Structure de Données
Itérations et Tris

Marie Pelleau
marie.pelleau@univ-cotedazur.fr

Semestre 3

Itérations

- En informatique, l'itération est la répétition d'un processus par un programme
- L'itération peut être utilisée
  - comme un terme général synonyme de répétition
  - pour décrire une forme spécifique de répétition qui modifie l'état courant
- Dans le sens premier, la récursion est une forme particulière d'itération. Cependant la notation récursive est bien différente de la notion itérative
- En mathématiques, l'itération est très couramment utilisée : $\sum$ pour la somme, $\prod$ pour le produit

Plan

Indice du plus petit élément

- On recherche l'indice du plus petit élément d'un tableau
- A partir de l'indice i, on retrouve facilement la valeur de i puisque c'est $T[i]$
- On balaie simplement tous les éléments du tableau et on mémorise le plus petit

Indice du plus petit élément

Itérations avec pour

```
entier indicePlusPetit(T[], n) {
  j <- 1
  pour (i de 1 à n) {
    si (T[i] < T[j]) {
      j <- i
    }
  }
  retourner j
}
```

Itérations avec tant que

```
entier indicePlusPetit(T[], n) {
  j <- 1; i <- 1
  tant que (i ≤ n) {
    si (T[i] < T[j]) {
      j <- i
    }
    i <- i + 1
  }
  retourner j
}
```

Plan

Recherche de début à fin

- Au lieu de parcourir tous les éléments du tableau, on veut limiter la portion du tableau considéré (on considère un sous-tableau)
- On ne balaie plus tous les éléments mais ceux entre l'indice d (début) et l'indice f (fin) (les deux indices inclus)

Indice du plus petit element

Itérations avec pour

```
entier indicePlusPetit(T[], n, d
    , f) {
  j <- d
  pour (i de d à f) {
    si (T[i] < T[j]) {
      j <- i
    }
  }
  retourner j
}
```

Itérations avec tant que

```
entier indicePlusPetit(T[], n, d
    , f) {
  j <- d; i <- d
  tant que (i ≤ f) {
    si (T[i] < T[j]) {
      j <- i
    }
    i <- i + 1
  }
  retourner j
}
```

Recherche de début à condition

- On introduit maintenant explicitement une condition d'arrêt
  - ✗ Ce n'est plus "on s'arrête à tel indice",
  - ✓ mais "on s'arrête lorsqu'une condition est satisfaite"
- Par exemple T[i] vaut -1
- Cette valeur ne doit pas être prise en compte
- Elle n'est pas forcément dans le tableau

Recherche de début à condition

Itérations avec pour

```
entier indicePlusPetit(T[], n, d
    , val) {
  j <- d
  pour (i de d à ...) {
    C'est compliqué
    ...
```

Itérations avec tant que

```
entier indicePlusPetit(T[], n, d
    , val) {
  j <- d; i <- d
  tant que (i≤n et T[i]≠val) {
    si (T[i] < T[j]) {
      j <- i
    }
    i <- i + 1
  }
  retourner j
}
```

Attention

- ✓ $i \leq n$ et $T[i] \neq val$
- ✗ $i \leq n$ ou $T[i] \neq val$
- ✗ $T[i] \neq val$ et $i \leq n$

Recherche de début à condition

Itérations avec tant que

```
entier indicePlusPetit(T[], n, d, val) {
  si (T[d] = val) {
    retourner -1
  }
  j <- d; i <- d
  tant que (i ≤ n et T[i] ≠ val) {
    si (T[i] < T[j]) {
      j <- i
    }
    i <- i + 1
  }
  retourner j
}
```

Recherche de début à condition

Itérations avec répéter

```
entier indicePlusPetit(T[], n, d, val) {
  si (T[d] = val) {
    retourner -1
  }
  j <- d; i <- d
  répéter {
    si (T[i] < T[j]) {
      j <- i
    }
    i <- i + 1
  } jusqu'à (i > n ou T[i] = val)
  retourner j
}
```

