Algorithmie :

Méthode permettant de résoudre un problème de manière systématique

Exemple : recette de cuisine, instructions d'un GPS…

Plusieurs étapes :

Etape 1 : poser un problème :

- 2 types de problèmes:
  - Instance de problème générale = 1 problème précis avec une seule solution
  - Classe de problème = 1 problème général sans valeur
- Spécifier le problème:
  - Paramètre d'entrée :
    - Caractériser une instance du problème
    - Variables de l'énoncé
- Pré-condition
  - Condition à respecter pour que le problème aie un sens
- Paramètres de sortie
  - Caractérise la solution à une instance du problème
  - Éléments de réponse
- Post-condition
  - Condition à respecter par les paramètre d'entrée et de sortie

Exemple : Calculer la factorielle du nombre réel positif p

→ Entrée a : un nombre

→ Pré-condition : a est un nombre réel positif

→ Sortie b : la factorielle de a

→ Post-condition : b = a!

・Pas d'initiative par l'exécutant (parfait pour un ordi)

・Éviter les ambiguïtés

・Utiliser un langage clair

Etape 2 : formalisation du problème :

Etapes

Etape 1 : Démarrer

Etape 2 : Lire nombre

Etape 3 : Mettre fact à 1 et i à 1

Etape 4 : Vérifier que i <= nombre. Si faux aller à l'étape 7

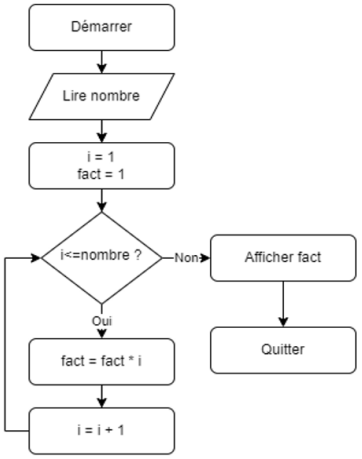
Etape 5 : fact = fact*i

Etape 6 : i = i+1 et retourner à l'étape 4

Etape 7 : Afficher fact

Etape 8 : Quitter

Schéma



Les structures :

- Structures de contrôle
  - Séquences
  - Conditionnelles (gère le flux)
  - Boucles
- Structures de données
  - Constantes
  - Variables
  - Tableaux
  - Structures récursives (listes, arbres, graphes…)

Notion importante :

- La terminaison
  - S'assurer que l'algo se terminera en un temps fini
- La correction
  - S'assurer que le résultat fourni par l'algorithme est bien une solution au problème et qu'elle est cohérente
- La complétude
  - S'assurer que pour une classe de problème, l'algorithme donnera bien l'ensemble des solutions correspondantes

La récursivité :

Démarche qui fait référence à l'objet même de la démarche pendant son processus

Exemples : Le calcul d'une factorielle, la suite de fibonacci, les tours de Hanoï

Problème des lapins:

« Quelqu'un a déposé un couple de lapins dans un certain lieu, clos de toutes parts, pour savoir combien de couples seraient issus de cette paire en une année, car il est dans leur nature de générer un autre couple en un seul mois, et qu'ils enfantent dans le second mois après leur naissance. »

Formulation de la classe de problème

Un couple de lapins génère un nouveau couple de lapin chaque mois à partir de leur 2ème mois d'existence, après **x** mois combien aurais-je de couple de lapins ?

Spécification de la classe de problème

- Entrée a : nombre de mois
- Pré-condition : a réel positif
- Sortie m : nombre de couples de lapins
- Post-condition : ?

Suite de Fibonacci

F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	...	Fn
0	1	1	2	3	5	8	13	21	34	55	...	Fn-1 + Fn-2

F9 = F7 + F8 ↔ 34 = 21 + 13

Cas général :

Un élément est égal à la somme des deux éléments qui le précèdent

Cas de base :

L'élément 0 vaut 0 et l'élément 1 vaut 1

En python :

```python
def fibo(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

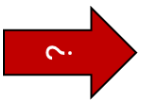
Les preuves :

- 1 - Preuve d'arrêt
    - Bien fondé
    - Appel avec des paramètres de valeurs inférieurs
- 2 - Preuve de validité
    - Correction partielle
    - Démontrer que si l'algorithme fonctionne pour n-1, alors il fonctionne pour n

Dérécursivation :

Passer d'un algorithme récursif à itératif

```python
def fibo(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```



```python
def fiboI(n):
    a, b = 0, 1
    for i in range(0, n):
        a, b = b, a+b
    return a
```

Pourquoi la récursivité ?

