ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 26 mars 2011

Résumé

– Traitement récursif des listes (partie 1)

*. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Rappels sur la récursivité

– La solution d'un problème donné peut être construite à partir des solutions de (plus petits) sous problèmes ;
– Ces sous problèmes sont de même nature et peuvent donc être résolus de la même manière ;
– Les sous problèmes sont de plus en plus petits (convergence) ;
– Finalement, il existe une taille de problèmes pouvant être résolus de façon triviale, sans recourt à des appels récursifs.

Quels problèmes avez-vous déjà résolus à l'aide de la récursivité ? Calcul d'une factorielle ? Trouver une valeur dans un tableau ? Ici, nous utiliserons la récursivité afin de traverser des listes (simplement) chaînées.

Intuition

Considérons le calcul de la somme des éléments d'un tableau, t.

Le problème entier consiste à calculer la somme des éléments, $t[k]$, pour l'intervalle $k = 0$, jusqu'à $length - 1 = |t| - 1$.

Posons s comme étant la somme des valeurs pour l'intervalle $k = 1$ jusqu'à $length - 1$.

De plus, supposons que la valeur de s est déjà connue. Quelle est la solution du problème initial ? $t[0] + s$.

Par quelle approche allons nous calculer s, la somme des éléments pour le sous intervalle $k = 1$, jusqu'à $length - 1$?

En appliquant la même approche, sauf que cet intervalle est plus petit d'un élément.

Quelle est la taille d'un problème pouvant être traité de façon triviale ? Un intervalle de taille 1.

(suite)

```
private static int sum( int[] t, int k ) {

    int s, result, length = t.length - k;

    if ( length == 1 ) { // Cas de base

        result = t[ k ];

    } else { // Cas général

       int k1 = k+1;
       s = sum( t, k1 );
       result = t[ k ] + s;

    }
    return result;
}
```

Quel sera l'appel initial ?

```
public static int sum( int[] t ) {
    return sum( t, 0 );
}
```

```
public class Sum {
    private static int sum( int[] t, int k ) {
        if ( k == ( t.length - 1 ) ) {
            return t[ k ];
        }
        return t[ k ] + sum( t, k+1 );
    }
    public static int sum( int[] t ) {
        if ( t.length == 0 ) {
            throw new IllegalArgumentException();
        }
        return sum( t, 0 );
    }
    public static void main( String[] args ) {
        int[] t = { 1, 2, 3, 4, 5 };
        System.out.println( sum( t ) );
    }
}
```

Remarques

Il est important que la taille des problèmes à traiter diminue, sinon il y aurait un nombre infini d'appels récursifs (équivalent de la boucle infinie) ; en pratique le programme terminera lorsque toute la mémoire réservée pour les appels de méthodes est épuisée.

Il faut donc qu'il y ait une taille de problèmes pouvant être résolus sans appels récursifs, afin de stopper la récursivité.

Ce sont les cas de base. Il faut qu'il y en ait au moins un, mais il peut y en avoir plusieurs.

Les cas de base doivent être traités en premier afin de stopper la récursivité, si nécessaire.

Les langages de programmation Lisp, Prolog et Haskell, pour en nommer que quelques-uns, n'ont aucun énoncé de contrôle itératif, tout se fait à l'aide de la récursivité.

Modèle (pseudo-code)

```
type méthode ( paramètres ) {

    type résultat;

    if ( test ) { // détecter le cas de base

        résultat = calcul du résultat directement (sans appel récursif)

    } else { // cas général

        // prétraitement; partitionner les données par exemple

        résultat = méthode( sous ensemble des données ); // récursion

        // posttraitement; combiner les résultats
    }

    return résultat;
}
```

Factorial

```
public static int factorial( int n ) {
    int s, result;
    if ( n<=1 ) { // cas de base
        result = 1;
    } else { // cas général
        int n1 = n-1;
        s = factorial( n1 );
        result = n * s;
    }
    return result;
}
```

La méthode ci-haut correspond au modèle proposé, on vérifie d'abord le cas de base, son résultat est calculé sans appel récursif (la récursivité s'arrête ici !), le cas général crée des sous problèmes de plus en plus petits.

Factorial

```
public static int factorial( int n ) {

    if ( n<=1 ) {
        return 1;
    }

    return n * factorial( n-1 );
}
```

L'énoncé **return** retourne le contrôle à l'appelant, il stop l'exécution de la méthode, aucun autre énoncé de cet appel ne sera exécuté.

