Méthodes de résolution

1) Force brute

Cette approche résout directement le problème à partir de sa définition ou d'une recherche exhaustive.

En appliquant la définition, elle mène par exemple au tri par sélections de complexité quadratique.

Par recherche exhaustive pour le voyageur de commerce dans un graphe non orienté, on générera tous les tableaux possibles par permutations, cette approche a une complexité qui croît très rapidement avec n, car il existe $\frac{1}{2}(n-1)!$ parcours distincts possibles.

Cette technique est très simple et d'application très large, il s'agit donc d'un bon point de départ et si nécessaire, on cherche ensuite d'autres algorithmes plus performants et plus élégants.

2) Diviser pour régner

On divise le problème en sous-problèmes plus faciles à résoudre (jusqu'au cas de base) que l'on fusionne pour obtenir la solution du problème original. On trouve donc trois étapes : diviser (diviser le problème), régner (résoudre récursivement les sous-problèmes jusqu'au cas de base, trivial à résoudre) et fusionner (à partir des solutions des sous-problèmes, trouver la solution du problème original).

Cette approche mène aux tri par fusion et au tri Quicksort (tri rapide), deux algorithmes de complexité $O(n \log n)$. Pour Quicksort, il est de complexité $O(n \log n)$ dans tous les cas si on considère la version améliorée avec un choix judicieux du pivot.

3) Programmation dynamique

On cherche à obtenir la solution optimale en combinant des solutions optimales de sous-problèmes. Il s'agit d'une généralisation de l'approche précédente : les sous-problèmes peuvent ici se chevaucher et dépendre les uns des autres. Lorsqu'un même sous-problème apparaît plusieurs fois, on s'arrange pour ne le résoudre qu'une seule en sauvegardant les solutions dans une table selon le principe de "mémoïsation" : on échange en fait du temps de calcul contre de la mémoire. Souvent, on peut passer de la complexité exponentielle à une complexité polynomiale.

La programmation dynamique nécessite de disposer d'une **propriété de sous-structure optimale** :
Par exemple, si $x_0 \to x_1 \to ... \to x_p$ est le plus court-chemin reliant x_0 à x_p, alors pour tous $0 \leq i \leq j \leq p$, $x_i \to x_{i+1} \to ... \to x_j$ est le plus court-chemin reliant x_i à x_j.

Remarque : En général, on ne peut pas utiliser la récursivité pour implémenter ces algorithmes de manière efficace (contrairement au cas relevant de l'approche diviser pour régner), car la taille des sous-structures peut très bien être $n-1$ (alors qu'elle est en $n/2$ lorsqu'on procède par dichotomie).

4) Approche gloutonne

On construit incrémentalement la solution en optimisant localement pour obtenir un optimum global. On doit donc effectuer à chaque fois un choix, alors que la programmation dynamique explore tous les choix possibles.

En fonction des problèmes, des entrées et des paramètres, on n'obtient pas nécessairement une solution optimale. Par ailleurs si on trouve une solution optimale, l'approche gloutonne ne pourra donner toutes les solutions optimales. Et il est souvent assez difficile de prouver la correction de l'algorithme.

A défaut d'optimalité, on peut néanmoins espérer obtenir une bonne solution, et avec un coût réduit, au minimum.

Remarque : Pour que l'approche gloutonne fonctionne, il faut a fortiori avoir la propriété de sous-structure optimale (puisqu'on optimise localement), mais ce n'est pas une condition nécessaire.

5) Approche avec heuristique

Un développement heuristique est une méthode de résolution utilisant des informations supplémentaires (venant soit du contexte particulier soit d'informations acquises lors d'essais précédents) permettant d'orienter efficacement la recherche, de sorte qu'on peut considérer plausible mais non certain qu'elle conduira à la détermination d'une solution satisfaisante (voire optimale) du problème.

