Récursivité et arbres

Récursion, Mola Kucher.

Un programme java

```
int log2(int nombre) {
        int res = 0;
        while(nombre >= 2) {
                res = res + 1;
                nombre = nombre / 2;
        }
        return res;
}
```

Définition d'informaticien (à savoir)

$log_2(x)$: nombre de fois qu'il faut diviser x par 2 pour atteindre 1.

Version récursive

```c
int log2(int nombre){
        if(nombre < 2)
                return 0;
        else
                return 1 + log2(nombre / 2);
}
```

Autre exemple

```
static int mystere(int n){
  return n * mystere(n - 1);
}
```

- Que fait cette fonction ?

Autre exemple

```
static int mystere(int n){
  return n * mystere(n - 1);
}
```

- Que fait cette fonction ? $\longrightarrow$ boucle infinie

Autre exemple

```
static int mystere(int n){
  return n * mystere(n - 1);
}
```

- Que fait cette fonction ? ⟶ boucle infinie
- prendre les mêmes précautions que pour un `while`

Règle du jeu :

```
<T> ma_fonction_recursive(<U> param){
  if(<condition d arret>){
    arret;
  }
  else{
    calcul recursif;
  }
}
```

- Dans le calcul récursif, on suppose que la fonction fonctionne déjà (sur des valeurs plus petites)

Ne suffit pas toujours

```
static void syracuse(int n){
  if(n <= 1){
    System.out.println("Finished");
  }
  else{
    if(n % 2 == 0)
      return syracuse(n / 2);
    else
      return syracuse(3 * n + 1);
  }
}
```

- Problème de l'arrêt... indécidable en général.

1.2 - Pile d'appel

Un exemple

```
void affiche(int n){
  if(n>0){
    System.out.println("A␣" + n);
    affiche(n-1);
    System.out.println("B␣" + n);
  }
}
```

Que fait l'appel `affiche(3)` ?

Après l'appel récursif

Attention

Un appel récursif n'est pas un `goto()`

- Lors de l'appel récursif, la fonction s'appelle elle-même.
- Elle reprend la main ensuite.

Une erreur courante

Que fait cette fonction ?

```
int somme(int n, int res){
  if(n<=1)
    return res;
  else
    somme(n - 1, res + n);
}
```

Une erreur courante

Que fait cette fonction ?

```
int somme(int n, int res){
  if(n<=1)
    return res;
  else
    somme(n - 1, res + n);
}
```

- Le **return** fait un retour au sein de l'appel...
  ⟶ ne traverse pas la pile.
- ligne 5 ⟶ `return somme(n-1, res+n);`

Récursivité terminale

Pas de pile

Principe :

Le résultat de l'appel récursif est le résultat de la fonction.

- Pas besoin de faire une pile
- La pile existe souvent quand même, selon le langage/compilateur.

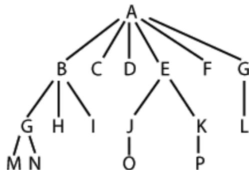
- `return f(n-1);` **terminale**
- `return 3 + f(n-1);` pas terminale

1.3 Arbres

Les arbres

Un arbre = un nœud parent (racine)
+ des enfants, eux-mêmes des arbres

- Racine : le nœud tout en haut
- Feuille : nœud sans enfants
  (Rq : la racine peut être une feuille)
- sous-arbre ancré en B : le bout de l'arbre dont B est la racine.



Structure de données

Structure de donnée pour un arbre

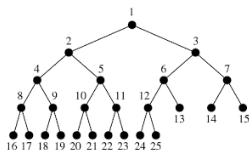
$$\text{Nœud} = \left\{ \begin{array}{l} \text{valeur,} \\ \text{enfants : Set<Nœud>} \end{array} \right\}$$

- La structure ne contient qu'un nœud et des références.
- la profondeur/le nombre de nœud non visibles.
- feuille : ensemble d'enfants vide

(Un peu comme un composite !)

