Un rapide résumé de la matière du cours LINFO1121

Guillaume Derval & contributeurs

21 décembre 2020

Un avertissement

Ces slides sont un résumé extrêmement incomplet du cours, et ne font aucunement figure d'autorité.

Matière du cours = chapitres du livre demandés + exercices TP + exercices INGInious

Utilisez ces slides pour vous rafraichir rapidement la mémoire, ou comme moyen mnémotechnique.

Personnellement, quand je dois comme vous apprendre un algorithme, je le lis, ainsi que ses propriétés, je le code au moins une fois sans référence, je vérifie, puis je me fais le genre de petit résumé tel que dans ces slides que je retiens. Le petit résumé permet de reconstruire le reste... si je l'ai bien étudié (le reste) au départ !

Ces slides ne recouvrent pas l'entièreté de la matière.

Les implémentations des TADs vus existantes en Java

Les implémentations des TADs vus existantes en Java

Ces slides passent quelques minutes sur les implémentations des différents TADs vus au cours. Ils mêlent TADs "purs" (sans implémentation) avec des TADs "pré implémentés" par Java. À vous de savoir faire la différence entre un TAD et une implémentation ;-)

Quand vous voyez une complexité en rouge, et que vous comptez utiliser cette méthode dans une structure de donnée, ça veut probablement dire que vous avez choisi la mauvaise structure de donnée.

Toutes les méthodes qui ne sont pas listées sont généralement non-standard ET couteuses, donc à éviter. La Javadoc est votre amie.

Vous n'êtes pas à l'abri d'une erreur dans ces slides ;-)

Tableaux (non-dynamiques)

Utilité	Stockage d'un nombre connu d'éléments
Contrainte	Nombre d'élément fixé
Instanciation	`new Type[taille]` (ex : `new int[8]`)

Méthodes

Nom	*Java*	*Complexité*
Accès à l'élément `i`	`tableau[i]`	$\Theta(1)$
Assignation de `x` à l'élem `i`	`tableau[i] = x`	$\Theta(1)$

Tableaux (dynamiques)

Utilité	Stockage d'éléments, accès aléatoire
Instanciation	`new ArrayList<Type>()`

Méthodes sur un objet `o` de taille n

Nom	*Java*	*Complexité*
Accès à l'élément `i`	`o.get(i)`	$\Theta(1)$
Assignation de `x` à l'élem `i`	`o.set(i, x)`	$\Theta(1)$
Ajouter à la fin	`o.add(x)`	$\Theta(1)$ amorti
Supprimer à la fin	`o.remove(n-1)`	$\Theta(1)$ amorti
Ajouter n'importe où	`o.add(i, x)`	$\mathcal{O}(n)$
Supprimer n'importe où	`o.remove(i)`	$\mathcal{O}(n)$
Contient `x` ?	`o.contains(x)`	$\mathcal{O}(n)$

Liste (doublement) chainées

Utilité	Stockage d'éléments, accès séquentiel
Instanciation	`new LinkedList<Type>()`

Méthodes sur un objet `o` de taille n

Nom	*Java*	*Complexité*
Accès au premier élément	`o.getFirst()`	$\Theta(1)$
Ajouter au début	`o.addFirst(x)`	$\Theta(1)$
Supprimer au début	`o.removeFirst()`	$\Theta(1)$
Accès au dernier élément	`o.getLast()`	$\Theta(1)$
Ajouter à la fin	`o.addLast(x)`	$\Theta(1)$
Supprimer à la fin	`o.removeLast()`	$\Theta(1)$
Accès à l'élément `i`	`o.get(i)`	$\mathcal{O}(n)$
Assignation de `x` à l'élem `i`	`o.set(i, x)`	$\mathcal{O}(n)$
Ajouter n'importe où	`o.add(i, x)`	$\mathcal{O}(n)$
Supprimer n'importe où	`o.remove(i)`	$\mathcal{O}(n)$
Contient `x` ?	`o.contains(x)`	$\mathcal{O}(n)$

Queue

Utilité	First-in, first-out (FIFO)
Instanciation	`Queue<Type> o = new LinkedList<>()`
	(`Queue` est une interface en Java !)

