1.1

Algorithmique et Structures de Données

Jean-Charles Régin

Licence Informatique 2ème année

1.2

Partie I : Introduction et tableaux

Jean-Charles Régin

Licence Informatique 2ème année

Remerciements

- Carine Fédèle
- Wikipedia

Plan du cours

- Séances 1, 2, 3, 4 introduction + tableaux + structures de données basées sur les tableaux (tas, matrice…)
  - tri par insertion, par tas, comptage
  - multiplication de matrice
  - intro au type abstrait par exemple ensemble (implémenté par tableaux)
- Séance 5 : interrogation 1 d'1h30
- Séances 6 : file/pile
- Séances 7 : listes
- Séances 8 : arbres
- Séance 9 : interrogation 2 d'1h30

Contrôle des connaissances

- interrogation 1 (40%)
- interrogation 2 (40%)
- projet en TP (20%)

Références

- T. Cormen, C. Leiserson, R. Rivest : “Introduction à l’algorithmique”, Dunod
- D. Knuth : “The Art of Computer Programming”
- R. Tarjan : “Data Structures and Network Algorithms”
- R. Ahuja, T. Magnanti et J. Orlin : “Network Flows”, Prentice Hall
- C. Berge : “Graphes et hypergraphes”, Dunod
- M. Gondran et M. Minoux : “Graphes et Algorithmes”
- Autres livres selon votre goût: ne pas hésiter à en consulter plusieurs

Plan

- Algorithmes
- Complexité
- Preuve
- Structures de données
- Pseudo Langage
- Tableaux
- Tri par insertion
- Calcul de x^n
- Logarithme
- Recherche dichotomique

Algorithmes classiques

- Chemin dans un graphe : votre GPS, votre téléphone IP, tout ce qui est acheminement (la Poste)
- Flots : affectations, emploi du temps, bagages (aéroport), portes d'embarquement
- Scheduling (ordonnancement) fabrication de matériel dans les usines, construction automobiles
- Coupes : coupes d'acier, poterie (assiette du Mont St Michel)
- Compression/Décompression : MP3, xvid, divx, h264 codecs audio et vidéo, tar, zip.
- Cryptage : RSA (achat électronique)
- Internet : Routage des informations, moteurs de recherche
- Simulation : Prévision météo, Explosion nucléaire
- Traitement d'images (médecine) , effets spéciaux (cinéma)

Algorithmes

- Les programmes deviennent gros
- On ne peut plus programmer et penser n'importe comment
- Voir Aide/ A propos d'Acrobat Reader Pro

Algorithmique : une science très ancienne

- Remonte à l'Antiquité (Euclide : calcul du pgcd de deux nombres, Archimède : calcul d'une approximation de π)
- Le mot algorithme vient du mathématicien arabe du 9ème siècle Al Khou Warismi
- L'algorithmique reste la base dure de l'Informatique. Elle intervient dans
  - Le software (logiciel)
  - Le hardware : un processeur est un câblage d'algorithmes fréquemment utilisés (multiplication …)

Algorithmique

- L'aspect scientifique de l'informatique
- « **L'informatique n'est pas plus la science des ordinateurs que l'astronomie n'est celle des téléscopes** »
  E. Dijkstra, Turing award 1972

Algorithme

- Un algorithme prend en entrée des données et fournit un résultat permettant de donner la réponse à un problème

- Un algorithme = une série d'opérations à effectuer :
  - Opérations exécutées en séquence ⇒ algorithme séquentiel.
  - Opérations exécutées en parallèle ⇒ algorithme parallèle.
  - Opérations exécutées sur un réseau de processeurs ⇒ algorithme réparti ou distribué.

- Mise en œuvre de l'algorithme
  - = implémentation (plus général que le codage)
  - = écriture de ces opérations dans un langage de programmation. Donne un programme.

Algorithme versus programme

- Un programme implémente un algorithme
- **Thèse de Turing-Church** : les problèmes ayant une solution algorithmique sont ceux résolvables par une machine de Turing (théorie de la calculabilité)
- On ne peut pas résoudre tous les problèmes avec des algorithmes (indécidabilité)
  - Problème de la halte
  - Savoir de façon indépendante si un algorithme est juste

Algorithmes

- Tous les algorithmes ne sont pas équivalents. On les différencie selon 2 critères
  - Temps de calcul : lents vs rapides
  - Mémoire utilisée : peu vs beaucoup
- On parle de complexité en temps (vitesse) ou en espace (mémoire utilisée)

Pourquoi faire des algorithmes rapides ?

- Pourquoi faire des algo efficaces ? Les ordinateurs vont super vite !
- Je peux faire un algorithme en n^2 avec n=10000
- Je voudrais faire avec n=100000. Tous les 2 ans la puissance des ordinateurs est multipliée par 2 (optimiste). Quand est-ce que je pourrais faire avec n=100000 ?

Pourquoi faire des algorithmes rapides ?

