1.1

Algorithmique et Structures de Données

Jean-Charles Régin

Licence Informatique 2ème année

1.2

Itérations

Jean-Charles Régin

Licence Informatique 2ème année

Itération : définition

- **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

Itérations : plan

- Une boucle simple : recherche du min, accès direct avec indice
- Cas particulier pour le début / la fin
- Pas seulement une opération
- Une boucle et deux indices
- Boucles imbriquées : calcul de matrice
- Indirections : tri par comptage
- Itérateurs (itération et structures de données)

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


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

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

Itérations : plan

- Une boucle simple : recherche du min, accès direct avec indice
- **Cas particulier pour le début / la fin**
- Pas seulement une opération
- Une boucle et deux indices
- Boucles imbriquées : calcul de matrice
- Indirections : tri par comptage
- Itérateurs (itération et structures de données)

Recherche de début à fin

- Au lieu de parcourir tous les éléments du tableau, on veut limiter la portion du tableau considéré. Cela revient à dire que l'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


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

```
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


```
entier indicePlusPetit(T[],n,d,val)
j ← d
Pour i de d à ???? { // C'est compliqué !!!
     …
```

```
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 ≤ n ET T[i] ≠ val** et non pas
  - i ≤ n OU T[i] ≠ val
  - T[i] ≠ val ET i ≤ n

Recherche de début à condition

- Il manque le test T[d]=val

- ```
  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

- Avec un **répéter{ }jusqu'à (condition)**
- entier indicePlusPetit(T[],n,d,val)

```
si (T[d] = -1){
     retourner -1
}
j ← d; i ← d
Répéter{
     si (T[i] < T[j]){
          j ← i
     }
      i ← i + 1
} jusqu'à (i > n ou T[i] = -1)
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.
- Exemple :
  - On écrit son nom avec un espace entre les lettres consécutives : DUPONT donne D U P O N T
  - Il n'y a pas d'espace après la dernière lettre

Cas particulier pour le début et la fin


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

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

Itérations : plan

- Une boucle simple : recherche du min, accès direct avec indice
- Cas particulier pour le début / la fin
- **Pas seulement une opération**
- Une boucle et deux indices
- Boucles imbriquées : calcul de matrice
- Indirections : tri par comptage
- Itérateurs (itération et structures de données)

Pas seulement une opération

- Ce qui est fait dans la boucle peut être complexe et dépendre des itérations précédentes
- Exemple :
  - Un tableau représente un nombre en binaire
  - On veut faire la somme de 2 tableaux (même taille) en tenant compte de la retenue

Pas seulement une opération

- T1 : [0 0 1 1 0 1 1 1]
- T2 : [1 0 1 0 0 0 1 1]
- \+ : [1 1 0 1 1 0 1 0]

Pas seulement une opération


```
somme(T1[],T2[],R[],n)
retenue ← 0
pour i de 1 à n {
      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
      }
}
```

Itérations : plan

- Une boucle simple : recherche du min, accès direct avec indice
- Cas particulier pour le début / la fin
- Pas seulement une opération
- **Une boucle et deux indices**
- Boucles imbriquées : calcul de matrice
- Indirections : tri par comptage
- Itérateurs (itération et structures de données)

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 » : ca marche
  - m = « rame » et r = « algorithme » : ca ne marche pas
- 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 vrai
}
retourner faux
```

Itérations : plan

- Une boucle simple : recherche du min, accès direct avec indice
- Cas particulier pour le début / la fin
- Pas seulement une opération
- Une boucle et deux indices
- **Boucles imbriquées : calcul de matrice**
- Indirections : tri par comptage
- Itérateurs (itération et structures de données)

Boucles imbriquées

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

Boucles imbriquées


```
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[(l-1)*numCol+c]=M[l][c]

- ```
  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
    }
}
```

Itérations : plan

- Une boucle simple : recherche du min, accès direct avec indice
- Cas particulier pour le début / la fin
- Pas seulement une opération
- Une boucle et deux indices
- Boucles imbriquées : calcul de matrice
- **Indirections : tri par comptage**
- Itérateurs (itération et structures de données)

Indirections

- Présentation habituelle de certains algorithmes :
  - On a un tableau d'entiers
  - On veut trier ce tableau
- Un élément du tableau est directement un type de base (un entier, un flottant, un booléen...)
- Parfois on ne voudrait pas avoir accès à la valeur en soi, mais plutôt à un objet lié à l'index et associé à cette valeur
- On veut simplement parcourir les éléments d'un ensemble, pas uniquement les valeurs de ces éléments : on associe l'élément à une valeur

Indirections

- On veut associer plusieurs informations à une personne
  - On fait plusieurs tableaux globaux
    - Noms des personnes
    - Prénoms des personnes
    - Date de naissance des personnes
  - Une personne est représentée par son indice

Permutation

- Proposer un algorithme qui permet de savoir si les éléments d'un tableau T forment une permutation de 1..n

Permutation

- Idée :
  - 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


```
Booleen 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 p, on peut les trier facilement en comptant le nombre d'occurrences de chaque nombre.
  - On crée un tableau de p 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

Tri par comptage


```
Booleen tri(T[],n)
min ← T[1]; max ← T[1]
pour i de 1 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 ?

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 : [2, 1, 5, 3, 4, 7, 6]

  - Essayer de trouver une solution !

Tri par comptage

- On va essayer de balayer le tableau T pour trouver la place des éléments !
- On va procéder par accumulation
- ```
  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
  }
  ```
- On va maintenant accumuler la position dans P
- ```
  pour k de 1 à (max-min) {
        P[k+1] ← P[k+1]+P[k]
  }
  ```
- P[i] contient maintenant le nombre d'éléments qui sont plus petits ou égal à i
- ```
  pour i de 1 à n {
        pos ← P[T[i]]
        rang[pos] ← i
        P[T[i]] ← P[T[i]] - 1
  }
  ```
  ```

Itérations : plan

- Une boucle simple : recherche du min, accès direct avec indice
- Cas particulier pour le début / la fin
- Pas seulement une opération
- Une boucle et deux indices
- Boucles imbriquées : calcul de matrice
- Indirections : tri par comptage
- **Itérateurs (itération et structures de 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
- 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

- Le 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

Itérateur : en Java

- L'interface java.util.Iterator permet l'utilisation d'itérateurs (iterator) avec les classes containers
- Chaque itérateur fournit les méthodes `next()` et `hasNext()` et peut supporter (non obligatoire) la méthode `remove()`.
- Un itérateur est typiquement créée par la classe container auquel il correspond, habituellement par la méthode `iterator()`.
- La méthode `next()` avance l'itérateur et renvoie la valeur pointée par l'itérateur. Quand un itérateur est créé, il pointe vers une valeur spéciale qui est avant le premier élément, de telle façon que le premier élément est obtenu avec le premier appel de `next()` :
- ```
  Iterator iter = list.iterator();
  while (iter.hasNext()){
      System.out.println(iter.next());
  }
  ```
- SI la fonction `remove()` est supportée, alors elle supprime l'élément le plus récemment visité.
  ```