) \geq aT(n-b)+f(n)$

Théorème : Soit $T(n)$ une fonction de complexité temporelle qui satisfait l'équation :
$T(n) = aT(n-b) + f(n)$, avec $a \geq 2, b \geq 1, f(n) \in \Omega(1)$.
Alors $\exists\, c > 1 \mid T(n) \in \Omega(c^n)$

Preuve : On pose $c = \sqrt[b]{a}$. Montrons par récurrence que $T(n) \geq c^n - a$.
Initialisation : Si $n \leq b$ alors : $c^n - a = c^n - c^b \geq 0$ (car $c > 1$ et $b \geq n$). Or T est une fonction de complexité donc $T(n) \geq 0$
Hérédité : Si $n > b$, on suppose que $T(n-b) \geq c^{n-b} - a$.
On a alors $T(n) = aT(n-b) + f(n) \geq a(c^{n-b} - a) + f(n) = c^n - a^2 + f(n) \geq c^n - a^2$ (car $f(n) \geq 0$) donc $T(n) \in \Omega(c^n)$

Exemple : Nombres de Fibonacci

$$\text{Fibo}(n) = \begin{cases} 0 \text{ si } n = 0 \\ 2 \text{ si } n = 2 \\ f(n-1) + f(n-2) \text{ sinon} \end{cases}$$

Algo : Découpe de barres

On dispose d'une barre de longueur n, et d'un tableau prix indicé de 1 à k. On cherche la manière de découper la barre de façon à maximiser le prix de vente (somme des prix des sous barres).

Algo : Distance de Levenstein

On se dote de 3 opérations sur les mots :

1. Enlever une lettre
2. Ajouter une lettre
3. Remplacer une lettre par une autre

Etant donnés deux mots u et v on cherche le nombre minimum d'opérations élémentaires pour transformer u en v.

Algo : Impression équilibrée d'un texte

On veut imprimer n mots sur maximum n lignes de taille N. Les espaces donnent des pénalités.

Principe : Un algo qui progresse vers une solution en faisant des choix plus ou moins arbitraires et qui revient en arrière lorsqu'il est bloqué avant d'atteindre une solution. Si tous les choix ont été explorés, on en conclut qu'il n'existe pas de solution.

Avantages :

1. Facilement adaptable à beaucoup de problèmes
2. Facile à implémenter grâce à la récursivité

Inconvénient :

1. Produit souvent une complexité exponentielle

Algo : Sudoku

Algo : Dancing Links

Algo polynomial : Complexité en $O(n^k)$

Types de problèmes :

1. D : Décision (OUI ou NON)
2. O : Optimisation (+ grande ou + petite valeur qui satisfait un critère)
3. E : Existence (on cherche un élément qui satisfait un critère)

La plupart des problèmes peuvent se décliner sous ces 3 formes, et si D est polynomial alors les autres aussi. On se contentera donc d'étudier la version D des problèmes.

Réduction polynomial : Soient P_1 et P_2 deux problèmes.
On appelle réduction polynomiale de P_1 vers P_2 un algo R polynomial qui transforme toute instance I_1 de P_1 en une instance I_2 de P_2 telle que $P_1(I_1) = P_2(I_2)$.
S'il existe une réduction polynomiale de P_1 vers P_2, on note $P_1 \leq_p P_2$.

Théorème : Soient P_1 et P_2 deux problèmes tels que $P_1 \leq_p P_2$.

- Si P_2 est polynomial alors P_1 aussi
- Si P_1 n'est pas polynomial alors P_2 non plus (forme contraposée)

Preuve Il suffit de poser $A_1 = A_2 \circ R$. Si A_2 est en $O(n^k)$ et R en $O(n^l)$, alors A_1 est en $O(n^{lk})$.

Equivalence polynomiale : On dit que P_1 et P_2 sont polynomialement équivalents si $P_1 \leq_p P_2$ et $P_2 \leq_p P_1$. On le note $P_1 =_p P_2$

Relations : $=_p$ est une relation d'équivalence et $\leq_p$ est une relation d'ordre.

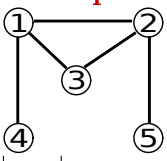
Théorème : Tous les problèmes polynomiaux sont polynomialement équivalents (même classe d'équivalence).

Preuve : Soit I_{OUI} et I_{NON} deux instances de P_2 telles que $P_2(I_{OUI}) = OUI$ et $P_2(I_{NON}) = NON$.
Il suffit de définir ainsi l'algo R :
Si $P_1(I) = OUI$ Alors retourner I_{OUI}
Sinon retourner I_{NON}

Clique : Une clique de taille n est un sous graphe complet de taille n (K_n). Une anticlique de taille n est un sous graphe vide de taille n ($\bar{K_n}$).

Théorème de Ramsey : Pour tout entier $k \geq 1, \exists R_k \in \mathbb{N}$ tel que tout graphe possédant au moins R_k sommets contient au moins une clique (K_k) ou une anticlique ($\bar{K_k}$) de taille k.

Exemple :

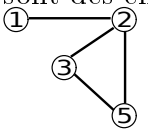


k	R_k
1	1
2	2
3	6
4	18
5	$\in [43; 49]$
6	$\in [102; 165]$
17	$\in [8917; 601080389]$

Essai d'algorithme naïf : On cherche tous les graphes de chaque taille et on cherche un contre exemple.
Pour une taille N donné il existe $2^{\frac{N(N-1)}{2}}$ graphes possibles.
Pour chaque graphe possible on va tester tous les sous graphes de taille k donc $\frac{N!}{k!(N-k)!}$.

Puis pour chacun il faut vérifier que c'est une clique ou une anti-clique donc $\frac{k(k-1)}{2}$.
Et il faut itérer sur N pour chercher un contre exemple.
Pour $N = 8$ et $k = 4$ le programme ne terminera pas de notre vivant.

Représentation d'une instance : Graphe dont les sommets sont des entiers.



Symboles = $\{(,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -\}$.
Schéma = $(1-2-3-5)(1-2-3-5-2-3-2-5)$ représente le graphe ci-dessus avec une taille 26.

Cycle Eulérien : Passe une et une seule fois par chacune des arêtes.

Théorème : Un graphe G est eulérien si et seulement si il est connexe et tous ses sommets sont de degré pair.

Preuve :

- $\Rightarrow$ Si G a un sommet de degré impaire, il ne peut pas être eulérien car chaque passage dans un sommet diminue son degré de 2.
- $\Leftarrow$ Si G a tous ses sommets de degré pair, dès qu'on arrive sur un sommet, on peut sortir, donc on revient forcément

au début. Pour chaque sommet dont tous les voisins n'ont pas été visités, on choisit une arête sortante non visitée, par le même raisonnement on revient à ce sommet au bout d'un moment. On intègre le cycle obtenu au cycle de départ et on recommence, jusqu'à ce que toutes les arêtes aient été visitées.

Cycle Hamiltonnien : Passe une et une seule fois par chacun des sommets.