

Algorithmes et structures de données avec Manuel Lafond

Introduction

- **Algorithme**: séquence d'instructions à suivre afin d'accomplir une tâche précise.
- Exemples de tâche
 - Faire un gâteau
 - Trier une liste d'éléments
 - Insérer dans un arbre binaire de recherche
 - Trouver le chemin le plus court entre 2 adresses
 - ...

Exemples de tâches

- Trouver une route optimale pour des paquets à transmettre d'un site web à des visiteurs.
 - Algorithme de *Dijkstra*
- Rechercher une séquence d'ADN dans une base de données.
 - Algorithme de *Boyer-Moore*
- Trier des pages web par pertinence dans un engin de recherche.
 - Algorithme *PageRank*.
- Trouver le classement le plus probable des mots dans une phrase.
 - Algorithme de Viterbi.

- Il y a souvent plusieurs algorithmes accomplissant une même tâche.
 - (et parfois on n'en connaît aucun)
- Lequel choisir?
- Il nous faut une mesure de performance d'un algorithme.

- Mesures de performance
 - Temps
 - Mémoire
 - Nombre de processeurs
 - Probabilité de succès
 - Nombre d'accès à la cache fructueux
 - Quantité d'énergie utilisée
 - Nombre de qubits
 - ...

- Mesures de performance
 - **Temps !!!**
 - Mémoire
 - Nombre de processeurs
 - Probabilité de succès
 - Nombre d'accès à la cache fructueux
 - Quantité d'énergie utilisée
 - Nombre de qubits
 - ...

- Mesures de performance

- **Temps !!! Temps asymptotique O**

- Mémoire

- Nombre de processeurs

- Probabilité de succès

- Nombre d'accès à la cache fructueux

- Quantité d'énergie utilisée

- Nombre de qubits

- ...

- C'est quoi une tâche?
- Il faut bien définir **ce qu'on a** et **ce qu'on veut**.
- Donc bien préciser **l'entrée** et la **sortie**.

- Exemple de tâche bien définie:
- Problème du **tri**:
 - **Entrée**: un tableau $T = [t_1, t_2, \dots, t_n]$ contenant n entiers.
 - **Sortie**: une permutation $T' = [t'_1, t'_2, \dots, t'_n]$ des éléments de T telle que $t'_1 \leq t'_2 \leq \dots \leq t'_n$.
- Rappel: une permutation est un réordonnement des éléments d'une séquence.

- Exemple de tâche bien définie:
- Problème du **chemin le plus court**:
 - **Entrée**: un graphe $G = (V, E)$ et deux sommets u et v .
 - **Sortie**: le chemin le plus court entre u et v dans G , s'il existe.
- En supposant que l'on a bien établi les définitions de "graphe", "sommet", "chemin", "plus court".

- Exemple de tâche bien définie:
- Problème du **chemin pas trop long**:
 - **Entrée**: un graphe $G = (V, E)$, deux sommets u et v et un entier k .
 - **Sortie**: VRAI s'il existe un chemin de longueur $\leq k$ entre u et v , et FAUX sinon.

- Exemple de tâche **mal** définie:

- Exemple de tâche **mal** définie:
- "Salut Manuel. J'ai plein de séquences d'ADN d'animaux à comparer pour mes recherches biologiques. Peux-tu les analyser?"

- Exemple de tâche **mal** définie:
- **Entrée**: un ensemble de séquences $A_1, A_2, \dots, A_n$.

- Exemple de tâche **mal** définie:
- **Entrée**: un ensemble de séquences $A_1, A_2, \dots, A_n$.
- **Sortie**: une analyse.

- Exemple de tâche **mal** définie:
- **Entrée**: un ensemble de séquences $A_1, A_2, \dots, A_n$.
- **Sortie**: une analyse.

IFT599 – Science des données

- Exemple de tâche ~~mal~~ bien définie:
- **Entrée**: un ensemble de séquences $A_1, A_2, \dots, A_n$.
- **Sortie**: la matrice des distances d'édition entre chaque paire de séquences.
 - où la distance d'édition est le nombre de caractères à modifier pour passer d'une séquence à l'autre.
 - calculer cette distance est une tâche algorithmique en soi.

- Une fois que l'on a précisé (et bien compris!!!) notre entrée et notre sortie, on peut travailler.

- Une fois que l'on a précisé (et bien compris!!!) notre entrée et notre sortie, on peut travailler.
- Reprenons le problème du tri:
 - **Entrée**: un tableau $T = [t_1, t_2, \dots, t_n]$ contenant n entiers.
 - **Sortie**: une permutation $T' = [t'_1, t'_2, \dots, t'_n]$ des éléments de T telle que $t'_1 \leq t'_2 \leq \dots \leq t'_n$.
- Nous allons voir 3 algorithmes de tri.

