Algorithmique avancée

Cours 1

## Objectif de la matière :

* Objectif général
  + Apprendre à utiliser des structures de données avancées en algorithmique et en programmation
* Compétences spécifiques
  + Définition et utilisation des tables de hachage
  + Définition, représentation et utilisation des graphes
  + Algorithmes de la théorie des graphes
  + Implémentations en langage C
* Prérequis
  + Info0301 : programmation en C
  + Info0401 : algorithmique

## Structure de la matière

* Partie 1 : introduction et rappels
  + Analyse et conception des algorithmes
* Partie 2 : graphes 1
  + Définition et représentation
  + Algorithmes de base
    - Parcours en largeur
    - Parcours en profondeur
    - Rappels sur les types de données élémentaires nécessaires : Piles, files, listes chainées, tables de hachage
* Partie 3 : graphes 2
  + Algorithmes classiques et/ou avancés
    - Tri topologique
    - Connexité
    - Plus courts chemins
    - Arbres couvrants de poids minimal
    - Flot maximal
    - Optimisation linéaire/combinatoire
    - Rappels sur les types de données élémentaire nécessaires : tas, arbres, ensembles disjoints

Support : T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, « algorithmique », 3e édition, 2010

# Le rôle des algorithmes en informatique

## Algorithme

* Procédure de calcul bien définie qui
  + Prend en entrée une valeur ou un ensemble de valeurs
  + Donne en sortie une valeur ou un ensemble de valeurs
* Séquence d’étapes de calcul qui transforment l’entrée en sortie
* Outil permettant de résoudre un problème de calcul
* Des problèmes complexes à résoudre par des ordinateurs nécessitent des algorithmes perfomants
  + Calcul scientifique : simulation de systèmes
  + Internet : recherche et manipulation de grands volumes de données
  + Commerce électronique : sécurité, encryptage
  + Industrie : allocation de ressource
  + …

## Algorithmes en tant que technologie

* Les ordinateurs
  + Ne sont pas infiniment rapides
  + N’ont pas une mémoire infinie
* Le temps machine et l’espace mémoire sont des ressources limitées
* Une machine moins performante qui exécute de bons algorithmes pourra surpasser une machine plus performante qui exécute de mauvais algorithmes

# Analyse des algorithmes

## Exemple : le problème du tri

* Entrée
  + Une suite de *n* nombres (a1, a2, …, an)
* Sortie
  + Une permutation (réorganisation)

(a1’, a2’, …, an’) de la suite donnée en entrée

* + De sorte que

a1’ < a2’ < … < an’

* La suite (31, 41, 59, 26, 41, 58)
  + Est une instance du problème de tri
* Opération majeure en informatique
  + Employée par une multitude de programmes comme phase intermédiaire
* L’algorithme optimal pour une application donnée dépend
  + Du nombre d’éléments à trier
  + De la façon dont les éléments sont plus ou moins triés initialement
  + Des restrictions sur les valeurs des éléments
  + …

(31, 41, 59, 26, 41, 58) 🡺 (26, 31, 41, 41, 58, 59)

## Un 1er algorithme : le tri par insertion

* Algorithme naturel pour l’être humain
  + Comment triez-vous vos cartes quand vous jouez aux cartes (si vous jouez aux cartes !) ?
  + On prend les cartes une à une et on les place au bon endroit dans sa main
* Efficace quand il s’agit de trier un petit nombre d’éléments
* Exemple 1
  + Application sur l’instance (5,2,4,6,1,3)
* Pseudo-code

|  |
| --- |
| TRI-INSERTION(t)  pour j=2 à t.longeur  clé = t[j]  //insère t[j] dans la séquence triée t[1...j-1]  i=j-1  tant que i>0 et t[i]>clé  t[i+1]=t[i]  i=i-1  t[i+1]=clé |

Fonctionnement d’un algorithme (Schéma 1) (slide 13 : tri par insertion)

## Correction d’un algorithme

* Un algorithme est dit correct si, pour chaque instance est entrée, il se termine en produisant la bonne sortie
  + Un algorithme correct résoud le problème donné
* Un algorithme incorrect peut
  + Ne pas se terminer pour certaines instances
  + Se terminer sur une réponse autre que celle voulue
* On montre qu’un algorithme est correct par les invariants de boucle

## Invariant de boucle

* Propriété qui est vraie à chaque passage dans la boucle
* On doit montrer 3 choses concernant un invariant de boucle
  + Initialisation : il est vrai avant la première itération de la boucle
  + Conservation : s’il est vrai avant une itération de la boucle, il le reste avant l’itération suivante
  + Terminaison : une fois terminée la boucle, l’invariant fournit une propriété utile qui aide à montrer la validité de l’algorithme

## Validité du tri par insertion

* Invariant de boucle

Au début de chaque itération de la boucle pour le sous-tableau t[1… j-1] se compose des éléments qui occupaient initialement les positions t[1…j-1] mais qui sont maintenant triés

* Initialisation
  + Avant la 1ère itération de la boucle, j=2
  + Le sous-tableau t[1…j-1] se compose donc uniquement de l'élément t[1]
  + … qui est l'élément originel de t[1]
  + … qui est trié (trivialité : un élément seul est nécessairement trié)
  + L'invariant est donc vérifié avant la 1ère itération de la boucle
* Conservation
  + Le corps de la boucle pour fonctionne en déplaçant t[j-1], t[j-2], t[j-3] etc. d'une position vers la droite jusqu'à ce qu'on trouve la bonne position pour t[j ]
  + Le sous-tableau t[1…j] se compose alors des éléments situés initialement dans t[1…j], mais en ordre trié
  + L'incrémentation de j pour l'itération suivante de la boucle pour préserve alors l'invariant
* Terminaison
  + La condition forçant la boucle pour à se terminer set que j>longueur =n
  + Comme chaque itération de la boucle augmente j de 1, on doit avoir j=n+1 à cet instant
  + En substituant n+1 à j dans la formation de l'invariant de boucle, on a que le sous-tableau t[1…n] se compose des éléments qui appartenaient originellement à t[1…n], mais qui ont été triés depuis
  + Or, le sous-tableau t[1…n] est le tableau complet, donc le tableau tout entier est trié, et donc l'algorithme est correct

## Analyse des algorithmes

* Prévoir les ressources nécessaires à cet algorithme
  + Temps de calcul
  + Mémoire
  + Largeur de bande
  + …
* C'est généralement le temps de calcul qui nous intéresse
* En analysant plusieurs algorithmes pour un problème, on peut identifier le plus efficace
* Nécessite un modèle de la technologie employée (ressources, coûts) : le modèle RAM

## Modèle RAM

* Modèle de calcul générique basé sur une machine à accès aléatoire à processeur unique
* Instructions exécutées l'une après l'autre
* Permet les instructions
  + Arithmétique (addition, soustraction,…)
  + De transfert de données (lecture, copie, …)
  + De contrôle (branchement, appel de routine,…)
* Chaque instruction a un temps d'execution constant
* Types de données : entier et réel (avec taille limitée)

## Analyse du tri par insertion

### Durée d'exécution de TRI-INSERTION

* Dépend de l'entrée
  + Temps pour 1000 nombres > temps pour 3 nombres
  + Temps peur être différent pour 2 entrées de même taille, selon qu'elles sont plus ou moins triées partiellement
* En général
  + Le temps d'exécution d'un algorithme croit avec la taille de l'entrée
  + On exprime donc le temps d'éxécution en fonction de cette taille