Méthodes sur un objet `o` de taille n

Nom	*Java*	*Complexité*
Accès au premier élément	`o.element()`	$\Theta(1)$
Ajouter (à la fin)	`o.add(x)`	$\Theta(1)$
Supprimer (au début)	`o.remove()`	$\Theta(1)$

Quand vous utilisez le TAD Queue, vous ne devriez JAMAIS devoir utiliser d'autres méthodes (si ce n'est `size` et `empty`).

Attention, ces fonctions peuvent retourner des exceptions si la Queue est vide.

Stack

Utilité	Last-in, first-out (LIFO)
Instanciation	`Stack<Type> o = new Stack<>()`

Méthodes sur un objet o de taille n

Nom	*Java*	*Complexité*
Accès à l'élément au dessus de la stack	`o.peek()`	$\Theta(1)$
Ajouter (au dessus)	`o.push(x)`	$\Theta(1)$
Supprimer (au dessus)	`o.pop()`	$\Theta(1)$

Quand vous utilisez le TAD Stack, vous ne devriez JAMAIS devoir utiliser d'autres méthodes (si ce n'est `size` et `empty`).

Vous pouvez (mais ne *devez* pas) jeter également un oeil à l'interface Deque (double-ended queue, qui implémente à la fois les TADs Stack et Queue) et ses implémentations : `LinkedList` et `ArrayDeque`.

Ne JAMAIS itérer (foreach ou `iterator`) sur une stack. Les éléments sont retournés à l'envers...

Liste triée (custom)

Utilité	Stockage d'éléments de manière triée
Instanciation	A implémenter vous-même à partir de LinkedList
	En pratique, je ne vois pas quand ça pourrait servir.
	Uniquement ici pour pouvoir comparer avec ce qui suit.

Méthodes sur un objet o de taille n

Nom	*Complexité*
Ajouter au bon endroit	$\mathcal{O}(n)$
Supprimer le plus petit/grand	$\Theta(1)$
Contient ?	$\mathcal{O}(n)$

Tableau trié (custom)

Utilité	Stockage d'éléments de manière triée
Instanciation	A implémenter vous-même à partir de ArrayList ou d'un tableau Java
	En pratique, je ne vois pas quand ça pourrait servir.
	Uniquement ici pour pouvoir comparer avec ce qui suit.

Méthodes sur un objet o de taille n

Nom	*Complexité*
Ajouter au bon endroit	$\mathcal{O}(n)$
Supprimer le plus petit	$\Theta(n)$
Supprimer le plus grand	$\Theta(1)$
Contient ?	$\mathcal{O}(\log n)$ (binary search)

TreeSet : Arbre autobalançant (ensemble)

Utilité	Ensemble, trié
Instanciation	`new TreeSet<Type>`
C'est quoi ?	Derrière, c'est un red-black tree
	Pas de doublons (c'est un ensemble)
	Les éléments doivent être comparable

Méthodes sur un objet `o` de taille n

Nom	*Java*	*Complexité*
Ajouter	`o.add(x)`	$\mathcal{O}(\log n)$
Supprimer	`o.remove(x)`	$\mathcal{O}(\log n)$
Contient ?	`o.contains(x)`	$\mathcal{O}(\log n)$
Trouver l'élément en dessous (ou =)	`o.floor(x)`	$\mathcal{O}(\log n)$
Trouver l'élément au dessus (ou =)	`o.ceil(x)`	$\mathcal{O}(\log n)$
Trouver l'élément str. en dessous	`o.higher(x)`	$\mathcal{O}(\log n)$
Trouver l'élément str. au dessus	`o.lower(x)`	$\mathcal{O}(\log n)$

Autre méthode utile : `subset`, qui permet d'extraire une partie de l'arbre, à cout constant.
`descendingIterator` est aussi fort pratique.