Tri par insertion

triInsertion(T): //tableau indicé à 1

n = T.taille

pour j = 2 à n

val = T[j]

i = j - 1

//décaler tout le monde jusqu'à ce qu'on

//trouve où remplacer val

tant que i > 0 **et** T[i] > val

T[i + 1] = T[i]

i--

T[i + 1] = val

Tri fusion

triFusion(T):

si T.taille == 1 alors retourner [T[1]]

sinon

A = triFusion(T[1 .. n/2]) //où n = T.taille

B = triFusion(T[n/2 + 1 .. n])

S = [] //sortie

cptA = 1, cptB = 1

tant que S.taille < A.taille + B.taille

si cptB > B.taille ou (cptA ≤ A.taille et A[cptA] < B[cptB])

ajouter A[cptA] à la fin de S

cptA++

sinon

ajouter B[cptB] à la fin de S

cptB++

retourner S

Tri Sous-gradué

triSousGradué(T):

nbSwap = 0

fini = FALSE

tant que fini == FALSE:

 a_fait_swap = FALSE

pour i = 1 à n - 1

si T[i] > T[i + 1]

 échanger T[i] avec T[n - nbSwap]

 nbSwap++

 a_fait_swap = TRUE

fin si

fin pour

si !a_fait_swap **alors** fini = TRUE

fin tant que

//essayer avec T = [3 4 1 2]

Tri par insertion

triInsertion(T): //tableau indicé à 1

n = T.taille

pour j = 2 **à** n

val = T[j]

i = j - 1

//décaler tout le monde jusqu'à ce qu'on

//trouve où remplacer val

tant que i > 0 **et** T[i] > val

T[i + 1] = T[i]

i--

T[i + 1] = val

$O(n^2)$

Tri fusion

triFusion(T):

si T.taille == 1 alors retourner [T[1]]

sinon

A = triFusion(T[1 .. n/2]) //où n = T.taille

B = triFusion(T[n/2 + 1 .. n])

S = [] //sortie

cptA = 1, cptB = 1

tant que S.taille < A.taille + B.taille

si cptB > B.taille ou A[cptA] < B[cptB]

ajouter A[cptA] à la fin de S

cptA++

sinon

ajouter B[cptB] à la fin de S

cptB++

retourner S

$O(n \log n)$

Tri Sous-gradué

triSousGradué(T):

nbSwap = 0

fini = FALSE

tant que fini == FALSE:

pour i = 1 à n - 1

si T[i] > T[i + 1]

 échanger T[i] avec T[n - nbSwap]

 nbSwap++

 i = n + 100

fin si

fin pour

si i != n + 100

 fini = TRUE

fin si

fin tant que

$O(n^2)$

Tri Sous-gradué

triSousGradué(T):

nbSwap = 0

fini = FALSE

tant que fini == FALSE:

pour i = 1 à n - 1

si T[i] > T[i + 1]

 échanger T[i] avec T[n - nbSwap]

 nbSwap++

 i = n + 100

fin si

fin pour

si i != n + 100

 fini = TRUE

fin si

fin tant que

$O(n^2)$

Ne fonctionne pas!

- Avant d'implémenter un algorithme, il faut démontrer qu'il **fonctionne**:
 - est-ce qu'il termine? (pas de boucle infinie)
 - est-ce qu'il donne la sortie attendue sur toutes les instances?
- C'est parfois évident, mais aussi parfois trompeur!

Commencer mesurer le temps?

- Approche expérimentale
 - Programmer l'algorithme
 - Générer des instances (souvent aléatoires)
 - Vérifier que la sortie est correcte
 - Mesurer les temps d'exécution

Commencer mesurer le temps?

- Approche expérimentale
 - **Problème 1**: les temps d'exécution **dépendent de la machine** utilisée. En soi, les temps calculés ne veulent rien dire.

Commencer mesurer le temps?

- Approche expérimentale
 - **Problème 1**: les temps d'exécution **dépendent de la machine** utilisée. En soi, les temps calculés ne veulent rien dire.
 - **Problème 2**: ça semble aller vite, jusqu'au jour où vous rencontrez une **instance plus complexe**, mais jamais testée. Et là c'est super-lent.

Commencer mesurer le temps?

- Approche expérimentale
 - **Problème 1**: les temps d'exécution **dépendent de la machine** utilisée. En soi, les temps calculés ne veulent rien dire.
 - **Problème 2**: ça semble aller vite, jusqu'au jour où vous rencontrez une **instance plus complexe**, mais jamais testée. Et là c'est super-lent.
 - **Problème 3**: ça semble aller vite, jusqu'au jour où vous rencontrez une **instance plus grande** que celles que vous aviez considérées. Et là c'est super-lent.

