```
Java  数组排序总结 (冒泡,选择,插入,希尔)
public     class    SortAll {


 /**
 *    冒泡排序，选择排序，插入排序，希尔（    Shell   ）排序 Java   的实现
 * 2008.11.09
 *    @author   YangL.     （http://www.idcn.org          ）
 */
 public      static       void   main(String[] args) {
   int   [] i = { 1, 5, 6, 12, 4, 9, 3, 23, 39, 403, 596, 87 };
System.out.println("----                  冒泡排序的结果：  ");
maoPao(i);
System.out.println();
System.out.println("----                  选择排序的结果：  ");
xuanZe(i);
System.out.println();
System.out.println("----                  插入排序的结果：  ");
 chaRu(i);
 System.out.println();
 System.out.println("----                  希尔（ Shell   ）排序的结果：  ");
 shell(i);
}


 //    冒泡排序
 public      static       void   maoPao(   int   [] x) {
   for   (  int    i = 0; i < x.length; i++) {
     for   (  int    j = i + 1; j < x.length; j++) {
       if   (x[i] > x[j]) {
          int    temp = x[i];
    x[i] = x[j];
    x[j] = temp;
   }
 }
}
```

```
   for    (   int     i : x) {

  System.out.print(i + " ");

 }

}



 //    选择排序

 public       static        void    xuanZe(     int    [] x) {

  for    (   int     i = 0; i < x.length; i++) {

    int     lowerIndex = i;

    //     找出最小的一个索引

    for    (   int     j = i + 1; j < x.length; j++) {

      if    (x[j] < x[lowerIndex]) {

   lowerIndex = j;

  }

 }

    //     交换

    int     temp = x[i];

  x[i] = x[lowerIndex];

  x[lowerIndex] = temp;

 }

   for    (   int     i : x) {

  System.out.print(i + " ");

 }

}



 //    插入排序

 public       static        void    chaRu(     int   [] x) {

  for    (   int     i = 1; i < x.length; i++) {                    // i      从一开始，因为第一个数已经是排好序的啦

    for    (   int     j = i; j > 0; j--) {

      if    (x[j] < x[j - 1]) {

         int     temp = x[j];

  x[j] = x[j - 1];

  x[j - 1] = temp;

 }
```

```
 }

 }

   for    (  int    i : x) {

  System.out.print(i + " ");

 }

}

//    希尔排序

public      static       void    shell(      int   [] x) {

  //     分组

  for    (  int    increment = x.length / 2; increment > 0; increment /= 2) {

    //     每个组内排序

    for    (  int    i = increment; i < x.length; i++) {

      int    temp = x[i];

      int    j = 0;

      for    (j = i; j >= increment; j -= increment) {

        if    (temp < x[j - increment]) {

    x[j] = x[j - increment];

   }         else    {

            break   ;

   }

  }

  x[j] = temp;

 }

}

   for    (  int    i : x) {

  System.out.print(i + " ");

 }

}
```

递归算法的复杂度

递归算法的复杂度通常很难衡量，一般都认为是每次递归分支数的递归深度次方。但通常情况下没有这个大，　如果我们可以保存每次子递归的结果的话，　递归算法的复杂性等于不同的节点个数。这也是动态规划算法思想的由来。

看一下下面这个算法题目，据称是百度的笔试题：

简述：实现一个函数，对一个正整数　n，算得到 1 需要的最少操作次数：
如果 n 为偶数，将其处以 2；如果 n 为奇数，可以加 1 或减 1；一直处理下去。

要求：实现函数（实现尽可能高效）　int func(unsigned int n)　；n 为输入，返回最小的运算次数。
我不确定是不是对 n 的操作次数有一个简单的刻画，　尝试着想了一会儿，　似乎不太容易想到。 但后来发现这个题目本质上不是算法题，　而是算法分析题。 因为仔细分析可以发现，题目中给的递归构造本身就是非常高效的。

直接按照题目中的操作描述可以写出函数：

```
int function(unsigned int n) {
if (n == 1) return 0;
if (n%2 == 0) return 1 + function(n/2);
return 1 + min(function((n + 1)/2), function((n - 1)/2));
}
```

在递归过程中，　每个节点可以引出一条或两条分支，　递归深度为，　所以总节点数为级别的，但为何还说此递归本身是非常高效的呢？

理解了动态规划的思想，　就很容易理解这里面的问题。　因为动态规划本质上就是保存运算结果的递归，　虽然递归算法经常会有指数级别的搜索节点，　但这些节点往往重复率特别高，　当保存每次运算的节点结果后，　在重复节点的计算时，　就可以直接使用已经保存过的结果，　这样就大大提高了速度 （每次不仅减少一个节点，而且同时消灭了这个节点后面的所有分支节点）。

