Structures de données et algorithmes

Pierre Geurts

Dernière mise à jour le 6/02/2019

E-mail	:	p.geurts@ulg.ac.be
URL	:	http://www.montefiore.ulg.ac.be/~geurts/sda.html
Bureau	:	R 134 (Montefiore)
Téléphone	:	04.366.48.15

Contact

- Professeur :
  - Pierre Geurts, p.geurts@ulg.ac.be, I.134 Montefiore, 04/3664815
- Assistant :
  - Jean-Michel Begon, jm.begon@ulg.ac.be, I.127 Montefiore, 04/3662972
  - Romain Mormont, r.mormont@ulg.ac.be, I.127 Montefiore, 04/3662880
- Sites web du cours :
  - Cours théorique : http://www.montefiore.ulg.ac.be/~geurts/sda.html
  - Répétitions et projets : http://www.montefiore.ulg.ac.be/~jmbegon/?sda2018_2019

Objectif du cours

- Introduction à l'étude systématique des algorithmes et des structures de données
- Vous fournir une boîte à outils contenant :
  - Des structures de données permettant d'organiser et d'accéder efficacement aux données
  - Les algorithmes les plus populaires
  - Des méthodes génériques pour la modélisation, l'analyse et la résolution de problèmes algorithmiques
- On insistera sur la généralité des algorithmes et structures de données et on les étudiera de manière formelle
- Les projets visent à vous familiariser à la résolution de problèmes

Organisation du cours

- Cours théoriques :
  - Les vendredis de 13h30 à 15h30, Amphi 02, Bâtiment B37 (Math)
  - 12 cours
- Répétitions :
  - Certains vendredis de 15h30 à 17h30, Amphi 02, Bâtiment B37 (Math)
  - Exercices portant sur la matière théorique + debriefing des projets
- Projets (sujet à modifications) :
  - Trois projets tout au long de l'année, de difficulté croissante
  - Les deux premiers individuels, le troisième en binôme
  - En C

- Evaluation sur base des projets (30%) et d'un examen écrit (70%).

Notes de cours

- Transparents disponibles sur la page web du cours.
- Pas de livre de référence obligatoire mais le cours se base fortement sur l'ouvrage suivant :
  - **Introduction to algorithms, Cormen, Leiserson, Rivest, Stein, MIT press, Third edition, 2009.**
    - `http://mitpress.mit.edu/algorithms/`
- Autres références :
  - Algorithms, Sedgewick and Wayne, Addison Wesley, Fourth edition, 2011.
    - `http://algs4.cs.princeton.edu/home/`
  - Data structures and algorithms in Java, Goodrich and Tamassia, Fifth edition, 2010.
    - `http://ww0.java4.datastructures.net/`
  - Algorithms, Dasgupta, Papadimitriou, and Vazirani, McGraw-Hill, 2006.
    - `http://cseweb.ucsd.edu/users/dasgupta/book/index.html`
    - `http://www.cs.berkeley.edu/~vazirani/algorithms/all.pdf`

Cours sur le web

Ce cours s'inspire également de plusieurs cours disponibles sur le web :

- Antonio Carzaniga, Faculty of Informatics, University of Lugano
  - `http://www.inf.usi.ch/carzaniga/edu/algo/index.html`
- Marc Gaetano, Polytechnique, Nice-Sophia Antipolis
  - `http://users.polytech.unice.fr/~gaetano/asd/`
- Robert Sedgewick, Princeton University
  - `http://www.cs.princeton.edu/courses/archive/spr10/cos226/lectures.html`
- Charles Leiserson and Erik Demaine, MIT.
  - `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/index.htm`
- Le cours de 2009-2010 de Bernard Boigelot

Contenu du cours

Partie 1: Introduction
Partie 2: Outils d'analyse
Partie 3: Algorithmes de tri
Partie 4: Structures de données
Partie 5: Dictionnaires
Partie 6: Résolution de problèmes
Partie 7: Graphes

Partie 1

Introduction

Plan

1. Algorithms + Data structures = Programs (Niklaus Wirth)

2. Introduction à la récursivité

Algorithmes

- Un **algorithme** est une suite *finie* et *non-ambiguë* d'opérations ou d'instructions permettant de résoudre un *problème*
- Provient du nom du mathématicien persan *Al-Khawarizmi* ($\pm$820), le père de l'algèbre
- Un problème algorithmique est souvent formulé comme la transformation d'un ensemble de valeurs, **d'entrée**, en un nouvel ensemble de valeurs, **de sortie**.
- Exemples d'algorithmes :
  - Une recette de cuisine (ingrédients $\rightarrow$ plat préparé)
  - La recherche dans un dictionnaire (mot $\rightarrow$ définition)
  - La division entière (deux entiers $\rightarrow$ leur quotient)
  - Le tri d'une séquence (séquence $\rightarrow$ séquence ordonnée)

Algorithmes

- On étudiera essentiellement les algorithmes **corrects**.
  - Un algorithme est (totalement) *correct* lorsque pour chaque instance, il se termine en produisant la bonne sortie.
  - Il existe également des algorithmes *partiellement corrects* dont la terminaison n'est pas assurée mais qui fournissent la bonne sortie lorsqu'ils se terminent.
  - Il existe également des algorithmes *approximatifs* qui fournissent une sortie inexacte mais néanmoins proche de l'optimum.

- Les algorithmes seront évalués en termes d'*utilisation de ressources*, essentiellement par rapport aux **temps de calcul** mais aussi à l'utilisation de la **mémoire**.

Algorithmes

Un algorithme peut être spécifié de différentes manières :

- en langage naturel,
- graphiquement,
- en pseudo-code,
- par un programme écrit dans un langage informatique
- ...

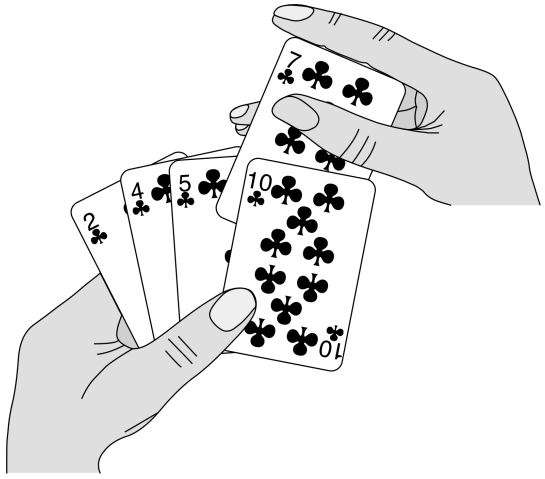
La seule condition est que la description soit précise.

Exemple : le tri

- Le problème de tri :
  - Entrée : une séquence de n nombres $\langle a_1, a_2, \ldots, a_n \rangle$
  - Sortie : une permutation de la séquence de départ $\langle a'_1, a'_2, \ldots, a'_n \rangle$ telle que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

- Exemple :
  - Entrée : $\langle 31, 41, 59, 26, 41, 58 \rangle$
  - Sortie : $\langle 26, 31, 41, 41, 58, 59 \rangle$

Tri par insertion



Description en langage naturel :

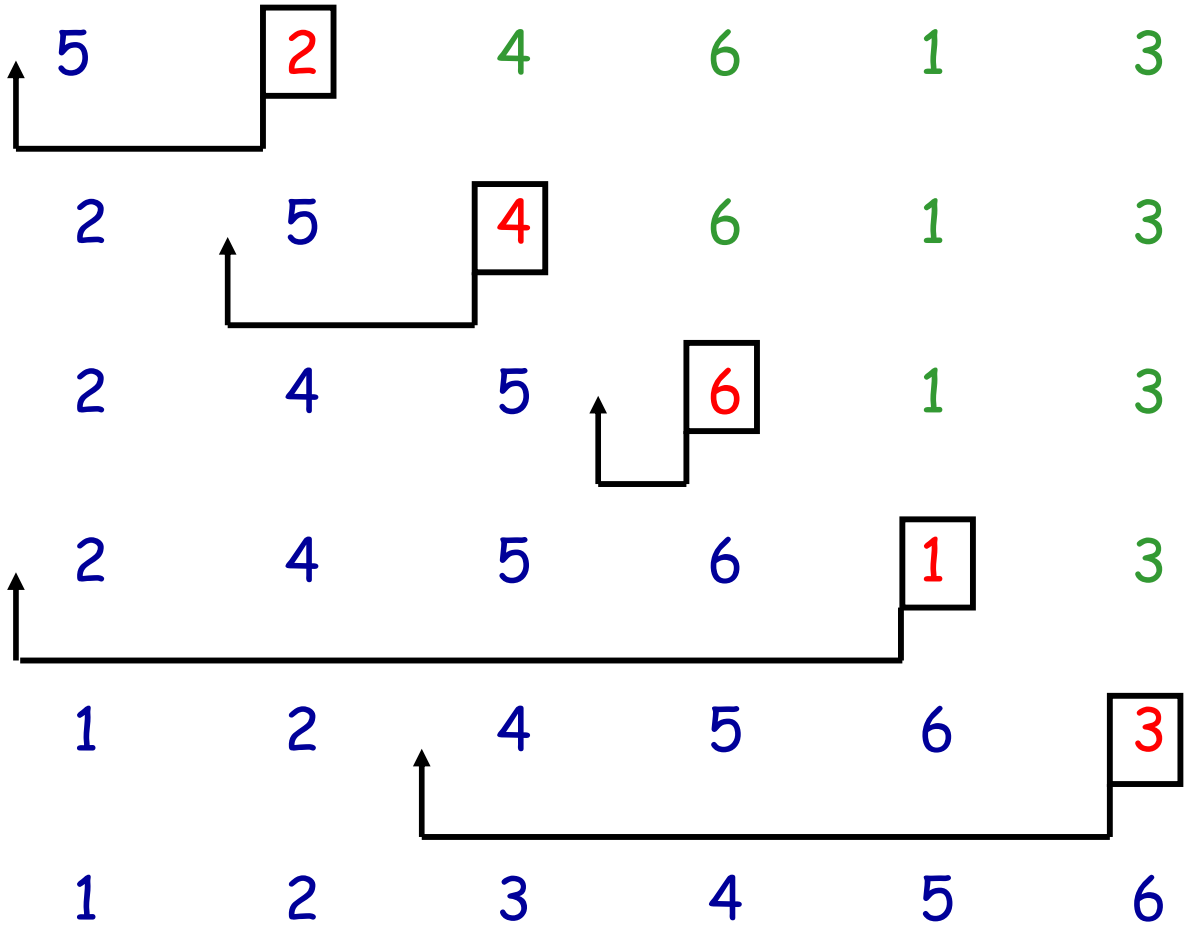
On parcourt la séquence de gauche à droite

Pour chaque élément a_j :

- On l'insère à sa position dans une nouvelle séquence ordonnée contenant les éléments le précédant dans la séquence.

On s'arrête dès que le dernier élément a été inséré à sa place dans la séquence.

Tri par insertion



Tri par insertion

Description en C (sur des tableaux d'entiers) :

```
void InsertionSort (int *a, int length) {
  int key, i;
  for(int j = 1; j < length; j++) {
    key = a[j];
    /* Insert a[j] into the sorted sequence a[0...j-1] */
    i = j-1;
    while (i>=0 && a[i]>key) {
      a[i+1] = a[i];
      i = i-1;
    }
    a[i+1] = key;
  }
}
```

Insertion sort

Description en **pseudo-code** (sur des tableaux d'entiers) :

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Pseudo-code

Objectifs :

- Décrire les algorithmes de manière à ce qu'ils soient compris par des humains.
- Rendre la description indépendante de l'implémentation
- S'affranchir de détails tels que la gestion d'erreurs, les déclarations de type, etc.

Très proche du C (langage procédural plutôt qu'orienté objet)

Peut contenir certaines instructions en langage naturel si nécessaire

Pseudo-code

Quelques règles

- Structures de blocs indiquées par l'indentation
- Boucles (**for**, **while**, **repeat**) et conditions (**if**, **else**, **elseif**) comme en C.
- Le compteur de boucle garde sa valeur à la sortie de la boucle
- En sortie d'un **for**, le compteur a la valeur de la borne max+1.

$$\textbf{for } i = 1 \textbf{ to } Max \quad Code \quad \Leftrightarrow \quad i = 1 \quad \textbf{while } i \leq Max \quad Code \quad i = i + 1$$

- Commentaires indiqués par //
- Affectation (=) et test d'égalité (==) comme en C.

Pseudo-code

- Les variables (i, j et *key* par exemple) sont locales à la fonction.
- $A[i]$ désigne l'élément i du tableau A. $A[i..j]$ désigne un intervalle de valeurs dans un tableau. $A.length$ est la taille du tableau.
- L'indexation des tableaux commence à 1.
- Les types de données composés sont organisés en *objets*, qui sont composés d'attributs. On accède à la valeur de l'attribut *attr* pour un objet x par $x.attr$.
- Un variable représentant un tableau ou un objet est considérée comme un pointeur vers ce tableau ou cet objet.
- Paramètres passés par valeur comme en C (mais tableaux et objets sont passés par pointeur).
- ...

Trois questions récurrentes face à un algorithme

1. Mon algorithme est-il correct, se termine-t-il ?
2. Quelle est sa vitesse d'exécution ?
3. Y-a-t'il moyen de faire mieux ?

Exemple du tri par insertion

1. Oui $\rightarrow$ technique des invariants (partie 2)
2. $O(n^2) \rightarrow$ analyse de complexité (partie 2)
3. Oui $\rightarrow$ il existe un algorithme $O(n \log n)$ (partie 1)

Correction de Insertion-Sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      i = j − 1
4      while i > 0 and A[i] > key
5          A[i + 1] = A[i]
6          i = i − 1
7      A[i + 1] = key
```

- Invariant : (pour la boucle externe) le sous-tableau $A[1\,..\,j-1]$ contient les éléments du tableau original $A[1\,..\,j-1]$ ordonnés.
- On doit montrer que
  - l'invariant est vrai avant la première itération
  - l'invariant est vrai avant chaque itération suivante
  - En sortie de boucle, l'invariant implique que l'algorithme est correct

Correction de Insertion-Sort

- Avant la première itération :
  - $j = 2 \Rightarrow A[1]$ est trivialement ordonné.
- Avant la $j$ème itération :
  - Informellement, la boucle interne déplace $A[j-1]$, $A[j-2]$, $A[j-3] \ldots$ d'une position vers la droite jusqu'à la bonne position pour *key* ($A[j]$).
- En sortie de boucle :
  - A la sortie de boucle, $j = A.length + 1$. L'invariant implique que $A[1 \,..\, A.length]$ est ordonné.

Complexité de Insertion-Sort

```
INSERTION-SORT(A)
1   for j = 2 to A.length
2       key = A[j]
3       i = j − 1
4       while i > 0 and A[i] > key
5           A[i + 1] = A[i]
6           i = i − 1
7       A[i + 1] = key
```

- Nombre de comparaisons $T(n)$ pour trier un tableau de taille n ?
- Dans le pire des cas :
  - La boucle **for** est exécutée $n - 1$ fois ($n = A.length$).
  - La boucle **while** est exécutée $j - 1$ fois

Complexité de Insertion-Sort

- Le nombre de comparaisons est borné par :

$$T(n) \leq \sum_{j=2}^{n} (j-1)$$

- Puisque $\sum_{i=1}^{n} i = n(n+1)/2$, on a :

$$T(n) \leq \frac{n(n-1)}{2}$$

- Finalement, $T(n) = O(n^2)$

(borne inférieure ?)

Structures de données

- Méthode pour stocker et organiser les données pour en faciliter l'accès et la modification
- Une structure de données regroupe :
  - un certain nombre de données à gérer, et
  - un ensemble d'opérations pouvant être appliquées à ces données
- Dans la plupart des cas, il existe
  - plusieurs manières de représenter les données et
  - différents algorithmes de manipulation.
- On distingue généralement l'interface des structures de leur implémentation.

Types de données abstraits

- Un type de données abstrait (TDA) représente l'interface d'une structure de données.
- Un TDA spécifie précisément :
  - la nature et les propriétés des données
  - les modalités d'utilisation des opérations pouvant être effectuées
- En général, un TDA admet différentes implémentations (plusieurs représentations possibles des données, plusieurs algorithmes pour les opérations).

Exemple : file à priorités

- Données gérées : des objets avec comme attributs :
  - une clé, munie d'un opérateur de comparaison selon un ordre total
  - une valeur quelconque

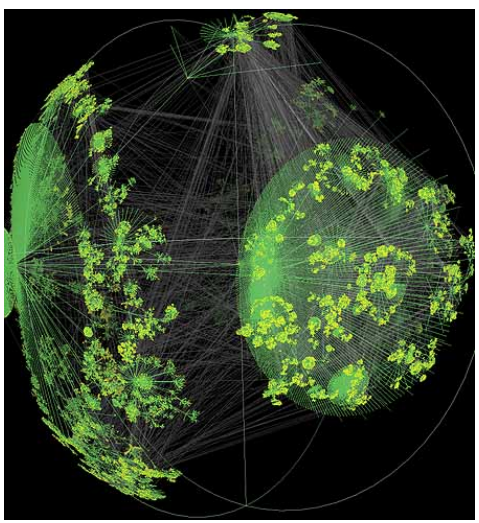
- Opérations :
  - Création d'une file vide
  - $\textsc{Insert}(S, x)$: insère l'élément x dans la file S.
  - $\textsc{Extract-Max}(S)$: retire et renvoie l'élément de S avec la clé la plus grande.

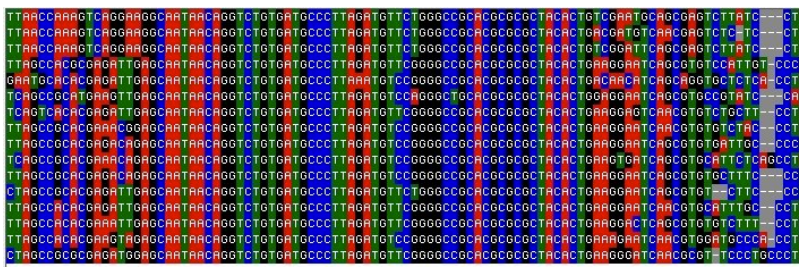
- Il existe de nombreuses façons d'implémenter ce TDA :
  - Tableau non trié ;
  - Liste triée ;
  - Structure de tas ;
  - ...

  Chacune mène à des complexités différentes des opérations $\textsc{Insert}$ et $\textsc{Extract-Max}$

Structures de données et algorithmes en pratique

- La résolution de problème algorithmiques requiert presque toujours la combinaison de structures de données et d'algorithmes sophistiqués pour la gestion et la recherche dans ces structures.
- D'autant plus vrai qu'on a à traiter des volumes de données importants.
- Quelques exemples de problèmes réels :
  - Routage dans les réseaux informatiques
  - Moteurs de recherche
  - Alignement de séquences ADN en bio-informatique





Un exemple (B. Boigelot)

- Un laboratoire de génie génétique désire développer un programme capable de repérer des répétitions de longueur M dans une séquence de nucléotides S de longueur N (avec $N >> M$) :

$$ACTG\underline{CGAC}GGTACGCTT\underline{CGAC}TTAG...(M = 4)$$

- Première approche :
  - Un indice i variant de 2 à $N - M + 1$
  - Un indice j variant de 1 à $i - 1$
  - Pour tout $k \in [0, \ldots, M - 1]$, on teste si $S[i + k] = S[j + k]$.
- Efficacité : le nombre de comparaisons à effectuer est égal à :

$$\begin{aligned} M \cdot (1 + 2 + \ldots + (N - M)) &= \frac{M(N - M + 1)(N - M)}{2} \\ &\approx 4,5.10^{21} \text{ pour } N = 3 \cdot 10^9 \text{ et } M = 1000 \\ &\approx 143.000 \text{ ans au rythme de } 10^9 \text{ opérations/s.} \end{aligned}$$

Une meilleure solution

1. On construit une table à $N - M + 1$ lignes et M colonnes dont la k-ème ligne contient la sous-séquence de longueur M commençant à la position k dans S :

   ACTG
   CTGC
   TGCG
   GCGA
   CGAC
   ⋮

2. On trie les lignes de cette table par ordre lexicographique ;
3. On parcourt la table triée afin de déterminer si elle contient deux lignes consécutives identiques

Note : Lors de la comparaison de deux lignes, on s'arrête à la première différence ($\Rightarrow$ moins de 4/3 comparaisons en moyenne).

Efficacité

- Construction de la table : $M(N - M + 1)$ opérations de copie.
- Tri par ordre lexicographique (algorithme de tri rapide, voir partie 3) :

$$\leq \frac{8}{3} N \ln N \text{ opérations de comparaison en moyenne}$$

- Détection de lignes consécutives :

$$\leq \frac{4}{3}(N - M) \text{ opérations de comparaison en moyenne}$$

En supposant des coûts identiques pour toutes les opérations, on obtient :

$$\begin{aligned} & N(M + \frac{8}{3} \ln N + \frac{4}{3}) - M(M + \frac{1}{3}) \\ \approx \quad & 3,179.10^{12} \text{ opérations pour } N = 3.10^9 \text{ et } M = 1000. \\ \approx \quad & 53 \text{ minutes au rythme de } 10^9 \text{ opérations/s.} \end{aligned}$$

Remarques

- Utiliser un ordinateur plus puissant ne permet généralement pas de résoudre les problèmes d'efficacité !
  Avec un ordinateur 1000 fois plus puissant : 143 ans pour la première approche, 3,2s pour la deuxième.
- La deuxième solution est plus rapide mais elle est très gourmande en espace mémoire (M fois plus que la première !)

Plan

1. Algorithms + Data structures = Programs (Niklaus Wirth)

2. Introduction à la récursivité

Algorithmes récursifs

Un algorithme est **récursif** s'il s'invoque lui-même directement ou indirectement.

Motivation : Simplicité d'expression de certains algorithmes

Exemple : Fonction factorielle :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

```
FACTORIAL(n)
1  if n == 0
2        return 1
3  return n · FACTORIAL(n − 1)
```

Algorithmes récursifs

```
FACTORIAL(n)
1  if n == 0
2        return 1
3  return n · FACTORIAL(n − 1)
```

Règles pour développer une solution récursive :

- On doit définir un cas de base ($n == 0$)
- On doit diminuer la "taille" du problème à chaque étape ($n \to n-1$)
- Quand les appels récursifs se partagent la même structure de données, les sous-problèmes ne doivent pas se superposer (pour éviter les effets de bord)

Exemple de récursion multiple

Calcul du $n$ième nombre de Fibonacci :

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ \forall n \geq 2 : F_n &= F_{n-2} + F_{n-1} \end{aligned}$$

Algorithme :

```
FIBONACCI(n)
1  if n ≤ 1
2        return n
3  return FIBONACCI(n − 2) + FIBONACCI(n − 1)
```

Exemple de récursion multiple

```
FIBONACCI(n)
1  if n ≤ 1
2        return n
3  return FIBONACCI(n − 2) + FIBONACCI(n − 1)
```

1. L'algorithme est-il correct ?
2. Quelle est sa vitesse d'exécution ?
3. Y-a-t'il moyen de faire mieux ?

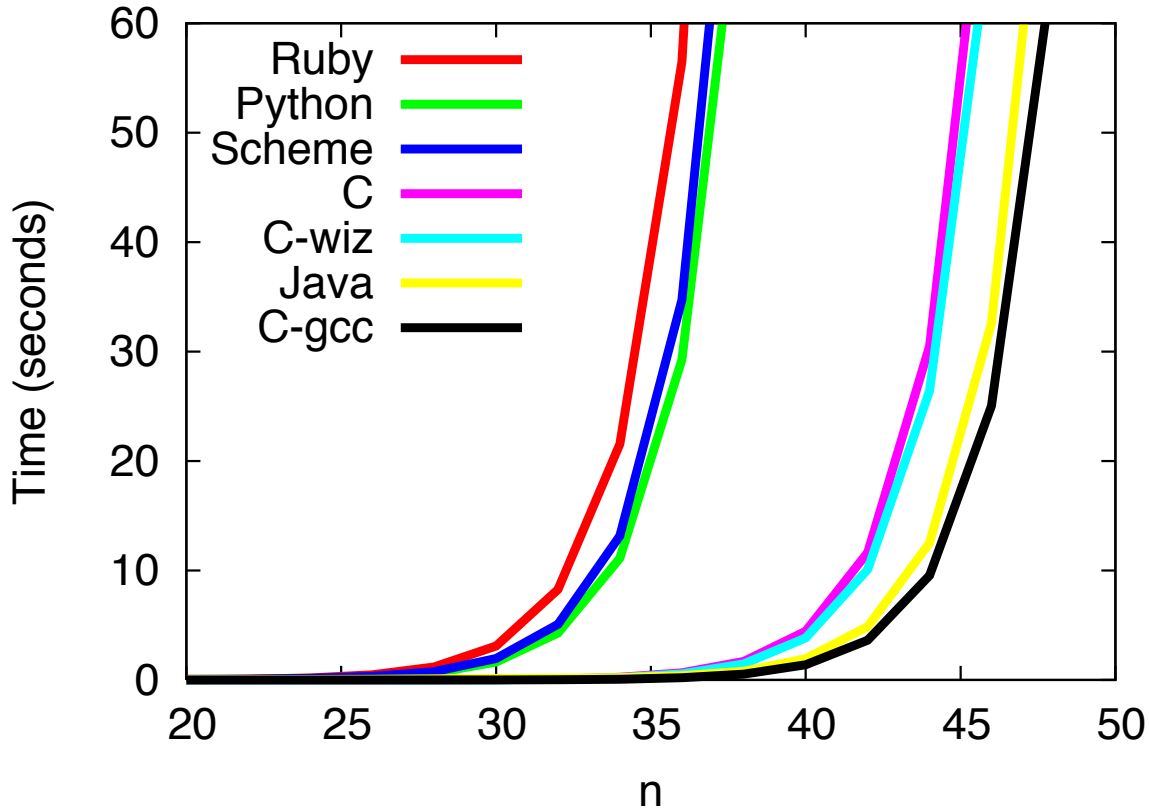
Exemple de récursion multiple

```
FIBONACCI(n)
1  if n ≤ 1
2        return n
3  return FIBONACCI(n − 2) + FIBONACCI(n − 1)
```

1. L'algorithme est correct ?
   - Clairement, l'algorithme est correct.
   - En général, la correction d'un algorithme récursif se démontre par induction.
2. Quelle est sa vitesse d'exécution ?
3. Y-a-t'il moyen de faire mieux ?

Vitesse d'exécution

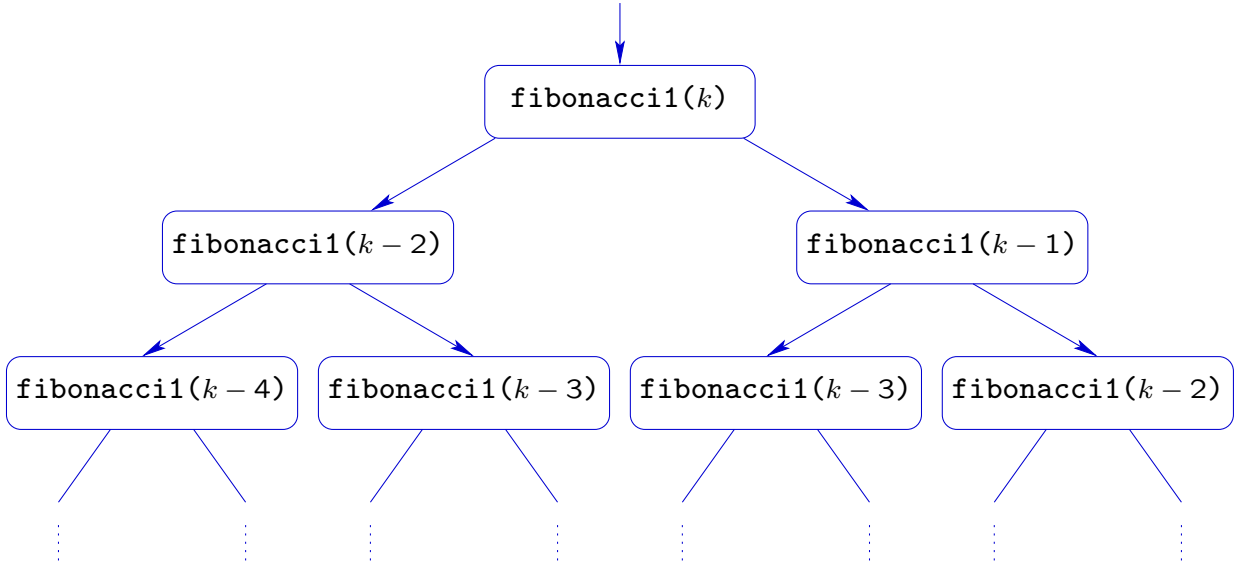
- Nombre d'opérations pour calculer $\textsc{Fibonacci}(n)$ en fonction de n
- Empiriquement :



(Carzaniga)

- Toutes les implémentations atteignent leur limite, plus ou moins loin

Trace d'exécution



(Boigelot)

Complexité

```
FIBONACCI(n)
1  if n ≤ 1
2        return n
3  return FIBONACCI(n − 2) + FIBONACCI(n − 1)
```

- Soit $T(n)$ le nombre d'opérations de base pour calculer FIBONACCI(n) :

$$\begin{aligned} T(0) &= 2, T(1) = 2 \\ T(n) &= T(n-1) + T(n-2) + 2 \end{aligned}$$

- On a donc $T(n) \geq F_n$ ($=$ le $n$ème nombre de Fibonacci).

Complexité

- Comment croît F_n avec n ?

$$T(n) \geq F_n = F_{n-1} + F_{n-2}$$

Puisque $F_n \geq F_{n-1} \geq F_{n-2} \geq \ldots$, on a :

$$F_n \geq 2F_{n-2} \geq 2(2F_{n-4}) \geq 2(2(2F_{n-6})) \geq 2^{\frac{n}{2}-1}F_2 = \frac{(\sqrt{2})^n}{2}$$

si $n \geq 2$ est pair et

$$F_n \geq 2F_{n-2} \geq 2(2F_{n-4}) \geq 2(2(2F_{n-6})) \geq 2^{\frac{n-1}{2}}F_1 = \frac{(\sqrt{2})^n}{\sqrt{2}} \geq \frac{(\sqrt{2})^n}{2}$$

si $n \geq 1$ est impair.

Et donc

$$T(n) \geq \frac{(\sqrt{2})^n}{2} \approx \frac{(1.4)^n}{2}$$

- $T(n)$ croît exponentiellement avec n

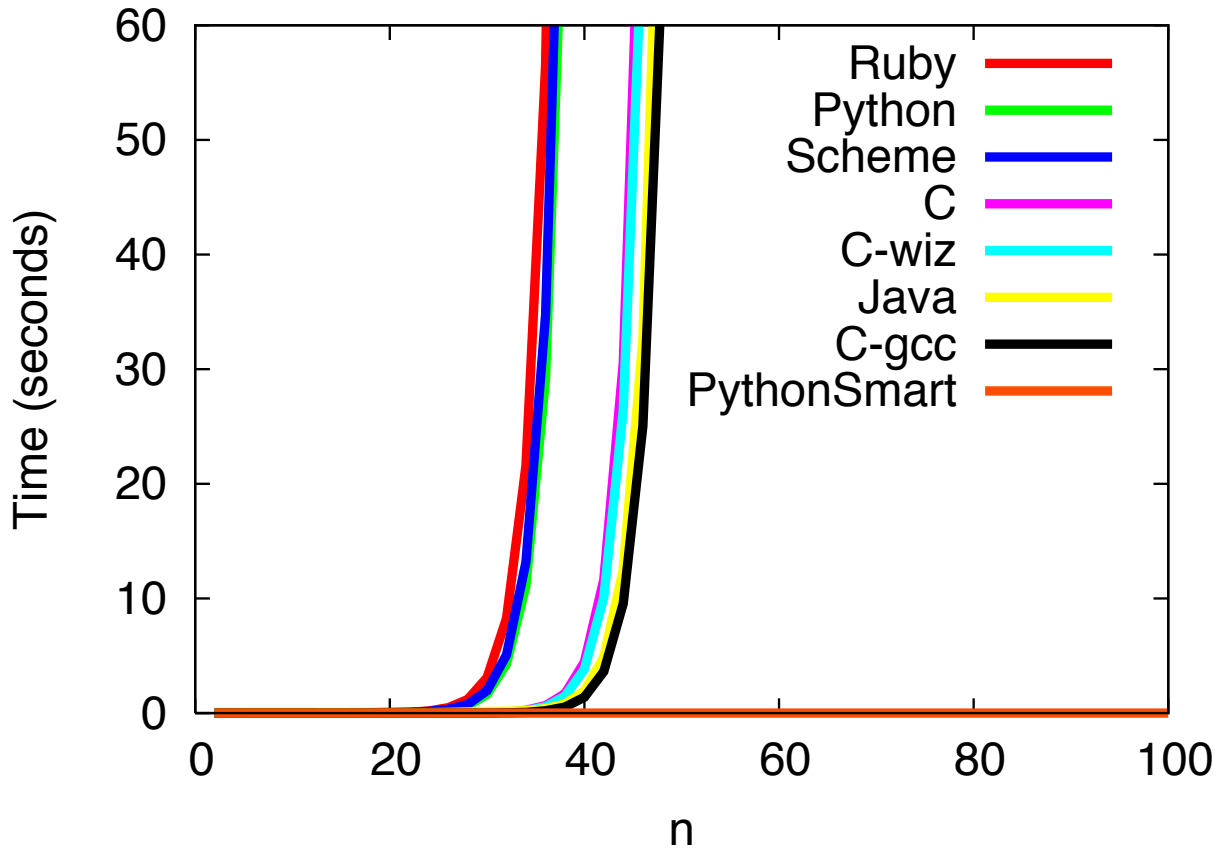
- Peut-on faire mieux ?

Solution itérative

```
FIBONACCI-ITER(n)
 1  if n ≤ 1
 2        return n
 3  else
 4        pprev = 0
 5        prev = 1
 6        for i = 2 to n
 7              f = prev + pprev
 8              pprev = prev
 9              prev = f
10        return f
```

Vitesse d'exécution

Complexité : $O(n)$



(Carzaniga)

Tri par fusion

Idée d'un tri basé sur la récursion :

- on sépare le tableau en deux sous-tableaux de la même taille
- on trie (récursivement) chacun des sous-tableaux
- on fusionne les deux sous-tableaux triés en maintenant l'ordre

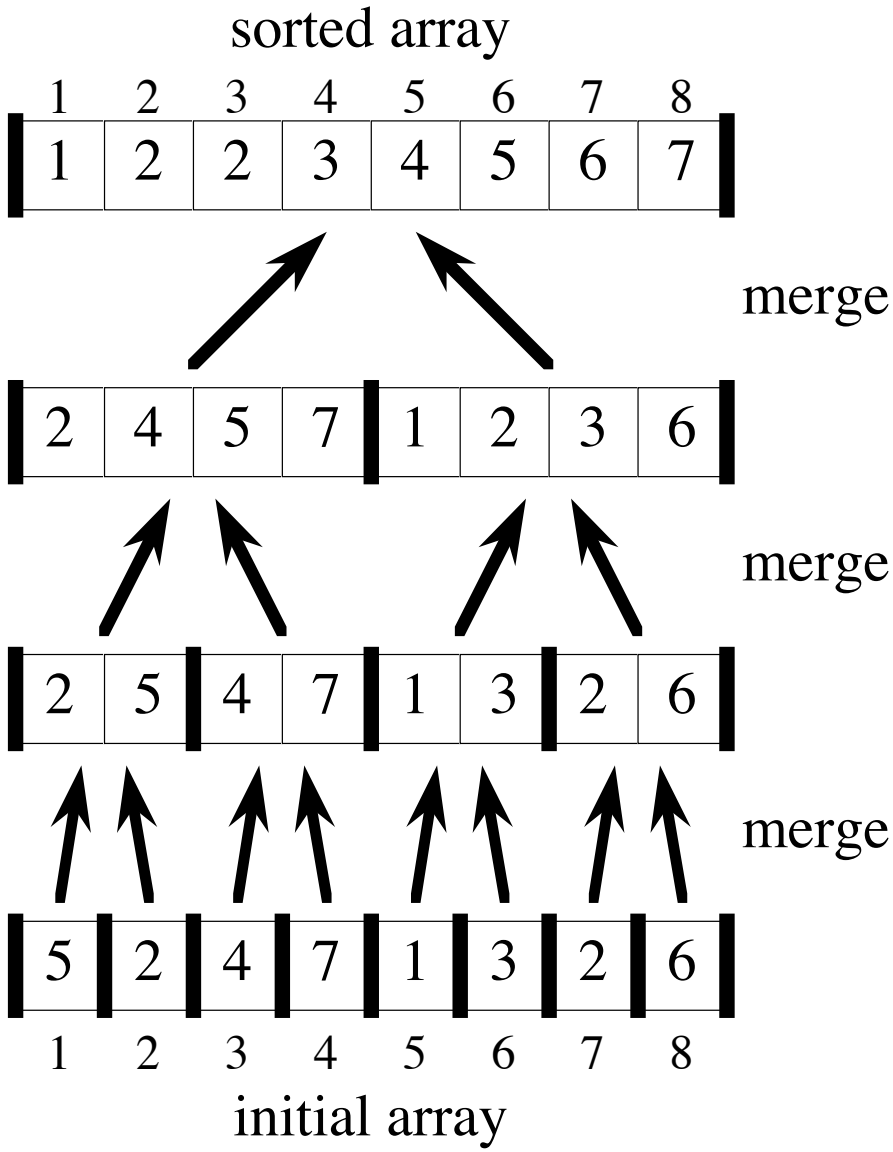
Le cas de base correspond à un tableau d'un seul élément.

```
MERGE-SORT(A, p, r)
1  if p < r
2        q = ⌊(p+r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q + 1, r)
5        MERGE(A, p, q, r)
```

Appel initial : MERGE-SORT$(A, 1, A.length)$

Exemple d'application du principe général de "diviser pour régner"

Tri par fusion : illustration



Fonction MERGE

MERGE(A, p, q, r) :

- **Entrée :** tableau A et indice p, q, r tels que :
  - $p \leq q < r$ (pas de tableaux vides)
  - Les sous-tableaux $A[p\,..\,q]$ et $A[q+1\,..\,r]$ sont ordonnés
- **Sortie :** Les deux sous-tableaux sont fusionnés en un seul sous-tableau ordonné dans $A[p\,..\,r]$

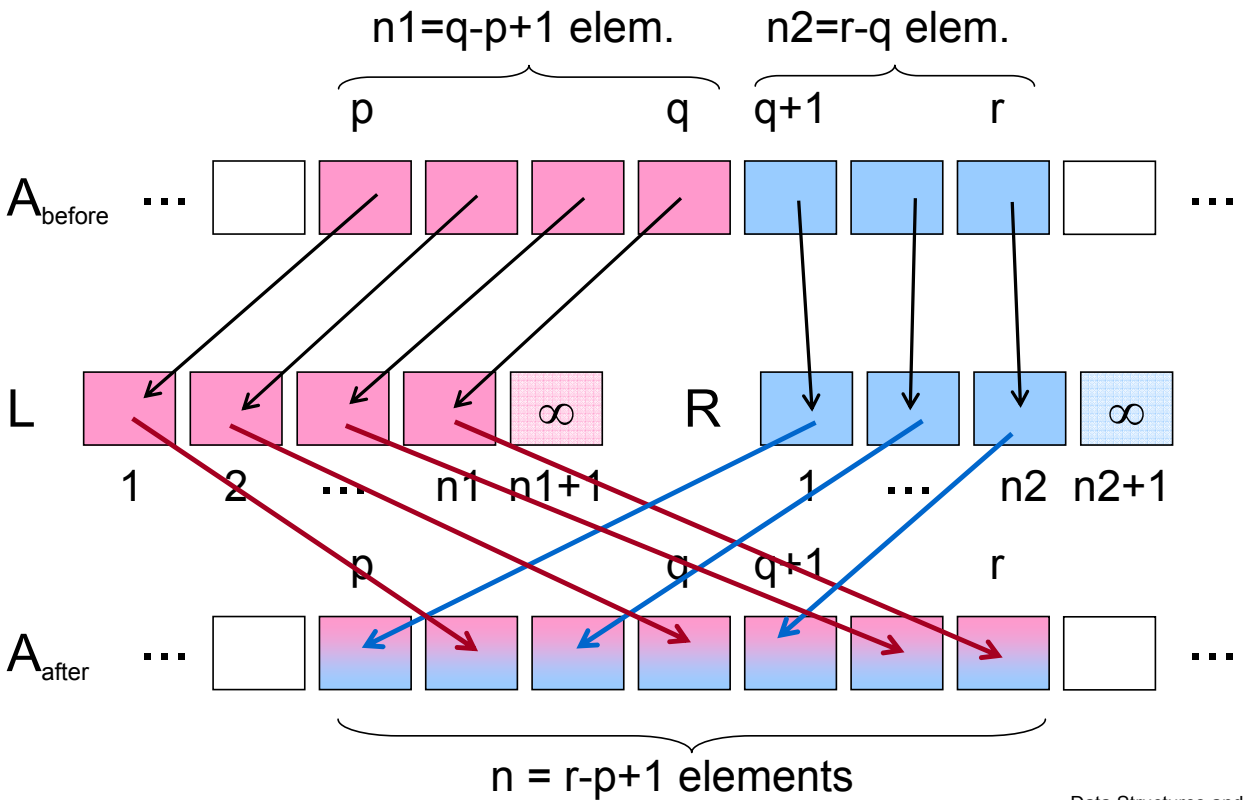
Idée :

- Utiliser un pointeur vers le début de chacun des sous-tableaux ;
- Déterminer le plus petit des deux éléments pointés ;
- Déplacer cet élément vers le tableau fusionné ;
- Avancer le pointeur correspondant

Fusion : algorithme

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1 ; n2 = r − q
 2  Soit L[1..n1 + 1] et R[1..n2 + 1] deux nouveaux tableaux
 3  for i = 1 to n1
 4        L[i] = A[p + i − 1]
 5  for j = 1 to n2
 6        R[j] = A[q + j]
 7  L[n1 + 1] = ∞ ; R[n2 + 1] = ∞                    // Sentinels
 8  i=1 ;j=1
 9  for k = p to r
10        if L[i] ≤ R[j]
11              A[k] = L[i]
12              i = i + 1
13        else
14              A[k] = R[j]
15              j = j + 1
```

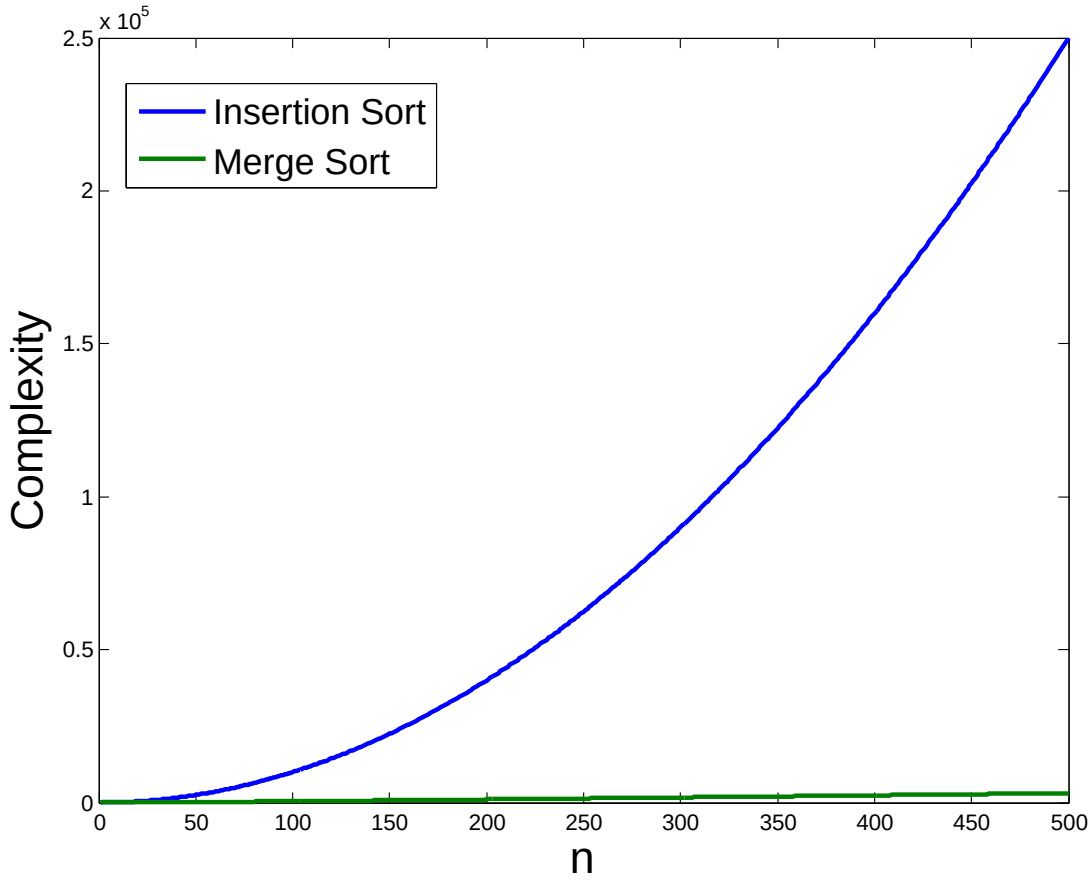
Fusion : illustration



Complexité : $O(n)$ (où $n = r - p + 1$)

Vitesse d'exécution

Complexité de MERGE-SORT : $O(n \log n)$ (voir partie 2)



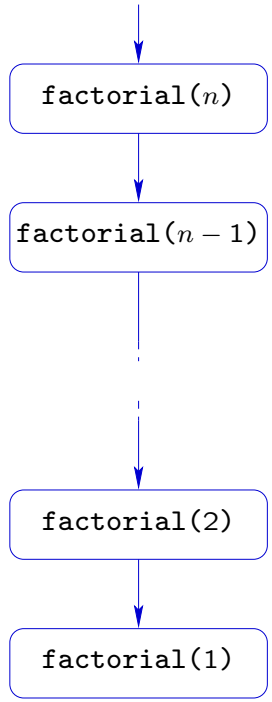
Remarques

- La fonction MERGE nécessite d'allouer deux tableaux L et R (dont la taille est $O(n)$).
- On pourrait réécrire MERGE-SORT de manière itérative (au prix de la simplicité)
- Version récursive du tri par insertion :

```
INSERTION-SORT-REC(A, n)
1  if n > 1
2        INSERTION-SORT-REC(A, n − 1)
3        MERGE(A, 1, n − 1, n)
```

Note sur l'implémentation de la récursivité

- Trace d'exécution de la factorielle



- Chaque appel récursif nécessite de mémoriser le contexte d'invocation
- L'espace mémoire utilisé est donc $O(n)$ (n appels récursifs)

Récursivité terminale

- Définition : Une procédure est récursive terminale (“tail recursive”) si elle n’effectue plus aucune opération après s’être invoquée récursivement.

- Avantages :
  - Le contexte d’invocation ne doit pas être mémorisé et donc l’espace mémoire nécessaire est réduit
  - Les procédures récursives terminales peuvent facilement être converties en procédures itératives

Version récursive terminale de la factorielle

```
FACTORIAL2(n)
1  return FACTORIAL2-REC(n, 2, 1)
```

```
FACTORIAL2-REC(n, i, f)
1  if i > n
2        return f
3  return FACTORIAL2-REC(n, i + 1, f · i)
```

Espace mémoire utilisé : $O(1)$ (si la récursion terminale est implémentée efficacement)

Ce qu'on a vu

- Définitions générales : algorithmes, structures de données, structures de données abstraites...
- Analyse d'un algorithme itératif (Insertion-Sort)
- Notions de récursivité
- Analyse d'un algorithme récursif (Fibonacci)
- Tri par fusion (MergeSort)

Partie 2

Outils d'analyse

Plan

1. Correction d'algorithmes

2. Complexité algorithmique

Plan

Analyse d'algorithmes

Questions à se poser lors de la définition d'un algorithme :

- Mon algorithme est-il correct ?
- Mon algorithme est-il efficace ?

Autres questions importantes seulement marginalement abordées dans ce cours :

- Modularité, fonctionnalité, robustesse, facilité d'utilisation, temps de programmation, simplicité, extensibilité, fiabilité, existence d'une solution algorithmique...

Correction d'un algorithme

- La correction d'un algorithme s'étudie par rapport à un problème donné
- Un problème est une collection d'instances de ce problème.
  - Exemple de problème : trier un tableau
  - Exemple d'instance de ce problème : trier le tableau $[8, 4, 15, 3]$
- Un algorithme est correct pour une instance d'un problème s'il produit une solution correcte pour cette instance
- Un algorithme est correct pour un problème s'il est correct pour toutes ses instances (on dira qu'il est totalement correct)
- On s'intéressera ici à la correction d'un algorithme pour un problème (et pas pour seulement certaines de ses instances)

Comment vérifier la correction ?

- Première solution : en testant concrètement l'algorithme :
  - Suppose d'implémenter l'algorithme dans un langage (programme) et de le faire tourner
  - Suppose qu'on peut déterminer les instances du problème à vérifier
  - Il est très difficile de prouver empiriquement qu'on n'a pas de bug
- Deuxième solution : en dérivant une preuve mathématique formelle :
  - Pas besoin d'implémenter et de tester toutes les instances du problème
  - Sujet à des "bugs" également
- En pratique, on combinera les deux

- Outils pour prouver la correction d'un algorithme :
  - Algorithmes itératifs : triplets de Hoare, invariants de boucle
  - Algorithmes récursifs : preuves par induction

Assertion

- Relation entre les variables qui est vraie à un moment donné dans l'exécution
- Assertions particulières :
  - Pre-condition P : conditions que doivent remplir les entrées valides de l'algorithme
  - Post-condition Q : conditions qui expriment que le résultat de l'algorithme est correct
- P et Q définissent resp. les instances et solutions valides du problème
- Un code est correct si le triplet (de Hoare) $\{P\}$ code $\{Q\}$ est vrai.
- Exemple :

$$\{x \geq 0\} y = \mathrm{SQRT}(x) \{y^2 == x\}$$

Correction : séquence d'instructions

$\{P\}$
S1
S2
...
Sn
$\{Q\}$

Pour vérifier que le triplet est correct :

- on insère des assertions P_1, P_2, ..., P_n décrivant l'état des variables à chaque étape du programme
- on vérifie que les triplets $\{P\}$ S1 $\{P_1\}$, $\{P_1\}$ S2 $\{P_2\}$, ..., $\{P_{n-1}\}$ Sn $\{Q\}$ sont corrects

Trois types d'instructions : affectation, condition, boucle

Correction : affectations

Le triplet suivant est correct :

$$\begin{array}{l}\{Q[x \to e]\}\\ x = e\\ \{Q\}\end{array}$$

$Q[x \to e]$ est obtenu en remplaçant les occurrences de x par e dans Q.

Pour prouver un triplet :

$$\begin{array}{l}\{P\}\\ x = e\\ \{Q\}\end{array}$$

il faut montrer que P implique $Q[x \to e]$.

Exemples : les triplets suivants sont corrects

$$\begin{array}{l}\{x == 2\}\\ y = x + 1\\ \{y == 3\}\end{array}$$

$$\begin{array}{l}\{x == 42\}\\ y = x + 1\\ z = y\\ \{z == 43\}\end{array}$$

Correction : conditions

$\{P\}$
if B
    C1
else
    C2
$\{Q\}$

Pour prouver que le triplet est correct, on doit prouver que

- $\{P$ et $B\}$ C1 $\{Q\}$
- $\{P$ et non $B\}$ C2 $\{Q\}$

sont corrects

Exemple :

$\{x < 6\}$
if $x < 0$
    $y = 0$
else
    $y = x$
$\{0 \leq y < 6\}$

Correction : boucles

$\{P\}$
INIT
while B
    CORPS
FIN
$\{Q\}$

$\{P\}$
INIT
$\{I\}$
while B
    $\{I$ et $B\}$ CORPS $\{I\}$
$\{I$ et non $B\}$
FIN
$\{Q\}$

Pour prouver que le triplet est correct :

- On met en évidence une assertion particulière I, appelée invariant de boucle, qui décrit l'état du programme pendant la boucle.
- On prouve que :
  - $\{P\}$ INIT $\{I\}$ est correct
  - $\{I$ et $B\}$ CORPS $\{I\}$ est correct
  - $\{I$ et non $B\}$ FIN $\{Q\}$ est correct

Si on a plusieurs boucles imbriquées, on les traite séparément, en démarrant avec la boucle la plus interne.

Correction : terminaison de boucle

```
INIT
while B
      CORPS
FIN
```

- Un fois qu'on a prouvé que le triplet était correct, il faut encore montrer que la boucle se termine
- Pour prouver la terminaison, on cherche une fonction de terminaison f :
  - définie sur base des variables de l'algorithme et à valeur entière naturelle (≥ 0)
  - telle que f décroît strictement suite à l'exécution du corps de la boucle
  - telle que B implique $f > 0$
- Puisque f décroît strictement, elle finira par atteindre 0 et donc à infirmer B.

Exemple : Fibonacci-iter

```
Fibonacci-Iter(n)
    if n ≤ 1
        return n
    else
        pprev = 0
        prev = 1
        for i = 2 to n
            f = prev + pprev
            pprev = prev
            prev = f
        return f
```

Proposition : Si $n \geq 0$, Fibonacci-iter(n) renvoie F_n.

Réécriture, post- et pré-conditions

```
Fibonacci-Iter(n)
    {n ≥ 0} // {P}
    if n ≤ 1
        prev = n
    else
        pprev = 0
        prev = 1
        i = 2
        while (i ≤ n)
            f = prev + pprev
            pprev = prev
            prev = f
            i = i + 1
    {prev == F_n} // {Q}
    return prev
```

Exemple : FIBONACCI-ITER

Analyse de la condition

$\{n \geq 0 \text{ et } n \leq 1\}$
$prev = n$
$\{prev == F_n\}$

correct $(F_0 = 0, F_1 = 1)$

$\{n \geq 0 \text{ et } n > 1\}$
$pprev = 0$
$prev = 1$
$i = 2$
while $(i \leq n)$
    $f = prev + pprev$
    $pprev = prev$
    $prev = f$
    $i = i + 1$
$\{prev == F_n\}$

$I = \{pprev == F_{i-2}, prev == F_{i-1}\}$

Analyse de la boucle

$\{n > 1\}$
$pprev = 0$
$prev = 1$
$i = 2$
$\{pprev == F_{i-2}, prev == F_{i-1}\}$

correct

$\{pprev == F_{i-2}, prev == F_{i-1}, i \leq n\}$
$f = prev + pprev$
$pprev = prev$
$prev = f$
$i = i + 1$
$\{pprev == F_{i-2}, prev == F_{i-1}\}$

correct

$\{pprev == F_{i-2}, prev == F_{i-1}, i == n + 1\}$
$\{prev == F_n\}$

correct

Exemple : Fibonacci-iter

$i = 2$
while $(i \leq n)$
$\quad f = prev + pprev$
$\quad pprev = prev$
$\quad prev = f$
$\quad i = i + 1$

- Fonction de terminaison $f = n - i + 1$:
  - $i \leq n \Rightarrow f = n - i + 1 > 0$
  - $i = i + 1 \Rightarrow f$ diminue à chaque itération
- L'algorithme est donc correct et se termine.

Exemple : tri par insertion

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

- Démontrons informellement que la boucle externe est correcte
- Invariant I : le sous-tableau $A[1..j-1]$ contient les éléments du tableau original $A[1..j-1]$ ordonnés.

Exemple : tri par insertion

$$\textbf{for } j = 2 \textbf{ to } A.length \quad \ldots \quad \Leftrightarrow \quad \begin{array}{l} j = 2 \\ \textbf{while } i \leq A.length \\ \quad \ldots \\ \quad j = j + 1 \end{array}$$

- $P =$"A est un tableau de taille $A.length$",
  $Q =$"Le tableau A est trié",
  $I =$"$A[1\,..\,j-1]$ contient les $j-1$ premiers éléments de A triés"

- $\{P\}j = 2\{I\}$ *(avant la boucle)*
  - $j = 2 \Rightarrow A[1]$ est trivialement ordonné

Exemple : tri par insertion

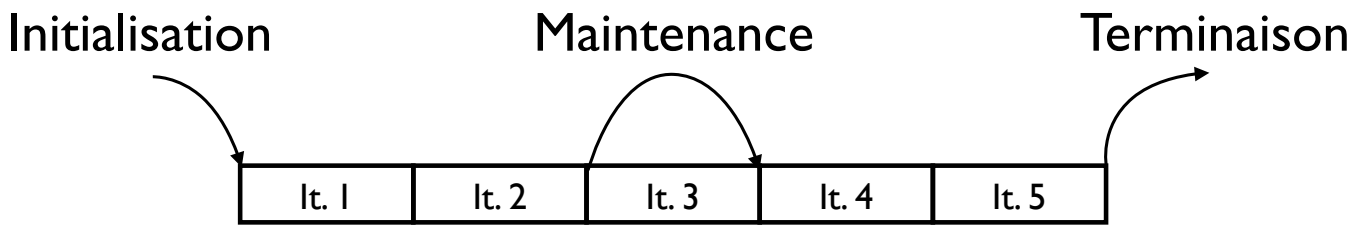
- $\{I$ et $j \leq A.length\}$ CORPS $\{I\}$ *(pendant la boucle)*
  - La boucle interne déplace $A[j-1]$, $A[j-2]$, $A[j-3]\ldots$ d'une position vers la droite jusqu'à trouver la bonne position pour *key* ($A[j]$).
  - $A[1\,..\,j]$ contient alors les éléments originaux de $A[1\,..\,j]$ triés.
  - $j = j+1$ rétablit l'invariant
- $\{I$ et $j = A.length + 1\}\{Q\}$ *(après la boucle)*
  - Puisque $j = A.length + 1$, l'invariant implique que $A[1\,..\,A.length]$ est ordonné.

- Fonction de terminaison $f = A.length - j + 1$

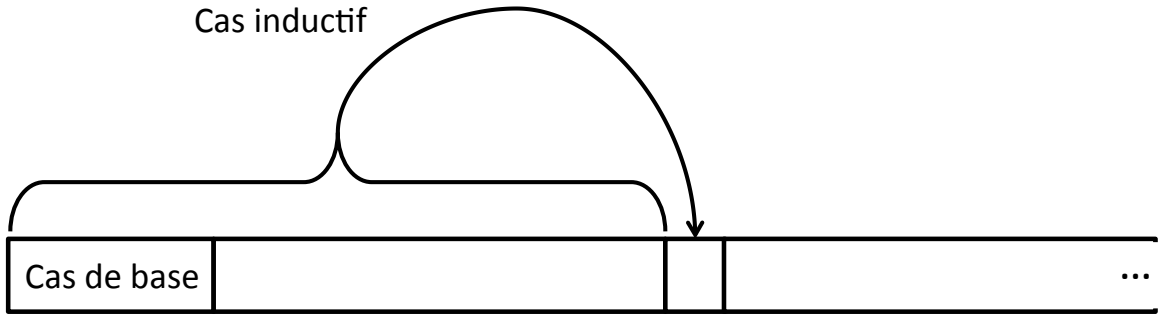
Invariant

- Un invariant peut être difficile à trouver pour certains algorithmes
- En général, l'algorithme découle de l'invariant et pas l'inverse
  - FIBONACCI-ITER : On calcule itérativement F_{i-1}et F_{i-2}
    ($I = \{pprev == F_{i-2}, prev == F_{i-1}\}$)
  - INSERTION-SORT : On ajoute l'élément j aux $j-1$ premiers éléments déjà triés
    ($I =$" $A[1..j-1]$ contient les $j-1$ premiers éléments de A triés")

- La preuve par invariant est basée sur le principe général de preuve par induction qu'on va utiliser aussi pour prouver la correction des algorithmes récursifs



Preuve par induction



- On veut montrer qu'une propriété est vraie pour une série d'instances
- On suppose l'existence d'un ordonnancement des instances
- Cas de base : on montre explicitement que la propriété est vraie pour la ou les premières instances
- Cas inductif : on suppose que la propriété est vraie pour les k premières instances et on montre qu'elle l'est alors aussi pour la $k+1$-ième instance (quel que soit k)
- Par le principe d'induction, la propriété sera vraie pour toutes les instances

Preuve par induction : exemple

Proposition : Pour tout $n \geq 0$, on a

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Démonstration :

- Cas de base : $n = 0 \Rightarrow \sum_{i=1}^{0} i = 0 = \frac{0(0+1)}{2}$
- Cas inductif : Supposons la propriété vraie pour n et montrons qu'elle est vraie pour $n+1$:

$$\begin{aligned} \sum_{i=1}^{n+1} i &= \left(\sum_{i=1}^{n} i\right) + (n+1) = \frac{n(n+1)}{2} + (n+1) \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

- Par induction, le propriété est vraie pour tout n.

Correction d'algorithmes récursifs par induction

- Propriété à montrer : l'algorithme est correct pour une instance quelconque du problème
- Instances du problème ordonnées par "taille" (taille du tableau, nombre de bits, un entier n, etc.)
- Cas de base de l'induction = cas de base de la récursion
- Cas inductif : on suppose que les appels récursifs sont corrects et on en déduit que l'appel courant est correct
- Terminaison : on montre que les appels récursifs se font sur des sous-problèmes (souvent trivial)

Exemple : FIBONACCI

```
FIBONACCI(n)
1  if n ≤ 1
2        return n
3  return FIBONACCI(n − 2) + FIBONACCI(n − 1)
```

Proposition : Pour tout n, FIBONACCI(n) renvoie F_n.

Démonstration :

- Cas de base : pour $n = 0$, FIBONACCI(n) renvoie $F_0 = 0$. Pour $n = 1$, FIBONACCI(n) renvoie $F_1 = 1$.
- Cas inductif :
  - Supposons $n \geq 2$ et que pour tout $0 \leq m < n$, FIBONACCI(m) renvoie F_m.
  - Pour $n \geq 2$, FIBONACCI(n) renvoie

$$\begin{aligned} & \text{FIBONACCI}(n-2) + \text{FIBONACCI}(n-1) \\ = \; & F_{n-2} + F_{n-1} \text{ (par hypothèse inductive)} \\ = \; & F_n. \end{aligned}$$

Exemple : merge sort

```
MERGE-SORT(A, p, r)
1  if p < r
2        q = ⌊(p+r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q + 1, r)
5        MERGE(A, p, q, r)
```

Proposition : Pour tout $1 \leq p \leq r \leq A.\mathit{length}$, MERGE-SORT$(A, p, r)$ trie le sous-tableau $A[p \,..\, r]$.