Avantages

- Simple à comprendre
- Simple à lire

Inconvénients

- Utilisation accrue de la mémoire
- Utilisation accrue du CPU

Les structure de données

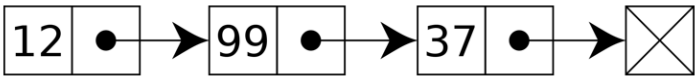
- Une infinité de classe de problèmes
- Représenter les données efficacement
- Il existe différentes structures de données

Tableaux

Index	0	1	2	3	4	5
Valeur	12	37	99	128	54	93

- Accès par index : tab[0] = 12
- Données contiguës (index qui se suivent)
- Ajout/suppression impossible sans recréer le tableau

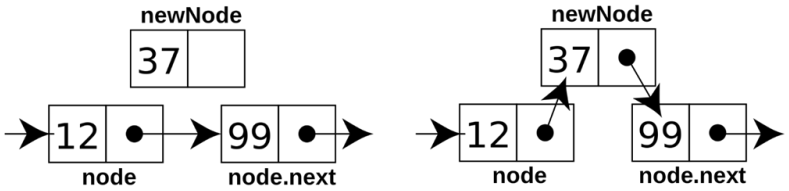
Listes chaînées



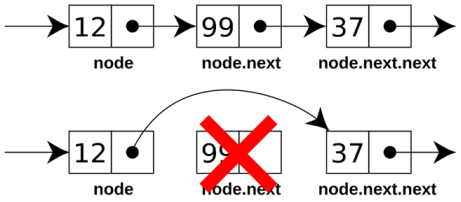
Chaque noeud contient :

1. L'élément
2. Un lien vers le noeud suivant

Ajout d'un noeud



Suppression d'un noeud



Généralisation :

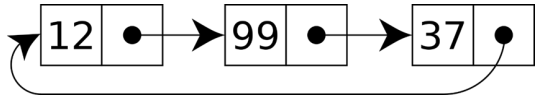
- Entrée L => liste sur laquelle on travaille
- Entrée c => élément courant
- Entrée a => élément qu'on souhaite ajouter
- Précondition => C doit appartenir à L
- Sortie L => Liste contenant le nouvel élément
- Postcondition => c.next est bien égal à A

Formalisation :

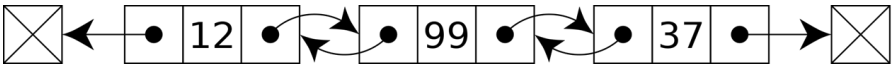
- Etape 1 : Démarrer
- Etape 2 : Récupérer L et c
- Etape 3 : Lire a
- Etape 4 : a.next = c.next
- Etape 5 : c.next = a
- Etape 6 : Stop

Autre cas

- Les listes chaînées circulaires



- Les listes doublement chaînées



Définition de la structure

1. Node
2. First Node
3. (Last Node)

```
class Node:
    data = None
    next = None


class LinkedList:
    firstNode = None
```

Avantages

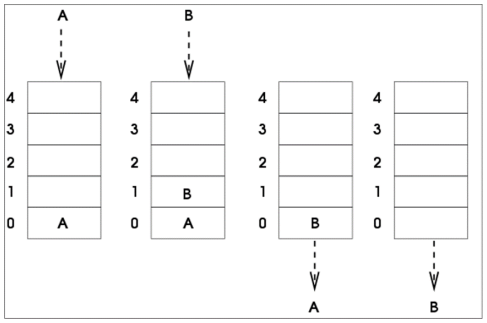
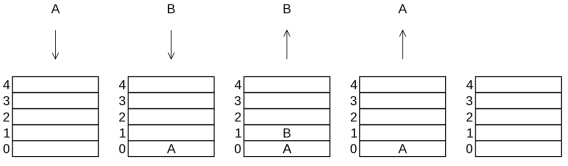
- Données non contiguës (espace de stockage moindre)
- Ajout / suppression après un élément au début très simple

