Algorithmique et structures de données

CM 9 – classes de complexité et complexité des algorithmes de tri

Plan du CM 9

Classification des algorithmes

Complexité d'un algorithme

Classification selon la complexité

Complexité des algorithmes de tri

Plan du CM 9

Classification des algorithmes

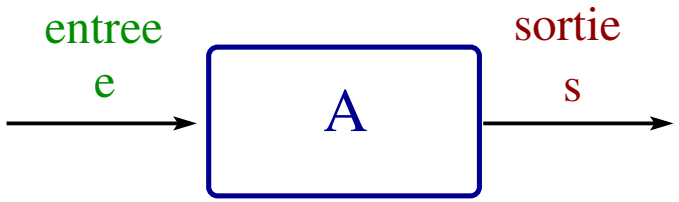
Complexité d'un algorithme

Classification selon la complexité

Complexité des algorithmes de tri

Classification des algorithmes

Schéma général d'un algorithme



- *e* appartient à un ensemble *E* qui est l'ensemble des entrées possibles

Vocabulaire

Les trois termes sont équivalents

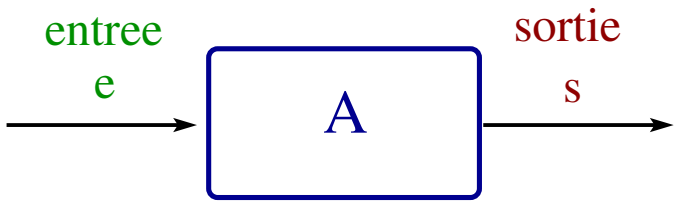
- entrée
- donnée
- instance

Classification

Il existe différents critères pour classifier les algorithmes.

Algorithmes déterministes

Schéma général d'un algorithme



Définition d'un algorithme déterministe

L'algorithme *A* est déterministe si, pour une entrée *e* fixée, il s'exécute toujours de la même façon.

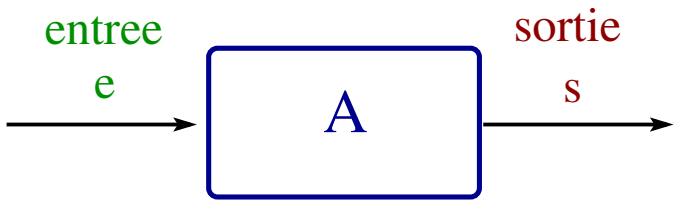
- même résultat : même sortie *s*
- même exécution : on effectue exactement les mêmes instructions

Remarque

- la plupart des algorithmes que l'on définit et utilise sont des algorithmes déterministes
- c'est le cas pour tous les algorithmes étudiés dans ce cours

Algorithmes probabilistes

Schéma général d'un algorithme

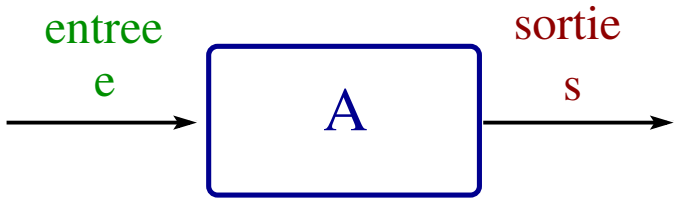


Définition d'un algorithme probabiliste

- l'algorithme A est un algorithme probabiliste lorsqu'il utilise des aléas
- un aléa est une tournure imprévisible que peut prendre un évènement
- exemple : on jette une pièce de monnaie pour savoir si on doit tourner à gauche ou à droite

Algorithme associé à un problème de décision

Schéma général d'un algorithme



s est un booléen : il prend la valeur VRAI ou FAUX.

Problème de décision

- le problème de décision correspond à une propriété que l'entrée e peut avoir ou ne pas avoir.
- la sortie est donc indépendante de l'algorithme et ne dépend que du problème de décision.
- les étapes pour atteindre le résultat peuvent être différentes d'un algorithme à un autre.

Exemple

- entrée $e = (T, x)$, où T est un tableau et x une valeur.
- sortie l'algorithme renvoie VRAI si x apparaît dans T et FAUX sinon.