(On supposera que MERGE est correct mais il faudrait le démontrer par un invariant)

Exemple : merge sort

```
MERGE-SORT(A, p, r)
1  if p < r
2        q = ⌊(p+r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q + 1, r)
5        MERGE(A, p, q, r)
```

Démonstration :

- Cas de base : pour $r - p = 0$, MERGE-SORT(A, p, r) ne modifie pas A et donc $A[p] = A[q]$ est trivialement trié
- Cas inductif :
  - Supposons $r - p > 0$ et que pour tout $1 \leq p' \leq r' \leq A.\mathit{length}$ tels que $r' - p' < r - p$, MERGE-SORT(A, p', r') trie $A[p' \,..\, r']$
  - Les appels MERGE-SORT(A, p, q) et MERGE-SORT$(A, q + 1, r)$ sont corrects par hypothèse inductive (puisque $q - p < r - p$ et $r - q - 1 < r - p$)
  - En supposant MERGE correct, MERGE-SORT(A, p, r) est correct.

Conclusion sur la correction

- Preuves de correction :
  - Algorithmes itératifs : invariant (=induction)
  - Algorithmes récursifs : induction

- Malheureusement, il n'existe pas d'outil automatique pour vérifier la correction (et la terminaison) d'algorithmes
- Dans la suite, on ne présentera des invariants ou des preuves par induction que très sporadiquement lorsque ce sera nécessaire (cas non triviaux)

Plan

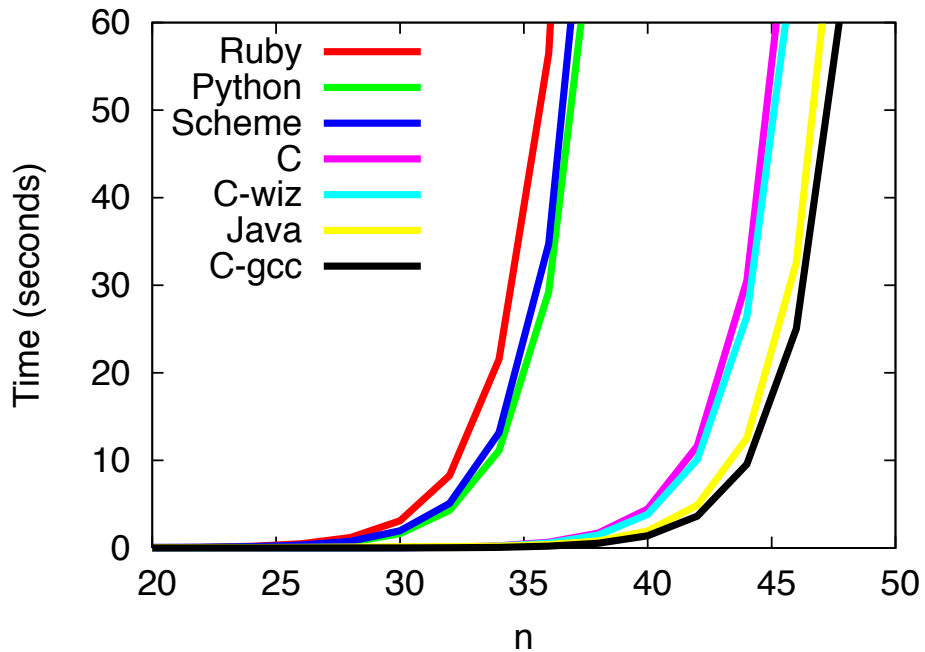
Performance d'un algorithme

- Plusieurs métriques possibles :
  - Longueur du programme (nombre de lignes)
  - Simplicité du code
  - Espace mémoire consommé
  - Temps de calcul
  - ...
- Les temps de calcul sont la plupart du temps utilisés
  - Ils peuvent être quantifiés et sont faciles à comparer
  - Souvent ce qui compte réellement
- Nous étudierons aussi l'espace mémoire consommé par nos algorithmes

Comment mesurer les temps d'exécution ?

Expérimentalement :

- On écrit un programme qui implémente l'algorithme et on l'exécute sur des données
- Problèmes :
  - Les temps de calcul vont dépendre de l'implémentation : CPU, OS, langage, compilateur, charge de la machine, OS, etc.
  - Sur quelles données tester l'algorithme ?



(Carzaniga)

Comment mesurer les temps d'exécution ?

Sur papier :

- Développer un modèle de machine ("Random-access machine", RAM) :
  - Opérations executées les unes après les autres (pas de parallélisme)
  - Opérations de base (addition, affectation, branchement, etc.) prennent un temps constant
  - Appel de sous-routines : temps de l'appel (constant) + temps de l'exécution de la sous-routine (calculé récursivement)
- Calculer les temps de calcul = sommer le temps d'exécution associé à chaque instruction du pseudo-code

- Le temps dépend de l'entrée (l'instance particulière du problème)
- On étudie généralement les temps de calcul en fonction de la "taille" de l'entrée
  - Généralement, le nombre de valeurs pour la décrire
  - Mais ça peut être autre chose (Ex : n pour FIBONACCI)

Analyse du tri par insertion

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

- t_j = nombre de fois que la condition du **while** est testée.
- Temps exécution $T(n)$ (pour un tableau de taille n) donné par :

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1)$$

$$+ c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

Différents types de complexité

- Même pour une taille fixée, la complexité peut dépendre de l'instance particulière
- Soit D_n l'ensemble des instances de taille n d'un problème et $T(i_n)$ le temps de calcul pour une instance $i_n \in D_n$.
- Sur quelles instances les performances d'un algorithme devraient être jugées :
  - Cas le plus favorable (best case) : $T(n) = \min\{T(i_n) | i_n \in D_n\}$
  - Cas le plus défavorable (worst case) : $T(n) = \max\{T(i_n) | i_n \in D_n\}$
  - Cas moyen (average case) : $T(n) = \sum_{i_n \in D_n} Pr(i_n) T(i_n)$ où $Pr(i_n)$ est la probabilité de rencontrer i_n
- On se focalise généralement sur le cas le plus défavorable
  - Donne une borne supérieure sur le temps d'exécution.
  - Le meilleur cas n'est pas représentatif et le cas moyen est difficile à calculer.

Analyse du tri par insertion

Meilleur cas :

- le tableau est trié $\Rightarrow t_j = 1$.
- Le temps de calcul devient :

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

- $T(n) = an + b \Rightarrow T(n)$ est une fonction linéaire de n

Analyse du tri par insertion

Pire cas :

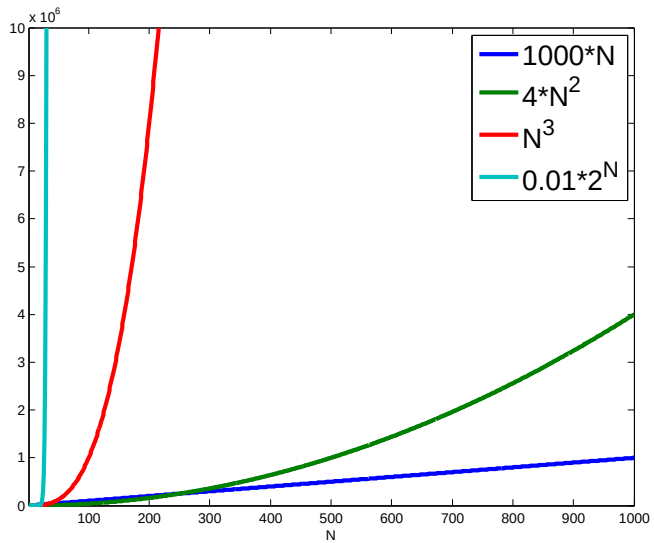
- le tableau est trié par ordre décroissant $\Rightarrow t_j = j$.
- Le temps de calcul devient :

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

- $T(n) = an^2 + bn + c \Rightarrow T(n)$ est une fonction quadratique de n

Analyse asymptotique

- On s'intéresse à la vitesse de croissance ("order of growth") de $T(n)$ lorsque n croît.
  - Tous les algorithmes sont rapides pour des petites valeurs de n
- On simplifie généralement $T(n)$:
  - en ne gardant que le terme dominant
    - Exemple : $T(n) = 10n^3 + n^2 + 40n + 800$
    - T(1000)=1000001040800, $10 \cdot 1000^3 = 10000000000$
  - en ignorant le coefficient du terme dominant
    - Asymptotiquement, ça n'affecte pas l'ordre relatif



- Exemple : Tri par insertion : $T(n) = an^2 + bn + c \rightarrow n^2$.

Pourquoi est-ce important ?

- Supposons qu'on puisse traiter une opération de base en $1\mu s$.
- Temps d'exécution pour différentes valeurs de n

T(n)	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
n	$10\mu s$	$0.1ms$	$1ms$	$10ms$
$400n$	$4ms$	$40ms$	$0.4s$	$4s$
$2n^2$	$200\mu s$	$20ms$	$2s$	$3.3m$
n^4	$10ms$	$100s$	$\sim$ 11.5 jours	317 années
2^n	$1ms$	4×10^{16} années	3.4×10^{287} années	...

(Dupont)

Pourquoi est-ce important ?

- Taille maximale du problème qu'on peut traiter en un temps donné :

T(n)	en 1 seconde	en 1 minute	en 1 heure
n	1×10^6	6×10^7	3.6×10^9
$400n$	2500	150000	9×10^6
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

- Si m est la taille maximale que l'on peut traiter en un temps donné, que devient cette valeur si on reçoit une machine 256 fois plus puissante ?

T(n)	Temps
n	$256m$
$400n$	$256m$
$2n^2$	$16m$
n^4	$4m$
2^n	$m + 8$

(Dupont)

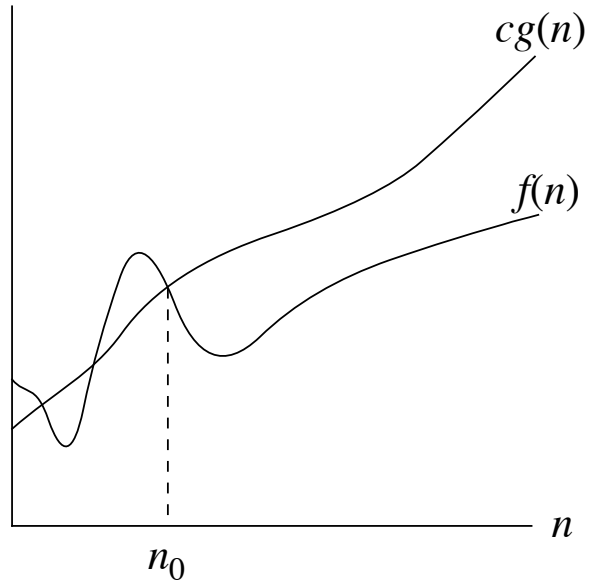
Notations asymptotiques

- Permettent de caractériser le taux de croissance de fonctions $f : \mathbb{N} \to \mathbb{R}^+$

- Trois notations :
  - Grand-O : $f(n) \in O(g(n)) \approx f(n) \leq g(n)$
  - Grand-Omega : $f(n) \in \Omega(g(n)) \approx f(n) \geq g(n)$
  - Grand-Theta : $f(n) \in \Theta(g(n)) \approx f(n) = g(n)$

Notation grand-O

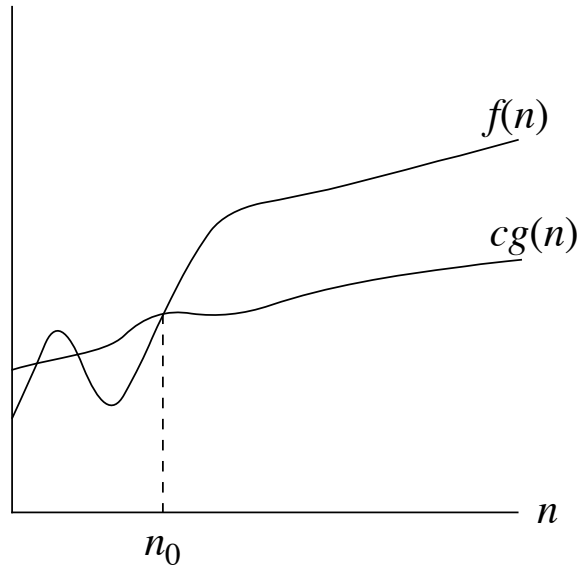
$$O(g(n)) = \{f(n) | \exists c > 0, \exists n_0 \geq 1 \text{ tels que } 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}$$



- $f(n) \in O(g(n)) \Rightarrow g(n)$ est une borne supérieure asymptotique pour $f(n)$.
- Par abus de notation, on écrira aussi : $f(n) = O(g(n))$.

Notation grand-Omega

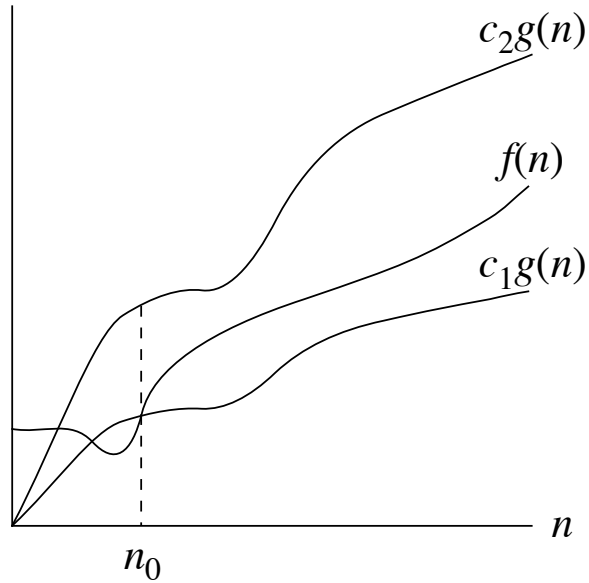
$$\Omega(g(n)) = \{f(n) | \exists c > 0, \exists n_0 \geq 1 \text{ tels que } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$$



- $f(n) \in \Omega(g(n)) \Rightarrow g(n)$ est une borne inférieure asymptotique pour $f(n)$.
- Par abus de notation, on écrira aussi : $f(n) = \Omega(g(n))$.

Notation grand-Theta

$$\begin{aligned}\Theta(g(n)) &= \{f(n) | \exists c_1, c_2 > 0, \exists n_0 \geq 1 \\ &\quad \text{tels que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}\end{aligned}$$



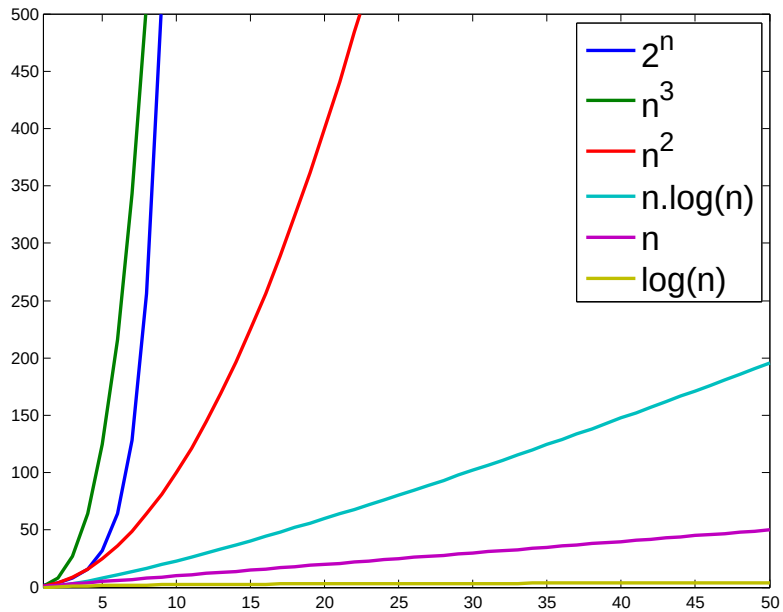
- $f(n) \in \Theta(g(n)) \Rightarrow g(n)$ est une borne serrée (“tight”) asymptotique pour $f(n)$.
- Par abus de notation, on écrira aussi : $f(n) = \Theta(g(n))$.

Exemples

- $3n^5 - 16n + 2 \in O(n^5) ? \in O(n) ? \in O(n^{17}) ?$
- $3n^5 - 16n + 2 \in \Omega(n^5) ? \in \Omega(n) ? \in \Omega(n^{17}) ?$
- $3n^5 - 16n + 2 \in \Theta(n^5) ? \in \Theta(n) ? \in \Theta(n^{17}) ?$
- $2^n + 100n^6 + n \in O(2^n) ? \in \Theta(3^n) ? \in \Omega(n^7) ?$

- Classes de complexité :

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^{a>1}) \subset O(2^n)$$



Quelques propriétés

- $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$
- $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ et $f(n) \in \Omega(g(n))$
- $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$

- Si $f(n) \in O(g(n))$, alors pour tout $k \in \mathbb{N}$, on a $k \cdot f(n) \in O(g(n))$
  - Exemple : $\log_a(n) \in O(\log_b(n))$, $a^{n+b} \in O(a^n)$
- Si $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$, alors
  $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$ et $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$
  - Exemple : $\sum_{i=1}^{m} a_i n^i \in O(n^m)$
- Si $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$, alors
  $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$

D'autres notations (pour information)

- Equivalence asymptotique : $f(n) \sim g(n)$ ssi $\lim_{n\to\infty} f(n)/g(n) = 1$
  - f et g sont asymptotiquement équivalents.
  - Permet de comparer deux fonctions en prenant en compte la constante
- Petit-o :

$$o(g(n)) = \{f(n) | \forall c > 0, \exists n_0 \geq 1 \text{ tels que } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$$

  - f est négligeable devant g asymptotiquement
  - $\approx f(n) < g(n)$
- Petit-ω :

$$\omega(g(n)) = \{f(n) | \forall c > 0, \exists n_0 \geq 1 \text{ tels que } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$$

  - f domine g asymptotiquement
  - $\approx f(n) > g(n)$

Complexité d'un algorithme

- On utilise les notations asymptotiques pour caractériser la complexité d'un algorithme.
- Il faut préciser de quelle complexité on parle : générale, au pire cas, au meilleur cas, en moyenne...
- La notation grand-O est de loin la plus utilisée
  - $f(n) \in O(g(n))$ sous-entend généralement que $O(g(n))$ est le plus petit sous-ensemble qui contient $f(n)$ et que $g(n)$ est la plus concise possible
  - Exemple : $n^3 + 100n^2 - n \in O(n^3) = O(n^3 + n^2) \subset O(n^4) \subset O(2^n)$
- Idéalement, les notations O et Ω devraient être limitées au cas où on n'a pas de borne serrée.

Complexité d'un algorithme

Exemples :

- On dira :
  "La complexité au pire cas du tri par insertion est $\Theta(n^2)$"
  plutôt que
  "La complexité au pire cas du tri par insertion est $O(n^2)$"
  ou "La complexité du tri par insertion est $O(n^2)$"
- On dira
  "La complexité au meilleur cas du tri par insertion est $\Theta(n)$"
  plutôt que
  "La complexité au meilleur cas du tri par insertion est $\Omega(n)$"
  ou "La complexité du tri par insertion est $\Omega(n)$"
- Par contre, on dira "La complexité de FIBONACCI est $\Omega(1.4^n)$", car on n'a pas de borne plus précise à ce stade.

Complexité d'un problème

- Les notations asymptotiques servent aussi à caractériser la complexité d'un problème
  - Un problème est $O(g(n))$ s'il existe un algorithme pour le résoudre dont le pire cas est $O(g(n))$
  - Un problème est $\Omega(g(n))$ si la complexité dans le pire cas de tout algorithme qui le résoud est forcément $\Omega(g(n))$.
  - Un problème est $\Theta(g(n))$ s'il est $O(g(n))$ et $\Omega(g(n))$
- Exemple du problème de tri :
  - Le problème du tri est $O(n \log n)$ (voir plus loin)
  - On peut montrer facilement que le problème du tri est $\Omega(n)$ (voir le transparent suivant)
  - On montrera plus tard que le problème de tri est en fait $\Omega(n \log n)$ et donc qu'il est $\Theta(n \log n)$.
- Exercice : montrez que la recherche du maximum dans un tableau est $\Theta(n)$

Le problème du tri est $\Omega(n)$

Preuve par l'absurde (ou par contraposition) :

- Supposons qu'il existe un algorithme moins que $O(n)$ pour résoudre le problème du tri
- Cet algorithme ne peut pas parcourir tous les éléments du tableau, sinon il serait au moins $O(n)$
- Il y a donc au moins un élément du tableau qui n'est pas vu par cet algorithme
- Il existe donc des instances de tableau qui ne seront pas triées correctement par cet algorithme
- Il n'y a donc pas d'algorithme plus rapide que $O(n)$ pour le tri.

Comment calculer la complexité en pratique ?

Quelques règles pour les algorithmes itératifs :

- Affectation, accès à un tableau, opérations arithmétiques, appel de fonction : $O(1)$
- Instruction If-Then-Else : O(complexité max des deux branches)
- Séquence d'opérations : l'opération la plus couteuse domine (règle de la somme)
- Boucle simple : $O(nf(n))$ si le corps de la boucle est $O(f(n))$

Comment calculer la complexité en pratique ?

- Double boucle complète : $O(n^2 f(n))$ où $f(n)$ est la complexité du corps de la boucle
- Boucles incrémentales : $O(n^2)$ (si corps $O(1)$)

```
for i = 1 to n
    for j = 1 to i
        ...
```

- Boucles avec un incrément exponentiel : $O(\log n)$ (si corps $O(1)$)

```
i = 1
while i ≤ n
    ...
    i = 2i
```

Exemple :

PREFIXAVERAGES(X) :

- **Entrée :** tableau X de taille n
- **Sortie :** tableau A de taille n tel que $A[i] = \frac{\sum_{j=1}^{i} X[j]}{i}$

```
PREFIXAVERAGES(X)
1  for i = 1 to X.length
2        a = 0
3        for j = 1 to i
4              a = a + X[j]
5        A[i] = a/i
6  return A
```

Complexité : $\Theta(n^2)$

```
PREFIXAVERAGES2(X)
1  s = 0
2  for i = 1 to X.length
3        s = s + X[i]
4        A[i] = s/i
5  return A
```

Complexité : $\Theta(n)$

Complexité d'algorithmes récursifs

- La complexité d'algorithme récursif mène généralement à une équation de récurrence

```
FACTORIAL(n)
1  if n == 0
2        return 1
3  return n · FACTORIAL(n − 1)
```

$$\begin{aligned} T(0) &= c_0 \\ T(n) &= T(n-1) + c_1 \\ &= c_1 n + c_0 \end{aligned}$$

$$\Rightarrow T(n) \in \Theta(n)$$

- La résolution de cette équation n'est pas toujours triviale

Autres exemples

- Fibonacci :

$$\begin{aligned} T(1) &= c_0 \\ T(n) &= T(n-1) + T(n-2) + c_1 \text{ pour } n > 1 \end{aligned}$$

- Tri par fusion :

$$\begin{aligned} T(1) &= c_0 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + c_1 n + c_2 \text{ pour } n > 1 \end{aligned}$$

- Tour de Hanoï (cf. INFO0902)

$$\begin{aligned} T(1) &= c_0 \\ T(n) &= 2T(n-1) + c_1 \text{ pour } n > 1 \end{aligned}$$

Quelques techniques de résolution de récurrences

- Méthodes “manuelles” :
  - “Plug-and-chug” (téléscopage)
  - Arbres de récursion
  - ...
- Méthodes systématiques :
  - Master theorem (récurrences diviser-pour-régner)
  - Equations caractéristiques (récurrences linéaires, pas dans ce cours)
  - ...

Méthode “Plug-and-Chug” (force brute)

(aussi appelée téléscopage ou méthode des facteurs sommants)

1. “Plug” (appliquer l’équation récurrente) et “Chug” (simplifier)

$$\begin{aligned} T(n) &= c_1 + 2T(n-1) \\ &= c_1 + 2(c_1 + 2T(n-2) \\ &= c_1 + 2c_1 + 4T(n-2) \\ &= c_1 + 2 + 4(c_1 + 2T(n-3)) \\ &= c_1 + 2c_1 + 4c_1 + 8T(n-3) \\ &= \dots \end{aligned}$$

Remarque : Il faut simplifier *avec modération*.

2. Identifier et vérifier un "pattern"
   - Identification :

$$T(n) = c_1 + 2c_1 + 4c_1 + \cdots + 2^{i-1}c_1 + 2^i T(n-i)$$

   - Vérification en développant une étape supplémentaire :

$$\begin{aligned} T(n) &= c_1 + 2c_1 + 4c_1 + \cdots + 2^{i-1}c_1 + 2^i(c_1 + 2T(n-(i+1))) \\ &= c_1 + 2c_1 + 4c_1 + \cdots + 2^{i-1}c_1 + 2^i c_1 + 2^{i+1} T(n-(i+1)) \end{aligned}$$

3. Exprimer le $n^{\text{ème}}$ terme en fonction des termes précédents
   En posant $i = n-1$, on obtient

$$\begin{aligned} T(n) &= c_1 + 2c_1 + 4c_1 + \cdots + 2^{n-2}c_1 + 2^{n-1} T(1) \\ &= c_1(1 + 2 + 4 + \cdots + 2^{n-2}) + 2^{n-1}c_0 \end{aligned}$$

4. Trouver une solution analytique pour le $n^{\text{ème}}$ terme

$$
\begin{aligned}
T(n) &= c_1(1+2+4+\cdots+2^{n-2})+2^{n-1}c_0 \\
&= c_1\Big(\sum_{i=0}^{n-2}2^i\Big)+2^{n-1}c_0 \\
&= c_1\frac{1-2^{n-1}}{1-2}+2^{n-1}c_0 \\
&= (c_0+c_1)2^{n-1}-c_1 \\
&\in \Theta(2^n)
\end{aligned}
$$

Tri par fusion

Appliquons le “plug-and-chug” à la récurrence (simplifiée) du tri par fusion *dans le cas où* $n = 2^k$:

$$\begin{aligned} T(1) &= c \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn = 2T(n/2) + cn \text{ pour } n > 1 \end{aligned}$$

- Pattern :

$$\begin{aligned} T(n) &= 2^i T(n/2^i) + cn + \ldots + cn \\ &= 2^i T(n/2^i) + icn \end{aligned}$$

- En posant $i = k$ et en utilisant $k = \log_2 n$:

$$\begin{aligned} T(n) &= 2^k T(n/2^k) + kcn \\ &= nT(1) + cn \log_2 n \\ &= cn \log_2 n + cn \\ &\in \Theta(n \log n) \end{aligned}$$

Arbres de récursion

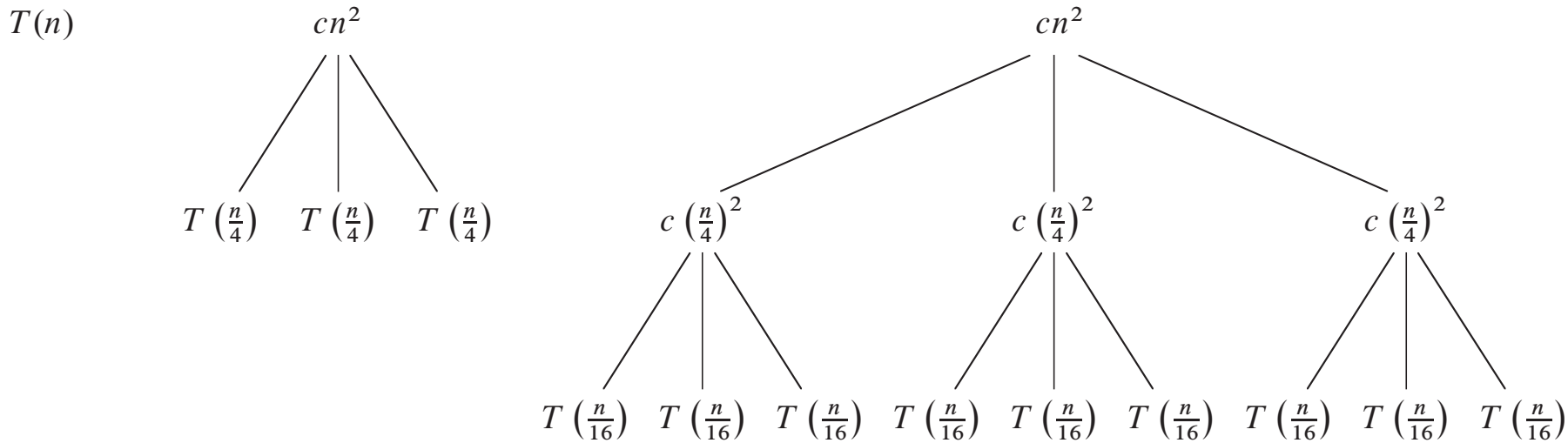
Approche graphique pour *deviner* une solution analytique (ou une borne asymptotique) à une récurrence.

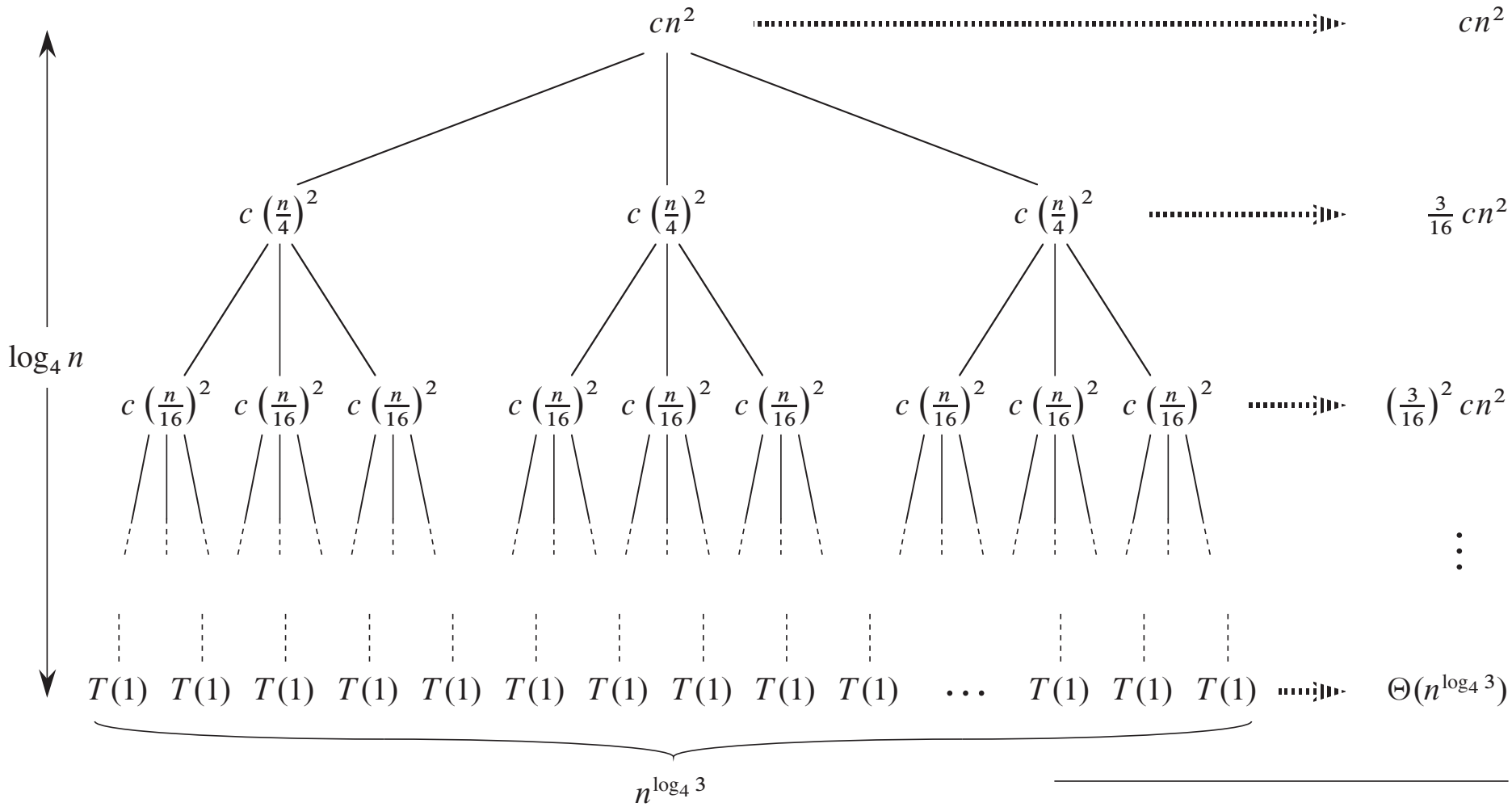
La solution devra toujours être démontrée ensuite (par induction).

Illustration sur le tri par fusion : Illustration sur la récurrence suivante :

- $T(1) = a$
- $T(n) = 3T(n/4) + cn^2$ (Pour $n > 1$)

(Introduction to algorithms, Cormen et al.)


$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$
cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$


cn^2
cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$\frac{3}{16}\,cn^2$
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$

- Le coût total est la somme du coût de chaque niveau de l'arbre :

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + an^{\log_4 3} \\ &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + an^{\log_4 3} \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + an^{\log_4 3} \\ &= \frac{1}{1-(3/16)} cn^2 + an^{\log_4 3} \\ &\in O(n^2) \end{aligned}$$

(à vérifier par induction)

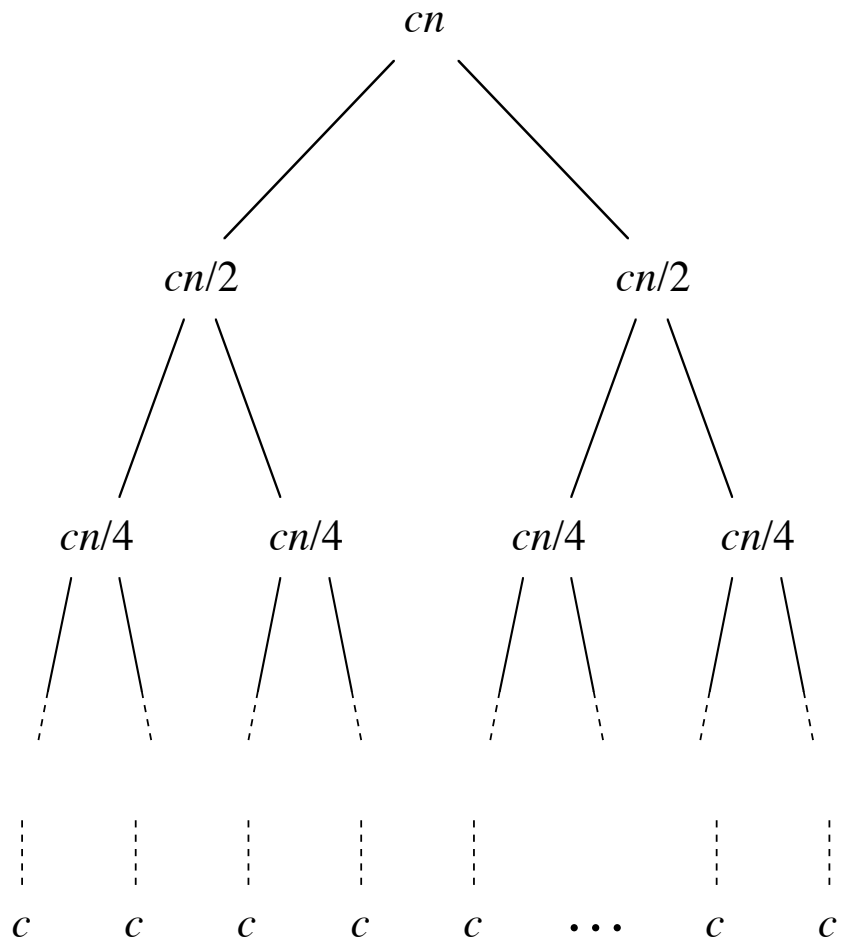
- Comme le coût de la racine est cn^2, on a aussi $T(n) \in \Omega(n^2)$ et donc $T(n) \in \Theta(n^2)$.

Analyse du tri par fusion

- Récurrence simplifiée :

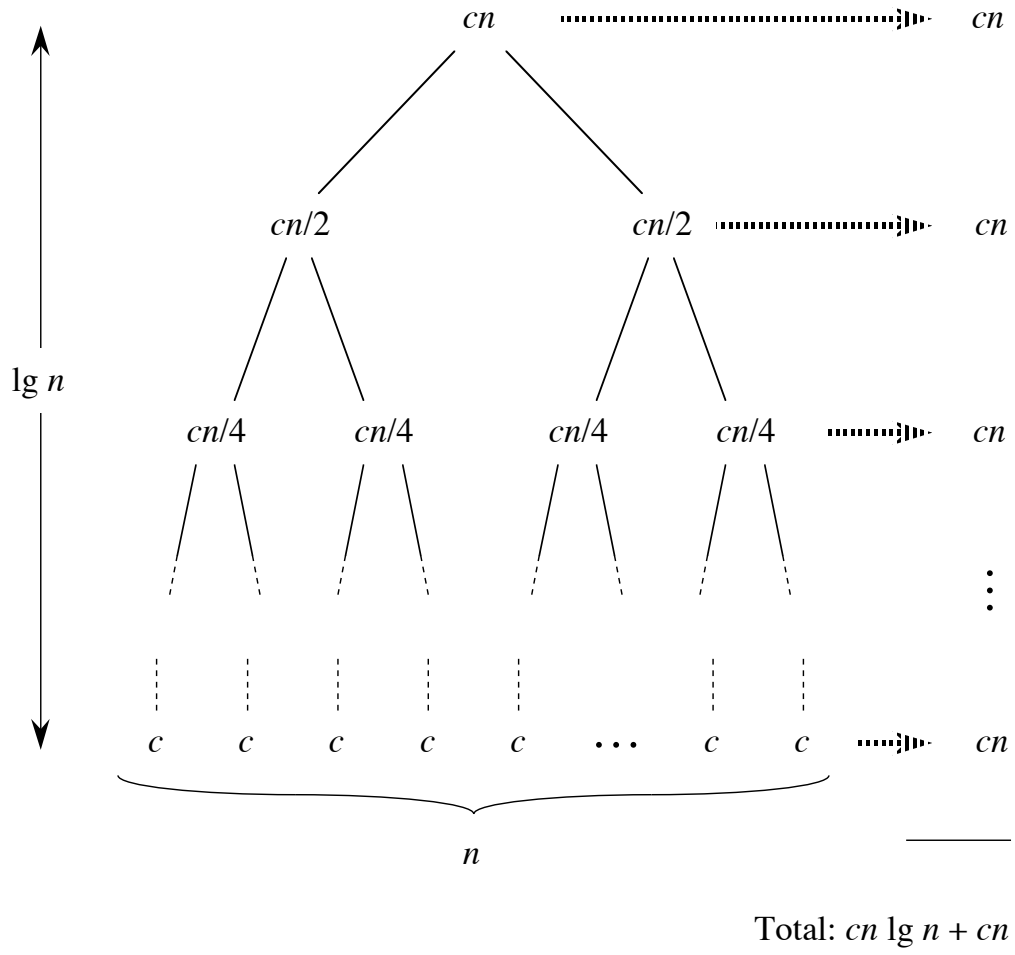
$$\begin{aligned} T(1) &= c \\ T(n) &= 2T(n/2) + cn \end{aligned}$$

- On peut représenter la récurrence par un arbre de récursion
- La complexité est la somme du coût de chaque noeud



Analyse du tri par fusion

- Chaque niveau a un coût cn
- En supposant que n est une puissance de 2, il y a $\log_2 n + 1$ niveaux
- Le coût total est $cn \log_2 n + cn \in \Theta(n \log n)$



Méthodes systématiques

Les approches “Plug-and-Chug” et par arbre de récursion peuvent être fastidieuses à utiliser.

Il existe des méthodes plus systématiques pour résoudre des récurrences particulières :

- Récurrences linéaires d'ordre $k \geq 1$ à coefficients constants :

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \ldots + c_k T(n-k) + g(n)$$

  On peut calculer une solution analytique exacte à ces récurrences (pas vu dans ce cours)
- Récurrences “diviser-pour-régner” : On peut dériver une borne asymptotique via un théorème.

Récurrence générale “diviser-pour-régner”

Définition : Un récurrence “diviser-pour-régner” est une récurrence de la forme :

$$T_n = \sum_{i=1}^{k} a_i T(b_i n) + g(n),$$

où $a_1, \ldots, a_k$ sont des constantes positives, $b_1, \ldots, b_k$ sont des constantes comprises entre 0 et 1 et $g(n)$ est une fonction non négative.

Exemple : $k = 1$, $a_1 = 2$, $b_1 = 1/2$ et $g(n) = cn$ correspond au tri par fusion

Sous certaines conditions, un théorème (“master theorem”) fournit des bornes asymptotiques sur les récurrences de ce type.

Master theorem

Théorème : Soit la récurrence suivante :

$$\begin{cases} T(n) = c & \text{si } n < d \\ T(n) = aT(\frac{n}{b}) + f(n) & \text{si } n \geq d \end{cases}$$

où $d \geq 1$ est une constantes entière, $a > 0$, $c > 0$ et $b > 1$ sont des constantes réelles, et $f(n)$ est une fonction positive pour $n \geq d$.

1. Si $f(n) \in O(n^{\log_b a - \epsilon})$ pour un $\epsilon > 0$, alors $T(n) \in \Theta(n^{\log_b a})$
2. Si $f(n) \in \Theta(n^{\log_b a})$, alors $T(n) \in \Theta(n^{\log_b a} \log n)$.
3. Si $f(n) \in \Omega(n^{\log_b a + \epsilon})$ pour un $\epsilon > 0$ et s'il existe $\delta < 1$ tel que $af(\frac{n}{b}) \leq \delta f(n)$ pour un n assez grand, alors $T(n) \in \Theta(f(n))$.

(Introduction to algorithms, Cormen et al.)

NB : les bornes ne dépendent pas de c et d. $\frac{n}{b}$ peut être interprété soit comme $\lfloor \frac{n}{b} \rfloor$, soit comme $\lceil \frac{n}{b} \rceil$.

Applications

Tri par fusion :

$$T(n) = 2T(n/2) + cn.$$

- T(n) satisfait aux conditions du théorème avec $a = 2$ et $b = 2$.
- $\log_b a = \log_2 2 = 1 \Rightarrow f(n) \in \Theta(n^1)$
- Par le deuxième cas du théorème, on a $T(n) \in \Theta(n \log n)$.

Méthode de Strassen (multiplication de matrice) :

$$T(n) = 7T(n/2) + \Theta(n^2).$$

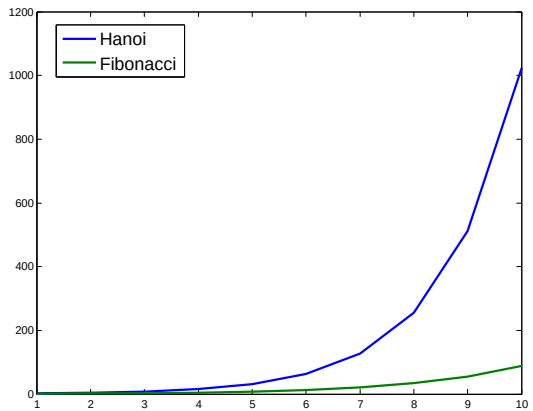
- $T(n)$ satisfait aux conditions du théorème avec $a = 7$, $b = 2$.
- $\log_b a = \log_2 7 = 2.807... \Rightarrow f(n) \in O(n^{\log_2 7 - \epsilon})$ avec $\epsilon = 0.8$.
- Par le premier cas du théorème, on a :

$$T(n) = \Theta(n^{\log_b a}) = O(n^{2.807\ldots}).$$

Comparaisons de récurrences : linéaires versus “d-p-r”

	Récurrence	Solution
Tours de Hanoï	$T_n = 2T_{n-1} + 1$	$T_n \sim 2^n$
Tours de Hanoï 2	$T_n = 2T_{n-1} + n$	$T_n \sim 2 \cdot 2^n$
Algo rapide	$T_n = 2T_{n/2} + 1$	$T_n \sim n$
Tri par fusion	$T_n = 2T_{n/2} + n - 1$	$T_n \sim n \log n$
Fibonacci	$T_n = T_{n-1} + T_{n-2}$	$T_n \sim (1.618\ldots)^{n+1}/\sqrt{5}$

- Récurrences “Diviser pour régner” généralement polynomiales
- Récurrences linéaires généralement exponentielles
- Générer des sous-problèmes petits est beaucoup plus important que réduire la complexité du terme non homogène
  - Tri par fusion et Fibanocci sont exponentiellement plus rapides que les tours de Hanoï



Comparaisons de récurrences : nombre de sous-problèmes

Récurrences linéaires :

$$T_n = 2T_{n-1} + 1 \quad \Rightarrow \quad T_n = \Theta(2^n)$$
$$T_n = 3T_{n-1} + 1 \quad \Rightarrow \quad T_n = \Theta(3^n)$$

Augmentation exponentielle des temps de calcul quand on passe de 2 à 3 sous-problèmes.

Récurrence “diviser-pour-régner” :

$$\begin{aligned} T_1 &= 0 \\ T_n &= aT_{n/2} + n - 1 \end{aligned}$$

Par le master théorème, on a :

$$T_n = \begin{cases} \Theta(n) & \text{pour } a < 2 \\ \Theta(n \log_2 n) & \text{pour } a = 2 \\ \Theta(n^{\log_2 a}) & \text{pour } a > 2. \end{cases}$$

La solution est complètement différente entre $a = 1.99$ et $a = 2.01$.

Remarques

Limitations de l'analyse asymptotique

- Les facteurs constants ont de l'importance pour des problèmes de petite taille
  - Le tri par insertion est plus rapide que le tri par fusion pour n petit
- Deux algorithmes de même complexité (grand-O) peuvent avoir des propriétés très différentes
  - Le tri par insertion est en pratique beaucoup plus efficace que le tri par sélection sur des tableaux presque triés

Complexité en espace

- S'étudie de la même manière, avec les mêmes notations
- Elle est bornée par la complexité en temps (pourquoi ?)

Ce qu'on a vu

- Correction d'algorithmes itératifs (par invariant) et récursifs (par induction)
- Notions de complexité algorithmique
- Notations asymptotiques
- Calcul de complexité d'algorithmes itératifs et récursifs

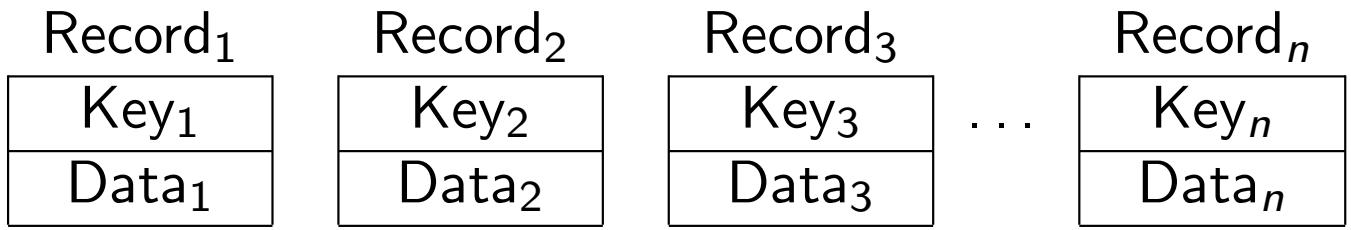
Partie 3

Algorithmes de tri

Plan

Tri

- Un des problèmes algorithmiques les plus fondamentaux.
- En général, on veut trier des enregistrements avec une clé et des données attachées.



- Ici, on va ignorer ces données satellites et se focaliser sur les algorithmes de tri

- Le problème de tri :
  - Entrée : une séquence de n nombres $\langle a_1, a_2, \ldots, a_n \rangle$
  - Sortie : une permutation de la séquence de départ $\langle a'_1, a'_2, \ldots, a'_n \rangle$ telle que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Applications

Applications innombrables :

- Tri des mails selon leur ancienneté
- Tri des résultats de requête dans un moteur de recherche
- Tri des facettes des objets pour l'affichage dans les jeux 3D
- Gestion des opérations bancaires
- ...

Le tri sert aussi de brique de base pour d'autres algorithmes :

- Recherche binaire dans un tableau trié
- Recherche des éléments dupliqués dans une liste
- Recherche du $k$ème élément le plus grand dans une liste
- ...

Des études montrent qu'environ 25% du temps CPU des ordinateurs est utilisé pour trier

Différents types de tri

- **Tri interne :** tri en mémoire centrale. **Tris externes :** données sur un disque externe.
- **Tri de tableau :** tri qui trie un tableau. Extensible à toutes structures de données offrant un accès en temps (quasi) constant à ses éléments.
- **Tri générique :** peut trier n'importe quel type d'objets pour autant qu'on puisse comparer ces objets.
- **Tri comparatif :** basé sur la comparaison entre les éléments (clés)

Différents types de tri

- **Tri itératif :** basé sur un ou plusieurs parcours itératifs du tableau
- **Tri récursif :** basé sur une procédure récursive
- **Tri en place :** modifie directement la structure qu'il est en train de trier. Ne nécessite qu'une quantité très limitée de mémoire supplémentaire.
- **Tri stable :** conserve l'ordre relatif des éléments égaux (au sens de la méthode de comparaison).

Jusqu'ici

Algorithme	*Complexité*			*En place ?*
	Pire	*Moyenne*	*Meilleure*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	oui
Bubble-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	non
??		$\Theta(n \log n)$		oui
??	$\Theta(n \log n)$			oui

Tri rapide

- *Quicksort* en anglais
- Inventé par Hoare en 1960
- Dans le top 10 des algorithmes du 20-ième siècle (SIAM)
- L'exemple le plus célèbre de la technique du "diviser pour régner"
- Tri en place, comme tri par insertion, et contrairement au tri par fusion
- Complexité : $\Theta(n^2)$ dans le pire des cas, $\Theta(n \log n)$ en moyenne

QuickSort : principe

Pour trier un sous-tableau $A[p\,..\,r]$:

- Partitionner $A[p\,..\,r]$ en deux sous-tableaux : $A[p..q-1]$ et $A[q+1\,..\,r]$ tels que tout élément de $A[p\,..\,q-1]$ est $\leq A[q]$ et $A[q] <$ à tout élément de $A[q+1\,..\,r]$.

  (diviser)

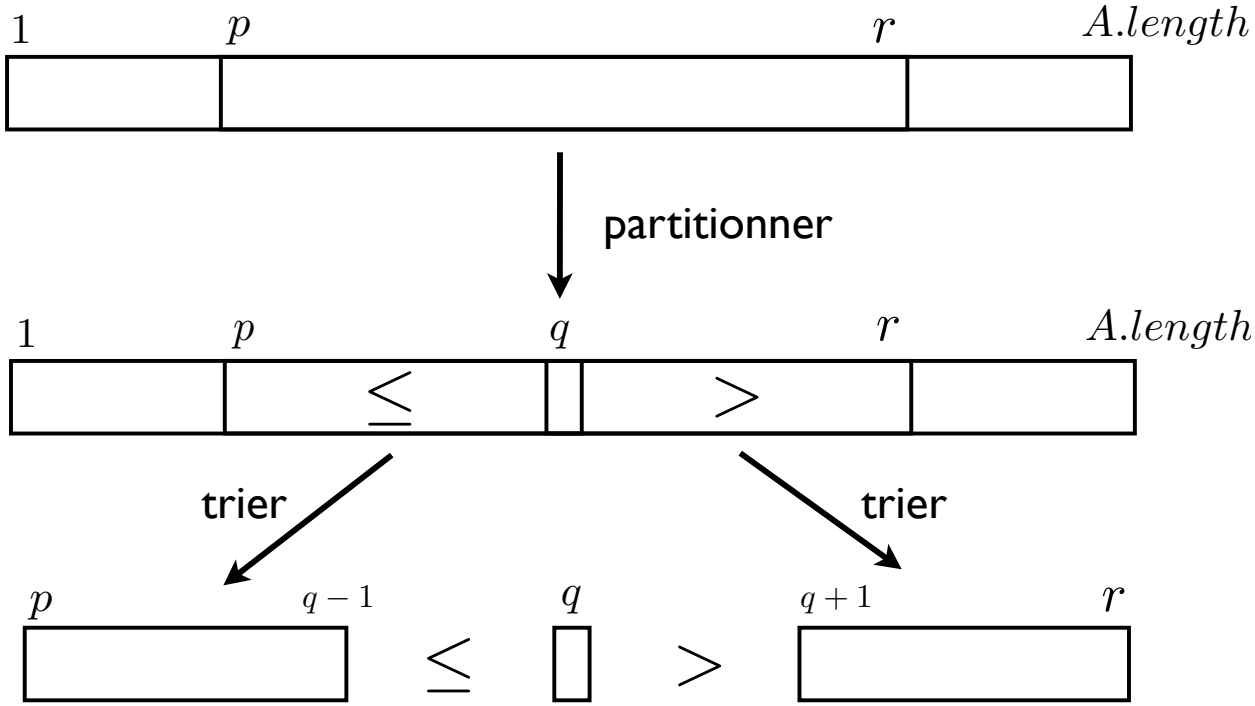
- Appeler récursivement l'algorithme pour trier $A[p\,..\,q-1]$ et $A[q+1\,..\,r]$

  (régner)

Remarques :

- $A[q]$ est appelé le "pivot"
- Par rapport au tri par fusion, il n'y a pas d'opération de combinaison

QuickSort : principe

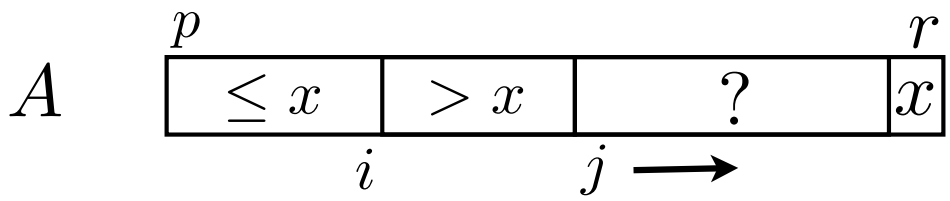


QuickSort Algorithm

```
QuickSort(A, p, r)
1  if p < r
2        q = Partition(A, p, r)
3        QuickSort(A, p, q − 1)
4        QuickSort(A, q + 1, r)
```

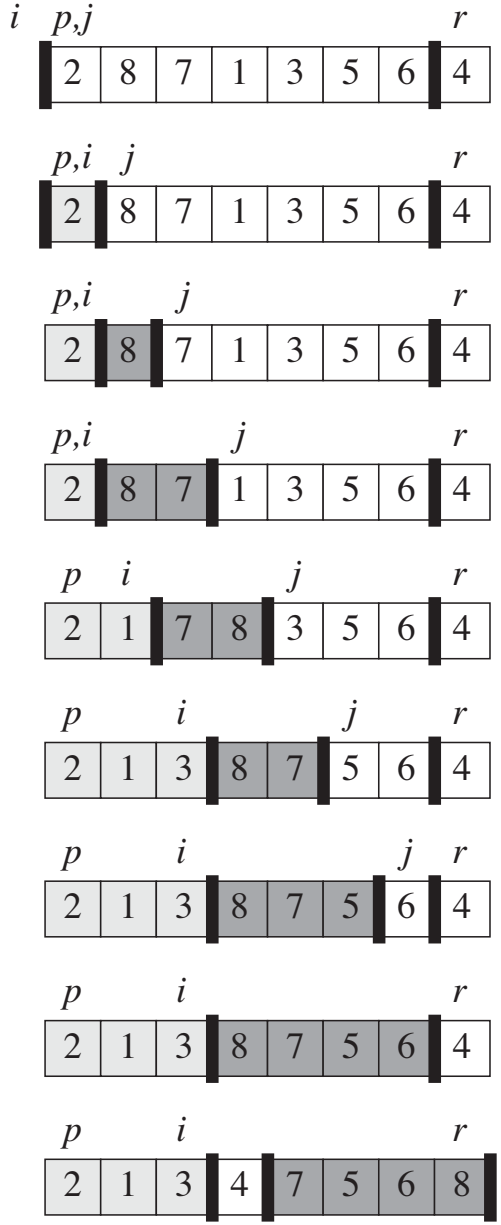
Appel initial : $\textsc{Quicksort}(A, 1, A.length)$

Partition : principe



- On sélectionne le dernier élément $A[r]$ comme le **pivot**
- On initialise un indice i à $p - 1$
- On parcourt le tableau de gauche à droite avec un indice $j = p \textbf{ to } r - 1$
- Si $A[j] \leq A[r]$, on incrémente i et on échange $A[j]$ et $A[i]$
- En sortie de boucle, on échange $A[i + 1]$ et $A[r]$ et on renvoie $i + 1$

Partition : illustration



$A[r]$ est le pivot
$A[p \,..\, i]$ contient des éléments $\leq$ au pivot
$A[i+1 \,..\, j-1]$ contient des éléments $>$ que le pivot
$A[j \,..\, r-1]$ est la partie du tableau non encore examinée

Partition : pseudo-code

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        if A[j] ≤ x
5              i = i + 1
6              swap(A[i], A[j])
7  swap(A[i + 1], A[r])
8  return i + 1
```

Partition : correction

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            swap(A[i], A[j])
7  swap(A[i + 1], A[r])
8  return i + 1
```

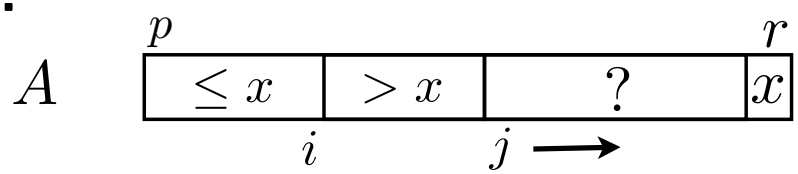
Pré-condition : $\{A[p\,..\,r],$ un tableau de nombres$\}$
Post-condition : $\{A[p\,..\,i] \leq A[i+1] < A[i+2\,..\,r]\}$
Invariant :

1. Les valeurs dans $A[p\,..\,i]$ sont $\leq$ au pivot
2. Les valeurs dans $A[i+1\,..\,j-1]$ sont $>$ que le pivot
3. $A[r] = pivot$

Partition : correction

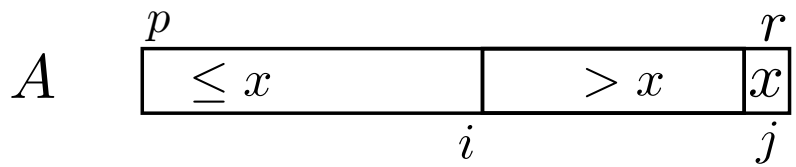
Avant la boucle : $i = p - 1$ et $j = p \Rightarrow A[p\,..\,i]$ et $A[i+1\,..\,j-1]$ sont vides

Pendant la boucle :



- Si $A[j] > x$, on incrémente juste j. Donc si l'invariant était vrai avant l'exécution du corps, il reste vrai après.
- Si $A[j] \leq x$, on échange $A[j]$ et $A[i+1]$ et i et j sont incrémentés. L'invariant reste donc vérifié également

Après la boucle :



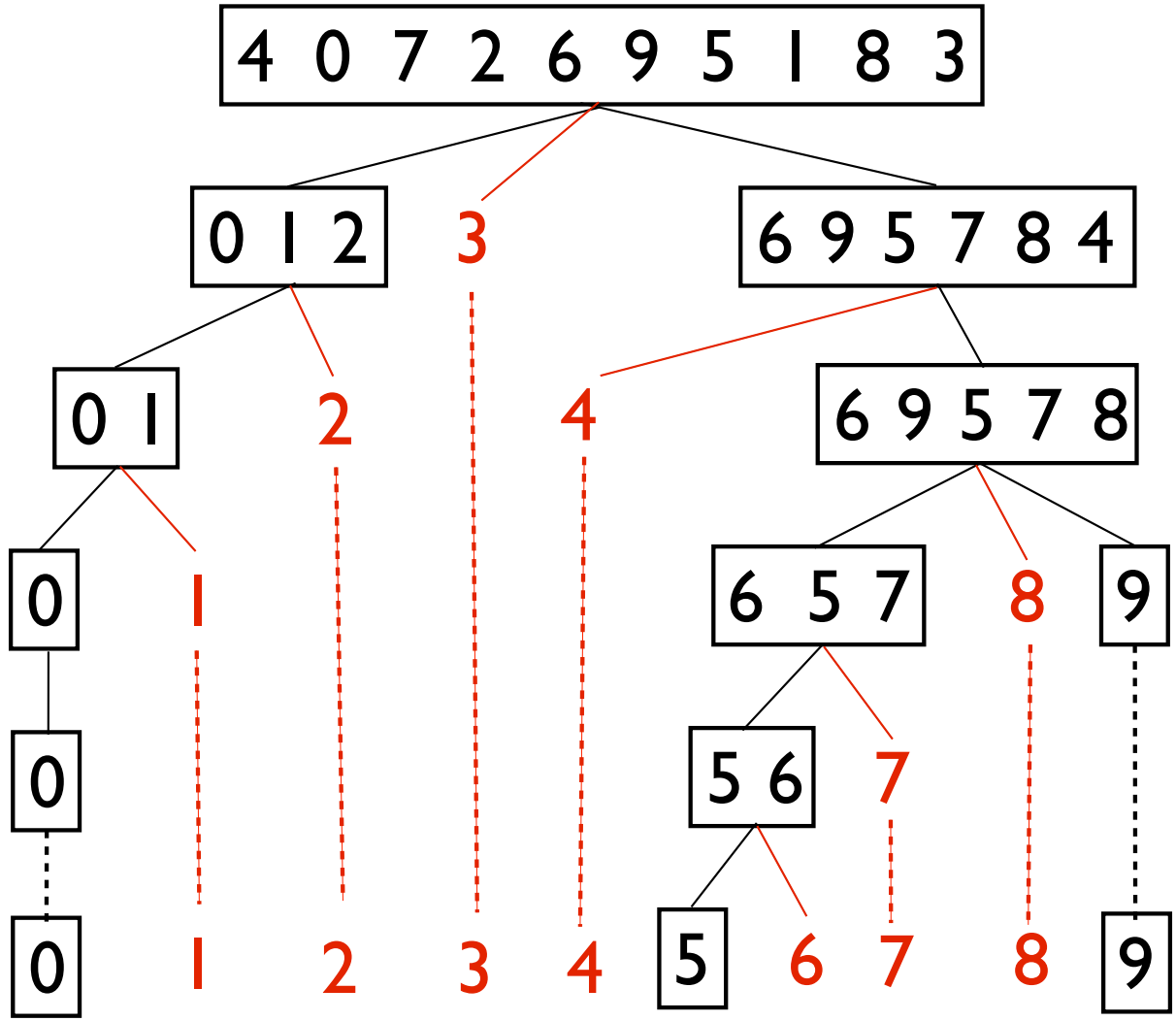
En sortie de boucle, l'invariant est vérifié et on a $j = r$. Echanger $A[i+1]$ et $A[r]$ établit la post-condition.