Cas particulier pour le début et la fin

Assez souvent il est quasiment obligatoire de faire un traitement avant ou après la boucle, autrement dit pour le premier ou le dernier élément

Example

- On écrit son nom avec un point entre les lettres consécutives : DUPONT donne D.U.P.O.N.T
- Il n'y a pas de point après la dernière lettre

Cas particulier pour le début et la fin

```
afficherAvecPoints(T[], n) {
   pour (i de 1 à n-1) {
      afficher(T[i])
      afficher('.')
   }
   afficher(T[n])
}
```

```
afficherAvecPoints(T[], n) {
   afficher(T[1])
   pour (i de 2 à n) {
      afficher('.')
      afficher(T[i])
   }
}
```

Plan

Pas seulement une opération

Ce qui est fait dans la boucle peut être complexe et dépendre des itérations précédentes

Example

- Un tableau représente un nombre en binaire
- On veut faire la somme de 2 tableaux (même taille) en tenant compte de la retenue

$$\begin{array}{rl} & [0\ 0\ 1\ 1\ 0\ 1\ 1\ 1] \quad (T1) \\ + & [1\ 0\ 1\ 0\ 0\ 0\ 1\ 1] \quad (T2) \\ \hline & [1\ 1\ 0\ 1\ 1\ 0\ 1\ 0] \end{array}$$

Pas seulement une opération

```
somme(T1[], T2[], R[], n) {
  retenue <- 0
  pour (i de n à 1) {
    s <- T1[i] + T2[i] + retenue
    si (s = 0) {
      R[i] <- 0
      retenue <- 0
    } si (s = 1) {
      R[i] <- 1
      retenue <- 0
    } si (s = 2) {
      R[i] <- 0
      retenue <- 1
    } si (s = 3) {
      R[i] <- 1
      retenue <- 1
    }
  }
}
```

Plan

Une boucle et deux indices

- Deux mots m et r
- Déterminer si les lettres de m se trouvent dans le mot r dans le même ordre

✓ m = "arme" et r = "algorithme"

✗ m = "rame" et r = "algorithme"

- m est composé de p lettres et r de n lettres

```
booléen memeOrdre(m[], p, r[], n) {
  i <- 1; k <- 1 // i dans m et k dans r
  tant que (i ≤ p et k ≤ n) {
    si (m[i] = r[k]) {
      i <- i + 1
    }
    k <- k + 1
  }
  si (i > p) { // retourner (i > p)
    retourner vrai
  }
  retourner faux
}
```

Plan

Boucles imbriquées

Calcul du nombre d'éléments à zéro dans une matrice M(l,c) : l lignes et c colonnes

```
entier numZeroLig(M[][], i, c) {
  cpt <- 0
  pour (j de 1 à c) {
    si (M[i][j] = 0) {
      cpt <- cpt + 1
    }
  }
  retourner cpt
}

entier numZero(M[][], l, c) {
  cpt <- 0
  pour (i de 1 à l) {
    cpt <- cpt + numZeroLig(M, i, c)
  }
  retourner cpt
}
```

Boucles imbriquées

Affichage d'une matrice linéarisée ($T[(i-1) \times \#col + j] = M[i][j]$)

```
afficherMatrice(T[], l, c) {
  pour (i de 1 à l) {
    pour (j de 1 à c) {
      afficher(T[(i-1)*c + j])
      afficher(' ')
    }
  }
  afficher('\n')
}
```

Boucles imbriquées

Multiplication de deux matrices de même dimension

$$\forall i, j, : c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}$$

```
multMatrice(M1[][], M2[][], n, R[][]) {
  pour (i de 1 à n) {
    pour (j de 1 à n) {
      s <- 0
      pour (k de 1 à n) {
        s <- s + M1[i][k]*M2[k][j]
      }
      R[i][j] <- s
    }
  }
}
```

Plan

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
- Une structure de données ne regroupe pas nécessairement des objets du même type