Autres types de sortie

Même type d'entrée que de sortie

La sortie et l'entrée (ou une partie de l'entrée) peuvent appartenir au même ensemble.
Exemples

- l'entrée et la sortie sont des tableaux
  - exemple : tri d'un tableau
- l'entrée et la sortie sont des listes chaînées
  - exemple : construction de la liste inversée
- l'entrée et la sortie sont des arbres binaires
  - exemple : échange entre A_G et A_D

La sortie est calculée avec les données de l'entrée

- plus grand élément d'un tableau ou d'une liste chaînée
- somme des éléments d'un tableau ou d'une liste chaînée
- nombre de nœuds et hauteur d'un arbre

Plan du CM 8

Classification des algorithmes

Complexité d'un algorithme

Classification selon la complexité

Complexité des algorithmes de tri

Complexité d'un algorithme

Complexité en temps

On s'intéresse au temps mis par l'algorithme A pour renvoyer la sortie s.

Complexité en mémoire ou en espace

On calcule la mémoire allouée pendant l'exécution de l'algorithme.

Calcul indépendant du langage de programmation et de la machine

On souhaite calculer la complexité en temps ou en espace sans tenir compte

- du choix du langage de programmation
- de l'environnement informatique : matériel informatique, système d'exploitation, ...

Nous nous intéresserons souvent d'abord à la complexité en temps.

Complexité en temps
Recherche d'un élément dans un tableau

Nous allons voir comment modéliser la complexité en temps d'un algorithme à partir d'un algorithme très simple, la recherche d'un élément dans un tableau

Algorithme A

```
Entrée e = (T : tableau d'entiers, taille : entier, x : entier)
Sortie s = i : entier, indice de la position de x dans T
ou -1 si x n'appartient pas à T

recherche(T : tableau, taille : entier, x : entier) : entier
   i = 0
   tant i < taille et T[i] <> x faire
      i = i + 1
   si i = taille alors retourner -1
   sinon retourner i
```

Coût de l'algorithme

- Opération élémentaire comparaison entre $T[i]$ et x
- nous noterons $f(A, e)$ le coût de l'algorithme A sur l'entrée e
- $f(A, e)$ sera donc ici est le nombre de comparaisons entre $T[i]$ et x

Complexité dans le pire des cas

Coûts possibles

Posons n la taille du tableau T.

- nous avons $f(A, e) = i$ avec $i \in \{1, \ldots, n\}$
- coût minimum $f_{min}(n) = 1$ l'élément x est en première position
- coût maximum $f_{max}(n) = n$ l'élément x est en dernière position ou n'appartient pas au tableau.

La complexité dans le pire des cas est le coût maximal, elle vaut donc

$$f_{max}(n) = n.$$

Généralement, lorsque ce n'est pas spécifié, on considère la complexité dans le pire des cas lorsque l'on parle de complexité.

Complexité en moyenne

Probabilité sur les entrées

- il existe plusieurs complexités en moyenne
- il faut fixer une distribution de probabilités sur toutes les entrées possibles
- le coût de l'algorithme devient une variable aléatoire C
- la complexité en moyenne est $E[C]$, l'espérance de cette variable aléatoire

Complexité en moyenne

Exemple – distribution uniforme sur les cases du tableau

- on suppose que x appartient au tableau
- et qu'il a la même probabilité de se trouver à toute case du tableau (distribution uniforme ou équiprobabilité ou équirépartition).

Distribution de probabilité

Soit C la variable aléatoire du coût de l'algorithme.
On associe à chaque valeur de la variable aléatoire une probabilité.
Avec la distribution uniforme, il s'agit de la même probabilité.

$$\Pr(C = i) = \frac{1}{n}, \text{ pour tout } i \in \{1, \ldots, n\}.$$

Espérance

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

Complexité en moyenne

On propose maintenant une autre distribution des entrées.

Probabilité sur les entrées

- x a une chance sur deux de ne pas appartenir au tableau
- pour les autres entrées, nous avons la distribution uniforme sur les indices

$$\begin{aligned} \Pr(C = n) &= \frac{1}{2} + \frac{1}{2n} \\ \Pr(C = i) &= \frac{1}{2n}, \text{ pour tout } i \in \{1, \ldots, n-1\}. \end{aligned}$$

Espérance

$$\begin{aligned} E[C] &= \sum_{i=1}^{n} i \Pr(C = i) \\ &= n\,\frac{1}{2} + \sum_{i=1}^{n} i \frac{1}{2n} \\ &= \frac{n}{2} + \frac{n+1}{4} \\ &= \frac{3n+1}{4} \end{aligned}$$

Plan du CM 8

Classification des algorithmes

Complexité d'un algorithme

Classification selon la complexité

Complexité des algorithmes de tri

Classification selon la complexité

Objectifs

- le calcul du coût n'est pas toujours exact (c'est généralement difficile ou impossible)
  - on choisit les opérations que l'on va compter

Comparaison entre les problèmes

- on veut pouvoir comparer deux algorithmes résolvant le même problème
  - quel est le meilleur algorithme ?