Algorithme complet

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            swap(A[i], A[j])
7  swap(A[i + 1], A[r])
8  return i + 1
```

```
QUICKSORT(A, p, r)
1  if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

Illustration



Complexité de Partition

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        if A[j] ≤ x
5              i = i + 1
6              swap(A[i], A[j])
7  swap(A[i + 1], A[r])
8  return i + 1
```

$$T(n) = \Theta(n)$$

Complexité de QuickSort

```
QuickSort(A, p, r)
1  if p < r
2        q = Partition(A, p, r)
3        QuickSort(A, p, q − 1)
4        QuickSort(A, q + 1, r)
```

- Pire cas : *(quand se produit-il ?)*
  - $q = p$ ou $q = r$
  - Le partitionnement transforme un problème de taille n en un problème de taille $n - 1$

$$T(n) = T(n-1) + \Theta(n)$$

  - Même complexité que le tri par insertion :

$$T(n) = \Theta(n^2)$$

Complexité de QuickSort

```
QuickSort(A, begin, end)
1  if begin < end
2        q = Partition(A, begin, end)
3        QuickSort(A, begin, q − 1)
4        QuickSort(A, q + 1, end)
```

- Meilleur cas :
  - $q = \lfloor n/2 \rfloor$
  - Le partitionnement transforme un problème de taille n en deux problèmes de taille $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor - 1$ respectivement

$$T(n) = 2T(n/2) + \Theta(n)$$

  - Même complexité que le tri par fusion :

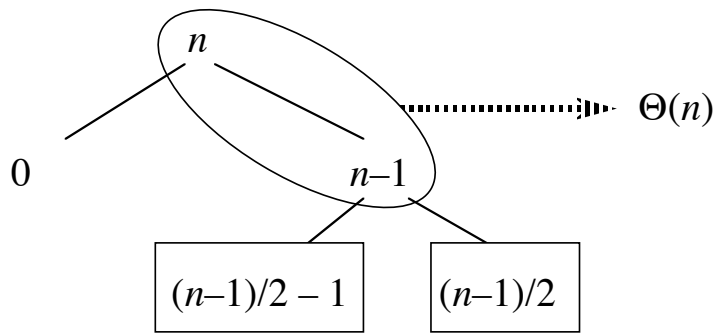
$$T(n) = \Theta(n \log n)$$

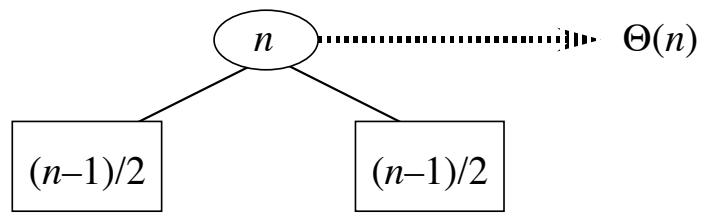
Complexité moyenne de QuickSort : intuitivement

- Complexité moyenne identique à la complexité du meilleur cas

$$T(n) = \Theta(n \log n)$$

- Intuitivement :
  - En moyenne, on s'attend à une alternance de "bons" et de "mauvais" partitionnements
  - La complexité d'un mauvais partitionnement suivi d'un bon est identique à la complexité d'une bon partitionnement directement (seule la constante est modifiée).





Complexité moyenne de QuickSort : mathématiquement

Modèle mathématique :

- Nombre de comparaisons pour le partitionnement : $n-1$
- Probabilité que le pivot soit à la position k : $1/n$
- Tailles des sous-tableaux dans ce cas-là : $k-1$ et $n-k$
- Les sous-tableaux sont aussi triés aléatoirement

Le nombre *moyen* de comparaisons utilisées par le quicksort est donné par la récurrence suivante :

$$\begin{aligned} C_1 &= 0 \\ C_n &= n-1+\sum_{k=1}^{n}\frac{1}{n}(C_{k-1}+C_{n-k}) \text{ (si } n>1) \end{aligned}$$

Formulation analytique

$$C_n = n - 1 + \sum_{k=1}^{n} \frac{1}{n}(C_{k-1} + C_{n-k})$$

Par symétrie :

$$C_n = n - 1 + \frac{2}{n} \sum_{k=1}^{n} C_{k-1}$$

En multipliant par n :

$$nC_n = n(n-1) + 2 \sum_{k=1}^{n} C_{k-1}$$

En soustrayant la même formule pour $n-1$:

$$nC_n - (n-1)C_{n-1} = 2(n-1) + 2C_{n-1}$$

En rassemblant les termes :

$$nC_n = (n+1)C_{n-1} + 2(n-1)$$

On divise par $n(n+1)$:

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2(n-1)}{n(n+1)}$$

Téléscopage :

$$\begin{aligned}
\frac{C_n}{n+1} &= \frac{C_{n-1}}{n} + \frac{2(n-1)}{n(n+1)} = \frac{C_{n-2}}{n-1} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\
&= \frac{C_1}{2} + \frac{2\cdot 1}{2\cdot 3} + \ldots + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\
&= \sum_{k=2}^{n} \frac{2(k-1)}{k(k+1)} = \sum_{k=2}^{n} \frac{2}{(k+1)} - \sum_{k=2}^{n} \frac{2}{k(k+1)}
\end{aligned}$$

En négligeant la deuxième somme devant la première, on obtient :

$$C_n = 2(n+1)H_n - 3(n+1), \text{ où } H_n = \sum_{k=1}^{n} \frac{1}{k} \text{ est la série harmonique.}$$

En utilisant l'approximation de la série harmonique $H_n \in \Theta(\log n)$:

$$C_n \in \Theta(n \log n),$$

Variantes de Quicksort

- Choix du pivot :
  - Prendre un élément au hasard plutôt que le dernier.
  - Prendre la médiane de 3 éléments
  - Diminue drastiquement les chances d'être dans le pire cas

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  swap(A[end], A[i])
3  return PARTITION(A, p, r)
```

```
MEDIAN-OF-3-PARTITION(A, p, r)
1  i = MEDIAN(A, p, ⌊(p + r)/2⌋, r)
2  swap(A[r], A[i])
3  return PARTITION(A, p, r)
```

Variantes de QUICKSORT

- Petits sous-tableaux
  - QUICKSORT est trop lourd pour des petits tableaux
  - Utiliser un tri naïf (par ex., par insertion) sur les sous-tableaux de longueur inférieure à k ($k \approx 20$).

```
QUICKSORT(A, p, r)
1  if r − p + 1 ≤ CUTOFF
2        INSERTIONSORT(A, p, r)
3        return
4  q = PARTITION(A, p, r)
5  QUICKSORT(A, p, q − 1)
6  QUICKSORT(A, q + 1, r)
```

Conclusion sur QuickSort

- Rapide en moyenne $\Theta(n \log n)$
- Pire cas en $\Theta(n^2)$ mais très improbable avec choix du pivot bien fait
- Bonne performance au niveau du cache
- Tri en place (mais utilise de la mémoire pour la trace récursive)
- Pas stable
- En pratique souvent un peu plus rapide que Merge-Sort
- Complexité en espace $O(\log n)$ si bien implémenté (récursif terminal, en développant d'abord la partition la plus petite)

Jusqu'ici

Algorithme	*Complexité*			*En place ?*
	Pire	*Moyenne*	*Meilleure*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	oui
Bubble-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	non
QuickSort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	oui
??	$\Theta(n \log n)$			oui

Tri par tas : introduction

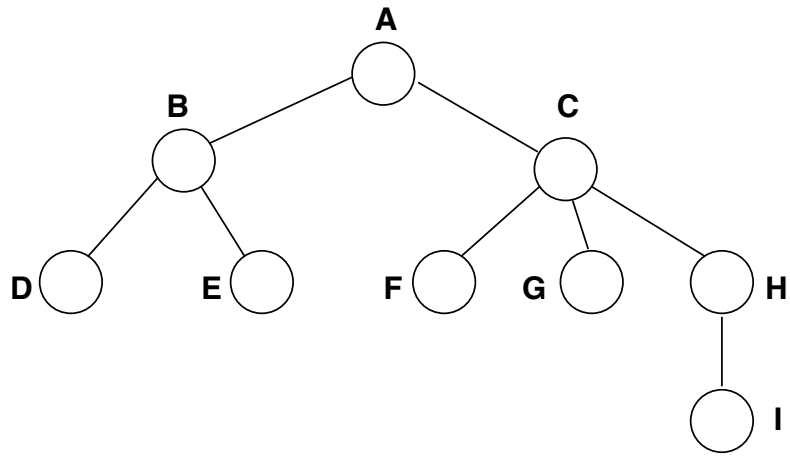
- *Heapsort* en anglais
- inventé par Williams en 1964
- basé sur une structure de données très utile, le *tas*
- complexité bornée par $\Theta(n \log n)$ (dans tous les cas)
- tri en place
- mise en oeuvre très simple

- Suite du cours :
  - Introduction aux arbres
  - Tas
  - Tri par tas

Arbres : définition

- Définition : Un arbre[1] (*tree*) T est un graphe dirigé (N, E), où :
  - N est un ensemble de nœuds, et
  - $E \subset N \times N$ est un ensemble d'arcs,

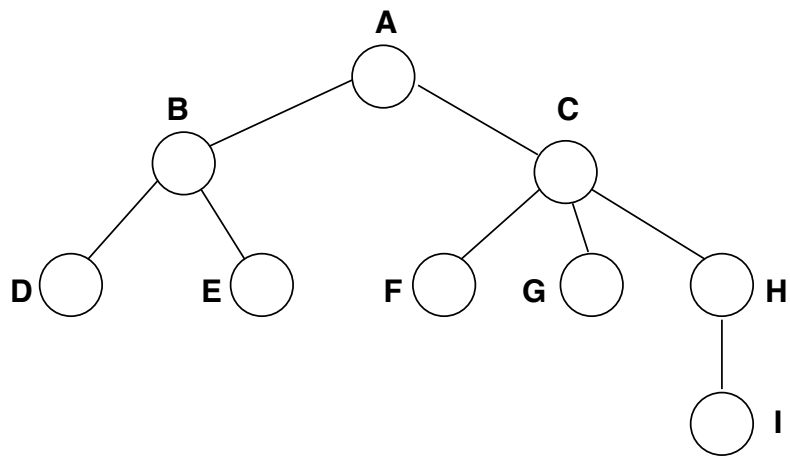
  possédant les propriétés suivantes :
  - T est connexe et acyclique
  - Si T n'est pas vide, alors il possède un nœud distingué appelé racine (*root node*). Cette racine est unique.
  - Pour tout arc $(n_1, n_2) \in E$, le nœud n_1 est le parent de n_2.
    - La racine de T ne possède pas de parent.
    - Les autres nœuds de T possèdent un et un seul parent.



1. En théorie des graphes, on parlera d'un arbre enraciné.

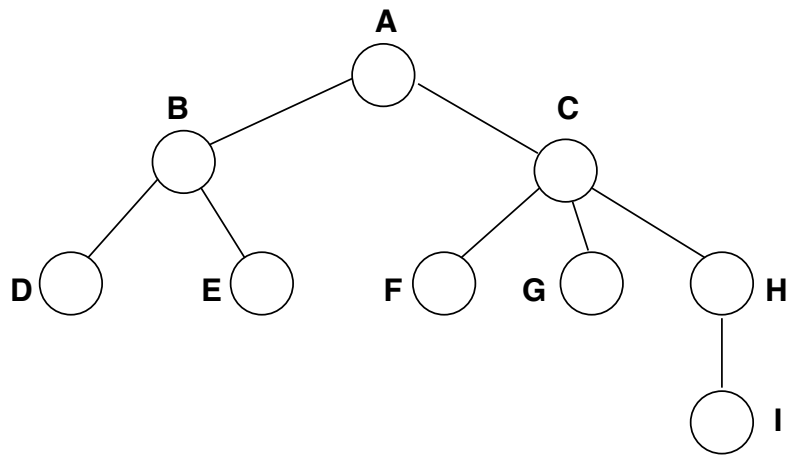
Arbres : terminologie

- Si n_2 est le parent de n_1, alors n_1 est le fils (*child*) de n_2.
- Deux nœuds n_1 et n_2 qui possèdent le même parent sont des frères (*siblings*).
- Un nœud qui possède au moins un fils est un nœud interne.
- Un nœud externe (c'est-à-dire, non interne) est une feuille (*leaf*) de l'arbre.
- Un nœud n_2 est un ancêtre (*ancestor*) d'un nœud n_1 si n_2 est le parent de n_1 ou un ancêtre du parent de n_1.
- Un nœud n_2 est un descendant d'un nœud n_1 si n_1 est un ancêtre de n_2.



Arbres : terminologie

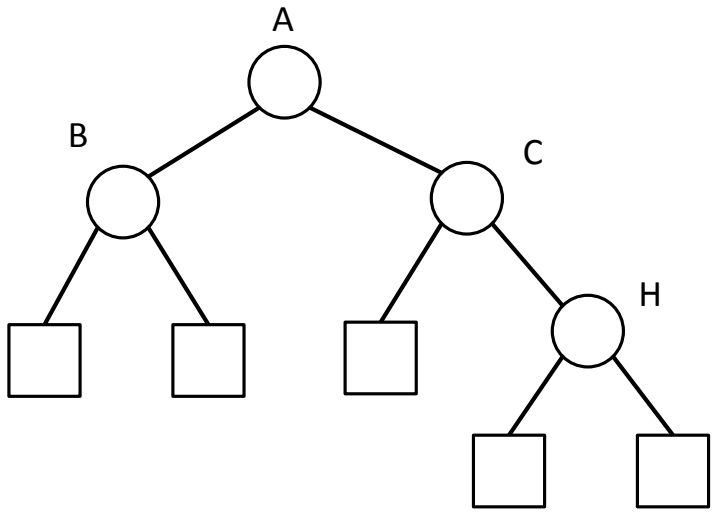
- Un chemin est une séquence de nœuds $n_1, n_2, \ldots, n_m$ telle que pour tout $i \in [1, m-1]$, (n_i, n_{i+1}) est un arc de l'arbre.
  Remarque : Il n'existe jamais de chemin reliant deux feuilles distinctes.
- La hauteur (*height*) d'un nœud n est le nombre d'arcs d'un plus long chemin de ce nœud vers une feuille. La *hauteur de l'arbre* est la hauteur de sa racine.
- La profondeur (*depth*) d'un nœud n est le nombre d'arcs sur le chemin qui le relie à la racine.



Arbre binaire

- Un arbre ordonné est un arbre dans lequel les ensembles de fils de chacun de ses nœuds sont ordonnés.
- Un arbre binaire est un arbre ordonné possédant les propriétés suivantes :
  - Chacun de ses nœuds possède au plus deux fils.
  - Chaque nœud fils est soit un fils gauche, soit un fils droit.
  - Le fils gauche précède le fils droit dans l'ordre des fils d'un nœud.
- Un arbre binaire entier ou propre (*full* or *proper*) est un arbre binaire dans lequel tous les nœuds internes possèdent exactement deux fils.
- Un arbre binaire parfait est un arbre binaire entier dans lequel toutes les feuilles sont à la même profondeur.

Propriétés des arbres binaires entiers



- Le nombre de nœuds externes est égal au nombre de nœuds internes plus 1.
- Le nombre de nœuds internes est égal à $\frac{n-1}{2}$, où n désigne le nombre de nœuds.
- Le nombre de nœuds à la profondeur (ou niveau) i est $\leq 2^i$.
- La hauteur h de l'arbre est $\leq$ au nombre de nœuds internes.
- Le lien entre hauteur et nombre de nœuds peut être résumé comme suit :

$$n \in \Omega(h) \text{ et } n \in O(2^h) \text{ (ou } h \in O(n) \text{ et } h \in \Omega(\log n))$$

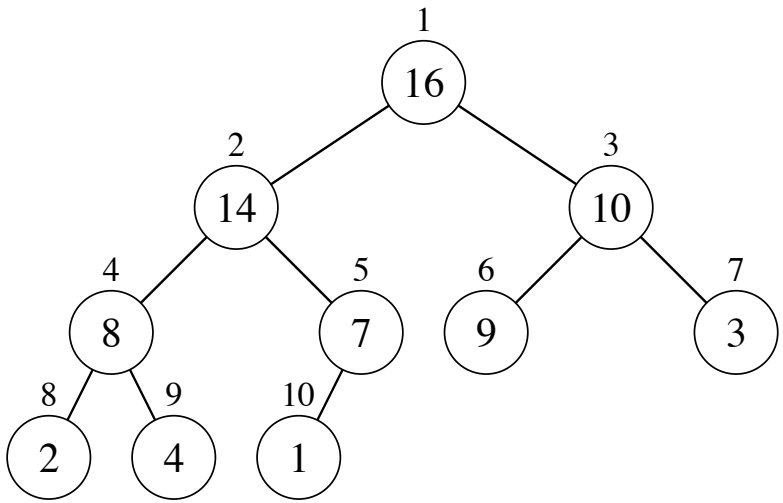
Tas : définition

Un arbre binaire complet est un arbre binaire tel que :

- Si h dénote la hauteur de l'arbre :
  - Pour tout $i \in [0, h-1]$, il y a exactement 2^i nœuds à la profondeur i.
  - Une feuille a une profondeur h ou $h-1$.
  - Les feuilles de profondeur maximale (h) sont "tassées" sur la gauche.

Un tas binaire (binary heap) est un arbre binaire complet tel que :

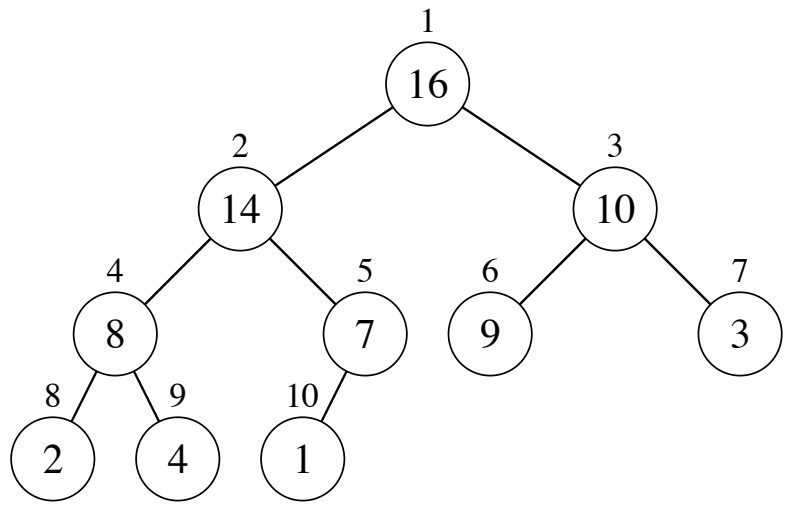
- Chacun de ses nœuds est associé à une clé.
- La clé de chaque nœud est supérieure ou égale à celle de ses fils (propriété d'ordre du tas).



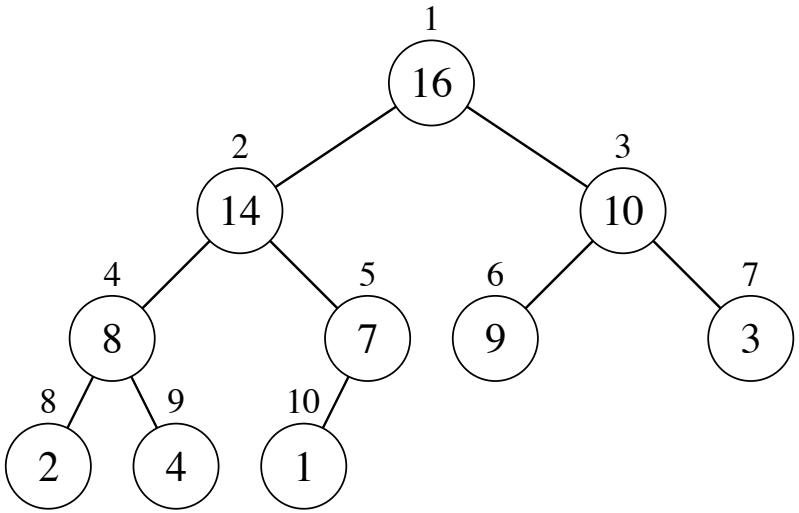
Propriété d'un tas

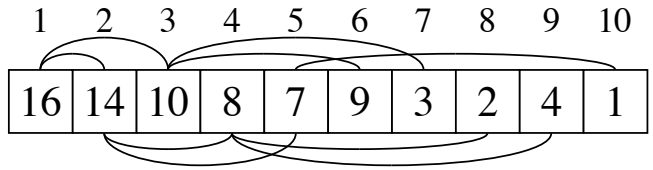
- Soit T un arbre binaire complet contenant n entrées et de hauteur h :
  - n est supérieur ou égal à la taille de l'arbre parfait de hauteur $h-1$ plus un, soit $2^{h-1+1} - 1 + 1 = 2^h$
  - n est inférieur ou égal à la taille de l'arbre parfait de hauteur h, soit $2^{h+1} - 1$

$$
\begin{aligned}
2^h \leq n \leq 2^{h+1} - 1 \quad &\Leftrightarrow \quad 2^h \leq n < 2^{h+1} \\
&\Leftrightarrow \quad h \leq \log_2 n < h + 1 \\
&\Leftrightarrow \quad h = \lfloor \log_2 n \rfloor
\end{aligned}
$$



Implémentation par un tableau





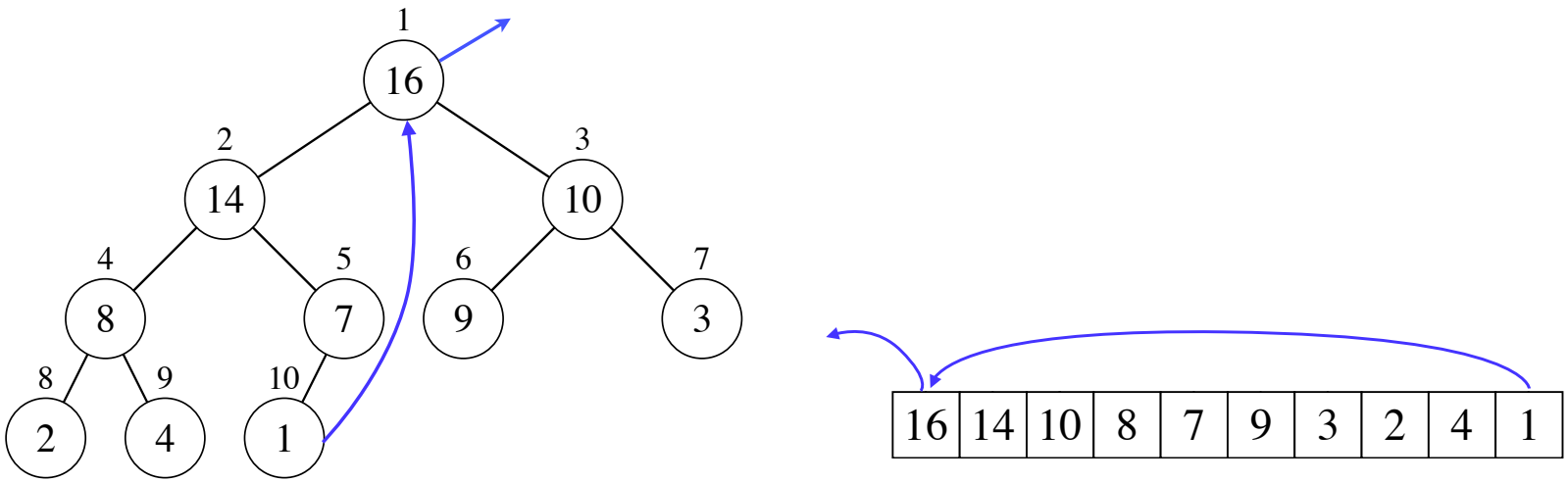
Un tas peut être représenté de manière compacte à l'aide d'un tableau A.

- La racine de l'arbre est le premier élément du tableau.
- $\textsc{Parent}(i) = \lfloor i/2 \rfloor$
- $\textsc{Left}(i) = 2i$
- $\textsc{Right}(i) = 2i + 1$

Propriété d'ordre du tas : $\forall i, A[\textsc{Parent}(i)] \geq A[i]$

Principe du tri par tas

- On construit un tas à partir du tableau à trier $\rightarrow$ BUILD-MAX-HEAP(A).
- Tant que le tas contient des éléments :
  - On extrait l'élément au sommet du tas qu'on place dans le tableau trié et on le remplace par l'élément le plus à droite

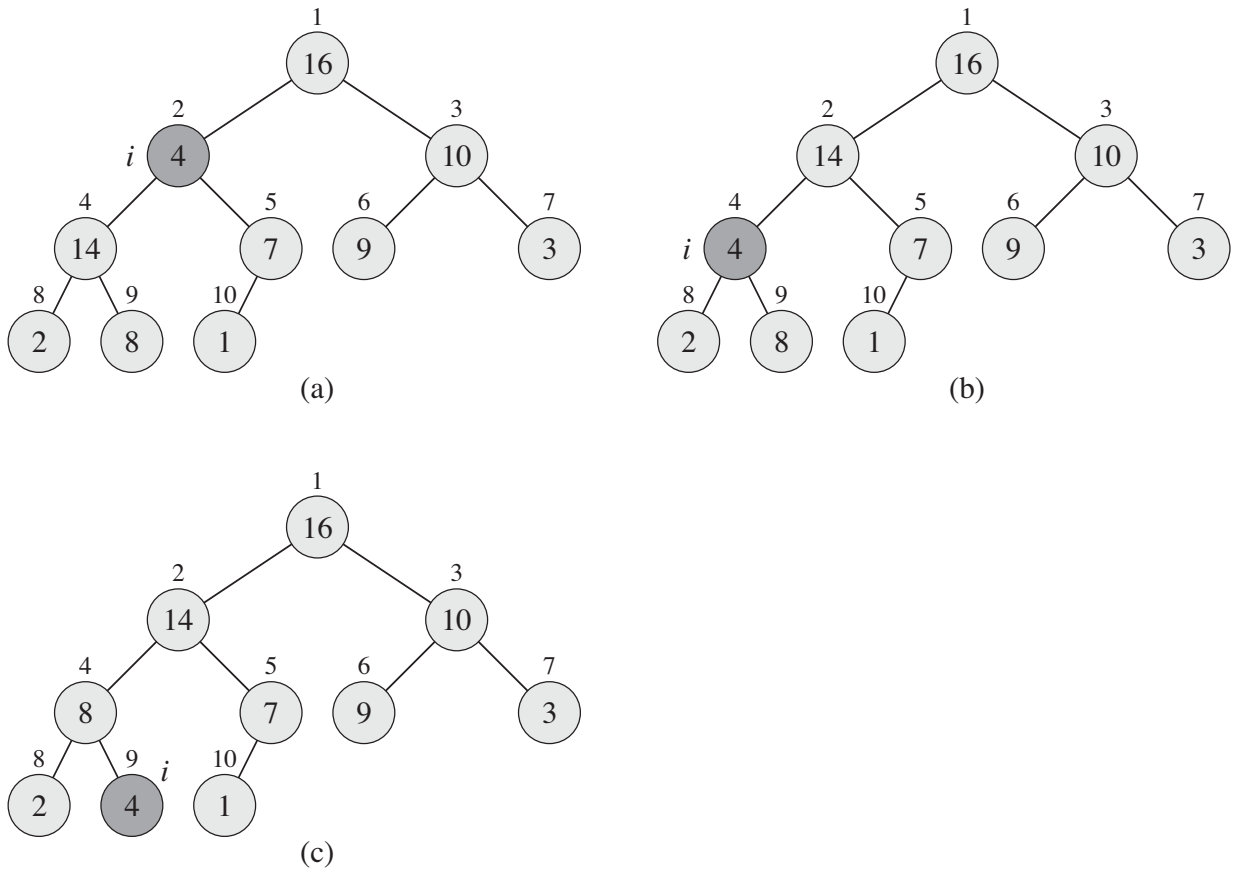


  - On rétablit la propriété de tas en tenant compte du fait que les sous-arbres de droite et de gauche sont des tas $\rightarrow$ MAX-HEAPIFY($A, 1$)

(Tout se fait dans le tableau initial $\rightarrow$ en place)

Max-Heapify

- Procédure Max-Heapify(A, i) :
  - Suppose que le sous-arbre de gauche du nœud i est un tas
  - Suppose que le sous-arbre de droite du nœud i est un tas
  - But : réarranger le tas pour maintenir la propriété d'ordre du tas
- Ex : Max-Heapify($A, 2$)



Max-Heapify

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size ∧ A[l] > A[i]
 4        largest = l
 5  else largest = i
 6  if r ≤ A.heap-size ∧ A[r] > A[largest]
 7        largest = r
 8  if largest ≠ i
 9        swap(A[i], A[largest])
10        MAX-HEAPIFY(A, largest)
```

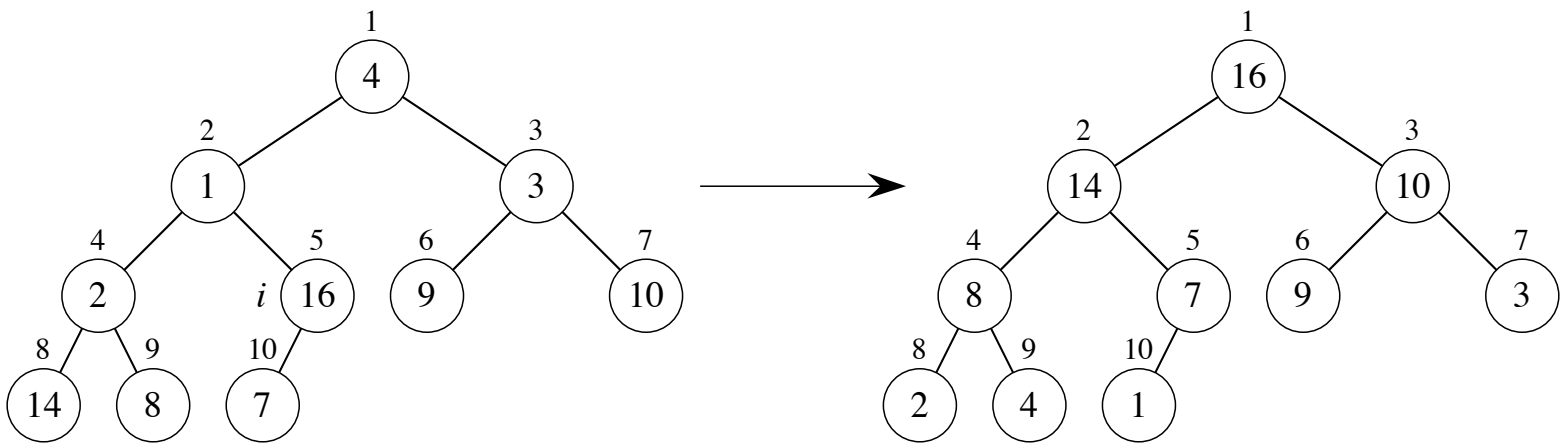
- Complexité ? La hauteur du nœud : $T(n) = O(\log n)$ (Transp. 165)

Construction d'un tas

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

(Invariant : chaque nœud i, i+1,. . . , n est la racine d'un tas)

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

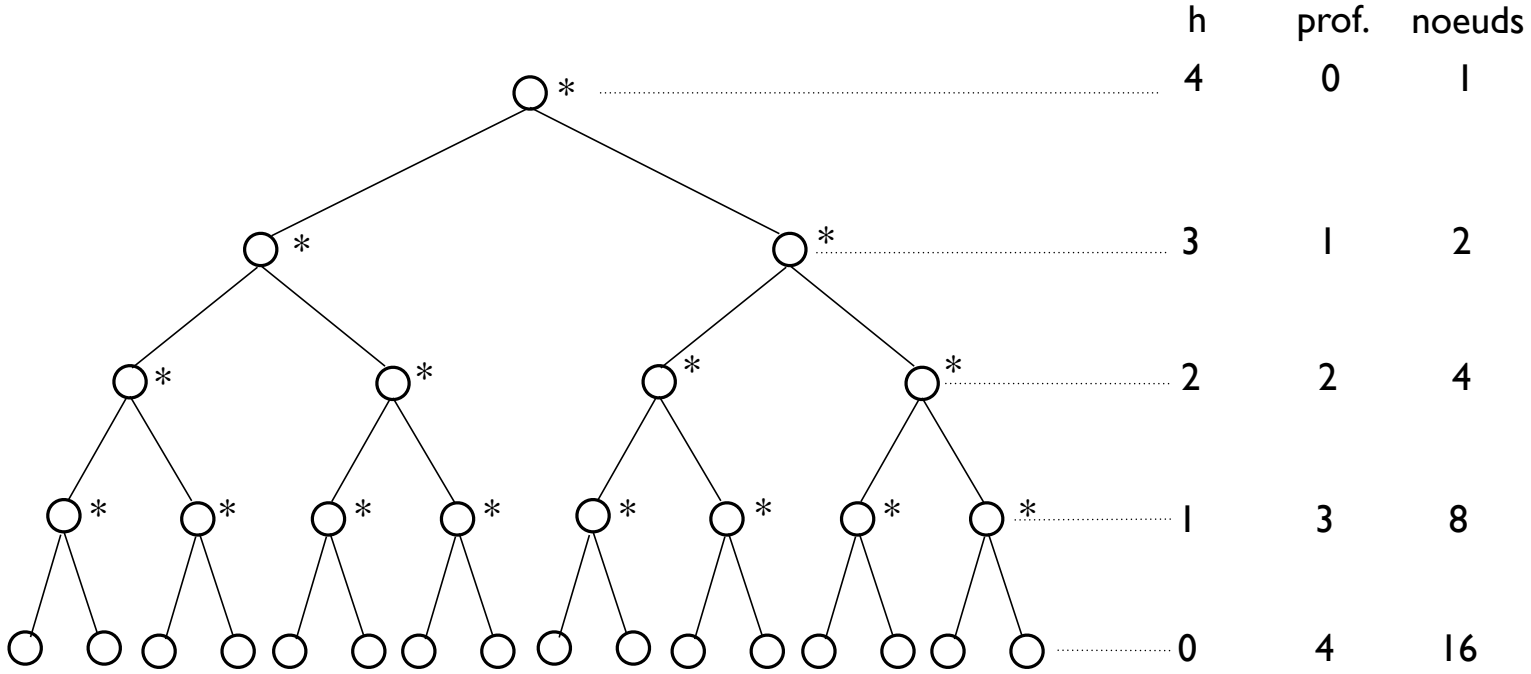


- La tableau initial est interprété comme un arbre binaire complet
- On tasse les nœuds internes de bas en haut et de droite à gauche

Complexité de Build-Max-Heap

- Borne simple :
  - $O(n)$ appels à Max-Heapify, chacun étant $O(\log n) \Rightarrow O(n \log n)$.
- Analyse plus fine :
  - Pour simplifier l'analyse, on suppose que l'arbre binaire est parfait.
  - On a donc $n = 2^{h+1} - 1$ pour un $h \geq 0$, qui est aussi la hauteur de l'arbre résultant

Complexité de Build-Max-Heap



* Noeuds sur lesquels on doit appeler Max-Heapify

- Il y a 2^i nœuds à la profondeur i ($=$ hauteur $h - i$).
- On doit appeler Max-Heapify sur chacun d'eux
- Chaque appel est au pire $\Theta(h - i)$.
- Nombre d'opérations en fonction de h au pire cas :

$$T(h) = \sum_{i=0}^{h-1} 2^i \Theta(h-i) = \Theta(\sum_{i=0}^{h-1} 2^i (h-i))$$

Complexité de Build-Max-Heap

- $\sum_{i=0}^{h-1} 2^i(h-i) = 1 \cdot 2^{h-1} + 2 \cdot 2^{h-2} + \ldots + (h-1) \cdot 2^1 + h \cdot 2^0$

$$
\begin{array}{ccccccccccc}
2^{h-1} & + & 2^{h-2} & + & 2^{h-3} & + & \ldots & + & 2^0 & = & 2^h - 1 \\
 & + & 2^{h-2} & + & 2^{h-3} & + & \ldots & + & 2^0 & = & 2^{h-1} - 1 \\
 & & & + & 2^{h-3} & + & \ldots & + & 2^0 & = & 2^{h-2} - 1 \\
 & & & & & & \ldots & & & \ldots & \\
 & & & & & & & & 2^0 & = & 2^1 - 1 \\
\hline
 & & & & & & & & & = & \left(\sum_{i=1}^{h} 2^i\right) - h
\end{array}
$$

(En utilisant $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$)

- On obtient donc $T(h) = \Theta(2^{h+1} - h - 2)$.
- Puisque $h = log_2(n+1) - 1 = \Theta(\log n)$, on a

$$T(n) = \Theta(n).$$

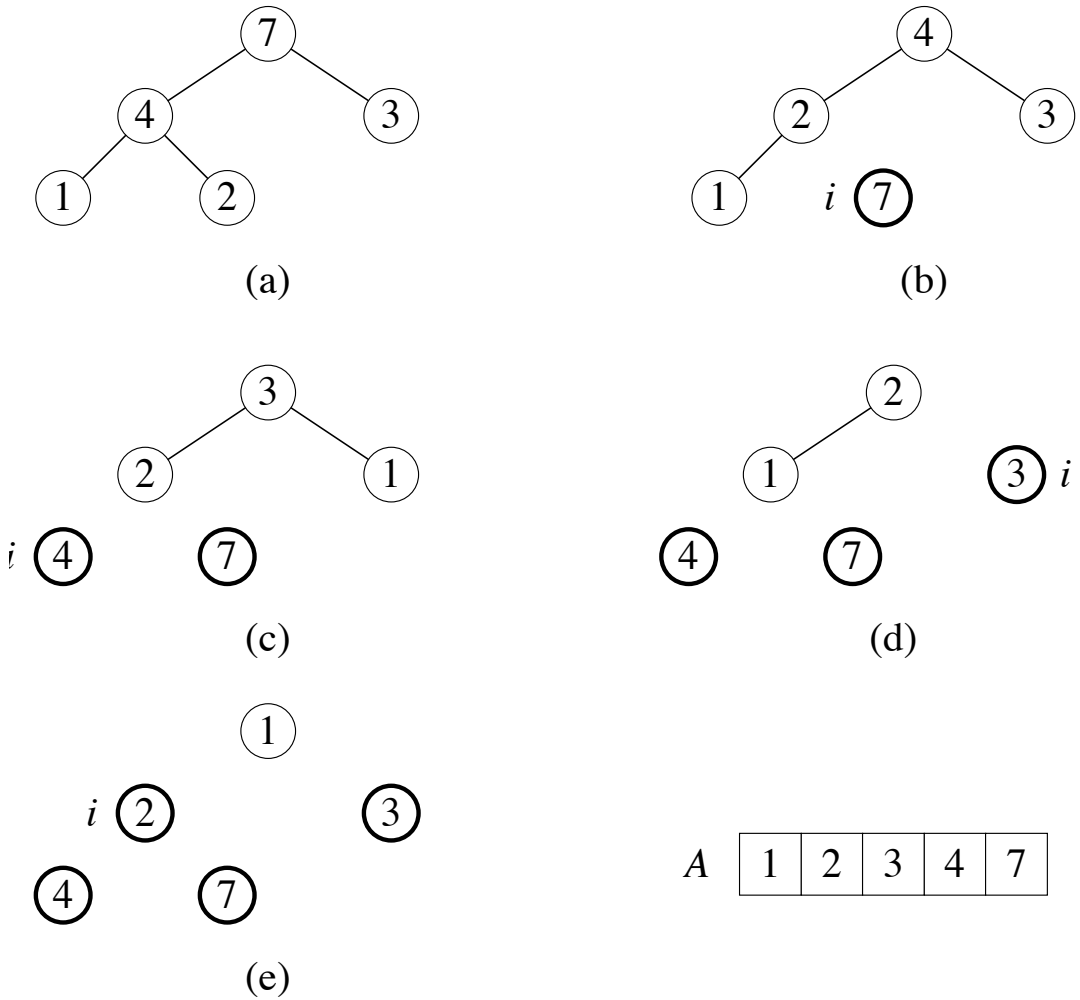
Tri par tas : algorithme

```
HEAP-SORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3       swap(A[i], A[1])
4       A.heap-size = A.heap-size − 1
5       MAX-HEAPIFY(A, 1)
```

Invariant :
$A[1\,..\,i]$ est un tas contenant les i éléments les plus petits de $A[1\,..\,A.length]$ et $A[i+1\,..\,A.length]$ contient les $n-i$ éléments les plus grands de $A[1\,..\,A.length]$ triés.

Tri par tas : illustration

Tableau initial : $A = [7, 4, 3, 1, 2]$



Complexité de Heap-Sort

```
Heap-Sort(A)
1  Build-Max-Heap(A)
2  for i = A.length downto 2
3      swap(A[i], A[1])
4      A.heap-size = A.heap-size − 1
5      Max-Heapify(A, 1)
```

- Build-Max-Heap : $O(n)$
- Boucle **for** : $n - 1$ fois
- Echange d'éléments : $O(1)$
- Max-Heapify : $O(\log n)$

Total : $O(n \log n)$ (pour le pire cas et le cas moyen)

Le tri par tas est cependant généralement battu par le tri rapide

Résumé

Algorithme	*Complexité*			*En place ?*
	Pire	*Moyenne*	*Meilleure*	
Insertion-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Selection-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	oui
Bubble-Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	oui
Merge-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	non
Quick-Sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	oui
Heap-Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)^*$	oui

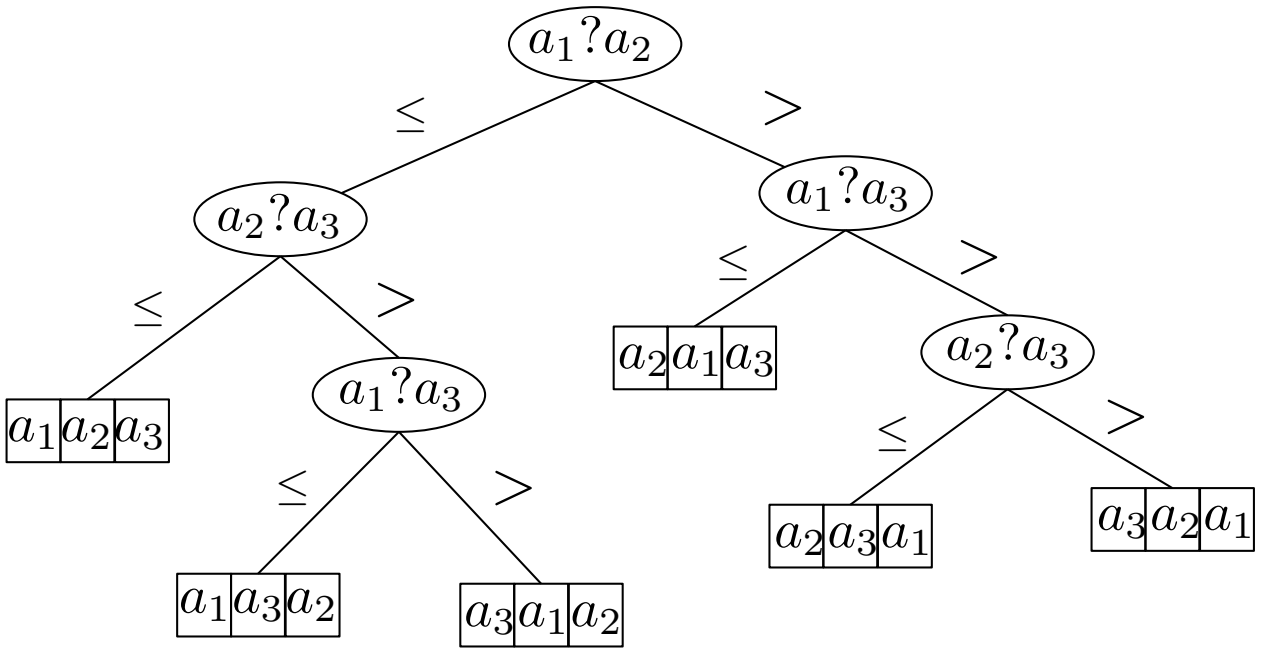
* pas montré dans ce cours

Peut-on faire mieux que $O(n \log n)$?

- Non, si on se restreint aux tri *comparatifs*, c'est-à-dire :
  - Aucune hypothèse sur les éléments à trier
  - Nécessité de les comparer entre eux
- Complexité d'un problème algorithmique versus complexité d'un algorithme
- Dans ce cas, un algorithme de tri est :
  - Une suite de comparaisons d'éléments suivant une certaine méthode
  - Un processus qui transforme un tableau $[e_0, e_1, \ldots, e_{n-1}]$ en un autre tableau $[e_{\sigma_0}, e_{\sigma_1}, \ldots, e_{\sigma_{n-1}}]$ où $(\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$ est une permutation de $(0, 1, \ldots, n-1)$.

Arbre de décision : exemple

Un algorithme de tri = un arbre binaire de décision (entier)



(arbre de décision pour le tri par insertion du tableau $[e_0, e_1, e_2]$)

Exercice : construire l'arbre pour le tri par fusion

Arbre de décision : définition

Un algorithme de tri = un arbre binaire de décision

- feuille de l'arbre : une permutation des éléments du tableau initial
- tri : le chemin de la racine à la feuille correspondant au tableau trié
- hauteur de l'arbre : le pire cas pour le tri
- branche la plus courte : le meilleur cas pour le tri
- hauteur moyenne de l'arbre : la complexité en moyenne du tri

Arbre de décision : propriété

- Un arbre binaire de hauteur h a au plus 2^h feuilles (cf transp. 163)
- Le nombre de feuilles de l'arbre de décision est exactement $n!$ où n est la taille du tableau à trier
  (par l'absurde : si moins que $n!$ certains tableaux ne seraient pas correctement triés)
- On a donc :

$$n! \leq 2^h \Rightarrow \log(n!) \leq h$$

- Formule de Stirling :

$$n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n})) \Rightarrow n! \geq (\frac{n}{e})^n$$

$$h \geq \log(n!) > \log((\frac{n}{e})^n) = n\log n - n\log e \Rightarrow h = \Omega(n\log n)$$

Le problème du tri comparatif est $\Omega(n\log n)$

Ce qu'on a vu

- Catégorisation des algorithmes de tri
- QuickSort (tri en place en $\Theta(n \log n)$)
- Analyse du cas moyen d'un algorithme
- Notre première structure de données : le tas
- HeapSort (tri en place en $\Theta(n \log n)$)
- Borne inférieure sur les tris comparatifs

- Illustration :
  - `http://www.sorting-algorithms.com/`
  - `http://www.youtube.com/user/algorythmics`

Ce qu'on n'a pas vu

- Analyse du meilleur cas du tri par tas
- Invariant pour le tri par tas
- Méthodes de tri linéaire
- Méthodes de sélection : trouver l'élément de rang i

Partie 4

Structures de données

Plan

1. Introduction
2. Pile et file
3. Structures linéaires : Liste, vecteur, séquence
4. Arbres
5. File à priorité
6. Ensembles disjoints

Concept

- Une structure de données est une manière d'organiser et de stocker l'information
  - Pour en faciliter l'accès ou dans d'autres buts
- Une structure de données a une interface qui consiste en un ensemble de procédures pour ajouter, effacer, accéder, réorganiser, etc. les données.
- Une structure de données conserve des données et éventuellement des méta-données
  - Par exemple : un tas utilise un tableau pour stocker les clés et une variable $A.\mathit{heap\text{-}size}$ pour retenir le nombre d'éléments qui sont dans le tas.
- Un type de données abstrait (TDA) = définition des propriétés de la structure et de son interface ("cahier des charges")

Structures de données

Dans ce cours :

- Principalement des ensembles dynamiques (dynamic sets), amenés à croître, se rétrécir et à changer au cours du temps.
- Les objets de ces ensembles comportent des attributs.
- Un de ces attributs est une clé qui permet d'identifier l'objet, les autres attributs sont la plupart du temps non pertinents pour l'implémentation de la structure.
- Certains ensembles supposent qu'il existe un ordre total entre les clés.

Opérations standards sur les structures

- Deux types : opérations de recherche/accès aux données et opérations de modifications
- Recherche : exemples :
  - $\textsc{Search}(S, k)$: retourne un pointeur x vers un élément dans S tel que $x.key = k$, ou NIL si un tel élément n'appartient pas à S.
  - $\textsc{Minimum}(S)$, $\textsc{Maximum}(S)$: retourne un pointeur vers l'élément avec la plus petite (resp. grande) clé.
  - $\textsc{Successor}(S, x)$,$\textsc{Predecessor}(S, x)$ retourne un pointeur vers l'élément tout juste plus grand (resp. petit) que x dans S, NIL si x est le maximum (resp. minimum).
- Modification : exemples :
  - $\textsc{Insert}(S, x)$: insère l'élément x dans S.
  - $\textsc{Delete}(S, x)$: retire l'élément x de S.

Implémentation d'une structure de données

- Etant donné un TDA (interface), plusieurs implémentations sont généralement possibles
- La complexité des opérations dépend de l'implémentation, pas du TDA.

- Les briques de base pour implémenter une structure de données dépendent du langage d'implémentation
  - Dans ce cours, les principaux outils du C : tableaux, structures à la C (objets avec attributs), liste liées (simples, doubles, circulaires), etc.
- Une structure de données peut être implémentée à l'aide d'une autre structure de données (de base ou non)

Quelques structures de données standards

- Pile : collection d'objets accessibles selon une politique LIFO
- File : collection d'objets accessibles selon une politique FIFO
- File double : combine accès LIFO et FIFO
- Liste : collection d'objets ordonnés accessibles à partir de leur position
- Vecteur : collection d'objets ordonnés accessibles à partir de leur rang
- Arbre : collection d'objets organisés en une structure d'arbre
- File à priorité : accès uniquement à l'élément de clé (priorité) maximale

- Dictionnaire : structure qui implémente les 3 opérations recherche, insertion, suppression (cf. partie 5)

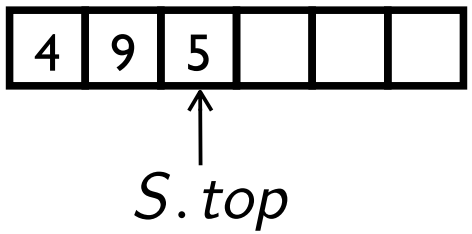
Plan

Pile

- Ensemble dynamique d'objets accessibles selon une discipline LIFO ("Last-in first-out").
- Interface
  - $\textsc{Stack-Empty}(S)$ renvoie vrai si et seulement si la pile est vide
  - $\textsc{Push}(S, x)$ pousse la valeur x sur la pile S
  - $\textsc{Pop}(S)$ extrait et renvoie la valeur sur le sommet de la pile S
- Applications :
  - Option 'undo' dans un traitement de texte
  - Langage postscript
  - Appel de fonctions dans un compilateur
  - ...
- Implémentations :
  - avec un tableau (taille fixée a priori)
  - au moyen d'une liste liée (allouée de manière dynamique)
  - ...

Implémentation par un tableau

- S est un tableau qui contient les éléments de la pile
- $S.top$ est la position courante de l'élément au sommet de S



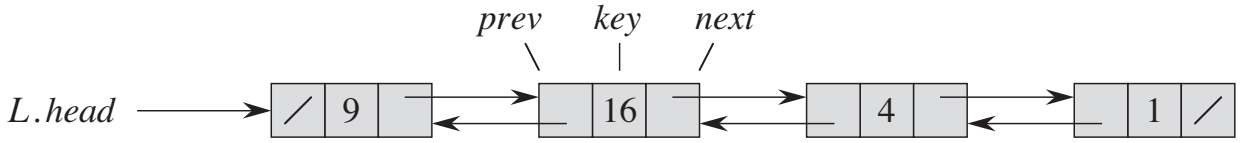

```
STACK-EMPTY(S)
1  return S.top == 0
```

```
PUSH(S, x)
1  if S.top == S.length
2        error "overflow"
3  S.top = S.top + 1
4  S[S.top] = x
```

```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```

- Complexité en temps et en espace : $O(1)$
  (Inconvénient : L'espace occupé ne dépend pas du nombre d'objets)

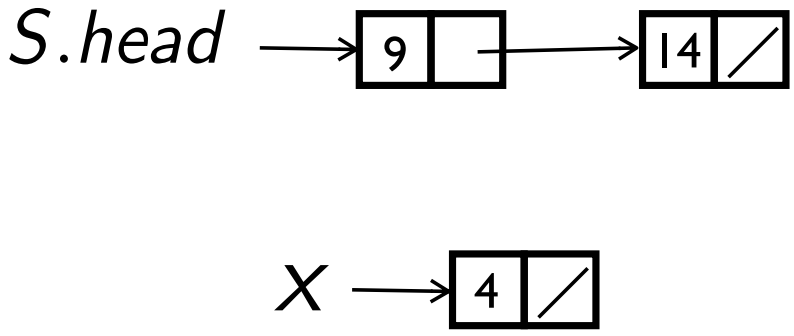
Rappel : liste simplement et doublement liée



- Structure de données composée d'une séquence d'éléments de liste.
- Chaque élément x de la liste est composé :
  - d'un contenu utile $x.data$ de type arbitraire (par exemple une clé),
  - d'un pointeur $x.next$ vers l'élément suivant dans la séquence
  - *Doublement liée :* d'un pointeur $x.prev$ vers l'élément précédent dans la séquence
- Soit L une liste liée
  - $L.head$ pointe vers le premier élément de la liste
  - *Doublement liée :* $L.tail$ pointe vers le dernier élément de la liste
- Le dernier élément possède un pointeur $x.next$ vide (noté NIL)
- *Doublement liée :* Le premier élément possède un pointeur $x.prev$ vide

Implémentation d'une pile à l'aide d'une liste liée

- S est une liste simplement liée ($S.head$ pointe vers le premier élément de la liste)




```
STACK-EMPTY(S)
1  if S.head == NIL
2        return TRUE
3  else return FALSE
```

```
PUSH(S, x)
1  x.next = S.head
2  S.head = x
```

```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else x = S.head
4        S.head = S.head.next
5        return x
```

- Complexité en temps $O(1)$, complexité en espace $O(n)$ pour n opérations

Application

- Vérifier l'appariement de parenthèses ([],() ou $\{\}$) dans une chaîne de caractères
  - Exemples : $((x)+(y)]/2 \rightarrow$ non, $[-(b)+\text{sqrt}(4*(a)*c)]/(2*a) \rightarrow$ oui
- Solution basée sur une pile :

```
ParenthesesMatch(A)
 1  S = pile vide
 2  for i = 1 to A.length
 3       if A[i] est une parenthèse gauche
 4            Push(S, A[i])
 5       elseif A[i] est une parenthèse droite
 6            if Stack-Empty(S)
 7                 return False
 8            elseif Pop(S) n'est pas du même type que A[i]
 9                 return False
10  return Stack-Empty(S)
```

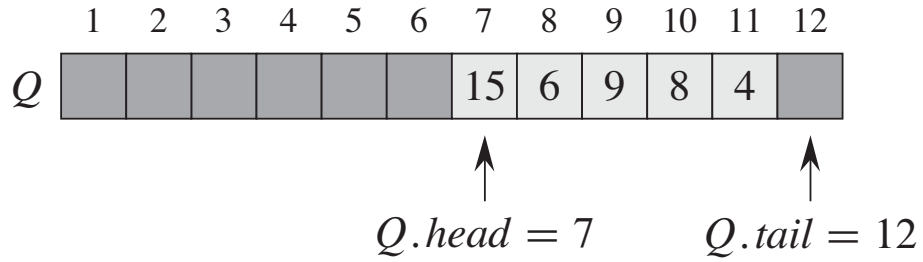
File

- Ensemble dynamique d'objets accessibles selon une discipline FIFO ("First-in first-out").
- Interface
  - $\textsc{Enqueue}(Q, s)$ ajoute l'élément x à la fin de la file Q
  - $\textsc{Dequeue}(Q)$ retire l'élément à la tête de la file Q

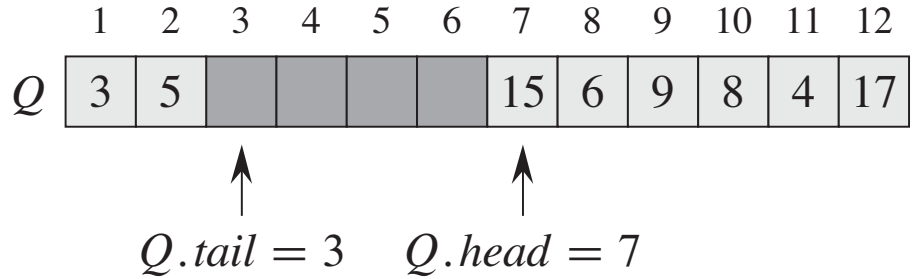
- Implémentation à l'aide d'un tableau circulaire
  - Q est un tableau de taille fixe $Q.\mathit{length}$
    - Mettre plus de $Q.\mathit{length}$ éléments dans la file provoque une erreur de dépassement
  - $Q.\mathit{head}$ est la position à la tête de la file
  - $Q.\mathit{tail}$ est la première position vide à la fin de la file
  - Initialement : $Q.\mathit{head} = Q.\mathit{tail} = 1$

Enqueue et Dequeue

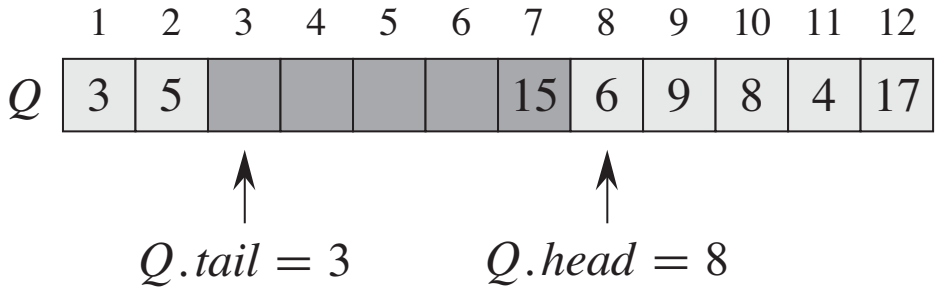
Etat initial :



Enqueue$(Q, 17)$, Enqueue$(Q, 3)$, Enqueue$(Q, 5)$



Dequeue$(Q) \rightarrow 15$



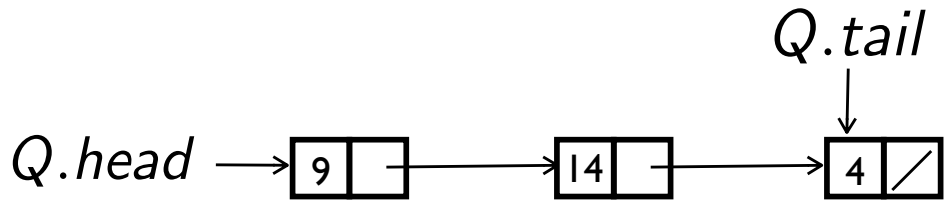
Enqueue et Dequeue

```
ENQUEUE(Q,x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

- Complexité en temps $O(1)$, complexité en espace $O(1)$.
- *Exercice : ajouter la gestion d'erreur*

Implémentation à l'aide d'une liste liée



- Q est une liste simplement liée
- $Q.head$ (resp. $Q.tail$) pointe vers la tête (resp. la queue) de la liste

```
ENQUEUE(Q,x)
1  x.next = NIL
2  if Q.head == NIL
3        Q.head = x
4  else Q.tail.next = x
5  Q.tail = x
```

```
DEQUEUE(Q)
1  if Q.head == NIL
2        error "underflow"
3  x = Q.head
4  Q.head = Q.head.next
5  if Q.head == NIL
6        Q.tail = NIL
7  return x
```

- Complexité en temps $O(1)$, complexité en espace $O(n)$ pour n opérations

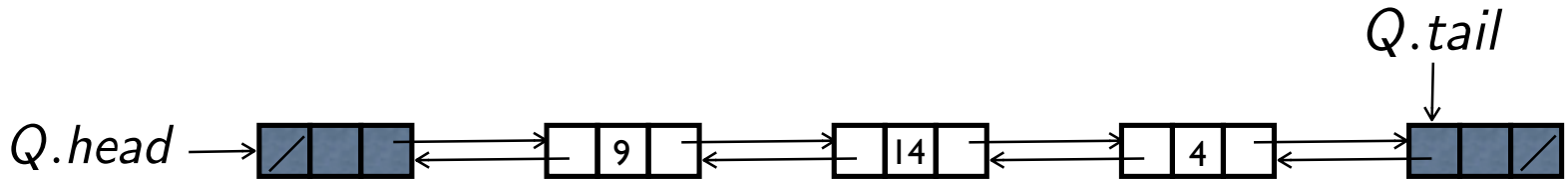
File double

Double ended-queue (deque)

- Généralisation de la pile et de la file
- Collection ordonnée d'objets offrant la possibilité
  - d'insérer un nouvel objet avant le premier ou après le dernier
  - d'extraire le premier ou le dernier objet
- Interface :
  - INSERT-FIRST(Q, x) : ajoute x au début de la file double
  - INSERT-LAST(Q, x) : ajoute x à la fin de la file double
  - REMOVE-FIRST(Q) : extrait l'objet situé en première position
  - REMOVE-LAST(Q) : extrait l'objet situé en dernière position
  - ...
- Application : équilibrage de la charge d'un serveur

Implémentation à l'aide d'une liste doublement liée

- A l'aide d'une liste doublement liée
- Soit la file double Q :
  - $Q.head$ pointe vers un élément sentinelle en début de liste
  - $Q.tail$ pointe vers un élément sentinelle en fin de liste
  - $Q.size$ est la taille courante de la liste



- Les sentinelles ne contiennent pas de données. Elles permettent de simplifier le code (pour un coût en espace constant).

Exercice : implémentation de la file double sans sentinelles, implémentation de la file simple avec sentinelle

Implémentation à l'aide d'une liste doublement liée