- Pourquoi faire des algorithmes efficaces ? Les ordinateurs vont super vite !
- Je peux faire un algorithme en n^2 avec n=10000
- Je voudrais faire avec n=100000. Tous les 2 ans, la puissance des ordinateurs est multipliée par 2 (optimiste). Quand est-ce que je pourrais faire avec n=100000 ?

- Réponse:
  - p=100000 q=10000; p=10*q; donc $p^2 = 100*q^2$. Donc besoin de 100 fois plus de puissance
  - $2^6 = 64$, $2^7 = 128$ donc obtenue dans 7*2 ans=14 ans !!!
- Je trouve un algo en n log(n) pour p, je ferai 17*100000= 1 700 000 opérations donc 100/1.7 fois moins de temps !!!

Complexité des algorithmes

- **But:**
  - **avoir une idée de la difficulté des problèmes**
  - **donner une idée du temps de calcul ou de l'espace nécessaire pour résoudre un problème**
- Cela va permettre de comparer les algorithmes
- Exprimée en fonction du nombre de données et de leur taille.
- A défaut de pouvoir faire mieux :
  - On considère que les opérations sont toutes équivalentes
  - Seul l'ordre de grandeur nous intéresse.
  - On considère uniquement le pire des cas
- Pour n données on aura : O(n) linéaire, $O(n^2)$ quadratique O(n log(n)),…

Pourquoi faire des algorithmes rapides ?

- Dans la vie réelle, ça n'augmente pas toujours !

- OUI et NON:
  - Certains problèmes sont résolus
  - Pour d'autres, on simplifie moins et donc la taille des données à traiter augmente ! Réalité virtuelle : de mieux en mieux définie
- Dès que l'on résout quelque chose : on complexifie!

Algorithmes : vitesse

- On peut qualifier de rapide un algorithme qui met un temps polynomial en n (nombre de données) pour être exécuté. Exemple n^2, n^8
- Pour certains problèmes : voyageur de commerce, remplissage de sac-à-dos de façon optimum. On ne sait pas s'il existe un algorithme rapide. On connaît des algorithmes exponentiels en temps : 2^n.
- 1 million de $, si vous résolvez la question!

Algorithme : preuve

- On peut prouver les algorithmes !
- Un algorithme est dit totalement correct si pour tout jeu de données il termine et rend le résultat attendu
- C'est assez difficile, mais c'est important
  - Codage/Décodage des données. Si bug alors tout est perdu
  - Centrale nucléaire
  - Airbus

- Attention : un algorithme juste peut être mal implémenté

Algorithme : résumé

- C'est ancien
- C'est fondamental en informatique
- Ca se prouve
- On estime le temps de réponse et la mémoire prise

Plan

- Algorithmes
- Complexité
- Preuve
- **Structures de données**
- Pseudo Langage
- Tableaux
- Tri par insertion
- Calcul de x^n
- Logarithme
- Recherche dichotomique

2 types de personnes

- En informatique il y a deux types de personnes
  - Ceux qui écrivent les algorithmes
  - Ceux qui les implémente
- En cuisine
  - Les chefs font les recettes
  - Ils ne font pas la cuisine. Eventuellement des prototypes et ils surveillent. Ou font des choses très précises.
  - En informatique c'est pareil
- Problème: il faut se comprendre!

Se comprendre…

- Pour se comprendre on va améliorer le langage
  - On définit des choses communes bien précises (pareil en cuisine)
  - On essaie de regrouper certaines méthodes ou techniques
- En Informatique
  - Langages : variables, boucle, incrémentation etc. . .
  - Regroupement : structures de données et types abstraits

Variable

- Une variable sert à mémoriser de l'information
- Ce qui est mis dans une variable est en fait mis dans une partie de la mémoire

Type de données

- Un **type de données**, ou simplement **type**, définit le genre de contenu d'une donnée et les opérations pouvant être effectuées sur la variable correspondante.
- Par exemple:
  - int représente un entier, on peut faire des additions, des multiplications. La division est entière et on peut faire des décalages de bits …
  - Date représente une date, l'addition aura un certain sens, on pourra écrire la date sous certaines formes (jj/mm/aaaa ou « 10 Septembre 2009 »)
- Les types les plus communs sont :
  - int, float, double, char, boolean et le type pointeur
  - Entier, réel, chaîne, caractère, booléen, pointeur

Structures de données

- Une **structure de données** est une manière particulière de stocker et d'organiser des données dans un ordinateur de façon à pouvoir être utilisées efficacement.

- Différents types de structures de données existent pour répondre à des problèmes très précis :
  - B-arbres dans les bases de données
  - Table de hash par les compilateurs pour retrouver les identificateurs.

Structures de données

- Ingrédient essentiel pour l'efficacité des algorithmes.
- Permettent d'organiser la gestion des données
- Une structure de données ne regroupe pas nécessairement des objets du même type.