在这个问题里是什么情况呢？仔细分析就会发现，　在整个搜索数中，　第层的节点只有两种可能性和。 这意味着整个搜索树事实上只有个节点。　所以这个递归算法本质上的运算复杂度只有。这已经是最优的了。

排序汇总

```
package com.softeem.jbs.lesson4;

import java.util.Random;

/**
 * 排序测试类
 *
 * 排序算法的分类如下：
 * 1. 插入排序（直接插入排序、折半插入排序、希尔排序）；
 * 2. 交换排序（冒泡泡排序、快速排序）；
 * 3. 选择排序（直接选择排序、堆排序）；
 * 4. 归并排序；
 * 5. 基数排序。
 *
 * 关于排序方法的选择：
 * (1) 若 n 较小 ( 如 n≤ 50)，可采用直接插入或直接选择排序。
 *     当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插人，应选直接选择排序为宜。
 * (2) 若文件初始状态基本有序 (指正序 )，则应选用直接插人、冒泡或随机的快速排序为宜；
 * (3) 若 n 较大，则应采用时间复杂度为 O(nlgn) 的排序方法：快速排序、堆排序或归并排序。
 *
 */
public class SortTest {

    /**
     * 初始化测试数组的方法
     * @return 一个初始化好的数组
     */
    public int [] createArray() {
        Random random = new Random();
        int [] array = new int [10];
        for (int i = 0; i < 10; i++) {
            array[i] = random.nextInt(100) - random.nextInt(100); // 生成两个随机数相减，保证生成的数中有负数
        }
        System. out .println( "========== 原始序列 ==========" );
        printArray(array);
        return array;
    }

    /**
     * 打印数组中的元素到控制台
```

```
     *  @param      source
     */
    public     void    printArray(    int  []   data)   {
        for   (int    i  :  data)   {
            System.   out .print(i     +  "  ");
        }
        System.   out  .println();
    }

    /**
     *  交换数组中指定的两元素的位置
     *  @param      data
     *  @param      x
     *  @param      y
     */
    private      void    swap(  int  []   data,    int    x,  int    y)   {
        int    temp    =   data[x];
        data[x]     =   data[y];
        data[y]     =   temp;
    }

    /**
     *  冒泡排序  ----  交换排序的一种
     *  方法：相邻两元素进行比较，如有需要则进行交换，每完成一次循环就将最大元素排在最后（如从小到大排序），下一次循环是将其他的数进行类似操作。
     *  性能：比较次数    O(n^2),n^2/2        ；交换次数   O(n^2),n^2/4
     *
     *  @param      data    要排序的数组
     *  @param      sortType     排序类型
     *  @return
     */
    public     void    bubbleSort(    int  []   data,    String    sortType)    {
        if   (sortType.equals(      "asc"  ))   {   // 正排序，从小排到大
            // 比较的轮数
            for   (int    i  =   1;   i  <   data.  length   ;  i++)    {
                // 将相邻两个数进行比较，较大的数往后冒泡
                for   (int    j  =   0;   j  <   data.  length    -  i;   j++)    {
                    if   (data[j]     >   data[j    +   1])   {
                        // 交换相邻两个数
                        swap(data,      j,   j   +   1);
                    }
                }
            }
        }   else    if   (sortType.equals(      "desc"  ))   {   // 倒排序，从大排到小
```

```
// 比较的轮数
for (int i = 1; i < data. length ; i++) {
    // 将相邻两个数进行比较，较大的数往后冒泡
    for (int j = 0; j < data. length - i; j++) {
        if (data[j] < data[j + 1]) {
            // 交换相邻两个数
            swap(data, j, j + 1);
        }
    }
} else {
        System. out .println( " 您输入的排序类型错误！ ");
    }
    printArray(data);       // 输出冒泡排序后的数组值
}

/**
 * 直接选择排序法 ---- 选择排序的一种
 * 方法：每一趟从待排序的数据元素中选出最小（或最大）的一个元素, 顺序放在已排好序的数列的最后，直到全部待排序的数据元素排完。
 * 性能：比较次数 O(n^2),n^2/2
 * 交换次数 O(n),n
 * 交换次数比冒泡排序少多了，由于交换所需 CPU 时间比比较所需的 CUP 时间多，所以选择排序比冒泡排序快。
 * 但是 N 比较大时，比较所需的 CPU 时间占主要地位，所以这时的性能和冒泡排序差不太多，但毫无疑问肯定要快些。
 *
 * @param data 要排序的数组
 * @param sortType 排序类型
 * @return
 */
public void selectSort( int [] data, String sortType) {

    if (sortType.equals( "asc" )) { // 正排序，从小排到大
        int index;
        for (int i = 1; i < data. length ; i++) {
index = 0;
for (int j = 1; j <= data. length - i; j++) {
    if (data[j] > data[index]) {
        index = j;

    }
}
// 交换在位置 data.length-i 和 index( 最大值 ) 两个数
```

```
                swap(data, data. length - i, index);
            }
        } else if (sortType.equals( "desc" )) { // 倒排序，从大排到小
            int index;
            for (int i = 1; i < data. length ; i++) {
                index = 0;
                for (int j = 1; j <= data. length - i; j++) {
                    if (data[j] < data[index]) {
                        index = j;
                    }
                }
                // 交换在位置 data.length-i 和 index( 最大值 ) 两个数
                swap(data, data. length - i, index);
            }
        } else {
