数学

07 斐波那契数列

- **题目**
  大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项（从0开始，第0项为0）。
  n<=39
- **思路**
  找规律，可以发现f(3)=f(1)+f(2)，f(4) = f(2)+f(3)...用循环可以算出来。
- **基础概念**
  斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
  a1=1，a2=1，an=an-1+an-2（n>=3,n∈N*）

- **代码实现**

```
class Solution {
public:
    int Fibonacci(int number) {

        int f_low = 0 ;
        int f_high = 1;
        int total = number;
        for(int i = 2, temp; i <= number; i++){
            total = f_low +f_high;
            f_low = f_high;
            f_high = total;
        }
        return total;
    }
};
```

08 跳台阶

- **题目**
  一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法（先后次序不同算不同的结果）。
- **思路**
  第一种：递归思想，每次return f(n - 1)+f(n - 2)。
  第二种：找规律，可以发现f(3)=f(1)+f(2)，f(4) = f(2)+f(3)...用循环可以算出来。
  递归调用函数，转成汇编语言后，消耗的时间占用很大。（递归460ms，该方法3ms）
- **代码实现**

```
//递归
class Solution {
public:
    int jumpFloor(int number) {
        switch(number){
            case 0:return 0;
            case 1:return 1;
```

```
            case 2:return 2;
            default:return jumpFloor(number-1)+jumpFloor(number-2);
        }
    }
};
```

```
//使用规律循环求解
class Solution {
public:
    int jumpFloor(int number) {
        int f_low = 0 ;
        int f_high = 1;
        for(int i = 1, temp; i <= number; i++){
            temp = f_high;
            f_high = f_low +f_high;
            f_low = temp;
        }
        return f_high;
    }
};
```

09 变态跳台阶

- **题目**
  一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。
- **思路**
  找规律，可以发现f(n)= 2的n-1次方。
- **代码实现**

```
class Solution {
public:
    int jumpFloorII(int number) {
        int total = 1;
        return total << number - 1;
    }
};
```

10 矩阵覆盖

- **题目**

  我们可以用2*1*的小矩形横着或者竖着去覆盖更大的矩形。请问用*n*个2*1*的小矩形无重叠地覆盖一个2*n的大矩形，总共有多少种方法？

- **思路**

  找规律，遇到这种麻烦，试试几个数是否有规律。

  斐波那契数列思想

  下面介绍来自一个牛人的思路。

  
  Follow

  依旧是斐波那契数列

  2*n的大矩形，和n个2*1的小矩形

  其中target*2为大矩阵的大小

  有以下几种情形：

  1target <= 0 大矩形为<= 2*0,直接return 1；

  2target = 1大矩形为2*1，只有一种摆放方法，return1；

  3target = 2 大矩形为2*2，有两种摆放方法，return2；

  4target = n 分为两步考虑：

  第一次摆放一块 2*1 的小矩阵，则摆放方法总共为f(target - 1)

  第一次摆放一块1*2的小矩阵，则摆放方法总共为f(target-2)

  因为，摆放了一块1*2的小矩阵（用√√表示），对应下方的1*2（用××表示）摆放方法就确定了，所以为f(targte-2)

  

  想法是这样的，第一种，竖着放在最左边，剩下的就是 2（*n-1*）空间，即 *f(target -1)*

  第二种横着放在最右边，这时候必须再横着放一个，剩下的就是 2（n-2）空间，即f(target -2)

  所以f(n) = f(n-1)+f(n-2)

- **代码实现**

```
class Solution {
public:
    int rectCover(int number) {
        if(number < 1)
            return 0;
        int f_high = 1,f_low = 0;
        for(int i = 1,temp; i <= number;i++){
```

```
            temp = f_high;
            f_high = f_high + f_low;
            f_low = temp;
        }
        return f_high;
    }
};
```

11 二进制中1的个数【1】【对正负数可自适应】

- **题目**
  输入一个整数，输出该数二进制表示中1的个数。其中负数用补码表示。
- **思路**
  不管是整数和负数，取一个无符号整形值flag为1，number与flag进行位运算，并对1的个数进行计数，然后flag左移（直到flag为0循环结束）
- **基础概念**
  8的原码为 00001000
  16的原码为 10001000
  正整数的补码是其二进制表示，与原码相同。
  负整数的补码，将其原码除符号位外的所有位取反（0变1，1变0，符号位为1不变）后加1。
- **代码实现**

```
class Solution {
public:
     int NumberOf1(int n) {
        int total= 0;
         unsigned int flag = 1;
         while( flag != 0){
             if(n & flag){        //与运算，判断该位上是否为1
                 total++;
             }
             flag <<= 1;
         }
         return total;
     }
};
```

12 数值的整数次方

- **题目**
  给定一个double类型的浮点数base和int类型的整数exponent。求base的exponent次方。
- **思路**
  1.全面考察指数的正负、底数是否为零等情况。
  2.写出指数的二进制表达，例如13表达为二进制1101。
  3.举例:10^1101 = 10^0001*10^0100*10^1000。
  4.通过&1和>>1来逐位读取1101，为1时将该位代表的乘数累乘到最终结果。
- **代码实现**

```
class Solution {
public:
    double Power(double base, int exponent) {
        double result = 1;
        int base_cp = base,expon;

        if(exponent > 0){
            expon = exponent;
        }else if(exponent < 0){
            if( base = 0){
                printf("error!\n");
                exit(-1);
            }
            expon = - exponent;
        }else
            return 1;
        while( expon!= 0){
            if(expon&1){
                result *= base_cp;
            }
            base_cp*=base_cp;
            expon >>= 1;
        }

        return(exponent > 0? result:1/result);
    }
```

```
};
```