HashSet : table de hachage (ensemble)

Utilité	Ensemble
Instanciation	`new HashSet<Type>`
C'est quoi ?	Une table de hachage
	Pas de doublons (c'est un ensemble)
	Les éléments doivent être hashable (! ! !)
	Non trié

Méthodes sur un objet o de taille n

Nom	*Java*	*Complexité*
Ajouter	`o.add(x)`	$\mathcal{O}(1)$ amorti
Supprimer	`o.remove(x)`	$\mathcal{O}(1)$ amorti
Contient ?	`o.contains(x)`	$\mathcal{O}(1)$ amorti

Sous contrainte que les hash sont bien répartis !

TreeMap : arbre autobalançant (dictionnaire)

Utilité	Dictionnaire
Instanciation	`new TreeMap<TypeCle,TypeValeur>`
C'est quoi ?	Un red-black tree (ou équivalent)
	Pas de doublons dans les clés (c'est un dictionnaire)
	Les clés doivent être comparables

Méthodes sur un objet o de taille n

Nom	*Java*	*Complexité*
Ajouter (clé `x`, valeur `y`)	`o.put(x, y)`	$\mathcal{O}(\log n)$
Get clé	`o.get(x)`	$\mathcal{O}(\log n)$
Supprimer clé	`o.remove(x)`	$\mathcal{O}(\log n)$
Contient clé ?	`o.containsKey(x)`	$\mathcal{O}(\log n)$

Les fonctions de TreeSet sont aussi disponible pour les clés (`ceilingKey`, `subMap`...).

Voir slide plus loin pour un tips pour itérer sur les Maps et notamment les TreeMaps.

Même s'il existe des méthodes pour chercher des valeur (`containsValue`, ...), elles sont en $\mathcal{O}(n)$ généralement.

HashMap : table de hachage (dictionnaire)

Utilité	Ensemble
Instanciation	`new HashMap<TypeClé,TypeValeur>`
C'est quoi ?	Une table de hachage
	Pas de doublons de clé (c'est un dictionnaire)
	Les clés doivent être hashable (! ! !)
	Non trié

Méthodes sur un objet o de taille n

Nom	*Java*	*Complexité*
Ajouter (clé x, valeur y)	`o.put(x, y)`	$\mathcal{O}(1)$ amorti
Get clé	`o.get(x)`	$\mathcal{O}(1)$ amorti
Supprimer clé	`o.remove(x)`	$\mathcal{O}(1)$ amorti
Contient clé ?	`o.containsKey(x)`	$\mathcal{O}(1)$ amorti

Sous contrainte que les hash sont bien répartis !

Voir slide plus loin pour un tips pour itérer sur les Maps et notamment les TreeMaps.

Même s'il existe des méthodes pour chercher des valeur (`containsValue`, ...), elles sont en $\mathcal{O}(n)$ généralement.

PriorityQueue : file de priorité

Utilité	Queue, mais avec des priorités.
Instanciation	`new PriorityQueue<Type>`
C'est quoi ?	Un heap binaire basée sur un tableau
	Les valeur doivent être comparables

Méthodes sur un objet `o` de taille n

Nom	*Java*	*Complexité*
Ajouter	`o.add(x)`	$\Theta(\log n)$
Voir premier élément	`o.peek(x)`	$\Theta(1)$
Retirer premier élément	`o.poll(x)`	$\Theta(\log n)$

Si vous utilisez n'importe quelle autre méthode (`remove`, `iterator`, ...), une PQ n'est pas la bonne structure.
Généralement c'est du $\mathcal{O}(n)$.

L'itérateur sur les PQ ne donne pas les éléments dans l'ordre.

Union-find

Utilité	Ensemble disjoints. Fusion d'ensembles.
Instanciation	n'existe pas en java, à faire vous-même !
	Implémenter le weighted quick union !

Méthodes sur un objet o de taille n

Nom	*Java*	*Complexité*
Find	`o.find(x)`	$\mathcal{O}(\log n)$
Union	`o.union(x, y)`	$\mathcal{O}(\log n)$

Avec la path compression, vous pouvez descendre jusque $\mathcal{O}(\alpha(n))$ amorti (α étant la fonction inverse d'Ackermann (cfr wikipédia ;-)). $\alpha(2^{32}) < 4$).

Quel TAD/implémentation choisir ?

Quel TAD/implémentation choisir ?