Binary Search

```
public static int binarySearch( int value, int[] array ) {
    return binSearch( value, array, 0, array.length-1 );
}
```

La méthode **binarySearch** retourne la position de la valeur recherchée ou -1 si la valeur est absente du tableau.

Cette méthode est très efficace parce que la taille de l'intervalle à traiter diminue de moitié à chaque appel.

Cas de base : intervalle de taille 0, la valeur est absente du tableau, ou encore, l'élément central contient la valeur recherchée.

Cas général : créer des intervalles de valeurs de plus en plus petits.

```
private static int binSearch( int value, int[] array, int lo, int hi ) {

    // Base case: value not found
    if ( lo > hi ) {
        return -1;
    }
    // Base case: value was found
    int middle = ( lo + hi ) / 2;
    if ( value == array[ middle ] ) {
        return middle;
    }
    // General case:
    int newLo, newHi;
    if ( value < array[ middle ] ) {
        newLo = lo;
        newHi = middle - 1;
    } else {
        newLo = middle + 1;
        newHi = hi;
    }
    return binarySearch( value, array, newLo, newHi );
}
```

Traitement récursif des listes — stratégie ≪head+tail≫

Développons une stratégie générale pour le traitement récursif des listes.

Considérons d'abord le calcul de la taille d'une liste (**size**).

Divisons la liste en deux parties, le premier élément (**head**) et le reste de la liste (**tail**)[1].

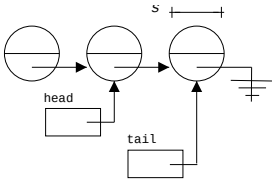
Par analogie avec le calcul de la somme, posons que s désigne la longueur de la liste désignée par **tail**.

Étant donné s, quelle est la longueur totale de la liste ?

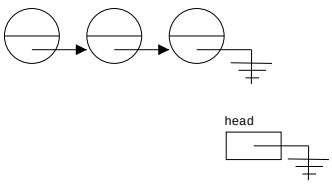
Réponse : $s + 1$.

Comment calcule-t-on la valeur de s ? Hum, s est la longueur d'une liste, de plus la liste désignée par **tail** est plus petite, appliquons récursivement la méthode que nous développons.

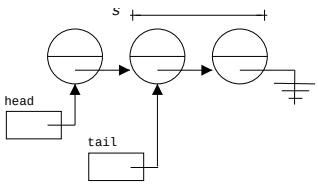
1. Ici, **head** et **tail** ne sont pas des variables d'instance.



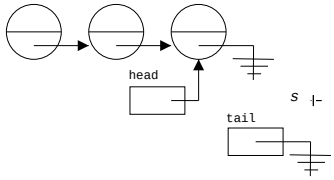
⇒ La taille de la liste désignée par **head** est $s + 1$.



⇒ La taille de la liste désignée par **head** est 0.



⇒ La taille de la liste désignée par **head** est $s + 1$. **head et tail ne sont pas des variables d'instance.**



⇒ La taille de la liste désignée par **head** est $s + 1$.

Implémentation

Voici la définition de la classe pour laquelle toutes les méthodes développées pour cette présentation sont définies.

```
public class OrderedList< E extends Comparable<E> > {
    private static class Node<E> {
        private E value;
        private Node<E> next;
        private Node ( E value, Node<E> next ) {
            this.value = value;
            this.next = next;
        }
    }
    private Node<E> first; // <---
    public OrderedList () {
        first = null;
    }
    // les méthodes ...
}
```

Stratégie ≪head+tail≫ (suite)

Étant donné une liste **l**, appliquer la méthode récursivement sur le reste de la liste (**tail**).

En général, le résultat final est obtenu en combinant le résultat de l'appel récursif avec un certain calcul sur le premier élément de la liste.

Cette stratégie ne permet pas de résoudre tous les problèmes, mais tous ceux présentés ici.

Pour la méthode **size**, quel est le cas de base ?

Avec chaque appel récursif, la liste devient de plus en plus petite, quelle est la plus petite liste à traiter ?

La liste vide ! Quelle est sa taille ? Zéro.