            System. out .println( " 您输入的排序类型错误！ ");
        }
        printArray(data); // 输出直接选择排序后的数组值
    }

    /**
     * 插入排序
     * 方法：将一个记录插入到已排好序的有序表（有可能是空表）中 ,从而得到一个新的记录数增 1 的有序表。
     * 性能：比较次数 O(n^2),n^2/4
     * 复制次数 O(n),n^2/4
     * 比较次数是前两者的一般，而复制所需的 CPU 时间较交换少，所以性能上比冒泡排序提高一倍多，而比选择排序也要快。
     *
     * @param data 要排序的数组
     * @param sortType 排序类型
     */
    public void insertSort( int [] data, String sortType) {
        if (sortType.equals( "asc" )) { // 正排序，从小排到大
            // 比较的轮数
            for (int i = 1; i < data. length ; i++) {
                // 保证前 i+1 个数排好序
                for (int j = 0; j < i; j++) {
                    if (data[j] > data[i]) {
                        // 交换在位置 j 和 i 两个数
                        swap(data, i, j);
                    }
                }
```

```
                }
            } else if (sortType.equals("desc")) { // 倒排序，从大排到小
                // 比较的轮数
                for (int i = 1; i < data.length ; i++) {
                    // 保证前 i+1 个数排好序
                    for (int j = 0; j < i; j++) {
                        if (data[j] < data[i]) {
                            // 交换在位置 j 和 i 两个数
                            swap(data, i, j);
                        }
                    }
                }
            } else {
                System.out.println("您输入的排序类型错误！");
            }
            printArray(data); // 输出插入排序后的数组值
        }

        /**
         * 反转数组的方法
         * @param data 源数组
         */
        public void reverse(int[] data) {

            int length = data.length ;
            int temp = 0; // 临时变量

            for (int i = 0; i < length / 2; i++) {
                temp = data[i];
                data[i] = data[length - 1 - i];
                data[length - 1 - i] = temp;
            }
            printArray(data); // 输出到转后数组的值
        }

        /**
             * 快速排序
             * 快速排序使用分治法（Divide and conquer）策略来把一个序列（list）分为两个子序列（sub-lists）。
             * 步骤为：
             * 1. 从数列中挑出一个元素，称为"基准"（pivot），
             * 2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分割之后，该基准是它的最后位置。这个称为分割（partition）操作。
```

```
         * 3. 递归地（ recursive ）把小于基准值元素的子数列和大于基准值元素的子数列排序。
         * 递回的最底部情形，是数列的大小是零或一，也就是永远都已经被排序好了。虽然一直递回下去，但是这个算法总会结束，因为在每次的迭代（ iteration ）中，它至少会把一个元素摆到它最后的位置去。
         * @param data 待排序的数组
         * @param low
         * @param high
         * @see SortTest#qsort(int[], int, int)
         * @see SortTest#qsort_desc(int[], int, int)
         */
        public void quickSort( int [] data, String sortType) {
            if (sortType.equals( "asc" )) { // 正排序，从小排到大
                qsort_asc(data, 0, data. length - 1);
            } else if (sortType.equals( "desc" )) { // 倒排序，从大排到小
                qsort_desc(data, 0, data. length - 1);
            } else {
                System. out .println( " 您输入的排序类型错误！ ");
            }
        }

        /**
         * 快速排序的具体实现，排正序
         * @param data
         * @param low
         * @param high
         */
        private void qsort_asc( int data[], int low, int high) {
            int i, j, x;
            if (low < high) { // 这个条件用来结束递归
                i = low;
                j = high;
                x = data[i];
                while (i < j) {
                    while (i < j && data[j] > x) {
                        j--; // 从右向左找第一个小于 x 的数
                    }
                    if (i < j) {
                        data[i] = data[j];
                        i++;
                    }
                    while (i < j && data[i] < x) {
                        i++; // 从左向右找第一个大于 x 的数
                    }
                    if (i < j) {
```

```
                    data[j]     =   data[i];
                    j--;
                }
            }
            data[i]     =   x;
            qsort_asc(data,     low,    i   -   1);
            qsort_asc(data,     i   +   1,   high);
        }
    }

    /**
     *  快速排序的具体实现，排倒序
     *  @param      data
     *  @param      low
     *  @param      high
     */
    private     void    qsort_desc(     int     data[],     int     low,    int     high)    {