Liste non-exhaustive et heuristique évidemment...

- Je veux juste stocker des éléments...
  - Et faire de l'accès séquentiel → `LinkedList`
  - Et faire de l'accès aléatoire...
    - Et ajouter des éléments au fur et à mesure → `ArrayList`
    - Et je connais la taille au départ → tableau `type[]`
- Je veux maintenir un ensemble (sans doublon) → `HashSet` (hashable) ou `TreeSet` (comparable)
- Je veux utiliser contains → `HashSet` (hashable) ou `TreeSet` (comparable)
- Je veux maintenir un ensemble trié → `TreeSet`
- Je veux faire un mapping clé vers valeur...
  - trié → `TreeMap`
  - non-trié → `HashMap`
- Je veux faire de l'ajout/suppression/get, seulement au début et à la fin → `LinkedList`, `Queue` ou `Stack` en fonction
- J'ai des priorités à gérer, toujours le truc le plus prioritaire en premier → `PriorityQueue`
- Je veux itérer sur le contenu → n'importe quoi sauf `Stack, Queue, PQ, UF, ...`
- Je veux fusionner des ensembles → UF

Mais comment ça marche ?

Binary Search

- Dans un tableau trié, disons qu'on cherche un élément x.
- Prendre le milieu du tableau, y. Si $x = y$, on a gagné.
- Si $x < y$, alors x est forcément à gauche ; sinon, il est à droite.
- Appeler récursivement sur la moitié gauche ou droite du tableau en fonction.

$\mathcal{O}(\log n)$

Propriétés des tris

- En-place : un tri en place utilise un quantité constante de mémoire supplémentaire (complexité en mémoire **supplémentaire** $\mathcal{O}(1)$).
- Stable : Deux valeurs égales du point de vue du tri restent dans le même ordre après le tri.

Selection sort

Dans un array non-trié :

- Trouver l'élément le plus petit pas encore trié
- Le mettre au bon endroit
- Répeter

$\mathcal{O}(n^2)$

Insertion sort

Créer un array. Pour chaque élément de l'array non trié :

- Trouver l'endroit où le mettre
- L'y mettre
- Eventuellement décaler tout ce qui se trouve à droite...

$\mathcal{O}(n^2)$

Merge sort

- Faire des petits paquets de 2
- Trier les paquets de deux
- Fusionner les paquets deux par deux, avec la technique du "double pointeur" (qui est en $\mathcal{O}(n)$ pour rappel)
- Et ce jusqu'à ce que tout soit fusionné

$\mathcal{O}(n \log n)$

Quick sort

- Choisir un pivot
- Mettre les éléments plus petit que le pivot à gauche du pivot, les autres à droite
- Appeler récursivement à gauche du pivot et à droite du pivot

$\mathcal{O}(n \log n)$ moyen, $\mathcal{O}(n^2)$ en général

Quick select

- Trouve le $k$ième plus grand élément dans un tableau non trié
- Choisir un pivot
- Mettre les éléments plus petits que le pivot à gauche du pivot, les autres à droite
- Si le pivot se retrouve à la position k, stop (c'est l'élément recherché).
- Si pas, si $k <$ position du pivot, appeler récursivement uniquement à gauche, et à droite sinon.

$\mathcal{O}(n)$ moyen, $\mathcal{O}(n^2)$ en général

Heap sort

- Transformer l'array en heap (heapify en $\mathcal{O}(n)$) ou mettre tout dans un heap un à un ($\mathcal{O}(n \log n)$)
- Depop les éléments du heap un à un. Ils sont donc triés. $\mathcal{O}(n \log n)$

$\mathcal{O}(n \log n)$

Heap binaires

- Arbre binaire complet (sauf éventuellement la dernière couche, qui est complète "a gauche")
- Propriété : les enfants d'un noeud sont plus grands que lui (pour un min-heap)
- Implémentation via un tableau. Les enfants du noeud i sont en position $2i$ et $2i + 1$. Attention, par simplicité on commence la numérotation à 1.
- Ajout : mettre dans la première position libre en bas à gauche. Faire remonter (échanger avec le parent) jusqu'à ce que la propriété soit respectée.
- Suppression : l'élément le plus petit est la racine. Remplacer la racine par l'élément le plus en bas à droite. Faire descendre (échanger avec l'enfant le plus petit) jusqu'à ce que la propriété soit respectée.