Comparaison entre deux algorithmes

- on veut pouvoir comparer deux algorithmes résolvant deux problèmes différents
  - quel est le problème le plus difficile ?

Classification selon la complexité

Méthode de classification

On décide de regrouper les algorithmes par classe.

Algorithmes en temps constant

Le coût de l'algorithme ne dépend pas de la taille de la donnée.

- accès à un élément d'un tableau
- insertion au début dans une liste chaînée

Algorithmes en temps linéaire

Le coût de l'algorithme vaut plusieurs fois la taille de la donnée.
Par exemple, il nécessite de parcourir une ou lusieurs fois l'entrée.

- calcul du nombre de nœuds d'une liste chaînée
- calcul du plus grand élément d'un tableau
- tri pour le drapeau hollandais

Comparaison asymptotique – O (grand o)

Définition mathématique

Soient $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$.
On note $f(n) = O(g(n))$ lorsque

$$\exists k \in \mathbb{N}\ \exists N \in \mathbb{N}\ \forall n \geq N \quad f(n) \leq k\, g(n).$$

Exemple

Prenons les deux fonctions f et g

- $f(n) = 4n + 15$
- $g(n) = 3n$

La majoration de f par g doit être vrai à partir d'un certain entier N.

Ici avec $k = 2$ et $N = 8$, nous avons bien

$$f(n) \leq 2g(n), \text{ pour tout } n \geq 8.$$

Nous pouvons donc écrire $f(n) = O(g(n))$.

Comparaison asymptotique – O (grand o)

Application à la complexité

$f(n)$ pourra représenter

- le coût d'un algorithme A sur une entrée e de taille n
- la complexité de A dans le pire des cas pour des entrées de taille n
- la complexité en moyenne de A pour une distribution de probabilité sur les entrées de taille n

Remarque : la notation est asymptotique

- l'algorithme doit s'appliquer sur des entrées de taille aussi grande que l'on souhaite
- pour des entrées de taille limitée la notation n'a pas de sens, elle ne doit pas être utilisée

Notation O pour la majoration

Majoration du coût

Pour majorer le coût la notation O est très utile. Par exemple, si $f(n)$ est le coût de la recherche d'un élément dans un tableau, nous avons, quelle que soit l'entrée e,

$$f(n) = O(n).$$

Une notation imprécise

Si $f(n) = O(g_1(n))$ et $g_1(n) \leq g_2(n)$ alors $f(n) = O(g_2(n)$.
Par exemple, pour le coût de la recherche d'un élément dans un tableau, nous avons

$$f(n) = O(n),$$

mais nous avons aussi les majorations suivantes

$$\begin{array}{lcl} f(n) & = & O(n^2) \\ f(n) & = & O(n^4) \\ f(n) & = & O(2^n) \end{array}$$

On voit bien que dire dans le cas de la recherche que l'on a $f(n) = O(2^n)$ n'est pas très informatif !

Comparaison asymptotique – Θ (grand theta)

Définition

Soient $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$.
On note $f(n) = \Theta(g(n))$ lorsque

$$f(n) = O(g(n)) \text{ et } g(n) = O(g(n).$$

Cela revient à dire que l'on a

$$\exists k_1 \in \mathbb{N}\ \exists k_2 \in \mathbb{N}\ \exists N \in \mathbb{N}\ \forall n \geq N \quad k_1 g(n) \leq f(n) \leq k_2\ g(n).$$

Relation d'équivalence

On définit la relation binaire $\mathcal{R}$ sur les fonctions de $\mathbb{N} \to \mathbb{N}$ par

$$f \ \ R\ g \text{ lorsque } f(n) = \Theta(g(n)).$$

$\mathcal{R}$ est une relation d'équivalence

- réflexivité $f(n) = \Theta(f(n))$
- symétrie si $f(n) = \Theta(g(n))$ alors $g(n) = \Theta(f(n))$
- transitivé si $f(n) = \Theta(g(n))$ et $g(n) = \Theta(h(n))$ alors $f(n) = \Theta(h(n))$

Θ – Terme dominant

La classe d'une fonction $f(n)$ dépend de son terme dominant.