Inconvénients

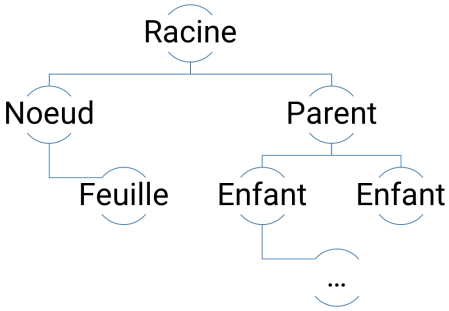
- Pas d'accès aléatoire ou d'indexation
- Ajout / suppression avant un élément très complexe
- Ajout d'une liste à la suite d'une autre complexe (exception)

Piles et files

Les piles	Les files

• Principe de LIFO (last in - first out)
• Primitives
  ○ Empiler
  ○ Dépiler
  ○ Vide ?
  ○ Nombre d'éléments
 
	• Principe de FIFO (first in, first out)
	• Primitives
	  ○ Enfiler
	  ○ Défiler
	  ○ Vide ?
	  ○ Nombre d'éléments
	 

Les arbres



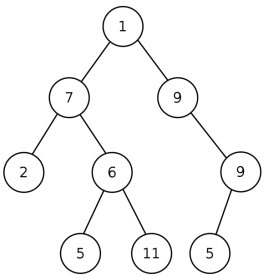
- **Racine (Root) :** il s'agit du nœud initial, il se situe tout au-dessus de l'arbre.
- **Nœud (node) :** C'est un élément d'un arbre qui comprend une valeur ainsi qu'une référence vers d'autres nœuds.
- **Feuille (Leaf) :** Élément en bout d'arbre, il s'agit d'un nœud qui n'a pas de référence vers d'autres nœuds.
- **Enfant (Child)** : Un nœud enfant est un nœud qui a été référencé dans un nœud dit parent.
- **Parent** : un nœud parent est un nœud qui fait référence à des nœuds dits enfants.
- **Frères (Siblings)** : Des nœuds frères sont des nœuds qui ont le même parent

Informations

- Représentation hiérarchique des données
- Utilisation régulière
- Implémentation récursive
- Définition ne permettant de parcourir que de la racine vers les feuilles
- Faire référence au parent dans la définition de Tree

```
class Tree:
    data = None
    children = LinkedList()
    parent = None
```