### Taille de l'entrée

* Dépend du problème étudié
* Pour le problème de tri
  + Nombre d'éléments constituant l'entrée
  + Longueur n du tableau à trier
* Peut etre le nombre total de bits nécessaire à la représentation de l'entrée en notation binaire
* Peut être plusieurs nombres plutôt qu'un seul
  + Pour les algorithmes de graphes
    - Nombre de sommets et nombre d'arc

## Temps d'exécution

* Nombre d'opération élémentaire exécutées
* On considère que chaque ligne de pseudo-code demande un temps constant
* Pour TRI-INSERTION
  + Temps d'exécution de la ligne i 🡪 ci
  + On multiplie ci par le nombre de fois que l'instruction est exécutée

## Analyse du temps d'exécution de TRI-INSERTION

## 

## Analyse du temps d'exécution

### Ce qui nous intéresse généralement le plus

* Temps d'éxécution dans le cas le plus défavorable
  + Temps d'exécution maximal pour une quelconque entrée de taille n
* Borne supérieure du temps d'exécution
  + Certitude qu'on ne pourra faire pire
  + Pour certains algorithmes, ce cas arrive souvent
  + Souvent, cas moyen presque aussi mauvais que le pire

### Ce qui nous intéresse vraiment

* Taux de croissance du temps d'exécution
  + Ou ordre de grandeur du temps d'exécution
* On ne considéra que le terme dominant d'une formule, sans les coefficients constants
  + an + b =
* Un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable a un ordre de grandeur inférieur

# Conception des algorithmes

## Techniques de conception

* Tri par insertion 🡪 approche incrémentale
  + Après avec trié t[1…j-1], on produit t[1…j]
* Approche diviser-pour-régner
  + Méthode récursive
  + L'algorithme s'appelle lui-même pour traiter des sous-problèmes similaires, mais de taille inférieure
  + Les solutions des sous-problèmes sont combinées pour produire la solution du problème original
* Les 3 étapes de l'approche diviser-pour-régner
  + Diviser
    - Création de sous-problèmes qui sont des instances plus petites du même problème
  + Régner
    - Traiter les sous-problèmes de façon récursive
    - Lorsque la taille d'un sous-problème est assez petite, on peut le résoudre directement
  + Combiner
    - Production de la solution du problème en combinant les solutions des sous-problèmes

## Exemple : le tri par fusion

* Diviser
  + Diviser la suite de n éléments à trier en 2 sous-parties de n/2 éléments chacune
* Régner
  + Trier les 2 sous-suites récursivement avec le tri par fusion
* Combiner
  + Fusionner les 2 sous-suites triées pour produire la réponse
* Arret de la récursivité
  + Séquence de longueur 1
* Exemple 2
  + Application de la fusion sur l'instance (2, 4, 5, 7, 1, 2, 3, 6)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 |

g :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 7 |  |

i

d :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 6 |  |

j

## La procédure FUSION

FUSION (t, p, q, r )

n1= q–p+ 1

n2= r–q

Créer tableaux g [1..n1 + 1] et d [1..n2+ 1]

pour i= 1 à n1

g[i] = t [p + i -1]

pour j= 1 à n2

d [i] = t [q+ j]

g [n1+ 1] = ∞

d [n2+ 1] = ∞

i= 1

j= 1

pour k= p à r

Si g[i] ≤ d[j]

t [k] = g [i]

i= i+ 1

Sinon

t [k] = d [j]

j= j+ 1

* 2 premières boucles pour
* 3e boucle pour
  + N itérations
  + Temps constant pour caque itération
* Temps d'exécution

## La procédure TRI-FUSION

TRI-FUSION (t, p, r)

Si p<r

q= [(p+r)/2]

TRI-FUSION(t, p, q)

TRI-FUSION(t, q+1, r)

FUSION(t, p, q, r)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 2 | 4 | 7 | 1 | 3 | 2 | 6 |
| 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 |

q :

|  |  |  |  |
| --- | --- | --- | --- |
| 5 | 2 | 4 | 7 |
| 2 | 4 | 5 | 7 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 3 | 2 | 6 |
| 1 | 2 | 3 | 6 |

|  |  |
| --- | --- |
| 1 | 3 |

|  |  |
| --- | --- |
| 2 | 6 |

|  |  |
| --- | --- |
| 2 | 5 |

|  |  |
| --- | --- |
| 4 | 7 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  | 5 |  | 4 |  | 7 |  |  |  |  |  |  | 1 |  | 3 |  |  |  | 2 |  | 6 |  |

* Exemple 3

Application du tri par fusion sur l'instance (5, 2, 4, 7, 1, 3, 2, 6)

* Analyse ?

## Analyse des algorithmes diviser-pour-régner

* Algorithme avec appel récursif à lui-même
  + Temps d'exécution décrit par une récurrence
  + Décrit le temps d'exécution global pour un problème de taille n à partir du temps d'exécution pour des entrées de taille moindre
  + On peut alors es servir d'outils mathématiques pour résoudre la récurrence
* T(n)
  + Temps d'exécution d'un problème de taille n
* Si la taille du problème est suffisamment petite (
  + La solution directe prend un temps constant
* Si on divise le problème en a sous-problèmes
  + La taille de chacun étant 1/b de la taille du problème initial
  + Il faut pour résoudre a sous-problèmes
  + Il faut un temps D(n) pour diviser le problème en sous-problèmes
  + Il faut un temps C(n) pour construire la solution finale
* On a donc la récurrence :

T(n)= {

{ sinon

## Analyse du tri par FUSION

* Diviser
  + On calcule le milieu du sous-tableau
* Régner
  + On résout récursivement 2 sous-problèmes, chacun ayant la taille n/2
* Combiner
  + Utiliser la procédure fusion sur un sous-tableau à n éléments

T(n)= { si n=1

{ si n>1

## Arbre récursif

C: -temps requis pour résoudre des problèmes de taille 1

-temps par élément de tableau des étapes diviser et combiner

## 

## Arbre récursif

# 

# Croissance des fonctions

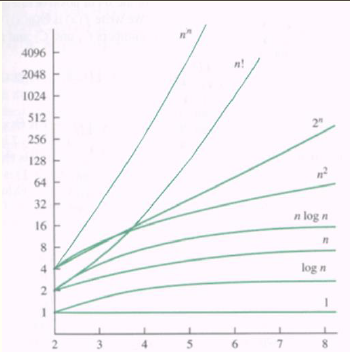
## Ordre de grandeur du temps d'exécution

* Caractérisation de l'efficacité de l'algorithme
* Comparatif des perfomances relatives de plusieurs algorithmes pour un même problème
  + Tri par insertion : dans le pire des cas
  + Tri par fusion : dans le pire des cas
* Performance asymptotique
  + Augmentation du temps d'exécution à la limite, quand la taille de l'entrée augmente indéfiniment

## Notation asymptotique

## 

## Ordre de grandeur de quelques fonctions parmi les plus connues



Cours 2 : Graphes notions de base et représentation

# Plan de la séance

* Notions de base sur les graphes
* Représentation des graphes en mémoire
* Bibliographie
  + Cormen, Leiserson, Rivest, Stein, "algorithmique" 3eme édition, Dunod, 2010

