Itératif et récursif

I Choisir entre itératif et récursif

Un problème peut se résoudre avec une implémentation récursive, comme avec une implémentation itérative. Utiliser l'implémentation récursive permet d'avoir un programme plus simple, plus facile à comprendre.

L'implémentation récursive a cependant deux principaux inconvénients:

1-Un appel de fonction prend plus de temps qu'une simple itération de boucle.

2-Un appel de fonction utilise de la mémoire pour son environnement d'exécution (segment de pile).

-Le premier inconvénient produit des programmes plus lents que leurs équivalents itératifs. Si le moindre gain de vitesse est important, il peut donc être préférable d'utiliser une implémentation itérative. Dans le cas contraire, la perte de performances peut être largement compensée par le gain en clarté du code, donc en réduction de risques de laisser des bugs.

-Le deuxième inconvénient peut être très gênant si le nombre d'appels imbriqués est très important. Chaque appel de fonction imbriqué utilise une certaine quantité de mémoire, plus ou moins importante selon le nombre de paramètres et de variables définis dans la fonction. Cette mémoire est libérée dès que l'exécution de la fonction se termine, mais dans le cas d'une fonction récursive, cette quantité de mémoire est multipliée par le nombre d'appels imbriqués à un moment donné.

Note : En général la version itérative est préférable pour son efficacité

II Transformation récursif en itératif

Tout algorithme récursif peut être transformé en un algorithme itératif équivalent : La méthode à suivre dépend du type de récursivité de l'algorithme.

II.1 transformation de fonctions récursives terminales en fonctions itératives

Une fonction f() est dite **récursive terminale** si elle n'effectue **aucun traitement après l'appel récursif**. Elles sont de la forme suivante :

```
type f(params)
{
if (condition)
       instructions A
else
{      instructions B
       f(nouv_params) }
}
```

Exemple

```
int fact(int n)
{ if (n == 1) return 1;
return n * fact(n-1);
}
```

La fonction fact ci-dessus n'est pas récursive terminale car on doit effectuer une multiplication après l'appel mais on peut introduire un paramètre supplémentaire pour la rendre terminale comme suit :

```
int fact (int n, int  R )
 { if (n == 1) return R;
   return fact (n-1, n*R);
 }
```

```
type f (params)
 if (condition)
     instruction  A
 else {
       instructions B
       f (nouv_params)
    }
```

```
int fact (int n, int  R )
  if (n == 1)
        return R ;

      (pas d'instructions B)
      return fact( n-1, n*R)  ;
```

La transformation est :

```
type f (params)
{
 while (non condition)
    { instructions 2
      params = nouv_params
    }
 instructions 1
}
```

Pour fac:

```
Int  fact (int n, int R)
 { while (n!=1)
     { R = n*R;
       n = n - 1;
     }
return R;
 }
```

II.2 transformation de fonctions récursives non terminales en fonctions itératives

Schéma d'une fonction récursive non terminale de base:

```
Type R(X)
{
 Instructions A
 if( C(X) ) { instructions B;
              R( φ(X));
             Instructions D;
             }
 else
 instructions E;
}
```

Où : X : liste de paramètres C : condition d'arrêt portant sur X

A, B, D, E : bloc d'instructions (éventuellement vide)

φ(X) : transformation des paramètres(nouv_params)

Algorithme itératif équivalent.

```
Type  I(X)
{p : Pile; init(&p);
  Instructions A ;
  while ( C(X) ) { instructions B;
                   empiler(&p,X)
                   X = φ(X);
                   Instructions A;
                 }
  Instructions E ;
  while ( !pilevide(p) )
         { depiler(&p,&X)
           instructions D;
         }
}
```

Exercice 1 :

1- Compléter la fonction récursive suivante en ajoutant l'instruction qui permet d'afficher la valeur binaire du nombre décimale n.

```
 void   NB (int n)
      {int   r ;
        r   =   n % 2 ;
        if  (    )  NB (  );
       }
```

2- Transformer cette fonction récursive en fonction itérative.

Exercice 2 :

Donner la fonction itérative de Quicksort ;

```
void quicksort(int v[],int i, int j)
 {int m;
  pile p; init(&p);
  empiler(&p,i);empiler(&p,j);
  while(!pilevide(p))
   { depiler(&p,&j);depiler(&p,&i);
      while(i<j)
      {m=partition(v,i,j);
       empiler(&p,m+1);empiler(&p,j);
        j=m-1;
      }
   }
  }
```