```
INSERT-FIRST(Q, x)
1  x.prev = Q.head
2  x.next = Q.head.next
3  Q.head.next.prev = x
4  Q.head.next = x
5  Q.size = Q.size + 1
```

```
INSERT-LAST(Q, x)
1  x.prev = Q.tail.prev
2  x.next = Q.tail
3  Q.tail.prev.next = x
4  Q.head.prev = x
5  Q.size = Q.size + 1
```

```
REMOVE-FIRST(Q)
1  if (Q.size == 0)
2        error
3  x = Q.head.next
4  Q.head.next = Q.head.next.next
5  Q.head.next.prev = Q.head
6  Q.size = Q.size − 1
7  return x
```

```
REMOVE-LAST(Q)
1  if (Q.size == 0)
2        error
3  x = Q.tail.prev
4  Q.tail.prev = Q.tail.prev.prev
5  Q.tail.prev.next = Q.head
6  Q.size = Q.size − 1
7  return x
```

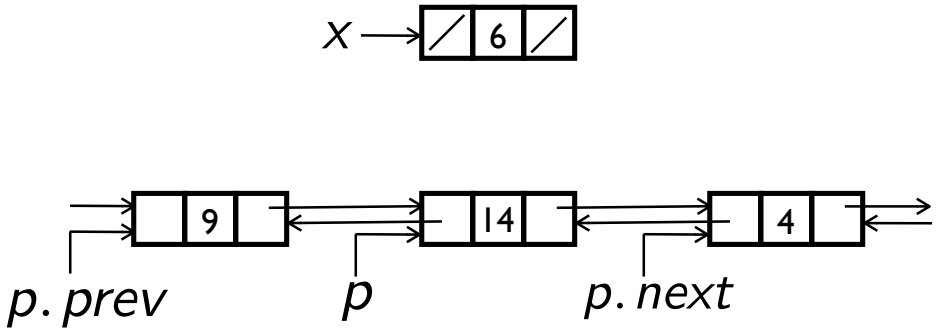
Complexité $O(1)$ en temps et $O(n)$ en espace pour n opérations.

Plan

Liste

- Ensemble dynamique d'objets ordonnés accessibles relativement les uns aux autres, sur base de leur position
- Généralise toutes les structures vues précédemment
- Interface :
  - Les fonctions d'une liste double (insertion et retrait en début et fin de liste)
  - INSERT-BEFORE(L, p, x) : insére x avant p dans la liste
  - INSERT-AFTER(L, p, x) : insère x après p dans la liste
  - REMOVE(L, p) : retire l'élement à la position p
  - REPLACE(L, p, x) : remplace par l'objet x l'objet situé à la position p
  - FIRST(L), LAST(L) : renvoie la première, resp. dernière position dans la liste
  - PREV(L, p), NEXT(L, p) : renvoie la position précédant (resp. suivant) p dans la liste
- Implémentation similaire à la file double, à l'aide d'une liste doublement liée (avec sentinelles)

Implémentation à l'aide d'une liste doublement liée




```
REMOVE(L, p)
1  p.prev.next = p.next
2  p.next.prev = p.prev
3  L.size = L.size − 1
4  return p
```

```
INSERT-BEFORE(L, p, x)
1  x.prev = p.prev
2  x.next = p
3  p.prev.next = x
4  p.prev = x
5  L.size = L.size + 1
```

```
INSERT-AFTER(L, p, x)
1  x.prev = p
2  x.next = p.next
3  p.next.prev = x
4  p.next = x
5  L.size = L.size + 1
```

Complexité $O(1)$ en temps et $O(n)$ en espace pour n opérations.

Vecteur

- Ensemble dynamique d'objets occupant des rangs entiers successifs, permettant la consultation, le remplacement, l'insertion et la suppression d'éléments à des rangs arbitraires
- Interface
  - ELEM-AT-RANK(V, r) retourne l'élément au rang r dans V.
  - REPLACE-AT-RANK(V, r, x) remplace l'élément situé au rang r par x et retourne cet objet.
  - INSERT-AT-RANK(V, r, x) insère l'élément x au rang r, en augmentant le rang des objets suivants.
  - REMOVE-AT-RANK(V, r) extrait l'élément situé au rang r et le retire de r, en diminuant le rang des objets suivants.
  - VECTOR-SIZE(V) renvoie la taille du vecteur.
- Applications : tableau dynamique, gestion des éléments d'un menu,...
- Implémentation : liste liée, tableau extensible...

Implémentation par un tableau extensible

- Les éléments sont stockés dans un tableau extensible $V.A$ de taille initiale $V.c$.
- En cas de dépassement, la capacité du tableau est doublée.
- $V.n$ retient le nombre de composantes.
- Insertion et suppression :

```
INSERT-AT-RANK(V, r, x)
 1  if V.n == V.c
 2        V.c = 2 · V.c
 3        W = "Tableau de taille V.c"
 4        for i = 1 to n
 5              W[i] = V.A[i]
 6        V.A = W
 7  for i = V.n downto r
 8        V.A[i + 1] = V.A[i]
 9  V.A[r] = x
10  V.n = V.n + 1
```

```
REMOVE-AT-RANK(V, r)
1  tmp = V.A[r]
2  for i = r to V.n − 1
3        V.A[i] = V.A[i + 1]
4  V.n = V.n − 1
5  return tmp
```

Complexité en temps

- Insert-At-Rank :
  - $O(n)$ pour une opération individuelle, où n est le nombre de composantes du vecteur
  - $\Theta(n^2)$ pour n opérations d'insertion en début de vecteur
  - $\Theta(n)$ pour n opérations d'insertion en fin de vecteur
- Justification :
  - Si la capacité du tableau passe de c_0 à $2^k c_0$ au cours des n opérations, alors le coût des transferts entre tableaux s'élève à

    $$c_0 + 2c_0 + \ldots + 2^{k-1}c_0 = (2^k - 1)c_0.$$

    Puisque $2^{k-1}c_0 < n \leq 2^k c_0$, ce coût est $\Theta(n)$.
  - On dit que le coût amorti par opération est $O(1)$
  - Si on avait élargi le tableau avec un incrément constant m, le coût aurait été

    $$c_0 + (c_0 + m) + (c_0 + 2m) + \ldots + (c_0 + (k-1)m) = kc_0 + \frac{k(k-1)}{2}m.$$

    Puisque $c_0 + (k-1)m < n \leq c_0 + km$, ce coût aurait donc été $\Theta(n^2)$.

Complexité en temps

- Remove-At-Rank :
  - $O(n)$ pour une opération individuelle, où n est le nombre de composantes du vecteur
  - $\Theta(n^2)$ pour n opérations de retrait en début de vecteur
  - $\Theta(n)$ pour n opérations de retrait en fin de vecteur

- Remarque : Un tableau circulaire permettrait d'améliorer l'efficacité des opérations d'ajout et de retrait en début de vecteur.

Séquence

- Ensemble dynamique d'objets ordonnés combinant les propriétés d'une liste et d'un vecteur, c'est-à-dire dont les objets sont accessibles tant sur base de leur position absolue que relative
- Interface :
  - Toutes les opérations d'un vecteur
  - Toutes les opérations d'une liste
  - $\textsc{atRank}(S, r)$: retourne la position de l'élément possédant le rang r.
  - $\textsc{rankOf}(S, p)$: retourne le rang de l'élément situé à la position p.

- Implémentation à l'aide d'une liste doublement liée (laissée comme exercice)
  - $\textsc{atRank}$,$\textsc{rankOf}$, $\textsc{Elem-At-Rank}$, $\textsc{Remove-At-Rank}$, $\textsc{Replace-At-Rank}$: $O(n)$, où n est le nombre d'éléments appartenant à la séquence.
  - Autres opérations : $O(1)$.

Plan

Type de données abstrait pour un arbre

- Principe :
  - Des données sont associées aux nœuds d'un arbre
  - Les nœuds sont accessibles les uns par rapport aux autres selon leur position dans l'arbre
- Interface : Pour un arbre T et un nœud n
  - $\textsc{Parent}(T, n)$: renvoie le parent d'un nœud n (signale une erreur si n est la racine)
  - $\textsc{isEmpty}(T)$: renvoie vrai si l'arbre est vide
  - $\textsc{Children}(T, n)$: renvoie une structure de données (ordonnée ou non) contenant les fils du nœud n (exemple : une liste)
  - $\textsc{isRoot}(T, n)$: renvoie vrai si n est la racine de l'arbre
  - $\textsc{isInternal}(T, n)$: renvoie vrai si n est un nœud interne
  - $\textsc{isExternal}(T, n)$: renvoie vrai si n est un nœud externe
  - $\textsc{GetData}(T, n)$: renvoie les données associées au nœud n
  - $\textsc{Root}(T)$: renvoie le nœud racine de l'arbre
  - $\textsc{Size}(T)$: renvoie le nombre de nœuds de l'arbre
  - Pour un arbre binaire (ordonné) :
    - $\textsc{Left}(T, n)$, $\textsc{Right}(T, n)$: renvoie les fils gauche et droit de n
    - $\textsc{hasLeft}(T, n)$, $\textsc{hasRight}(T, n)$: détermine si le nœud n possède un fils respectivement gauche et droit.

Exemples d'opération sur un arbre

- Calcul de la profondeur d'un nœud

```
DEPTH-REC(T, n)
1  if ISROOT(T, n)
2        return 0
3  return 1 + DEPTH-REC(T, PARENT(T, n))
```

- Version itérative

```
DEPTH-ITER(T, n)
1  d = 0
2  while not ISROOT(T, n)
3        d = d + 1
4        n = PARENT(T, n)
5  return d
```

- Complexité en temps : $O(n)$, où n est la taille de l'arbre (si les opérations de l'interface sont $O(1)$)

Exemples d'opération sur un arbre

- Calcul de la hauteur de l'arbre

```
HEIGHT(T, n)
1  if ISEXTERNAL(T, n)
2        return 0
3  h = 0
4  for each n2 in CHILDREN(T, n)
5        h = max(h, HEIGHT(T, n2))
6  return h + 1
```

- Complexité en temps : $O(n)$, où n est la taille de l'arbre (si les opérations de l'interface sont $O(1)$)

Implémentation d'un arbre binaire

Première solution : numérotation de niveaux

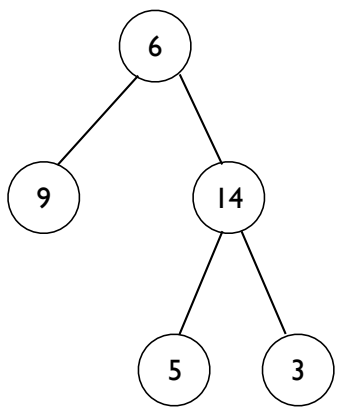
- L'arbre est représenté par un vecteur (ou un tableau)
- Chaque position dans l'arbre est associée à un rang particulier :
  - La racine est en position 1
  - Si un nœud est au rang r, son successeur gauche est au rang $2r$, son successeur droit au rang $2r + 1$
- Si l'arbre binaire n'est pas un arbre binaire complet, le vecteur contiendra des trous (qu'il faudra pouvoir identifier)
- Complexité en temps des opérations : $O(1)$
- Complexité en espace : $O(2^n)$ pour un arbre de n nœuds ($\Theta(n)$ pour un arbre binaire complet)

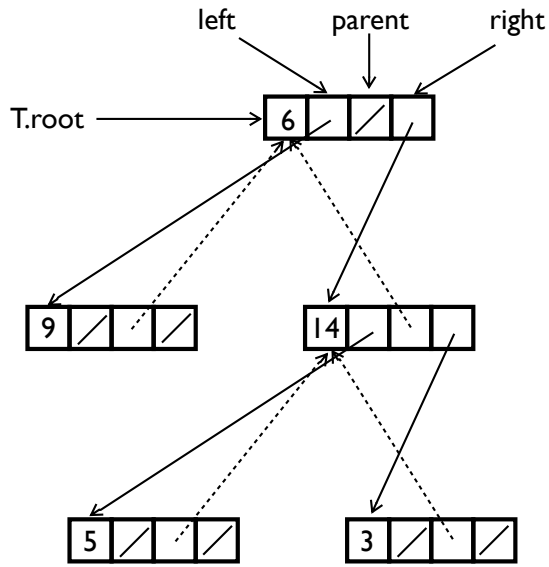
(Exercice : peut-on étendre à des arbres quelconques ?)

Implémentation d'un arbre binaire

Deuxième solution : structure liée

- Principe : on retient pour chaque nœud n de l'arbre :
  - Un champ de données ($n.data$)
  - Un pointeur vers son nœud parent ($n.parent$)
  - Un pointeur vers ses fils gauche et droit ($n.left$ et $n.right$)
  - $T.root$ pointe vers la racine de l'arbre
- Complexité en temps des opérations : $O(1)$
- Complexité en espace : $\Theta(n)$ pour n nœuds



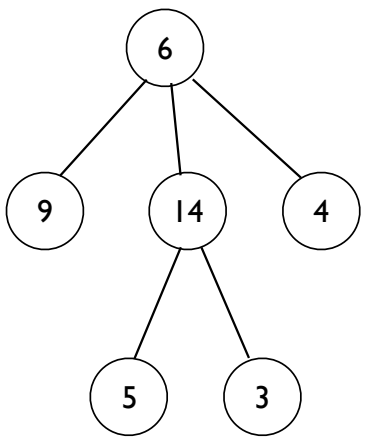


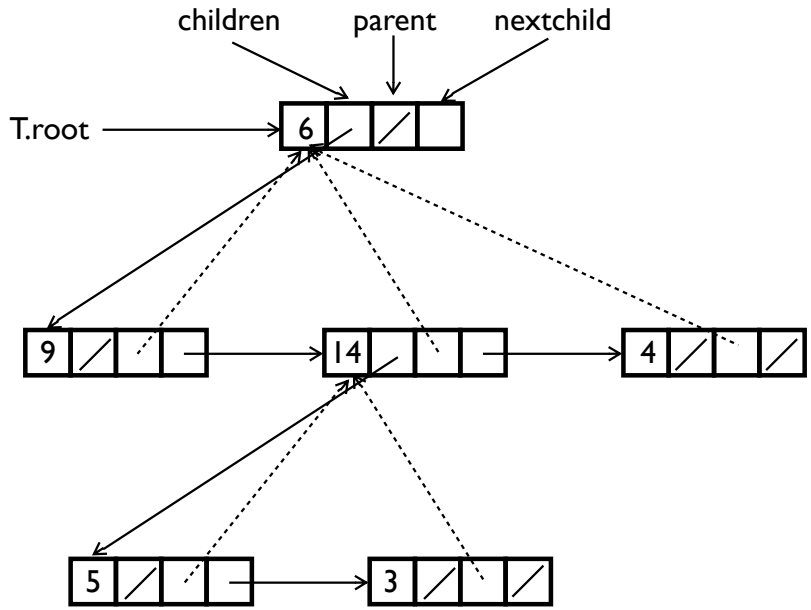
(généralise la notion de liste liée)

Implémentation d'un arbre quelconque

Première solution : structure liée

- Comme pour un arbre binaire
- Mais on remplace $n.left$ et $n.right$ par un pointeur $n.children$ vers un ensemble dynamique.
- Le type d'ensemble dynamique (vecteur, liste, ...) dépendra des opérations devant être effectuées
- Exemple avec une liste liée :

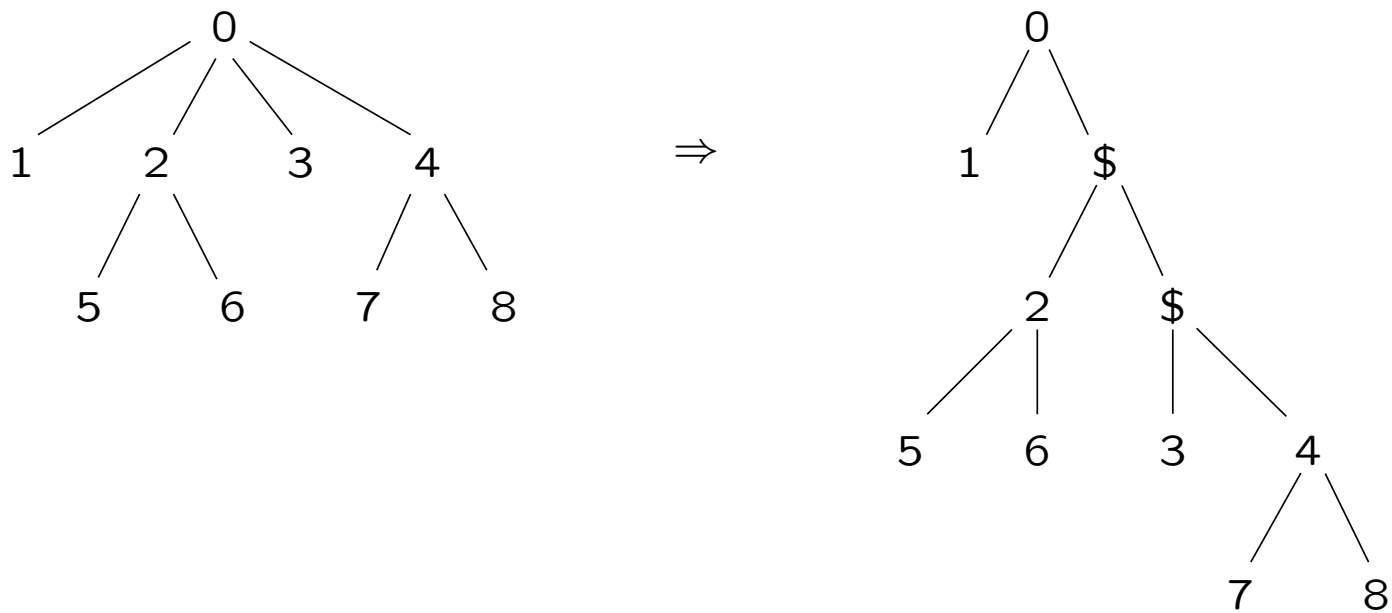




Implémentation d'un arbre quelconque

Deuxième approche : représenter l'arbre quelconque par un arbre binaire

- Si le nœud n possède les fils $n_1, n_2, \ldots, n_p$, avec $p > 2$, alors le sous-arbre issu de n est représenté par une arbre binaire dont :
  - n est la racine
  - Le fils gauche est la racine du sous-arbre issu de n_1
  - Le fils droit est associé à une valeur distinguée "$", et représente un arbre dont la racine possède les fils $n_2, n_3, \ldots, n_p$.
- Illustration :

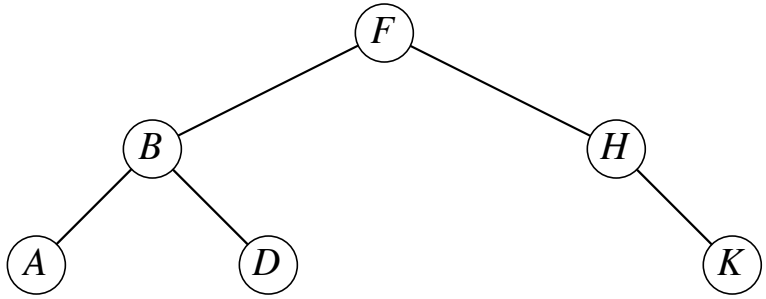


- Complexité des opérations ?

Parcours d'arbres (binaire)

- Un parcours d'arbre est une façon d'ordonner les nœuds d'un arbre afin de les parcourir
- Différents types de parcours :
  - Parcours en profondeur :
    - Infixe (en ordre)
    - Préfixe (en préordre)
    - Suffixe (en postordre)
  - Parcours en largeur

Parcours infixe

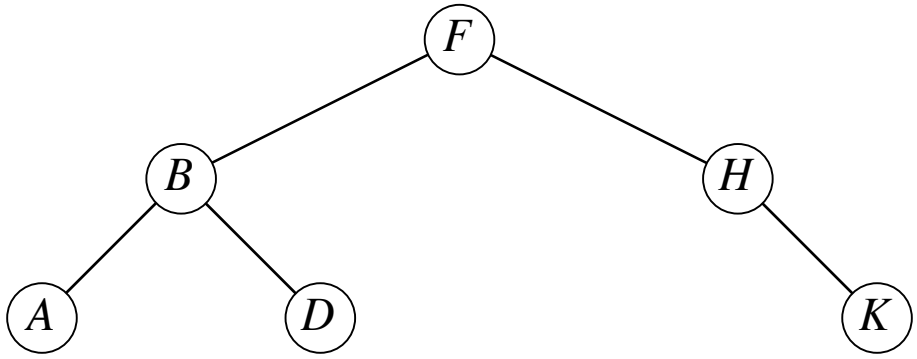


$$\Rightarrow \langle A, B, D, F, H, K \rangle$$

- Parcours infixe (en ordre) : Chaque nœud est visité après son fils gauche et avant son fils droit

```
INORDER-TREE-WALK(T, x)
1  if HASLEFT(T, x)
2        INORDER-TREE-WALK(T, LEFT(x))
3  print GETDATA(T, x)
4  if HASRIGHT(T, x)
5        INORDER-TREE-WALK(T, RIGHT(x))
```

Parcours préfixe

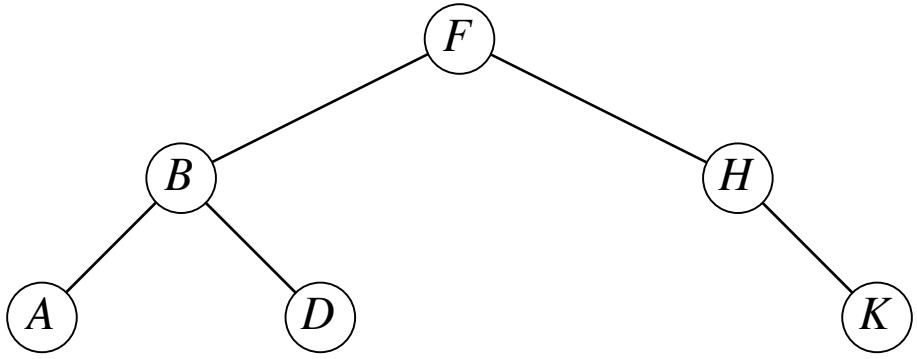


$$\Rightarrow \langle F, B, A, D, H, K \rangle$$

- Parcours préfixe (en préordre) : chaque nœud est visité avant ses fils

```
PREORDER-TREE-WALK(T, x)
1  print GETDATA(T, x)
2  if HASLEFT(T, x)
3       PREORDER-TREE-WALK(T, LEFT(x))
4  if HASRIGHT(T, x)
5       PREORDER-TREE-WALK(T, RIGHT(x))
```

Parcours postfixe



$$\Rightarrow \langle A, D, B, K, H, F \rangle$$

- Parcours postfixe (en postordre) : chaque nœud est visité après ses fils

```
POSTORDER-TREE-WALK(T, x)
1  if HASLEFT(T, x)
2        POSTORDER-TREE-WALK(T, LEFT(x))
3  if HASRIGHT(T, x)
4        POSTORDER-TREE-WALK(T, RIGHT(x))
5  print GETDATA(T, x)
```

Complexité des parcours

Tous les parcours en profondeur sont $\Theta(n)$ en temps

- Soit $T(n)$ le nombre d'opérations pour un arbre avec n nœuds
- On a $T(n) = \Omega(n)$ (on doit au moins parcourir chaque nœud).
- Etant donné la récurrence, on a :

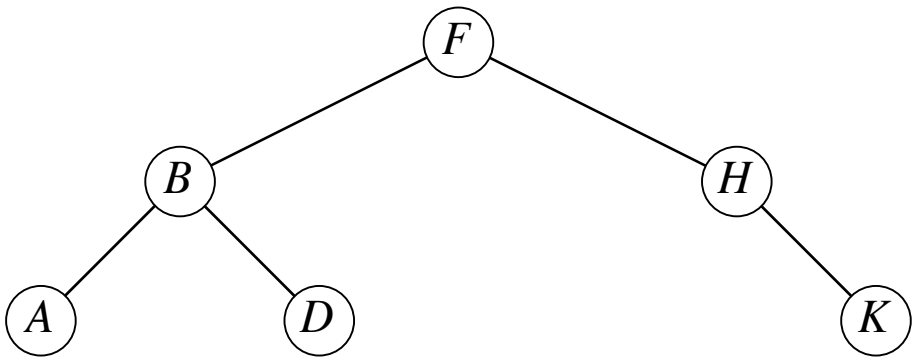
$$T(n) \leq T(n_L) + T(n - n_L - 1) + d$$

où n_L est le nombre de nœuds du sous-arbre à gauche et d une constante

- On peut prouver par induction que $T(n) \leq (c + d)n + c$ où $c = T(0)$.
- $T(n) = \Omega(n)$ et $T(n) = O(n) \Rightarrow T(n) = \Theta(n)$

Parcours en largeur

- Parcours en largeur : on visite le nœud le plus proche de la racine qui n'a pas déjà été visité. Correspond à une visite des nœuds de profondeur 1, puis 2, ...
- Implémentation à l'aide d'une file en $\Theta(n)$ *(complexité en espace ?)*



$\Rightarrow \langle F, B, H, A, D, K \rangle$

```
BREADTH-TREE-WALK(T)
 1  Q = "Empty queue"
 2  if not ISEMPTY(T)
 3       ENQUEUE(Q, ROOT(T))
 4  while not QUEUE-EMPTY(Q)
 5       y = DEQUEUE(Q)
 6       print GETDATA(T, y)
 7       if HASLEFT(T, y)
 8            ENQUEUE(Q, LEFT(y))
 9       if HASRIGHT(T, y)
10            ENQUEUE(Q, RIGHT(y))
```

(Exercice : Implémenter les parcours en profondeur de manière non récursive)

Plan

File à priorité

- Ensemble dynamique d'objets classés par ordre de priorité
  - Permet d'extraire un objet possédant la plus grande priorité
  - En pratique, on représente les priorités par les clés
  - Suppose un ordre total défini sur les clés
- Interface :
  - $\textsc{Insert}(S, x)$: insère l'élément x dans S.
  - $\textsc{Maximum}(S)$: renvoie l'élément de S avec la plus grande clé.
  - $\textsc{Extract-Max}(S)$: supprime et renvoie l'élément de S avec la plus grande clé.
- Remarques :
  - Extraire l'élément de clé maximale ou minimale sont des problèmes équivalents
  - La file FIFO est une file à priorité où la clé correspond à l'ordre d'arrivée des élements.
- Application : gestion des jobs sur un ordinateur partagé

Implémentations

- Implémentation à l'aide d'un tableau statique
  - Q est un tableau statique de taille fixée $Q.length$.
  - Les éléments de Q sont triés par ordre croissant de clés. $Q.top$ pointe vers le dernier élément.
  - Complexité en temps : extraction en $O(1)$ et insertion en $O(n)$ où n est la taille de la file
  - Complexité en espace : $O(1)$
- Implémentation à l'aide d'une liste liée
  - Q est une liste liée où les éléments sont triés par ordre décroissant de clés
  - Complexité en temps : extraction en $O(1)$ et insertion en $O(n)$ où n est la taille de la file
  - Complexité en espace : $O(n)$
- Peut-on faire mieux ?

Exercice : comment obtenir $O(1)$ pour l'insertion et $O(n)$ pour l'extraction ?

Implémentation à l'aide d'un tas

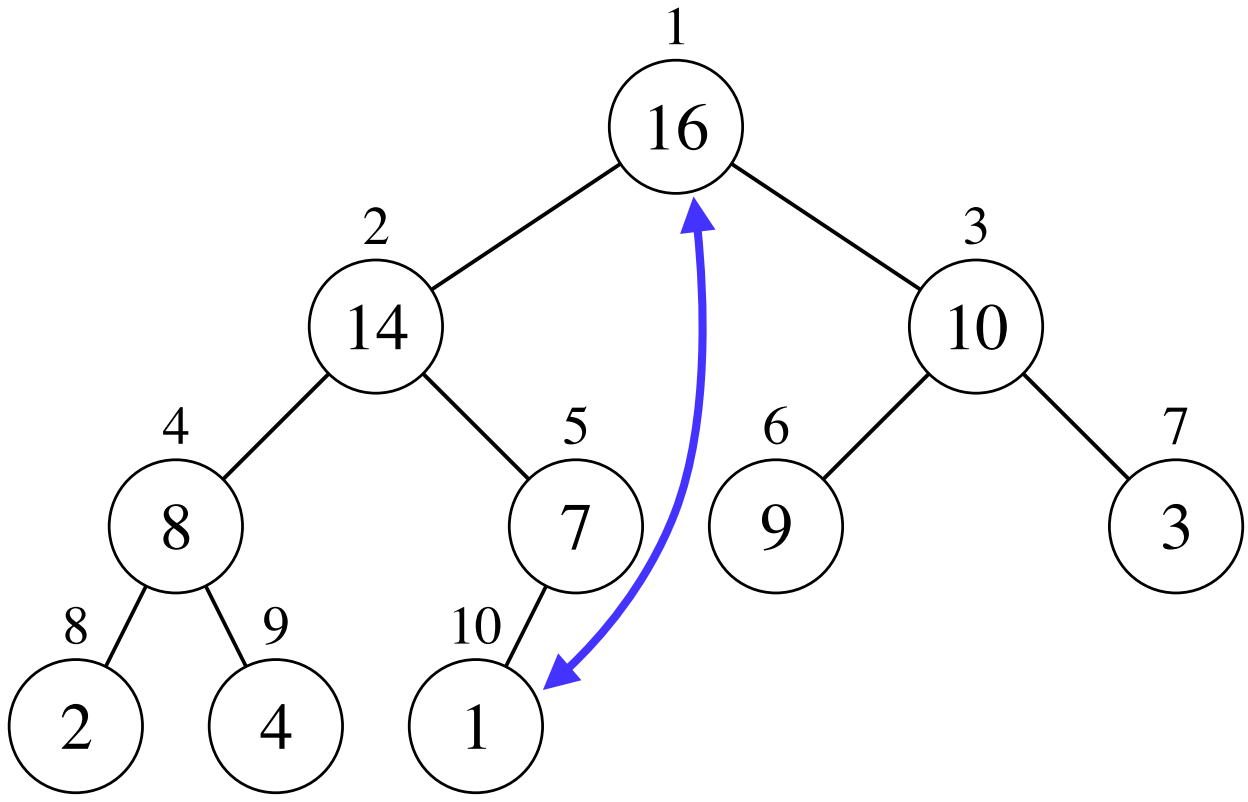
- La file est implémentée à l'aide d'un tas(-max) (voir transp. 164)
- Accès et extraction du maximum :

```
HEAP-MAXIMUM(A)
1  return A[1]
```

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1) // reconstruit le tas
7  return max
```

- Complexité : $\Theta(1)$ et $O(\log n)$ respectivement (voir chapitre 3)

Implémentation à l'aide d'un tas



Implémentation à l'aide d'un tas : insertion

- INCREASE-KEY(S, x, k) augmente la valeur de la clé de x à k (on suppose que $k \geq$ à la valeur courante de la clé de x).

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2        error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5        swap(A[i], A[PARENT(i)])
6        i = PARENT(i)
```

- Complexité : $O(\log n)$ (la longueur de la branche de la racine à i étant $O(\log n)$ pour un tas de taille n).

Implémentation à l'aide d'un tas : insertion

- Pour insérer un élément avec une clé *key* :
  - l'insérer à la dernière position sur le tas avec une clé $-\infty$,
  - augmenter sa clé de $-\infty$ à *key* en utilisant la procédure précédente

```
HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

- Complexité : $O(\log n)$.

$\Rightarrow$ Implémentation d'une file à priorité par un tas : $O(\log n)$ pour l'extraction et l'insertion.

Plan

Ensembles disjoints ("Union-Find")

- Structure de données qui maintient à jour une collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ d'ensembles dynamiques disjoints.
- Chaque ensemble est identifié par un représentant, qui est un élément quelconque de l'ensemble.

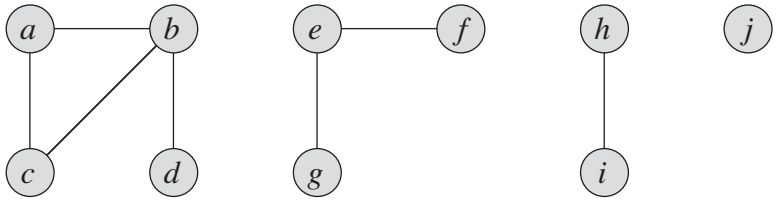
- Interface :
  - $\textsc{Make-Set}(D, x)$: crée un nouvel ensemble dont le seul membre, et donc représentant, est x. x ne doit pas déjà appartenir à un autre ensemble.
  - $\textsc{Union}(D, x, y)$: réunit les ensembles dynamiques qui contiennent x et y. x et y doivent appartenir à des ensembles différents avant l'union.
  - $\textsc{Find}(D, x)$: Renvoie un pointeur sur le représentant de l'ensemble (unique) contenant x. Chaque appel doit renvoyer le même représentant tant que la structure n'est pas modifiée.

Application

Calcul des composantes connexes d'un graphe

```
CONNECTED-COMPONENTS(G)
1  D = "Empty disjoint set"
2  for each vertex v ∈ G.V
3       MAKE-SET(D, v)
4  for each edge (u, v) ∈ G.E
5       if FIND-SET(D, u) ≠ FIND-SET(D, v)
6            UNION(D, u, v)
```

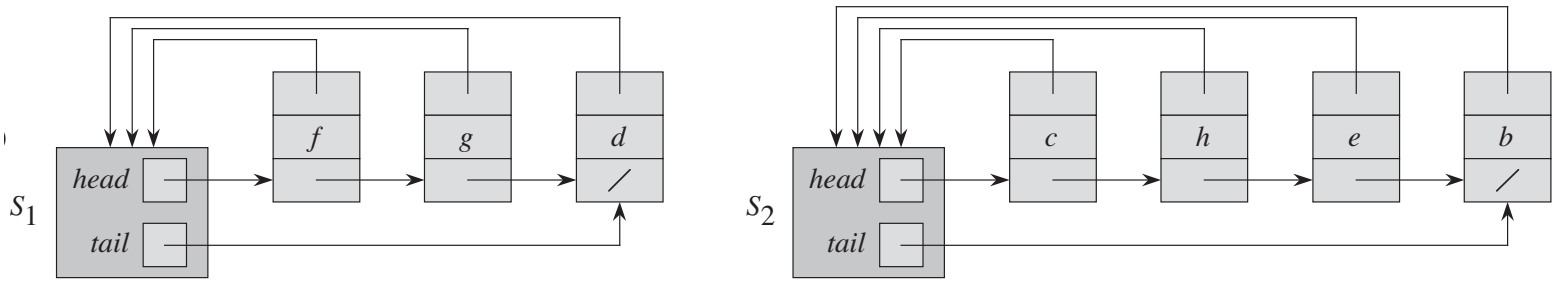
```
SAME-COMPONENT(u, v)
1  if FIND-SET(D, u) == FIND-SET(D, v)
2       return TRUE
3  else return FALSE
```



$$\mathcal{S} = \{\{a, b, c, d\}, \{e, f, g\}, \{h, i\}, \{j\}\}$$

Implémentation par une liste liée

- Chaque ensemble est représenté par une liste liée dont le représentant est le premier objet
- Une sentinelle pour chaque ensemble contient un pointeur vers le début de la liste (*head*) et un pointeur vers la fin (*tail*)
- Chaque objet de la liste contient un élément de l'ensemble, un pointeur sur l'objet contenant l'élément suivant, et un pointeur sur le représentant de l'ensemble.



- MAKE-SET : crée une nouvelle liste contenant seulement x.
- FIND : Suit le pointeur vers la sentinelle et puis renvoie le premier élément.
- $\Theta(1)$ pour les deux fonctions

Union : implémentation naïve

- On fusionne les listes contenant x et y :
  - On concatène la liste de y à la suite de la liste x
  - Nécessite de mettre à jour les pointeurs vers la sentinelle pour tous les éléments de la liste de y
  - Complexité linéaire en fonction de la taille de la liste contenant y
  - $\Theta(n)$ dans le pire cas si n appels à MAKE-SET ont précédé

- Analyse de complexité pour m opérations :
  - Supposons
    - m opérations (MAKE-SET, FIND, ou UNION)
    - n appels à MAKE-SET (et donc $n-1$ appels à UNION au plus)
    - Les n premières opérations sont des appels à MAKE-SET (pour simplifier l'analyse mais pas nécessaire)
  - Complexité au pire cas en fonction de m et n ?

Pire cas

En ignorant les appels à FIND :

Opération	Nombre d'objets mis à jour
MAKE-SET(x_1)	1
MAKE-SET(x_1)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
$\vdots$	$\vdots$
UNION(x_n, x_{n-1})	$n-1$

Nombre de mises à jour d'objets : $n + \sum_{i=1}^{n-1} i \Rightarrow \Theta(n^2)$ ($\Rightarrow$ coût amorti par opération : $\Theta(n)$)

En intercalant $m - 2n - 1$ appels à FIND ($O(1)$), on obtient $\Theta(m + n^2)$ pour le pire cas.

Union pondérée

Version plus efficace :

- On maintient la taille de chaque liste dans la sentinelle
- A chaque appel à UNION, on attache la liste la plus courte à la fin de la plus longue
- Complexité linéaire en fonction de la taille de la liste la plus courte
- Toujours $\Theta(n)$ dans le pire cas si n appels à MAKE-SET ont précédé

Analyse pour m opérations (dont n MAKE-SET) :

- Meilleur cas correspond au pire cas de l'approche naïve : $\Theta(m+n)$
- Pire cas ?

Borne supérieure sur le pire cas de l'union pondérée

Montrons que le pire cas est $O(m + n \log n)$

- Soit un objet x, combien de fois son pointeur vers la sentinelle sera-t-il mis à jour au plus ?
  - Quand il est mis à jour, la liste à laquelle il appartient est la plus courte des deux qui sont fusionnées
  - La taille de sa liste est donc au moins doublée
  - Si k mises à jour ont lieu, on doit donc avoir $2^k \leq n \Rightarrow k \leq \log(n)$
- Pour n éléments, les $n - 1$ opérations d'union demandent donc un temps $O(n \log(n))$.
- Pour m opérations au total : $O(m + n \log(n))$

(Exercice : Montrez que le pire correspond à fusionner à chaque étape les deux ensembles les plus petits)

Note : Il existe une implémentation à base d'arbres qui est $O(m \cdot \alpha(n))$ avec $\alpha(n)$ un fonction de croissante très lente en fonction de n ($\alpha(10^{80}) = 4$).

Ce qu'on a vu

- Quelques structures de données classiques et différentes implémentations pour chacune d'elles
  - Liste, files simples, doubles et à priorité
  - Listes, vecteurs, séquences
  - Arbres
  - Ensembles disjoints
- Analyse amortie pour un vecteur

Ce qu'on n'a pas vu

- Notion d'itérateur
- Tas binomial : alternative au tas binaire qui permet la fusion rapide de deux tas
- Evolution dynamique de la taille d'un tas (analyse amortie)
- Implémentation à base d'arbres d'une structure d'ensembles disjoints
- ...

Partie 5

Dictionnaires

Plan

1. Introduction
2. Arbres binaires de recherche
3. Tables de hachage

Dictionnaires

- Définition : un dictionnaire est un ensemble dynamique d'objets avec des clés comparables qui supportent les opérations suivantes :
  - $\textsc{Search}(S, k)$ retourne un pointeur x vers un élément dans S tel que $x.key = k$, ou NIL si un tel élément n'appartient pas à S.
  - $\textsc{Insert}(S, x)$ insère l'élément x dans le dictionnaire S. Si un élément de même clé se trouve déjà dans le dictionnaire, on met à jour sa valeur
  - $\textsc{Delete}(S, x)$ retire l'élément x de S. Ne fait rien si l'élément n'est pas dans le dictionnaire.
- Pour faciliter la recherche, on peut supposer qu'il existe un ordre total sur les clés.

Dictionnaires

- Deux objectifs en général :
  - minimiser le coût pour l'insertion et l'accès au données
  - minimiser l'espace mémoire pour le stockage des données
- Exemples d'applications :
  - Table de symboles dans un compilateur
  - Table de routage d'un DNS
  - ...
- Beaucoup d'implémentations possibles

Liste liée

Première solution :

- On stocke les paires clé-valeur dans une liste liée
- Recherche :

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL ∧ x.key ≠ k
3        x = x.next
4  return x
```

- Insertion (resp. Suppression)
  - On recherche la clé dans la liste
  - Si elle existe, on remplace la valeur (resp. on la supprime)
  - Si elle n'existe pas, on la place en début de liste (resp. on ne fait rien)
- Complexité au pire cas *(meilleur cas ?)*
  - Insertion : $\Theta(N)$
  - Recherche : $\Theta(N)$
  - Suppression : $\Theta(N)$

Vecteur trié

Deuxième solution :

- On suppose qu'il existe un ordre total sur les clés
- On stocke les éléments dans un vecteur qu'on maintient trié
- Recherche dichotomique (approche "diviser-pour-régner")

```
BINARY-SEARCH(V, k, low, high)
1  if low > high
2        return NIL
3  mid = ⌊(low + high)/2⌋
4  x = ELEM-AT-RANK(V, mid)
5  if k == x.key
6        return x
7  elseif k > x.key
8        return BINARY-SEARCH(V, k, mid + 1, high)
9  else return BINARY-SEARCH(V, k, low, mid − 1)
```

Complexité au pire cas : $\Theta(\log n)$

Vecteur trié

- Insertion : recherche de la position par BINARY-SEARCH puis insertion dans le vecteur par INSERT-AT-RANK (=décalage des éléments vers la droite).
- Suppression : recherche puis suppression par REMOVE-AT-RANK (=décalage des éléments vers la gauche).
- Complexité au pire cas *(meilleur cas ?)*
  - Insertion : $\Theta(N)$ (on doit décaler les éléments à droite de la clé)
  - Recherche : $\Theta(\log N)$ (recherche dichotomique)
  - Suppression : $\Theta(N)$ (on doit décaler les éléments à gauche de la clé)

  (Si le vecteur est implémenté par un tableau extensible !)

Dictionnaires : jusqu'ici

	Pire cas			*En moyenne*		
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$

Peut-on obtenir à la fois une insertion et une recherche "efficaces" ?

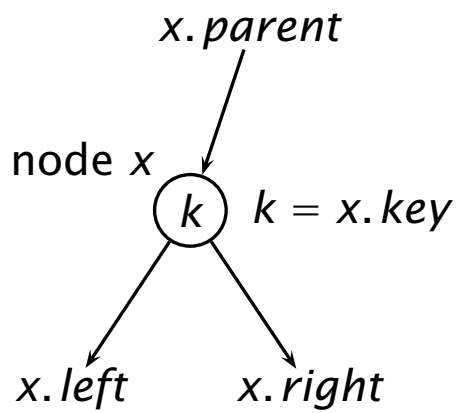
Plan

Plan

Implémentation des arbres binaires

Implémentation des arbres binaires dans ce chapitre (pour simplifier les algorithmes à venir) :

- T représente l'arbre, qui consiste en un ensemble de nœuds
- $T.root$ est le nœud racine de l'arbre T
- Nœud x
  - $x.parent$ est le parent du nœud x
  - $x.key$ est la clé stockée au nœud x
  - $x.left$ est le fils de gauche du nœud x
  - $x.right$ est le fils de droite du nœud x

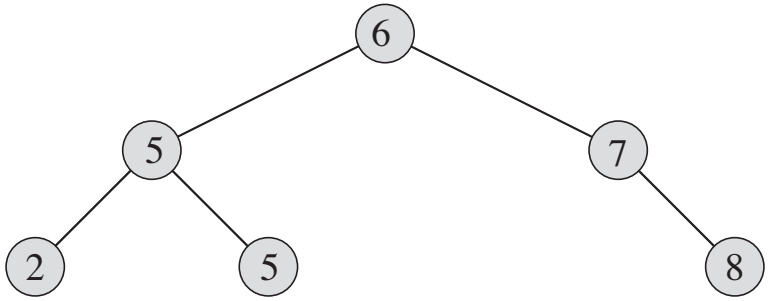


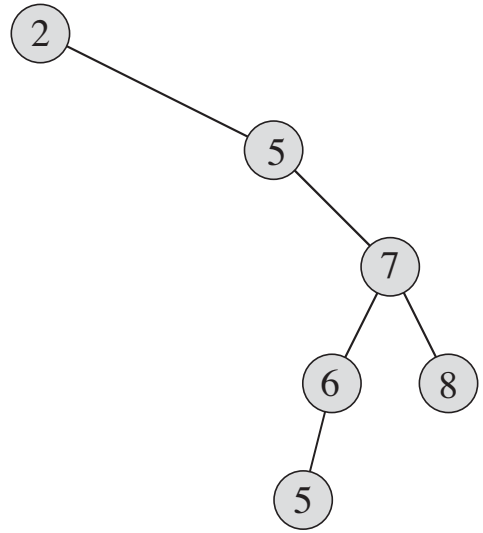
Pour simplifier les notations, nos fonctions seront implémentées directement sur base de cette implémentation (et pas de l'interface générale vue précédemment)

Arbres binaires de recherche

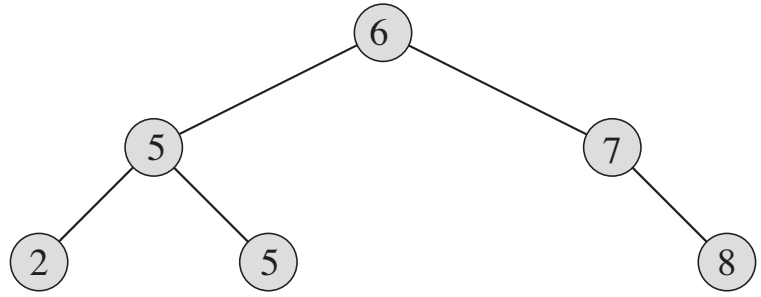
- Une structure d'arbre binaire implémentant un dictionnaire, avec des opérations en $O(h)$ où h est la hauteur de l'arbre

- Chaque nœud de l'arbre binaire est associé à une clé
- L'arbre satisfait à la propriété d'arbre binaire de recherche
  - Soient deux nœuds x et y.
  - Si y est dans le sous-arbre de gauche de x, alors $y.key < x.key$
  - Si y est dans le sous-arbre de droite de x, alors $y.key \geq x.key$





Parcours d'un arbre binaire de recherche



$$\Rightarrow \langle 2, 5, 5, 6, 7, 8 \rangle$$

- Le parcours infixe d'un arbre binaire de recherche permet d'afficher les clés par ordre croissant

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2        INORDER-TREE-WALK(x.left)
3        print x.key
4        INORDER-TREE-WALK(x.right)
```

Recherche dans un arbre binaire

- Recherche binaire

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2        return x
3  if k < x.key
4        return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

Appel initial (à partir d'un arbre T)

```
TREE-SEARCH(T.root, k)
```

- Complexité ? $T(n) \in O(h)$, où h est la hauteur de l'arbre
- Pire cas : $h = n$

Recherche dans un arbre binaire

- TREE-SEARCH est récursive terminale.
- Version itérative

```
ITERATIVE-TREE-SEARCH(T, k)
1  x = T.root
2  while x ≠ NIL and k ≠ x.key
3        if k < x.key
4              x = x.left
5        else x = x.right
6  return x
```

Clés maximale et minimale

- Etant donné la propriété d'arbre binaire
  - La clé minimale se trouve dans le nœud le plus à gauche
  - La clé maximale se trouve dans le noœud le plus à droite

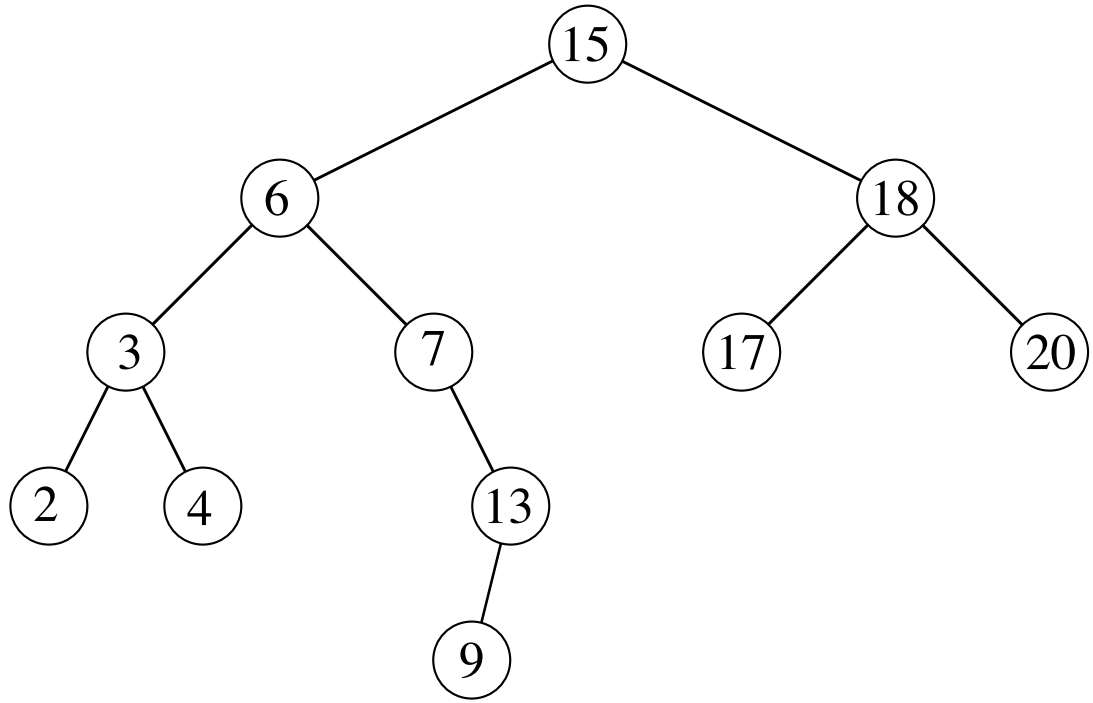
```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2        x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2        x = x.right
3  return x
```

- Complexité : $O(h)$, où h est la hauteur de l'arbre.

Successeur et prédécesseur

- Etant donné un nœud x, trouver le nœud contenant la valeur de clé suivante (dans l'ordre)

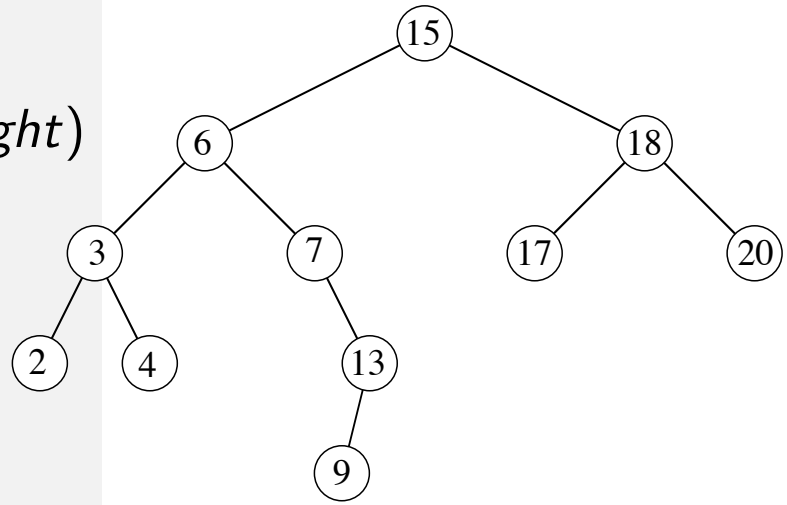


Ex : successeur de 15 $\rightarrow$ 17, successeur de 4 $\rightarrow$ 6.

- Le successeur de x est le minimum du sous-arbre de droite s'il existe
- Sinon, c'est le premier ancêtre a de x tel que x tombe dans le sous-arbre de gauche de a.

Successeur et prédécesseur

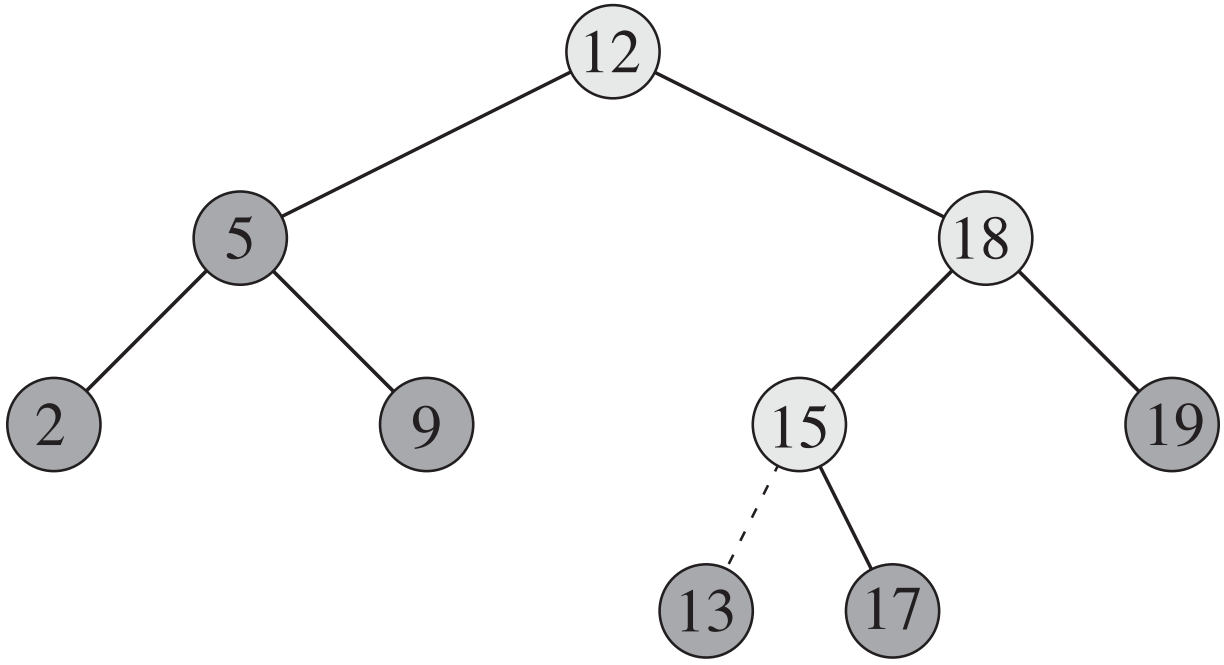
```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2        return TREE-MINIMUM(x.right)
3  y = x.parent
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.parent
7  return y
```



Complexité : $O(h)$, où h est la hauteur de l'arbre

(Exercice : TREE-PREDECESSOR*)*

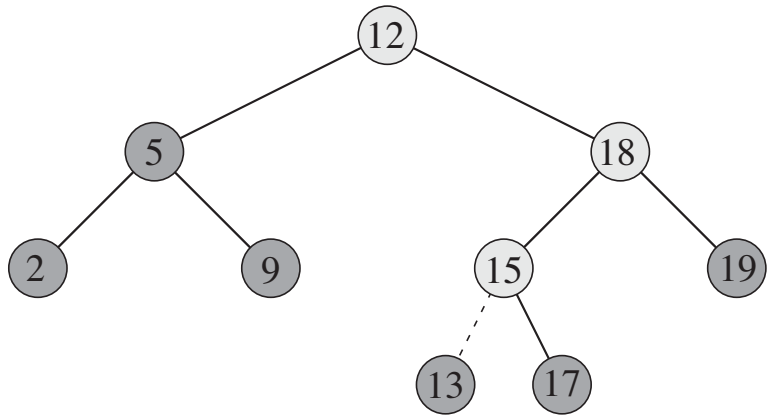
Insertion



- Pour insérer x, on recherche la clé $x.key$ dans l'arbre
- Si on ne la trouve pas, on l'ajoute à l'endroit où la recherche s'est arrêtée.

Insertion

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y == NIL
10        // Tree T was empty
11        T.root = z
12  elseif z.key < y.key
13        y.left = z
14  else y.right = z
```

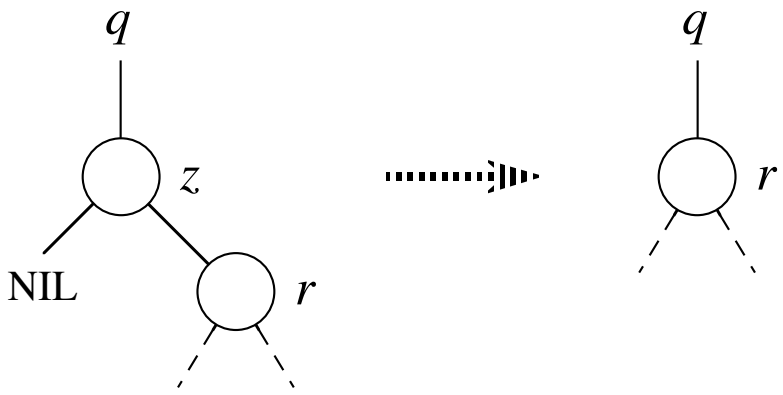


Complexité : $O(h)$ où h est la hauteur de l'arbre

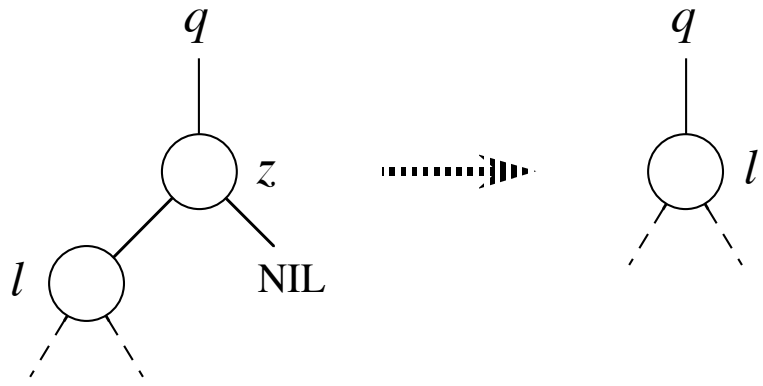
Suppression

3 cas à considérer en fonction du nœud z à supprimer :

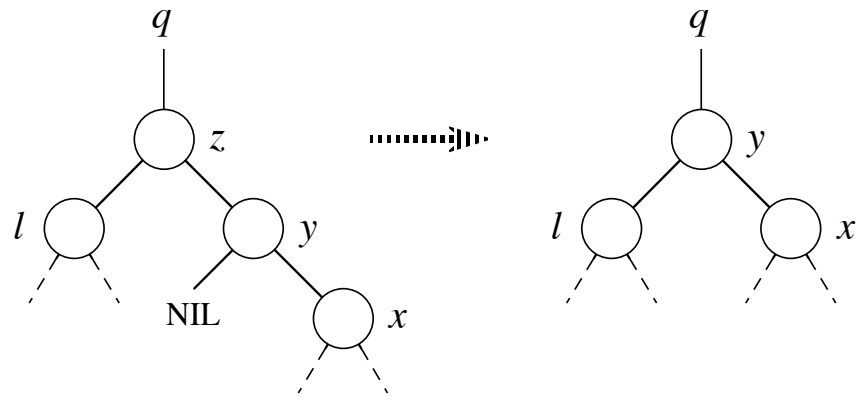
- z n'a pas de fils gauche : remplacer z par son fils droit



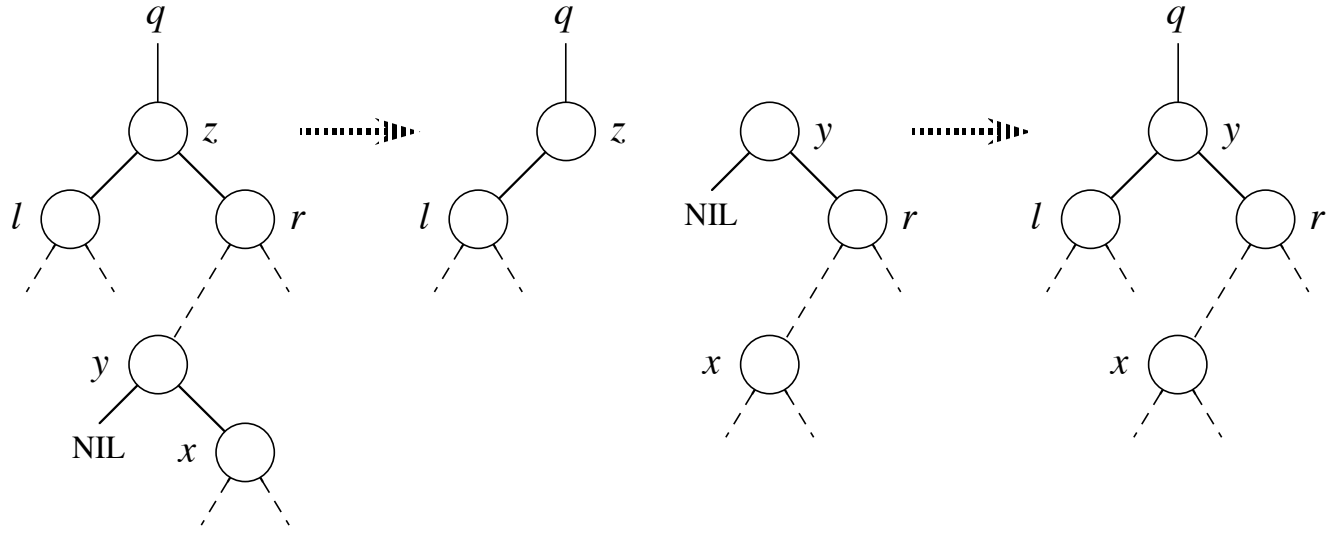
- z n'a pas de fils droit : remplacer z par son fils gauche



- z a deux fils : rechercher le successeur y de z.
  NB : y est dans le sous-arbre de droite et n'a pas de fils gauche.
  - Si y est le fils droit de z, remplacer z par y et conserver le fils droit de y



  - Sinon, y est dans le sous-arbre droit de z mais n'en est pas la racine. On remplace y par son propre fils droit et on remplace z par y.



Suppression

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2        TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSPLANT(T, z, z.left)
 5  else // z has two children
 6        y = TREE-SUCCESSOR(z)
 7        if y.parent ≠ z
 8              TRANSPLANT(T, y, y.right)
 9              y.right = z.right
10              y.right.parent = y
11        // Replace z by y
12        TRANSPLANT(T, z, y)
13        y.left = z.left
14        y.left.parent = y
```

```
TRANSPLANT(T, u, v)
1  if u.parent == NIL
2        T.root = v
3  elseif u == u.parent.left
4        u.parent.left = v
5  else u.parent.right = v
6  if v ≠ NIL
7        v.parent = u.parent
```

Complexité : $O(h)$ pour un arbre de hauteur h
(Tout est $O(1)$ sauf l'appel à TREE-SUCCESSOR).