# Notions de base sur les graphes

## 

## Graphes ?

* Les graphes permettent de modéliser une multitude de problèmes
* Omniprésents en informatique
  + Réseaux
  + Systèmes distribués
  + Optimisation combinatoire
  + …
* Les algorithmes pour les manipuler sont fondamentaux

## Graphe non orienté

* Un graphe non orienté …
* … est défini par deux ensembles
  + Ensemble S des sommets
  + Ensemble A des arêtes
* Une arête, un élément a de A
  + Est défini par une paire de sommets distincts x et y de S
  + N'apparait pas plusieurs fois dans A
* On dit que
  + x et y sont incidents à a
  + x et y sont les extrémités de a
  + x et y sont adjacents

## 

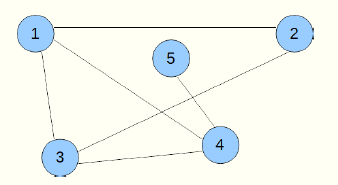
## Graphe orienté

* Un graphe orienté …
* … est défini par deux ensembles
  + Ensemble S des sommets
  + Ensemble A des arcs
* Une arc, un élément a de A
  + Est défini par un couple de sommets distincts x et y de S
  + N'apparait pas plusieurs dois dans A
  + Mais on peut avoir (x,y) et (y,x) qui sont deux arcs distincts
* On dit que
  + a admet x comme origine ou extrémité initiale
  + a admet y comme extrémité finale ou terminale

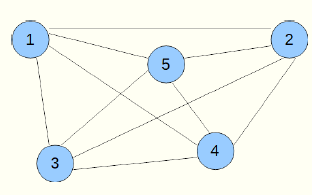
## 

* S={1, 2, 3, 4, 5}
* A={(1,2),(1,3),(1,4),(2,1),(3,2),(4,3),(4,5)}

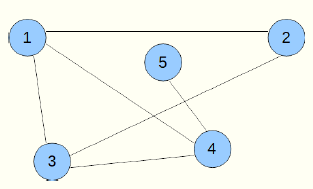
## Graphe fini

* le cardinal de S
  + est appelé ordre du graphe (nombre de sommets)
  + n=|S|
* le cardinal de A
  + est appelé taille du graphe (nombre d'arêtes ou arcs)
  + m=|A|
* ****les cardinaux de S et de A sont finis
* L'ordre du graphe n est 5
* La taille du graphe m est 6

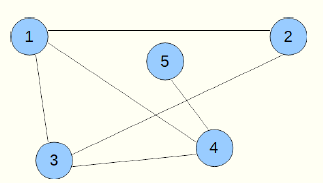
## Graphe complet (ou clique)

* Un graphe complet à n sommets…
  + Noté
* … est un graphe non orienté d'ordre n dont deux sommets quelconques sont adjacents
  + ****Il est donc de taille
* Graphe complet d'ordre 5
* La taille du graphe est de

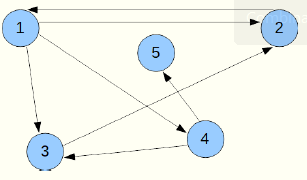
## Graphe partiel et sous-graphe

* Un graphe partiel de G=(S,A) est un graphe
  + Ayant le même ensemble de sommets S que G
  + Ayant pour ensemble d'arrêtes une partie de A
* Etant donnée une partie Y de S …
* … un sous-graphe F de G engendré par Y
  + Est un graphe ayant pour ensemble de sommets Y
  + Une arête (arc) de G donnant naissance à une arête (arc) de F si et seulement si les deux extrémités de cette arrête (arc) sont dans Y
* Autrement dit, sous-graphe F d'un graphe G est un graphe composé de certains sommets de G et de toutes les arêtes qui relient ces sommets dans G
* est un graphe partiel
* est un sous-graphe complet d'ordre 3

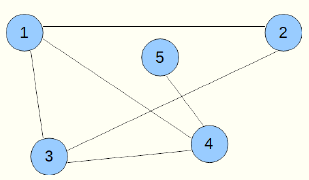
## Degré d'un sommet

* Etant donné un sommet x d'un graphe G non orienté
  + Le degré de x est le nombre d'arêtes incidentes à x
  + Les autres extrémités de ces arêtes constituent l'ensemble des voisins de x
* Autrement dit, le degré d'un sommet est le nombre d'arêtes dont ce sommet est une extrémité
* Dans un graphe orienté, on parle de degré entrant et sortant selon l'orientation des arcs
* Degré de 1 : 3, degré de 5 : 1
* 3, 4 et 2 sont les voisins de 1

## Prédécesseur et Successeur

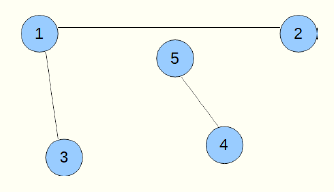
* y est un prédécesseur de x
  + Si l'arc (y,x) existe
* y est un successeur de x
  + si l'arc (x,y) existe
* Quand le prédécesseur de x est unique
  + Ce sommet est le père de x
  + x est un fils de ce sommet
* 1 est un prédécesseur de 3
* 5 est un successeur de 4
* 1 est le père de 4
* 4 est le fils de 1

## Chaîne et Cycle

* Soit G = (S, A) un graphe non orienté
* Une chaîne est une suite
  + s1a1s2a2…sk-1ak-1sk
  + avec
* Autrement dit, une chaîne est une liste ordonnée de sommets telle que chaque sommet de la liste soit adjacent au sommet suivant
* Un cycle est une chaine dont les deux extrémités coïncident et composé d'arêtes distinctes
* (2, 1, 3, 4, 5) est une chaîne de longueur 4
* (1, 2, 3, 4, 1) est un cycle de longueur 4

## Graphe connexe

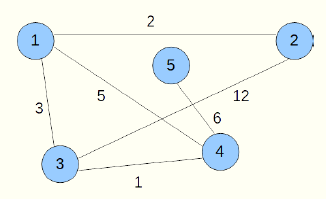
* Un graphe est connexe si, pour toute paire de sommets, il existe une chaîne les joignant
* Une composante connexe d'un graphe est un sous-graphe connexe maximal
  + On en peut y ajouter d'autres sommets en conservant la connexité du sous-graphe
* A noter d'un arbre est un graphe connexe et sans cycle



* Ce graphe n'est pas connexe
* Ce graphe possède 2 composantes connexes
* Les 2 sous-graphes sont des arbres

## Graphe pondéré

* Graphe dont les arêtes ou les arc sont munis d'une valuation
  + Coût ou poids ou longueur
* Le coût (ou poids, ou longueur) d'un graphe partiel ou d'une chaîne est alors la somme des coûts des arêtes ou des arc le constituant
* Une plus courte chaîne entre deux sommets est, parmi les chaînes qui la relient, une chaîne de poids minimum



* Le poids de la chaine (2, 1, 3, 4, 5) est 12
* C'est la plus courte chaîne reliant 2 et 5

## Représentation informatique

* On suppose les sommets de G numérotés par les entiers de 1 à |S|, chaque sommet étant repéré par son numéro
* Deux structures de données sont souvent utilisées pour représenter un graphe
  + Listes d'adjacences
  + Matrices d'adjacences