Comment mesurer le temps

- **Problème 1**: les temps d'exécution **dépendent de la machine** utilisée. En soi, les temps calculés ne veulent rien dire.
- **Solution 1**: compter le **nombre d'opérations** effectuées par l'algorithme – il ne dépend pas de la machine.

Comment mesurer le temps

- **Problème 2**: ça semble aller vite, jusqu'au jour où vous rencontrez une **instance plus complexe**, mais jamais testée. Et là c'est super-lent.

Comment mesurer le temps

- **Problème 2**: ça semble aller vite, jusqu'au jour où vous rencontrez une **instance plus complexe**, mais jamais testée. Et là c'est super-lent.
- **Solution 2**: considérer le temps requis pour votre algorithme sur la *pire instance possible*. C'est ce qu'on appelle l'**analyse du pire cas**.

Comment mesurer le temps

- **Problème 3**: ça semble aller vite, jusqu'au jour où vous rencontrez une **instance plus grande** que celles que vous aviez considérées. Et là c'est super-lent.

Comment mesurer le temps

- **Problème 3**: ça semble aller vite, jusqu'au jour où vous rencontrez une **instance plus grande** que celles que vous aviez considérées. Et là c'est super-lent.
- **Solution 3**: considérer le temps requis en **fonction de la taille de l'entrée** donnée.

Comment mesurer le temps

- **Problème 3**: ça semble aller vite, jusqu'au jour où vous rencontrez une **instance plus grande** que celles que vous aviez considérées. Et là c'est super-lent.
- **Solution 3**: considérer le temps requis en **fonction de la taille de l'entrée** donnée.
- De cette façon, on sait comment se comportera l'algorithme même si les données explosent dans le futur (ce qui finit toujours par arriver).
 - Voir Big Data

Commencer mesurer le temps?

- Approche analytique
 - Compter le **nombre d'opérations** effectuées par l'algorithme.
 - Considérer le **pire cas**.
 - Exprimer la dépendance du temps requis en **fonction de la taille de l'entrée**.

Tri par insertion

triInsertion(T):

n = T.taille

Une opération (x1)

pour j = 2 à n

val = T[j]

Une opération (x n-1)

i = j - 1

Une opération (x n-1)

tant que i > 0 **et** T[i] > val

Trois opérations (x ???)

T[i + 1] = T[i]

Une opération (x ???)

i--

Une opération (x ???)

T[i + 1] = val

Une opération (x n-1)

Tri par insertion

triInsertion(T):

n = T.taille

Une opération (x1)

pour j = 2 à n

val = T[j]

Une opération (x n-1)

i = j - 1

Une opération (x n-1)

tant que i > 0 **et** T[i] > val

Trois opérations (x ???)

T[i + 1] = T[i]

Une opération (x ???)

i--

Une opération (x ???)

T[i + 1] = val

Une opération (x n-1)

$$??? = 1 + 2 + 3 + \dots + n = n(n+1)/2$$

Tri par insertion

triInsertion(T):

n = T.taille

Une opération (x1)

pour j = 2 à n

val = T[j]

Une opération (x n-1)

i = j - 1

Une opération (x n-1)

tant que i > 0 **et** T[i] > val

Trois opérations (x ???)

T[i + 1] = T[i]

Une opération (x ???)

i--

Une opération (x ???)

T[i + 1] = val

Une opération (x n-1)

$$\text{Total} = 1 + (n - 1) * 3 + (n(n+1)/2) * 5$$

Tri par insertion

triInsertion(T):

n = T.taille

Une opération (x1)

pour j = 2 à n

val = T[j]

Une opération (x n-1)

i = j - 1

Une opération (x n-1)

tant que i > 0 **et** T[i] > val

Trois opérations (x ???)

T[i + 1] = T[i]

Une opération (x ???)

i--

Une opération (x ???)

T[i + 1] = val

Une opération (x n-1)

$$\text{Total} = 1 + (n - 1) * 3 + (n(n+1)/2) * 5 = 2.5n^2 + 5.5n - 2$$

Tri par insertion

triInsertion(T):

n = T.taille

Une opération (x1)

pour j = 2 à n

val = T[j]

Une opération (x n-1)

i = j - 1

Une opération (x n-1)

tant que i > 0 **et** T[i] > val

Trois opérations (x ???)

T[i + 1] = T[i]

Une opération (x ???)

i--

Une opération (x ???)