Type abstrait de données

- Un **type abstrait** ou une **structure de données abstraite** est une spécification mathématique d'un ensemble de données et de l'ensemble des opérations qu'elles peuvent effectuer. On qualifie d'abstrait ce type de données car il correspond à un cahier des charges qu'une structure de données doit ensuite implémenter.
- Par exemple : le type abstrait de pile sera défini par 2 opérations :
  - Push qui insère un élément dans la structure
  - Pop qui extrait les éléments de la structure, avec la contrainte que pop retourne l'élément qui a été empilé le plus récemment et qui n'a pas encore été dépilé
- Les types abstraits sont des entités purement théoriques utilisés principalement pour simplifier la description des algorithmes

Structures de données et type abstrait

- Quand la nature des données n'a pas d'influence sur les opérations à effectuer, on parle alors de type abstrait générique et on fait la confusion avec les structures de données.

- Dans ce cours, on considérera que les structures de données sont indépendantes du type des données et qu'elles sont définies par l'ensemble des opérations qu'elles effectuent sur ces données

Structures de données

- Permettent de gérer et d'organiser des données
- Sont définies à partir d'un ensemble d'opérations qu'elles peuvent effectuer sur les données

Structures de données et langage à objets

- Les structures de données permettent d'organiser le code
- Une Sdd correspond à une classe contenant un ensemble d'objets
- 2 parties:
  - Une visible correspondant aux opérations que la structure de données peut effectuée. Dans les langages à objets c'est la partie publique de la classe
  - Une cachée qui contient les méthodes internes permettant de réaliser les opérations de la Sdd. Dans les langages à objets c'est la partie privée de la classe

- La partie visible de la Sdd est parfois appelée API de la Sdd : Application Programming Interface, autrement dit l'interface de programmation de la Sdd qui permet son utilisation.

Plan

- Algorithmes
- Complexité
- Preuve
- Structures de données
- **Pseudo Langage**
- Tableaux
- Tri par insertion
- Calcul de x^n
- Logarithme
- Recherche dichotomique

Notion de pseudo langage

- On a besoin d'un langage formel minimum pour décrire un algorithme
- Un langage de programmation (Java, C, Pascal, etc.) est trop contraignant
- Dans la littérature, les algorithmes sont décrits dans un pseudo langage qui ressemble à un langage de programmation (le pseudo langage utilisé dépend donc de l'auteur et peut être spécifié par celui-ci en début d'ouvrage)

Pseudo langage

- Tous les pseudo langages recouvrent les mêmes concepts
  - Variables, affectation
  - Structures de contrôle : séquence, conditionnelle, itération
  - Découpage de l'algorithme en sous-programmes (fonctions, procédures).
  - Structures de données simples ou élaborées (tableaux, listes, dictionnaires, etc.)

Pseudo langage : variables, affectation

- Les variables sont indiquées avec leur type : booléen b, entier n, réel x, caractère c, chaîne s, etc.
- On est souple du moment qu'il n'y a pas d'ambiguïté

- Le signe de l'affectation n'est pas « = », ni « := » (comme en Pascal) mais « ← » qui illustre bien la réalité de l'affectation (« mettre dedans »)

Pseudo langage : structures de données

- Les tableaux sont utilisés. Si A est un tableau, A[i] est le $i^{ème}$ élément du tableau

- Les structures sont utilisées. Si P est une structure modélisant un point et x un champ de cette structure représentant l'abscisse du point, P.x est l'abscisse de P

  Remarque : une structure est une classe sans les méthodes

Pseudo langage : le séquencement des instructions

- Les instructions simples sont séquencés par « ; » (si besoin)
- Les blocs d'instructions sont entourés par
  - { … }
  - début … fin

Pseudo langage : la conditionnelle

- La conditionnelle est donnée par :

```
si (condition) {
    instruction1;
} sinon {
  instruction2;
}
```

Pseudo langage : les itérations

- Nous utilisons plusieurs types de boucles :
  - tant que (condition) { … }
  - faire { … } tant que (condition)
  - répéter {…} jusqu'à (condition)
  - pour i de min à max { … }

Pseudo langage : les fonctions

- Une fonction est un « petit » programme qui renvoie une valeur

- Elles permettent un découpage de l'algorithme qui rend sa compréhension et son développement plus facile

Pseudo langage : les fonctions

- Une fonction a une liste de paramètres typés et un type de retour (son prototype)

- Devant chaque paramètre, la manière dont le paramètre est passé est mentionné.
- Passage de paramètre : **un paramètre est soit**
  - **En Entrée** : la donnée est fournie du code appelant au code appelé, autrement dit on passe la valeur à la fonction. On précède le paramètre par **e ou i** (entrée ou input)
  - **En Sortie** : la donnée est fournie par le code appelé au code appelant, autrement dit la fonction passe cette valeur au code appelant. On précède le paramètre par **s ou o** (sortie ou output)
  - **En Entrée/Sortie** : la donnée est fournie par le code appelé à la fonction (on passe la valeur à la fonction) qui ensuite fournit à son tour la valeur au code appelant (la fonction passe à nouveau la valeur). On précède le paramètre par **es ou io** (entrée/sortie ou input/output)