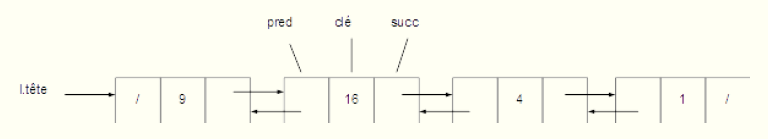
# Flashback

## Listes chaînées

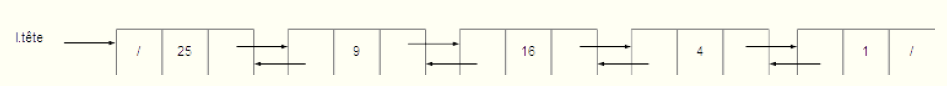
* Objets arrangés linéairement
  + Ordre déterminé par un pointeur dans chaque objet
* Liste doublement chaînée
  + Chaque élément/objet x comporte 3 attributs
    - Clé
    - Succ 🡪 pointeur sur le successeur de x dans la liste (NULL pour le dernier élément de la liste
    - Pred 🡪 pointeur sur le prédécesseur de x dans la liste (ou NULL pour le premier élément de la liste)
  + Tete 🡪 pointeur sur le premier élément (NULL si vide)

## Listes (doublement) chaînées

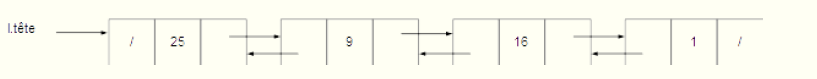
* Liste représentant l'ensemble {9, 16, 4, 1}



* Liste après insertion d'un élément dont la clé vaut 25



* Liste après suppression de l'élément dont la clé vaut 4



* RECHERCHER-LISTE(l, k)
  + O(n)
* INSERER-LISTE(l, x)
  + O(1)
* SUPPRIMER-LISTE(l, x)
  + O(1) si on a déjà un pointeur sur l'élément x à supprimer
  + O(n) si on supprime à partir de la clé k

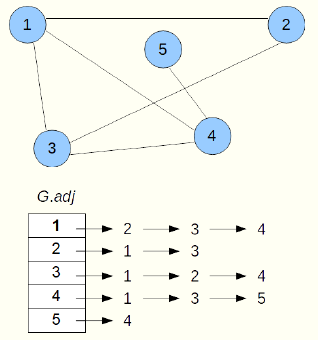
## Listes ou tableau ?



# Fin du flashback

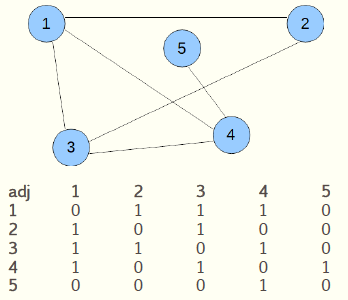
## Listes d'adjacences

* On suppose données (ex : lus dans un fichier)
  + L'ordre du graphe
  + L'ensemble des arêtes {x,y} du graphe
* On définit un tableau adj de |S| listes
  + Les |S| cases du tableau correspondent aux sommets 1, 2, … |S| du graphe
* Pour chaque
  + La liste adj[u] est une liste de sommets v
  + … tels qu'il existe un arc
* Autrement dit
  + Le pointeur en position u est la tête d'une liste chaînée qui contient les sommets adjacents au sommet u
  + Un voisin v de u doit appartenir à la liste correspondant au sommet u
* Adj 🡪 attribut du graphe
* Somme des longueurs de toutes les listes d'adjacences du graphe
  + Graphe orienté 🡪 |A|
  + Graphe non orienté 🡪 2|A|
* Quantité de mémoire requise
  + O(S+A)
* Graphes pondérés
  + On stocke le poids p(u,v) de l'arc (u,v) avec le sommet v dans la liste de u



## Matrice d'adjacences

* On suppose donnée (ex : lus dans un fichier)
  + L'ordre du graphe
  + L'ensemble des arêtes {x,y} du graphe
* On définit une matrice carrée adj à |S| lignes et |S| colonnes, telle que
  + Adj[i,j] = 1 si (i et j sont adjacents)
  + Adj[i,j] = 0 sinon
  + Adj[k,k] sera pris égal à 0 ou 1 selon le problème
* Quantité de mémoire requise
  + O(S²) (peu importe la densité du graphe)
* Graphe non orienté
  + Adj[i,j] = adj[j,i] = 1
* Graphe pondéré
  + On remplace le 1 par le poids p(u,v) de l'arc (u,v)



## Listes d'adjacences ou matrice d'adjacences ?

* Espace mémoire ?
  + On peut sauver de l'espace mémoire avec les listes d'adjacences si le graphe est peu dense
* Recherche d'un arc ?
  + Liste d'adjacences 🡪 parcours d'une liste 🡪 O(n)
  + Matrice d'adjacences 🡪 accès direct 🡪 O(1)
* Dépend de l'algorithme que l'on veut implémenter

Cours 3 : graphes, algorithmes élémentaires

* Parcours en largeur
* Parcours en profondeur

## Parcours de graphes ?

* Sert de base à plusieurs algorithmes
* Permet d'étudier les propriétés du graphe
  + Le graphe est-il connexe ?
  + Le graphe est-il biparti ?
* On pourra aussi faire des traitements sur les sommets et les arc/arêtes durant le parcours
* 2 types de parcours
  + Largeur
  + Profondeur

# Parcours de graphes en largeur

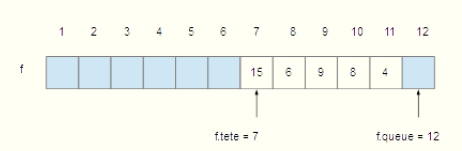
## Parcours en largeur

* Soit un graphe G =(S,A) et un sommet origine s
* Emprunte-les arêtes /arc de G pour découvrir touts les sommets accessibles depuis s
* Calcule la distance (plus petit nombre d'arcs)
  + Entre s et chaque sommet accessible
* Construit un arbre de parcours en largeur
  + De racine s
  + Qui découvre tous les sommets situés à une distance k avant les sommets de distance k+1
* Durant le parcours, chaque sommet devient successivement
  + Non découvert
  + Découvert
  + Découvert et tous ses sommets adjacents ont été découverts
* Utilise une file pour gérer la découverte des sommets

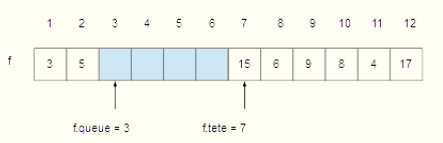
# Flashback (Files)

## Files

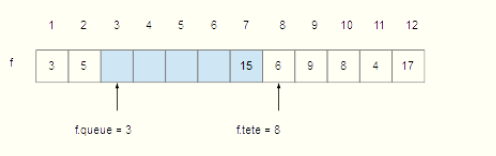
* Premier entré, premier sorti 🡪 FIFO
* Insérer 🡪 Enfiler
* Supprimer 🡪 Défiler
* Implémentation par tableau (circulaire)
  + Au plus n-1 éléments 🡪 tableau f[1…n]
  + Attribut f.queue qui indexe la queue
    - Enfiler : insertion en f[f.queue]
  + Attribut f.tete qui indexe la tête
    - Défiler : suppression en f[f.tete]
* File contenant 5 éléments



* Après enfilage des valeurs 17, 3 et 5



* Après défilage
  + Retourne la valeur 15
  + L'emplacement 7 n'est plus accessible