Arbres binaires de recherche

- Toutes les opérations sont $O(h)$ où h est la hauteur de l'arbre
- Si n éléments ont été insérés dans l'arbre :
  - Au pire, $h = n - 1 \in \Theta(n)$
    - Elements insérés en ordre
  - Au mieux, $h = \lceil \log_2 n \rceil \in \Theta(\log n)$
    - Pour un arbre binaire complet
  - En moyenne, on peut montrer que $h \in \Theta(\log n)$
    - En supposant que les éléments ont été insérés en ordre aléatoire

Profondeur moyenne d'un nœud

- Montrons que la profondeur moyenne d'un nœud dans un ABR construit aléatoirement est $\Theta(\log n)$
- Soit $P(n)$ la somme totale moyenne des profondeurs des nœuds d'un ABR construit à partir de n clés introduites dans un ordre aléatoire. La profondeur moyenne recherchée est $P(n)/n$.
- On a :

$$P(n) = \sum_{i=0}^{n-1} \frac{1}{n}(P(i) + P(n-i-1) + n - 1), P(1) = 0$$

  En effet :

  - La première clé introduite dans l'ABR a autant de chance d'être à chaque position de la liste triée des clés.
  - Si elle est à la position $i+1$ le sous-arbre de gauche contient i clés et celui de droite $n-i-1$ clés.
  - La profondeur de chaque nœud à droite et à gauche de la racine est augmentée de 1 par rapport à sa profondeur dans les sous-arbres à gauche et à droite.

Profondeur moyenne d'un nœud

- Cette récurrence peut se réécrire en :

$$P(n) = \sum_{i=0}^{n-1} \frac{1}{n}(P(i) + P(n-i-1)) + n - 1.$$

  qui est identique à celle du QUICKSORT (voir le transp. 151)
- On en déduit :

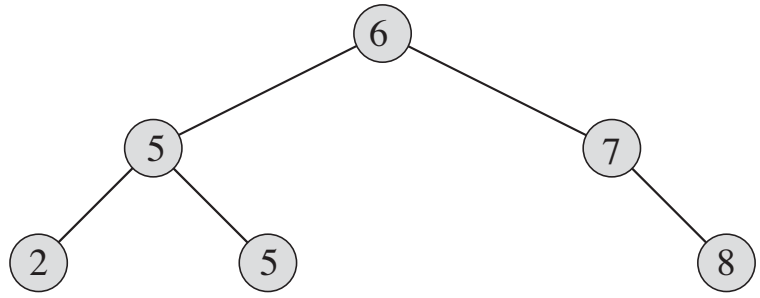
$$\Rightarrow \quad P(n) \in \Theta(n \log n)$$
$$\Rightarrow \quad \frac{P(n)}{n} \in \Theta(\log n)$$

- En moyenne, la recherche d'une clé parmi celles stockées dans un ABR obtenu après insertion des clés dans un ordre aléatoire est donc $\Theta(\log n)$

Tri avec un arbre binaire de recherche

```
BINARY-SEARCH-TREE-SORT(A)
1  T = "Empty binary search tree"
2  for i = 1 to n
3        TREE-INSERT(T, A[i])
4  INORDER-TREE-WALK(T.root)
```

- Exemple : $A = [6, 5, 7, 2, 5, 8]$



- Complexité en temps identique au quicksort
  - Insertion : en moyenne, $n \cdot O(\log n) = O(n \log n)$, pire cas : $\Theta(n^2)$
  - Parcours de l'arbre en ordre : $\Theta(n)$
  - Total : $\Theta(n \log n)$ en moyenne, $\Theta(n^2)$ pour le pire cas
- Complexité en espace cependant plus importante, pour le stockage de la structure d'arbres.

Dictionnaires : jusqu'ici

	Pire cas			*En moyenne*		
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

- Peut-on obtenir $\Theta(\log n)$ dans le pire cas ? Oui !
- Deux solutions :
  - Utiliser de la randomisation pour que le probabilité de rencontrer le pire cas soit négligeable
  - Maintenir les arbres équilibrés

Plan

Arbres équilibrés

- Solution générale pour obtenir une complexité au pire cas en $\Theta(\log n)$:
  - Définir un invariant sur la structure d'arbre
  - Prouver que cet invariant garantit une hauteur $\Theta(\log n)$
  - Implémenter les opérations d'insertion et suppression de manière à maintenir l'invariant
  - Si ces opérations ne sont pas trop coûteuses (p.ex., $O(\log n)$), on aura gagné

- Plusieurs types d'arbres équilibrés :
  - Arbres AVL
    - Invariant : Arbres H-équilibrés
  - Arbres rouges et noirs (voir INFO0027)
  - Arbres 2-3-4
  - Splay trees, Scapegoat trees, treaps...

Arbres H-équilibrés

- Définition :

$$T \text{ est } H-\text{équilibré} \Leftrightarrow |h(g(T')) - h(d(T'))| \leq 1,$$

pour tout sous-arbre T' de T, et où $g(X)$, $d(X)$ et $h(X)$ sont resp. le sous-arbre gauche, le sous-arbre droit et la hauteur [2] de l'arbre X.

(Les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un)

- Propriété :
Pour tout arbre H-équilibré de taille n et de hauteur h, on a

$$h \in \Theta(\log n)$$

Plus précisément, on peut prouver :

$$\log(n+1) \leq h+1 < 1,44 \log(n+2)$$

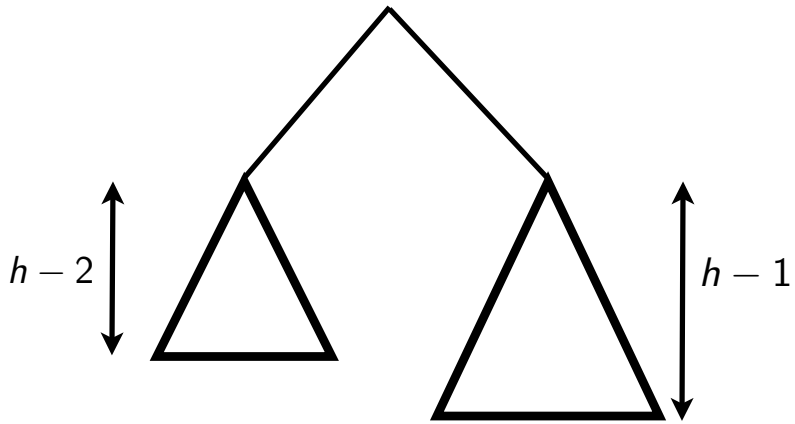
2. Par définition, la hauteur d'un arbre vide est -1.

Arbres H-équilibrés

Démonstration

Etant donné un arbre H-équilibré de taille n et de hauteur $h \geq 1$, pour h fixé, n est

- Maximum : quand l'arbre est complet, soit quand $n = 2^{h+1} - 1 \Rightarrow n + 1 \leq 2^{h+1} \Rightarrow \log(n+1) \leq h + 1 \Rightarrow h \in \Omega(\log n)$
- Minimum : quand $n = N(h)$ où $N(h)$ est la taille d'un arbre H-équilibré de hauteur h qui a le moins d'éléments.
  - $N(h)$ peut être défini par récurrence par $N(h) = 1 + N(h-1) + N(h-2)$ avec $N(0) = 1$ et $N(1) = 2$.



- On a donc
$$\begin{aligned} & N(h) = 1 + N(h-1) + N(h-2) \\ \Rightarrow \quad & N(h) > 2N(h-2) \text{ (car } N(h-1) > N(h-2)) \\ \Rightarrow \quad & N(h) > 2^{h/2} \\ \Rightarrow \quad & h < 2 \log N(h) \end{aligned}$$
- dont on peut tirer que $h \in O(\log n)$

- On en déduit que
$$h = \Theta(\log n)$$

□

Borne supérieure plus précise

- En notant $F(h) = N(h) + 1$, on a $F(h) = F(h-1) + F(h-2)$ avec $F(0) = 2$, $F(1) = 3$
- F est un récurrence de Fibonacci qui a pour solution

$$F(h) = \frac{1}{\sqrt{5}}(\phi^{h+3} - \phi'^{h+3}) \text{ avec } \phi = \frac{1+\sqrt{5}}{2} \text{ et } \phi' = \frac{1-\sqrt{5}}{2}$$

- On a

$$N(h) + 1 = \frac{1}{\sqrt{5}}(\phi^{h+3} - \phi'^{h+3})$$

- ce qui donne

$$n + 1 \geq \frac{1}{\sqrt{5}}(\phi^{h+3} - \phi'^{h+3}) > \frac{1}{\sqrt{5}}(\phi^{h+3} - 1)$$

(car $|\phi'| < 1$)

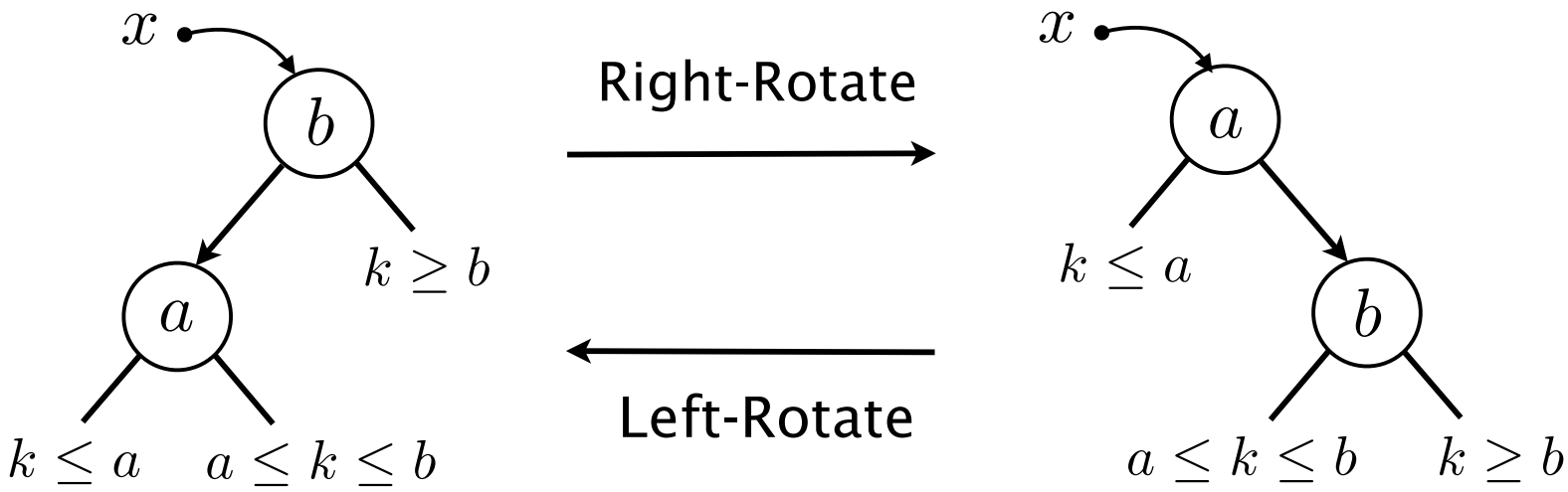
- En prenant le $\log_\phi$ des deux membres :

$$h + 1 < 1,44 \log(n+2)$$

Arbres AVL

- Définition : Un arbre AVL est un arbre binaire de recherche H-équilibré
- Inventé par Adelson-Velskii et Landis en 1960
- Recherche :
  - Par la fonction TREE-SEARCH puisque c'est un arbre binaire
  - Complexité $\Theta(\log n)$ étant donné la propriété
- Insertion :
  - On insère l'élément comme dans un arbre binaire classique
  - On vérifie que l'invariant est respecté
  - Si ce n'est pas le cas, on modifie l'arbre

Rotations

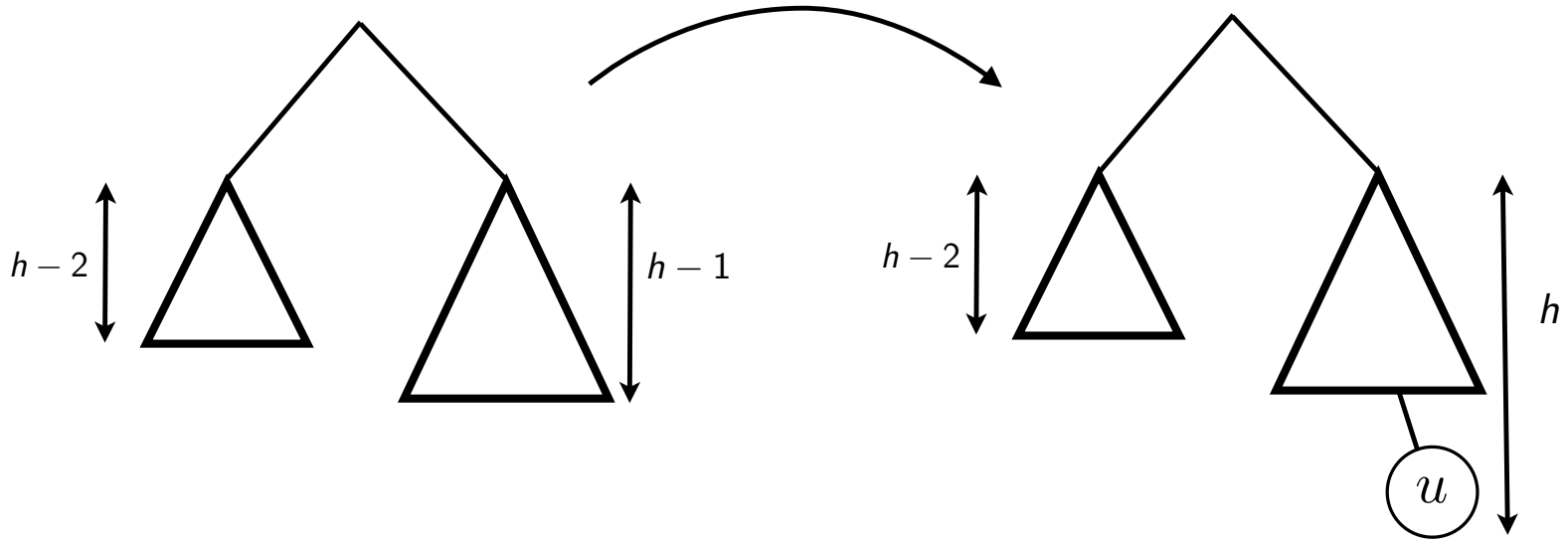



```
LEFT-ROTATE(x)
1  r = x.right
2  x.right = r.left
3  r.left = x
4  return r
```

```
RIGHT-ROTATE(x)
1  l = x.left
2  x.left = l.right
3  l.right = x
4  return l
```

Les rotations maintiennent la propriété d'arbre binaire de recherche

Insertion dans un AVL

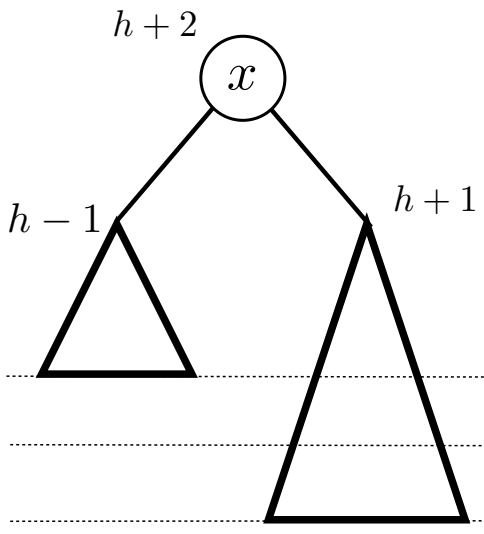


- Insérer le nouvel élément comme dans un arbre binaire de recherche ordinaire
- L'insertion peut créer un déséquilibre (l'arbre n'est plus H-équilibré)
- Remonter depuis le nouveau nœud jusqu'à la racine en restaurant l'équilibre des sous-arbres rencontrés si nécessaire

Equilibrage

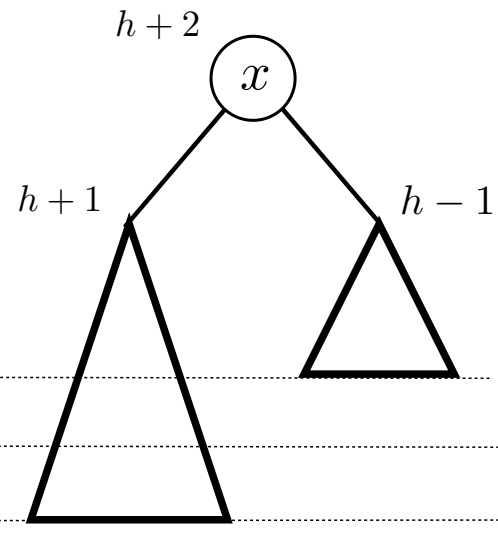
- Soit x le nœud le plus bas violant l'invariant après l'insertion
  - Tous ses sous-arbres sont H-équilibrés
  - Il y a une différence d'au plus 2 niveaux entre ses sous-arbres gauche et droit
- Comment rétablir l'équilibre ?
- Deux cas possibles (selon insertion à droite ou à gauche) :

Cas 1



(Déséquilibre à droite)

Cas 2

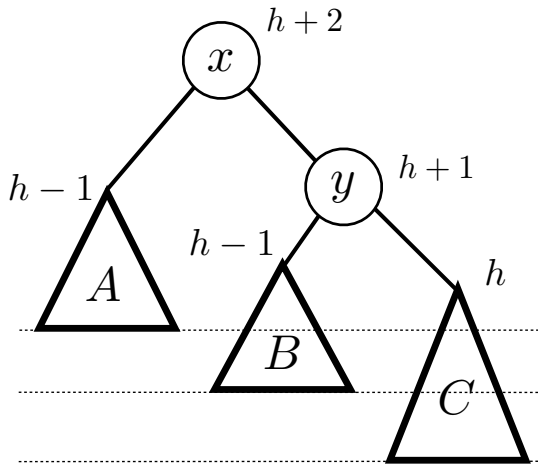


(Déséquilibre à gauche)

Cas 1 : déséquilibre à droite

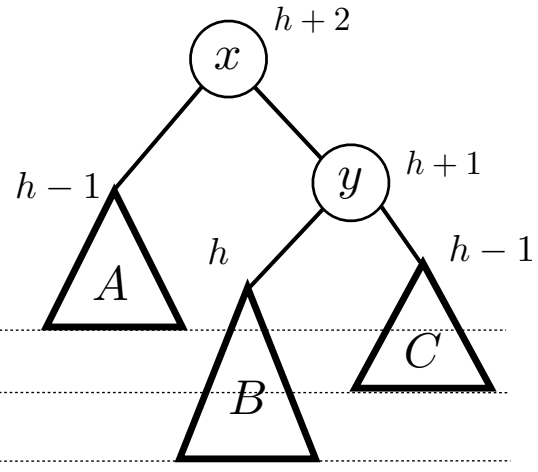
- Deux sous-cas possibles

Cas 1.1



Déséquilibre à l'extérieur
(Cas droite-droite)

Cas 1.2

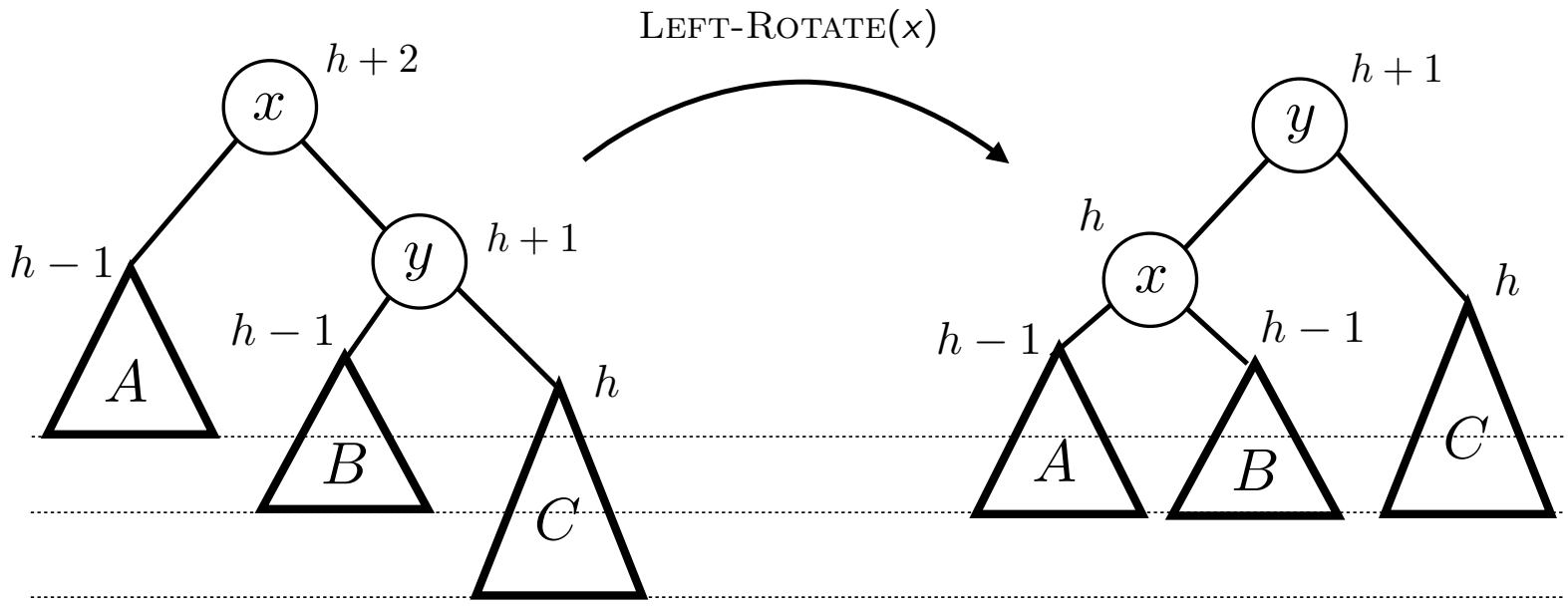


Déséquilibre à l'intérieur
(Cas droite-gauche)

(Pourquoi le cas B et C de hauteur h n'est pas possible ?)

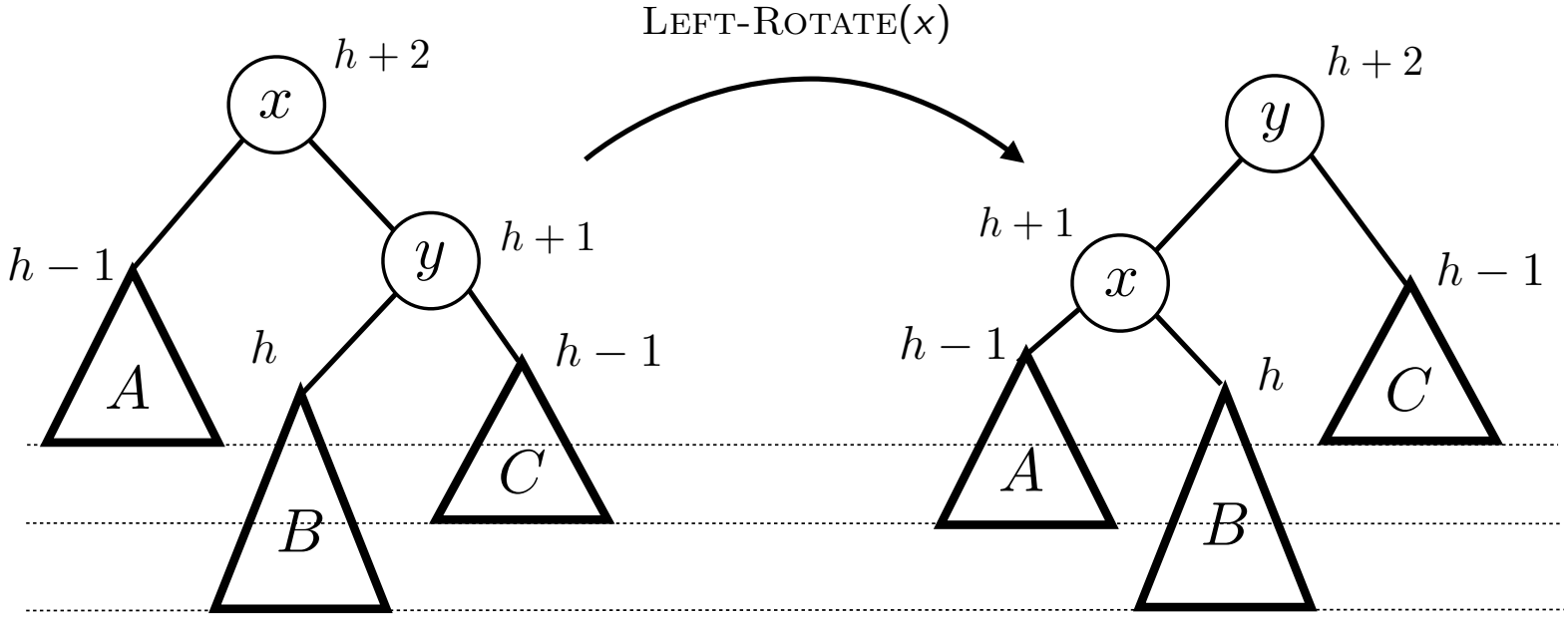
Cas 1.1 : déséquilibre à droite, extérieur (droite-droite)

- Equilibre rétabli par une rotation à gauche de x

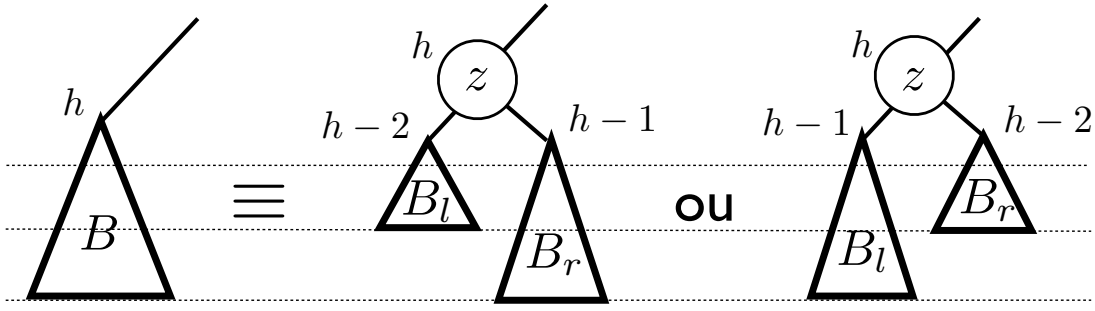


Cas 1.2 : déséquilibre à droite, intérieur (droite-gauche)

- Une rotation à gauche ne permet pas de rétablir l'équilibre

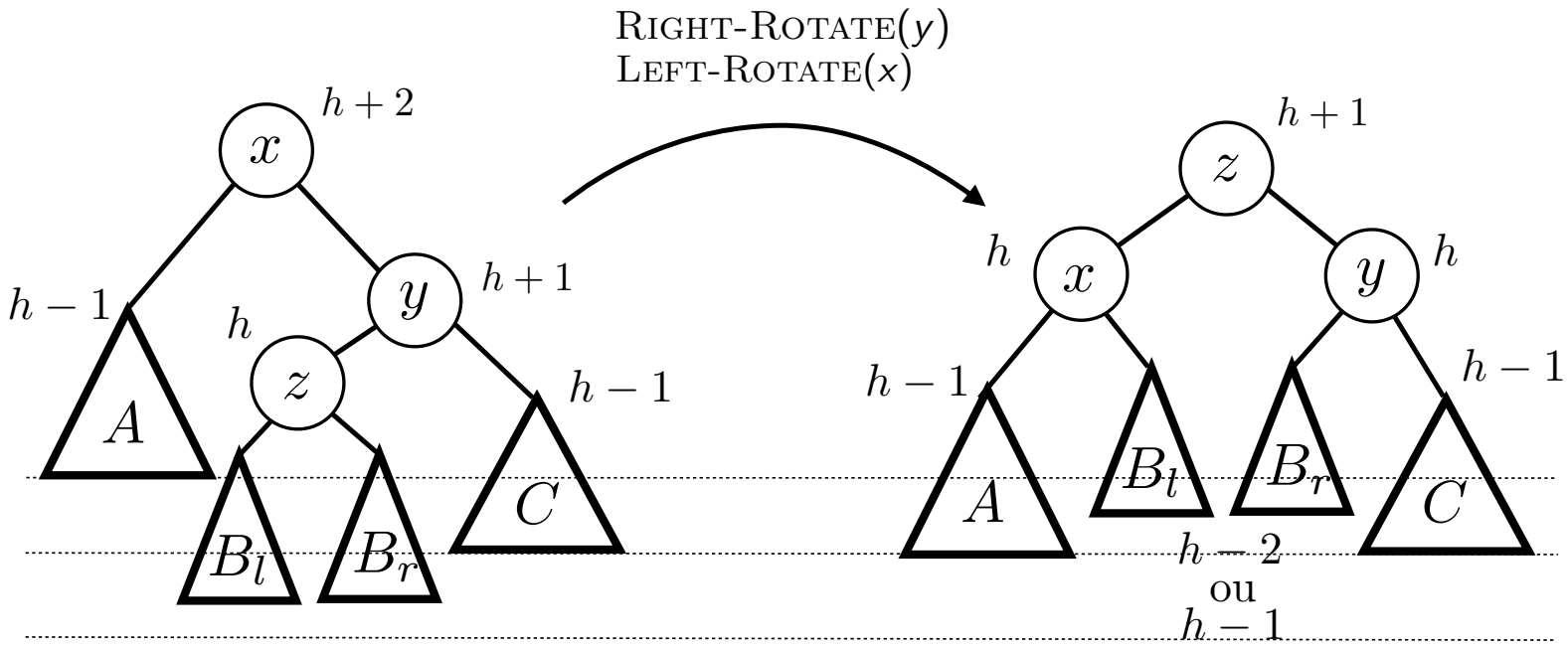


- Le sous-arbre B contient au moins un élément (l'élément inséré)



Cas 1.2 : déséquilibre à droite, intérieur (droite-gauche)

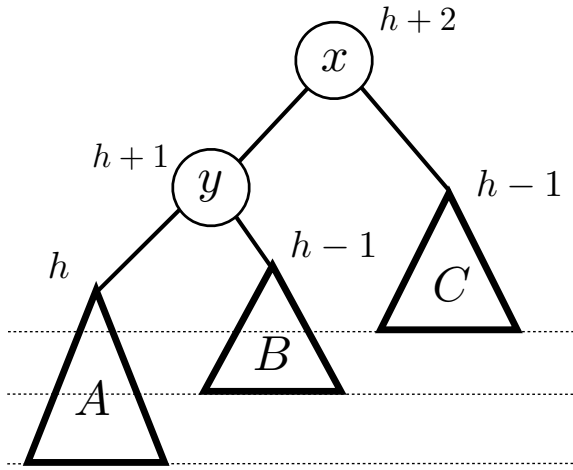
- Equilibre rétabli par deux rotations



Cas 2 : déséquilibre à gauche

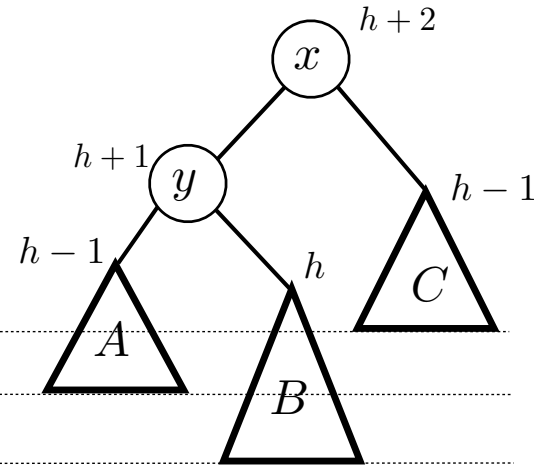
- Symétrique du cas 1
- Deux sous-cas possibles

Cas 2.1



Déséquilibre à l'extérieur
(Cas gauche-gauche)

Cas 2.2



Déséquilibre à l'intérieur
(Cas gauche-droite)

- Résolus respectivement par une rotation (à droite) et une double rotation.

Implémentation

- Algorithme récursif : Pour insérer une clé dans un arbre T :
  - On l'insère (récursivement) dans le sous-arbre approprié (gauche ou droit)
  - Si l'arbre résultant T devient déséquilibré, on effectue une rotation simple ou double selon le cas dans lequel on se trouve
- L'arbre après rééquilibrage étant de la même hauteur qu'avant l'insertion, on n'aura à faire qu'au plus une rotation (simple ou double).
- L'implémentation est facilitée si on maintient en chaque nœud x un attribut $x.h$ avec la hauteur du sous-arbre en x.
- Complexité :
  - $O(h)$ où h est la hauteur de l'arbre,
  - c'est-à-dire $O(\log n)$ vu que l'arbre est H-équilibré.

Suppression

- Comme pour l'insertion, on doit rétablir l'équilibre suite à la suppression
- La suppression d'un nœud peut déséquilibrer le parent de ce nœud
- Contrairement à l'insertion, on peut devoir rééquilibrer plusieurs ancêtres du nœud supprimé.
- Chaque rotation étant d'ordre $O(1)$, la complexité d'une suppression reste cependant $O(h)$ pour un arbre de hauteur h et donc $O(\log n)$ pour un AVL.

Tri avec un AVL

- Comme avec un arbre de binaire de recherche ordinaire, on peut trier avec un AVL
  - On insère les éléments successivement dans l'arbre
  - On effectue un parcours en ordre de l'arbre
- Complexité en temps : $\Theta(n \log n)$ (comme pour le tri par tas)
- Complexité en espace : $\Theta(n)$ (pour la structure d'arbre temporaire) (versus $O(1)$ pour le heap-sort)

Dictionnaires : jusqu'ici

	Pire cas			*En moyenne*		
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
AVL	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

- Peut-on faire mieux ?
- Oui, en changeant radicalement de philosophie

Demo

Illustrations :

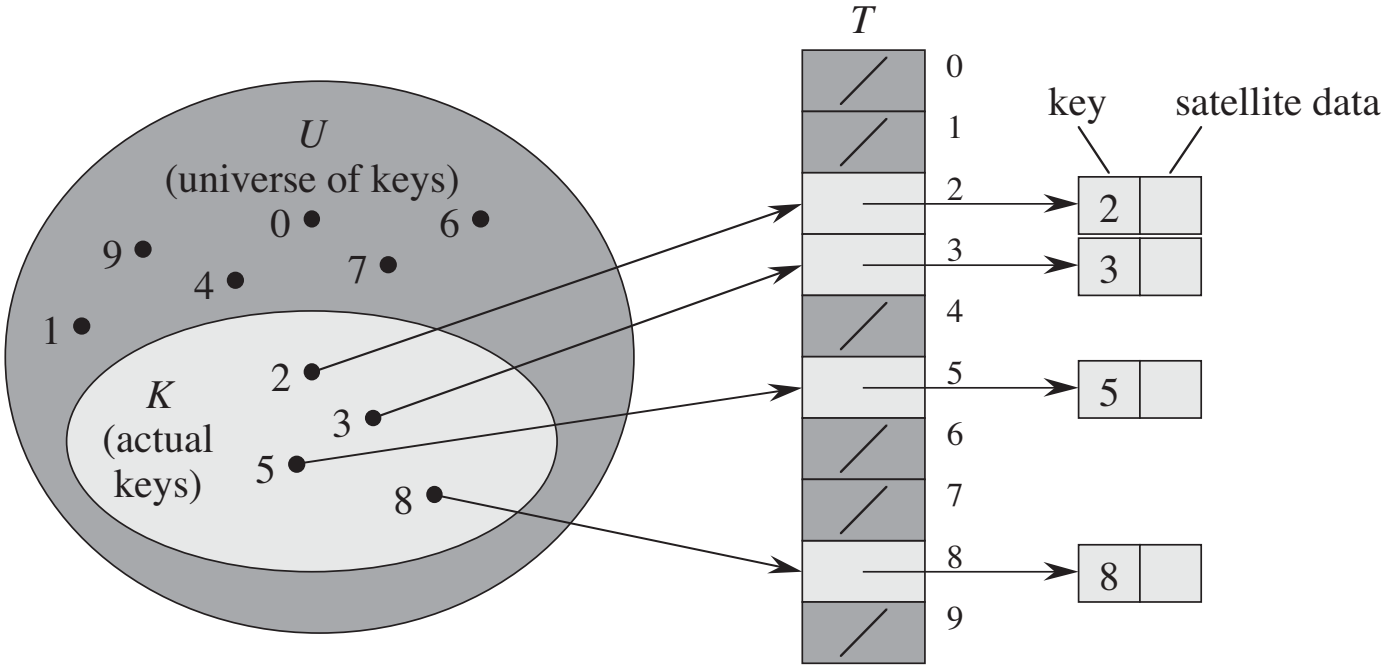
- `http://people.ksp.sk/~kuko/bak/`
- `http://www.csi.uottawa.ca/~stan/csi2514/applets/avl/BT.html`
- `http://www.cs.jhu.edu/~goodrich/dsa/trees/avltree.html`
- `http://www.cs.usfca.edu/~galles/visualization/flash.html`

Plan

Tableau à accès direct

- On suppose :
  - que chaque élément a une clé tirée d'un univers $U = \{0, 1, \ldots, m-1\}$ où m n'est pas trop grand
  - qu'il ne peut pas y avoir deux éléments avec la même clé.
- Le dictionnaire est implémenté par un tableau $T[0 \ldots m-1]$:
  - Chaque position dans la table correspond à une clé de U.
  - S'il y a un élément x avec la clé k, alors $T[k]$ contient un pointeur vers x.
  - Sinon, $T[k]$ est vide ($T[k] = \mathrm{NIL}$).

Tableau à accès direct




```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]
```

```
DIRECT-ADDRESS-INSERT(T, x)
1  return T[x.key] = x
```

```
DIRECT-ADDRESS-DELETE(T, x)
1  return T[x.key] = NIL
```

Tableau à accès direct

- Complexité de toutes les opérations : $O(1)$ (dans tous les cas)
- Problème :
  - Complexité en espace : $\Theta(|U|)$
  - si l'univers de clés U est grand, stocker une table de taille $|U|$ peut être peu pratique, voire impossible
- Souvent l'ensemble des clés réellement stockées, noté K, est petit comparé à U et donc l'espace alloué est gaspillé.

- Comment bénéficier de l'accès rapide d'une table à accès direct avec une table de taille raisonnable ?

  $\Rightarrow$ Table de hachage :
  - Réduit le stockage à $\Theta(|K|)$
  - Recherche en $O(1)$ (en moyenne !)

Table de hachage

- Inventée en 1953 par Luhn
- Idée :
  - Utiliser une table T de taille $m \ll |U|$
  - stocker x à la position $h(x.key)$, où h est une fonction de hachage :

$$h : U \to \{0, \ldots, m-1\}$$

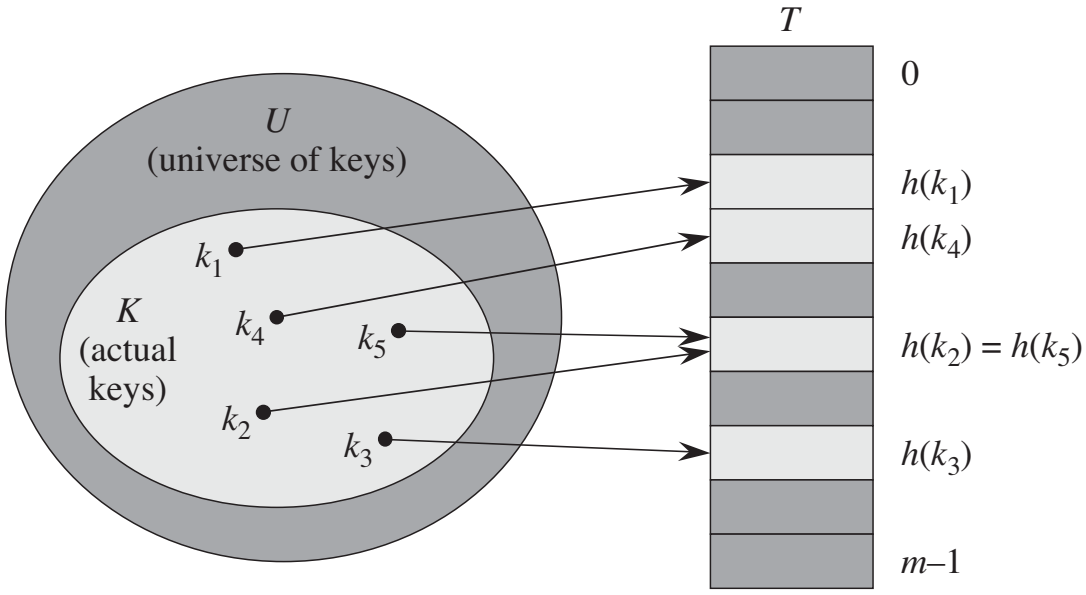
```
HASH-INSERT(T, x)
1  T[h(x.key)] = x
```

```
HASH-DELETE(T, x)
1  T[h(x.key)] = NIL
```

```
HASH-SEARCH(T, x)
1  return T[h(x.key)]
```

Est-ce que ces algorithmes sont corrects ?

Table de hachage : collisions



- Collision : lorsque deux clés distinctes k_1 et k_2 sont telles que $h(k_1) = h(k_2)$
- Cela se produit toujours lorsque le nombre de clés observées est plus grand que la taille du tableau T ($|K| > m$)
- Très probable, même lorsque la fonction de hachage répartit les clés uniformément $\Rightarrow$ Paradoxe des anniversaires

Paradoxe des anniversaires

- Hypothèse :
  - On néglige les années bissextiles
  - Les 365 jours présentent la même probabilité d'être un jour d'anniversaire
- Si p est la probabilité d'une collision d'anniversaires :

$$1 - p = \frac{364}{365} \cdot \frac{363}{365} \cdot \frac{362}{365} \cdots \frac{365 - (n-1)}{365} = \frac{365!}{(365-n)!365^n}$$

  Exemples :
  - $n = 23 \Rightarrow p > 0,5$
  - $n = 57 \Rightarrow p > 0,99$
  - $n = 70 \Rightarrow p > 0,999$

- Pour une table de hachage :
  - $m = 365$ et 57 clés $\Rightarrow$ plus de 99% de chance de collision
  - $m = 1000000$ et 2500 clés $\Rightarrow$ plus de 95% de chance de collision

Collision

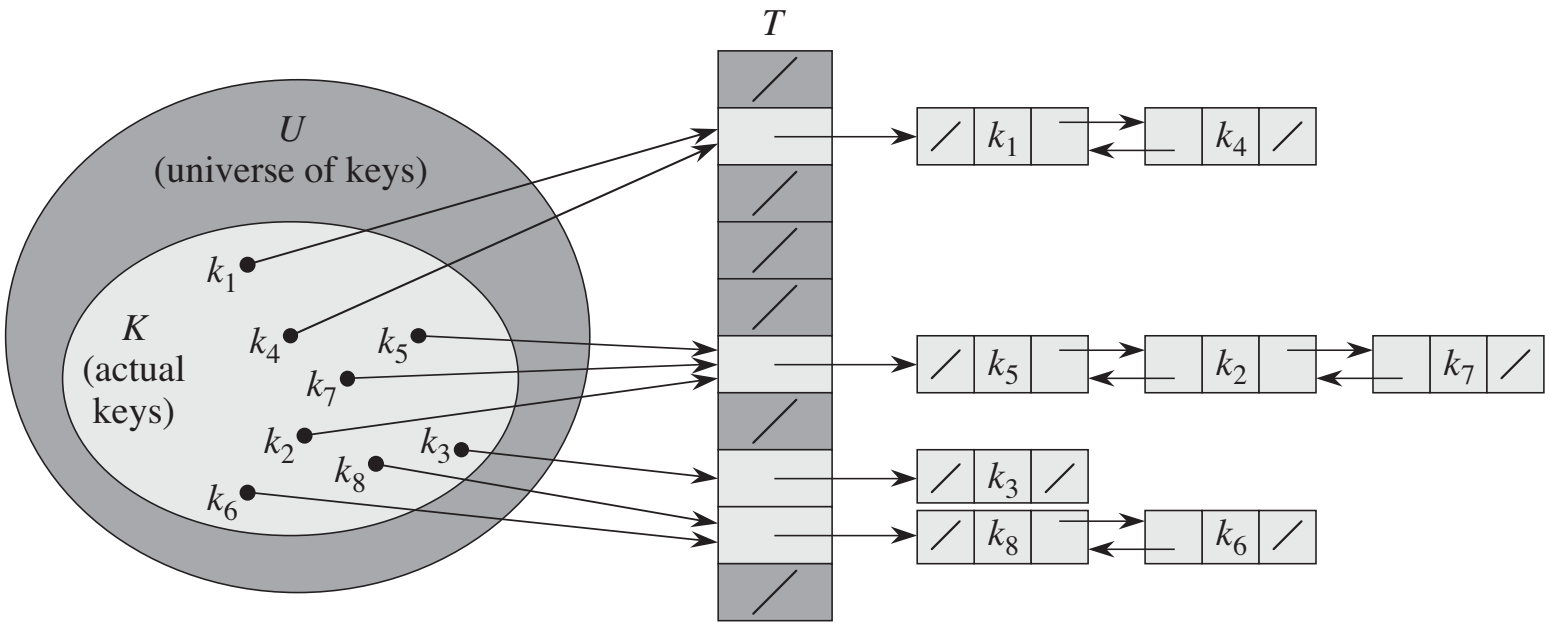
- Pour éviter les collisions :
  - on veille à utiliser une fonction de hachage qui disperse le plus possible les clés vers les différents compartiments.
  - on utilise un nombre de compartiments suffisamment grand

  Cependant, même dans ce cas, la probabilité de collision peut être non négligeable.

- Deux approches pour prendre en compte les collisions :
  - Le chaînage
  - Le sondage (adressage ouvert)

Résolution des collisions par chaînage

Solution : mettre les éléments qui sont “hachés” vers la même position dans une liste liée (simple ou double)



Implémentation des opérations

```
CHAINED-HASH-INSERT(T, x)
1  LIST-INSERT(T[h(x.key)], x)
```

```
CHAINED-HASH-DELETE(T, x)
1  LIST-DELETE(T[h(x.key)], x)
```

```
CHAINED-HASH-SEARCH(T, k)
1  return LIST-SEARCH(T[h(k)], k)
```

- Complexité :
  - Insertion : $O(1)$
  - Suppression : $O(1)$ si liste doublement liée, $O(n)$ pour une liste de taille n si liste simplement liée.
  - Recherche : $O(n)$ si liste de taille n.

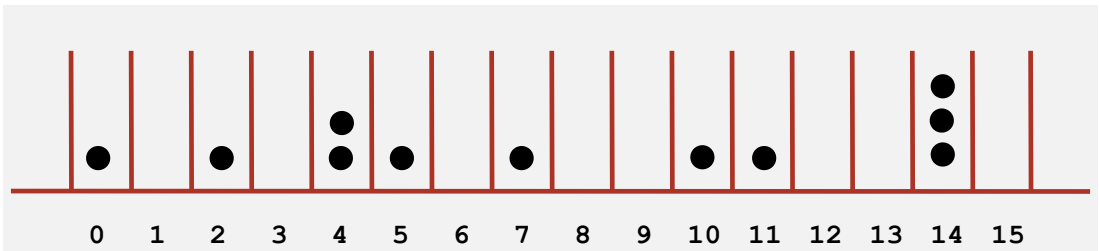
Analyse du cas moyen

- Recherche d'une clé k dans la table :
  - recherche positive : la clé k se trouve dans la table
  - recherche négative : la clé k n'est pas dans la table

- Le facteur de charge d'une table de hachage est donné par $\alpha = \frac{n}{m}$ où :
  - n est le nombre d'éléments dans la table
  - m est la taille de la table (c'est-à-dire, le nombre de listes liées)

- hachage uniforme simple : Pour toute clé $k \in U$,

$$Proba\{h(k) = i\} = \frac{1}{m}, \forall i \in \{0, \ldots, m-1\}$$



Analyse du cas moyen

- Hypothèses :
  - h produit un hachage uniforme simple
  - le calcul de $h(k)$ est $\Theta(1)$
  - Insertion en début de liste
- $\Rightarrow$ complexités moyennes :
  - recherche négative : $\Theta(1+\alpha)$
  - recherche positive : $\Theta(1+\alpha)$
- Si $n = O(m)$, *(m croît au moins linéairement avec n)*,

$$\alpha = \frac{O(m)}{m} = O(1)$$

- Toutes les opérations sont donc $O(1)$ en moyenne

Analyse du cas moyen : recherche négative

- La clé k ne se trouve pas dans la table
- Par la propriété de hachage uniforme simple, elle a la même probabilité d'être envoyée vers chaque position dans la table.
- Recherche négative requière le parcours de la liste $T[h(k)]$ complète
- Cette liste a une longueur moyenne $E[n_{h(k)}] = \alpha$.
- Le nombre d'éléments à examiner lors d'une recherche négative est donc α.
- En ajoutant le temps de calcul de la fonction de hachage, on arrive à une complexité moyenne $\Theta(1 + \alpha)$.

Analyse du cas moyen : recherche positive

- La clé k se trouve dans la table
- Supposons qu'elle ait été insérée à la i-ième étape (parmi n).
  - Le nombre d'éléments à examiner pour trouver la clé est le nombre d'éléments insérés à la position $h(k)$ après k plus 1 (la clé k elle-même).
  - En moyenne, sur les $n - i$ insertions après k, il y en aura $(n - i)/m$ qui correspondront à la position $h(k)$.
- k ayant pu être insérée à n'importe quelle étape parmi n avec une probabilité $1/n$:

$$\sum_{i=1}^{n} \frac{1}{n}(1+\frac{n-i}{m}) = 1+\frac{1}{nm}(\sum_{i=1}^{n} n-\sum_{i=1}^{n} i) = 1+\frac{1}{nm}(n^2-\frac{n(n+1)}{2}) = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

- En tenant compte du coût du hachage, la complexité en moyenne est donc $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$.

Plan

Fonctions de hachage

- Idéalement, la fonction de hachage
  - devrait être facile à calculer ($O(1)$)
  - devrait satisfaire l'hypothèse de hachage uniforme simple
- La deuxième propriété est très difficile à assurer en pratique :
  - La distribution des clés est généralement inconnue
  - Les clés peuvent ne pas être indépendantes
- En pratique, on utilise des heuristiques basées sur la nature attendue des clés
- Si toutes les clés sont connues, il existe des algorithmes pour construire une fonction de hachage parfaite, sans collision (Exemple : le logiciel gperf)

Fonctions de hachage : codage préalable

- Les fonctions de hachage supposent que les clés sont des nombres naturels
- Si ce n'est pas le cas, il faut préalablement utiliser une fonction de codage
- Exemple : codage des chaînes de caractères :
  - On interprète la chaîne comme un entier dans une certaine base
  - Exemple pour "SDA" : valeurs ASCII (128 possibles) :

$$S = 83, D = 68, A = 65$$

  - Interprété comme l'entier : *(Pourquoi pas 83+68+65 ?)*

$$(83 \cdot 128^2) + (68 \cdot 128^1) + (65 \cdot 128^0) = 1368641$$

  - Calculé efficacement par la méthode de Horner :

$$((83 \cdot 128 + 68) \cdot 128 + 65)$$

Méthode de division

La fonction de hachage calcule le reste de la division entière de la clé par la taille de la table

$$h(k) = k \bmod m.$$

Exemple : $m = 20$ et $k = 91 \Rightarrow h(k) = 11$.

Avantages : simple et rapide (juste une opération de division)

Inconvénients : Le choix de m est très sensible et certaines valeurs doivent être évitées

Exemples :

- Si $m = 2^p$ pour un entier p, $h(k)$ ne dépend que des p bits les moins significatifs de k
  - Exemple : "SDA" mod 128= "GAGA" mod 128=65
- Si k est une chaîne de caractères codée en base 2^p et $m = 2^p - 1$, permuter la chaîne ne modifie pas le valeur de hachage
  - Exemple : "SDA"=1368641, "DSA"=1124801
    $\Rightarrow$ 1368641 mod 127 = 1124801 mod 127 = 89

Méthode de division

- Si la fonction de codage produit des séquences périodiques, il vaut mieux choisir m premier
- En effet, si m est premier avec b, on a :

$$\{(a + b \cdot i) \bmod m | i = 0, 1, 2, \ldots\} = \{0, 1, 2, \ldots, m - 1\}$$

- Exemple : hachage de $\{206, 211, 216, 221, \ldots\}$
  - $m = 100$: valeurs hachées possibles : 6, 11,. . . , 96
  - $m = 101$: toutes les entrées sont exploitées

$\Rightarrow$ Bonne valeur de m : un nombre premier pas trop près d'une puissance exacte de 2

Méthode de multiplication

- Fonction de hachage :

$$h(k) = \lfloor m \cdot (kA \bmod 1) \rfloor$$

où

  - A est une constante telle que $0 < A < 1$.
  - $kA \bmod 1 = kA - \lfloor kA \rfloor$ est la partie fractionnaire de kA.

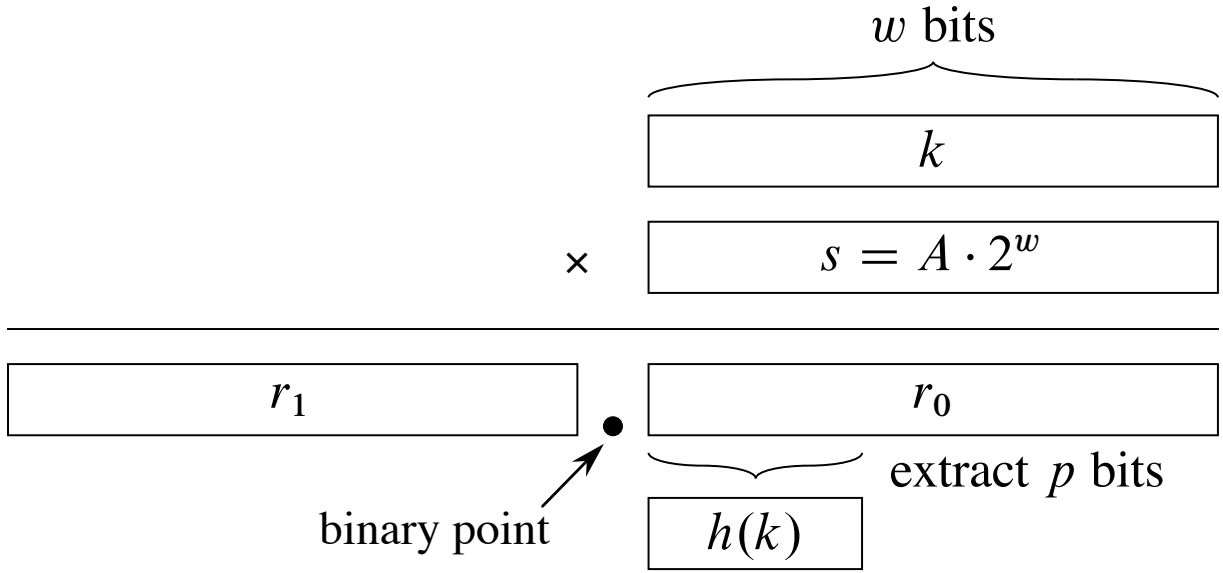
- Inconvénient : plus lente que la méthode de division
- Avantage : la valeur de m n'est plus critique
- La méthode marche mieux pour certaines valeurs de A. Par exemple :

$$A = \frac{\sqrt{5} - 1}{2}$$

Méthode de multiplication : implémentation

Calcul aisé si :

- $m = 2^p$ pour un entier p
- Les mots sont codés en w bits et les clés k peuvent être codées par un seul mot
- A de la forme $s/2^w$ pour $0 < s < 2^w$



Exemple : $m = 2^3$, $w = 5$ $(\Rightarrow 0 < s < 2^5)$, $s = 13$, $A = 13/32 \Rightarrow h(21) = 4$

Plan

Adressage ouvert : principe

- Alternative au chaînage pour gérer les collisions
- Tous les éléments sont stockés dans le tableau (pas de listes chaînées)
- Ne fonctionne que si $\alpha \leq 1$
- Pour insérer une clé k, on sonde les cases systématiquement à partir de $h(k)$ jusqu'à en trouver une vide.
- Différentes méthodes en fonction de la stratégie de sondage

Adressage ouvert : stratégie de sondage

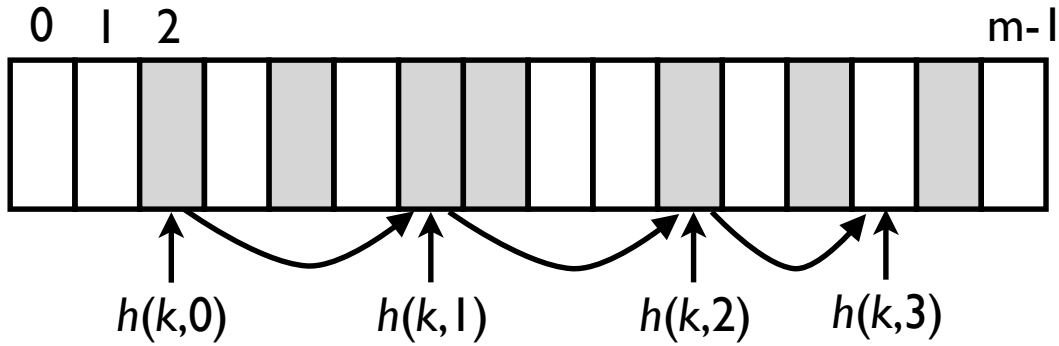
On définit une nouvelle fonction de hachage qui dépend de la clé et du numéro du sondage :

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

et qui est telle que

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

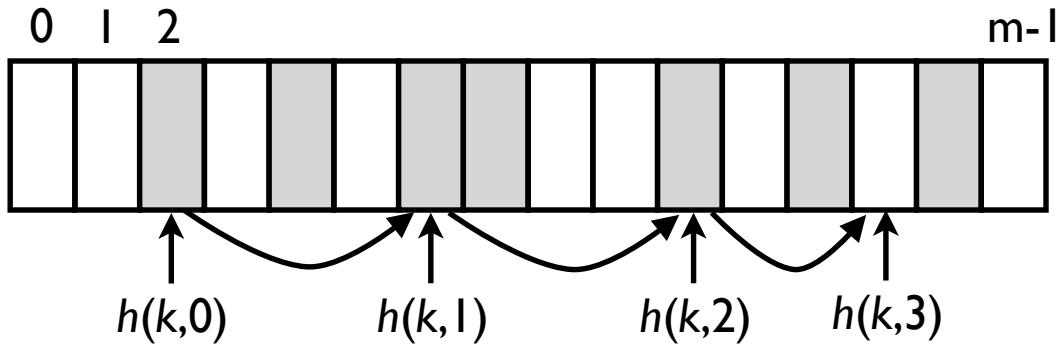
est une permutation de $\langle 0, 1, \ldots, m-1 \rangle$.