$\mathcal{O}(\log n)$ pour tout, $\mathcal{O}(1)$ pour voir l'élément le plus petit

Min-max heap

- Variante du heap binaire, qui maintient à la fois le minimum et le maximum et permet de les retirer en $\mathcal{O}(\log n)$
- Propriété : sur les couches "paires" (niveau 0 (racine), 2, 4, ...) la propriété du min-heap est d'application (tous les noeuds en dessous sont plus grands). Pour les couches impaires, c'est la propriété de max-heap.
- Forcément, le minimum est la racine, et le maximum est le maximum des deux premiers enfants.
- Il faut adapter les règles pour swim/sink les valeurs (une feuille de papier et un petit dessin devraient être suffisants pour retrouver la règle, un peu trop longue à mettre dans ce slide).

$\mathcal{O}(\log n)$ pour tout, $\mathcal{O}(1)$ pour voir l'élément le plus petit/grand.

Arbres de recherche

- Ici : binaire. Facile à adapter dans les autres cas...
- Propriété : **tous** les noeuds à gauche (enfants, petits-enfants...) sont plus petits que la valeur du noeud courant. Et plus grands à droite.
- Difficiles à équilibrer, donc généralement (si on ne précise pas qu'ils sont équilibrés) toutes les opérations sont en $\mathcal{O}(n)$.
- Pour info il existe des implémentations binaires équilibrées, comme les arbres AVL.
- Mais dans ce cours on a vu les arbres 2-3 et red-black (left leaning).
- Ajout : descendre récursivement en voyant de quel côté est ce qu'on cherche.

Arbre 2-3

- Arbre contenant soit des noeuds 2 (2 enfants), soit des noeuds 3 (3 enfants).
- Les noeuds 2 fonctionnent comme les arbres de recherche. Les noeuds 3 ont deux valeurs. À gauche plus petit que la première, au milieu entre la première et la deuxième valeur, à droite plus grand que la deuxième valeur
- Les arbres 2-3 sont toujours complets et donc équilibrés.
- Les règles d'ajout sont complexes à énumérer, MAIS intuitives. On descend là où la valeur devrait être placée, on l'ajoute au noeud. Si le noeud devient un noeud 4 (trois valeurs) on doit le diviser en faisant remonter une valeur au noeud supérieur.

Tout est en $\mathcal{O}(\log n)$ du coup.

Red-black tree

- Arbre binaire de recherche, avec un twist : certains liens pointant à gauche peuvent être rouges.
- Il y a mapping one to one avec les arbres 2-3 $\rightarrow$ un noeud 2 est encodé normalement, un noeud 3 est encodé par deux noeuds, reliés par un lien rouge.
- Attention ; les RB-Tree d'internet sont différents (moins bien) que ceux vus ici. Ils peuvent avoir des liens rouges vers la droite.

Hash tables

- Principe : hasher l'objet, et le mettre dans un tableau à l'endroit ou pointe son hash.
- Attention à la qualité du hash, aux collisions et à la taille de la table

Si tout va bien, tout se fait en $\mathcal{O}(1)$ amorti.

Hash tables - Separate chaining

- Chaque entrée de la table est en fait une liste chainée
- Quand collision, ajout à la liste chainée

Hash tables - Linear probing

- Quand collision, aller voir à la case "après"
- Attention à la suppression...

Rabin-Karp

- Sert à trouver un/des mots-clés (de même taille) dans une longue string (ou suite de nombre...)
- Principe : hash roulant
- On calcule en premier le hash de tout les mots-clés à trouver
- On met ces hash dans un dictionnaire (hash -> mot-clé)
- On calcule ensuite le hash roulant au fur et à mesure dans le string original, et on regarde dans le dictionnaire si le hash est présent ou pas.
- Attention aux collisions ! Vérifier le string original.
- Si le dictionnaire est un `HashMap`, on est en $\mathcal{O}(n)$ si pas trop de collisions. Techniquement $\mathcal{O}(nm)$ (m=taille du plus grand mot clé).