Les arbres binaires

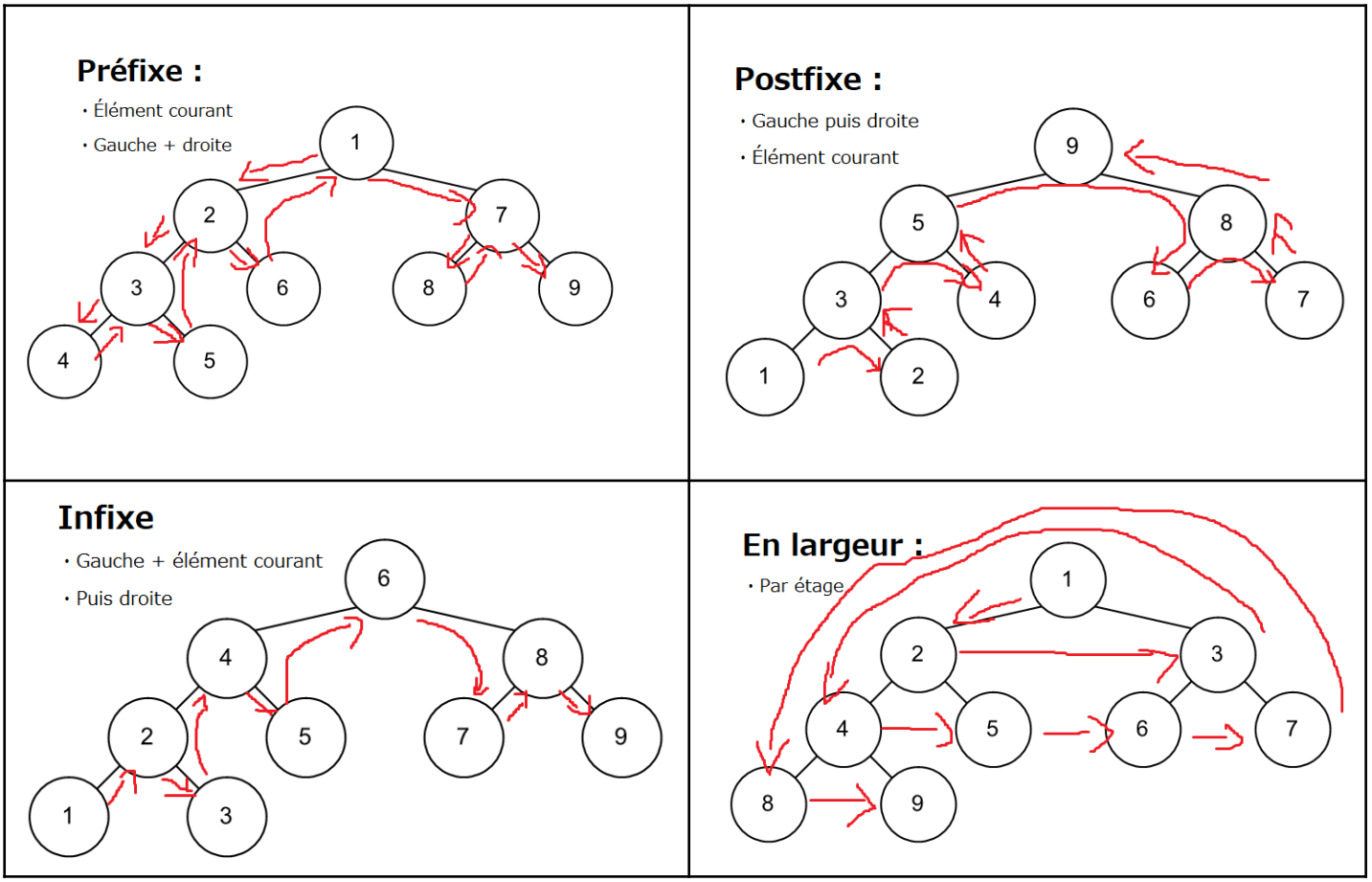


Définition

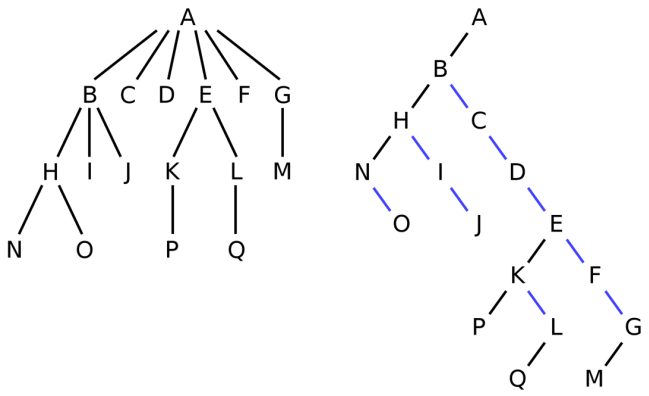
- **Profondeur** : Distance entre le noeud et la racine, aussi appelé niveau
- **Hauteur** : profondeur maximale d'un noeud

Différentes méthodes de parcours

- En profondeur :
    1. Préfixe
    2. Postfixe
    3. Infixe
- En largeur



Transformer un arbre n-aire en arbre binaire



Structures en vrac

- Tas
- Table de hachage:
    - Valeur unique caché sans retour en arrière différent de chiffrement avec retour en arrière
- Arbre binaire de recherche
- Ensemble → sans ordre particulier, sans répétition.

Les algorithmes de tri

A quoi ça sert ?

Ordonne des informations pour mieux les traiter (+ facile de rechercher des éléments dans des données triées)

2 sortes de tri

- Tri non en place (peut devenir tri en place)
- Tri en place (ne peut pas devenir non en place)

Complexité

La différence est dite de complexité :

- Permet de mesurer la performance
- Complexité **temporelle** -> quantifier la vitesse d'exécution
    - Compter le nombre d'opérations élémentaires
    - Taille des données, notée ***n***
    - La donnée en question
        - Calcul dans le meilleur des cas
        - Calcul dans le pire des cas
        - (Calcul dans le cas moyen)
- Complexité **spatiale** -> quantifier l'utilisation de la mémoire

