```
/*

$ javac L.java
$ java L 1000000   #echec pour un million
$ java -Xss20m L 1000000  #ca passe
javac ne traite pas la rÃ©cursion terminale, mais scala la traite
*/
class A {
        static public A emptyA= null;
        public int key;
        public A filsgauche, filsdroit;

        static public A         btree( A fg, int e, A fd)
        { A noeud=new A();
          noeud.filsgauche= fg; noeud.filsdroit=fd; noeud.key = e;
          return noeud;
        }
        static public A         left( A n) { return n.filsgauche; }
        static public A         right( A n) { return n.filsdroit; }
        static public int       value( A n) { return n.key; }

        // insertion d'un entier dans un arbre
        static public A   insrt( A n, int v)
        { if (emptyA==n) return btree( emptyA, v, emptyA);
          int e= value( n);
          if (v <= e) return btree( insrt( left(n), v), e, right(n));
          return btree(  left(n), e, insrt( right(n), v));
        }
        // insertion d'une liste d'entiers dans un arbre
        static public A   insrtn( A n, L l)
        { if ( L.isempty( l)) return n;
          return insrtn( insrt( n, L.head( l)), L.tail( l));
        }
        static public L    infix( A n, L l)
        { if( emptyA==n) return l;
          return infix( left(n), L.cons( value(n), infix( right(n), l)));
        }
        static public L treesort( L l)
        { return infix( insrtn( emptyA, l), L.nil); }
}


class Comparison { int compare( int a, int b) { return 0; }}
class Croissant extends Comparison { int compare( int a, int b) { return a-b;}}
class Decroissant extends Comparison { int compare( int a, int b){ return b-a;}}

class Fonction { int eval( int x) { return x; }}
class Fact extends Fonction { int eval( int x) { return L.fact(x); }}
class Fib extends Fonction { int eval( int x) { return L.fib(x); }}

public class L {
   public int elt;
   public L next;
   static public L cons( int x, L tail)
        { L cell=new L(); cell.elt=x; cell.next=tail; return cell;}
   static public L nil= null;
   static public boolean isempty( L liste) { return liste==nil ; }
   static public int head( L liste) { return liste.elt; }
   static public L tail(  L liste) { return liste.next; }
   static int fib( int x) { if (x < 2) return x; return fib(x-1) + fib(x-2); }
   static public int fact( int x) { if (x < 2) return 1; return x * fact(x-1); }
   static public L map( Fonction f, L l) { if (isempty( l)) return nil;
                return cons( f.eval( head( l)), map( f, tail( l))); }
   static public L iaj( int i, int j)
        { if (i==j) return cons(i, nil);
          if (i<j) return cons( i, iaj( i+1, j));
          return cons( i, iaj( i-1, j)); }
```



```
// concat de 2 listes
   static public L concat( L l1, L l2)
   { if (isempty( l1) ) return l2;
     if (isempty( l2) ) return l1;
     return cons( head( l1), concat( tail( l1), l2)); }

// generer une liste aleatoire;
   static public int random( int x ) //{ return (311 * x) % 1001; }
   { return (int) (10000. * Math.random() ); }

   static public L lalea( int n) { if (n==0) return nil; return cons( random(n),
 lalea(n-1)); }
   static public L lalea2( int n) { L l=nil;
        for (int i=0; i<n; i++) l=cons( random(i), l);
        return l; }

//*************************************************************************
// tri fusion
//*************************************************************************

 // rend la liste des elements de rang pair (le premier est de rang 0 donc pair)

   static public L pairs( L l)
   { if (isempty(l)) return nil;
     if (isempty( tail( l))) return cons( head( l), nil);
     return cons( head( l), pairs( tail( tail( l))));
   }
 // rend la liste des elements de rang impair
   static public L impairs( L l) { if (isempty(l)) return nil; return pairs( tai
l(l)); }
 // fusion de 2 listes triees
   static public L fusion( L l1, L l2)
   { if (isempty( l1)) return l2;
     if (isempty( l2)) return l1;
     int e1= head( l1); int e2= head( l2);
     if (e1 <= e2) return cons( e1, fusion( tail( l1), l2));
     return cons( e2, fusion( l1, tail( l2)));
   }
   static public  L mergesort( L l)
   { if (isempty( l) || isempty( tail( l))) return l;
     return fusion( mergesort( pairs( l)), mergesort( impairs( l)));
   }

//*************************************************************************
   // QUICKSORT
//*************************************************************************
   // filtre les elts + petits que key dans une liste l :
/* trop de recursion pour le compilateur java...
   static public L smallerthan( L l, int key)
   { if (isempty( l)) return nil;
     if (head( l) < key) return cons( head( l), smallerthan( tail( l), key));
     return smallerthan( tail( l), key);
   }
*/

   static public L smallerthan( Comparison cmp, L l, int key)
   { if (isempty( l)) return nil;
    L r=nil;
    for( ; ! isempty(l); l=tail(l) )
    { if (cmp.compare( head( l), key) < 0)   r = cons( head( l), r); }
    return r;
   }

   // filtre les elts + grands que key dans une liste l :
/* trop de recursion Ã  l'exÃ©cution pour le compilateur java...