Itérateur

- Un **itérateur** est un objet qui permet de parcourir tous les éléments contenus dans un autre objet, le plus souvent un conteneur (liste, arbre, etc). Un synonyme d'itérateur est **curseur**, notamment dans le contexte des bases de données
- Un itérateur dispose essentiellement de deux primitives :
  - Accéder à l'élément en cours dans le conteneur
  - Se déplacer vers l'élément suivant
- Il faut aussi pouvoir créer un itérateur sur le premier élément ; ainsi que déterminer à tout moment si l'itérateur a épuisé la totalité des éléments du conteneur. Diverses implémentations peuvent également offrir des comportements supplémentaires

Itérateur

But d'un itérateur

- Permettre à son utilisateur de parcourir le conteneur, c'est-à-dire d'accéder séquentiellement à tous ses éléments pour leur appliquer un traitement
- Isoler l'utilisateur de la structure interne du conteneur, potentiellement complexe

Avantage

- le conteneur peut stocker les éléments de la façon qu'il veut, tout en permettant à l'utilisateur de le traiter comme une simple liste d'éléments

Le plus souvent l'itérateur est conçu en même temps que la classe-conteneur qu'il devra parcourir, et ce sera le conteneur lui-même qui créera et distribuera les itérateurs pour accéder à ses éléments

Itérateur

On utilise souvent un index dans une simple boucle, pour accéder séquentiellement à tous les éléments, notamment d'un tableau; l'utilisation des itérateurs a certains avantages :

- Un simple compteur dans une boucle n'est pas adapté à toutes les structures de données, en particulier
  - celles qui n'ont pas de méthode d'accès à un élément quelconque
  - celles dont l'accès à un élément quelconque est très lent (c'est le cas des listes chaînées et des arbres)
- Les itérateurs fournissent un moyen cohérent d'itérer sur toutes sortes de structures de données, rendant ainsi le code client plus lisible, réutilisable, et robuste même en cas de changement dans l'organisation de la structure de données
- Un itérateur peut implanter des restrictions additionnelles sur l'accès aux éléments, par exemple pour empêcher qu'un élément soit "sauté", ou qu'un même élément soit visité deux fois

Itérateur

- Un itérateur peut dans certains cas permettre que le conteneur soit modifié, sans être invalidé pour autant
- Par exemple, après qu'un itérateur s'est positionné derrière le premier élément, il est possible d'insérer d'autres éléments au début du conteneur avec des résultats prévisibles
- Avec un index on aurait plus de problèmes, parce que la valeur de l'index devrait elle aussi être modifiée en conséquence
- Important : il est indispensable de bien consulter la documentation d'un itérateur pour savoir dans quels cas il est invalidé ou non

Itérateur

- Cela permet de faire des algorithmes sans connaître la structure de données sous-jacente
- On recherche un plus court chemin dans un graphe :
  - On ne sait pas comment le graphe est représenté
  - Cela ne nous empêche pas de faire un algorithme efficace

Plan

Permutation

Exemple

Proposer un algorithme qui permet de savoir si les éléments d'un tableau T forment une permutation de $1 \dots n$

- on crée un tableau P de 1 à n
- on met 0 dans chaque case
- on traverse T et pour la valeur $T[i]$ on met un 1 dans la case $P[T[i]]$
- à la fin si aucune case ne vaut 0 alors T est une permutation de P

Permutation

```
booléen estPermutation(T[], n) {
  pour (i de 1 à n) {
    P[i] <- 0
  }
  pour (i de 1 à n) {
    P[T[i]] <- 1
  }
  ok <- vrai; i <- 1
  tant que (i ≤ n et ok) {
    si (P[i] = 0) {
      ok <- faux
    }
    i <- i + 1
  }
  retourner ok
}
```

Tri par comptage

- Quand on a des nombres de 1 à m, on peut les trier facilement en comptant le nombre d'occurrences de chaque nombre
- On crée un tableau de m valeurs
- On met toutes ces valeurs à 0
- On traverse T. On compte le nombre de fois où $T[i]$ est pris (on incrémente $P[T[i]]$)
- Ensuite on balaie le tableau P et on copie autant de fois une valeur qu'elle apparait dans P