Adressage ouvert : recherche et insertion

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == k
5            return j
6       i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == NIL
5            T[j] = k
6            return j
7       else i = i + 1
8  until i == m
9  error "hash table overflow"
```



Adressage ouvert : suppression

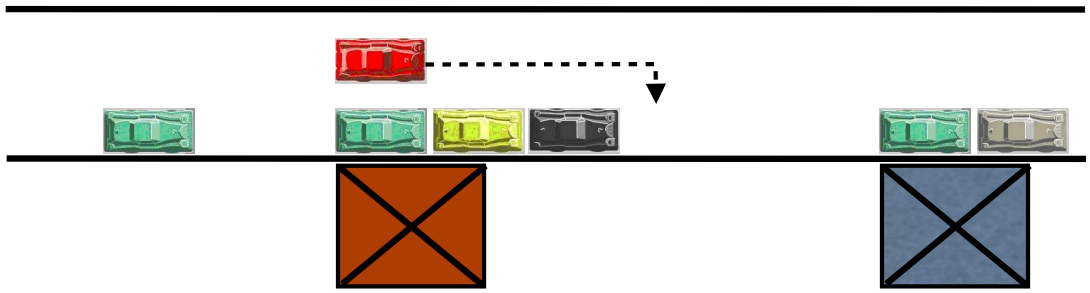
- La suppression est possible mais pas aisée
  - On évitera l'utilisation de l'adressage ouvert si on prévoit de nombreuses suppressions de clés dans le dictionnaire
- On ne peut pas naïvement mettre NIL dans la case contenant la clé k qu'on désire effacer
- Solution :
  - Utiliser une valeur spéciale $\mathrm{DELETED}$ au lieu de NIL pour signifier qu'on a effacé une valeur dans cette case
  - Lors d'une recherche : considérer un case contenant $\mathrm{DELETED}$ comme une case contenant une clé
  - Lors d'une insertion : considérer une case contenant $\mathrm{DELETED}$ comme une case vide.
- Inconvénient : le temps de recherche ne dépend maintenant plus du facteur de charge α de la table

Stratégies de sondage

- Soit $h_k = \langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ la séquence de sondage correspondant à la clé k.
- Hachage uniforme :
  - chacune des $m!$ permutations de $\langle 0, 1, \ldots, m-1 \rangle$ a la même probabilité d'être la séquence de sondage d'une clé k.
  - Difficile à implémenter.
- En pratique, on se contente d'une garantie que la séquence de sondage soit une permutation de $\langle 0, 1, \ldots, m-1 \rangle$.

- Trois techniques pseudo-uniformes :
  - sondage linéaire
  - sondage quadratique
  - double hachage

Sondage linéaire



$$h(k,i) = (h'(k) + i) \bmod m,$$

où $h'(k)$ est une fonction de hachage ordinaire à valeurs dans $\{0, 1, \ldots, m-1\}$.
Propriétés :

- très facile à implémenter
- effet de grappe fort : création de longues suites de cellules occupées
  - La probabilité de remplir une cellule vide est $\frac{i+1}{m}$ où i est le nombre de cellules pleines précédant la cellule vide
- pas très uniforme

Sondage quadratique

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$

où h' est une fonction de hachage ordinaire à valeurs dans $\{0, 1, \ldots, m-1\}$, c_1 et c_2 sont deux constantes non nulles.

Propriétés :

- nécessité de bien choisir les constantes c_1 et c_2 (pour avoir une permutation de $\langle 0, 1, \ldots, m-1 \rangle$)
- effet de grappe plus faible mais tout de même existant :
  - Deux clés de même valeur de hachage suivront le même chemin

$$h(k,0) = h(k',0) \Rightarrow h(k,i) = h(k',i)$$

- meilleur que le sondage linéaire

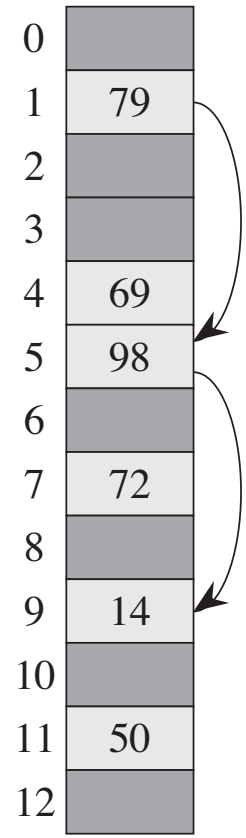
Double hachage

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m,$$

où h_1 et h_2 sont des fonctions de hachage ordinaires à valeurs dans $\{0, 1, \ldots, m-1\}$.

Propriétés :

- difficile à implémenter à cause du choix de h_1 et h_2 ($h_2(k)$ doit être premier avec m pour avoir une permutation de $\langle 0, 1, \ldots, m-1 \rangle$).
- très proche du hachage uniforme
- bien meilleur que les sondages linéaire et quadratique



Exemple : $h_1(k) = k \bmod 13$, $h_2(k) = 1 + (k \bmod 11)$, *insertion de la clé 14*

Adressage ouvert : élément d'analyse

Pour une table de hachage à adressage ouvert de taille m contenant n éléments ($\alpha = n/m < 1$) et en supposant le hachage uniforme

- Le nombre moyen de sondages pour une recherche négative ou un ajout est borné par $\frac{1}{1-\alpha}$
- Le nombre moyen de sondages pour une recherche positive est borné par $\frac{1}{\alpha} \log \frac{1}{1-\alpha}$

$\Rightarrow$ Si α est constant ($n = O(m)$), la recherche est $O(1)$.

- Si $\alpha = 0.5$, une recherche nécessite en moyenne 2 sondages ($1/(1-0.5)$).
- Si $\alpha = 0.9$, une recherche nécessite en moyenne 10 sondages ($1/(1-0.9)$).

Adressage ouvert versus chaînage

- Chaînage :
  - Peut gérer un nombre illimité d'éléments et de collisions
  - Performances plus stables
  - Surcoût lié à la gestion et le stockage en mémoire des listes liées
- Adressage ouvert :
  - Rapide et peu gourmand en mémoire
  - Choix de la fonction de hachage plus difficile (pour éviter les grappes)
  - On ne peut pas avoir $n > m$
  - Suppression problématique
- D'autres alternatives existent :
  - Two-probe hashing
  - Cuckoo hashing
  - ...

Le rehachage

- Lorsque α se rapproche de 1, les performances s'effondrent
- Solution : rehachage : création d'une table plus grande
  - allocation d'une nouvelle table
  - détermination d'une nouvelle fonction de hachage, tenant compte du nouveau m
  - parcours des entrées de la table originale et insertion dans la nouvelle table
- Si la taille est doublée, le coût asymptotique constant des opérations est conservé (voir transp. 209).

Universal hashing

- Les performances d'un table de hachage se dégrade fortement en cas de collisions multiples
- Connaissant la fonction de hachage, un adversaire malintentionné pourrait s'amuser à entrer des clés créant des collisions. Exemples :
  - Création de fichiers avec des noms bien choisis dans le kernel Linux 2.4.20
  - 28/12/2011 : `http://www.securityweek.com/hash-table-collision-attacks-could-trigger-ddos-massive-scale`
- C'est un exemple d'attaque par déni de service

- Parade : hachage universel : choisir la fonction de hachage aléatoirement à chaque création d'une nouvelle instance de la table
- Exemple :

$$h(k) = ((ak + b) \bmod p) \bmod m,$$

où p est un premier très grand et a et b deux entiers choisis aléatoirement

Demo

- `http://groups.engin.umd.umich.edu/CIS/course.des/cis350/hashing/WEB/HashApplet.htm`

Plan

Dictionnaires : résumé

	Pire cas			*En moyenne*		
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
AVL	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Table de hachage	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

- Cas moyen valable uniquement sous l'hypothèse de hachage uniforme
- Comment obtenir $\Theta(\log n)$ dans le pire cas avec une table de hachage ?

ABR/AVL versus table de hachage

Tables de hachage :

- Faciles à implémenter
- Seule solution pour des clés non ordonnées
- Accès et insertion très rapides en moyenne (pour des clés simples)
- Espace gaspillé lorsque α est petit
- Pas de garantie au pire cas (performances "instables")

Arbres binaire de recherche (équilibrés) :

- Performance garantie dans tous les cas (stabilité)
- Taille de structure s'adapte à la taille des données
- Supportent des opérations supplémentaires lorsque les clés sont ordonnées (parcours en ordre, successeur, prédécesseur, etc.)
- Accès et insertion plus lente en moyenne

Partie 6

Résolution de problèmes

Plan

Méthodes de résolution de problèmes

Quelques approches génériques pour aborder la résolution d'un problème :

- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Diviser pour régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original
- Programmation dynamique : obtenir la solution optimale à un problème en combinant des solutions optimales à des sous-problèmes similaires plus petits et se chevauchant
- Approche gloutonne : construire la solution incrémentalement, en optimisant de manière aveugle un critère local

Approche par force brute (brute-force)

- Consiste à appliquer la solution la plus directe à un problème
- Généralement obtenue en appliquant à la lettre la définition du problème
- Exemple simple :
  - Rechercher un élément dans un tableau (trié ou non) en le parcourant linéairement
  - Calculer a^n en multipliant a n fois avec lui-même
  - Implémentation récursive naïve du calcul des nombres de Fibonacci
  - ...
- Souvent pas très efficace en terme de temps de calcul mais facile à implémenter et fonctionnel

Exemple : tri

Approches par force brute pour le tri :

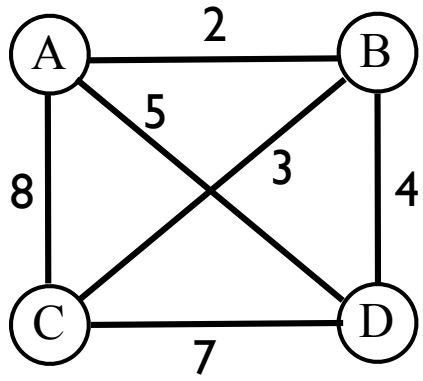
- Un tableau est trié (en ordre croissant) si tout élément est plus petit que l'élément à sa droite
- $\Rightarrow$ tri à bulle : parcourir le tableau de gauche à droite en échangeant toutes les paires d'éléments consécutifs ne respectant pas cette définition
- Complexité : $O(n^2)$
- $\Rightarrow$ tri par sélection : trouver le minimum du tableau, l'échanger avec le premier élément, répéter pour trier le reste du tableau
- Complexité : $\Theta(n^2)$

Recherche exhaustive

- Une solution par force brute au problème de la recherche d'un élément possédant une propriété particulière
- Générer toutes les solutions possibles jusqu'à en obtenir une qui possède la propriété recherchée
- Exemple pour le tri :
  - Générer toutes les permutations du tableau de départ (une et une seule fois)
  - Vérifier si chaque tableau permuté est trié. S'arrêter si c'est le cas.
  - Complexité : $O(n! \cdot n)$
- Généralement utilisable seulement pour des problèmes de petite taille
- Dans la plupart des cas, il existe une meilleure solution
- Dans certains cas, c'est la seule solution possible

Problème du voyageur de commerce

- Etant donné n villes et les distances entre ces villes
- Trouver le plus court chemin qui passe par toutes les villes exactement une fois avant de revenir à la ville de départ



Tour	Coût
A-B-C-D-A	17
A-B-D-C-A	21
A-C-B-D-A	20
A-C-D-B-A	21
A-D-B-C-A	20
A-D-C-B-A	17

- Recherche exhaustive : $O(n!)$
- On n'a pas encore pu trouver un algorithme de complexité polynomiale (et il y a peu de chance qu'on y arrive)

Force brute/recherche exhaustive

Avantages :

- Simple et d'application très large
- Un bon point de départ pour trouver de meilleurs algorithmes
- Parfois, faire mieux n'en vaut pas la peine

Inconvénients :

- Produit rarement des solutions efficaces
- Moins élégant et créatif que les autres techniques

Dans ce qui suit, on commencera la plupart du temps par fournir la solution par force brute des problèmes, qu'on cherchera ensuite à résoudre par d'autres techniques

Méthodes de résolution de problèmes

Quelques approches génériques pour aborder la résolution d'un problème :

- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Diviser pour régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original
- Programmation dynamique : obtenir la solution optimale à un problème en combinant des solutions optimales à des sous-problèmes similaires plus petits et se chevauchant
- Approche gloutonne : construire la solution incrémentalement, en optimisant de manière aveugle un critère local

Plan

Approche diviser-pour-régner *(Divide and conquer)*

Principe général :

- Si le problème est trivial, on le résoud directement
- Sinon :
  1. Diviser le problème en sous-problèmes de taille inférieure (Diviser)
  2. Résoudre récursivement ces sous-problèmes (Régner)
  3. Fusionner les solutions aux sous-problèmes pour produire une solution au problème original

Exemples déjà rencontrés

- Merge sort :
  1. Diviser : Couper le tableau en deux sous-tableaux de même taille
  2. Régner : Trier récursivement les deux sous-tableaux
  3. Fusionner : fusionner les deux sous-tableaux

  Complexité : $\Theta(n \log n)$ (force brute : $\Theta(n^2)$)
- Quicksort :
  1. Diviser : Partionner le tableau selon le pivot
  2. Régner : Trier récursivement les deux sous-tableaux
  3. Fusionner : /

  Complexité en moyenne : $\Theta(n \log n)$ (force brute : $\Theta(n^2)$)
- Recherche binaire (dichotomique) :
  1. Diviser : Contrôler l'élement central du tableau
  2. Régner : Chercher récursivement dans un des sous-tableaux
  3. Fusionner : trivial

  Complexité : $O(\log n)$ (force brute : $O(n)$)

Exemple 1 : Calcul du minimum/maximum d'un tableau

- Approche par force brute pour trouver le minimum ou le maximum d'un tableau

```
MIN(A)
1  min = A[1]
2  for i = 2 to A.length
3       if min > A[i]
4            min = A[i]
5  return min
```

```
MAX(A)
1  max = A[1]
2  for i = 2 to A.length
3       if max < A[i]
4            max = A[i]
5  return max
```

- Complexité : $\Theta(n)$ ($n-1$ comparaisons)
- Peut-on faire mieux ?
  - Non, pas en notation asymptotique (le problème est $\Theta(n)$)
  - Par contre, on peut diminuer le nombre total de comparaisons pour calculer à la fois le minimum et le maximum

Calcul simultané du minimum et du maximum

- Approche diviser-pour-régner pour le calcul simultané du minimum et du maximum

```
MAX-MIN(A, p, r)
1  if r − p ≤ 1
2        if A[p] < A[r]
3              return (A[r], A[p])
4        else return (A[p], A[r])
5  q = ⌊(p+r)/2⌋
6  (max1, min1) = MAX-MIN(A, p, q)
7  (max2, min2) = MAX-MIN(A, q + 1, r)
8  return (MAX(max1, max2), MIN(min1, min2))
```

Appel initial : $\textsc{Max-Min}(A, 1, A.length)$

- Correct ? Oui (preuve par induction)
- Complexité ?

Analyse de complexité

- En supposant que n est une puissance de 2, le nombre de comparaisons $T(n)$ est donné par :

$$T(n) = \begin{cases} 1 & \text{si } n = 2 \\ 2T(n/2) + 2 & \text{sinon} \end{cases}$$

qui se résoud en :

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &= 8T(n/4) + 8 + 4 + 2 \\ &= 2^i T(n/2^i) + \sum_{j=1}^{i} 2^j \\ &= 2^{\log_2(n)-1} T(2) + \sum_{j=1}^{\log_2(n)-1} 2^j \\ &= 3/2n - 2 \end{aligned}$$

- C'est-à-dire 25% de comparaisons en moins que les méthodes séparées

Exemple 2 : Recherche de pics

- Soit un tableau $A[1\,..\,A.length]$. On supposera que $A[0] = A[A.length + 1] = -\infty$.
- Définition : $A[i]$ est un pic s'il n'est pas plus petit que ses voisins :

$$A[i-1] \leq A[i] \geq A[i+1]$$

  ($A[i]$ est un maximum local)
- But : trouver un pic dans le tableau (n'importe lequel)
- Note : il en existe toujours un

Approche par force brute

- Tester toutes les positions séquentiellement :

```
PEAK1D(A)
1  for i = 1 to A.length
2       if A[i − 1] ≤ A[i] ≥ A[i + 1]
3            return i
```

- Complexité : $\Theta(n)$ dans le pire acas

Approche par force brute 2

- Le maximum global du tableau est un maximum local et donc un pic

```
PEAK1D(A)
1  m = A[0]
2  for i = 1 to A.length
3       if A[i] > A[m]
4            m = i
5  return m
```

- Complexité : $\Theta(n)$ dans tous les cas

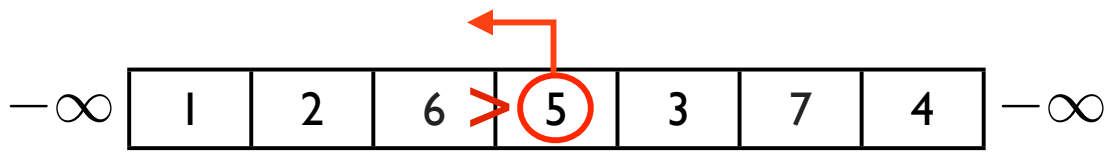
Une meilleure idée

Approche diviser-pour-régner :

- Sonder un élément $A[i]$ et ses voisins $A[i-1]$ et $A[i+1]$
- Si c'est un pic : renvoyer i
- Sinon :
  - les valeurs doivent croître au moins d'un côté

$$A[i-1] > A[i] \text{ ou } A[i] < A[i+1]$$

  - Si $A[i-1] > A[i]$, on cherche le pic dans $A[1\,..\,i-1]$
  - Si $A[i+1] > A[i]$, on cherche le pic dans $A[i+1\,..\,A.\mathit{length}]$



- A quel position i faut-il sonder ?

Algorithme

```
PEAK1D(A, p, r)
1  q = ⌊(p+r)/2⌋
2  if A[q − 1] ≤ A[q] ≥ A[q + 1]
3        return q
4  elseif A[q − 1] > A[q]
5        return PEAK1D(A, p, q − 1)
6  elseif A[q] < A[q + 1]
7        return PEAK1D(A, q + 1, r)
```

Appel initial : $\textsc{Peak1d}(A, 1, A.length)$

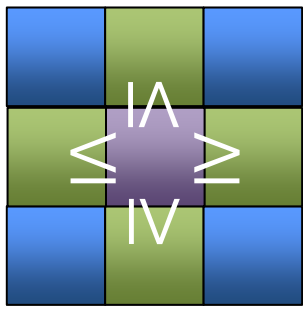
Analyse

- Correction : oui
  - On doit prouver qu'il y aura un pic du côté choisi
  - Preuve par l'absurde :
    - Supposons que $A[q+1] > A[q]$ et qu'il n'y ait pas de pic dans $A[q+1\,..\,r]$
    - On doit avoir $A[q+2] > A[q+1]$ (sinon $A[q+1]$ serait un pic)
    - On doit avoir $A[q+3] > A[q+2]$ (sinon $A[q+2]$ serait un pic)
    - ...
    - On doit avoir $A[r] > A[r-1]$ (sinon $A[r-1]$ serait un pic)
    - Comme $A[r] > A[r+1] = -\infty$, $A[r]$ est un pic, ce qui contredit l'hypothèse

- Complexité :
  - Dans le pire cas, on a $T(n) = T(n/2) + c_1$ et $T(1) = c_2$ (idem recherche binaire)
  - $\Rightarrow T(n) = O(\log n)$

Extension à un tableau 2D

- Soit une matrice $n \times n$ de nombres
- Trouver un élement plus grand ou égal à ses 4 voisins (max)



9	3	5	2	4	9	8
7	2	5	1	4	0	3
9	8	9	3	2	4	8
7	6	3	1	3	2	3
9	0	6	0	4	6	4
8	9	8	0	5	3	0
2	1	2	1	1	1	1

(Demaine & Leiserson)

- Approche par force brute : $O(n^2)$
- Recherche du maximum : $\Theta(n^2)$

Approche diviser-pour-régner

- Chercher le maximum global dans la colonne centrale
- Si c'est un pic, le renvoyer
- Sinon appeler la fonction récursivement sur les colonnes à gauche (resp. droite) si le voisin à gauche (resp. droite) est plus grand

9	3	5	2	4	9	8
7	2	5	1	4	0	3
9	8	9	3	2	4	8
7	6	3	1	3	2	3
9	0	6	0	4	6	4
8	9	8	0	5	3	0
2	1	2	1	1	1	1

9	9	9	3	5	9	8

(Demaine & Leiserson)

Analyse : correction

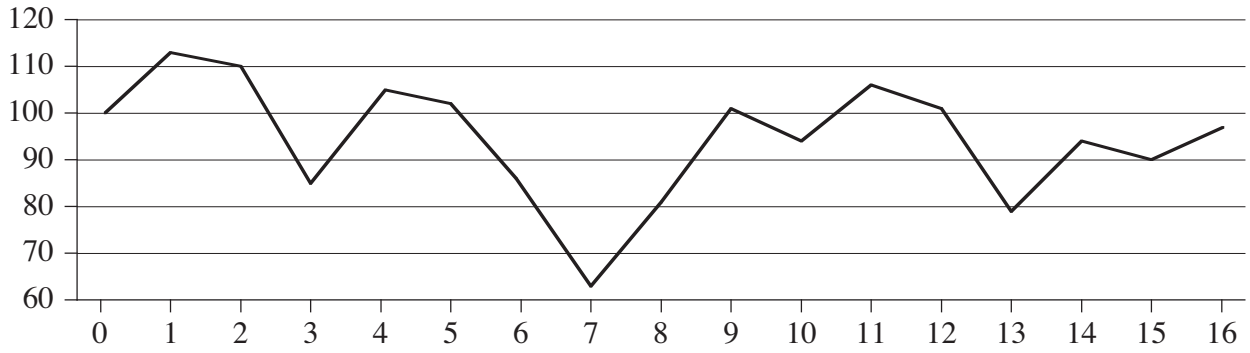
- On doit prouver qu'il y a bien un pic du côté choisi
- Preuve par l'absurde :
  - Supposons qu'il n'y ait pas de pic
  - Soient $A[i, j]$ le maximum de la colonne centrale et $A[i, k]$ le voisin le plus grand ($k = j - 1$ ou $k = j + 1$)
  - $A[i, k]$ doit avoir un voisin $A[p_1, q_1]$ avec une valeur plus élevée (sinon, ce serait un pic)
  - $A[p_1, q_1]$ doit avoir un voisin $A[p_2, q_2]$ avec une valeur plus élevée (sinon, ce serait un pic)
  - ...
  - Le voisin doit toujours rester du même côté de la colonne centrale (puisque $A[i, k] > A[i, j]$ et $A[i, j]$ est le maximum de la colonne j)
  - A un certain point, on va manquer de points
  - Il doit donc y avoir un pic

Analyse : complexité

- $\Theta(n)$ pour trouver le maximum d'une colonne
- $O(\log n)$ itérations
- $O(n \log n)$ au total

- Peut-on faire mieux ? Oui, il est possible de proposer un algorithme en $O(n)$ (pas vu dans ce cours)

Exemple 3 : Achat/vente d'actions



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

- Soit le prix d'une action au cours de n jours consécutifs (prix à la fermeture)
- On aimerait déterminer rétrospectivement :
  - à quel moment, on aurait dû acheter et
  - à quel moment, on aurait dû vendre

  de manière à maximiser notre gain

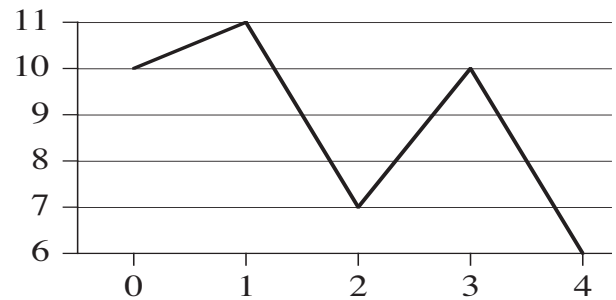
Exemple 3 : Achat/vente d'actions

Première stratégie :

- Acheter au prix minimum, vendre au prix maximum
- Pas correct : Le prix maximum ne suit pas nécessairement le prix minimum

Deuxième stratégie :

- Soit acheter au prix minimum et vendre au prix le plus élevé qui suit
- Soit vendre au prix maximum et acheter au prix le plus bas qui précède
- Pas correct :



Troisième stratégie :

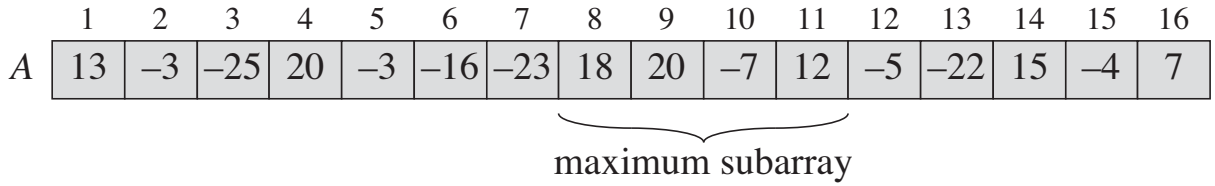
- Tester toutes les paires (force brute)
- Correct ? Complexité ?

Achat/vente d'actions : transformation

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- Transformation du problème :
  - Calculer le tableau $A[i] =$ (prix du jour i)-(prix du jour i-1) (de taille $A.length = n$ en supposant qu'on démarre avec un prix au jour 0)
  - Déterminer la sous-séquence non vide contiguë de somme maximale dans A
  - Soit $A[i\,..\,j]$ cette sous-séquence. Il aurait fallu acheter juste avant le jour i (juste après le jour $i-1$) et vendre juste après le jour j.
- Exemple dans le tableau ci-dessus : $A[8\,..\,11]$ est la sous-séquence maximale de somme 43 $\Rightarrow$ acheter juste avant le jour 8 et vendre juste après le jour 11.
- Si on peut trouver la sous-séquence maximale dans un tableau, on aura une solution à notre problème d'achat/vente d'actions

Approche par force brute



- Implémentation naïve :
  - On génère tous les sous-tableaux
  - On calcule la somme des éléments de chaque sous-tableau
  - On renvoie les bornes du (d'un) sous-tableau de somme maximale
- Complexité : $\Theta(n^2)$ sous-tableaux et $O(n)$ pour le calcul de la somme d'un sous-tableau $\Rightarrow O(n^3)$
- On peut l'implémenter en $\Theta(n^2)$

Approche par force brute

```
MAX-SUBARRAY-BRUTE-FORCE(A)
 1  n = A.length
 2  max-so-far = −∞
 3  for l = 1 to n
 4        sum = 0
 5        for h = l to n
 6              sum = sum + A[h]
 7              if sum > max-so-far
 8                    max-so-far = sum
 9                    low = l
10                    high = h
11  return (low, high)
```

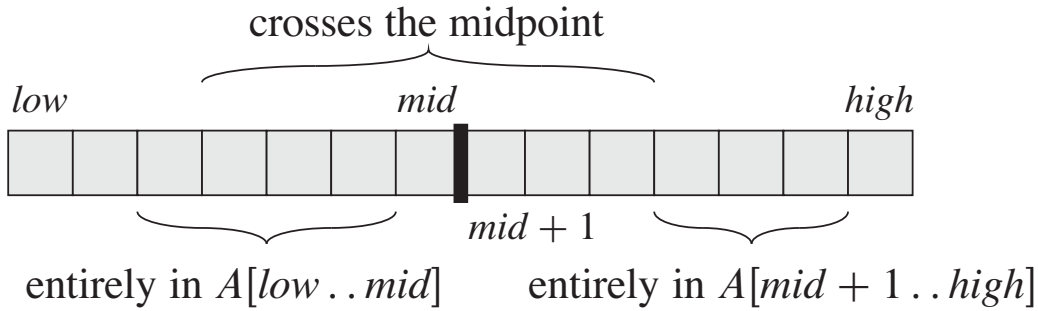
Complexité : $\Theta(n^2)$

Peut-on faire mieux ?

Approche diviser-pour-régner

- Nouveau problème :
  - trouver un sous-tableau maximal dans $A[low\,..\,high]$
  - fonction $\textsc{maximum-subarray}(A, low, high)$
- Diviser :
  - diviser le sous-tableau en deux sous-tableaux de tailles aussi proches que possible
  - choisir $mid = \lfloor (low + high)/2 \rfloor$
- Régner :
  - trouver récursivement les sous-tableaux maximaux dans ces deux sous-tableaux
  - appeler $\textsc{maximum-subarray}(A, low, mid)$ et $\textsc{maximum-subarray}(A, mid + 1, high)$
- Fusionner : ?

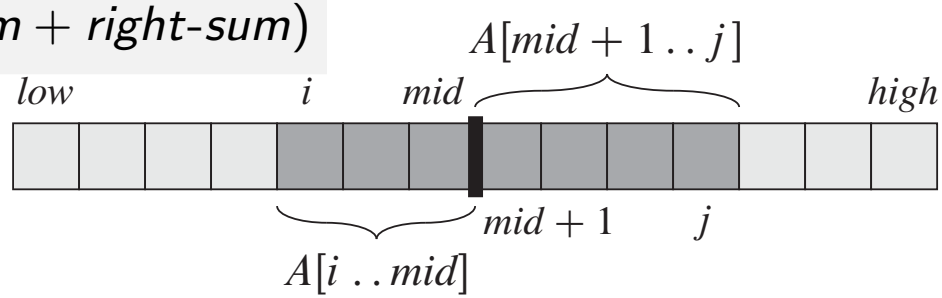
Approche diviser-pour-régner



- Fusionner :
  - Rechercher un sous-tableau maximum qui traverse la jonction
  - Choisir la meilleure solution parmi les 3
- MAX-CROSSING-SUBARRAY(A, *low*, *mid*, *high*)
  - Force brute : $\Theta(n^2)$ (car n/2 choix pour l'extrémité gauche, n/2 choix pour l'extrémité droite)
  - Meilleure solution : on recherche indépendamment les extrémités gauche et droite

MAX-CROSSING-SUBARRAY

```
MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4        sum = sum + A[i]
 5        if sum > left-sum
 6              left-sum = sum
 7              max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11        sum = sum + A[j]
12        if sum > right-sum
13              right-sum = sum
14              max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```



Complexité : $\Theta(n)$

Max-subarray

```
MAX-SUBARRAY(A, low, high)
 1  if high == low
 2        return (low, high, A[low])
 3  else mid = ⌊(low + high)/2⌋
 4        (left-low, left-high, left-sum) = MAX-SUBARRAY(A, low, mid)
 5        (right-low, right-high, right-sum) = MAX-SUBARRAY(A, mid + 1, high)
 6        (cross-low, cross-high, cross-sum) =
 7              MAX-CROSSING-SUBARRAY(A, low, mid, high)
 8        if left-sum ≥ right-sum and left-sum ≥ cross-sum
 9              return (left-low, left-high, left-sum)
10        elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
11              return (right-low, right-high, right-sum)
12        else return (cross-low, cross-high, cross-sum)
```

Analyse

- Si on suppose que n est un multiple de 2, le nombre d'opérations $T(n)$ est donné par :

$$T(n) = \begin{cases} c_1 & \text{si } n = 1 \\ 2T(n/2) + c_2 n & \text{sinon} \end{cases}$$

- Même complexité que le tri par fusion $\Rightarrow \Theta(n \log n)$
- Peut-on faire mieux ? On verra plus loin que oui

Diviser pour régner : résumé

- Mène à des algorithmes très efficaces
- Pas toujours applicable mais quand même très utile

- Applications :
  - Tris optimaux
  - Recherche binaire
  - Problème de sélection
  - Trouver la paire de points les plus proches
  - Recherche de l'enveloppe convexe (convex-hull)
  - Multiplication de matrice (méthode de Strassens)
  - ...

Méthodes de résolution de problèmes

Quelques approches génériques pour aborder la résolution d'un problème :

- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Diviser pour régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original
- Programmation dynamique : obtenir la solution optimale à un problème en combinant des solutions optimales à des sous-problèmes similaires plus petits et se chevauchant
- Approche gloutonne : construire la solution incrémentalement, en optimisant de manière aveugle un critère local

Plan

Exemple 1 : découpage de tiges d'acier

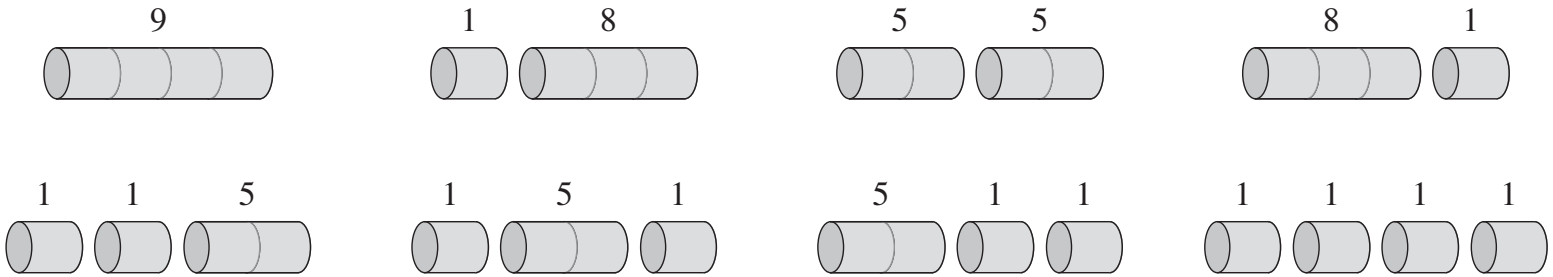
- Soit une tige d'acier qu'on découpe pour la vendre morceau par morceau
- La découpe ne peut se faire que par nombre entier de centimètres
- Le prix de vente d'une tige dépend (non linéairement) de sa longueur
- On veut déterminer le revenu maximum qu'on peut attendre de la vente d'une tige de n centimètres
- Problème algorithmique :
  - Entrée : une longueur $n > 0$ et une table de prix p_i, pour $i = 1, 2, \ldots, n$
  - Sortie : le revenu maximum qu'on peut obtenir pour des tiges de longueur n

Illustration

- Soit la table de prix :

Longueur i	1	2	3	4	5	6	7	8	9	10
Prix p_i	1	5	8	9	10	17	17	20	24	30

- Découpes possibles d'une tige de longueur $n = 4$



- Meilleur revenu : découpage en 2 tiges de 2 centimètres, revenu de 10

Approche par force brute

- Enumérer toutes les découpes, calculer leur revenu, déterminer le revenu maximum
- Complexité : exponentielle en n :
  - Il y a 2^{n-1} manières de découper une tige de longueur n (on peut couper ou non après chacun des $n-1$ premiers centimètres)
  - Plusieurs découpes sont équivalentes (1+1+2 et 1+2+1 par exemple) mais même en prenant cela en compte, le nombre de découpes reste exponentiel
- Infaisable pour n un peu grand

Idée

- Soit r_i le revenu maximum pour une tige de longueur i
- Peut-on formuler r_n de manière récursive ?
- Déterminons r_i pour notre exemple :

i	r_i	solution optimale
1	1	1 (pas de découpe)
2	5	2 (pas de découpe)
3	8	3 (pas de découpe)
4	10	2+2
5	13	2+3
6	17	6 (pas de découpe)
7	18	1+6 ou 2+2+3
8	22	2+6 ...

Formulation récursive de r_n : version naïve

- r_n peut être calculé comme le maximum de :
  - p_n : le prix sans découpe
  - $r_1 + r_{n-1}$: le revenu max pour une tige de 1 et une tige de $n-1$
  - $r_2 + r_{n-2}$: le revenu max pour une tige de 2 et une tige de $n-2$
  - ...
  - $r_{n-1} + r_1$.
- C'est-à-dire

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

Formulation récursive de r_n : version simplifiée

- Toute solution optimale a un découpe la plus à gauche
- On peut calculer r_n en considérant toutes les tailles pour la première découpe et en combinant avec le découpage optimal pour la partie à droite
- Pour chaque cas, on n'a donc qu'à résoudre un seul sous-problème (au lieu de deux), celui du découpage de la partie droite
- En supposant $r_0 = 0$, on obtient ainsi :

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

Implémentation récursive directe

- La formule récursive peut être implémentée directement

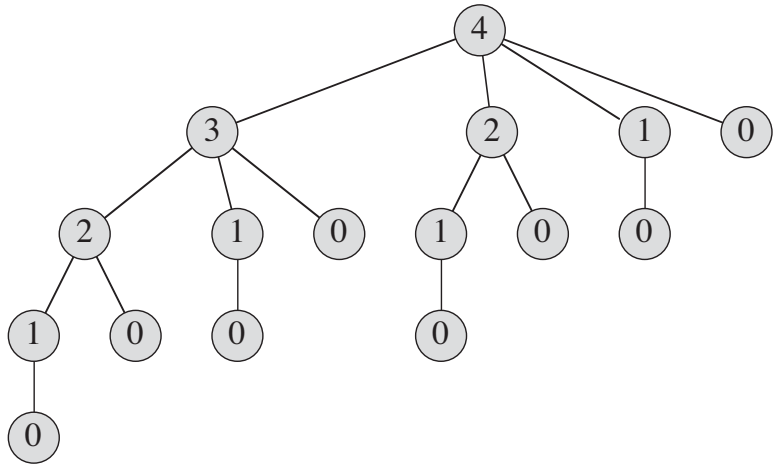
```
CUT-ROD(p, n)
1  if n == 0
2        return 0
3  q = -∞
4  for i = 1 to n
5        q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

(p est un tableau de taille n contenant les prix des tiges de tailles 1 à n)

- Complexité ?

Implémentation récursive directe : analyse

- L'algorithme est extrêmement inefficace à cause des appels récursifs redondants
- Exemple : arbre des appels récursifs pour le calcul de r_4



- En général, le nombre de nœuds $T(n)$ de l'arbre est 2^n.
  Preuve par induction :
  - Cas de base : $T(0) = 1$
  - Cas inductif :

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + \sum_{j=0}^{n-1} 2^j = 1 + 2^n - 1 = 2^n$$

- Complexité de l'algorithme est exponentielle en n

Solution par programmation dynamique

- Solution : plutôt que de résoudre les mêmes sous-problèmes plusieurs fois, s'arranger pour ne les résoudre chacun qu'une seule fois
- Comment ? En sauvegardant les solutions dans une table et en se référant à la table à chaque demande de résolution d'un sous-problème déjà rencontré
- On échange du temps de calcul contre de la mémoire
- Permet de transformer une solution en temps exponentiel en une solution en temps polynomial
- Deux implémentations possibles :
  - descendante (top-down) avec mémoization
  - ascendante (bottom-up)

Approche descendante avec mémoization

```
MEMOIZED-CUT-ROD(p, n)
1  Let r[0..n] be a new array
2  for i = 1 to n
3       r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

(Attention : suppose que le tableau est passé par pointeur)

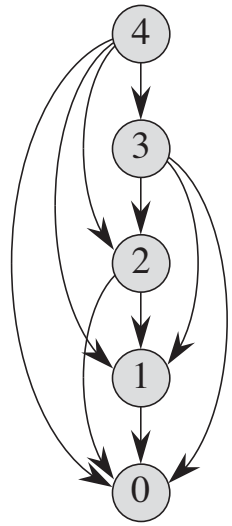
Approche ascendante

Principe : résoudre les sous-problèmes par taille en commençant d'abord par les plus petits

```
BOTTOM-UP-CUT-ROD(p, n)
1  Let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4        q = -∞
5        for i = 1 to j
6              q = max(q, p[i] + r[j - i])
7        r[j] = q
8  return r[n]
```

Programmation dynamique : analyse

- Solution ascendante est clairement $\Theta(n^2)$ (deux boucles imbriquées)
- Solution descendante est également $\Theta(n^2)$
  - Chaque sous-problème est résolu une et une seule fois
  - La résolution d'un sous-problème passe par une boucle à n itérations
- Graphes des sous-problèmes :



(une flèche de x à y indique que la résolution de x dépend de la résolution de y)

Reconstruction de la solution

- Fonction BOTTOM-UP-CUT-ROD calcule le revenu maximum mais ne donne pas directement la découpe correspondant à ce revenu
- On peut étendre l'approche ascendante pour enregistrer également la solution dans une autre table

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  Let r[0..n] and s[1..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4        q = -∞
 5        for i = 1 to j
 6              if q < p[i] + r[j - i]
 7                    q = p[i] + r[j - i]
 8                    s[j] = i
 9        r[j] = q
10  return r and s
```

- $s[j]$ contient la coupure la plus à gauche d'une solution optimale au problème de taille j

Reconstruction de la solution

- Pour afficher la solution, on doit "remonter" dans s

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      PRINT s[n]
4      n = n − s[n]
```

- Exemple :

i	0	1	2	3	4	5	6	7	8
$p[i]$	0	1	5	8	9	10	17	17	20
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$	0	1	2	3	2	2	6	1	2

$$\text{PRINT-CUT-ROD-SOLUTION}(p, 8) \Rightarrow \text{"2 6"}$$

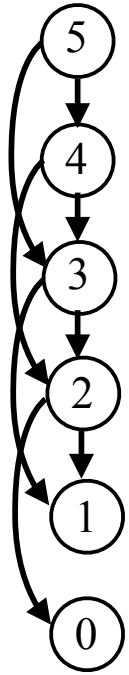
Programmation dynamique : généralités

- La programmation dynamique s'applique aux problèmes d'optimisation qui peuvent se décomposer en sous-problèmes de même nature, et qui possèdent les deux propriétés suivantes :
  - Sous-structure optimale : on peut calculer la solution d'un problème de taille n à partir de la solution de sous-problèmes de taille inférieure
  - Chevauchement des sous-problèmes : Certains sous-problèmes distincts partagent une partie de leurs sous-problèmes
- Implémentation directe récursive donne une solution de complexité exponentielle
- Sauvegarde des solutions aux sous-problèmes donne une complexité linéaire dans le nombre d'arcs et de sommets du graphe des sous-problèmes

Exemple 2 : Fibonacci

- La fonction FIBONACCI-ITER vue au début du cours est un exemple de programmation dynamique (ascendante)

```
FIBONACCI-ITER(n)
 1  if n ≤ 1
 2        return n
 3  else
 4        pprev = 0
 5        prev = 1
 6        for i = 2 to n
 7              f = prev + pprev
 8              pprev = prev
 9              prev = f
10        return f
```



- On peut se contenter de ne stocker que les deux dernières valeurs
- Complexité $\Theta(n)$ (graphe contient $n+1$ nœuds et $2n-2$ arcs)

(Exercice : écrivez la version descendante avec mémoization)

Interlude : Fibonacci en $\Theta(\log n)$

- Peut-on faire mieux que $\Theta(n)$ pour Fibonacci ? Oui !
- Propriété :

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

- Preuve par induction :
  - Cas de base ($n = 1$) : ok puisque $F_0 = 0$, $F_1 = 1$, et $F_2 = 1$
  - Cas inductif ($n \geq 2$) :

$$\begin{aligned} \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} &= \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \end{aligned}$$

□

Interlude : Fibonacci en $\Theta(\log n)$

- Approche par force brute pour le calcul de $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$: $\Theta(n)$
- Idée : utiliser le diviser-pour-régner pour le calcul de a^n

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{si } n \text{ est pair} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{si } n \text{ est impair} \end{cases}$$

- Complexité : $\Theta(\log n)$ (comme la recherche binaire)

(Exercice : implémenter l'algorithme)

Exemple 3 : sous-séquence maximale

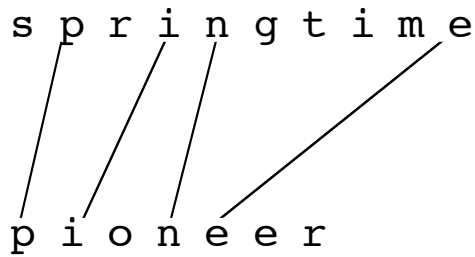
```
MAX-SUBARRAY-LINEAR(A)
 1  Let m[1..n] be a new array
 2  max-so-far = A[1]
 3  m[1] = A[1]
 4  for i = 2 to A.length
 5        if m[i − 1] > 0
 6              m[i] = m[i − 1] + A[i]
 7        else m[i] = A[i]
 8        if m[i] > max-so-far
 9              max-so-far = m[i]
10  return max-so-far
```

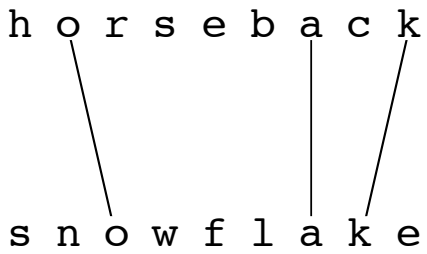
- Complexité : $\Theta(n)$ (diviser pour régner : $\Theta(n \log n)$)
- $m[i]$ est la somme de la sous-séquence maximale qui se termine en i
- L'algorithme calcule $m[i]$ à partir de $m[i-1]$
- Forme de programmation dynamique ascendante (très simple)

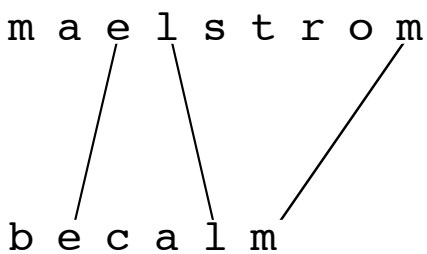
(Exercice : ajouter le calcul des bornes d'un sous-tableau solution, remplacer le tableau m par une seule variable)

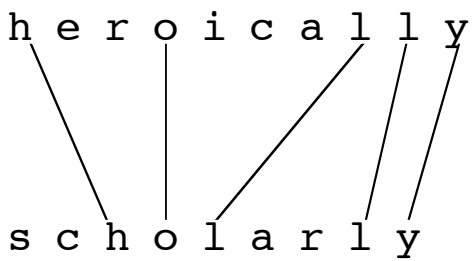
Exemple 4 : plus longue sous-séquence commune

- Définition : Une sous-séquence (non contiguë) d'une séquence $\langle x_1, \ldots, x_m \rangle$ est une séquence $\langle x_{i_1}, x_{i_2}, \ldots, x_{i_k} \rangle$, où $1 \leq i_1 < i_2 < \ldots < i_k \leq m$.
- Problème : Etant donné 2 séquences, $X = \langle x_1, \ldots, x_m \rangle$ et $Y = \langle y_1, \ldots, y_n \rangle$, trouver une plus grande sous-séquence commune aux deux séquences
- Exemples :









Solution par force brute

- On énumère toutes les sous-séquences de la séquence la plus courte
- Pour chacune d'elles, on vérifie si c'est une sous-séquence de la séquence la plus longue

- Complexité : $\Theta(n \cdot 2^m)$ (en supposant que $n < m$)
  - 2^m sous-séquences possibles dans une séquence de longueur m
  - Vérification de l'occurence d'une sous-séquence dans une séquence de longueur n en $\Theta(n)$

(Exercice : implémenter la vérification)

Solution par programmation dynamique

Propriété de sous-structure :

- Soit $X_i = \langle x_1, \ldots, x_i \rangle$ un préfixe de X et $Y_i = \langle y_1, \ldots, y_i \rangle$ un préfixe de Y
- Soit $Z = \langle z_1, \ldots, z_k \rangle$ une plus longue sous-séquence commune de X et Y
- Les propriétés suivantes sont vérifiées :
  - Si $x_m = y_n$, alors $z_k = x_m = y_n$ et Z_{k-1} est une plus longue sous-séquence commune de X_{m-1} et Y_{n-1}.
  - Si $x_m \neq y_n$, alors $z_k \neq x_m \Rightarrow Z$ est une plus longue sous-séquence commune à X_{m-1} et Y
  - Si $x_m \neq y_n$, alors $z_k \neq y_n \Rightarrow Z$ est une plus longue sous-séquence commune à X et Y_{n-1}

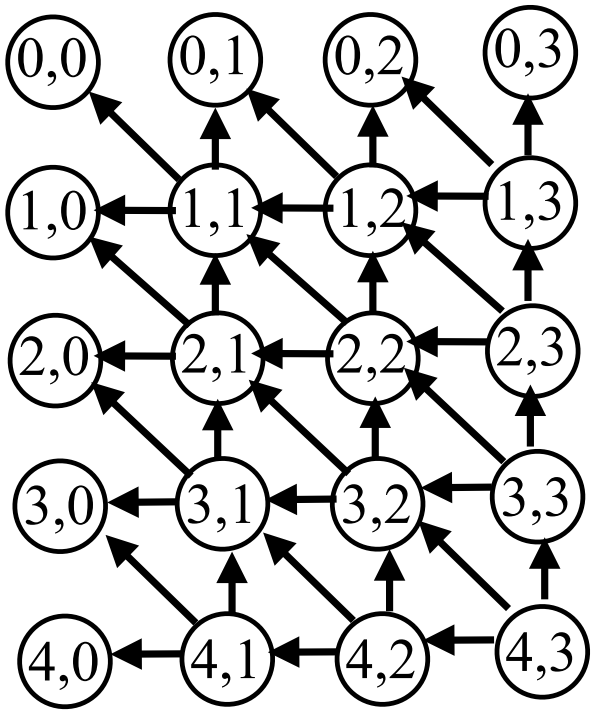
$\Rightarrow$ Une plus longue sous-séquence commune de deux séquences a pour préfixe une plus longue sous-séquence des préfixes des deux séquences.

Solution par programmation dynamique

- Soit $c[i,j]$ la longueur d'une plus longue sous-séquence de X_i et Y_j.
- Formulation récursive :

$$c[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0, \\ c[i-1,j-1]+1 & \text{si } i,j > 0 \text{ et } x_i = y_j, \\ \max(c[i-1,j], c[i,j-1]) & \text{si } i,j > 0 \text{ et } x_i \neq y_j, \end{cases}$$

- Graphe des sous-problèmes :



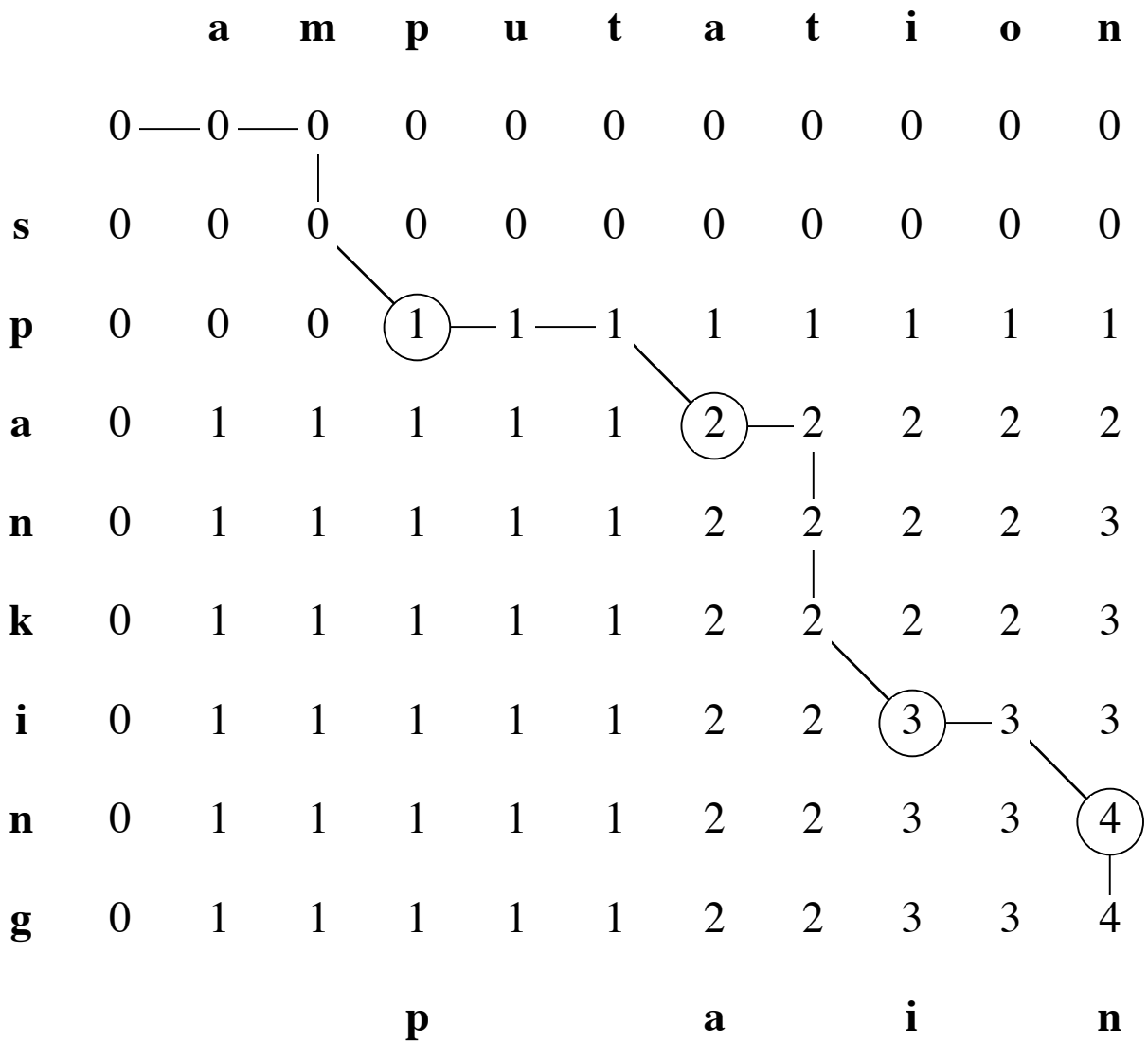
Implémentation (ascendante)

```
LCS-LENGTH(X, Y, m, n)
 1  Let c[0..m, 0..n] be a new table
 2  for i = 1 to m
 3        c[i,0]=0
 4  for j = 0 to n
 5        c[0,j]=0
 6  for i = 1 to m
 7        for j = 1 to n
 8              if x_i == y_j
 9                    c[i,j] = c[i-1,j-1] + 1
10              elseif c[i-1,j] ≥ c[i,j-1]
11                    c[i,j] = c[i-1,j]
12              else c[i,j] = c[i,j-1]
13  return c
```

Complexité : $\Theta(m \cdot n)$

Illustration

amputation versus *spanking*



Trouver la plus longue sous-séquence

```
LCS-Length(X, Y, m, n)
 1  Let c[0..m, 0..n] be a new table
 2  Let b[1..m, 1..n] be a new table
 3  for i = 1 to m
 4       c[i,0]=0
 5  for j = 0 to n
 6       c[0,j]=0
 7  for i = 1 to m
 8       for j = 1 to n
 9            if x_i == y_j
10                 c[i,j] = c[i-1,j-1] + 1
11                 b[i,j] = " ↖ "
12            elseif c[i-1,j] ≥ c[i,j-1]
13                 c[i,j] = c[i-1,j]
14                 b[i,j] = " ↑ "
15            else c[i,j] = c[i,j-1]
16                 b[i,j] = " ← "
17  return c and b
```

```
Print-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i,j] == " ↖ "
4       Print-LCS(b, X, i-1, j-1)
5       print x_i
6  elseif b[i,j] == " ↑ "
7       Print-LCS(b, X, i-1, j)
8  else Print-LCS(b, X, i, j-1)
```

Exemple 5 : le problème du sac à dos (knapsack)

Problème :

- Un voleur se rend dans un musée pour commettre un méfait avec un sac à dos pouvant contenir W kg.
- Le musée comprend n œuvres d'art, chacune de poids p_i et de prix v_i $(i = 1, \ldots, n)$
- Le problème pour le voleur est de déterminer une sélection d'objets de valeur totale maximale et n'excédant pas le poids total admissible dans le sac à dos.

Formellement :

- Soit un ensemble S de n objets de poids $p_i > 0$ et de valeurs $v_i > 0$
- Trouver $x_1, x_2, \ldots, x_n \in \{0, 1\}$ tels que :
  - $\sum_{i=1}^{n} x_i \cdot p_i \leq W$, et
  - $\sum_{i=1}^{n} x_i \cdot v_i$ est maximal.

Exemple

Capacité du sac à dos :

$$W = 11$$

i	v_i	p_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Exemple :

- $\{5, 2, 1\}$ a un poids de 10 et une valeur de 35
- $\{3, 4\}$ a un poids de 11 et une valeur de 40

Approche par force brute

- Recherche exhaustive : on énumère tous les sous-ensembles de S, et on calcule leur poids et leur valeur
- Complexité en temps : $O(n2^n)$
- Améliorations :
  - Ne tester que les sous-ensembles de W/p_{min} objets où p_{min} est la taille minimale
  - Tester les objets par ordre croissant et s'arrêter dès que l'un d'entre eux n'entre plus
- Diminue la constante mais la complexité reste la même

Approche par programmation dynamique

- Définition : soit $M(k, w)$, $0 \leq k \leq n$ et $0 \leq w \leq W$, le bénéfice maximum qu'on peut obtenir avec les objets $1, \ldots, k$ de S et un sac à dos de charge maximale w
  (On suppose que les poids p_i et W sont entiers)
- Deux cas :
  - On ne sélectionne pas l'objet k : $M(k, w)$ est le bénéfice maximum en sélectionnant parmi les $k-1$ premiers objets avec comme limite w ($M(k-1, w)$)
  - On sélectionne l'objet k : $M(k, w)$ est la valeur de l'objet k plus le bénéfice maximum en sélectionnant parmi les $k-1$ premiers objets avec la limite $w - p_k$

$$M(k,w) = \begin{cases} 0 & \text{si } k = 0 \\ M(k-1,w) & \text{si } p_k > w \\ \max\{M(k-1,w), v_k + M(k-1, w-p_k)\} & \text{sinon} \end{cases}$$

Implementation

```
KnapSack(p, v, n, W)
 1  Let M[0..n, 0..W] be a new table
 2  for w = 0 to W
 3        M[0,w]=0
 4  for k = 1 to n
 5        M[k, 0] = 0
 6  for k = 1 to n
 7        for w = 1 to W
 8              if p[k] > w
 9                    M[k, w] = M[k − 1, w]
10              elseif M[k − 1, w] > v[k] + M[k − 1, w − p[k]]
11                    M[k, w] = M[k − 1, w]
12              else M[k, w] = v[k] + M[i − 1, w − p[k]]
13  return M[n, W]
```

Exemple

M	0	1	2	3	4	5	6	7	8	9	10	11
$\emptyset$	0	0	0	0	0	0	0	0	0	0	0	0
$\{1\}$	0	1	1	1	1	1	1	1	1	1	1	1
$\{1,2\}$	0	1	6	7	7	7	7	7	7	7	7	7
$\{1,2,3\}$	0	1	6	7	7	18	19	24	25	25	25	25
$\{1,2,3,4\}$	0	1	6	7	7	18	22	24	28	29	29	40
$\{1,2,3,4,5\}$	0	1	6	7	7	18	22	28	29	34	35	40

Solution optimale : $\{4,3\}$
Bénéfice : $22 + 18 = 40$

$W = 11$

i	v_i	p_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Récupération des x_i

En remontant dans le tableau M :

```
KnapSack(p, v, n, W)
 1  // Compute M
 2  ...
 3  // Retrieve solution
 4  Let x[1..n] be a new table
 5  w = W
 6  for k = n downto 1
 7        if M[k, w] == M[k − 1, w]
 8              x[k] = 0
 9        else
10              x[k] = 1
11              w = w − p[k]
12  return x
```

Complexité

- Complexité en temps et en espace : $\Theta(nW)$
  - Remplissage de la matrice M : $\Theta(nW)$
  - Recherche de la solution : $\Theta(n)$

  (Exercice : proposez une version $\Theta(n + W)$ en espace)

- Note : L'algorithme n'est en fait pas polynomial en fonction de la taille de l'entrée
  - Si W nécessite n_w bits pour son codage, la complexité est $\Theta(n2^{n_w})$
  - Comme pour le voyageur de commerce, on n'a pas encore trouvé d'algorithme polynomial pour le problème du sac à dos (et il y a peu de chance qu'on y arrive)

Programmation dynamique : résumé

Grandes étapes :

- Caractériser la structure du problème
- Définir de manière récursive la valeur de la solution optimale
- Calculer les valeurs de la solution optimale (c'est-à-dire remplir un tableau)
- Reconstruire la (une) solution optimale à partir de l'information calculée ("bottom-up")

Applications :

- Command unix diff (comparaison de fichiers)
- Algorithme de Viterbi (reconnaissance vocale)
- Alignement de séquences d'ADN (Smith-Waterman)
- Plus court chemin dans un graphe (Bellman-Ford)
- Compilateurs (analyse syntaxique et optimisation du code)
- ...

Programmation dynamique versus diviser-pour-régner

- L'approche Diviser-pour-régner décompose aussi le problème en sous-problèmes
- Mais ces sous-problèmes sont significativement plus petits que le problème de départ ($n \rightarrow n/2$)
  - Alors que la programmation dynamique réduit généralement un problème de taille n en sous-problèmes de taille $n - 1$
- Et ces sous-problèmes sont indépendants
  - Alors qu'en programmation dynamique, ils se recouvrent
- Pour ces deux raisons, la récursivité ne fonctionne pas pour la programmation dynamique

Méthodes de résolution de problèmes

Quelques approches génériques pour aborder la résolution d'un problème :

- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Diviser pour régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original
- Programmation dynamique : obtenir la solution optimale à un problème en combinant des solutions optimales à des sous-problèmes similaires plus petits et se chevauchant
- Approche gloutonne : construire la solution incrémentalement, en optimisant de manière aveugle un critère local

Plan

Algorithme glouton (greedy)

- Utilisé pour résoudre des problèmes d'optimisation (comme la programmation dynamique)
- Idée principale :
  - Quand on a un choix local à faire, faire le choix (glouton) qui semble le meilleur tout de suite (et ne jamais le remettre en question)
- Pour que l'approche fonctionne, le problème doit satisfaire deux propriétés :
  - Propriété des choix gloutons optimaux : On peut toujours arriver à une solution optimale en faisant un choix localement optimal
  - Propriété de sous-structure optimale : Une solution optimale du problème est composée de solutions optimales à des sous-problèmes
- Même si ces propriétés ne sont pas satisfaites, l'approche gloutonne peut parfois fournir une approximation intéressante au problème
- Parfois, il est possible de caractériser la distance de la solution gloutonne à la solution optimale

Exemple 1 : rendre la monnaie

- Objectif : Etant donné des pièces de 1, 2, 5, 10, et 20 cents, trouver une méthode pour rembourser une somme de x cents en utilisant le moins de pièces possible.
- Exemple : 34 cents :
  - 1ère possibilité : $\{1, 1, 2, 5, 5, 20\} \rightarrow 6$ pièces
  - 2ième possibilité : $\{2, 2, 10, 20\} \rightarrow 4$ pièces

- Algorithme de la caissière : A chaque itération, ajouter une pièce de la plus grande valeur qui ne dépasse pas la somme restant à rembourser
- Exemple : 49 cents $\rightarrow \{20, 20, 5, 2, 2\}$ (5 pièces)

Implémentation

- Algorithme de la caissière : A chaque itération, ajouter une pièce de la plus grande valeur qui ne dépasse pas la somme restant à rembourser

```
CoinchangingGreedy(x, c, n)
1  // c[1..n] contains the n coin values in decreasing order
2  Let s[1..n] be a new table
3  // s[i] is the number of ith coin in solution
4  CoinCount = 0
5  for i = 1 to n
6        s[i] = ⌊x/c[i]⌋
7        x = x − s[i] ∗ c[i]
8        CoinCount = CoinCount + s[i]
9  return (s, CoinCount)
```

- Complexité : $\Theta(n)$ (sans compter le tri des pièces)
- Cet algorithme permet-il de trouver une solution optimale ?