Pseudo-langage : les fonctions

- maFonction(**e** int i, **s** int j, **es** int k);
  - **En Entrée** : la fonction lit la valeur du paramètre, ici **i**. Les modifications qu'elle fera avec i ne seront pas transmises au programme appelant
  - **En Sortie** : la fonction ne lit pas la valeur du paramètre, ici **j**. Elle écrit dans j et le programme appelant récupère cette valeur, donc j peut être modifié par la fonction
  - **En Entrée/Sortie** : la fonction lit la valeur du paramètre, ici **k**. et elle passe au programme appelant les modifications faites pour k

- En l'absence de précision, on considérera que le paramètre est passé en entrée.
- Le passage en entrée/sortie est souvent appelé passage par référence ou par variable

Pseudo-langage : les fonctions

- Code appelant :

```
i ← 3
j ← 5
k ← 8
maFonction(i,j,k);
Afficher(i,j,k)
```

- maFonction(**e** int i, **s** int j, **es** int k) {

```
        i ← i + 1
        j ← 6
        k ← k + 2
}
```

- Résultat de Afficher(i,j,k) : 3 6 10

Pseudo-langage : les fonctions

- Le passage de paramètre en e/s est aussi souvent appelé passage par référence.
- Certains langages vont donner ou non la possibilité de définir le mode de passage que l'on souhaite
  - Java :
    - types de base (int, float, double, …) sont passés uniquement en entrée (i.e. par valeur)
    - Les objets, ou plus précisément les références, sont passés uniquement en e/s (i.e. par référence)

Plan

- Algorithmes
- Complexité
- Preuve
- Structures de données
- Pseudo Langage
- **Tableaux**
- Tri par insertion
- Calcul de x^n
- Logarithme
- Recherche dichotomique

Tableaux

- un **tableau** (*array* en anglais) est une Sdd de base qui est un ensemble d'éléments, auquel on accède à travers un numéro d'indice.
- Le temps d'accès à un élément par son indice est constant, quel que soit l'élément désiré
- Les éléments d'un tableau sont contigus dans l'espace mémoire. Avec l'indice, on sait donc à combien de cases mémoire se trouve l'élément en partant du début du tableau.
- On désigne habituellement les tableaux par des lettres majuscules. SI T est un tableau alors T[i] représente l'élément à l'indice i.

Tableaux

- Avantages : accès direct au ième élément
- Inconvénients : les opérations d'insertion et de suppression sont impossibles
  - sauf si on crée un nouveau tableau, de taille plus grande ou plus petite (selon l'opération). Il est alors nécessaire de copier tous les éléments du tableau original dans le nouveau tableau. Cela fait donc beaucoup d'opérations.

Tableaux

- Un tableau peut avoir une dimension, on parle alors de **vecteur**
- Un tableau peut avoir plusieurs dimensions, on dit qu'il est **multidimensionnel**. On le note T[i][k]
- La taille d'un tableau doit être définie avant son utilisation et ne peut plus être changée.
- Les seules opérations possibles sont set et get (on affecte un élément à un indice et on lit un élément à un indice).

Tableaux multidimensionnel

- On peut linéariser un tableau à plusieurs dimensions

i\ j	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

T[i][j]=L[i *5 +j]

T[1][3]=L[1*5+3]=L[8]

T[i][j]=[i * #col + j]

Tableaux de tableaux

- On peut définir un tableau de tableaux (ou de n'importe quoi en fait)

T[0]	0	1	2	3	4	5	6
T[1]	0	1	2	3			
T[2]	0	1	2	3	4	5	
T[3]	0	1	2				

Tableaux : élément ou indice ?

- Il ne faut pas confondre un élément du tableau et l'indice de cet élément

Plan

- Algorithmes
- Complexité
- Preuve
- Structures de données
- Pseudo Langage
- Tableaux
- **Tri par insertion**
- Calcul de x^n
- Logarithme
- Recherche dichotomique

Tri par insertion

- D F A G B H E C
- On fait 2 paquets : 1 non trié - 1 trié
- Un paquet d'un élément est trié
- On prend un élément non trié et on le range à sa place dans le paquet trié
- On prend D : D - F A G B H E C
- On prend F : D F - A G B H E C
- On prend A : A D F - G B H E C
- On prend G : A D F G - B H E C
- On prend B : A B D F G - H E C

Tri par insertion

- Comment ranger dans le paquet trié ?
- On doit ranger B dans A D F G – B H E C
- On a A D F G B
- On parcourt de droite à gauche A D F G
- Tant que la valeur est > B, on l'échange avec B
  - A D F **B G**
  - A D **B F** G
  - A **B D** F G
  - Fin