La liste vide est forcément un cas de base puisqu'il serait impossible de faire un appel récursif avec le reste ; il n'y a pas de reste !

```
int size( Node<E> p ) {
    int s, length;

    if ( p == null ) { // Cas de base

        length = 0;

    } else { // Cas général

        s = size( p.next );

        length = 1 + s;

    }
    return length;
}
```

Ou simplement,

```
int size( Node<E> p ) {
    if ( p == null ) {
        return 0;
    }
    return 1 + size( p.next );
}
```

On ne peut faire un appel à cette méthode de l'extérieur de la classe **OrderedList** parce que le paramètre de la méthode est type **Node** et privé.

Il nous faut donc une méthode auxiliaire, qui sert à initier le premier appel.

```
public int size() {
    return size( first );
}
```

⇒ Peut-on faire un appel à cette méthode de l'extérieur de la classe ? Que sera l'appel initial ?

Puisque toutes les méthodes récursives décrites ici nécessitent un (plusieurs) paramètre(s) de type **Node**, elles seront toutes **privées** et auront toutes une méthode auxiliaire (publique) afin d'initier le premier appel.

```
public int size() {
    return size( first );
}

private int size( Node<E> p ) {
    if ( p == null ) {
        return 0;
    }
    return 1 + size( p.next );
}
```

Remarque : les méthodes récursives auront toujours au moins un paramètre de plus que leur contrepartie publique.

La stratégie ≪head+tail≫ n'est pas la seule. On pourrait diviser la liste en deux parties plus ou moins égales, et la longueur de chacune d'elles serait déterminée au moyen d'un appel récursif.

La stratégie ≪head+tail≫ n'est qu'un cas particulier de celle-ci, ou le calcul de la longueur de la liste de gauche (**head**) ne nécessite aucun appel récursif.

Pour une liste de longueur n, il y a $n-1$ décompositions en deux listes possibles.

Étant donné l'implémentation de nos listes, la stratégie ≪head + strategy≫ est plus simple.

Pour certains types de problèmes, tels que le tri, diviser la liste en deux parties égales donne lieu à un algorithme plus efficace (voir CSI 2514 – structures de données).

"Head + tail"

```
if ( ... ) { // Cas de base

   calcul du résultat (sans appel récursif)

} else { // Cas général

  // prétraitement

  s = méthode( p.next ); // récursivité

  // posttraitement

}
```

"Head + tail"

Étapes afin de développer une méthode selon la stratégie "head+tail".

Que signifie **méthode(p.next)** ?

La solution d'un problème, plus petit d'un élément.

Comment allons-nous utiliser ce résultat afin de construire la solution du problème pour une liste débutant par l'élément **p** ?

Quels sont les cas de base ?

Quel est la plus petite liste valide ? Quel est le résultat ?

E findMax()

Vous devez concevoir une méthode récursive afin de trouver la valeur maximale d'une liste d'éléments de type < **E extends Comparable**<**E**> >.

Considérons la partie **publique non-récursive** d'abord.

Quelle est la plus petite liste valide ? La liste contenant un seul élément.

Qu'allons nous faire si la liste est vide ? Lancer une exception.

Par quel élément débute la récursion ? Le premier élément de la liste.

Comparable findMax()

```
public E findMax() {
    if ( first == null ) {
        throw new NoSuchElementException();
    }
    return findMax( first );
}
```

Remarque : la méthode récursive possède toujours au moins un paramètre de plus que sa contrepartie publique.

E findMax(Node<E> p)

Appliquons la stratégie «head+tail» telle que proposée.

Le cas général contiendra l'énoncé suivant :

```
E result = findMax( p.next );
```

Que représente **result** ?

Hum, **result** représente la valeur maximale pour le reste de la liste.

Que fait-on de ce résultat ?

Comparer cette valeur à la valeur courante (celle de l'élément désigné par **p**).

```
if ( result.compareTo( p.value ) > 0 ) {
    return result;
} else {
    return p.value; {
}
```

E findMax(Node p)

Quel est le cas de base ?

Ce processus construit des problèmes de plus en plus petits, quelle est la plus petite liste valide ?

Non,pas la liste vide, mais la liste contenant un seul élément.

Quelle sera la valeur retournée ?

C'est ça, la valeur se trouvant à cette position.

```
if ( p.next == null ) {
    return p.value;
}
```

E findMax(Node p)

```
public E findMax() {
    if ( first == null ) {
        throw new NoSuchElementException();
    }
    return findMax( first );
}
private E findMax( Node<E> p ) {
    if ( p.next == null ) {
        return p.value;
    }
    E result = findMax( p.next );
    if ( result.compareTo( p.value ) > 0 ) {
        return result;
    } else {
        return p.value;
    }
}
```