Analyse de CoinChangingGreedy

Théorème : l'algorithme CoinChangingGreedy est optimal pour $c = [20, 10, 5, 2, 1]$

Preuve :

- Soit $S^*(x)$ l'ensemble optimal de pièces pour un montant x et soit c^* le plus grand $c[i] \leq x$. On doit montrer que :
  1. $S^*(x)$ contient c^* *(propriété des choix gloutons optimaux)*
  2. $S^*(x) = \{c^*\} \cup S^*(x - c^*)$ *(propriété de sous-structure optimale)*

- Propriété (2) découle directement de (1)
  - $S^*(x)$ contient c^* par (1)
  - Donc $S^*(x) \setminus \{c^*\}$ représente le change pour un montant de $x - c^*$
  - Ce change doit être optimal sinon $S' = \{c^*\} \cup S^*(x - c^*)$ serait une meilleure solution que $S^*(x)$ pour un montant de x
  - On a donc $S^*(x) = \{c^*\} \cup S^*(x - c^*)$

- Propriété (1) : $S^*(x)$ contient c^*
  - Avec $c = [20, 10, 5, 2, 1]$, une solution optimale ne contient jamais :
    - plus d'une pièce de 1, 5, ou de 10 (car $2 \times 1 = 2$, $2 \times 5 = 10$, $2 \times 10 = 20$)
    - plus de deux pièces de 2 (car $3 \times 2 = 5 + 1$)
  - Analysons les différents cas pour x :

$x = 1$: $c^* = 1$, meilleure solution $S^*(x) = \{1\}$ contient c^*

$2 \leq x < 5$: $c^* = 2$, avec un seul 1, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$

$x = 5$: $c^* = 5$, meilleure solution $S^*(x) = \{5\}$ contient c^*

$5 < x < 10$: $c^* = 5$, avec un seul 1, et deux 2, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$

$x = 10$: $c^* = 10$, meilleure solution $S^*(x) = \{10\}$ contient c^*

$10 < x < 20$: $c^* = 10$, avec un seul 1, deux 2, et 1 seul 5, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$

$x = 20$: $c^* = 20$, meilleure solution $S^*(x) = \{20\}$ contient c^*

$x > 20$: $c^* = 20$, avec un seul 1, deux 2, 1 seul 5 et 1 seul 10, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$

  - $S^*(x)$ contient donc toujours bien c^*

□

Analyse de CoinChangingGreedy

- L'approche greedy n'est correcte que pour certains choix particuliers de valeurs de pièces
  - ok pour la plupart des monnaies courantes, euros, dollars...
- Contre-exemple : $C = [1, 10, 21, 34, 70, 100]$ (valeurs de timbres aux USA) et $x = 140$
  - Algorithme glouton : $100, 34, 1, 1, 1, 1, 1, 1$
  - Solution optimale : $70, 70$

- Solution pour résoudre le cas général : programmation dynamique
- Très proche du problème de découpage de tige et du sac à dos

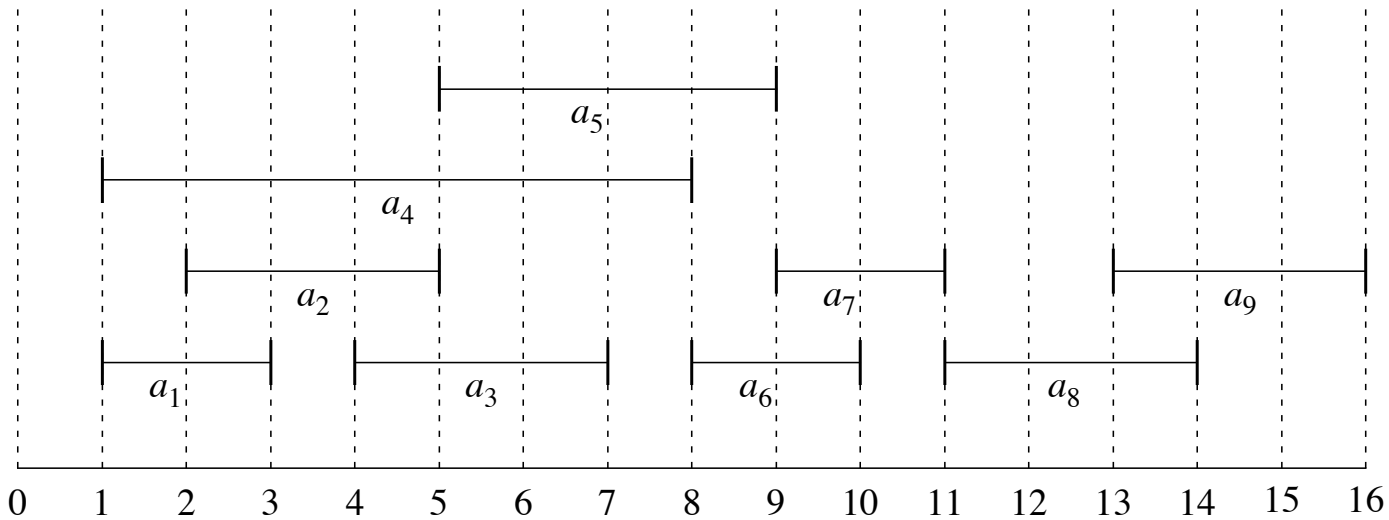
(Exercice : écrivez une fonction CoinChangeDP*)*

Exemple 2 : sélection d'activités

- Un salle est utilisée pour différentes activités
  - Soit $S = \{a_1, a_2, \ldots, a_n\}$ un ensemble de n activités
  - a_i démarre au temps s_i et se termine au temps f_i
  - Deux activités a_i et a_j sont compatibles si soit $f_i \leq s_j$, soit $f_j \leq s_i$

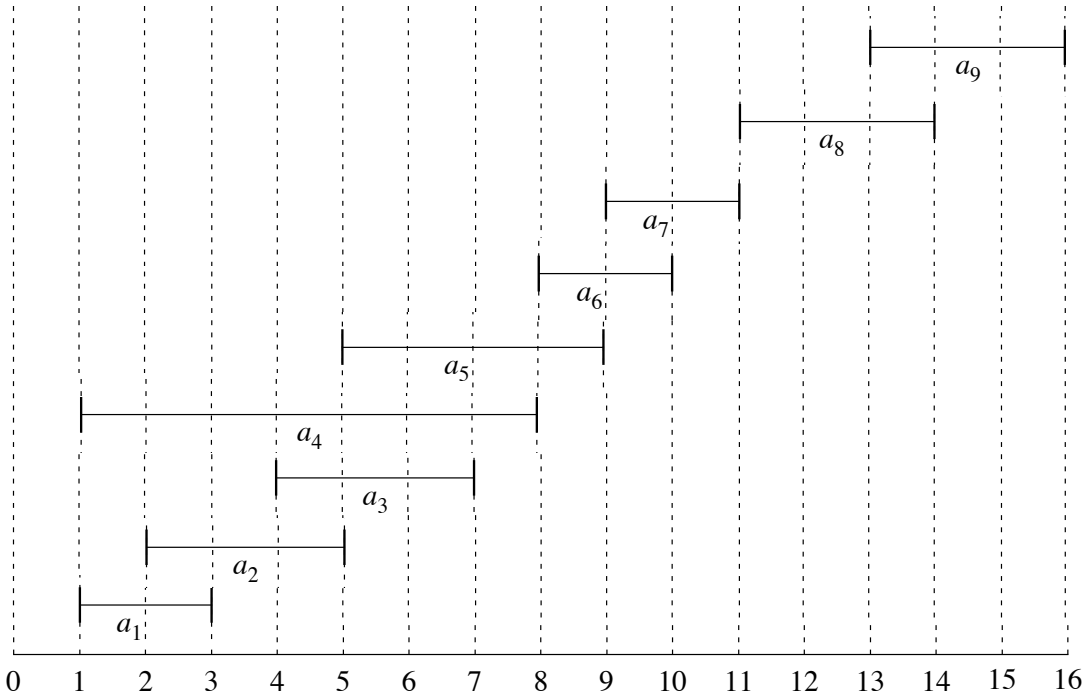
  Problème : trouver le plus grand sous-ensemble de tâches compatibles
- Exemple :

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



Sélection d'activités : approche gloutonne

- Schéma d'une solution gloutonne :
  - définir un ordre "naturel" sur les activités
  - sélectionner les activités dans cet ordre pour autant qu'elles soient compatibles avec celles déjà choisies
- Exemples : trier les activités selon s_i (début), selon f_i (fin), selon $f_i - s_i$ (durée), nombre de conflits avec d'autres activités...
- Montrer par des contre-exemples que seul le tri selon f_i fonctionne



Sélection d'activités : approche gloutonne

- Considérer les activités par ordre croissant de f_i et sélectionner chaque activité compatible avec celles déjà prises
- Implémentations : en supposant s et f ordonnés selon f

```
REC-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]
3        m = m + 1
4  if m ≤ n
5        return {a_m} ∪ ...
6              ...REC-ACTIVITY-SELECTOR(s, f, m, n)
7  else return ∅
```

```
ITER-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5        if s[m] ≥ f[k]
6              A = A ∪ {a_m}
7              k = m
8  return A
```

Appel initial :

REC-ACTIVITY-SELECTOR$(s, f, 0, s.length)$

- Complexité : $\Theta(n)$ ($+\Theta(n \log n)$ pour le tri selon f_i)

Sélection d'activités : analyse

- La solution gloutonne est-elle correcte ?

- 1. Propriété des choix gloutons optimaux : Soit $a_x \in S$ tel que $f_x \leq f_i$ pour tout $a_i \in S$. Il existe une solution optimale OPT^* qui contient a_x.
- Preuve :
  - Soit une solution optimale OPT telle que $a_x \notin OPT$
  - Soit a_m l'activité qui se termine en premier dans OPT
  - Construisons $OPT^* = (OPT \setminus \{a_m\}) \cup \{a_x\}$
  - OPT^* est valide :
    - Toute activité $a_i \in OPT \setminus \{a_m\}$ débute en un temps $s_i \geq f_m$
    - Par définition de a_x, $f_m \geq f_x$ et donc pour tout activité a_i, $s_i \geq f_x$
    - Toute activité a_i est donc compatible avec a_x
  - OPT^* est donc optimale puisque $|OPT^*| = |OPT|$

Sélection d'activités : analyse

- La solution gloutonne est-elle correcte ?

- 2. Propriété de sous-structure optimale : Soit $a_x \in S$ le choix glouton et $S' = \{a_i | s_i \geq f_x\}$ les activités de S compatibles avec a_x. Soit $OPT^* = \{a_x\} \cup OPT'$. Si OPT' est une solution optimale pour S' alors OPT^* est une solution optimale pour S.
- Preuve :
  - Soit OPT une solution optimale pour S
  - Si OPT^* n'est pas une solution optimale pour S, alors $|OPT^*| < |OPT|$ et donc aussi $|OPT'| < |OPT| - 1$
  - Soit a_m l'activité qui se termine en premier dans OPT et $\bar{S} = \{a_i | s_i \geq f_m\}$
  - Par construction, $OPT \setminus \{a_m\}$ est une solution pour $\bar{S}$
  - Par construction, $\bar{S} \subseteq S'$ et $OPT \setminus \{a_m\}$ est une solution valide pour S' (pas nécessairement optimale)
  - Ce qui veut dire qu'il existe une solution pour S' de taille $|OPT| - 1$, ce qui contredit $|OPT'| < |OPT| - 1$ et OPT' optimal pour S' (par hypothèse).

Problèmes similaires

D'autres problèmes similaires pour lesquels il existe un algorithme glouton :

- Allocation de resources :
  - Etant donnée un ensemble d'activités S avec leurs temps de début et de fin, trouver le nombre minimum de salles permettant de les réaliser toutes
- Planification de tâches :
  - Soit un ensemble de tâches avec leur durée et l'instant auquel elles doivent chacune être terminées (leur deadline)
  - Sachant qu'on ne peut exécuter qu'une seule tâche simultanément, trouver l'ordonnancement de ces tâches qui minimise le dépassement maximal des deadlines associées aux tâches (latence).

Exemple 3 : problème du sac à dos

Rappel : problème (0/1) du sac à dos :

- Soit un ensemble S de n objets de poids $p_i > 0$ et de valeur $v_i > 0$
- Trouver $x_1, x_2, \ldots, x_n \in \{0, 1\}$ tels que :
  - $\sum_{i=1}^{n} x_i \cdot p_i \leq W$, et
  - $\sum_{i=1}^{n} x_i \cdot v_i$ est maximal.

Solution par programmation dynamique : $\Theta(nW)$

Peut-on le résoudre par une approche gloutonne ?

Programmation dynamique versus approche gloutonne

- Rappel du transparent 396 :
  - soit $M(k, w)$, $0 \leq k \leq n$ et $0 \leq w \leq W$, le bénéfice maximum qu'on peut obtenir avec les objets $1,\ldots,k$ de S et un sac à dos de charge maximale w. On a :

$$M(k,w) = \begin{cases} 0 & \text{si } i = 0 \\ M(k-1,w) & \text{si } p_i > w \\ \max\{M(k-1,w), v_k + M(k-1,w-p_k)\} & \text{sinon} \end{cases}$$

- Approche gloutonne : consisterait à remplacer le max par le choix qui nous semble le meilleur localement
- Quels choix possibles ?
  - Le moins lourd, le plus lourd ?
  - Le moins coûteux, le plus coûteux ?
  - Le meilleur rapport valeur/poids ?

Approche gloutonne

- Idée d'algorithme :
  - Ajouter à chaque itération l'objet de rapport $\frac{v_i}{p_i}$ maximal qui rentre dans le sac
  - Implémentation très proche du problème de change : $\Theta(n \log n)$
- Est-ce que ça fonctionne ? Non !

i	v_i	p_i	v_i/p_i
1	1	1	1
2	6	2	3
3	18	5	3,6
4	22	6	3,7
5	28	7	4

W=11 :

- Solution greedy : $\{5, 2, 1\} \Rightarrow$ valeur=35
- Solution DP : $\{4, 3\} \Rightarrow$ valeur=40

Problème fractionnel du sac à dos (fractional knapsack)

Par rapport au problème 0/1, il est maintenant permis d'inclure des fractions d'objets (≤ 1) :

- Soit un ensemble S de n objets de poids $p_i > 0$ et de valeur $v_i > 0$
- Trouver $x_1, x_2, \ldots, x_n \in [0, 1]$ tels que :
  - $\sum_{i=1}^{n} x_i \cdot p_i \leq W$, et
  - $\sum_{i=1}^{n} x_i \cdot v_i$ est maximal.

Exemple :

i	v_i	p_i	v_i/p_i
1	1	1	1
2	6	2	3
3	18	5	3,6
4	22	6	3,7
5	28	7	4

W=11 :

- Solution optimale 0/1 : $x_1 = 0$, $x_2 = 0$, $x_3 = 1$, $x_4 = 1$, $x_5 = 0 \Rightarrow$ valeur=40
- Solution optimale fractionnelle : $x_1 = 0$, $x_2 = 0$, $x_3 = 0$, $x_4 = 2/3$, $x_5 = 1$ $\Rightarrow$ valeur=42,66

Algorithme glouton

- Pour la version fractionnelle, l'algorithme glouton est optimal
- Implémentation :

```
FracKnapSack(p, v, n, W)
1  // Assume the objects are sorted according to v[i]/p[i]
2  Let x[1..n] a new table
3  w = 0
4  for i = 1 to n
5        d = min(p[i], W − w)
6        w = w + d
7        x[i] = d/p[i]
8  return x
```

- Complexité : $\Theta(n)$ ($+ \Theta(n \log n)$ pour le tri)

Correction

Théorème : Le problème fractionnel du sac à dos possède la propriété des choix gloutons optimaux

Preuve :

- Soit deux objets i et j tels que
$$\frac{v_i}{p_i} > \frac{v_j}{p_j}$$
- Etant donné un choix $(x_1, x_2, \ldots, x_n)$, on le transforme en $(x'_1, x'_2, \ldots, x'_n)$ tel que :
  - $\forall k \in [1, n] \setminus \{i, j\} : x'_k = x_k$,
  - $x'_i = x_i + \frac{\Delta}{p_i}$, et
  - $x'_j = x_j - \frac{\Delta}{p_j}$,

  où $\Delta = \min(p_i(1 - x_i), p_j x_j)$.
- Cette transformation ne modifie pas le poids total, mais améliore le bénéfice.
- On en déduit qu'il est toujours avantageux de prendre la fraction maximale de l'objet i possédant le plus grand rapport $\frac{v_i}{p_i}$. □

Algorithme glouton : résumé

- Très efficaces quand ils fonctionnent. Simples et faciles à implémenter
- Ne fonctionnent pas toujours. Leur correction peut être assez difficile à prouver

Applications :

- Arbre de couverture minimal (Kruskal, Prim)
- Plus court chemin dans un graphe (algorithme de Dijkstra)
- Allocation de resources
- Codage de Huffman
- ...

Approche gloutonne versus programmation dynamique

- Tous deux nécessitent la propriété de sous-structure optimale
- Les algorithmes gloutons nécessitent que la propriété de choix gloutons optimaux soit satisfaite
  - On n'a pas besoin de solutionner plus d'un sous-problème
  - Le choix glouton est fait avant de résoudre le sous-problème
  - Il n'y a pas besoin de stocker les résultats intermédiares
- La programmation dynamique marche sans la propriété des choix gloutons optimaux
  - On doit solutionner plusieurs sous-problèmes et choisir dynamiquement l'un deux pour obtenir la solution globale
  - La solution doit être assemblée "bottom-up"
  - Les sous-solutions aux sous-problèmes sont réutilisées et doivent donc être stockées

Exemple 4 : codage de Huffman

- Soit une séquence S très longue définie sur base de 6 caractères : a, b, c, d, e et f
  - Par exemple, $n = |S| = 10^9$
- Quelle est la manière la plus efficace de stocker cette séquence ?
- Première approche : encoder chaque symbole par un mot binaire de longueur fixe :

Symbole	a	b	c	d	e	f
Codage	000	001	010	011	100	101

  - 6 symboles nécessitent 3 bits par symbole
  - $3 \times 10^9/8 = 3.75 \times 10^8$bytes (un peu moins de 400Mb)
- Peut-on faire mieux ?

Idée

- Codage avec des mots de longueur fixe :

Symbole	a	b	c	d	e	f
Codage	000	001	010	011	100	101

- Observation : l'encodage de e et f est redondant :
  - Le second bit ne nous aide pas à distinguer e de f
  - En d'autres termes, si le premier bit est 1, le second ne nous donne pas d'information et peut être supprimé
- Suggère de considérer un codage avec des mots binaires de longueurs variables

Symbole	a	b	c	d	e	f
Codage	000	001	010	011	10	11

- Encodage et décodage sont bien définis et non ambigüs
- Permet de gagner $n_e + n_f$ bits, où n_e et n_f sont les nombres de e et de f dans la séquence

Définition du problème

- Soit un ensemble de symboles C et $f(c)$ la fréquence du symbole $c \in C$.
- Trouver un code $E : C \to \{0,1\}^*$ tel que
  - E est un code sans préfixe
    - Aucun mot de code $E(c_1)$ n'est le préfixe d'un autre mot de code $E(c_2)$
  - La longueur moyenne des mots de code est minimale

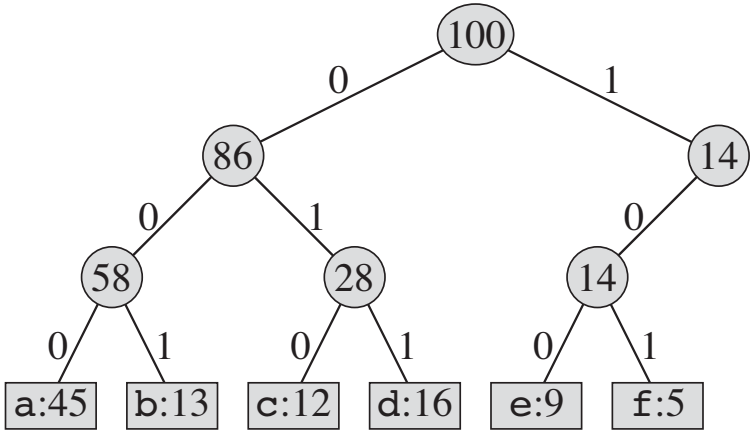
$$B(S) = \sum_{c \in C} f(c)|E(c)|$$

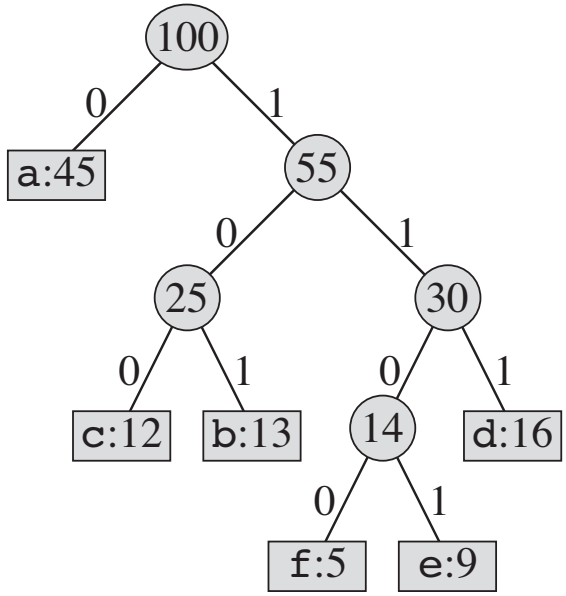
($nB(S)$ est la longueur de l'encodage de S)

- Exemple :

c	a	b	c	d	e	f	
f(c)	45%	13%	12%	16%	9%	5%	$B(S)$
Code 1	000	001	010	011	100	101	3.00
Code 2	000	001	010	011	10	11	2.86
Code 3	0	101	100	111	1101	1100	2.24

Code sans préfixe





- Un code sans préfixe peut toujours se représenter sous la forme d'une arbre binaire
  - Chaque feuille est associée à un symbole
  - Le chemin de la racine à une feuille est le code du symbole
  - La fréquence d'un nœud est la fréquence du préfixe
- Un code optimal est toujours représenté par un arbre binaire entier *(Pourquoi ?)*

Algorithme glouton

On peut montrer que le codage optimal peut être obtenu par un algorithme glouton

- On construit l'arbre de bas en haut en partant des feuilles
- A chaque étape, on fait le choix "glouton" de fusionner les deux nœuds les moins fréquents (symboles ou préfixes)

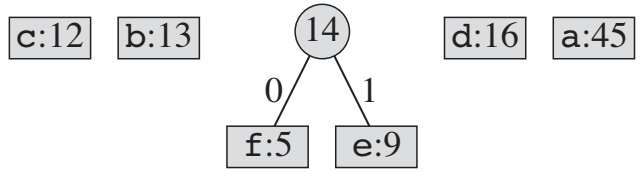
Idée de la preuve (pour information) :

- Choix gloutons optimaux :
  - Il existe un code sans préfixe optimal où les deux symboles les moins fréquents sont frères et à la profondeur maximale
  - Par l'absurde : si un tel code n'existait pas, on pourrait l'obtenir en échangeant la position des deux symboles les moins fréquents avec les feuilles les plus profondes sans augmenter $B(S)$
- Sous-structure optimale :
  - Si l'arbre qui a pour feuille le nouveau nœud issu de la fusion gloutonne est optimal, l'arbre complet est optimal
  - Plus difficile à montrer

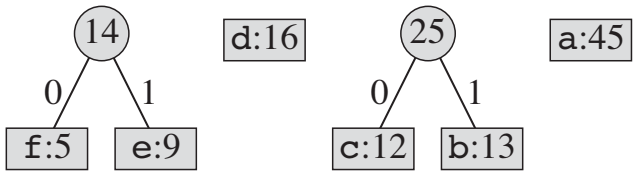
Algorithme glouton : exemple

(a)

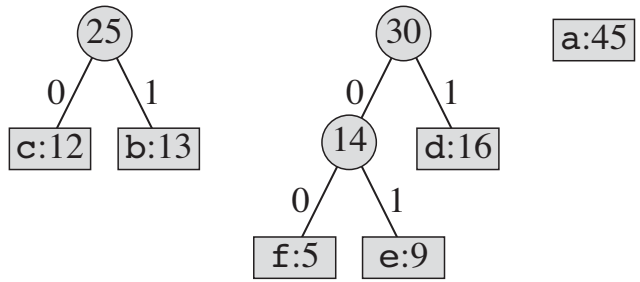
(b)



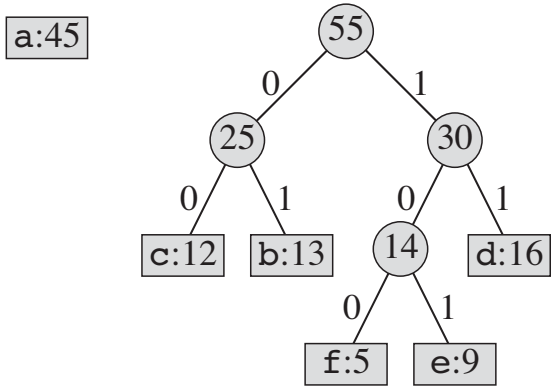
(c)



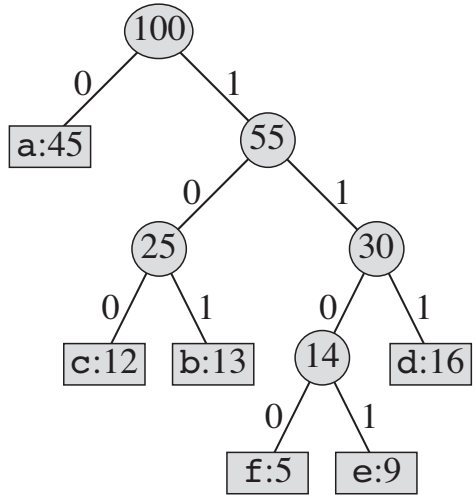
(d)



(e)



(f)



Algorithme glouton : implémentation

```
HUFFMAN(C)
1   n = |C|
2   Q =”create a min-priority queue from C”
3   for i = 1 to n − 1
4         Allocate a new node z
5         z.left = EXTRACT-MIN(Q)
6         z.right = EXTRACT-MIN(Q)
7         z.freq = z.left.freq + z.right.freq
8         INSERT(Q, z)
9   return EXTRACT-MIN(Q)
```

- Implémentation avec une file à priorité
- Complexité : $O(n \log n)$ si Q est implémentée avec un tas (min)
  - Ligne 2 : $O(n)$ si on utilise BUILD-MIN-HEAP
  - Ligne 8 : $O(\log n)$ (répétée $n - 1$ fois)

Partie 7

Graphes

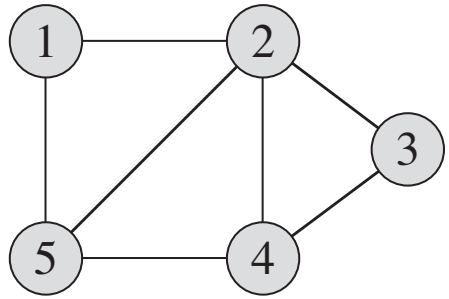
Plan

1. Définitions
2. Représentation des graphes
3. Parcours de graphes
4. Plus courts chemins
   - Définitions et algorithme général
   - Bellman-Ford
   - Dijkstra
   - Floyd-Warshall
5. Arbre couvrant

Graphes

- Un graphe (dirigé) est un couple (V, E) où :
  - V est un ensemble de nœuds (*nodes*), ou sommets (*vertices*) et
  - $E \subseteq V \times V$ est un ensemble d'arcs, ou arêtes (*edges*).
- Un graphe non dirigé est caractérisé par une relation symmétrique entre les sommets
  - Une arête est un ensemble $e = \{u, v\}$ de deux sommets
  - On la notera tout de même (u, v) (équivalent à (v, u)).
- Applications : modélisation de :
  - Réseaux sociaux
  - Réseaux informatiques
  - World Wide Web
  - Cartes routières
  - ...

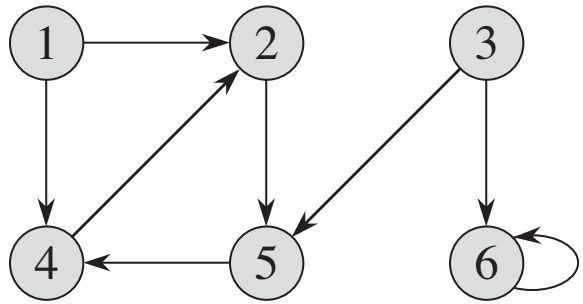
Terminologie : graphe non dirigé



$$V = \{1, 2, 3, 4, 5\}, E = \{(1, 2), (1, 5), (2, 4), (2, 5), (2, 3), (3, 4), (4, 5)\}$$

- Deux nœuds sont adjacents s'ils sont liés par une même arête
- Une arête (v_1, v_2) est dite incidente aux nœuds v_1 et v_2
- Le degré d'un nœud est égal au nombre de ses arêtes incidentes
- Le degré d'un graphe est le nombre maximal d'arêtes incidentes à tout sommet.
- Un graphe est connexe s'il existe un chemin de tout sommet à tout autre.
- Une composante connexe d'un graphe non orienté est un sous-graphe connexe maximal de ce graphe

Terminologie : graphe dirigé



$V = \{1, 2, 3, 4, 5, 6\}, E = \{(1, 2), (1, 4), (2, 5), (3, 5), (3, 6), (4, 2), (5, 4), (6, 6)\}$

- Une arête (v_1, v_2) possède l'origine v_1 et la destination v_2. Cette arête est sortante pour v_1 et entrante pour v_2
- Le degré entrant (*in-degree*) et le degré sortant (*out-degree*) d'un nœud v sont respectivement égaux aux nombres d'arêtes entrantes et d'arêtes sortantes de v
- Un graphe est acyclique s'il n'y a aucun cycle, c'est-à-dire s'il n'est pas possible de suivre les arêtes du graphes à partir d'un sommet x et de revenir à ce même sommet x

Type de graphes

- Un graphe est simple s'il ne possède pas de boucle composée d'une seule arête, c'est-à-dire tel que :

$$\forall v \in V : (v, v) \notin E$$

- Un arbre est un graphe acyclique connexe
- Un multigraphe est une généralisation des graphes pour laquelle il est permis de définir plus d'une arête liant un sommet à un autre

- Un graphe est pondéré si les arêtes sont annotées par des poids
  - Exemple : réseau entre villes avec comme poids la distance entre les villes, réseau internet avec comme poids la bande passante entre routeurs, etc.

Représentation I : listes d'adjacences

Un objet G de type graphe est composé :

- d'une liste de nœuds $G.V = \{1, 2, \ldots, |V|\}$
- d'un tableau $G.Adj$ de $|V|$ listes tel que :
  - Chaque sommet $u \in G.V$ est représenté par un élément du tableau $G.Adj$
  - $G.Adj[u]$ est la liste d'adjacence de u, c'est-à-dire la liste des sommets v tels que $(u, v) \in E$

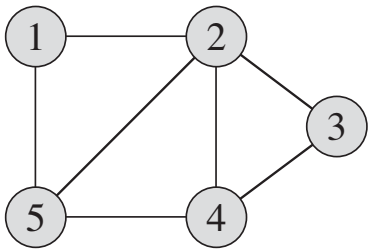
Permet de représenter des graphes dirigés ou non

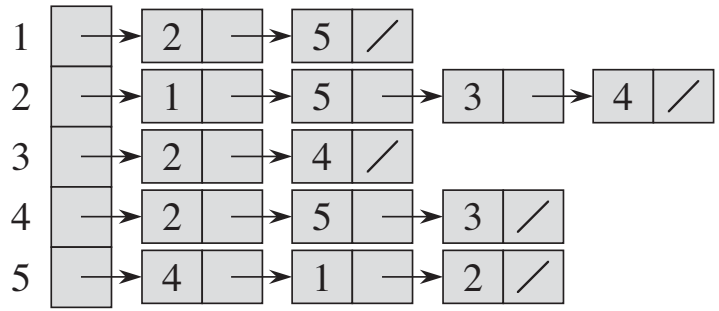
- Si le graphe est dirigé (resp. non dirigé), la somme des longueurs des listes de $G.Adj$ est $|E|$ (resp. $2|E|$).

Permet de représenter un graphe pondéré en associant un poids à chaque élément de liste

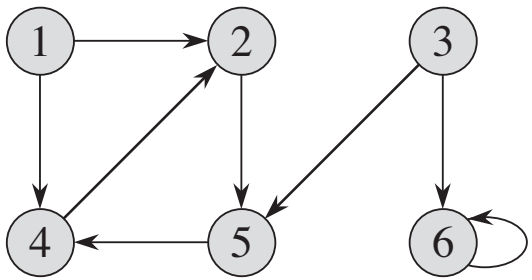
Exemple

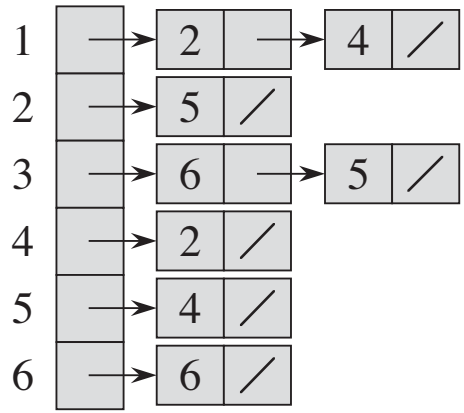
Graphe non dirigé





Graphe dirigé





Complexités

- Complexité en espace : $\Theta(|V| + |E|)$
  - optimal
- Accéder à un sommet : $\Theta(1)$
  - optimal
- Parcourir tous les sommets : $\Theta(|V|)$
  - optimal
- Parcourir toutes les arêtes : $\Theta(|V| + |E|)$
  - ok (mais pas optimal)
- Vérifier l'existence d'une arête $(u, v) \in E$: $O(|V|)$
  - ou encore $O(min(degree(u), degree(v)))$
  - mauvais

(Exercice : insertion, suppression de nœuds et d'arêtes ?)

Réprésentation II : matrice d'adjacence

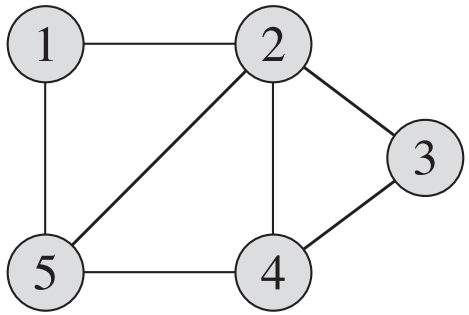
- Les nœuds sont les entiers de 1 à $|V|$, $G.V = \{1, 2, \ldots, |V|\}$
- G est décrit par une matrice $G.A$ de dimension $|V| \times |V|$
- $G.A = (a_{ij})$ tel que

$$a_{ij} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{sinon} \end{cases}$$

- Permet de représenter des graphes dirigés ou non
  - $G.A$ est symmétrique si le graphe est non dirigé
- Graphe pondéré : a_{ij} est le poids de l'arête (i,j) si elle existe, NIL (ou 0, ou $+\infty$) sinon

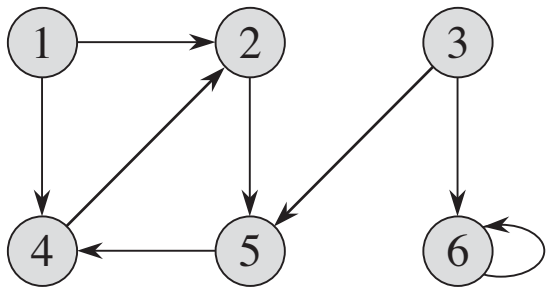
Exemple

Graphe non dirigé



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Graphe dirigé



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Complexités

- Complexité en espace : $\Theta(|V|^2)$
  - potentiellement très mauvais
- Accéder à un sommet : $\Theta(1)$
  - optimal
- Parcourir tous les sommets : $\Theta(|V|)$
  - optimal
- Parcourir toutes les arêtes : $\Theta(|V|^2)$
  - potentiellement très mauvais
- Vérifier l'existence d'une arête $(u, v) \in E$: $\Theta(1)$
  - optimal

(Exercice : insertion, suppression de nœuds et d'arêtes ?)

Représentations

- Listes d'adjacence :
  - Complexité en espace optimal
  - Pas appropriée pour des graphes denses [3] et des algorithmes qui ont besoin d'accéder aux arêtes
  - Préférable pour des graphes creux [4] ou de degré faible

- Matrice d'adjacence :
  - Complexité en espace très mauvaise pour des graphes creux
  - Appropriée pour des algorithmes qui désirent accéder aléatoirement aux arêtes
  - Préférable pour des graphes denses

3. $|E| \approx |V|^2$
4. $|E| \ll |V|^2$

Plan

Parcours de graphes

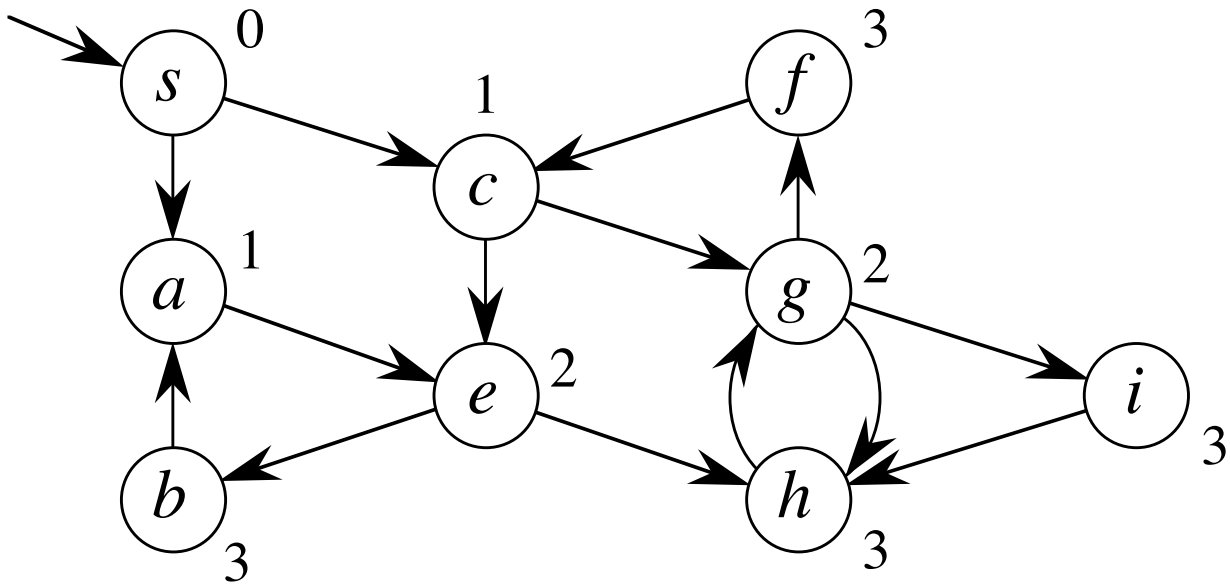
- Objectif : parcourir tous les sommets d'un graphe qui sont accessibles à partir d'un sommet v donné
- Un sommet v' est accessible à partir de v si :
  - soit $v' = v$,
  - soit v' est adjacent à v,
  - soit v' est adjacent à un sommet v'' qui est accessible à partir de v
- Différents types de parcours :
  - En profondeur d'abord (*depth-first*)
  - En largeur d'abord (*breadth-first*)

Parcours en largeur d'abord (*breadth-first search*)

- Un des algorithmes les plus simples pour parcourir un graphe
- A la base de plusieurs algorithmes de graphe importants

- Entrées : un graphe $G = (V, E)$ et un sommet $s \in V$
  - Parcourt le graphe en visitant tous les sommets qui sont accessibles à partir de s
  - Parcourt les sommets par ordre croissant de leur distance (en nombre minimum d'arêtes) par rapport à s
    - on visite s
    - tous les voisins de s
    - tous les voisins des voisins de s
    - etc.
  - Fonctionne aussi bien pour des graphes dirigés que non dirigés

Exemple



Un parcours en largeur à partir de s : s-a-c-e-g-b-h-i-f

Pour l'implémentation :

- On doit retenir les sommets déjà visités de manière à éviter de boucler infiniment
- On doit retenir les sommets visités dont on n'a pas encore visité les voisins

Parcours en largeur d'abord : implémentation

```
BFS(G, s)
 1  for each vertex u ∈ G.V \ {s}
 2        u.d = ∞
 3  s.d = 0
 4  Q = "create empty Queue"
 5  ENQUEUE(Q, s)
 6  while not QUEUE-EMPTY(Q)
 7        u = DEQUEUE(Q)
 8        for each v ∈ G.Adj[u]
 9              if v.d = ∞
10                    v.d = u.d + 1
11                    ENQUEUE(Q, v)
```

- $v.d$ est la distance de v à s
  - si un sommet v a été visité, $v.d$ est fini
  - on peut remplacer d par un drapeau binaire
- Q est une file (LIFO) qui contient les sommets visités mais dont les voisins n'ont pas encore été visités

Parcours en largeur d'abord : complexité

```
BFS(G, s)
 1  for each vertex u ∈ G.V \ {s}
 2        u.d = ∞
 3  s.d = 0
 4  Q = ∅
 5  ENQUEUE(Q, s)
 6  while Q ≠ ∅
 7        u = DEQUEUE(Q)
 8        for each v ∈ G.Adj[u]
 9              if v.d = ∞
10                    v.d = u.d + 1
11                    ENQUEUE(Q, v)
```

- Chaque sommet est enfilé au plus une fois ($v.d$ infini $\rightarrow$ $v.d$ fini)
- Boucle **while** exécutée $O(|V|)$ fois
- Boucle interne : $O(|E|)$ au total
- Au total : $O(|V| + |E|)$

Parcours en largeur d'abord

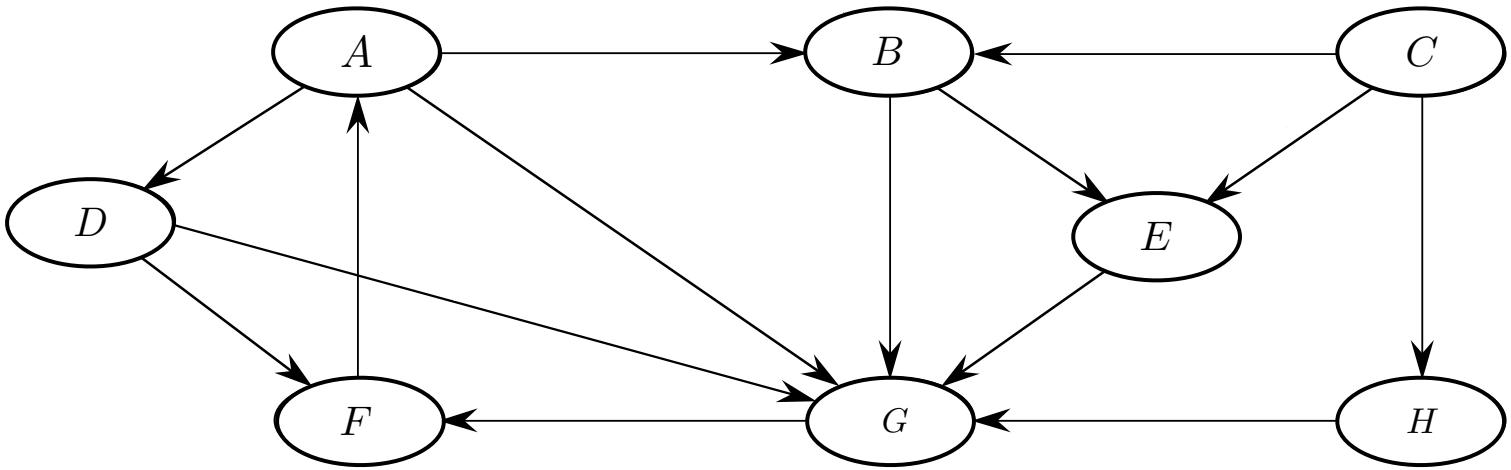
- Correction :
  - L'algorithme fait bien un parcours du graphe en largeur et $v.d$ contient bien la distance minimale de s à v
  - Ok intuitivement mais pas évident à montrer formellement. On le fera plus loin pour l'algorithme de Dijkstra (calcul du plus court chemin)

- Applications :
  - Calcul des plus courtes distances d'un sommet à tous les autres
  - Recherche du plus court chemin entre deux sommets
  - Calcul du diamètre d'un arbre
  - Tester si un graphe est biparti
  - ...

Parcours en profondeur d'abord

- Parcours du graphe en profondeur :
  - On suit immédiatement les arêtes incidentes au dernier sommet visité
    - Au lieu de les mettre en attente dans une file
  - On revient en arrière (*backtrack*) quand le sommet visité n'a plus de sommets adjacents non visités

- Exemple :



Parcours en profondeur à partir de A : A-D-F-G-B-E (C et H pas accessibles)

Parcours en profondeur : implémentation avec une pile

```
DFS(G, s)
 1  for each vertex u ∈ G.V
 2        u.visited = FALSE
 3  S ="create empty stack"
 4  PUSH(S, s)
 5  while not STACK-EMPTY(S)
 6        u = POP(S)
 7        if u.visited == FALSE
 8              u.visited = TRUE
 9              for each v ∈ G.Adj[u]
10                    if v.visited == FALSE
11                          PUSH(S, v)
```

- On remplace la file Q par une pile S
- L'attribut *visited* marque les sommets visités

- Initialisation : $\Theta(|V|)$
- Boucle **while** : $O(|E| + |V|)$ car :
  - Ligne 8 : $O(|V|)$ fois au total
  - Ligne 10-11 : $O(|E|)$ fois au total
  - Ligne 6 : $O(|E|)$ fois au total
- Complexité totale : $O(|V| + |E|)$

Parcours en profondeur : implémentation récursive

```
DFS(G, s)
1  for each vertex u ∈ G.V
2        u.visited = FALSE
3  DFS-REC(G, s)
```

```
DFS-REC(G, s)
1  s.visited = TRUE
2  for each v ∈ G.Adj[s]
3        if v.visited == FALSE
4              DFS-REC(G, v)
```

- Remplace la pile par la récursion
- DFS-REC appelée au plus $|V|$ fois
- Chaque arête est considérée au plus une fois dans la boucle **for**
- Complexité : $O(|V| + |E|)$

Parcourir tous les sommets d'un graphe

- BFS et DFS ne visitent que les nœuds accessibles à partir de la source s
  - Graphe non dirigé : seule la composante connexe contenant s est visitée
  - Graphe dirigé : certains sommets peuvent ne pas être accessibles de s en suivant le sens des arêtes
- Pour parcourir tous les sommets d'un graphe :
  1. On choisit un sommet arbitraire v
  2. On visite tous les sommets accessibles depuis v (en profondeur ou en largeur)
  3. S'il reste certains sommets non visités, on en choisit un et on retourne en (2)

Parcours en profondeur de tous les sommets

```
DFS-ALL(G)
1  for each vertex u ∈ G.V
2        u.visited = FALSE
3  for each vertex u ∈ G.V
4        if u.visited == FALSE
5              DFS-REC(G, u)
```

```
DFS-REC(G, s)
1  s.visited = TRUE
2  for each v ∈ G.Adj[s]
3        if v.visited == FALSE
4              DFS-REC(G, v)
```

- Complexité : $\Theta(|V| + |E|)$
  - DFS-REC est appelé sur chaque sommet une et une seule fois

$$\Theta(|V|)$$

  - La boucle **for** de DFS-REC parcourt chaque liste d'adjacence une et une seule fois

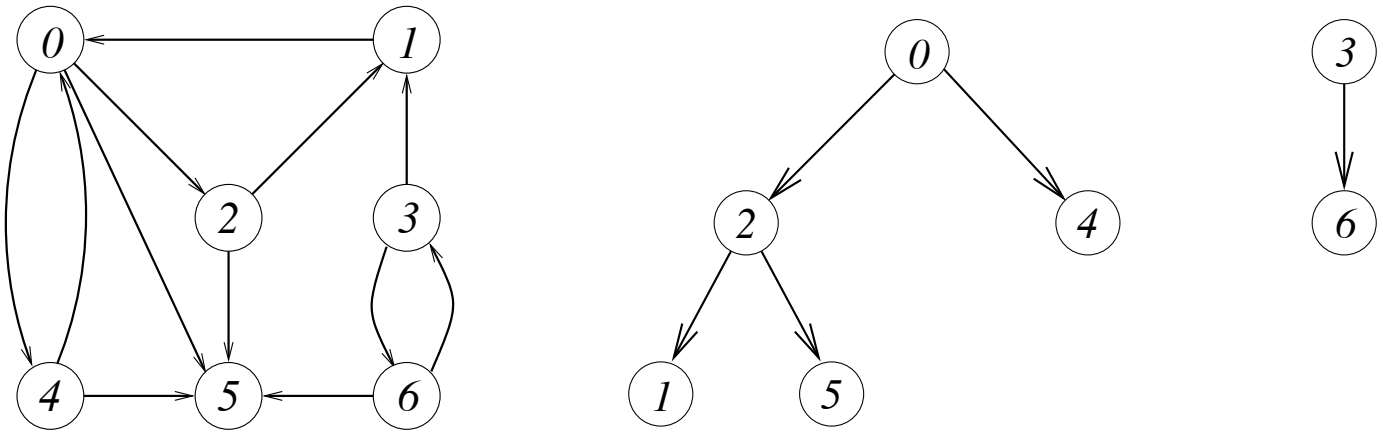
$$\Theta(\sum_{u \in G.V} outdegree(u)) = \Theta(|E|)$$

Sous-graphe de liaison

Un parcours en profondeur de tous les sommets d'un graphe construit un ensemble d'arbres (une forêt), appelé sous-graphe de liaison, où :

- les sommets sont les sommets du graphe,
- un sommet w est le fils d'un sommet v dans la forêt si $\text{DFS-REC}(G, w)$ est appelé depuis $\text{DFS-REC}(G, v)$

Exemple :



(Exercice : modifiez DFS-ALL *et* DFS-REC *pour construire la forêt)*

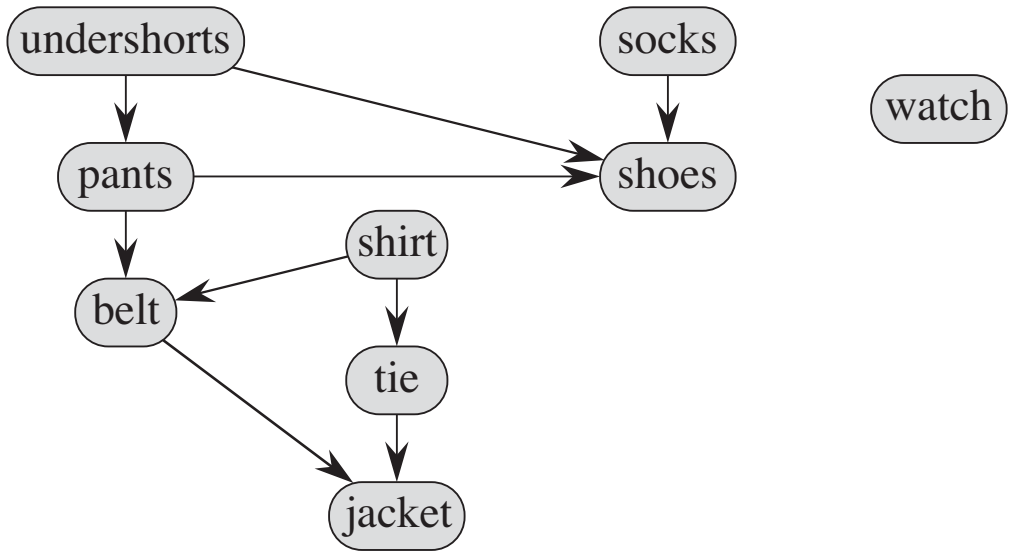
Application : tri topologique

- Tri topologique :
  - Etant donné un graphe acyclique dirigé (DAG), trouver un ordre des sommets tel qu'il n'y ait pas d'arête d'un nœud vers un des nœuds qui le précèdent dans l'ordre
  - On peut montrer que c'est possible si (et seulement si) le graphe est acyclique

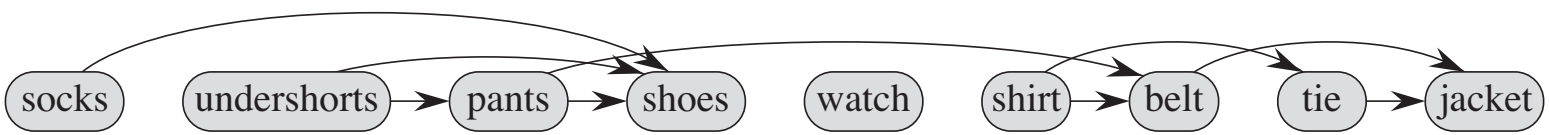
- Exemples d'applications :
  - Trouver un ordre pour suivre un ensemble de cours qui tienne compte des prérequis de chaque cours
    - Pour suivre SDA, il faut avoir suivi Introduction à la programmation
  - Résoudre les dépendances pour l'installation de logiciels
    - Trouver un ordre d'installation de manière à ce que chaque logiciel soit installé après tous ceux dont il dépend

Illustration

Graphe



Un tri topologique



Marquage des sommets pour le parcours en profondeur

- Dans le cadre d'un parcours en profondeur de tous les sommets, DFS-REC est appelé une et une seule fois sur chaque sommet
- Lors de l'exécution de DFS-ALL, on dira qu'un sommet v est fini si l'appel DFS-REC(G, v) est terminé
- A un moment donné, les sommets peuvent être dans les trois états suivants :
  - pas encore visité (on dira que v est blanc)
  - visité mais pas encore fini (v est gris)
  - fini (v est noir)

Marquage des sommets pour le parcours en profondeur

```
DFS-ALL(G)
1  for each vertex u ∈ G.V
2        u.color = WHITE
3  for each vertex u ∈ G.V
4        if u.color == WHITE
5              DFS-REC(G, u)
```

```
DFS-REC(G, s)
1  s.color = GRAY
2  for each v ∈ G.Adj[s]
3        if s.color == WHITE
4              DFS-REC(G, v)
5  s.color = BLACK
```

- Lemme. Soit s un sommet de G. Considérons le moment de l'exécution de DFS-ALL(G) où DFS-REC(G, s) est appelé. Pour tout sommet v, on a :
  1. Si v est blanc et accessible depuis s, alors v sera noir avant s
  2. Si v est gris, alors s est accessible depuis v

Trouver un tri topologique par DFS

- Soit un graphe $G = (V, E)$ et l'ordre suivant défini sur V :

$$s \prec v \Leftrightarrow v \text{ devient noir avant } s$$

- Si G est un DAG, alors $\prec$ définit un ordre topologique sur G

- Preuve :
  - Soit $(s, v) \in E$. On doit montrer que $s \prec v$.
  - Considérons le moment où $\mathrm{DFS\text{-}REC}(G, s)$ est appelé :
    - Si v est déjà noir, alors $s \prec v$ par définition de $\prec$
    - Si v est blanc, alors v sera noir avant s par le lemme précédent. Donc $s \prec v$
    - Si v est gris, s est accessible depuis v et donc il y a un cycle (puisque $(s, v) \in E$). Ce qui ne peut pas arriver vu que G est un DAG

□

Tri topologique : implémentation

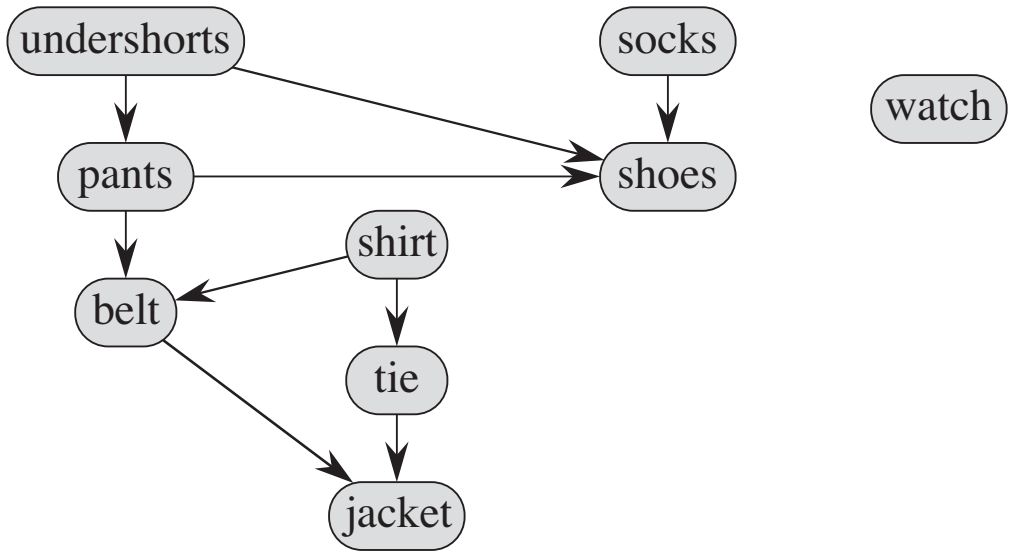
```
TOP-SORT(G)
1  for each vertex u ∈ G.V
2        u.color = WHITE
3  L ="create empty linked list"
4  for each vertex u ∈ G.V
5        if u.color == WHITE
6              TOP-SORT-REC(G, u, L)
7  return L
```

```
TOP-SORT-REC(G, s, L)
1  s.color = GRAY
2  for each v ∈ G.Adj[s]
3        if s.color == WHITE
4              TOP-SORT-REC(G, v, L)
5        elseif s.color == GREY
6              ERROR "G has a cycle"
7  s.color = BLACK
8  INSERT-FIRST(L, s)
```

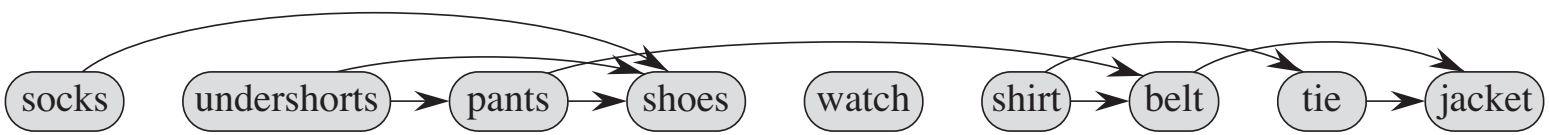
Complexité : $\Theta(|V| + |E|)$

Illustration

Graphe



Un tri topologique



Une autre solution

- Approche gloutonne :
  - Rechercher un sommet qui n'a pas d'arête entrante
    - C'est toujours possible dans un graphe acyclique
  - Ajouter ce sommet à un tri topologique du graphe dont on a retiré ce sommet et toutes ses arêtes
    - Ce graphe reste acyclique
- Complexité identique à l'approche DFS : $\Theta(|E| + |V|)$

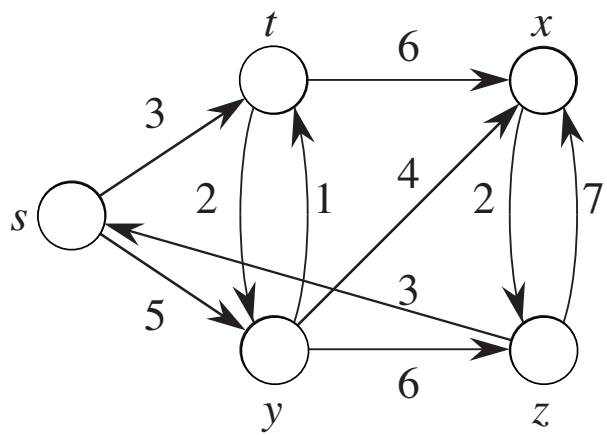
Plan

Définitions

- Soit un graphe dirigé $G = (V, E)$ et une fonction de poids $w : E \to IR$
- Un chemin (du sommet v_1 au sommet v_k) est une séquence de nœuds $v_1, v_2, \ldots, v_k$ telle que $\forall i = 1, \ldots, k-1$, $(v_i, v_{i+1}) \in E$.
- Le poids (ou coût) d'un chemin p est la somme du poids des arêtes qui le composent :

$$w(p) = w(v_1, v_2) + w(v_2, v_3) + \ldots + w(v_{k-1}, v_k)$$

- Exemple



$$w(s \to y \to t \to x \to z) = 5 + 1 + 6 + 2 = 14$$

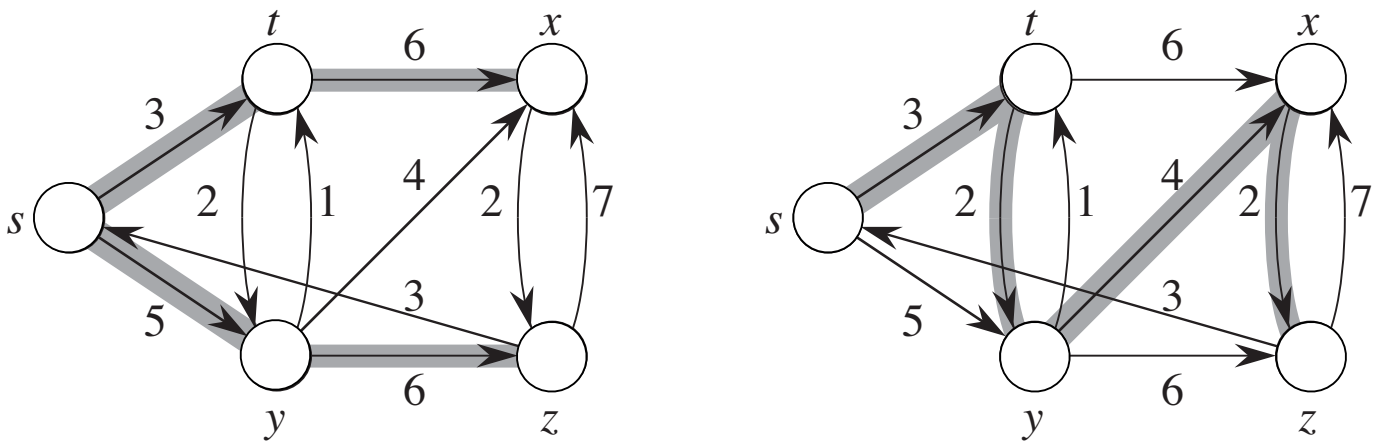
Plus courts chemins : définition

- Un plus court chemin entre deux sommets u et v est un chemin p de u à v de poids $w(p)$ le plus faible possible
- $\delta(u, v)$ est le poids d'un plus court chemin de u à v :