Tri par insertion

- On pourrait dire :
  - On fait 2 tas : un trié, puis un non trié.
  - Au début le tas trié est vide, celui non trié contient tous les éléments
  - On prend un élément non trié et on le range à sa place dans le tas trié
- Cet algorithme est exact, mais
  - sa formulation est ambiguë, par exemple « prendre » veut dire le supprimer du tas non trié aussi.
  - Il n'est pas assez précis : comment range-t-on un élément, qu'est-ce que sa place ?
- Pour avoir la précision et supprimer l'ambiguïté on utilise un pseudo langage

Tri par insertion : algorithme


```
triInsertion(entier[] t) {
    pour (i de 2 à n) { // n taille de t
      entier c ← t[i];
      entier k ← i-1;
      // c doit être inséré dans le tableau ordonné t[1..i-1]
      // on cherche de droite à gauche la première valeur t[k]
      // plus petite que c
      tant que (k≥1 et t[k]>c) {
        t[k+1] ← t[k]; // on décale
        k ← k-1;
      }
      // on a trouvé la bonne place
      t[k+1] ← c;
    }
}
```

Tri par insertion

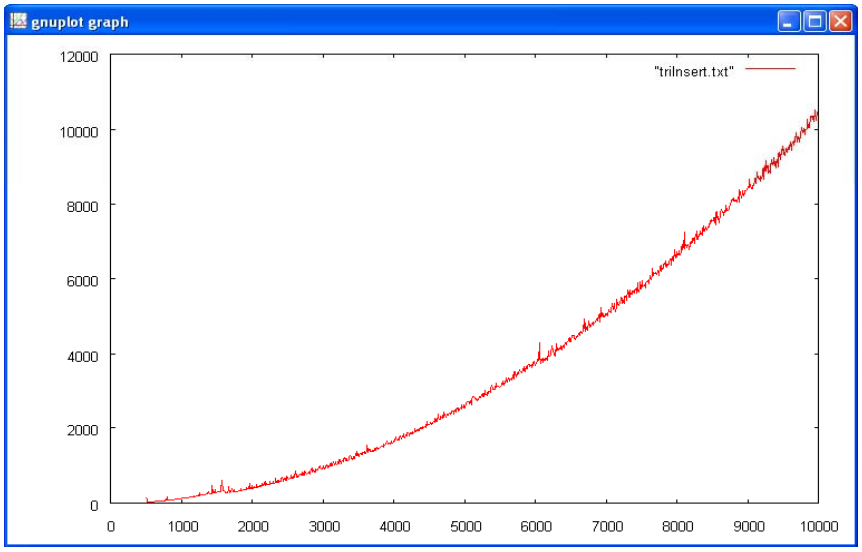
- Francis Avnaim a fait l'étude suivante de la complexité expérimentale de cet algorithme, implémenté en Processing