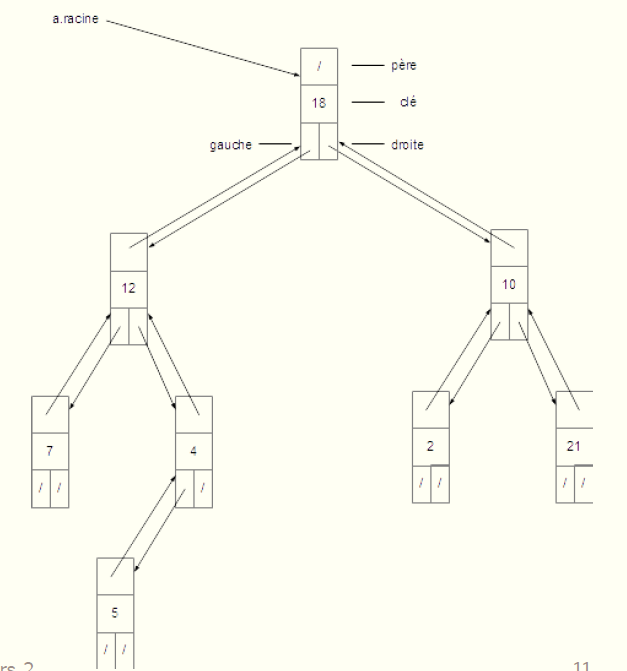


* ENFILER (f,x)
  + O(1)
* DEFILER (f)
  + O(1)
* RECHERCHER ?

# Flashback (arbres)

## Arbres (binaires)

* Représentation chaînée
  + Chaque nœud / objet contient les attributs
    - Clé
    - Père 🡪 pointeur vers le père (NUL pour la racine)
    - Gauche 🡪 pointeur vers le fils gauche
    - Droit 🡪 pointeur vers le fils droit
  + Racine 🡪 pointeur sur l'élément racine (NULL si arbre vide)
* Temps d'exécution ?
  + Dépend de l'ordre et de l'organisation des nœuds
  + Certaines propriétés doivent être respectées pour que les arbres puissent constituer un dictionnaire efficace
  + On verra ça un peu plus tard…



# Fin des flashbacks

## Parcours en largeur

* Exemple 1 : parcours en largeur

## Temps d'exécution du parcours en largeur

* Initialisation de S sommets
  + O(S)
* Chaque sommet est enfilé une fois et défilé une fois
  + O(S)
* Chaque liste d'adjacence est parcourue une fois (quand le sommet est défilé)
  + Somme des longueurs de toutes les listes 🡪 O(A)
* Total
  + O(S+A)

# Parcours de graphes en profondeur

## Parcours en profondeur

* On descend plus profondément dans le graphe chaque fois que c'est possible
  + On explore les arcs du sommet découvert le plus récemment
  + Si on trouve un sommet non découvert, on l'explore tout de suite même si on n'a pas exploré tous les autres arcs du sommet en cours
  + On revient en arrière plus tard pour explorer les arcs restant
* Construit une forêt de parcours en profondeur
* Le parcours en profondeur date chaque sommet
  + Date de début 🡪 découverte du sommet
  + Date de fin 🡪 toute la liste d'adjacence du sommet a été exéminée
* Exemple 2
  + Parcours en profondeur

## Temps d'exécution du parcours en profondeur

* Initialisation de S sommets
  + O(S)
* Chaque liste d'adjacence est parcourue une fois
  + Somme des longueurs de toutes les listes 🡪 O(A)
* Total
  + O(S+A)

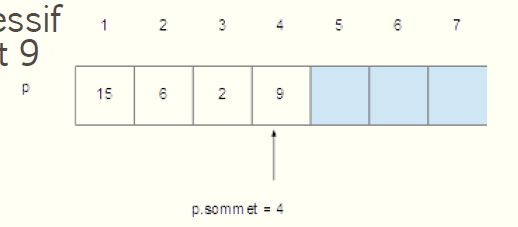
## Parcours en profondeur itératif

* Utilise une pile pour gérer la découverte des sommets

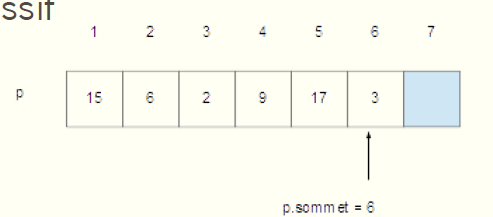
# Flashback (Piles)

## Piles

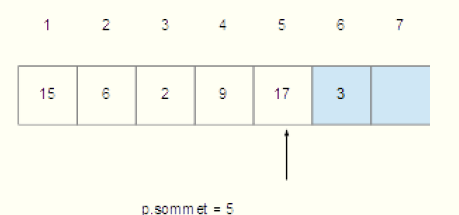
* Dernier entré, premier sorti 🡪 LIFO
* Insérer 🡪 Empiler
* Supprimer 🡪 Dépiler
* Implémentation par tableau
  + Au plus n éléments 🡪 tableau [1…n]
  + Possède un attribut p.sommet qui indexe l'élément le plus récemment inséré
  + P[1] 🡪 élément situé à la base de la pile
  + P[p.sommet] 🡪 élément situé au sommet de la pile
* Après empilage successif des valeurs 15, 6, 2, 9



* Après empilage successif des valeurs 17 et 3



* Après un dépilage
  + On récupère la valeur 3
  + La case 6 est ensuite non définie, on ne peut pas la réutiliser



* EMPILER (p,x)
  + O(1)
* DEPILER (p)
  + O(1)
* PILE-VIDE (p)
  + O(1)
* RECHERCHER ?

## Programmation ?

### Java

* Piles
  + Classe Stack (legacy)
* Piles et Files
  + Interface Deque (ArrayDeque, LinkedList)
  + addFirst(…), addLast(…), removeFirst(…), removeLast(…)
* Listes
  + Interface List (ArrayList, LinkedList)
* Tas/Files de priorité
  + Classe PriorityQueue

### C++

* Piles
  + Template stack (containers : vector, deque, list)
  + push(…) (push\_back), pop() (pop\_front), …
* Files
  + Template queue (containers : deque, list)
  + push(…) (push\_back), pop()(pop\_front), …
* Listes
  + Template list (container : doubly linked list)
* Tas/Files de priorité
  + Template priority\_queue (cont : vector, deque)

# Fin du flashback

## Parcours en profondeur itératif

* Exemple 3 : parcours en profondeur itératif en utilisant une pile -Version simple
  + La version itérative avec comportement identique à la version récursive est un peu plus complexe et sera vue en TD

Cours 4 : arbres couvrant de poids minimum

* Arbres couvrant de poids minimal
* Algorithme de Kruskal
  + Structure de données pour ensembles disjoints
* Algorithme de Prim
  + Structure de file de propriété