        int     i,  j,  x;
        if  (low    <   high)   {   // 这个条件用来结束递归
            i   =   low;
            j   =   high;
            x   =   data[i];
            while   (i  <   j)  {
                while   (i  <   j   &&  data[j]     <   x)  {
                    j--;    // 从右向左找第一个小于    x 的数
                }
                if  (i  <   j)  {
                    data[i]     =   data[j];
                    i++;
                }
                while   (i  <   j   &&  data[i]     >   x)  {
                    i++;    // 从左向右找第一个大于    x 的数
                }
                if  (i  <   j)  {
                    data[j]     =   data[i];
                    j--;
                }
            }
            data[i]     =   x;
            qsort_desc(data,     low,    i   -   1);
            qsort_desc(data,     i   +   1,   high);
        }
    }
```

```
/**
 * 二分查找特定整数在整型数组中的位置       ( 递归 )
 * 查找线性表必须是有序列表
 * @paramdataset
 * @paramdata
 * @parambeginIndex
 * @paramendIndex
 * @returnindex
 */
public int binarySearch( int [] dataset, int data, int beginIndex,
        int endIndex) {
    int midIndex = (beginIndex + endIndex) >>> 1; // 相当于 mid = (low + h
igh) / 2，但是效率会高些
    if (data < dataset[beginIndex] || data > dataset[endIndex]
            || beginIndex > endIndex)
        return -1;
    if (data < dataset[midIndex]) {
        return binarySearch(dataset, data, beginIndex, midIndex - 1);
    } else if (data > dataset[midIndex]) {
        return binarySearch(dataset, data, midIndex + 1, endIndex);
    } else {
        return midIndex;
    }
}

/**
 * 二分查找特定整数在整型数组中的位置       ( 非递归 )
 * 查找线性表必须是有序列表
 * @paramdataset
 * @paramdata
 * @returnindex
 */
public int binarySearch( int [] dataset, int data) {
    int beginIndex = 0;
    int endIndex = dataset. length - 1;
    int midIndex = -1;
    if (data < dataset[beginIndex] || data > dataset[endIndex]
            || beginIndex > endIndex)
        return -1;
    while (beginIndex <= endIndex) {
        midIndex = (beginIndex + endIndex) >>> 1; // 相当于 midIndex = (b
eginIndex + endIndex) / 2，但是效率会高些
        if (data < dataset[midIndex]) {
            endIndex = midIndex - 1;
```

```
                } else if (data > dataset[midIndex]) {
                    beginIndex = midIndex + 1;
                } else {
                    return midIndex;
                }
            }
            return -1;
        }

        public static void main(String[] args) {
            SortTest sortTest = new SortTest();

            int [] array = sortTest.createArray();

            System.out.println("========== 冒泡排序后(正序)==========");
            sortTest.bubbleSort(array, "asc");
            System.out.println("========== 冒泡排序后(倒序)==========");
            sortTest.bubbleSort(array, "desc");

            array = sortTest.createArray();

            System.out.println("========== 倒转数组后==========");
            sortTest.reverse(array);

            array = sortTest.createArray();

            System.out.println("========== 选择排序后(正序)==========");
            sortTest.selectSort(array, "asc");
            System.out.println("========== 选择排序后(倒序)==========");
            sortTest.selectSort(array, "desc");

            array = sortTest.createArray();

            System.out.println("========== 插入排序后(正序)==========");
            sortTest.insertSort(array, "asc");
            System.out.println("========== 插入排序后(倒序)==========");
            sortTest.insertSort(array, "desc");

            array = sortTest.createArray();
            System.out.println("========== 快速排序后(正序)==========");
            sortTest.quickSort(array, "asc");
            sortTest.printArray(array);
            System.out.println("========== 快速排序后(倒序)==========");
            sortTest.quickSort(array, "desc");
```

```
            sortTest.printArray(array);

            System.  out .println(  "==========      数组二分查找 ==========");
            System.  out .println(  "您要找的数在第 " + sortTest.binarySearch(array,      74)
                    + "个位子。（下标从 0 计算）");
        }
    }
```