Hash pour les k premiers caractères après la position i dans le tableau x et une constante R ($R = 31$ est un bon choix) :

$$h_i = x_i R^{k-1} + x_{i+1} R^{k-2} + \ldots + x_{i+k-2} R + x_{i+k-1} = \sum_{j=0} k - 1 x_{i+j} R^{k-1-j}$$

Pour calculer le hash d'après :

$$h_{i+1} = (h_i - x_i R^{k-1}) \cdot R + x_{i+k}$$

Union-find

- Représente un ensemble d'ensembles
- Chaque ensemble est représenté par un de ses membres, appelé le représentant
- L'opération find permet de trouver le réprésentant d'un élément (autrement dit, le représentant de l'ensemble auquel il appartient)
- L'opération union merge deux ensembles, étant donné deux éléments appartenant à deux ensembles différents.
- Au départ, chaque élément est dans un ensemble tout seul.

Union-find (implémentation)

- Ici : weighted quick-union
- Un tableau `t`, avec un élément par... élément
- Les ensembles forment un arbre. Ce tableau indique, si `t[i]=j` que le noeud `i` pointe vers `j` (et qu'ils sont donc tous les deux dans le même arbre/ensemble).
- La racine de l'arbre est le représentant
- Un second tableau stocke la taille de chaque ensemble
- On unionne toujours le plus petit ensemble sur le plus grand, pour réduire la taille de l'arbre.
- Path compression : après chaque find, on fait pointer le noeud vers son représentant, pour compresser l'arbre. Ca diminue fortement la complexité !

Tout est en $\mathcal{O}(\log n)$ sans path compression. Bonus (pas dans la matière) : $\mathcal{O}(\alpha(n))$ avec path compression.

Petit rappel sur les graphes

- Graphe : ensemble de noeuds et d'arêtes reliant ces noeuds. ($G =< V, E >$)
- Graphe *pondéré* : les arêtes ont un poids associé. Mathématiquement, usuellement, on définit ça via une fonction $f : E \to \mathbb{R}$ qui donne un poids quand on lui donne une arête.
- Graphe *simple* : il y a, entre toute paire de noeuds, au plus une arête. Cela implique que $E \leq \frac{V \cdot (V-1)}{2} \in \mathcal{O}(V^2)$ et donc que $\mathcal{O}(\log E) \subseteq \mathcal{O}(\log V^2) = \mathcal{O}(2 \log V) = \mathcal{O}(\log V)$.
- Graphe *orienté* : les arêtes ont une direction. Généralement, on appelle alors ces arêtes des *arcs*.
- Degré d'un noeud : nombre d'arêtes touchant un noeud.
- Degré entrant (resp. sortant) : nombre d'arcs dont le noeud est la destination (resp origine.).
- Somme des degrés des noeuds $= 2E$
- Graphe *acyclique* : sans cycle (orienté ou pas, en fonction).

BFS

- Objectif : visiter tout le graphe en partant d'un/plusieurs noeuds. D'abord visiter tous les noeuds à distance 1, puis distance 2, etc. Les distances sont sur le graphe **non-pondéré** (on compte juste le nombre d'arêtes).
- On utilise une Queue avec au départ tout les noeuds de départ
- Boucle tant que la queue n'est pas vide :
  - Pop queue
  - Ajouter tous les voisins du noeud pop si pas encore dans la Queue (faire tableau de booléen sur le côté pour savoir)
  - Eventuellement maintenant un tableau de distance en même temps

$\mathcal{O}(V + E)$

DFS (while loop)

- Objectif : visiter tout le graphe en partant d'un/plusieurs noeuds. Ordre : plus profond d'abord
- On utilise une Stack avec au départ tout les noeuds de départ
- Boucle tant que la stack n'est pas vide :
  - Pop stack
  - Ajouter tous les voisins du noeud pop si pas encore dans la Queue (faire tableau de booléen sur le côté pour savoir)
- Même algo que BFS à une ligne près (instantiation du Stack). Pas d'excuses !