- On trie des tableaux de taille croissante de 500 à 10000 par pas de 10

  - Aléatoires
  - Triés en ordre inverse
  - Déjà triés

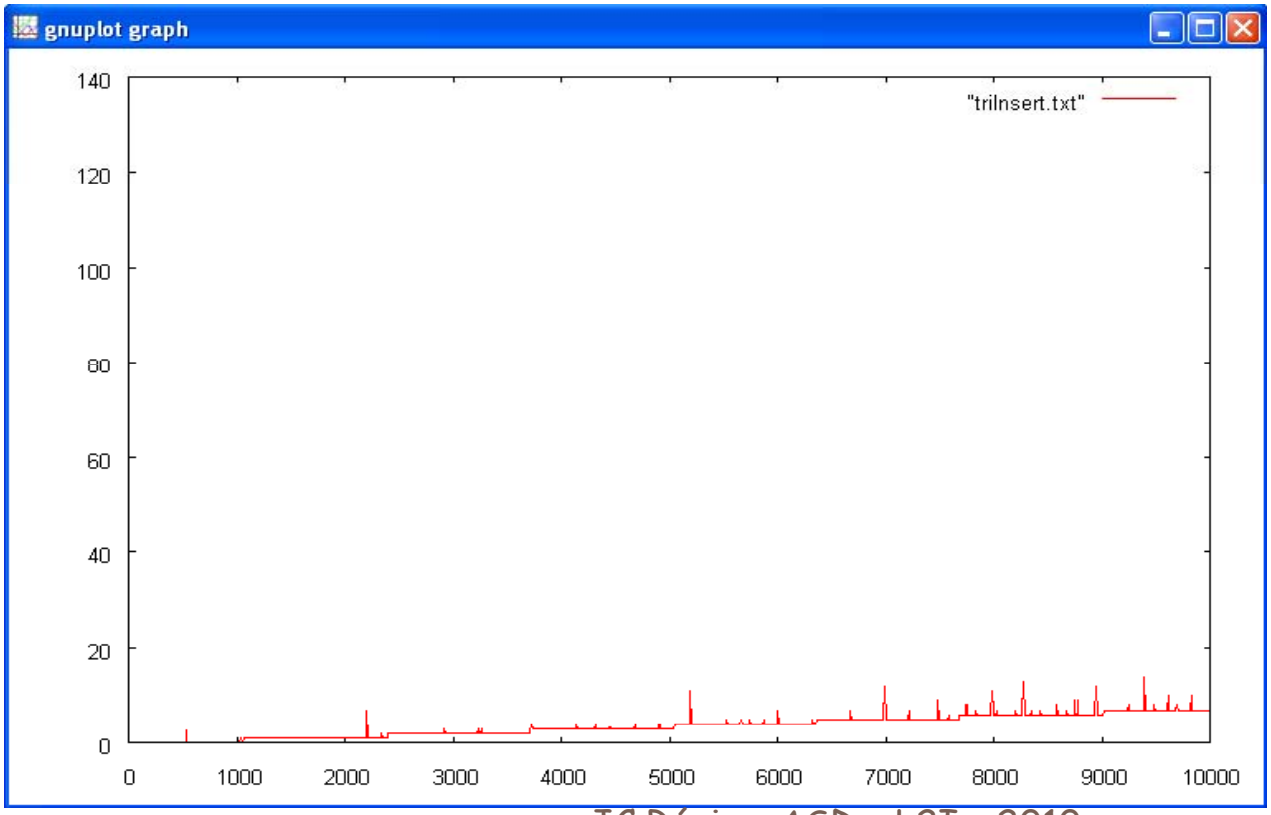
Tri par insertion

- Tableaux aléatoires



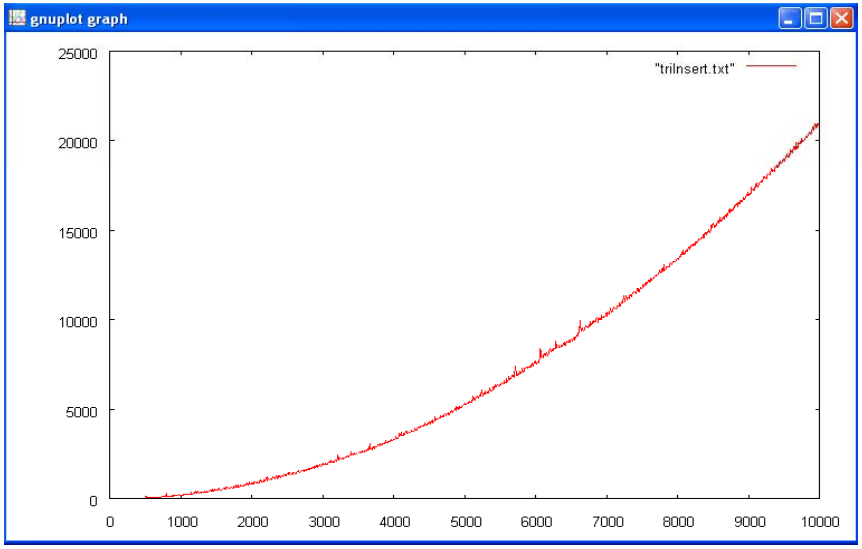
Tri par insertion

- Tableaux ordonnés



Tri par insertion

- Tableaux en ordre inverse



Tri par insertion

- Conclusions de l'expérimentation
  - Tableaux aléatoires et en ordre inverse : fonction carré ?
  - Tableaux ordonnés : fonction linéaire ?

Tri par insertion : complexité

- Parties bleues : 7 op. el. n-1 fois soit 7(n-1)
- Partie jaune : 6 op. el.
  Combien de fois ?
- Partie mauve : 3 op. el.
  Combien de fois ?
- Nombre de fois ou les parties mauves et jaunes sont appelées ?
- Complexité mauve + jaune:
- Complexité totale inférieure à ?

```
triInsertion(entier[] t) {
    pour (i de 2 à n) {
      entier c ← t[i];
      entier k ← i-1;
      tant que (k≥1 et t[k]>c) {
        t[k+1] ← t[k];
        k ← k-1;
      }
      t[k+1] ← c;
    }
}
```

Tri par insertion : complexité

- Parties bleues : 7 op. el. n-1 fois soit 7(n-1)
- Partie jaune : 6 op. el.
  Combien de fois ? **au pire i fois (k de i-1 à 1), donc 6i à chaque fois**
- Partie mauve : 3 op. el.
  Combien de fois ? **au pire i fois, donc 3i à chaque fois**
- Nombre de fois ou les parties mauves et jaunes sont appelées ?
- Complexité mauve + jaune:

```
triInsertion(entier[] t) {
    pour (i de 2 à n) {
      entier c ← t[i];
      entier k ← i-1;
      tant que (k≥1 et t[k]>c) {
        t[k+1] ← t[k];
        k ← k-1;
      }
      t[k+1] ← c;
    }
}
```