Example

- T : [2, 3, 1, 5, 6, 10, 25, 2, 15]
- P : [1, 2, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
- T trié : [1, 2, 2, 3, 5, 6, 10, 15, 25]

Tri par comptage

```
booléen tri(T[], n, m) {
  pour (k de 1 à m) {
    P[k] <- 0
  }
  pour (i de 1 à n) {
    P[T[i]] <- P[T[i]] + 1
  }
  i <- 1
  pour (k de 1 à m) {
    pour (j de 1 à P[k]) {
      T[i] <- k
      i <- i + 1
    }
  }
}
```

Tri par comptage

```
booléen tri(T[], n) {
  min <- T[1]; max <- T[1]
  pour (i de 2 à n) {
    si (T[i] < min) {
      min <- T[i]
    } si (T[i] > max) {
      max <- T[i]
    }
  }
  pour (k de 1 à (max-min+1)) {
    P[k] <- 0
  }
  pour (i de 1 à n) {
    P[T[i]-min+1] <- P[T[i]-min+1] + 1
  }
  i <- 1
  pour (k de 1 à (max-min+1)) {
    pour (j de 1 à P[k]) {
      T[i] <- k + min - 1
      i <- i + 1
    }
  }
}
```

Tri par comptage

Complexité ?

- La complexité de ce tri est de $O(n + p)$
- Si p est de l'ordre de n alors le tri est linéaire, mais si $p >> n$?
  - $n = 100$ et $p = 10000$: c'est quadratique
  - $n = 100$ et $p = 1000000$: c'est cubique

Tri d'indices

- Que se passe-t-il si on veut trier des indices et non pas les valeurs ?
  - T : [3, 2, 4 ,5, 3 ,7 ,5]
  - T trié : [2, 3, 3, 4, 5 ,5, 7]
  - Indices de T triés : [3, 1, 4, 6, 2, 7, 5]

Tri par comptage

- On va procéder par accumulation

```
pour (k de 1 à m) {
  P[k] <- 0
}
pour (i de 1 à n) {
  P[T[i]] <- P[T[i]] + 1
}
```

- On va maintenant accumuler la position dans P

```
pour (k de 1 à m) {
  P[k + 1] <- P[k + 1] + P[k]
}
```

$P[i]$ contient maintenant le nombre d'éléments qui sont plus petits ou égaux à i

- ```
  pour (i de 1 à n) {
    rang[i] <- P[T[i]]
    P[T[i]] <- P[T[i]] - 1
  }
  ```
  ```

- Le tri par base (ou tri radix ou radix sort) est un algorithme de tri rapide qui suppose que les éléments à trier sont des nombres ou des chaînes de caractères
- Cet algorithme était utilisé pour trier des cartes perforées en plusieurs passages

Exemple

On veut trier des cartes à jouer

- On fait 4 tas : un pour les ♦, un pour les ♣, un pour les ♥ et un pour les ♠
- Puis on trie les cartes de chaque couleur par un tri par comptage

Exemple

On veut trier des noms de famille, des idées ?

Tri par base

- L'ordre de tri est typiquement le suivant :
  - les clefs courtes viennent avant les clefs longues
  - les clefs de même taille sont triées selon un ordre lexical
- Cette méthode correspond à l'ordre naturel des nombres s'ils sont représentés par des chaînes de chiffres
- Son mode opératoire est :
  - prend le chiffre (ou groupe de bits) le moins significatif de chaque clef,
  - trie la liste des éléments selon ce chiffre, mais conserve l'ordre des éléments ayant le même chiffre (ce qui est un tri stable)
  - répète le tri avec chaque chiffre plus significatif

Tri par base

Trier la liste : 170, 45, 75, 90, 2, 24, 802, 66

- tri par le chiffre le moins significatif (unités) :

  170, 90, 2, 802, 24, 45, 75, 66
- tri par le chiffre suivant (dizaines) :

  2, 802, 24, 45, 66, 170, 75, 90
- tri par le chiffre le plus significatif (centaines) :

  2, 24, 45, 66, 75, 90, 170, 802

Tri par base