Classe linéaire

Le fonctions suivantes appartiennent toutes à la classe linéaire :

- $f(n) = 2n + 10$
- $f(n) = \frac{n}{3}$
- $f(n) = n + 3 \ln n + 7$
- $f(n) = 3n + \sqrt{n} - 3 \ln n$

Classe quadratique

Le fonctions suivantes appartiennent toutes à la classe quadratique :

- $f(n) = n^2 - 10n$
- $f(n) = 2n^2 + \frac{n}{3}$
- $f(n) = 3n^2 + 4n \ln n + 3n + 6 \ln n$
- $f(n) = n^2 + n^{\frac{3}{4}}$

Complexité d'un problème $\neq$ complexité d'un algorithme

Complexité d'un problème

La complexité d'un problème est une fonction $f(n)$ telle que pour tout algorithme A résolvant ce problème et $g(n)$ la complexité de cet algorithme, nous avons

$$f(n) \leq g(n).$$

Observations

- $f(n)$ n'est pas toujours connu
- on ne sait pas toujours s'il n'existe pas un meilleur algorithme (de complexité plus petite) que les algorithmes connus
- la définition s'applique aussi bien pour la complexité dans le pire des cas et en moyenne

Complexité d'un problème $\neq$ complexité d'un algorithme

Complexité d'un problème

La complexité d'un problème est une fonction $f(n)$ telle que pour tout algorithme A résolvant ce problème et $g(n)$ la complexité de cet algorithme, nous avons

$$f(n) \leq g(n).$$

Complexité d'un algorithme

La complexité d'un algorithme A résolvant un problème donne une majoration de la complexité du problème.

- si $g(n)$ est la complexité de A alors $f(n) = O(g(n))$
- si la complexité de A est en $O(g(n))$ alors $f(n) = O(g(n))$

Plan du CM 8

Classification des algorithmes

Complexité d'un algorithme

Classification selon la complexité

Complexité des algorithmes de tri

Algorithmes de tri

Problème

```
Entrée T tableau de n entiers
Sortie T ou T' tableau trié contenant ces n entiers
```

Si la sortie est T, on dit que le tri s'effectue sur place.

Coût de l'algorithme

On compte le nombre de comparaisons entre deux éléments.

Beaucoup d'algorithmes de tri

Il existe beaucoup d'algorithmes de tri.

- sont-ils tous équivalents ?
- connaît-on la complexité du tri ?

Algorithmes de tri

Complexité en moyenne

Si ce n'est pas précisé, on choisira le modèle suivant :

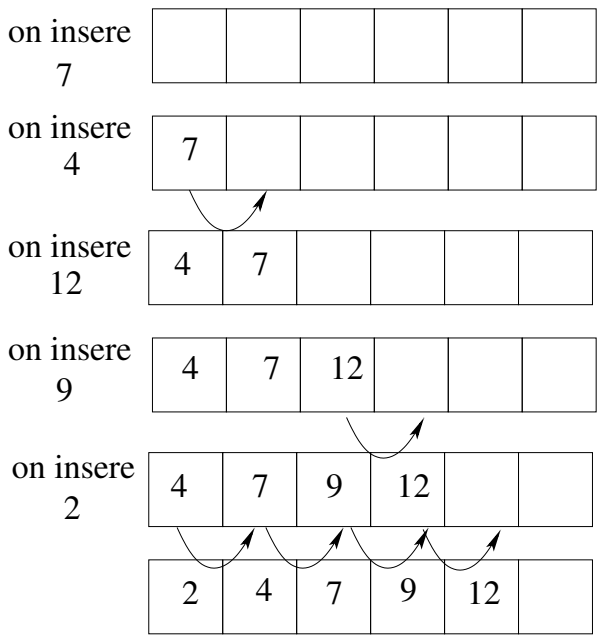
- les n entiers sont $[n] = \{1, \ldots, n\}$ (pas de doublons)
- $n!$ entrées possibles (nombre de permutations sur $[n]$)
- $\Pr(\text{on tire l'entrée } e) = \frac{1}{n!}$, équiprobabilité ou équirépartition ou distribution uniforme

Tri par insertion

Principe

- n étapes, à l'étape i on insère le ième élément
- on insère les éléments un par un en les mettant à chaque fois à la bonne place
- si l'élément est inséré à la place i, on décale à droite tous les éléments en position j avec $j \geq i$