Tri par insertion : complexité

- Parties bleues : 7 op. el. n-1 fois soit 7(n-1)
- Partie jaune : 6 op. el.
  Combien de fois ? au pire i fois, donc 6i à chaque fois
- Partie mauve : 3 op. el.
  Combien de fois ? au pire i fois, donc 3i à chaque fois
- Nombre de fois ou les parties mauves et jaunes sont appelées ? **(n-1) fois**
- Complexité mauve + jaune:
  9 + 9x2 +…+ 9x(n-1)=9n(n-1)/2
- Complexité totale inférieure à ?

```
triInsertion(entier[] t) {
    pour (i de 2 à n) {
      entier c ← t[i];
      entier k ← i-1;
      tant que (k≥1 et t[k]>c) {
        t[k+1] ← t[k];
        k ← k-1;
      }
      t[k+1] ← c;
    }
}
```

Tri par insertion : complexité

- Parties bleus : 7 op. el. n-1 fois soit 7(n-1)
- Partie jaune : 6 op. el.
  Combien de fois ? au pire i fois, donc 6i à chaque fois
- Partie mauve : 3 op. el.
  Combien de fois ? au pire i fois, donc 3i à chaque fois
- Nombre de fois ou les parties mauves et jaunes sont appelées ? (n-1) fois
- Complexité mauve + jaune:
  $9 + 9x2 + \ldots + 9x(n-1) = 9n(n-1)/2$
- Complexité totale inférieure à
  $9n(n-1)/2 + 7(n-1)$
- De l'ordre de n^2

```
triInsertion(entier[] t) {
    pour (i de 2 à n) {
      entier c ← t[i];
      entier k ← i-1;
      tant que (k≥1 et t[k]>c) {
        t[k+1] ← t[k];
        k ← k-1;
      }
      t[k+1] ← c;
    }
}
```

Tri par insertion

- Si tableau ordonné, la boucle tant que (zone jaune) n'est jamais exécutée. La complexité est alors exactement 10(n-1)
- Si tableau en ordre inverse (cas le pire), la complexité est exactement 7(n-1) + 9((n-1)n / 2)
- Si tableau aléatoire, la complexité est entre les deux complexités précédentes.
  Résultat expérimental montre une fonction carré
- Pire des cas quadratique

```
triInsertion(entier[] t) {
    pour (i de 2 à n) {
      entier c ← t[i];
      entier k ← i-1;
      tant que (k≥1 et t[k]>c) {
        t[k+1] ← t[k];
        k ← k-1;
      }
      t[k+1] ← c;
    }
}
```

Plan

- Algorithmes
- Complexité
- Preuve
- Structures de données
- Pseudo Langage
- Tableaux
- Tri par insertion
- **Calcul de x^n**
- Logarithme
- Recherche dichotomique

Calcul de x^n

- Comment calculez-vous x^n à partir de x ?
- Par itérations:

```
y ← 1
pour i de 1 à n : y ← y * x
```

- Complexité ?
- x^8 ?
- Combien d'opérations ?
- x^{16} ?
- x^{12} ?
- Généralisation:
  - si n est pair
  - si n est impair

Calcul de x^n

- Comment calculez-vous x^n à partir de x ?
- Par itérations:

```
y ← 1
pour i de 1 à n : y ← y * x
```

- Complexité ? Linéaire O(n)
- x^8 ?
- Combien d'opérations ?
- x^{16} ?
- x^{12} ?
- Généralisation:
  - si n est pair
  - si n est impair

Calcul de x^n

- Comment calculez-vous x^n à partir de x ?
- Par itérations:

```
y ← 1
pour i de 1 à n : y ← y * x
```

- Complexité ? Linéaire O(n)
- x^8 ? $x^4 * x^4$ et $x^4 = x^2 * x^2$ et $x^2 = x * x$
- Combien d'opérations ? 3
- x^{16} ? $x^8 * x^8$, donc 4 opérations
- x^{12} ?
- Généralisation:
  - si n est pair
  - si n est impair

Calcul de x^n

- Comment calculez-vous x^n à partir de x ?
- Par itérations:

```
y ← 1
pour i de 1 à n : y ← y * x
```

- Complexité ? Linéaire O(n)
- x^8 ? $x^4 * x^4$ et $x^4 = x^2 * x^2$ et $x^2 = x * x$
- Combien d'opérations ? 3
- x^{16} ? $x^8 * x^8$, donc 4 opérations
- x^{12} ? $x^6 * x^6$ et $x^6 = x^3 * x^3$ et $x^3 = x * x * x$, donc 4 opérations
- Généralisation:
  - si n est pair $x^n =$
  - si n est impair $x^n =$

Calcul de x^n

- Comment calculez-vous x^n à partir de x ?
- Par itérations:

```
y ← 1
pour i de 1 à n : y ← y * x
```

