递归

定义：程序调用自身的编程技巧称为递归（ recursion ）

条件：

1、递归需要有边界条件（一般写在最前面）

2、递归前进段。当边界条件不满足时，递归前进

3、递归返回段，当边界条件满足时，递归返回。

递归代码的一般代码流程：

伪代码示例：

```
def recursion(level, param1, param2, ...):

    # recursion terminator
    if level > MAX_LEVEL:
      print_result
      return

    # process logic in current level
    process_data(level, data...)

    # drill down
    self.recursion(level + 1, p1, ...)

    # reverse the current level status if needed
    reverse_state(level)
```


递归的缺点：

递归算法解题相对常用的算法如普通循环等，运行效率较低。因此，应该尽量避免使用递归，除非没有更好的算法或者某种特定情况，递归更为适合的时候。在递归调用的过程当中系统为每一层的返回点、局部量等开辟了栈来存储。递归次数过多容易造成栈溢出等。在运算过程中有些子过程的可能会重复计算，此时最好的方式是加入缓冲变量，记录曾经计算过的子节点

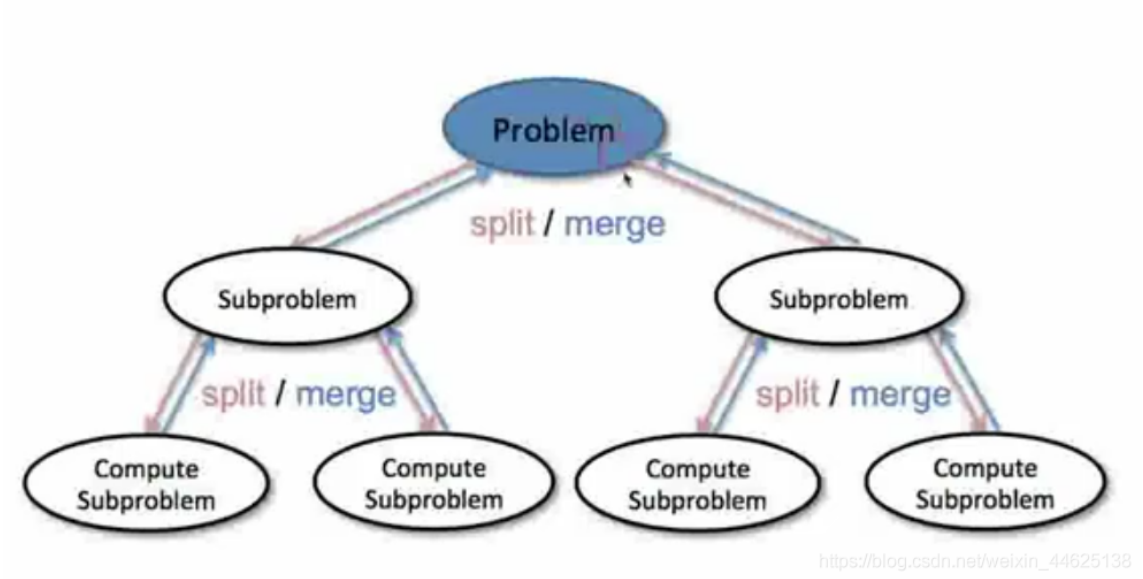
迭代:利用变量的原值推算出变量的一个新值.如果递归是自己调用自己的话,迭代就是A不停的调用B.

迭代虽然效率高，运行时间只因循环次数增加而增加，没什么额外开销，空间上也没有什么增加，但缺点就是不容易理解，编写复杂问题时困难

分治

定义：分治算法的基本思想是将一个规模为N的问题分解为K个规模较小的子问题，这些子问题相互独立且与原问题性质相同。求出子问题的解，就可得到原问题的解。即一种分目标完成程序算法，简单问题可用二分法完成（子问题不会出现重复的计算，子问题之间互不干扰）

分治的示意图



分治步骤的伪代码：

```
def divide_conquer(problem, param1, param2, ...):

  # recursion terminator
  if problem is None:
    print_result
    return

  # prepare data
  data = prepare_data(problem)
  subproblems = split_problem(problem, data)

  # conquer subproblems
  subresult1 = self.divide_conquer(subproblems[0], p1, ...)
  subresult2 = self.divide_conquer(subproblems[1], p1, ...)
  subresult3 = self.divide_conquer(subproblems[2], p1, ...)
  ...

  # process and generate the final result
  result = process_result(subresult1, subresult2, subresult3, ...)
```