Tri à bulles

8 5 3 1 4 7 9

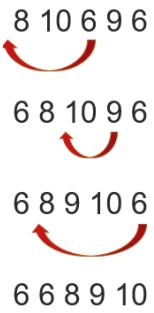
- Parcourt la liste
- A chaque élément, compare avec le suivant et échange leur position

8	**5**	**3**	**1**	**4**	**7**	**9**
5	**8**	**3**	1	4	7	9
5	3	**8**	**1**	4	7	9
5	3	1	**8**	**4**	7	9
5	3	1	4	**8**	**7**	9
5	**3**	1	4	7	8	9
3	**5**	**1**	4	7	8	9
etc etc jusqu'à arriver à 1 3 4 5 7 8 9 (trié)						

Complexité temporelle

Meilleur	O(n)
Pire	O(n²)
Moyenne	O(n²)
Complexité spatiale	**O(1)**
Stabilité	**Oui**

Tri par insertion



8 10 6 9 6

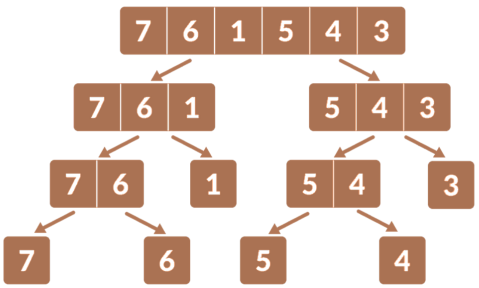
- Parcourt la liste
- Compare à l'élément précédent
- Déplace l'élément le + grand pour "faire de la place"

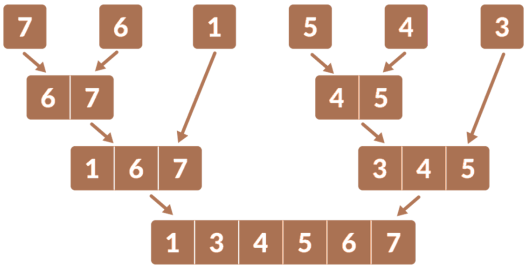
Complexité temporelle

Meilleur	O(n)
Pire	O(n²)
Moyenne	O(n²)
Complexité spatiale	**O(1)**
Stabilité	**Oui**

Tri par fusion

- Concept « Divide and Conquer »
    1. Diviser
    2. Régner
    3. Combiner
- Méthode récursive





Algorithme :

- Si la liste n'a qu'1 élément, elle est triée
- Séparer liste en 2 listes + - égales

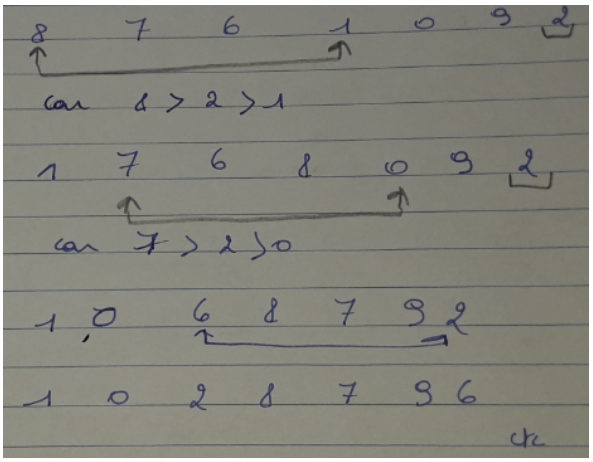
- Trier chacune des listes selon le tri par fusion
- Fusionner les 2 listes en 1 seule liste triée

Complexité temporelle

Meilleur	O(n log(n))
Pire	O(n log(n))
Moyenne	O(n log(n))
Complexité spatiale	**O(n)**
Stabilité	**Oui**

Tri rapide

- Principe similaire au tri par fusion
- Divide and Conquer
- Une des méthodes les plus utilisées
- Utilisation d'un pivot pour séparer la liste en 2 sous listes
- Différents choix possibles du pivot:
    - **Premier** élément
    - **Dernier** élément
    - Élément **aléatoire**
    - Élément **médian**