Exemple : L'algorithme A^* de recherche d'un chemin optimal pour joindre deux points fixés dans un plan eulclidien avec obstacles : on choisit comme heuristique la distance à vol d'oiseau de sorte à orienter la recherche dans les chemins qui suivent le plus possible cette trajectoire directe.

Preuves de programmes : terminaison et correction

1) Terminaison

Il s'agit de justifier que l'algorithme termine en un nombre fini d'étapes.

Le problème se pose d'une part avec la présence de boucle `while` et d'autre part avec les fonctions récursives (il faut montrer que le test d'arrêt est atteint).

En général, on explicite une grandeur entière (dans $\mathbb{N}$) qui décroît strictement à chaque itération (ou à chaque appel dans le cas des fonctions récursives).

La terminaison se démontre donc à l'aide d'**un "variant de boucle" et du principe de Fermat :**

Principe de Fermat : Toute suite décroissante d'entiers naturels est stationnaire. Autrement dit, il n'existe pas de suite d'entiers naturels strictement décroissante.

Exemple : Le calcul de $f(n,m)$ défini par
$\begin{cases} f(n,0) = f(0,n) = n \\ f(n,m) = f(n-m,m) \text{ si } n \geq m \text{ et } f(n,m) = f(n,m-n) \text{ si } n < m \end{cases}$
termine, car la valeur de $n+m$ décroît strictement à chaque itération (ou à chaque appel selon qu'on programme en itératif avec une boucle while ou en récursif).

2) Correction

Il s'agit de justifier que l'algorithme renvoie bien ce qui est attendu : on prouve souvent la correction de l'algorithme en précisant ce que stockent les variables aux cours des itérations ou des appels récursifs.

On utilise des **invariants de boucles**, qui sont des propriétés vérifiées à toute étape de l'exécution.

Un invariant de boucle peut certes désigner une grandeur qui se conserve au cours des itérations. En fait, **un invariant de boucle consiste à expliciter une propriété $\mathcal{P}(i)$ vérifiée par les variables au cours de la**

boucle d'indice i. On justifie par récurrence que la propriété $\mathcal{P}(i)$ est vraie pour tout i, et on déduit la valeur des variables lors de la sortie de boucle.

- *Exemple* :

Dans le cas des fonctions récursives, on peut être amené à utiliser aussi des raisonnements par récurrence (par exemple, sur le nombre d'appels ou sur la taille des objets entrés comme arguments).

- *Exemple* : Considérons à nouveau $\begin{cases} f(n,0) = f(0,n) = n \\ f(n,m) = f(n-m,m) \text{ si } n > m \text{ et } f(n,m) = f(n,m-n) \text{ si } n < m \end{cases}$

On sait mathématiquement que $\text{pgcd}(a,b) = \text{pgcd}(a, a-bq)$.

On en déduit que $f(n,m) = \text{pgcd}(n,m)$ par récurrence forte sur $n+m$.

En termes informatiques, la propriété $f(n,m) = \text{pgcd}(n,m)$ est un invariant de boucle.

Remarque terminologique : La correction partielle consiste à prouver que l'agorithme renvoie bien ce qu'on attend lorsqu'il termine. La correction totale consiste donc à prouver la correction partielle et la terminaison.

3) Exemple : Distance à un sommet fixé dans un graphe orienté (non valué)

On suppose le graphe sur $[\![n]\!] = \{0, 1, ..., n-1\}$ défini par la liste des listes d'adjacence.

On fixe un sommet s (sommet source).

On utilise **un parcours en largeur** et on met à jour le tableau d `correspond` à la distance de s à x, c'est-à-dire la longueur (= nombre d'arêtes) minimale d'un chemin reliant s à x.

On procède ainsi :

Etape 1 : Initialisation :

- On crée un tableau distance d de longueur n, chaque cas initialisée à -1 (correspond à $+\infty$)
- On crée une file F (initialement vide) de capacité n
- On ajoute s à la file F et on initialise $d[s] = 0$.

