Algorithmique avancée

Cours 1

## Objectif de la matière :

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

## Structure de la matière

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

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

# Le rôle des algorithmes en informatique

## Algorithme

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

## Algorithmes en tant que technologie

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

# Analyse des algorithmes

## Exemple : le problème du tri

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

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

* + De sorte que

a1’ < a2’ < … < an’

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

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

## Un 1er algorithme : le tri par insertion

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

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

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

## Correction d’un algorithme

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

## Invariant de boucle

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

## Validité du tri par insertion

* Invariant de boucle

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

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

## Analyse des algorithmes

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

## Modèle RAM

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

## Analyse du tri par insertion

### Durée d'exécution de TRI-INSERTION

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

### Taille de l'entrée

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

## Temps d'exécution

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

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

(slide 21)

## Analyse du temps d'exécution

### Ce qui nous intéresse généralement le plus

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

### Ce qui nous intéresse vraiment

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

# Conception des algorithmes

## Techniques de conception

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

## Exemple : le tri par fusion

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

**Schéma a**

## La procédure FUSION

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

## La procédure TRI-FUSION

(slide 27)

**Schéma b**

* Exemple 3

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

* Analyse ?

## Analyse des algorithmes diviser-pour-régner

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

T(n)= {

{ sinon

## Analyse du tri par FUSION

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

T(n)= { si n=1

{ si n>1

## Arbre récursif

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

-temps par élément de tableau des étapes diviser et combiner

Slide 30

## Arbre récursif

Slide 31

# Croissance des fonctions

## Ordre de grandeur du temps d'exécution

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

## Notation asymptotique

Slide 34

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

Slide 35

Cours 2 : notions de base et représentation

# Plan de la séance

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