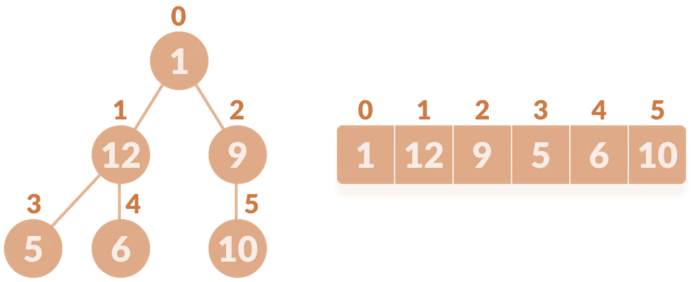
Complexité temporelle

Meilleur	O(n log(n))
Pire	O(n²)
Moyenne	O(n log(n))
Complexité spatiale	**O(log n)**
Stabilité	**Non**

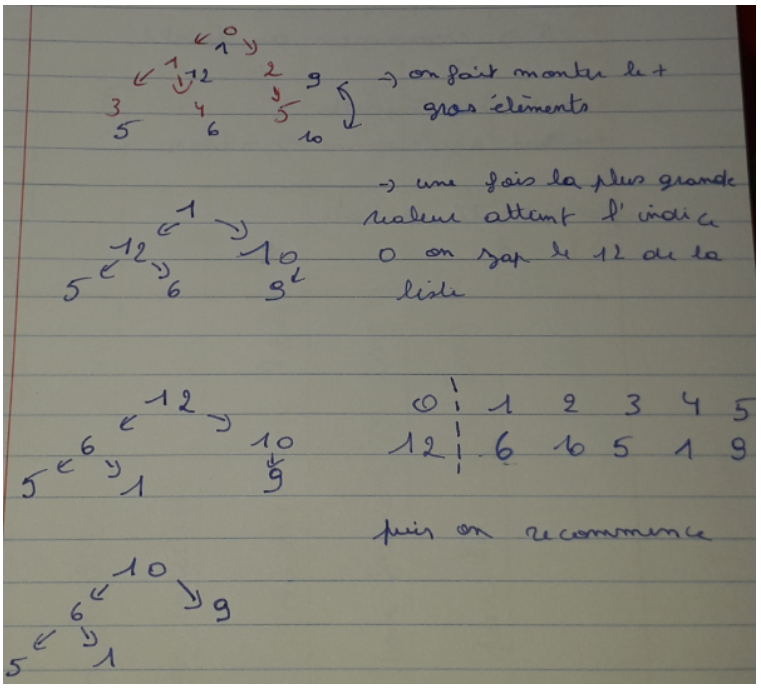
Tri par tas

- Utilise les listes et les arbres

- Se base sur le fonctionnement de la structure de données du tas
- Associe un arbre binaire et une liste



Comment ça fonctionne ?



Algorithme

1. S'assurer qu'on respecte la propriété « Max-Heap »
2. Echanger la racine avec le dernier élément – Swap
3. Réduire la taille du tas de 1 – Remove
4. Repositionner l'élément le plus grand à la racine – Heapify
5. Recommencer au point 2

Complexité temporelle

Meilleur	O(n log(n))
Pire	O(n log(n))
Moyenne	O(n log(n))
Complexité spatiale	**O(1)**
Stabilité	**Non**

Récapitulatif

Nom	Meilleur cas	Pire cas	Cas moyen	Mémoire
Tri à bulles	n	n^2	n^2	1
Tri par insertion	n	n^2	n^2	1
Tri rapide	n logn	n^2	n logn	1
Tri par fusion	n logn	n logn	n logn	n
Tri par tas	n logn	n logn	n logn	1