Etape 2 : Tant que la file F n'est pas vide :

- Retirer l'élément x en début de file
- Sinon, pour chaque successeur y de x pour lequel $d[y]$ vaut -1 :

  ajouter y à la fin de la file F, et donner à $d[y]$ la valeur $d[x]+1$

```
01.  def plus_court_chemin(G,s,z) :
         # étape 1 : initialisation
02.      n = len(G) ; d = [-1]*n ; F = creerFile(n)
03.      ajouter(F,s) ; d[s] = 0
         # étape 2
04.      while F :
05.          x = defiler(F) # retire un élément en début de la file F
06.          for y in G[x] :
07.              if d[y] == -1 :
```

```
08.                         ajouter(F,y) ; d[y] = d[x]+1
            # étape 3 : final
09.         return d
```

On montre que l'algorithme admet **les invariants suivants** :

(a) Les éléments x_1, x_2, ... , x_p de la file d'attente vérifient : $d[x_1] \leq ... \leq d[x_p] \leq d[x_1] + 1$

(b) Tous les sommets de distance $\leq d[x_1]$ ont déjà été insérés dans la file (et éventuellement retirés)

(c) Si $d[x] \neq -1$, alors $d[x]$ est la longueur minimale d'un chemin de s à x.

Cette propriété est vraie lors de l'initialisation (étape 1) car la file ne contient que s qui est effectivement à distance 0 et c'est le seul sommet de distance 0. Donc (a), (b) et (c) sont vraies.

On suppose que cette propriété est vraie avant un passage ; on retire le sommet x_1 et on ajoute en fin de F les voisins de x_1 non encore insérés y_1, ... , y_q (ceux pour lesquels $d[y]$ vaut -1).

Montrons que les propriétés (a), (b) et (c) sont conservées :

(a) On accède à ces sommets depuis x_1 donc leur distance est au plus $d[x_1] + 1$.
Comme les sommets à distance $d[x_1]$ ont été déjà ajoutés, leur distance est exactement $d[x_1] + 1$.
On a donc $d[x_1] \leq ... \leq d[x_p] \leq d[x_1] + 1 = d[y_1] \leq ... \leq d[y_q]$.

(b) Le nouveau sommet en début de file est x_2 (s'il existe), et y_1 sinon.

Si on a $d[x_2] = d[x_1]$ alors tous les sommets de distance $\leq d[x_2]$ ont été par hypothèse insérés.

Si on a $d[x_2] = d[x_1] + 1$ alors tous les sommets de distance $\leq d[x_1]$ ont étés traités (et retirés de la file) donc tous les sommets de distance $\leq d[x_2]$ ont été insérés (car ils sont nécessairement voisins d'au moins un sommet de distance $\leq d[x_1]$).

(c) Comme voisins de x_1, la distance de chaque y_j est bien $\leq d[x_1] + 1$.
Par la propriété (b) de l'hypothèse de récurrence, on a $d[y_j] > d[x_1]$, car sinon, y_j aurait déjà été inséré dans la file. Donc la distance à s de chaque y_j vaut effectivement $d[x_1] + 1$.

Ainsi la propriétés (a), (b) et (c) restent vraies.

Correction (partielle) : S'il existe un chemin de s à x, le sommet x est atteint (en un nombre d'étapes égal à sa distance) donc la valeur de $d[x]$ qui est renvoyée est bien la valeur minimale de la longueur d'un chemin de s à x. Sinon, $d[x]$ reste à -1. On en déduit qu'à la fin, la valeur de $d[x]$ est bien la distance pour les sommets x descendants de s, et vaut -1 sinon.

Terminaison : Lorsqu'un sommet x est ajouté à la file, $d[x]$ vaut nécessairement -1, et on affecte alors à $d[x]$ une valeur ≥ 0, qui n'est plus modifiée par la suite.
De ce fait, tout sommet est ajouté au plus une fois. D'autre part, à chaque étape, un élément est supprimé de la file. Donc l'algorithme termine en au plus n itérations (formellement, le nombre de sommets déjà extraits de la file est un entier qui croît strictement et est $\leq n$).