(♣) **Algorithmique : Structures et algorithmes de parcours** (♣)

1) Structures

a) **Piles** : FILO (first in, last out). Ajout et retrait en $O(1)$.

Implémentation : Liste (et les opérations append et pop).

b) **Files d'attente** : FIFO (first in, first out). Ajout et retrait en $O(1)$.

Implémentation : Tableau (de taille $\geq$ file) avec un indice de début et un indice de fin.

Autre implémentation : Avec deux piles.

c) **File de priorité** : Retrait selon l'ordre de priorité attribué à chaque élément.

Gestion par une liste : Retrait en $O(1)$, ajout et modificaiton de priorité d'un élément en $O(n)$.

Gestion par un tas : Ajout, retrait et modification de priorité d'un élément en $O(\log n)$.

d) **Dictionnaires** : Table de couples (*clé,valeur*) : Accès, ajout et suppression en temps amorti $O(1)$.

2) Algorithmes gloutons

On peut essayer de résoudre un problème d'optimisation en écrivant un algorithme qui énumère toutes les possibilités afin de trouver la meilleure. C'est un algorithme souvent inutilisable à cause du coût (souvent exponentiel).
Un algorithme glouton permet d'obtenir une solution rapidement, mais qui ne sera pas toujours optimale.
Les algorithmes gloutons suivent une stratégie simple : à chaque étape exécutée par un algorithme, se présente un ensemble de choix et un algorithme glouton fait le meilleur choix parmi les propositions. Un choix glouton est donc un choix localement optimal.
La question est de savoir si en faisant une série de choix localement optimaux, on finit par aboutir à une solution optimale. C'est parfois le cas mais pas toujours.

Exemple du rendu de monnaie. Selon le système S de pièces utilisé, l'algorithme glouton est ou non optimal. Le système usuel $S = \{1, 2, 5, 10, 20, 50, 100, 200\}$ donne une solution optimale.
En revanche, ce n'est pas le cas avec $S = \{1, 4, 6\}$. L'algorithme glouton donne $8 = 6 + 1 + 1$.

3) Problème de plus-courts chemins issu d'un sommet s (source) dans un graphe valué

a) Parcours en largeur

On considère ici le cas où tous les poids des arêtes valent 1.
La distance $d(x)$ de s à un sommet x est la longueur (= nombre d'arêtes) minimale d'un chemin de s à x.
L'idée pour calculer $d(x)$ est de parcourir les sommets dans un ordre correspondant à la distance à s.

- On pourrait calculer successivement la liste $L[k]$ des sommets à distance k :

Pour ce faire, on utilise un tableau de marquage pour indiquer les sommets à distance $\leq k$.
On détermine alors $L[k+1]$ comme les successeurs non visités des sommets de $L[k]$.

- Généralement, on utilise plutôt une file d'attente dont les éléments $[x_0, ..., x_{r-1}]$ vérifient :

$k = d(x_0) = d(x_1) = ... = d(x_{s-1}) \leq d(x_s) = ... = d(x_{r-1}) \leq k+1$.

A chaque étape, on sélectionne x_0 qu'on supprime de la file, et on ajoute les successeurs de x_0 non encore visités qui sont donc à une distance $d(x_0)+1$. Initialement, la file vaut $[s]$.

En pratique, on gère le tableau d des distances qu'on met à jour et un tableau de marquage (en fait, **le tableau d peut servir lui-même de tableau de marquage** en prenant par exemple initialement $d[x] = -1$).

- La complexité est linéaire en $O(n+m)$.

b) **Algorithme de Dijkstra**

On suppose qie toutes les arpêtes ont un poids positifs, c'est-à-dire $\omega(x,y) \geq 0$.

Soit s et x deux sommets. On appelle distance de s à x le poids minimum d'un chemin de s à x.

On recherche ou bien les distances de s à *tout* sommet, ou bien la distance de s à *un* sommet t fixé.

Idée clé de l'algorithme de Dijkstra :

On parcourt les sommets x selon le poids minimal de s à x.

- gérer une liste F des sommets déjà traités

- gérer une liste B (dite bordure) des successeurs de F (qui ne sont pas dans F).

On utilise le tableau d des distances mis à jour à chaque étape et vérifiant :

$$\begin{cases} \text{Pour } x \in F,\ d(x) = \text{poids minimal d'un chemin reliant } s \text{ à } x \\ \text{Pour } x \in B,\ d(x) = \text{poids minimal d'un chemin reliant } s \text{ à } x \textbf{ de sommets intermédiaires dans } F \end{cases}$$

- Principe de l'algorithme : à chaque étape, on choisit le point $x \in B$ minimisant $d(x)$.

On ajoute x à F et on met à jour B et les $d(y)$ pour $y \in B$ voisins de x.

- *Remarque* : Lorsque on cherche seulement la distance de s à t fixé, l'algorithme de Dijkstra parcourt seulement les sommets dont la distance est $\leq d(t)$.

Complexité : Pour gérer B, il faut utiliser **une file de priorité** (dont on en extrait l'élément de poids minimal).

La complexité va dépendre du choix de la structure utilisée.