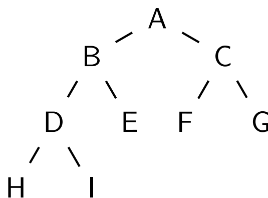
Exemple de problèmes sur les arbres

- Calculer la somme des valeurs dans l'arbre ?
- Calculer la plus petite valeur d'un arbre.
- Calculer la profondeur d'un arbre.
- Insérer un valeur, modifier la forme
- ...



Traiter tous les sommets d'un arbre

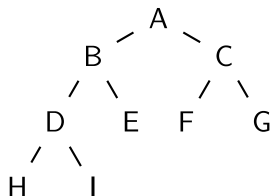
```
void traverse(Node node){
    act_on(node);
    for(Node subTreee:
            node.getChildren()){
        traverse(subTree);
    }
}
```



- condition d'arrêt ?
- ordre de parcours
- Les noeuds en attente sont dans la pile d'appel.

Traitement postfixe

```
void parcourir(Node noeud){
  for(Node sousArbre:
           noeud.getChildren()){
      parcourir(sousArbre);
  }
  traitement(noeud);
}
```



Ordre de traitement

Parfois utile de déplacer l'appel de `act_on`.

- **prefixe :** on traite le parent avant les enfants (en descendant).
- **postfixe :** on traite le parent après les enfants (en remontant).
- **infixe :** on traite le parent au milieu (au cours du demi tour), que pour les arbres binaires.

Méthode d'écriture des fonctions sur les arbres

Stratégie :

- On écrit la fonction en admettant qu'elle fonctionne déjà sur les sous-arbres.

```
int sum(Node tree){
        int sum = tree.getValue();
        for(Node child: tree.getChildren()){
                sum += child.getValue();
        }
        return sum;
}
```

- Portée de la variable *sum* : une variable par appel
- Attention à la gestion des `return`.

Autre exemple

Maximum des valeurs des nœuds

```
int max(Node tree){
        int max = tree.getValue();
        for(Node child: tree.getChildren()){
                if(max(child) > max){
                        max = max(child);
                }
        }
        return max;
}
```

Nombre d'appels exponentiel !

Autre exemple

Maximum des valeurs des nœuds

```
int max(Node tree){
        int max = tree.getValue();
        for(Node child: tree.getChildren()){
                int maxChild = max(child);
                if(maxChild > max){
                        max = maxChild;
                }
        }
        return max;
}
```

- Attention aux appels multiples.

1.4 Parcours itératifs

Parcours itératif

- `add(elt)` : ajoute à la fin
- `pop()` : retire et retourne le premier élément

```
void parcours_iteratif(Node node){
  LinkedList<Node> queue =  new LinkedList<Node>();
  queue.add(node);
  while(!queue.isEmpty()){
    Node current = queue.pop();
    traitement(current);
    for(Node child: current.getChildren()){
      queue.add(child);
    }
  }
}
```

Parcours itératif

- `add(elt)` : ajoute à la fin
- `pop()` : retire et retourne le premier élément
- `push(elt)` : insère au début

```
void parcours_iteratif(Node node){
  LinkedList<Node> queue =  new LinkedList<Node>();
  queue.add(node);
  while(!queue.isEmpty()){
    Node current = queue.pop();
    traitement(current);
    for(Node child: current.getChildren()){
      queue.add(child);
    }
  }
}
```

- Que se passe-t-il si on remplace le `add` par un `push` ?

(différence entre *pile* et *file*)

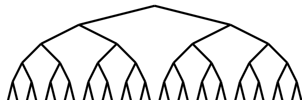
Parcours d'arbre : deux directions

En largeur : Breadth First Search (BFS)

- on parcourt la fratrie avant la génération suivante
- solution itérative, avec une **file** explicite
- taille de la file : largeur de l'arbre

En profondeur : Depth First Search (DFS)

- on parcourt les enfants avant la fratrie.
- solution récursive, plus simple
- possible aussi avec une **pile**
- taille de la pile : hauteur de l'arbre



Conclusion

- Utilisation de la récursivité.
- Algorithmes sur les arbres :
  - le plus simple, en profondeur avec un parcours récursif,
  - possible en itératif avec des structures dédiées, mais plus pénible.