# Problème de l'arbre couvrant de poids minimal

## Définition du problème

## Applications

## Construction d'un arbre couvrant minimal

# Algorithme de kruskal

## Principe de l'algorithme

## Mise en œuvre

## Principe : gérer l'évolution des composantes connexes

## Gérer l'évolution des composantes connexes par tableau

## Remarques

# Structures de données pour ensembles disjoints

## Ensembles disjoints

## Opérations sur les ensembles disjoints

## Représentation d'ensemble disjoints par listes chaînées

## Représentation d'ensembles disjoints par arbres

## Algorithme de Kruskal avec utilisation d'ensembles disjoints

# Algorithme de Prim

## Principe de l'algorithme

## Remarques

# Flashback Tas

## Tas (binaire)

## Conservation de la propriété de tas

## Construction d'un tas

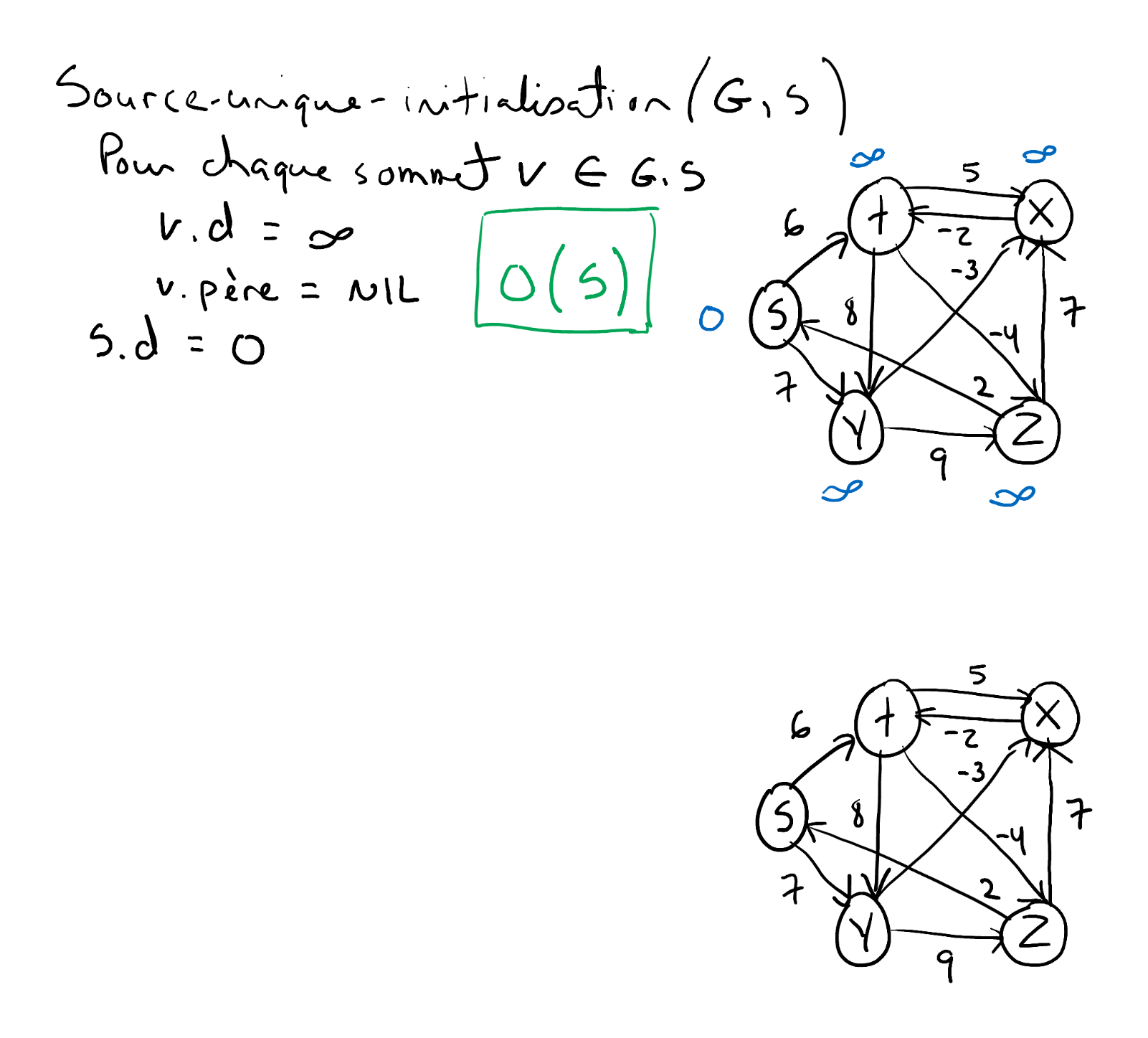
## Tri par tas

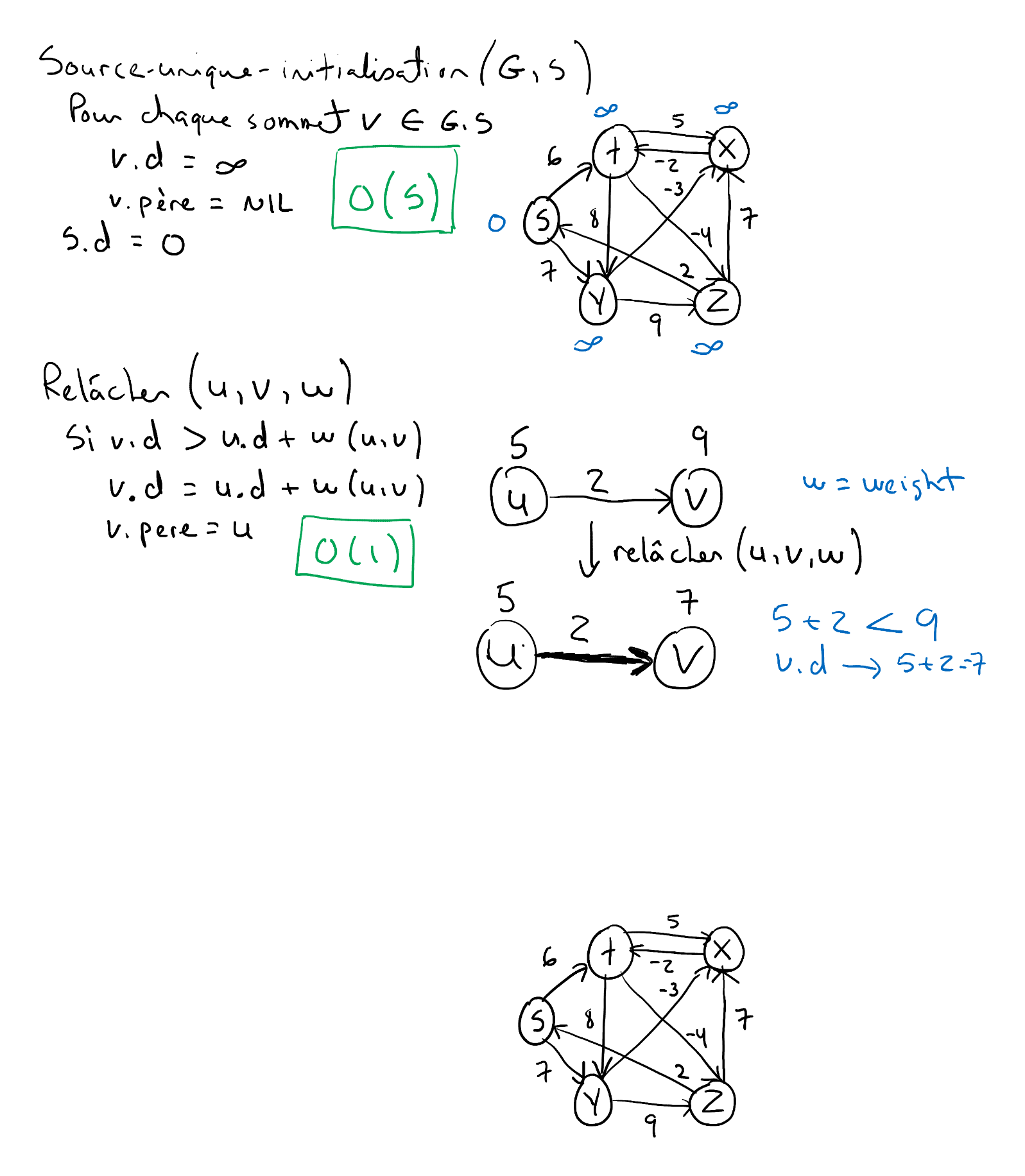
# Fin du flashback

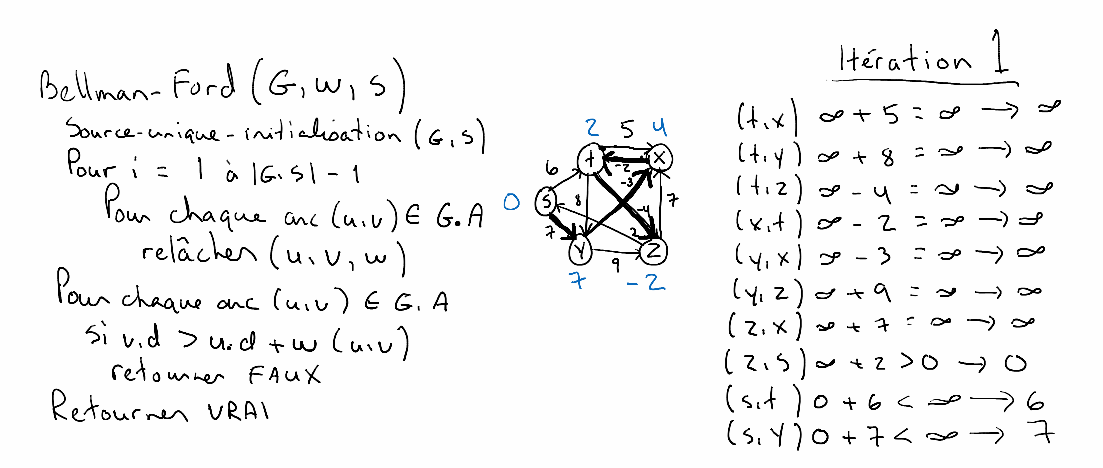
## Algorithme de Prim – implémentation par file priorités

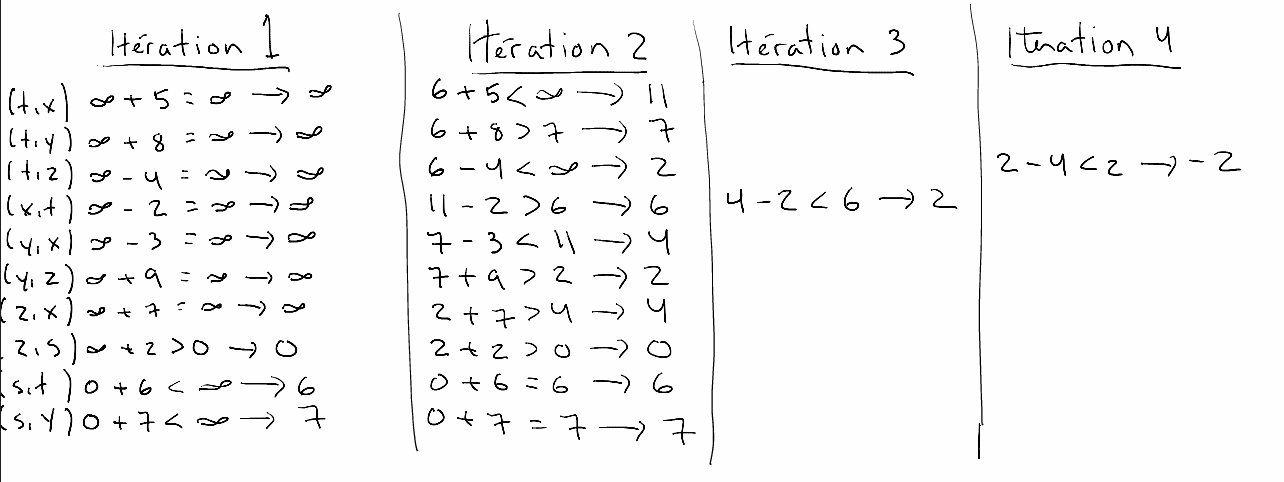
Cours 5 : plus de courts chemins

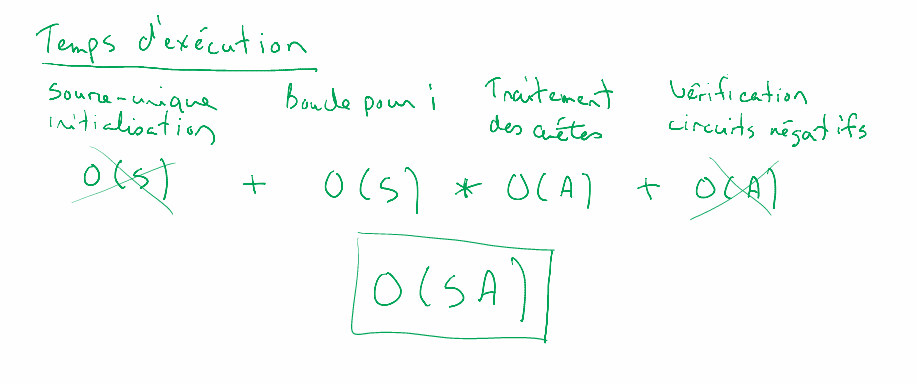
Cours 6 (13/24)

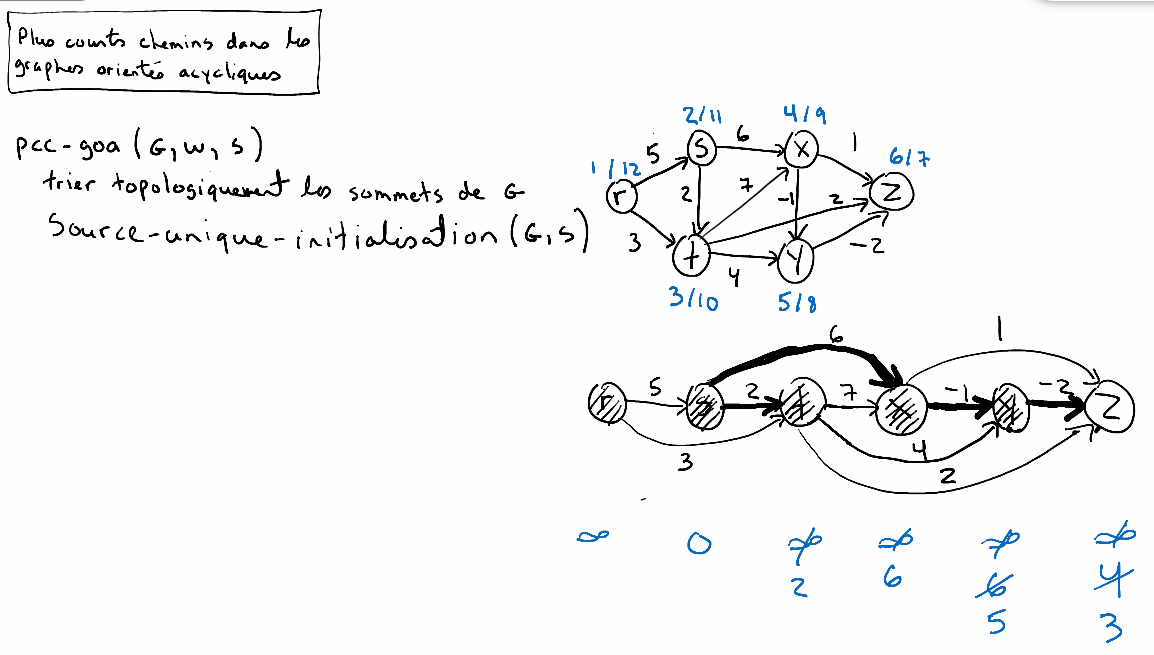












pcc\_goa(G,w,s)