$\mathcal{O}(V + E)$

DFS - Récursif

L'implémentation récursive est très simple à coder et permet de maintenir l'état de chaque noeud :

- Non visité (pas encore "vu")
- Ouvert (le noeud a été vu et on est en train de visiter ses enfants/petits-enfants...)
- Fermé (le noeud a été vu et on a fini de visiter ses enfants/petits-enfants/...)

```
// neis is an adjacency list (LinkedList[])
// state is an array containing the state of each node, initially set to NOTSEEN

void dfs(int nodeIdx) {
        state[nodeIdx] = OPEN;

        // generalement, vous devez inserer le code pour resoudre le probleme ici

        for(int nei: neis[nodeIdx]) {
                if(state[nei] == NOTSEEN)
                        dfs(nei);
        }
        state[nodeIdx] = CLOSED;
}
```

Trouver un cycle

En utilisant le DFS récursif ; si on visite un noeud déjà OPEN, c'est que nous sommes en train de visiter ses enfants/petits-enfants/... sauf qu'on le retrouve de nouveau, et donc nous sommes dans un cycle.

Attention, ce n'est pas vrai avec CLOSED ; dans un graphe dirigé, cela peut arriver (faites un dessin !).

```
bool hasCycle(int nodeIdx) {
        state[nodeIdx] = OPEN;

        for(int nei: neis[nodeIdx]) {
                if(state[nei] == UNSEEN)
                        if(dfs(nei))
                                return true;
                if(state[nei] == OPEN)
                        return true;
        }
        state[nodeIdx] = CLOSED;
        return false;
}
```

Savoir si un graphe est biparti

- Graphe biparti : le graphe peut être séparé en deux ensembles de noeuds, tel qu'il n'y a pas d'arête à l'intérieur d'un ensemble.
- Colorer le graphe alternativement avec deux couleurs et un DFS. Si un noeud se retrouver à être déjà coloré de la mauvaise couleur, il n'est pas biparti.

$\mathcal{O}(V + E)$

Toposort

- Etant donné un graphe dirigé acyclique, trouver un ordre des noeuds tel que toutes les edges vont "vers la droite" (pointent vers des noeuds situés après dans l'ordre).
- Algorithme : tant qu'il y a des noeuds sans arêtes entrantes, les ajouter à l'ordre. Retirer les arêtes sortantes de ces noeuds du graphe. Répeter.
- Ca permet de trouver les cycles aussi. Mais un DFS sait aussi le faire ça ;-)

$\mathcal{O}(V + E)$

Arbre sous-tendant de poids minimal

- Arbre : graphe connexe acyclique. n noeuds $\Rightarrow n-1$ arêtes
- sous-tendant : par rapport à un autre graphe, l'arbre en utilise tous les noeuds et un subset des arêtes.
- de poids minimal : il n'existe pas d'arbre sous-tendant avec un poids (somme des poids des arêtes) plus petit

Prim

- Calcul d'un arbre sous-tendant
- On prend un noeud au hasard
- On ajoute toutes ses arêtes dans une PQ
- On retire l'arête la plus petite
- Si elle pointe vers un noeud pas encore dans l'arbre, on l'y ajoute (et le noeud aussi), et on ajoute les arêtes dudit noeud à la PQ
- Repeat

$\mathcal{O}(E \log V)$

Kruskal

- Calcul d'un arbre sous-tendant
- Partir d'un graphe sans les arêtes, mais avec tout les noeuds
- Trier toute les arêtes par poids
- Pour chaque arête :
- Si mettre l'arête ne crée pas de cycle (à vérifier avec un UF), l'ajouter au graphe
- S'arrêter après $n - 1$ arêtes ajoutées.