2.2 如何设计递归算法

1.确定递归公式

2.确定边界（终了)条件

典型的递归算法：fibonacia函数：

典型例题：

1、快速排序

public class QuickSort {

```
1         public static void quickSort1(int[] arr) {
2                 if(arr==null || arr.length<=1) return;
3                 sort2(arr, 0, arr.length-1);
```

```
4
5         }
6
7          static void sort1(int[] arr,int left,int right) {
8                 // TODO Auto-generated method stub
9                 int i=left;
10                int j=right;
11                int t;
12                int temp;   //基准数
13                if(left>right) return;
14                temp=arr[left];
15                while(i!=j) {
16
17                        ///顺序很重要，要先从右边开始找，因为是以左边为基准数
18                        while(arr[j]>=temp && j>i) {  // j>i控制j的范围
19                                j--;
20                        }
21
22                        if(i<j) {
23                                arr[i++]=arr[j];
24
25                        }
26                        while(arr[j]<=temp && j>i) {  // j>i控制j的范围
27                                i++;
28                        }
29                        if(i<j) {
30                                arr[j--]=arr[i];
31
32                        }
33
34                }
35                arr[i]=temp;
36
37                sort1(arr,left,i-1);
38                sort1(arr,i+1,right);
39        }
40
41         public static void quickSort2(int[] arr) {
42                        if(arr==null || arr.length<=1) return;
43                        sort2(arr, 0, arr.length-1);
```

```
44
45                  }
46                  private static void sort2(int[] arr,int left,int right) {
47                          // TODO Auto-generated method stub
48                          int i=left;
49                          int j=right;
50                          int t;
51                          int temp;   //基准数
52                          if(left>right) return;
53                          temp=arr[left];
54                          while(i!=j) {
55                                  ///顺序很重要，要先从右边开始找，因为是以左边为基准数
56                                  while(arr[j]>=temp && j>i) {  // j>i控制j的范围
57                                          j--;
58                                  }
59                                  while(arr[i]<=temp && j>i) {  // j>i控制j的范围
60                                          i++;
61                                  }
62                                  if(i<j) {
63                                          t=arr[i];
64                                      arr[i]=arr[j];
65                                      arr[j]=t;
66                                  }
67                          }
68                          //将基准数交换到两者相遇的位置
69                          arr[left]=arr[i];
70                          arr[i]=temp;
71                          //
72
73                          sort2(arr,left,i-1);
74                          sort2(arr,i+1,right);
75                  }
76
```

}

2、实现 pow(x, n) ，即计算 x 的 n 次幂函数。(https://leetcode-cn.com/problems/powx-n/)

public class ImplementPow {

```
1    //方法一： 直接循环法 ,此方法在leetcode上会超时
2  public double myPow(double x, int n) {
```

```
    double sum=1;

    if(n<0 && x!=0){
        n=-n;
        x=1/x;
    }
    while(n>0){
        sum=x*sum;
        n--;
    }
    return sum;
}

//方法二：递归和分治法，将问题分成很多子问题；
public double myPow1(double x, int n) {
        if(n<0 && x!=0){
        n=-n;
        x=1/x;
    }
        return helper(x, n);
}
public double helper(double x, int n) {
        if (n==0) {
                return 1.0;  //注意此处返回1.0不是1；
        }
        double half=helper(x, n/2);
        //分奇偶的情况；
        if(n%2==0){
                return half*half;
        }else {
                return half*half*x;
        }
}

}
```

3、给定一个大小为 n 的数组，找到其中的众数。众数是指在数组中出现次数大于 [n/2] 的元素。你可以假设数组是非空的，并且给定的数组总是存在众数。（https://leetcode-cn.com/problems/majority-element/）

package recurseAnddivide;

import java.util.Arrays;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import javax.swing.text.Highlighter.Highlight;

import bfsAnddfs.LowestCommonAncestor;

public class MajorElement {

```java
//方法1：对数组排序
public int majorityElement(int[] nums) {
    Arrays.sort(nums);
    return nums[nums.length/2];

}

//方法二：暴力法，出现的次数动态跟nums/2比较
public int majorityElement2(int[] nums) {

    int majorCount=nums.length/2;
    for (int i : nums) {
                int count=0;
                for (int j : nums) {
                        if (i==j) {
                                count++;
                        }
                }
                if (count>majorCount) {
                        return i;
                }
        }
        return -1;
}

//方法三：哈希表，统计每个数字出现的次数
//时间复杂度：

```

// 时间：O(n);

// 空间：O(n),哈希表最多包含 n - \lfloor \dfrac{n}{2} \rfloorn−⌊ 2n ⌋ 个关系，所以占用的空间为 O(n)，但题中保证 nums 一定有一个众数，会占用（最少） \lfloor \dfrac{n}{2} \rfloor + 1⌊ 2n⌋+1 个数字。因此

最多有 n - (\lfloor \dfrac{n}{2} \rfloor + 1)n−([2n]+1) 个不同的其他数字，所以最多有 n - \lfloor \dfrac{n}{2} \rfloorn−[2n] 个不同的元素

```
public int majorityElement3(int[] nums) {

    int majorCount=nums.length/2;
    Map<Integer, Integer> count=new HashMap();
    for (int i : nums) {
                count.put(i, count.containsKey(i)?count.get(i)+1:1);
        }
        //遍历MAp
    for(Integer key:count.keySet()){
        if (count.get(key)>majorCount) {
                        return key;
                }
    }
    return -1;

}

//方法四：分治：
public int majorityElement4(int[] nums) {

    return helper(nums, 0, nums.length);
}
 private int helper(int[] nums,int Low,int high){

         if (Low==high) return nums[Low];
         int mid=(high-Low)/2+Low;
         int left=helper(nums, Low, mid);
         int right=helper(nums, mid+1, high);
         //计算两者出现的次数
         if (left==right) {
                return left;
        }
          int leftCount = countInRange(nums, left, Low, high);
      int rightCount = countInRange(nums, right, Low, high);
      return leftCount>rightCount?left:right;

 }
```

```
38  private int countInRange(int[] nums, int num, int lo, int hi) {
39      int count = 0;
40      for (int i = lo; i <= hi; i++) {
41          if (nums[i] == num) {
42              count++;
43          }
44      }
45      return count;
46  }
47
```

}