   static public L greaterthan( L l, int key)
   { if (isempty( l)) return nil;
     if (cmp.compare( head( l), key) > 0) //(head( l) > key)
        return cons( head( l), greaterthan( tail( l), key));
```

```
    return greaterthan( tail( l), key); }
*/
   static public L greaterthan( Comparison cmp, L l, int key)
   { if (isempty( l)) return nil;
     L r=nil;
     for( ; ! isempty(l); l=tail(l) )
     { if (cmp.compare( head( l), key) > 0)  r = cons( head( l), r); }
     return r;
   }

   // filtre les elts equal to key dans une liste l :
/* trop de recursion Ã  l'exÃ©cution pour le compilateur java...
   static public L equalto( L l, int key)
   { if (isempty( l)) return nil;
     if (head( l) == key) return cons( head( l), equalto( tail( l), key));
     return equalto( tail( l), key); }
*/
   static public L equalto( Comparison cmp, L l, int key)
   { if (isempty( l)) return nil;
     L r=nil;
     for( ; ! isempty(l); l=tail(l) )
     { if (cmp.compare( head( l), key)==0) r = cons( head( l), r); }
     return r;
   }
   static public L quicksort( Comparison cmp,  L l)
   {  if (isempty( l) || isempty( tail( l))) return l;
      int key = head( l); L tl= tail( l);
      return concat( quicksort( cmp, smallerthan( cmp, tl, key)),
                     concat( equalto( cmp, l, key),
                             quicksort( cmp, greaterthan( cmp, tl, key))));
   }

// ************************************************************************
   // TRI NAIF
// ************************************************************************
   static int min( int a, int b) { if (a<=b) return a; return b; }
   // minimm d'une liste :
/* trop de recursion ...
   static public int minimum( L l, int sivide)
   { if ( isempty( l)) return sivide;
     return minimum( tail( l), min( head( l), sivide));
   }
*/
   static public int minimum( L l, int sivide)
   { if ( isempty( l)) return sivide;
     int smallest=head(l);
     for( ; ! isempty( l); l=tail( l))
       if (smallest > head(l)) smallest=head(l);
     return smallest;
   }
/* trop de recursion ...
   static public L remove( L l, int key)
   { if ( isempty( l)) return l;
     if (key == head( l)) return tail( l);
     return cons( head( l), remove( tail( l), key)); }
*/
   static public L remove( L l, int key)
   { L r=nil;
     for( ; ! isempty(l); l=tail(l))
     {   if (key==head( l)) return concat( r, tail(l));
         else r= cons( head( l), r);
     }
     return r;
   }
   static public L slowsort( L l)
   {
        if ( isempty( l) || isempty( tail( l))) return l ;
        int key = minimum( l, head( l));
        return cons( key, slowsort( remove( l, key)));
```



```
   }

   public static void printL( L l)
   { int n=0;
     for (  L c=l; ! isempty( c) && n < 100 ; c=tail(c))
        { System.out.print( c.elt + " "); n++; }
     System.out.println(); }

   public static void main (String[] args)
   {
    {
    L l= cons( 1, cons( 2, cons( 3, nil)));
    for (L c=l; c != nil; c=c.next) System.out.print( c.elt + " ");
    System.out.println();
    for (L c=l; ! isempty( c); c= tail( c)) System.out.print( head(c) + ";");
    System.out.println();
    }

    System.out.println("Hello World");
    System.out.println("Fib(5)=");
    System.out.println( fib(5));
    for( int i=0; i<14; i++)
    {
            System.out.print("Fib(");
            System.out.print(i);
            System.out.print(")=");
            System.out.println( fib(i));
    }

    L l_0_10 = iaj( 0, 10);
    System.out.print("map Fact (iaj 0 10):\n");
    printL( map( new Fact(), l_0_10));
    System.out.print("map Fib (iaj 0 10):\n");
    printL( map( new Fib(), l_0_10));

    { // test de mergesort
    int n= 5000;
    if (1==args.length) n=Integer.parseInt(args[0]);
/*
for( int i=0; i<args.length; i++)
{ System.out.print("arg=");
  System.out.print(args[i]);
  System.out.print("\n");
}
*/
    System.out.print("n="); System.out.print( n); System.out.print("\n");

    // DM: sur ma machine, plante pour dix mille elements. Java gere mal la recu
rsion
    L l= lalea2( n);
    System.out.print( "\nLa liste non triee:\n");
    printL( l);

    System.out.print( "\nLa liste triee par mergesort :\n");
    L l2=mergesort( l);
    printL( l2);

    System.out.print( "\nLa liste triee par quicksort :\n");
    L l3= quicksort( new Croissant(),  l);
    printL( l3);

    System.out.print( "\nLa liste triee par quicksort decroissant :\n");
    L l3bis= quicksort( new Decroissant(), l);
    printL( l3bis);

    System.out.print( "\nLa liste triee par insertion dans 1 arbre binaire non Ã©quilibrÃ©:\n");
    L l4= A.treesort( l);
    printL( l4);
    System.out.print( "\nLa liste triee par slowsort :\n");
```

```
//    if (n<1000)
    printL( slowsort( l));
    // else System.out.print( "\n n trop grand\n");
    }
   }
}
```