- Complexité ? Linéaire O(n)
- x^8 ? $x^4 * x^4$ et $x^4 = x^2 * x^2$ et $x^2 = x * x$
- Combien d'opérations ? 3
- x^{16} ? $x^8 * x^8$, donc 4 opérations
- x^{12} ? $x^6 * x^6$ et $x^6 = x^3 * x^3$ et $x^3 = x * x * x$, donc 4 opérations
- Généralisation:
  - si n est pair $x^n = x^{n/2} * x^{n/2}$
  - si n est impair $x^n = x^{(n/2)} * x^{(n/2)} * x$
  - / est la division entière (5/2= 2)

Rappel sur les Logarithmes

- La fonction logarithme est simplement l'inverse de la fonction exponentielle. Il y équivalence entre $b^x=y$ et $x=\log_b(y)$.

- En informatique le logarithme est souvent utilisé parce que les ordinateurs utilisent la base 2.
- On utilisera donc souvent les logarithmes de base 2.

- Le logarithme de n reflète combien de fois on doit doubler le nombre 1 pour obtenir le nombre n.
- De façon équivalent elle reflète aussi le nombre de fois où l'on doit diviser n pour obtenir 1.

Calcul de x^n : complexité

- Généralisation:
  - si n est pair $x^n = x^{n/2} * x^{n/2}$
  - si n est impair $x^n = x^{(n/2)} * x^{(n/2)} * x$
- À chaque fois on divise par 2
- Pour chaque impair on ajoute 1
- Complexité : log(n) + nombre de bit à 1 dans n - 1

Plan

- Algorithmes
- Complexité
- Preuve
- Structures de données
- Pseudo Langage
- Tableaux
- Tri par insertion
- Calcul de x^n
- Logarithme
- **Recherche dichotomique**

Recherche dans un tableau

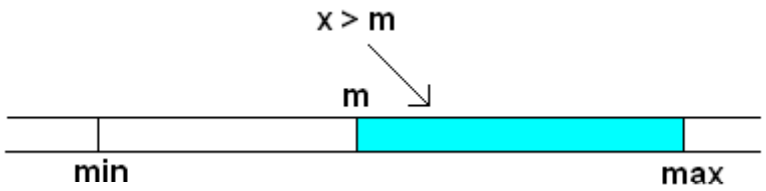
- Considérons un tableau a de nombres entiers, de taille n
- On désire savoir si un nombre donné x est dans le tableau
- Pour le savoir, on parcourt le tableau en comparant les éléments de a à x (recherche séquentielle)
- Au plus (quand x n'est pas dans a), on fera n comparaisons
- L'algorithme est donc de complexité O(n)

Recherche dichotomique

- Peut-on faire mieux ?
- Oui, si le tableau est préalablement trié. C'est la recherche dichotomique

- Idée : on compare x à la valeur centrale m de a.
  Si x = m, on a trouvé x dans a.
  Sinon, si x < m, on cherche x dans la moitié inférieure du tableau, sinon on cherche x dans la moitié supérieure

-



Recherche dans un tableau


```
entier rechercheDicho(entier a[min..max], entier x) {
    si (min > max) {
        retourner -1;
    }
    sinon {
        entier i ← (min + max) / 2;
        entier m ← a[i];
        si (x = m)
            retourner i;
        sinon si (x < m)
            retourner rechercheDicho(a[min..i-1], x);
        sinon
            retourner rechercheDicho(a[i+1..max], x);
    }
}
```

Analyse de la recherche dichotomique

- Appelons C(n) le nombre d'opérations élémentaires exécutées pour effectuer la recherche dichotomique dans un tableau de taille n

- On a, avec k_1 et k_2 deux constantes majorant respectivement le nombre d'opérations des parties en bleu et en mauve :

$$C(n) \leq (k_1 + k_2) + C(n/2)$$
$$\leq 2.(k_1 + k_2) + C(n/2^2)$$
$$\leq 3.(k_1 + k_2) + C(n/2^3)$$
$$\ldots$$
$$\leq r.(k_1 + k_2) + C(1)$$
$$\leq r.(k_1 + k_2) + k_1 \leq 2r\,.\,(k_1 + k_2)$$

Analyse de la recherche dichotomique

- Il nous reste à estimer r
- r est tel que $2^r \leq n < 2^{r+1}$
- On a donc $r \leq \log_2(n)$
- Et finalement C(n) est en $O(\log_2 n)$

Recherche dichotomique : version itérative


```
entier rechercheDichotomique (entier a[1..n], entier x)
fini ← faux; min ←  1; max ← n;
tant que (min ≤ max){
     p ← (min + max)/2;
     si (t[p] ≤ x){
          min ← p+1
     }
     si (t[p] ≥ x){
          max ← p-1
     }
}
si (t[p]=x){
     renvoyer p
} sinon {
     renvoyer -1
}
```