```
pour (i de 1 à d)
    Utiliser un tri stable pour trier le tableau T sur
        le digit i
```

- digit i dans la base b du nombre n
  - $digit(i, b, n) = [n/b^{(i-1)}]$ modulo b
  - $digit(2, 10, 324) =$
  - $digit(4, 10, 31245) =$

Tri par insertion

- Dans l'algorithme, on parcourt le tableau à trier du début à la fin. Au moment où on considère le $i^{\text{ème}}$ élément, les éléments qui le précèdent sont déjà triés
- L'objectif d'une étape est d'insérer le $i^{\text{ème}}$ élément à sa place parmi ceux qui précèdent
  - Trouver où l'élément doit être inséré en le comparant aux autres
  - Décaler les éléments afin de pouvoir effectuer l'insertion
- Le tri par insertion est
  - Un tri stable (conservant l'ordre d'apparition des éléments égaux)
  - Un tri en place (il n'utilise pas de tableau auxiliaire)

Tri par insertion

```
tri_insertion(T[], n) {
  pour (i de 2 à n) {
    x <- T[i]
    j <- i
    tant que (j > 1 et T[j - 1] > x) {
      T[j] <- T[j - 1]
      j <- j - 1
    }
    T[j] <- x
  }
}
```

Tri par insertion

Voici les étapes de l'exécution du tri par insertion sur le tableau $T = [9, 6, 1, 4, 8]$. Le tableau est représenté au début et à la fin de chaque itération

- $i = 2$, [9, 6, 1, 4, 8], [6, 9, 1 ,4, 8]
- $i = 3$, [6, 9, 1 ,4, 8], [1, 6, 9, 4, 8]
- $i = 4$, [1, 6, 9, 4, 8], [1, 4, 6, 9, 8]
- $i = 5$, [1, 4, 6, 9, 8], [1, 4, 6, 8, 9]

Invariant

À l'étape i : Le tableau est une permutation et les i premiers éléments sont triés

Tri par insertion

Avantages

- Implémentation simple
- Efficace pour les petits ensemble de données
- Efficace pour les ensembles de données qui sont presque déjà triés : complexité $O(n + d)$, où d est le nombre d'inversions (**si tableau presque trié alors très rapide**)
- Le meilleur des cas est linéaire
- Stable
- En place
- Au vol (online) : on peut trier une liste comme on la reçoit

Inconvénients

- Pire des cas quadratique
- Fait beaucoup d'échanges (décalages)

Tri fusion

L'algorithme peut être décrit récursivement

- On découpe en deux parties à peu près égales les données à trier
- On trie les données de chaque partie
- On fusionne les deux parties

La récursivité s'arrête car on finit par arriver à des listes composées d'un seul élément et le tri est alors trivial

Complexité

$O(n \log(n))$

- Chaque niveau procède à des fusions dont le coût total est $O(n)$
- La profondeur maximale est $\log(n)$

Tri fusion

Avantages

- Très bonne complexité
- Toujours au pire $O(n \log(n))$

- Pénible à écrire pour des tableaux

$\Rightarrow$ On peut utiliser d'autres tris par moment

Tri par selection

Sur un tableau de n éléments (numérotés de 1 à n), le principe du tri par sélection est le suivant :

- Rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 1
- Rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 2
- Continuer de cette façon jusqu'à ce que le tableau soit entièrement trié

Tri sur place (les éléments sont triés dans la structure)

Tri par Sélection