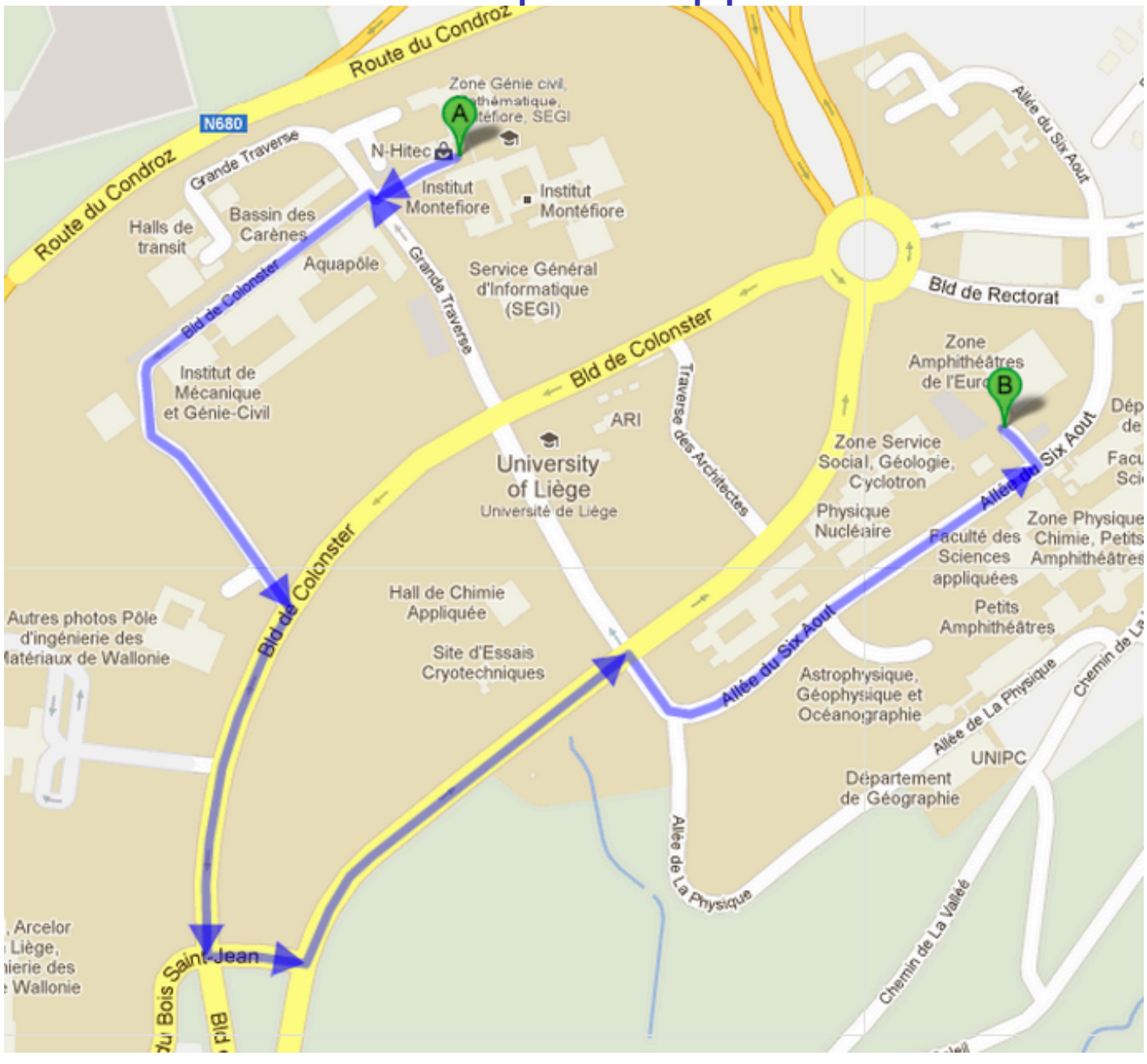
$$\delta(u, v) = \min\{w(p) | p \text{ est un chemin de } u \text{ à } v\}$$

(S'il n'y a pas de chemin entre u et v, $\delta(u, v) = \infty$ par définition)

- Exemples :



Plus courts chemins : exemple d'application

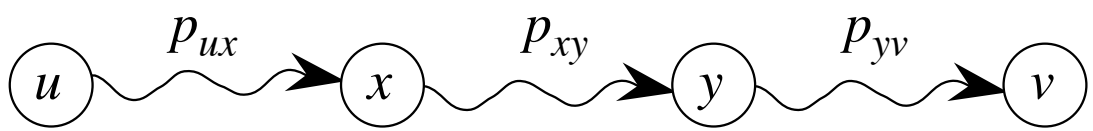


Propriété de sous-structure optimale

- Lemme : Tout sous-chemin d'un chemin le plus court est un chemin le plus court entre ses extrémités

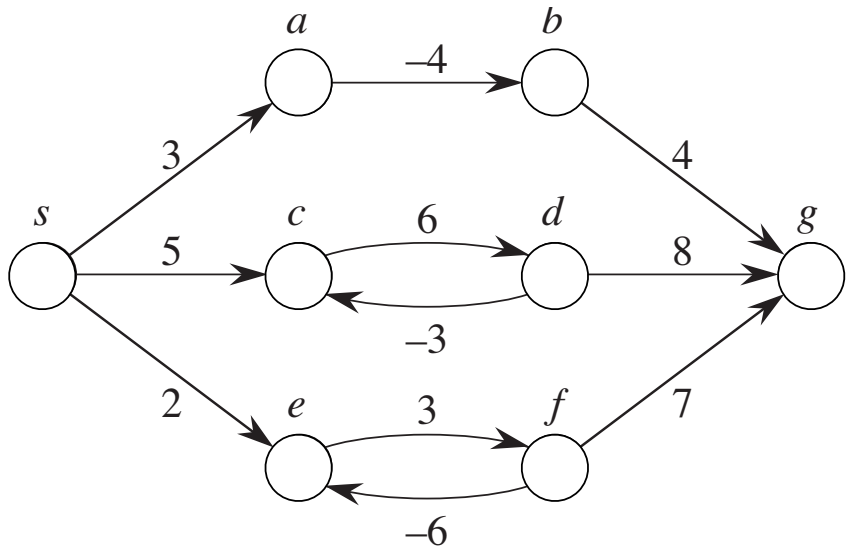
- Preuve : Par l'absurde
  - Soit un plus court chemin p_{uv} entre u et v et soit un sous-chemin p_{xy} de ce chemin défini par ses extrémités x et y
  - S'il existe un chemin plus court que p_{xy} entre x et y, on pourrait remplacer p_{xy} par ce chemin dans le chemin entre u et v et obtenir ainsi un chemin plus court que p_{uv} entre u et v

□



Poids négatifs

- Les poids peuvent être négatifs
- Ok la plupart du temps mais non permis par certains algorithmes (Dijkstra)
- Problème en cas de cycle de poids négatif (cycle absorbant) :
  - En restant dans le cycle négatif, on peut diminuer arbitrairement le poids du chemin
  - Par définition, on fixera $\delta(u, v) = -\infty$ s'il y a un chemin de u à v qui passe par un cycle négatif



$$\delta(s, e) = \delta(s, f) = \delta(s, g) = -\infty$$

Cycles

Un chemin le plus court ne peut pas contenir de cycles

- Cycles de poids négatifs : on les a déjà exclus par définition
- Cycles de poids positifs : on peut obtenir un chemin plus court en les supprimant
- Cycles de poids nuls : il n'y a pas de raison de les utiliser et donc, on ne le fera pas

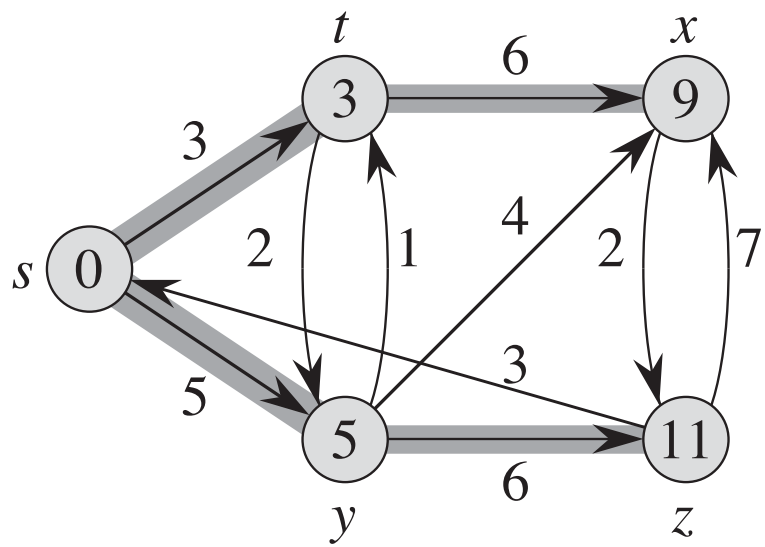
Plus courts chemins : variantes de problèmes

Différentes variantes du problème :

- Origine unique : trouver tous les plus courts chemins d'un sommet à tous les autres
  - Algorithmes de Dijkstra (glouton) et Bellman-Ford (programmation dynamique)
- Destination unique : trouver tous les plus courts chemins de tous les sommets vers un sommet donné
  - Essentiellement le même problème que l'origine unique
- Paire unique : trouver un plus court chemin entre deux sommets donnés.
  - Pas de meilleure solution que de résoudre le problème "origine unique".
- Toutes les paires : trouver tous les plus courts chemins de u à v pour toutes les paires de sommets $(u, v) \in V \times V$.
  - Algorithme de Floyd-Warshall

Recherche du plus court chemin, origine unique

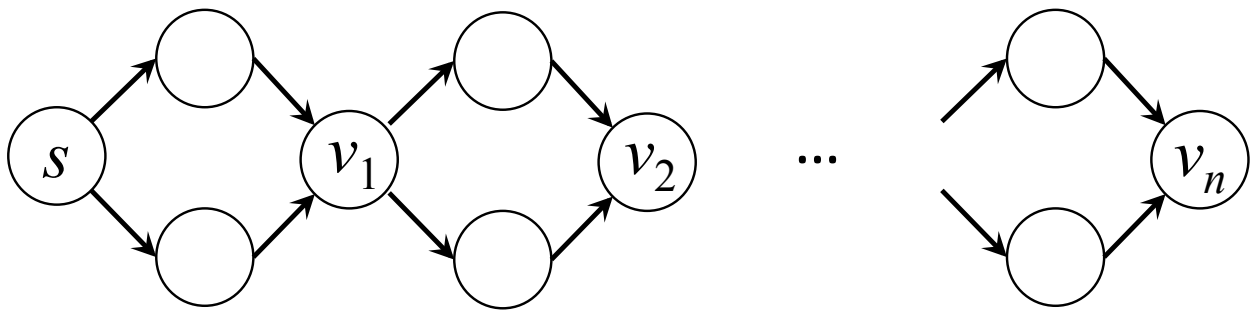
- Entrées : un graphe dirigé pondéré et un sommet s (l'origine)
- Sorties : deux attributs pour chaque sommet :
  - $v.d = \delta(s, v)$, la plus courte distance de s vers chaque nœud
  - $v.\pi =$ le prédécesseur de chaque sommet v dans un plus court chemin de s à v
    (formant l'arbre des plus courts chemins partant de s)



(Chaque nœud v est marqué de la valeur de $\delta(s, v)$)

Approche force brute

- Calculer le poids de tous les chemins entre deux nœuds et renvoyer le plus court
- Problème :
  - Le nombre de chemins peut être infini (dans le cas de cycles)
  - Le nombre de chemins peut être exponentiel par rapport au nombre de sommets et d'arêtes



($O(n)$ nœuds et 2^n chemins entre s et v_n)

Schéma général d'un algorithme

- Objectif : calculer $v.d = \delta(s, v)$ pour tout $v \in V$
- Idée d'algorithme :
  - $v.d$ à une itération donnée contient une estimation du poids d'un plus court chemin de s à v
  - Invariant : $v.d \geq \delta(s, v)$
  - Initialisation : $v.d = +\infty$ $(\forall v \in V)$
  - A chaque itération, on tente d'améliorer (c'est-à-dire diminuer) $v.d$ en maintenant l'invariant
  - A la convergence, on aura $v.d = \delta(s, v)$
  - L'amélioration est basée sur l'utilisation de l'inégalité triangulaire

Inégalité triangulaire et relâchement

- Théorème : Pour tout $u, v, x \in V$, on a

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v)$$

- Preuve : Aller de u à v en empruntant un plus court chemin passant par x ne peut pas être plus court qu'un plus court chemin de u à v.

- Corollaire : Pour tout $(u, v) \in E$, on a

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

- Amélioration d'une arête (Relâchement) :

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
```

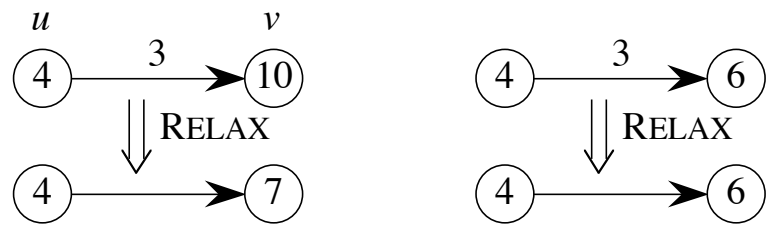


Schéma général d'un algorithme

```
SINGLE-SOURCE-SP(G, w, s)
1  INIT-SINGLE-SOURCE(G, s)
2  while ∃(u, v) : v.d > u.d + w(u, v)
3        Pick one edge (u,v)
4        RELAX(u, v, w)
```

```
INIT-SINGLE-SOURCE(G, s)
1  for each v ∈ G.V
2        v.d = ∞
3  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
```

- On obtient différents algorithmes en modifiant la manière dont on sélectionne les arêtes

Schéma général d'un algorithme

```
SINGLE-SOURCE-SP(G, w, s)
1  INIT-SINGLE-SOURCE(G, s)
2  while ∃(u, v) : v.d ≥ u.d + w(u, v)
3        Pick one edge (u,v)
4        RELAX(u, v, w)
```

```
INIT-SINGLE-SOURCE(G, s)
1  for each v ∈ G.V
2        v.d = ∞
3        v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```

- En ajoutant la construction de l'arbre des plus courts chemins

Propriétés de l'algorithme général

- Propriété 1 : L'algorithme général maintient toujours l'invariant

- Preuve : Par induction sur le nombre d'itérations
  - Cas de base : l'invariant est vérifié après l'initialisation
  - Cas inductif :
    - Soit un appel à $relax(u, v, w)$
    - Avant l'appel, on suppose que l'invariant est vérifié et donc $u.d \geq \delta(s, u)$
    - Par l'inégalité triangulaire :

$$\begin{aligned} \delta(s, v) &\leq \delta(s, u) + \delta(u, v) \\ &\leq u.d + w(u, v) \end{aligned}$$

    - Suite à l'assignation $v.d = u.d + w(u, v)$, on a bien

$$v.d \geq \delta(s, v)$$

Propriétés de l'algorithme général

- Propriété 2 : Une fois que $v.d = \delta(s, v)$, il n'est plus modifié
- Preuve : On a toujours $v.d \geq \delta(s, v)$ et un relâchement ne peut que diminuer $v.d$

- Vu les propriétés 1 et 2, pour montrer qu'un algorithme du plus court chemin est correct, on devra montrer que le choix des arêtes à relâcher mènera bien à $v.d = \delta(s, v)$ pour tout v.

Algorithme de Bellman-Ford

```
SINGLE-SOURCE-SP(G, w, s)
1  INIT-SINGLE-SOURCE(G, s)
2  while ∃(u, v) : v.d ≥ u.d + w(u, v)
3        Pick one edge (u,v)
4        RELAX(u, v, w)
```

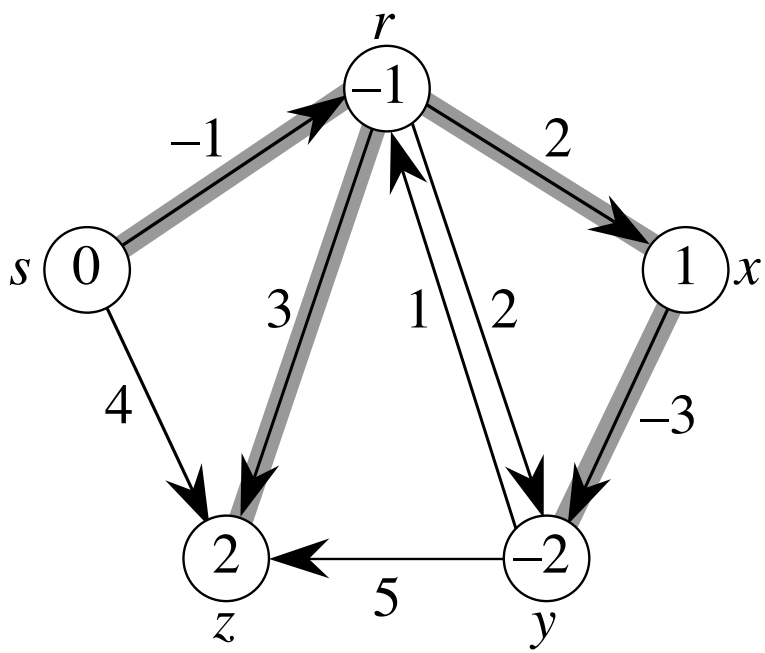
- Algorithme basé sur le relâchement
- Soit les arêtes $e_1, \ldots, e_m$, dans un ordre quelconque.
- Le relâchement se fait dans cet ordre :

$$\underbrace{\underbrace{e_1, e_2, \ldots, e_m}; \underbrace{e_1, e_2, \ldots, e_m}; \ldots; \underbrace{e_1, e_2, \ldots, e_m}}_{|V|-1 fois}$$

Algorithme de Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1  INIT-SINGLE-SOURCE(G, s)
2  for i=1 to |G.V| − 1
3        for each edge (u, v) ∈ G.E
4              RELAX(u, v, w)
```

Illustration sur un exemple :



Analyse : complexité

```
BELLMAN-FORD(G, w, s)
1  INIT-SINGLE-SOURCE(G, s)
2  for i=1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
```

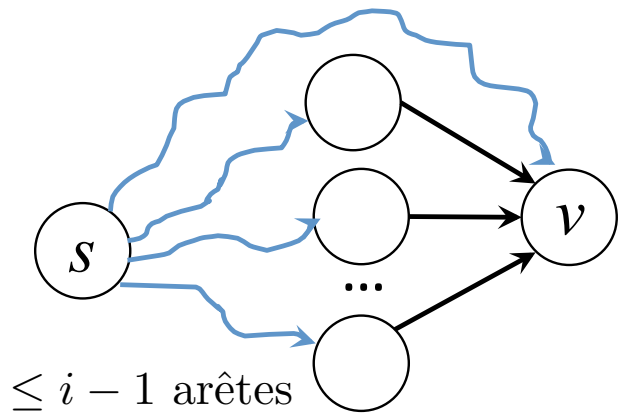
- La boucle principale relâche toutes les arêtes $|V| - 1$ fois
- Complexité : $\Theta(|V| \cdot |E|)$
  - En supposant qu'on puisse parcourir les arêtes en $O(|E|)$

Analyse : correction

- On supposera qu'il n'y a pas de cycle de poids négatif
- Propriété 3 : Après i itérations de l'algorithme, $v.d$ est le poids d'un plus court chemin de s à v utilisant au plus i arêtes :

$$v.d \leq \min\{w(p) : |p| \leq i\}$$

- Preuve : Par induction :
  - Cas de base : $v.d = +\infty$ si $i = 0$ et $s.d = 0$
  - Cas inductif :
    - Avant l'itération i, on a $v.d \leq \min\{w(p) : |p| \leq i - 1\}$
    - Cette propriété reste vraie à tout moment de l'itération puisque RELAX ne peut que diminuer les $v.d$
    - L'itération i considère tous les chemins avec i arêtes ou moins en relâchant toutes les arêtes entrantes en v



Analyse : correction

- Si le graphe ne contient pas de cycles de poids négatif, alors, à la fin de l'exécution de l'algorithme de Bellman-Ford, on a $v.d = \delta(s, v)$ pour tout $v \in V$.

- Preuve :
  - Sans cycle négatif, tout plus court chemin est simple, c'est-à-dire sans cycle
  - Tout chemin simple a au plus $|V|$ sommets et donc $|V| - 1$ arêtes
  - Par la propriété 3, on a $v.d \leq \delta(s, v)$ après $|V| - 1$ itérations
  - Par l'invariant, on a $v.d \geq \delta(s, v) \Rightarrow v.d = \delta(s, v)$

□

Programmation dynamique

- L'algorithme de Bellman-Ford implémente en fait une approche par programmation dynamique [5]
- Soit $v.d[i]$, la longueur du plus court chemin de s à v utilisant au plus i arêtes
- On a

$$v.d[i] = \begin{cases} 0 \text{ si } v = s \text{ et } i = 0 \\ +\infty \text{ si } v \neq s \text{ et } i = 0 \\ \min\{v.d[i-1], \min_{(u,v)\in E}\{u.d[i-1] + w(u,v)\}\} \text{ sinon} \end{cases}$$

(Exercice : implémenter l'algorithme de Bellman-Ford à partir de la récurrence et le comparer avec la version précédente)

5. Bellman est en fait l'inventeur de la programmation dynamique

Détection des cycles négatifs

```
BELLMAN-FORD(G, w, s)
1  INIT-SINGLE-SOURCE(G, s)
2  for i=1 to |G.V| − 1
3        for each edge (u, v) ∈ G.E
4              RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6        if v.d > u.d + w(u, v)
7              return TRUE
8  return FALSE
```

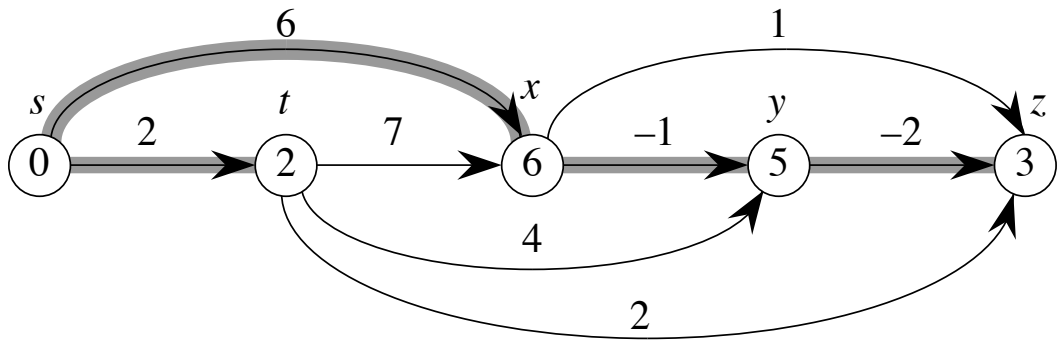
- Renvoie TRUE si un cycle négatif (accessible depuis s) existe, FALSE sinon
- En cas de cycle négatif, il existe toujours (et donc aussi en sortie de boucle) au moins un $v.d$ qu'on peut améliorer par relâchement d'un arc (u, v)

Graphe dirigé acyclique (DAG)

Version plus efficace dans le cas d'un graphe dirigé acyclique :

```
DAG-Shortest-Path(G, w, s)
1  L = Top-Sort(G)
2  Init-single-source(G, s)
3  for each vertex u, taken in their order in L
4       for each vertex v ∈ G.Adj[u]
5            Relax(u, v, w)
```

Exemple :



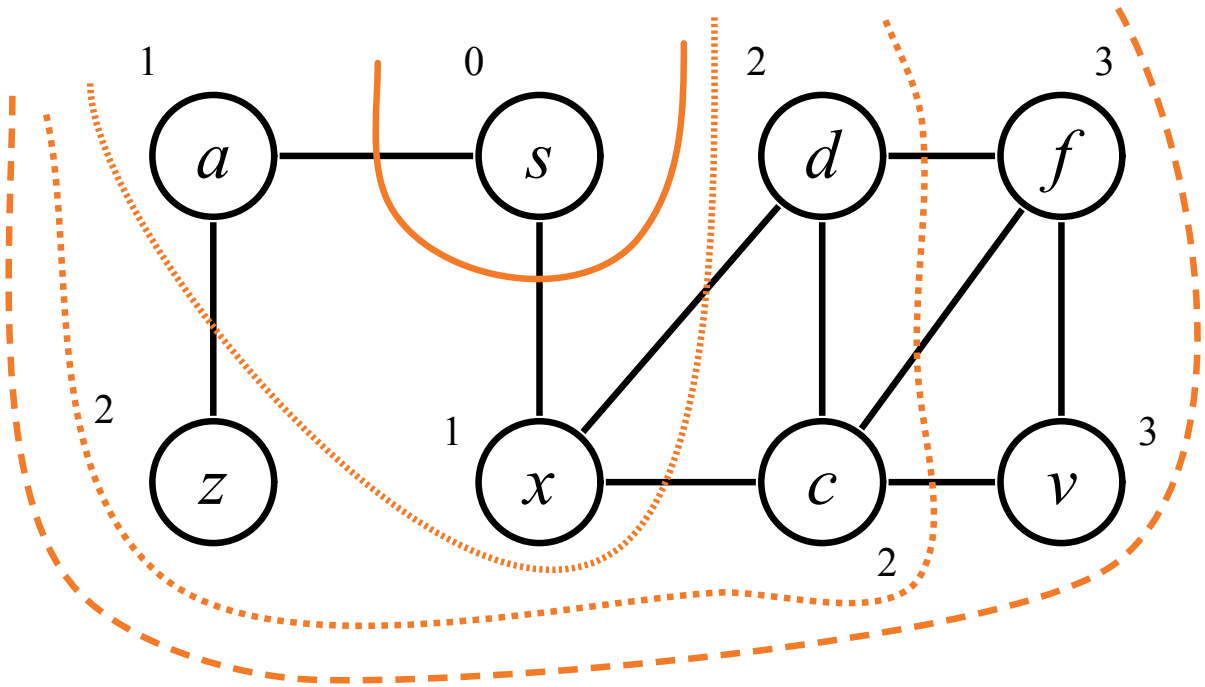
Complexité : $\Theta(|V| + |E|)$

Poids unitaires : parcours en largeur d'abord

- On peut obtenir une solution plus rapide en imposant certaines contraintes sur la nature des poids
- Si les poids sont tous égaux à 1, le parcours en largeur permet de calculer les $v.d$ en $O(|V| + |E|)$
- Rappel :

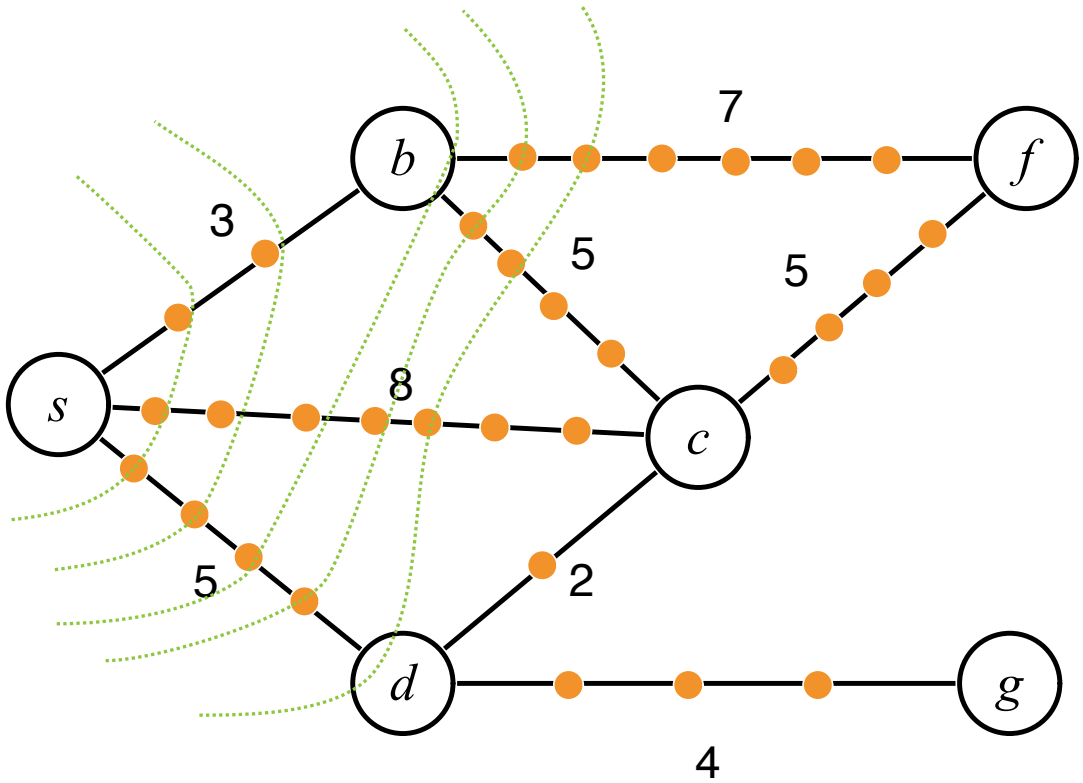
```
BFS(G, s)
 1  for each vertex u ∈ G.V \ {s}
 2        u.d = ∞
 3  s.d = 0
 4  Q ="create empty Queue"
 5  ENQUEUE(Q, s)
 6  while not QUEUE-EMPTY(Q)
 7        u = DEQUEUE(Q)
 8        for each v ∈ G.Adj[u]
 9              if v.d = ∞
10                    v.d = u.d + 1
11                    ENQUEUE(Q, v)
```

Poids unitaires : parcours en largeur d'abord



Poids entiers, positifs et bornés

- Si les poids sont des entiers compris entre $1 \dots W$:
  - On définit un nouveau graphe en éclatant chaque arête de poids w en w arêtes de poids 1
  - On applique le parcours en largeur sur ce nouveau graphe
- Complexité : $O(|V| + W|E|)$



Poids positifs : approche gloutonne

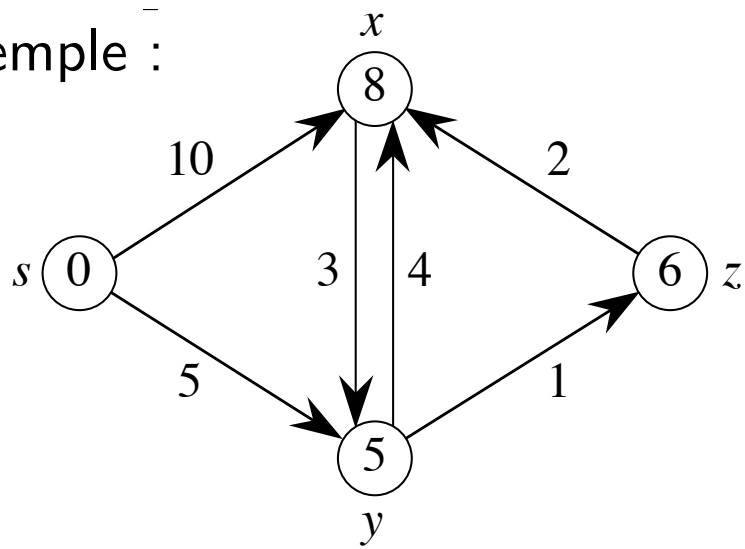
- Algorithme de Dijkstra : généralisation du parcours en largeur à des poids positifs réels

- Idée :
  - On maintient un ensemble S de sommets dont le poids d'un plus court chemin à partir de s est connu
  - A chaque étape, on ajoute à S un sommet $v \in V \setminus S$ dont la distance à s est minimale
  - On met à jour, par relâchement, les distances estimées des sommets adjacents à v

Algorithme de Dijkstra

```
DIJKSTRA(G, w, s)
1  INIT-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q ="create an empty min priority queue from G.V"
4  while not EMPTY(Q)
5        u = EXTRACT-MIN(Q)
6        S = S ∪ {u}
7        for each v ∈ G.Adj[u]
8              RELAX(u, v, w) // ! RELAX doit modifier la clé de v dans Q
```

Illustration sur un exemple :



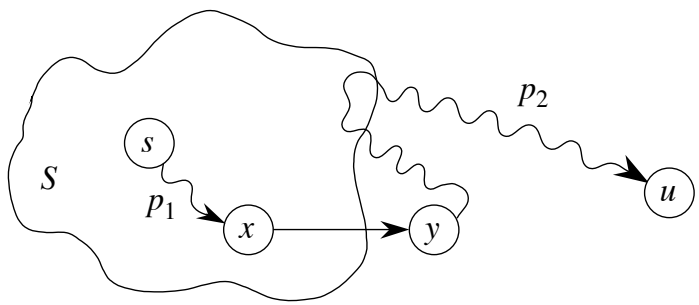
Analyse : complexité

- Si la file à priorité est implémentée par un tas (min), l'extraction et l'ajustement de la clé sont $O(\log |V|)$
- Chaque sommet est extrait de la file à priorité une et une seule fois
  $\Rightarrow$ $O(|V| \log |V|)$
- Chaque arête est parcourue une et une seule fois et entraîne au plus un ajustement [6] de clé
  $\Rightarrow$ $O(|E| \log |V|)$
- Total : $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$
  - $|E| \log |V|$ domine $|V| \log |V|$ si le graphe est connexe

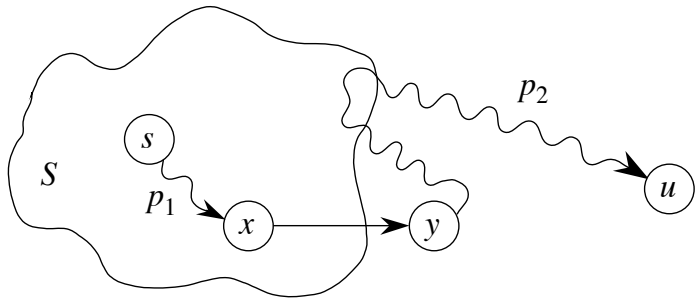
6. Similaire à un HEAP-DECREASE-KEY

Analyse : correction

- Théorème : l'algorithme de Dijkstra se termine avec $v.d = \delta(s, v)$ pour tout $v \in V$
- Preuve :
  - Lorsqu'un nœud v est extrait de la file, son $v.d$ n'est plus modifié. Il suffit donc de montrer que $v.d = \delta(s, v)$ lorsque v est extrait de Q
  - Par l'invariant (propriété 1), on a $v.d \geq \delta(s, v)$ à tout moment
  - Par l'absurde, supposons qu'il existe un nœud u tel que $u.d > \delta(s, u)$ lors de son extraction et soit u le premier nœud satisfaisant cette propriété.
  - Soit y le premier nœud d'un plus court chemin de s à u qui se trouve dans Q avant l'extraction de u et soit x son prédécesseur



Analyse : correction

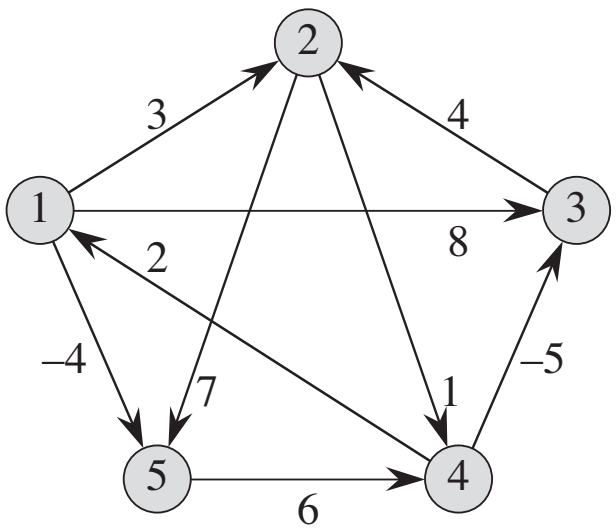


- Puisque u est le premier nœud violant l'invariant, on a $x.d = \delta(s, x)$
- Par la propriété de sous-structure optimale, le sous-chemin de s à y est un plus court chemin et $y.d$ a été assigné à $x.d + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y)$ lors de l'extraction de x
- On a donc $y.d = \delta(s, y) \leq \delta(s, u) \leq u.d$
- Or, $y.d \geq u.d$ puisqu'on s'apprête à extraire u de la file
- D'où $y.d = \delta(s, y) = \delta(s, u) = u.d$, ce qui contredit notre hypothèse □

Plus court chemin pour toutes les paires de sommets

Déterminer les plus courts chemins pour toutes les paires de sommets :

- Entrées : un graphe dirigé $G = (V, E)$, une fonction de poids w. Les sommets sont numérotés de 1 à n
- Sortie : une matrice $D = (d_{ij})$ de taille $n \times n$ où $d_{ij} = \delta(i, j)$ pour tous sommets i et j



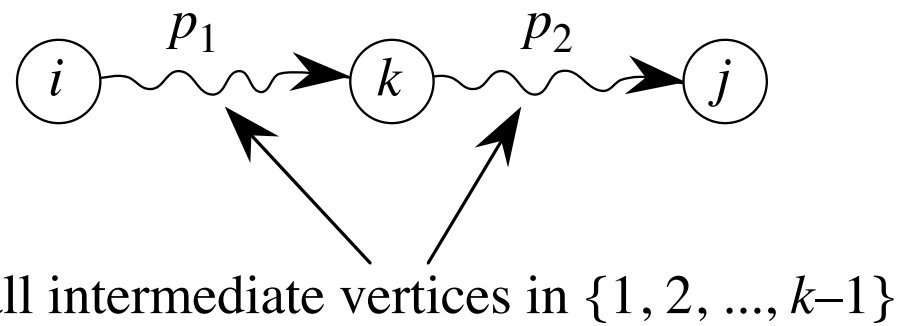
$$D = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Plus court chemin pour toutes les paires de sommets

- Dans le cas général, on peut appliquer Bellman-Ford sur chaque sommet
  - $O(|V|^2|E|)$, ou $O(V^4)$ si le graphe est dense ($E = \Theta(V^2)$)
- S'il n'y a pas de poids négatifs, on peut appliquer Dijkstra sur chaque sommet
  - $O(|V||E| \log |V|)$, ou $O(V^3 \log |V|)$ si le graphe est dense
- Il est possible d'obtenir $O(V^3)$ par programmation dynamique

Une solution par programmation dynamique

- Pour un chemin $p = \langle v_1, v_2, \ldots, v_l \rangle$, un sommet intermédiaire est un sommet de p autre que v_1 ou v_l
- Soit $d_{ij}^{(k)}$ le poids d'un plus court chemin entre i et j tel que tous les sommets intermédiaires sont dans le sous-ensemble de sommets $\{1, 2, \ldots, k\}$
- Soit un plus court chemin p entre i et j avec tous les sommets dans $\{1, 2, \ldots, k\}$:
  - Si k n'est pas un sommet intermédiaire de p, alors tous les sommets intermédiaires de p sont dans $\{1, 2, \ldots, k-1\}$
  - Si k est un sommet intermédiaire, tous les sommets intermédiaires des sous-chemins entre i et k et entre k et j appartiennent à $\{1, 2, \ldots, k-1\}$



Algorithme de Floyd-Warshall

- Formulation récursive :

$$d_{ij}^{(k)} = \begin{cases} w(i,j) & \text{si } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{si } k \geq 1. \end{cases}$$

- Implémentation ascendante : $\Theta(|V|^3)$
  - $W = (w_{ij})$ est la matrice d'adjacence pondérée
  - $w_{ij} = w(i,j)$ si $(i,j) \in E$, $+\infty$ sinon

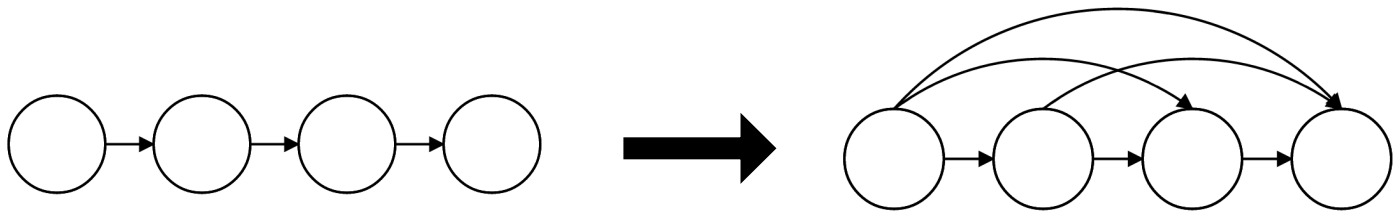
```
FLOYD-WARSHALL(W, n)
1  D^(0) = W
2  for k = 1 to n
3        let D^(k) = (d_ij^(k)) be a new n × n matrix
4        for i = 1 to n
5              for j = 1 to n
6                    d_ij^(k) = MIN(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7  return D^(n)
```

Fermeture transitive d'un graphe

- Soit un graphe dirigé $G = (V, E)$. La fermeture transitive de G est le graphe $G^* = (V, E^*)$ tel que :

$$E^* = \{(i, j) : \exists \text{ un chemin de } i \text{ à } j \text{ dans } G\}$$

- Exemple :



- Solution directe :
  - Assigner un poids $w_{ij} = 1$ à toute arête $(i, j) \in E$
  - Appliquer l'algorithme de Floyd-Warshall
  - Si $d_{ij} < \infty$, il y a un chemin entre i et j dans G
  - Sinon, $d_{ij} = \infty$ et il n'y a pas de chemin

Fermeture transitive d'un graphe

Une solution plus simple en modifiant l'algorithme de Floyd-Warshall :

- Soit $t_{ij}^{(k)} = 1$ s'il y a un chemin de i à j avec tous les nœuds intermédiaires dans $\{1, 2, \ldots, k\}$, 0 sinon
- On a :

$$t_{ij}^{(k)} = \begin{cases} 0 & \text{si } k = 0, i \neq j \text{ et } (i,j) \notin E \\ 1 & \text{si } k = 0 \text{ et } i = j \text{ ou } (i,j) \in E \\ t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}) & \text{si } k \geq 1. \end{cases}$$

- Même implémentation que Floyd-Warshall
  - on remplace min par $\vee$ et $+$ par $\wedge$

Fermeture transitive d'un graphe : algorithme

TRANSITIVE-CLOSURE(G, n)

```
 1  Let T^(0) = (t_ij^(0)) be a new n × n matrix
 2  for i = 1 to n
 3      for j = 1 to n
 4          if i == j or (i, j) ∈ G.E
 5              t_ij^(0) = 1
 6          else t_ij^(0) = 0
 7  for k = 1 to n
 8      let T^k = (t_ij^(k)) be a new n × n matrix
 9      for i = 1 to n
10          for j = 1 to n
11              t_ij^(k) = t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
12  return D^(n)
```

- Complexité : $\Theta(|V|^3)$
  - Idem Floyd-Warshall mais opérations plus simples

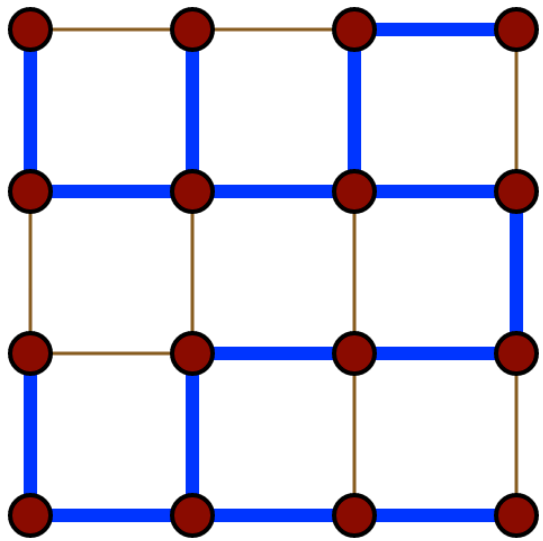
Plus court chemin : synthèse

- Origine unique, graphe dirigé acyclique :
  - Relâchement en suivant un ordre topologique
  - $\Theta(|V| + |E|)$
- Origine unique, graphe dirigé, poids positifs :
  - Algorithme de Dijkstra
  - $\Theta(|E| \log |V|)$
- Origine unique, graphe dirigé, poids quelconques :
  - Algorithme de Bellman-Ford
  - $\Theta(|V| \cdot |E|)$
- Toutes les paires, graphe dirigé ou non :
  - Algorithme de Floyd-Warshall
  - $\Theta(|V|^3)$

Plan

Arbre couvrant

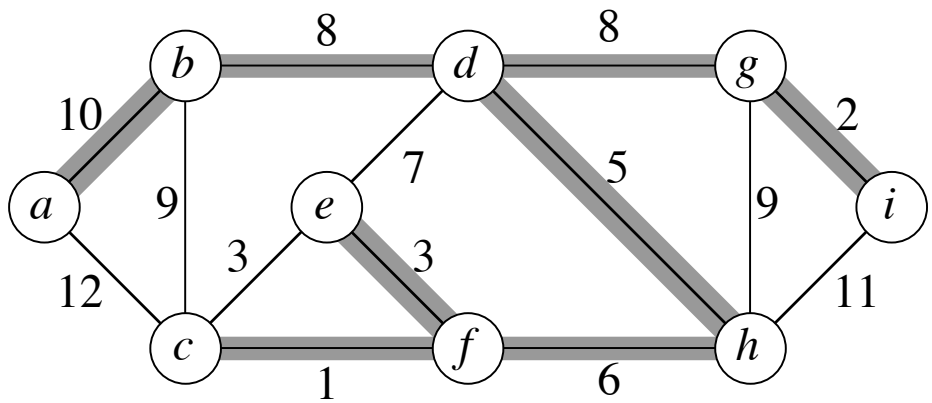
- Définition : un arbre couvrant *(spanning tree)* pour un graphe connexe (V, E) non dirigé est un arbre (i.e. un graphe acyclique) T tel que :
  - l'ensemble des nœuds de T est égal à V, et
  - l'ensemble des arcs de T est un sous-ensemble de E
- Construction : par un parcours en largeur ou en profondeur (graphe de liaison)
- Exemple



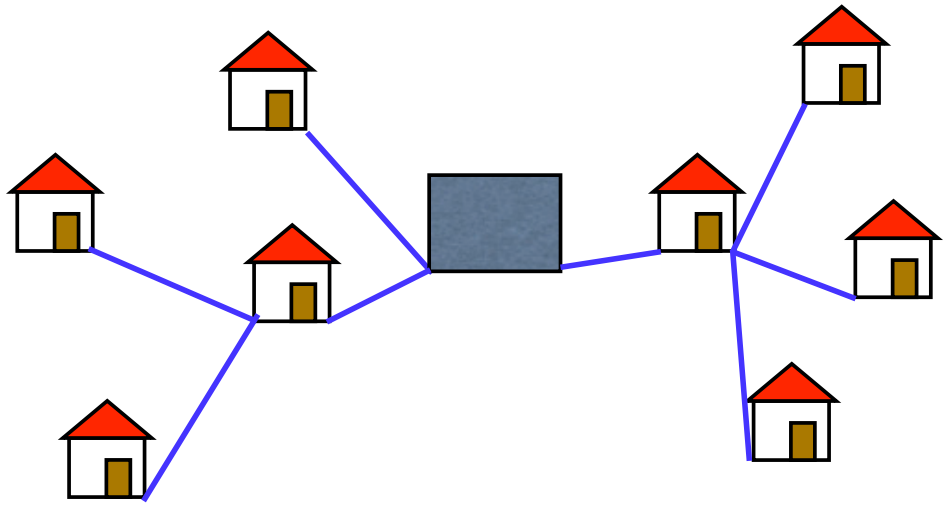
Arbre couvrant de poids minimum

- Définition : un arbre couvrant de poids minimum, ACM, *(minimum spanning tree, MST)* pour un graphe pondéré connexe (V, E) est un arbre (V, E') tel que :
  - (V, E') est un arbre couvrant de (V, E), et
  - la valeur de $\sum_{e \in E'} w(e)$ est minimale parmi tous les arbres couvrants de (V, E), où $w(e)$ dénote le poids de l'arc e

- Exemple :



Applications



- Conception de réseaux : connecter des entités en minimisant le coût de la connection
  - Raccorder des maisons à un central téléphonique en minimisant les longueurs de cables
  - Elaborer un système routier pour connecter des maisons
  - ...
- Dissémination de contenu/routage sur internet
- Design de circuits imprimés
- ...

Approche générique

- Idée :
  - Un ACM est un sous-ensemble d'arêtes du graphe initial
  - On démarre avec un ensemble d'arêtes $A = \emptyset$ vide
  - On ajoute dans A des arêtes en respectant l'invariant suivant :
    - Il existe un ACM qui contient les arêtes de A
  - S'il existe un ACM contenant les arêtes de A, une arête (u, v) est sûre pour A ssi il existe un ACM contenant les arêtes de $A \cup \{(u, v)\}$

- Algorithme générique :

```
GENERIC-MST(G, w)
1  A = ∅
2  while A is not a spanning tree
3        find an edge (u, v) that is safe for A
4        A = A ∪ {(u, v)}
5  return A
```

- Comment trouver des arêtes sûres ?

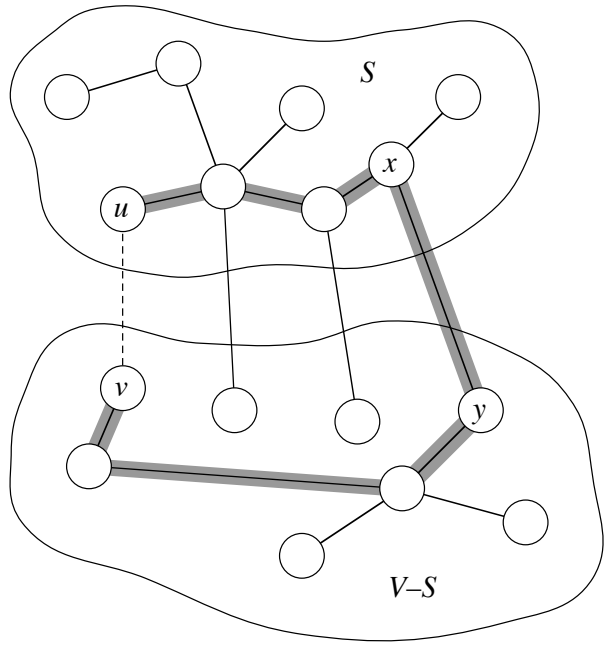
Arêtes sûres

- Soit $S \subset V$ et $A \subseteq E$:
  - Une coupure (*cut*) $(S, V \setminus S)$ est une partition des sommets en deux ensembles disjoints S et $V \setminus S$
  - Une arête traverse (*crosses*) une coupure $(S, V \setminus S)$ si une extrémité est dans S et l'autre dans $V \setminus S$
  - Une coupure respecte A ssi il n'y a pas d'arête dans A qui traverse la coupure

- Théorème : Soit A un sous-ensemble d'un ACM, $(S, V \setminus S)$ une coupure qui respecte A et (u, v) une arête de poids minimal qui traverse la coupure $(S, V \setminus S)$. (u, v) est sûre pour A.

  (Propriété des choix gloutons optimaux)

Arêtes sûres

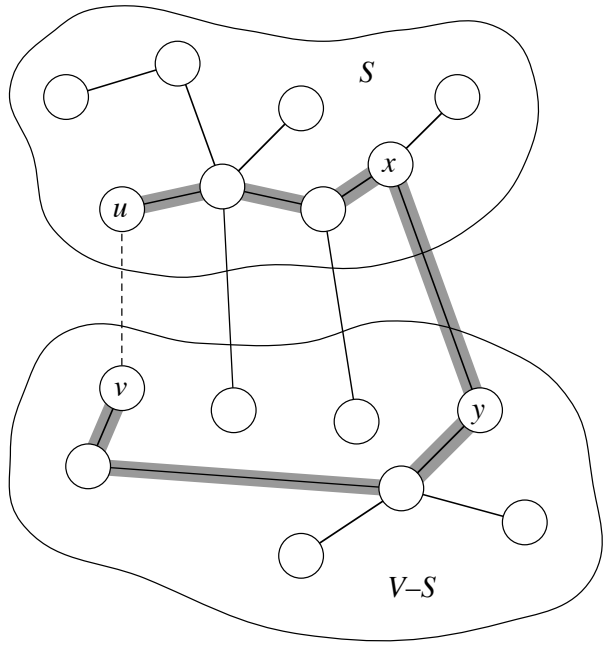


Preuve :

- Soit T un ACM qui inclut A
- Supposons que T ne contienne pas (u, v) et montrons qu'il est possible de construire un arbre T' qui inclut $A \cup \{(u, v)\}$
- Puisque T est un arbre, il n'y a qu'un unique chemin p entre u et v et ce chemin traverse la coupure $(S, V \setminus S)$.
- Soit (x, y) une arête de p qui traverse la coupure $(S, V \setminus S)$
- Puisque (u, v) est l'arête de poids minimum qui traverse la coupure, on a :

$$w(u, v) \leq w(x, y)$$

Arêtes sûres



- Puisque la coupure respecte A, l'arête (x, y) n'est pas dans A
- Soit $T' = (T \setminus \{(x, y)\}) \cup \{(u, v)\}$:
  - T' est un spanning tree
  - $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$ puisque $w(u, v) \leq w(x, y)$
- T' est donc bien un ACM tel que $A \cup \{(u, v)\} \subseteq T'$
- $\Rightarrow (u, v)$ est sûre pour A

Algorithme de Kruskal

- Approche gloutonne :
  - On construit incrémentalement une forêt (c'est-à-dire, un ensemble d'arbres), en ajoutant progressivement des arêtes à un graphe initialement dépourvu d'arcs
  - On maintient en permanence une partition du graphe en cours de construction en ses composantes connexes
  - Pour relier des composantes connexes, on choisit à chaque fois l'arête de poids minimal qui les connecte
  - On s'arrête dès qu'il ne reste plus qu'une composante connexe

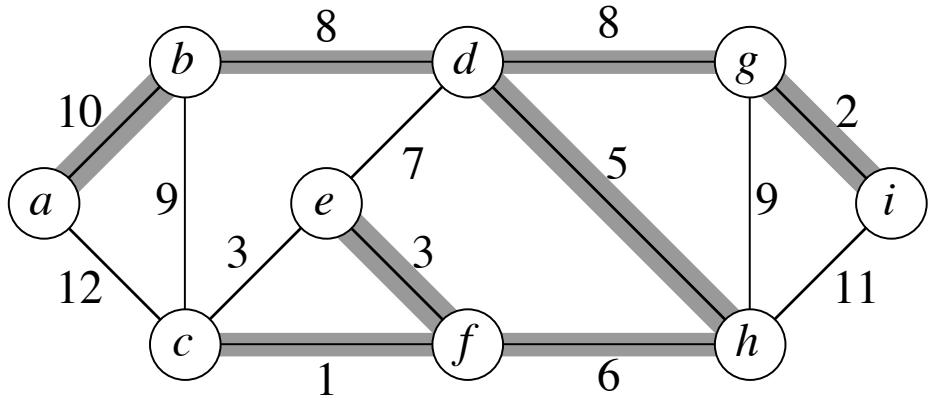
- Correction :
  - Puisqu'on connecte à chaque fois deux composantes connexes disjointes, le graphe reste acyclique et à la terminaison, on obtient un arbre couvrant
  - Puisqu'on sélectionne une arête de poids minimal à chaque étape, le théorème précédent garantit qu'on arrivera à un ACM

Algorithme de Kruskal

```
KRUSKAL(G, w)
 1  A = ∅
 2  P = ∅
 3  for each vertex v ∈ G.V
 4        P = P ∪ {{v}}
 5  for each (u, v) ∈ G.E taken into nondecreasing order of weight w
 6        P_1 = subset in P containing u
 7        P_2 = subset in P containing v
 8        if P_1 ≠ P_2
 9              A = A ∪ {(u, v)}
10              Merge P_1 and P_2 in P
11  return A
```

- Les choix des composantes connexes à combiner est arbitraire
- On fixe cet ordre en parcourant les arêtes par ordre croissant

Illustration



		P
		$\{\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g\},\{h\},\{i\}\}$
(c,f)	fusion	$\{\{a\},\{b\},\{c,f\},\{d\},\{e\},\{g\},\{h\},\{i\}\}$
(g,i)	fusion	$\{\{a\},\{b\},\{c,f\},\{d\},\{e\},\{g,i\},\{h\}\}$
(e,f)	fusion	$\{\{a\},\{b\},\{c,f,e\},\{d\},\{g,i\},\{h\}\}$
(c,e)	rejet	
(d,h)	fusion	$\{\{a\},\{b\},\{c,f,e\},\{d,h\},\{g,i\}\}$
(f,h)	fusion	$\{\{a\},\{b\},\{c,f,e,d,h\},\{g,i\}\}$
(e,d)	rejet	
(b,d)	fusion	$\{\{a\},\{b,c,f,e,d,h\},\{g,i\}\}$
(d,g)	fusion	$\{\{a\},\{b,c,f,e,d,h,g,i\}\}$
(b,c)	rejet	
(g,h)	rejet	
(a,b)	fusion	$\{\{a,b,c,f,e,d,h,g,i\}\}$

Implémentation

- Problème : comment représenter les partitions de l'ensemble des sommets du graphe ?
- Une solution possible :
  - On numérote les parties P_1, P_2,..., P_k d'une partition $\{P_1, P_2, \ldots, P_k\}$ à l'aide des nombres de 1 à k
  - Pour chaque sommet v, on retient le numéro de la partie à laquelle il appartient (attribut $v.p$)
  - Pour chaque numéro de partie, on retient une liste des sommets contenus dans cette partie
  - Lors de la fusion de deux parties, on insère la plus petite partie à fusionner dans l'autre et on met à jour les numéros de partie
- Complexité :
  - Trouver la partie associée à un sommet : $O(1)$
  - Fusionner deux parties de tailles n_1 et n_2, avec $n_1 < n_2$: $\Theta(n_1)$

Complexité

- Initialisation : $O(|V|)$
- Tri des arêtes : $O(|E| \log |V|)$
  - Tri : $O(|E| \log |E|)$
  - Or, $|E| < |V|^2 \Rightarrow \log |E| = O(2 \log |V|) = O(\log |V|)$
- Coût total des fusions : $O(|V| \log |V|)$
  - Chaque fusion est linéaire par rapport à la taille de la plus petite partie
  - Chaque fusion produit un nouvelle partie au moins deux fois plus grande que la plus petite
  - Chaque sommet n'est ajouté à une partie qu'au plus $O(\log |V|)$ fois
- Temps d'exécution total :
  $O(|E| \log |V| + |V| \log |V|) = O(|E| \log |V|)$
  - Car $|E|$ domine $|V|$ dans le cas d'un graphe connexe

Algorithme de Prim

- Principe :
  - A est toujours un arbre (plus une forêt)
  - Initialisé comme une seule racine r choisie de manière arbitraire
  - A chaque étape, choisir une arête de poids minimal traversant la coupure $(V_A, V \setminus V_A)$, où V_A est l'ensemble des sommets connectés par des arêtes de A, et l'ajouter à A.

```
PRIM(G, w, r)
1  A = ∅
2  V_A = {r}
3  while |V_A| < |G.V|
4        (u, v) =”an edge of minimal weight from V_A to V \ V_A”
5        V_A = V_A ∪ {u, v}
6        A = A ∪ {(u, v)}
7  return A
```

- Correction : toujours en application du théorème

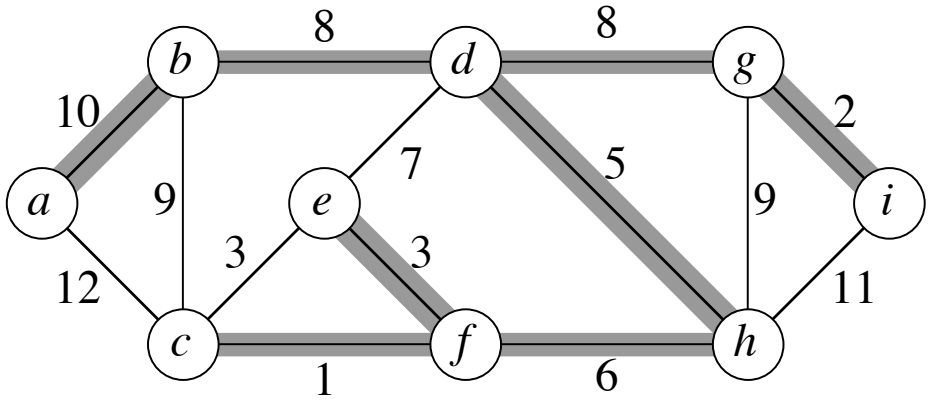
Implémentation

- Comment extraire efficacement l'arête de poids minimal ?
- Utiliser une file à priorité :
  - Chaque élément de la file est un sommet de $V \setminus V_A$ (pas encore couvert par l'arbre courant)
  - La clé de v est le poids minimum de tout arête (u, v) où $u \in V_A$
  - Cette clé est mise à jour à chaque ajout d'un sommet dans V_A
- L'arbre est "stocké" par le biais d'un pointeur $v.\pi$
  - $v.\pi$ est le parent de v dans l'arbre couvrant minimal
  - $v.\pi = \mathrm{NIL}$ si $v = r$ ou v n'a pas de parents

Implémentation

```
PRIM(G, w, r)
 1  Q = ∅
 2  for each u ∈ G.V
 3        u.key = ∞
 4        u.π = NIL
 5        INSERT(Q, u)
 6  DECREASE-KEY(Q, r, 0) // r.key = 0
 7  while Q ≠ ∅
 8        u = EXTRACT-MIN(Q)
 9        for each v ∈ G.Adj[u]
10              if v ∈ Q and w(u, v) < v.key
11                    v.π = u
12                    DECREASE-KEY(Q, v, w(u, v))
```

Illustration



A partir du nœud d

Nœud	Q^0	Q^1	Q^2	Q^3	Q^4	Q^5	Q^6	Q^7	Q^8
a	∞	∞	∞	∞	12	12	10	10	10
b	∞	8	8	8	8	8			
c	∞	∞	∞	1					
d	0								
e	∞	7	7	3	3				
f	∞	∞	6						
g	∞	8	8	8	8	8			
h	∞	5							
i	∞	∞	11	11	11	11	2		

(valeurs de clé des sommets dans Q au fur et à mesure des itérations)

Complexité

- En supposant que Q est implémentée à l'aide d'un tas (min)
- Initialisation et première boucle **for** : $O(|V| \log |V|)$
- Diminuer la clé de r : $O(\log |V|)$
- Boucle **while** : $O(|V| \log |V| + |E| \log |V|)$
  - $|V|$ appels à EXTRACT-MIN $\Rightarrow O(|V| \log |V|)$
  - $|E|$ appels à DECREASE-KEY $\Rightarrow O(|E| \log |V|)$
- Temps d'exécution total :
  $O(|E| \log |V| + |V| \log |V|) = O(|E| \log |V|)$
  - Car $|E|$ domine $|V|$ dans le cas d'un graphe connexe

Fin

Pour aller plus loin :