第三次书面作业

1. 归并排序

   归并排序（merge sort）是一种常用的o(nlogn)的排序方法，简单好写，适合初学者手打。相比快速排序，归并排序有更稳定的最差复杂度，但是需要多开o(n)的空间。

   归并排序采用了**分治法**的思想。对一个乱序的数组s（假设我们从1存储到n），我们将数组分成等长的两部分，对这两部分都进行**归并排序**（注意，这里已经有**递归**的行为了），然后我们得到了左和右两个**已经有序**的数组。然后，我们将这两个数组**归并**成一个有序的数组。注意，这个归并的过程的速度决定了整个排序的速度。

   关于递归的出口，类似于二分法，思考：什么情况下我们得到的数组是不用排序的？

   根据以上提示，完成归并排序的代码。

```c
#include <stdio.h>

int s[10010],temp[10010];//数组足够大

void merge_sort(int l ,int r);

int main(){
  int n;
  scanf("%d" , &n);
  for(int i = 1 ; i <= n ; i++) scanf("%d" , s + i);
  merge_sort(1 , n);
  printf("排序后的数组：\n");
  for(int i = 1 ; i <= n ; i++) printf("%d " , s[i]);
  printf("\n");
}

void merge_sort(int l ,int r){
  //函数意义：将数组s中从l到r的下标进行升序排序。
  if(___(1)___) return;
```

```
  int mid = ( l + r ) / 2;
  merge_sort(l , mid);
  merge_sort(mid + 1 , r);//递归
  for(int i = l; i <= r ; i++) t[i]=s[i];
  //最重要的一步是归并：现在，t[l]到t[mid]已经升序，t[mid+1]到t[r]也
升序
  //你需要将这两部分合并成一个升序数组存到s中，使得s[l]到s[r]升序
  //要求：你可以分成多轮循环，但是循环的总次数应该约等于r-l
  int p1 = l , p2 = mid + 1;
  ___(2)___
  ...
  _________
}
```

2. 括号匹配问题

   栈（stack）是人们发现的一种重要的数据结构。栈遵守**后进先出**原则（last in first out，LIFO）。

   想象一个生活中常见的例子：大家都见过羽毛球筒，一般情况下我们只开一边的口子。在这种情况下，我们只能取出最顶上的羽毛球，或者把新的羽毛球放到最顶上，并且后放进去的球会先取出来。

   我们可以简单用数组模拟一个栈：一般我们定义一个stack数组，和一个top变量表示栈顶元素的下标。存入时我们将top自增1，然后向stack[top]里存入元素；删除时，将top自减1。

   栈是人们在长期的现实经验中总结出来的数据结构，它没有什么深刻的算法设计，但是却在很多问题中都能找到对应，比如括号匹配问题。

   在接下来的题目中，你要对一个**只含有括号的字符串**进行匹配。注意，可能会出现`()`，`{}`,`[]`三种括号，**我们不考虑他们的优先级顺序**，但是我们需要验证它们是不是匹配。