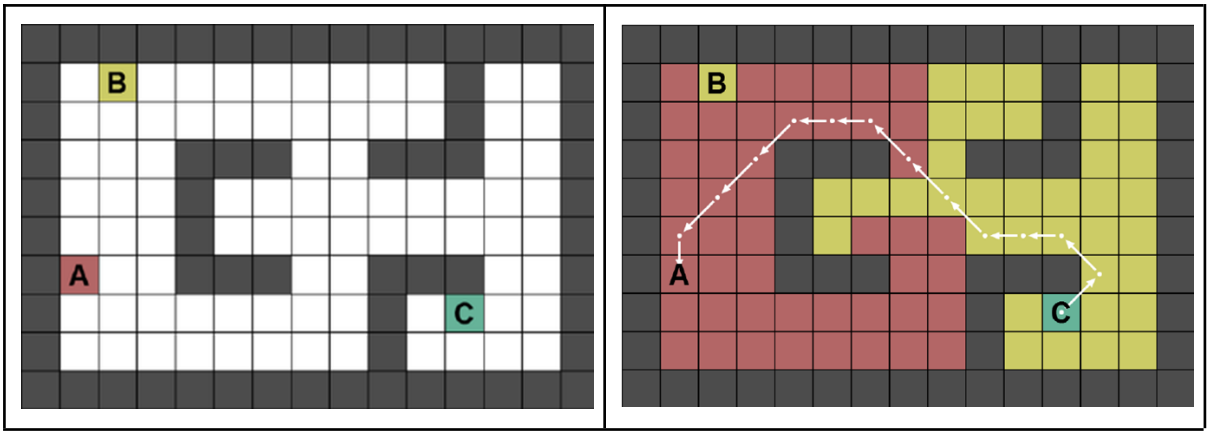
Applications

- Tri à bulle
    - Code simple et facile
    - Complexité peu / pas importante
- Tri par insertion
    - Peu d'éléments dans la liste
    - Peu d'éléments restants à trier
- Tri par fusion
    - Compter le nombre d'inversion restantes
    - Tri externe
- Tri rapide
    - Complexité temporelle et spatiales importantes
- Tri par tas
    - Systèmes embarqués et sécurisés (Kernel Linux)

La théorie des graphes

Pathfinding

- Trouver le chemin le plus court entre un point A et un point B
- Différents critères :
    - La distance
    - Le coût
    - La vitesse



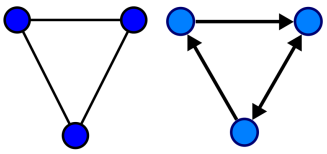
- Approche intuitive
    1) Prendre l'élément de fin et définir ses coordonnées ainsi que l'initialisation d'un compteur ici A = (1, 6, 0)
    2) Ajouter cet élément à une liste sous forme FIFO
    3) Parcourir la liste, en incluant les futurs éléments rajoutés et réaliser les opérations suivants :
        a) Créer une liste des 4 cases adjacentes en augmentant le compteur
        b) Si la case est un mur ou si elle existe déjà dans la liste principale, la retirer
        c) Ajouter toutes les cases restantes à la fin de la liste principale
    4) Réaliser a, b et c jusqu'à tomber sur l'élément de début
    5) Démarrer de cet élément et prendre l'élément adjacent avec le compteur le plus bas
    6) On définit ainsi le chemin le plus court

Les graphes

Un graphe est un couple G = (V,E) où

- V est un ensemble de sommets (noeuds, points, vertex)
- E est un ensemble d'arêtes (liens, lignes) qui sont des paires de sommets

Quantité non négligeable de types de graphes



- (Non-) connexe
- (Non-) orientés
- etc…

Structure de données abstraite

- Opérations de base
- Adjacents(G, x, y)
- Voisins(G, x)
- Ajouter_Sommet(G, x)
- Supprimer_Sommet(G, x)
- Ajouter_Arete(G, x, y)
- Supprimer_Arete(G, x, y)

- Retourner_Valeur(G, x)
- Fixer_Valeur(G, x, v)

Représentation des graphes

Listes d'adjacence

- Chaque sommet est un objet qui comprend une liste des sommets adjacents

Matrice d'adjacence

- Matrice carrée où les lignes représentent les sommets de départ et les colonnes les sommets d'arrivée

Matrice d'incidence

- Matrice où les lignes représentent les sommets et les colonnes les arêtes

	Liste d'adjacence	Matrice d'adjacence	Matrice d'incidence
Créer le graphe	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Ajouter un sommet	$O(1)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Ajouter une arête	$O(1)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Supprimer un sommet	$O(\|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Supprimer une arête	$O(\|V\|)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Test d'adjacence entre deux sommets	$O(\|V\|)$	$O(1)$	$O(\|E\|)$
Remarques	Lent dans la suppression parce qu'il faut trouver les sommets ou arêtes	Lent dans l'adjonction ou suppression de sommets parce que la matrice doit être reformatée	Lent dans l'adjonction ou suppression de sommets ou d'arêtes parce que la matrice doit être reformatée