```
tri_selection(T[], n) {
  pour (i de 1 à n-1) {
    min <- i
    pour (j de i + 1 à n) {
      si (T[j] < T[min]) {
        min <- j
      }
    }
    si (min ≠ i) {
      échanger(T[i], T[min])
    }
  }
}
```

Tri par Sélection

Invariant

À la fin de l'étape i, le tableau est une permutation du tableau initial et les i premiers éléments du tableau coïncident avec les i premiers éléments du tableau trié

Complexité

- Dans tous les cas, pour trier n éléments, le tri par sélection effectue $n(n-1)/2$ comparaisons. Sa complexité est donc $O(n^2)$. De ce point de vue, il est inefficace. Il est même moins bon que le tri par insertion
- Par contre, le tri par sélection n'effectue que peu d'échanges :
  - $n-1$ échanges dans le pire cas, qui est atteint par exemple lorsqu'on trie la séquence $2, 3, \ldots, n, 1$;
  - aucun si l'entrée est déjà triée
- Ce tri est donc intéressant lorsque les éléments sont aisément comparables, mais coûteux à déplacer dans la structure

Tri par tas

Tri par Sélection et Tri par Tas

- Le tri par tas fonctionne comme le tri par sélection mais utilise une structure de données particulière pour accélérer la recherche : un tas

Tas : structure de données

Un tas descendant est un arbre binaire vérifiant les propriétés suivantes

- La différence maximale de profondeur entre deux feuilles est de 1 (i.e. toutes les feuilles se trouvent sur la dernière ou sur l'avant-dernière ligne)
- Les feuilles de profondeur maximale sont "tassées" sur la gauche
- Chaque noeud est de valeur inférieure à celle de ces deux fils

- Un tas ou un arbre binaire presque complet peut être stocké dans un tableau, en posant que les deux descendants de l'élément d'indice n sont les éléments d'indices $2n$ et $2n+1$ (pour un tableau indicé à partir de 1)
- En d'autres termes, les noeuds de l'arbre sont placés dans le tableau ligne par ligne, chaque ligne étant décrite de gauche à droite

Tas : structure de données

Un tas est **descendant** si chaque noeud est de valeur **inférieure** à celle de ces deux fils

Propriétés

- Chaque chemin est croissant
- Le père est le minimum

Un tas est **ascendant** si chaque noeud est de valeur **supérieure** à celle de ces deux fils

Propriétés

- Chaque chemin est décroissant
- Le père est le maximum

Tas : insertion

L'insertion d'un élément dans un tas descendant se fait de la façon suivante

- On place l'élément sur la première case libre
- On échange l'élément et son père tant que ce dernier est supérieur et qu'il existe

Tas : insertion

```
inserer(T[], n, d, elt) {
  T[d] <- elt
  père <- d/2
  fils <- d
  tant que (père > 0 et T[père] > T[fils]) {
    échanger(T[père], T[fils]) {
      fils <- père
      père <- père/2
    }
  }
}
```

Tas : extraction

Tamisage

L'opération de **tamisage** consiste à échanger la racine avec le plus petit de ses fils (si elle est plus grande), et ainsi de suite récursivement jusqu'à ce qu'elle soit à sa place

Extraction

L'extraction consiste à

- Supprimer la racine
- Mettre à sa place le dernier élément du tas
- Tamiser le tas

Tas : tamisage et extraction

```
tamiser(T[], n) {
  père <- 1
  fils <- 2*père
  fini <- faux
  tant que (fils ≤ n et non fini) {
    si (fils < n et T[fils + 1] < T[fils]) {
      fils <- fils +1
    }
    si (T[père] > T[fils]) {
      échanger(T[père],T[fils])
      père <- fils
      fils <- 2*père
    } sinon {
      fini <- vrai
    }
  }
}
```

Tas : tamisage et extraction

```
entier extraire(T[], n) {
  res <- T[1]
  échanger(T[1], T[n])
  tamiser(T, n-1)
  retourner res
}
```

Tri par tas

- On insère tous les éléments dans le tableau
- On extrait successivement toutes les racines

Tas

- Le tri par tas a certains défauts mais la structure de tas est très pratique pour répondre à d'autres problèmes