Si on code B via une liste (extraction, modification et ajout en $O(n)$), le coût est en $O((n+m)n)$.

Si on utilise un tas (opérations en $O(\log n)$), alors la complexité est en $O((n+m)\log n)$.

c) **Extension à l'algorithme A^* avec une heuristique évaluant le coût de s à t (sommet terminal)**:

On définit le coût total $f(x) = d(x) + h(x)$, où $h(x)$ est une évaluation du coût de x à t.

Principe de l'algorithme :

- à chaque étape, on choisit le point $x \in B$ minimisant $f(x)$

- on ajoute x à S et on met à jour B et les $d(y)$ pour $y \in B$ voisins de x.

Remarque : Lorsque h vérifie $h(x) - h(y) \leq \omega(x,y)$, alors l'algorithme est optimal.

4) Programmation dynamique

Exemple fondamental :

On considère un graphe orienté acyclique $G = (S, A, \omega)$ valué (S ensemble des sommets, $A \subset S \times S$ ensemble des arêtes et $\omega : A \times A \to \mathbb{N}$), considérons f définie par

$$\begin{cases} f(x) = 0 \text{ si } x \in S \text{ est terminal (c'est-à-dire sans successeur)} \\ f(x) = \min\{\ f(y) + \omega(x, y),\ y \text{ successeur de } x\ \} \end{cases}$$

a) Programmation de haut en bas : Fonction récursive. Complexité exponentielle.

b) Programmation de bas en haut.

Utilisation d'un dictionnaire en utilisant les couples clé-valeur $(x, f(x))$.

On calcule $f(x)$ après que les $f(y)$ ont été calculés et **stockés** dans le dictionnaire. Le stockage des valeurs (*mémoïsation*) est un point essentiel : il permet de ne calculer qu'une seule fois la valeur de chaque $f(x)$.

Remarque : **Dans certaines situations, on peut directement utiliser un tableau :**

Il suffit en effet de disposer d'une numérotation des sommets de sorte que pour les successeurs de tout sommet i soients des sommets de numéro $j < i$.

Par exemple, si f est définie sur $S = [\![0, n-1]\!]$ par $f(0) = 0$ et $f(i) = \min_{j<i}(f(j) + c(i, j))$.

c) **Lien entre la programmation dynamique et la recherche d'un chemin de poids minimal.**

On considère le graphe $G = (S, A)$ et $\omega(y, x) = c(y, x)$ si $y < x$, et $\omega(y, x) = +\infty$ sinon.

Quitte à ajouter un sommet racine reliant tous les sommets sans prédécesseur (avec des coûts de valeur 0), on se ramène à la recherche d'un chemin de poids minimal issu du de la racine.

5) Parcours en profondeur

Les parcours en profondeur se codent aisément par récurrence (ou bien par une pile).

Principe :

- on utilise un tableau de marquage
- on définit une fonction auxiliaire récursive.

Exemple : Tableau des *profondeurs dans un graphe acyclique* :

Dans un graphe acyclique, la profondeur $p(x)$ est la longueur du plus long chemin issu du sommet :$p(x) = 0$ si x est terminal, et $p(x) = 1 + \max_{x \to y} p(y)$ si x non terminal.

```
def profondeur(G:((int))) :     ### le graphe G est la liste des listes d'adjacence
    n = len(G) ; p = [-1]*n
    def traite(x:int) -> None :   ###  traite est une procédure
        if p[x] == -1 :      # si x non encore traité
            if G[x] == [] :  p[x] = 0     # si x terminal
```

```
            else :
                 m = 0
                 for y in G[x] :
                      traite(y) ; m = max(m,p(y))
                 p[x] = 1 + m
    for x in range(n) :  traite(x)
    return p
```

Remarque :

On peut de même calculer le tableau des *hauteurs* en utilisant les profondeurs du graphe transposé.

De façon générale, on peut ainsi construire toute fonction f définie à l'exemple fondamental du 4).

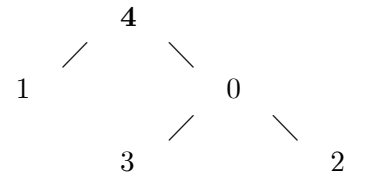
6) Arbres (*complément*)

a) **Un arbre** est un graphe (non orienté) acyclique connexe. Étant donnés deux sommets arbitraires, il existe donc un unique chemin (injectif) les reliant.

Culturel : Un graphe a n sommets est un arbre ssi il est connexe et admet exactement $(n-1)$ arêtes.

Codage des arbres enracinés : On représente le tableau des pères qui à chaque sommet associe son père (en considérant en général que la racine admet elle-même comme père).

Par exemple,



est codé par le tableau des pères

4	4	0	0	**4**
0	1	2	3	**4**

b) **Arbres couvrant associés aux parcours de graphe**

Les parcours (en largeur ou profondeur issus d'un sommet donné s) dans un graphe $G = (S, A)$ permettent d'obtenir un arbre couvrant tous les sommets de la composante connexe de s et dont les arêtes appartiennent à A. Pour récupérer les chemins associés, il suffit de construire le tableau des pères au fur et à mesure des itérations.

Remarque : En itérant le procédé, on obtient une forêt couvrante.