Parcours

- Parcours en largeur
  - Utilisation d'une file pour les noeuds voisins
- Parcours en profondeur
  - Similaire aux labyrinthes
  - Marquages des sommets visités

Types d'algorithmes

- Shortest path : Trouver le chemin le plus court
- Spanning Tree : Trouver l'arbre couvrant de poids minimal
- Min-cost flow : Trouver la manière la plus économe d'utiliser un réseau de transport

Algorithme de Dijkstra

- Graphe pondéré (non-)orienté
- Trouver un chemin entre 2 sommets avec le poids minimum
- Construction d'un sous-graphe :
  - Distance de chaque sommet avec celui de départ = ∞

    - Choix du sommet à distance minimale hors du sous-graphe
    - Mise-à-jour des distances des sommets voisins
    - Plus de sommets / Sélection du sommet d'arrivée

Autres algorithmes

- Bellman-Ford
    - Autorise des poids négatifs
    - Permet de déterminer des circuits absorbants
- Floyd-Warshall
    - Distance la plus courte entre toutes les paires de sommets
    - Nécessite une représentation en matrice d'adjacence
- Viterbi, Johnson, A*
- Algorithme de Kruskal
- Algorithme de Prim
- Algorithme de Borůvka
- Algorithme de Busacker et Gowen
- Algorithme de Ford-Fulkerson
    - Variante du premier
- Algorithme d'Edmonds-Karp
    - Cherche le flot maximum
- Algorithme de Goldberg-Tarjan
    - « Poussage/réétiquetage »

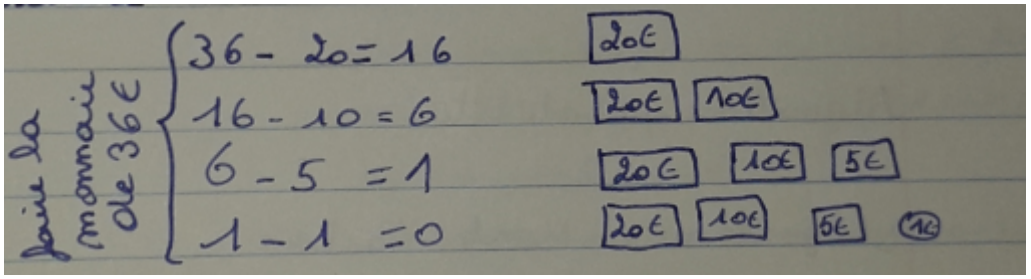
Méthodes algorithmiques

Méthodes rencontrées

- Récursivité
- Diviser pour régner
- La recherche exhaustive (essayer toutes les possibilités)

Algorithme glouton

- Trouver l'optimum local, étape par étape, jusqu'à obtenir un résultat optimum global
- Exemples : Kruskal et Prim



Heuristique

- Dans l'algorithme A* (astar), on utilise la méthode heuristique
- Méthode de calcul qui fournit rapidement une solution réalisable, mais pas forcément optimale (voire exacte)
- Utilisation :
    - Théorie des graphes
    - IA
    - Programmation de jeux

Plusieurs critères d'évaluation

1. Qualité du résultat
    - ➢ Comparaison avec résultat optimal connu
2. Coût de l'heuristique
    - ➢ Complexité de l'heuristique
3. Etendue du domaine d'application

Algorithmes probabilistes

- Algorithme de Monte-Carlo
    - Temps : déterministe
    - Résultat : probabilité minime d'incorrection
- Algorithme de Las Vegas
    - Temps : aléatoire
    - Résultat : correct
- Algorithme d'Atlantic City

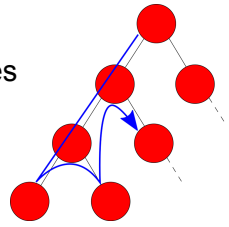
Résolution Sudoku

- Essayer toutes les configurations
- Bit Masks
- Backtracking

5	3				7			
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Backtracking

- Méthode qui répond au problème de satisfaction de contraintes
- Construire une solution de manière récursive, étape par étape, en retirant les solutions qui ne répondent pas à la contrainte
- ≡ Parcours en profondeur de l'arbre de décision d'un problème



Problèmes résolvables par Backtracking

- Problème des 8 dames
- Problème de mots-croisés

- Problème du solitaire
- Problème du sac à dos
- …