Exemple



Tri par insertion

Complexités

- complexité dans le meilleur des cas
  - on insère toujours en première position, 1 comparaison à chaque étape
  - complexité $0 + 1 + \ldots + 1 = n - 1$

- complexité dans le pire des cas
  - on insère toujours en dernière position, $k - 1$ comparaisons à l'étape k
  - complexité : $0 + 1 + 2 + \ldots + n - 1 = \frac{n(n-1)}{2}$

- complexité en moyenne (distribution uniforme)
  - en moyenne $\frac{k}{2}$ comparaisons à l'étape k
  - complexité : $0 + \frac{1}{2} + \ldots + \frac{n-1}{2} = \frac{n(n-1)}{4}$

La complexité dans le pire des cas et en moyenne est en $\Theta(n^2)$.

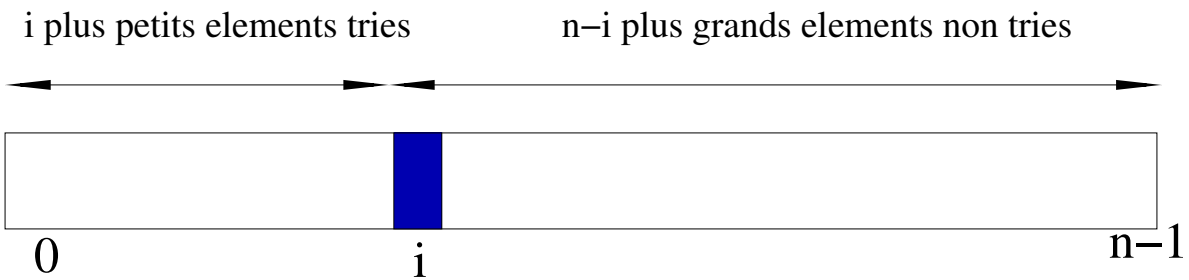
Exercice

Recalculer les différentes complexités en comptant cette fois-ci le nombre de comparaisons + le nombre de décalages pour le coût.

Tri par sélection

Principe

- rechercher le plus petit élément et le placer en tête (indice 0), puis recommencer à partir du second élément pour rechercher le deuxième plus petit élément et le placer en second (indice 1).
- après avoir placé l'élément d'indice $i - 1$, la situation est la suivante



- pour sélectionner le ième élément, on recherche l'indice du plus petit élément parmi ceux d'indice i à $n - 1$.
  On échange ensuite cet élément avec l'élément d'indice i.

Tri par sélection

Complexité

- le coût est toujours le même quelle que soit l'entrée
- la sélection du ième élément nécessite $n - i$ comparaisons
- soit un coût de $n - 1 + n - 2 + \ldots + 1 = \frac{n(n-1)}{2}$

La complexité est donc en $\Theta(n^2)$.

Tri rapide (quickSort)

Placement du pivot (procédure *partition(T,debut,fin)*)

- à la première étape debut = 0 et fin = $n - 1$
- on fixe un pivot entre les positions debut et fin (généralement l'élément en position debut)
- on met à gauche du pivot tous les éléments plus petits que le pivot
- on met à droite tous les éléments plus grands que le pivot
- le pivot se retrouve à la bonne position (notée positionPivot)
- les instructions s'effectue sur place (par des échanges entre deux éléments)

Appels récursifs

On effectue deux appels résursifs :

- à gauche, debut = debut et fin = positionPivot - 1
- à droite, debut = positionPivot + 1 et fin = fin
- on arrête les appels récursifs lorsque debut > fin

Tri rapide (quickSort)

Procédure quickSort

```
quickSort(T : tableau d'entiers,debut : entier,fin : entier)
   si debut < fin alors
       positionPivot=partition(T,debut,fin)
       quickSort(T,debut,positionPivot-1)
       quickSort(T,positionPivot+1,fin)
```

Procédure partition

La coût de la procédure *partition* est de fin-debut.
En effet, le pivot est comparé avec les fin-debut autres éléments entre les positions debut et fin.
(voir le TP 4 sur les algorithmes de tri)

Tri rapide (quickSort)

Complexités

- complexité dans le meilleur des cas
  - le pivot est toujours en position $\frac{fin - debut}{2}$ pour $n = 2^k - 1$
  - la complexité est $\approx n \log_2 n$
- complexité dans le pire des cas
  - le tableau est déjà trié, le pivot est alors toujours en position debut
  - la complexité vaut $n - 1 + n - 2 + \ldots + 1 = \frac{n(n-1)}{2}$
- complexité en moyenne (distribution uniforme)
  - nous devons considérer toutes les partitions possibles
  - la complexité vaut $\approx 1.39 n \log_2 n$.