   比如:

```
([]{})
(([])(){})
()[]{}()
```

是匹配的，

```
[)
(((()))))
((((((()))
[{](){}
```

是不匹配的。

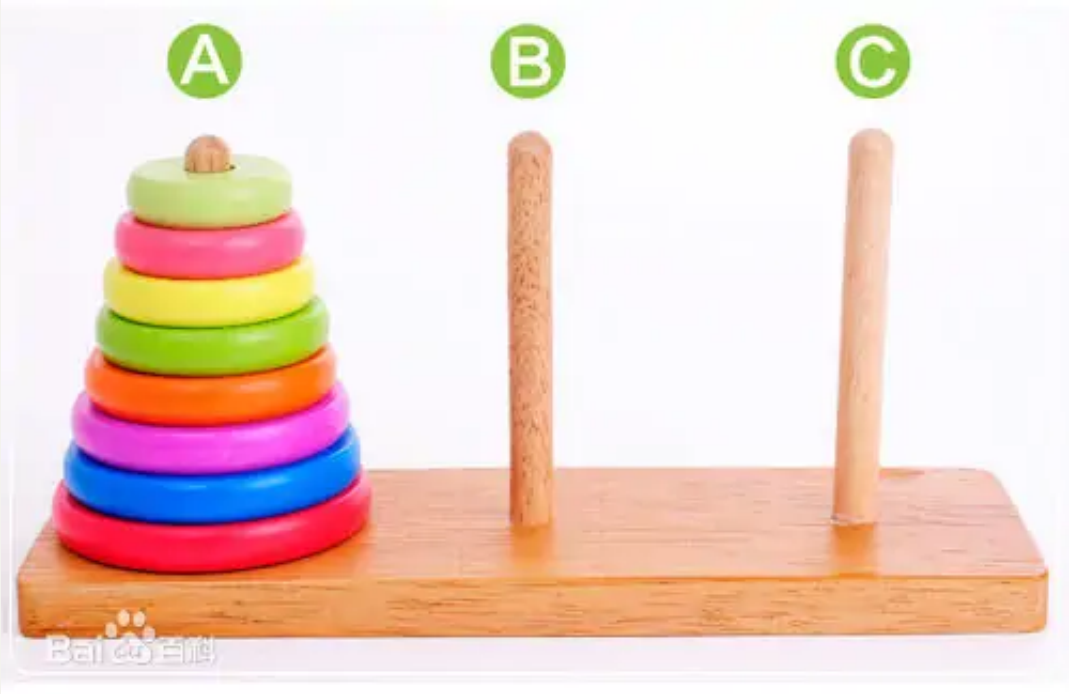
根据以上关于栈的提示，手动模拟以下上面的几组例子，完成以下代码：

```
#include <stdio.h>
#include <string.h>

char stack[10010],s[10010];
int top=0;

void push(char p){
  //将栈顶向上移一位，并将元素p存入栈顶
  ___(1)___;
}
void pop(){
  //将栈顶元素删除，栈顶下降一位
  ___(2)___;
}
char get_top(){
  //给出栈顶元素的值
  return stack[top];
}
int is_empty(){
  //返回1表示栈空，0表示栈非空。
  return top==0;
}
int check(char l,char r){
  //判断一对括号是否匹配
  return l=='('&&r==')' || l=='['&&r==']' || l=='{'&&r=='}';
}
```

```
int main(){
  scanf("%s",stack);
  int len=strlen(s);
  for(int i=0;i<len;i++){
    char c=s[i];
    if(c=='('||c=='['||c=='{'){
      ___(3)___
      ...
      _________
    }
    else{
      ___(4)___
      ...
      _________
    }
  }
  //输出结果：括号是否匹配。
  ___(5)___
  ...
  _________
}
```

3. 汉诺塔问题

   印度传说中有汉诺塔：塔有三座，第一座上由上到下堆着64块盘片。要在遵守“小的盘片不能放在大的盘片的下面”的规则下，将第一座塔上的所有盘片移动到第三座上。据说，成功后，宇宙将会毁灭。



这里，我们给出n曾汉诺塔问题的递归解法。

- (a)先给所有盘片定一个编号，假设最大的为n，最小的为1。我们要将1到n的盘片从当前塔A上移动到目标塔C上，其中我们可以使用中继塔C，来暂存一些盘片。

但是，要移动n号盘，必须将1到n-1号盘先移动走。那么，一个直观的方法就是：

- (b)将1到n-1的盘片从A移动到B上，这个时候可以以C作为中继，因为C上面不存在1到n-1的盘，不会违反规则。这时，任务(a)里的中继变成了目标，但是(a)的目标变成了中继。
- (c)将n号盘从A移动到C上。
- (d)现在1到n-1都在B上，要移动到C上，这时可以用A作为中继，不会违反规则。这时，任务(a)里的起始、目标和中继在任务(b)中也发生了相应的变化，请自行归纳。

观察任务(b)和任务(d)，对比任务(a)，我们发现，其实这三个问题的做法都是一样的，只是他们的阶数、起始、目标和中继都不一样。另外，我们在解决1到n-1规模问题的时候，不用去管n以上的盘片，因为它们在之前的步骤中都压在下面了。所以这个解法其实是递归的。

希望同学们通过以上例子来理解递归的含义和使用场景。递归是自己调用自己，为什么这么做呢，因为要解决的问题的方法是相同的，但是具体的变量可能不同。同时，递归也要有出口，比如汉诺塔问题的出口就比较简单。递归的奇妙之处就在于，只要递归的函数正确、出口也正确，那么整个过程就是正确的，就像大家常用的数学归纳法一样。因此递归的重要思想就是，从问题中提炼出相同形式的子问题，然后考虑出正确的递归出口。

请根据以上提示，补全下列代码。

```c
#include <stdio.h>

void hanoi(int n, char src, char dst, char mid){
  //函数意义：要将1到n的盘，从初始塔src，移动到目标塔dst，可以用mid塔来暂存
  if(___(1)___) printf("%c => %c\n",src,dst);
  else{
    ___(2)___
    printf("%c => %c\n",___(3)___);
    ___(4)___
  }
}

int main(){
  int n;
  scanf("%d",&n);
  hanoi(n,'A','B','C');
  return 0;
}
```

要求：每一空只能填一行代码。你会发现，递归版的汉诺塔代码量非常小。

同样地，大家在学习完链表之后可以尝试将链表问题也用上面的方法转换为递归的方法，可以有效减少代码量，增加可读性。

大家可以尝试用比较小的数字来验证答案的正确性。当n=64时，移动的步数时2的64次方减去1，其规模已经远远超出了宇宙的存在时间。