T[i + 1] = val

Une opération (x n-1)

$$\text{Total} = 1 + (n - 1) * 3 + (n(n+1)/2) * 5 = 2.5n^2 + 5.5n - 2$$

$O(n^2)$

Tri fusion

triFusion(T):

si T.taille == 1 alors retourner [T[1]]

sinon

A = triFusion(T[1 .. n/2]) //où n = T.taille

B = triFusion(T[n/2 + 1 .. n])

S = [] //sortie

cptA = 1, cptB = 1

tant que S.taille < A.taille + B.taille

si cptB > B.taille ou A[cptA] < B[cptB]

ajouter A[cptA] à la fin de S

cptA++

sinon

ajouter B[cptB] à la fin de S

cptB++

retourner S

$O(n \log n)$

Tri fusion

triFusion(T):

si T.taille == 1 alors retourner [T[1]]

sinon

A = triFusion(T[1 .. n/2]) //

B = triFusion(T[n/2 + 1 .. n])

S = [] //sortie

cptA = 1, cptB = 1

tant que S.taille < A.taille + B.taille

si cptB > B.taille ou A[cptA] < B[cptB]

ajouter A[cptA] à la fin de S

cptA++

sinon

ajouter B[cptB] à la fin de S

cptB++

retourner S

Le premier appel prend un temps
 $f(n) = 2*f(n/2) + O(n)$

Et par le théorème maître,
 $f(n) = O(n \log n)$

Comment mesurer le temps

- Mais c'est quoi une opération?
- Qu'est-ce qu'on compte, qu'est-ce qu'on ne compte pas?

Comment mesurer le temps

- Il n'y a pas de standard. On compte généralement les opérations "atomiques":
 - déclarer/affecter une variable ($x = 10$)
 - comparer deux valeurs ($x > 10$)
 - additionner/soustraire ($x = x + y, i--$)
 - évaluer un prédicat booléen binaire (si $x > y$)
 - ...

Comment mesurer le temps

- Et parfois on triche un peu:
 - multiplier $a*b$
 - pas vraiment atomique – il faut un temps proportionnel au nombre de bits $O(\log(a) + \log(b))$
 - on considère souvent que c'est en temps constant
 - même chose pour l'addition en fait

Comment mesurer le temps

- Et parfois on triche un peu:
 - `s1 = "allo", s2 = "salut"`
 - `if (s1 == s2)`
 - l'évaluation ci-dessus prend un temps $O(|s1| + |s2|)$, mais on l'ignore souvent.

Comment mesurer le temps

- Souvent, on ignore les détails des sous-opérations qui ne dépendent pas de la taille de l'entrée.
- Dans le doute, soyez aussi précis que possible dans vos analyses.

Qu'est-ce que du pseudo-code?

- Le pseudo-code sert à:
 - décrire un algorithme sans dépendre d'un langage particulier
 - faire abstraction des détails d'implémentation qui sont longs, mais faciles à gérer

Qu'est-ce que du pseudo-code?

- À quel niveau de détail aller?
 - Un programmeur devrait pouvoir coder votre code sans ambiguïté

Tri par insertion (trop ambigu)

$n = T.\text{taille}$

pour $j = 2$ **à** n

$\text{val} = T[j]$

$i = j - 1$

décaler tout le monde jusqu'à ce qu'on trouve où remplacer val

$T[i + 1] = \text{val}$

Qu'est-ce que du pseudo-code?

- À quel niveau de détail aller?
 - Un programmeur devrait pouvoir coder votre code sans ambiguïté
 - Les instructions principales doivent apparaître : pas de ligne qui cache une complexité non-constante

Qu'est-ce que du pseudo-code?

- À quel niveau de détail aller?
 - Un programmeur devrait pouvoir coder votre code sans ambiguïté
 - Les instructions principales doivent apparaître : pas de ligne qui cache une complexité non-constante
 - Sauf quand c'est évident et qu'on comprend
 - En particulier, on pourra faire appel à des algorithmes vus en classe en "boîte noire"

Qu'est-ce que du pseudo-code?

- À quel niveau de détail aller?
 - Un programmeur devrait pouvoir coder votre code sans ambiguïté
 - Les instructions principales doivent apparaître : pas de ligne qui cache une complexité non-constante
 - Sauf quand c'est évident et qu'on comprend
 - En particulier, on pourra faire appel à des algorithmes vus en classe en "boîte noire"
 - Ressemble à du code - mais **parfois** on utilise des phrases pour s'éviter du travail ennuyant.

Compétences à développer/parfaire

- **Visualiser** une exécution en lisant un algorithme.
- **Créer** des algorithmes avec les meilleures complexités possibles.
- **Estimer** rapidement la complexité d'un algorithme.
- **Démontrer** la complexité d'un algorithme.
- **Démontrer** qu'un algorithme est correct (i.e. qu'il fait vraiment ce qu'il prétend).