La complexité dans le pire des cas et en moyenne est donc en $\Theta(n \log n)$.

Tri rapide à pivot aléatoire (randomQuickSort)

Choix du pivot

- le pivot est tiré aléatoirement entre les positions debut et fin.
- chaque position a donc une chance $\frac{1}{fin - debut + 1}$ d'être tirée

Nouvelle complexité en moyenne

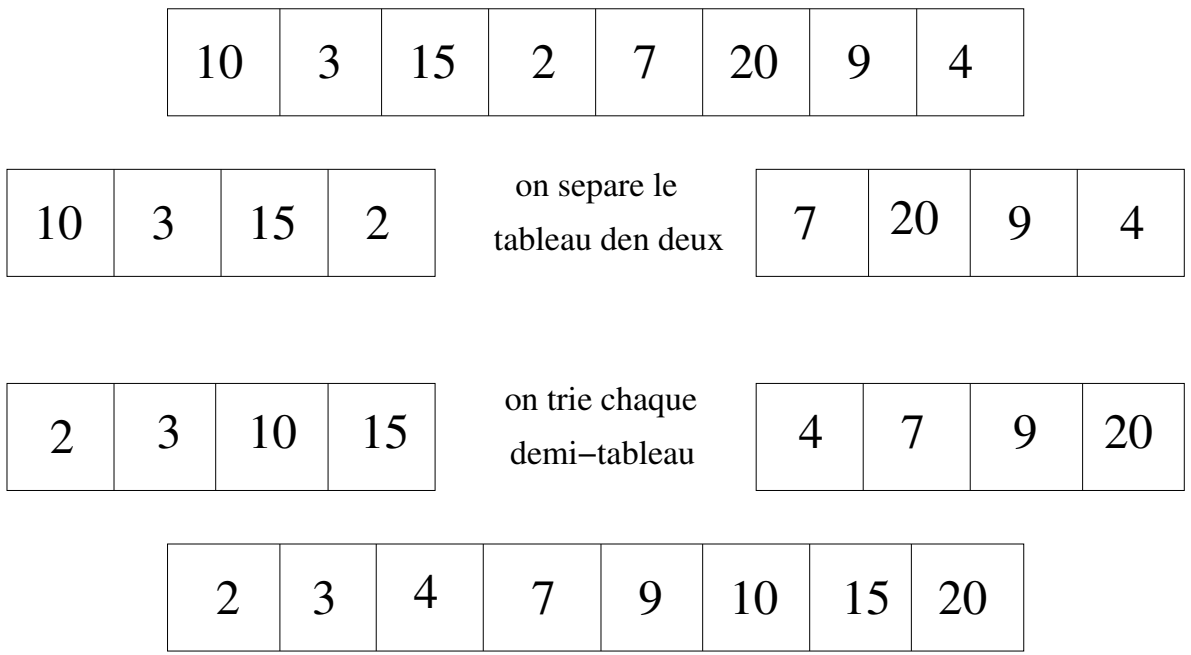
- on fixe l'entrée, c'est le choix du pivot qui change
- plus de meilleur des cas et pire des cas
- la complexité est en $\Theta(n \log n)$
- le calcul est différent, il faut considérer tous les choix de pivot possibles

Tri fusion

Principe

- on sépare le tableau en deux
- on trie chaque partie du tableau
- on fusionne les deux parties

Procédure triFusion



Tri fusion

Principe

- on sépare le tableau en deux
- on trie chaque partie du tableau
- on fusionne les deux parties

Procédure triFusion

```
triFusion(T : tableau d'entiers, debut : entier, fin : entier)
    si debut < fin alors
        milieu = partie entière de (debut+fin)/2
        T1 = triFusion(T, debut, milieu)
        T2 = triFusion(T, milieu+1, fin)
        fusion(T, T1, T2)
```

Tri fusion

Procédure fusion

- c'est cette procédure qui donne la complexité de l'algorithme
- son coût vaut fin-debut+1
- déjà étudiée : voir TD 3, exercice 2