$\mathcal{O}(E \log V)$

Dijkstra

- Calcul de tous les chemins les plus courts (au sens "somme des poids des arêtes du chemin") partant d'un noeud donné, dans un graphe **pondéré**.
- Maintien d'un tableau de distance vers tout les noeuds, initialement à l'infini
- La distance vers le noeud de départ est 0
- Une PQ maintien les noeuds vus, mais pas encore visités
- Mettre le noeud original et sa distance actuelle dans la PQ
- Tant que la PQ n'est pas vide :
  - Pop un noeud (et le poids associé)
  - Si le poids associé n'est pas bon, ignore. Sinon...
  - Regarder tous les voisins du noeud. Si la nouvelle distance est plus courte, mettre à jour le tableau distance et ajouter noeud+distance à la PQ.
- Dès qu'un noeud est pop par la PQ, la distance min est connue.
- Attention, pas de support des poids négatif du coup...

$\mathcal{O}((V + E) \log V)$

Bellman-ford

- Calcul de tout les chemins les plus courts partant d'un noeud donné.
- L'idée est de calculer tout d'abord tous les chemins de taille 1 (d'une seule arête), puis tous les chemins de taille 2, ...
- L'implémentation fait 4 lignes.
- Support pour les poids négatif, mais pas les cycle négatifs !

```
fonction Bellman_Ford(G, s)
        pour chaque sommet u du graphe
                d[u] = +inf
        d[s] = 0

        pour k = 1 jusqu'a n-1 faire
                pour chaque arc (u, t) du graphe faire
                        d[t] := min(d[t], d[u] + poids(u, t))

        retourner d
```

$\mathcal{O}(V^3)$

Chemin le plus long

- Il est conjecturé qu'il n'existe pas d'algorithme en temps polynomial pour trouver le chemin le plus long dans un graphe quelconque. (plus d'information dans le cours de Calculabilité (Q6) et d'Algorithmique Avandée pour l'Optimisation (master)).
- Si le graphe est un DAG (graphe dirigé acyclique) alors il existe un algorithme polynomial : prendre l'opposé des poids des arêtes, et utiliser Bellman-ford.
- Cela marche car il y a aura des poids négatifs, mais pas de cycles négatifs vu que le graphe est acyclique.

Retenir les chemins dans les algorithmes de recherche de chemin

- Quand on fait un BFS/DFS/Dijkstra/Bellman-Ford, on a parfois (/souvent) besoin de retenir le chemin en plus de la distance
- La technique est simple : on va maintenir un tableau disant "d'où on vient" et construire le chemin à l'envers.
- Créer un tableau supplémentaire. Taille V, init à null.
- Quand vous "trouvez un meilleur chemin" (autrement dit :
  - DFS : quand vous ajoutez à la Stack/faites l'appel récursif
  - BFS : quand vous ajoutez à la Queue
  - Dijkstra/Bellman-ford : quand vous mettez à jour l'array distance

) vous mettez à jour l'entrée du noeud de destination pour la faire pointer vers le noeud courant.
- Vous pouvez ensuite reconstruire le chemin avec une simple boucle while et une `LinkedList` par exemple.

Tips & tricks

Itérer sur des collections

Je ne veux JAMAIS voir ceci :

```
List<String> x = ...;
for(int i = 0; i < x.size(); i++) {
        String elem = x.get(i);
        //...
}
```

C'est du $\mathcal{O}(n^2)$!

Utilisez svp les foreach :

```
for(String elem: x) {
        //...
}
```

Ou un itérateur :

```
Iterator<String> itr = x.iterator();
while(itr.hasNext()) {
        String elem = ite.next();
        //...
}
```

Itérer sur des collections (2)

Pour les dictionnaires (`HashMap`, `TreeMap`), trois possibilités raisonnables :

```
HashMap<X, Y> hasmap = ...;
for(Map.Entry<X, Y> pair: hashmap.entrySet()) {
        X cle = pair.getKey();
        Y valeur = pair.getValue();
        // ...
}

for(X key: hashmap.keySet()) {
        Y valeur = hashmap.get(key);
        // ...
}
```

Ou avec un itérator (sur `hashmap.entrySet().iterator()` ou `hashmap.keySet().iterator()`).

Si vous utilisez un TreeMap, c'est donné de manière triée ! Attention que c'est toujours du $\mathcal{O}(n)$ sauf dans le second cas avec la TreeMap, où là c'est $\mathcal{O}(n \log n)$.