    //tri\_topologique O(S+A)

    //+ source unique initialisation O(S)

    //+ double boucle pour O(A) \* relacher O(1)

    //-> O(S+A)

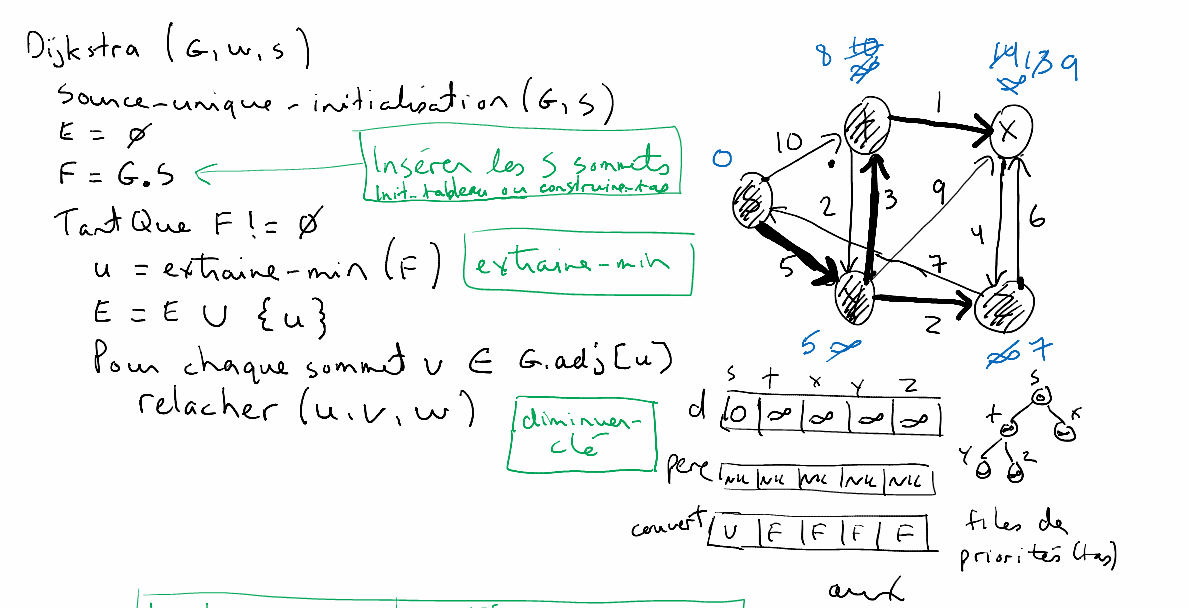
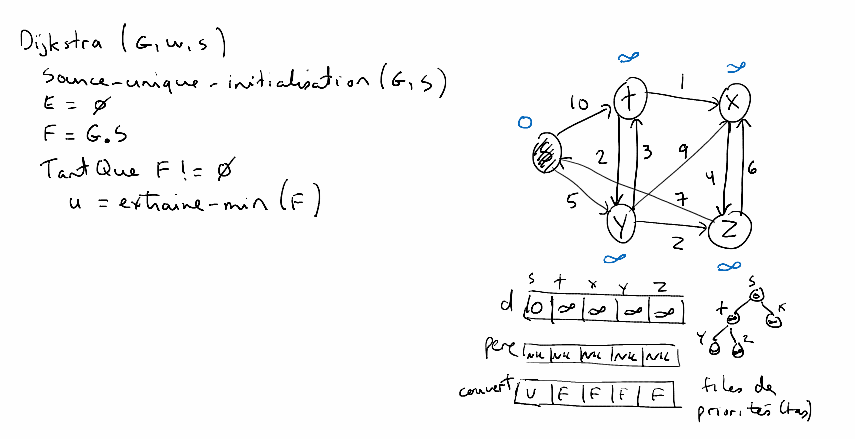
    trier topologiquement les sommets de G

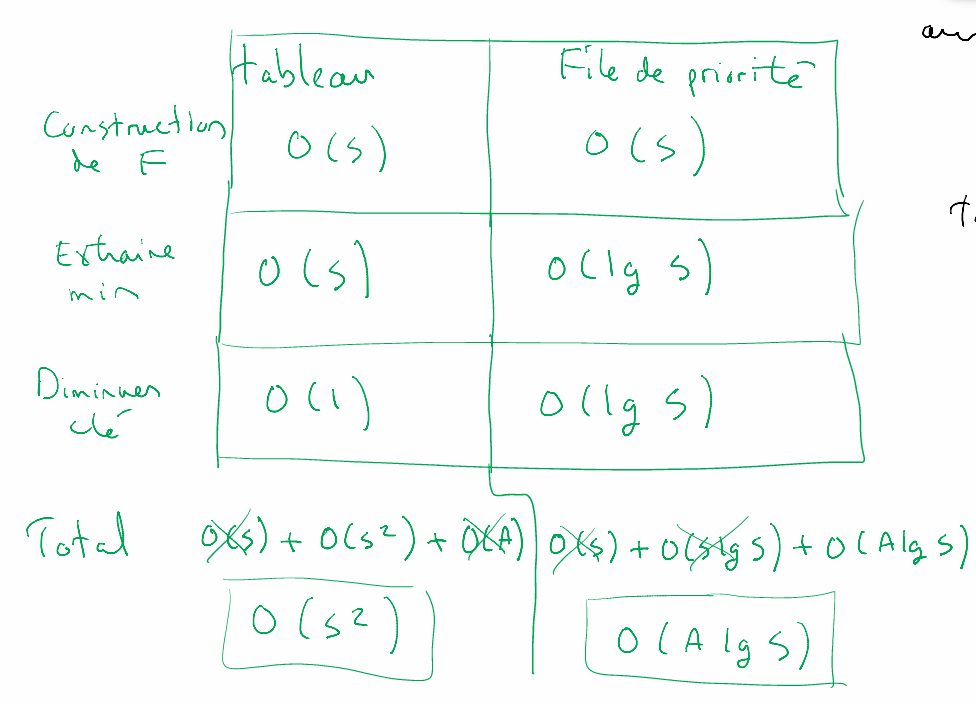
    source\_unique\_intialisation(G,s)

    pour chaque sommet u pris dans l ordre topologique

        pour chaque sommet v appartenant à G.adj[u]

            relacher(u,v,w)





Floyd-warshall(P)

    n = P.lignes

    d(0) = P