Complexités

- complexité dans le meilleur des cas
  - avec $n = 2^k$, on divise à chaque fois en deux parties égales
  - la complexité est $\approx n \log_2 n$ (même récurrence que pour quickSort)
- complexité en moyenne
  - la complexité est en $\Theta(n \log n)$.
- pas de pire des cas

Autres algorithmes de tri

Tri à bulles (Bubble sort)

- on permute successivement les éléments consécutifs d'un tableau
- comme des bulles d'air qui remontent à la surface
- pas efficace, complexité dans le pire des cas et en moyenne en $\Theta(n^2)$

Tri par tas (Heap sort)

- le tri par tas code un arbre binaire avec un tableau.
- les éléments sont partiellement ordonnés par priorité
- sa complexité dans le pire des cas et en moyenne est en $\Theta(n \log n)$.

Timsort

- algorithme utilisé par Python *maListe.sort*()
- mélange entre tri insertion et tri fusion
- repère si des parties sont déjà triées

Variantes

- on modifie les conditions terminales
- on change d'algorithme quand on a peu d'éléments

Complexité du problème

Peut-on trouver un meilleur algorithme

On peut montrer que les meilleurs algorithmes de tri sont en $\Theta(n \log g)$.

Idée de la preuve

- on fixe un algorithme de tri A
- on construit un arbre où chaque branche de l'arbre (nœuds allant de la racine jusqu'à une feuille)correspond à l'exécution de A sur une entrée.
- chaque nœud correspond à une comparaison entre deux éléments
- nous avons $n!$ entrées possibles donc au moins $n! = \Theta(n^{n+1/2}e^{-n})$ feuilles
- la complexité de A est la hauteur h de l'arbre
- on montre que la hauteur est au mieux en $\Theta(n \log n)$

Algorithmes de tri – récapitulatif

Comparaisons entre les algorithmes

- algorithmes lents en $\Theta(n^2)$
- algorithmes rapides en $\Theta(n \log n)$
- la complexité en moyenne est plus importante que celle dans le pire des cas
- il peut y avoir plusieurs complexités en moyenne (listes triées en partie, doublons)

Tableau récapitulatif

Nom de l'algorithme	complexité dans le pire des cas	complexité en moyenne
Tri sélection	$\Theta(n^2)$	$\Theta(n^2)$
Tri insertion	$\Theta(n^2)$	$\Theta(n^2)$
Tri bulle	$\Theta(n^2)$	$\Theta(n^2)$
QuickSort	$\Theta(n^2)$	$\Theta(n \log n)$
Random quickSort	$\Theta(n \log n)$	$\Theta(n \log n)$
Tri fusion	$\Theta(n \log n)$	$\Theta(n \log n)$
Tri par tas	$\Theta(n \log n)$	$\Theta(n \log n)$
Timsort	$\Theta(n \log n)$	$\Theta(n \log n)$

Ordre de complexité des algorithmes

$ns = 10^{-9}s$, nanoseconde, $\mu s = 10^{-6}s$, micro seconde.

Ordres de grandeur et taille des entrées

ordre de grandeur	nom de la classe	exemple d'algorithme	$n = 10$	$n = 50$	$n = 250$	$n = 10^3$	$n = 10^4$	$n = 10^5$
$\Theta(1)$	constant	accès dans un tableau	$10ns$	$10ns$	$10ns$	$10ns$	$10ns$	$10ns$
$\Theta(\log n)$	logarithmique	recherche dichotomique	$10ns$	$20ns$	$30ns$	$30ns$	$40ns$	$60ns$
$\Theta(n)$	linéaire	parcours d'une liste	$100ns$	$500ns$	$2.5\mu s$	$10\mu s$	$100\mu s$	$10ms$
$\theta(n \log n)$	quasi-linéaire	tri rapide	$100ns$	$850ns$	$6\mu s$	$30\mu s$	$400\mu s$	$60ms$
$\Theta(n^2)$	quadratique	tri lent	$1\mu s$	$25\mu s$	$625\mu s$	$10ms$	$1s$	$2.8h$
$\Theta(n^3)$	cubique	multiplication matricielle naïve	$10\mu s$	$1.25ms$	$156ms$	$10s$	$2.7h$	316 ans
$\Theta(n^k)$	polynomial	test de primalité						
$\Theta(2^n)$	exponentiel	SAT	$10\mu s$	130 jours	10^{59} ans			