31 整数中1出现的次数（从1到n整数中1出现的次数）

- **题目**
  求出1~13的整数中1出现的次数,并算出100~1300的整数中1出现的次数？为此他特别数了一下1~13中包含1的数字有1、10、11、12、13因此共出现6次,但是对于后面问题他就没辙了。ACMer希望你们帮帮他,并把问题更加普遍化,可以很快的求出任意非负整数区间中1出现的次数（从1 到 n 中1出现的次数）。
- **思路**
  这个背代码。
- 代码实现

```
class Solution {
public:
    int NumberOf1Between1AndN_Solution(int n){
        int count = 0;
        for(long long i = 1;i <= n; i *= 10){
            //i表示当前分析的是哪一个数位
            int a = n/i,b = n%i;
            count += (a + 8)/10 * i + (a%10 == 1)*(b + 1);
        }
        return count;
    }
};
```

33 丑数

- **题目**
  把只包含质因子2、3和5的数称作丑数（Ugly Number）。例如6、8都是丑数，但14不是，因为它包含质因子7。 习惯上我们把1当做是第一个丑数。求按从小到大的顺序的第N个丑数。
- **思路**
  丑数p = 2 ^ x * 3 ^ y * 5 ^ z，换句话说一个丑数一定由另一个丑数乘以2或者乘以3或者乘以5得到
  所以首先把丑数表达式构建出来，x,y,z为一个计数器，通过三个都加1进行计

算，找出那个最小的，然后对应的计数器加1。

- 代码实现

```
class Solution {
public:
    int GetUglyNumber_Solution(int index) {
        if(index < 7) // 0-6的丑数分别为0-6
            return index;
        vector<int> res(index);
        res[0] = 1;
        int t2 = 0, t3 = 0, t5 = 0;
        for(int i = 1; i < index; ++i){
            res[i] = min(res[t2] * 2, min(res[t3] * 3, res[t5] * 5));
//选出下一个最小的丑数，肯定是由之前的一个丑数乘以2或3或5
            if(res[i] == res[t2] * 2)  //找出对应乘的数，并且计数器加1
                ++t2;
            if(res[i] == res[t3] * 3)
                ++t3;
            if(res[i] == res[t5] * 5)
                ++t5;
        }
        return res[index - 1];
    }
};
```

46 圆圈中最后剩下的数【2】

- **题目**
  首先,让小朋友们围成一个大圈。然后,他随机指定一个数m,让编号为0的小朋友开始报数。每次喊到m-1的那个小朋友要出列唱首歌,并且不再回到圈中,从他的下一个小朋友开始,继续0...m-1报数....这样下去....直到剩下最后一个小朋友。请你试着想下,哪个小朋友会得到这份礼品呢？(注：小朋友的编号是从0到n-1)
- **思路**
  第一种，相当于约瑟夫环问题，定义分arr[n]。开始时，p++，如果p等于n的话p赋为0；如果遇到退出的，则回到上一步跳过该人；step++;step等于m的退出并把arr[i] = -1，step = 0，总人数减1;此方法时间复杂度很大，很容易超时，但是思路比较常规，得掌握。

第二种，采用数学归纳法，得出 f(n,m) = (f(n-1，m) + m) %n，记住有这个方法吧，以后直接拿来用。

- 代码实现

```
class Solution {
public:
    int LastRemaining_Solution(int n, int m)
    {
        if( n < 1|| m < 1)
            return -1;
        int* arr = new int[n];
        int p = -1,step = 0,num = n;
        while(num > 0){
            p++;
            if(p >= n)
              p = 0;
            if(arr[p]== -1){
                continue;
            }
            step++;
            if(step == m){
                arr[p] = -1;step = 0;num--;
            }
        }
        return p;
    }
};
//了解背一下
class Solution {
public:
    int LastRemaining_Solution(int n, int m)//利用数学分析得到的解法
    { //f(n,m)=[f(n-1,m)+m]%n，其中f(n,m)为长度为n的删除第m个节点，最后剩下的数字
        if(n<=0||m<=0)
            return -1;
        int last=0;
        for(int i=2;i<=n;i++)
            last=(last+m)%i;
        return last;
```

```
    }
};
```

47 求多个数的和不用判断和乘除

- **题目**
  求1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。
- **思路**
  主要是怎么完成判断，采用递归去运算，通过ans && ans +=sum(n-1)来判断结束和运算和。当ans为0的时候，就不会运行&&右边的程序。
- 代码实现

```
class Solution {
public:
    int Sum_Solution(int n) {
        n && (n += Sum_Solution(n-1));
        return n;
    }
};
```

48 不用加减乘除做加法

- **题目**
  写一个函数，求两个整数之和，要求在函数体内不得使用+、-、*、/四则运算符号。
- **思路**
  首先10进制的计算过程是这样子的，首先计算个位相加，有进位则加到十位
  二进制加法，先通过与或运算得出每一位相加的和t1，再通过与运算并左移一位得出需要进位的结果t2；t1与t2再进行异或运算，再与运算，为0则结束，否则继续以上循环。
- 代码实现

```
class Solution {
public:
    int Add(int num1, int num2){
```

```
        if(!num1)
            return num2;
        while(num2){  //num1保存不需进位的值，num2保存进位的值
            int temp_1 = num1 ^ num2;
            num2 = (num1 & num2) << 1; //进位的值
            num1 = temp_1; //非进位的值
        }
        return num1;
    }
};

```

数据结构

05 用两个栈实现队列

- **题目**
  用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。
- **思路**

1. 栈1负责进队列，栈2负责出队列。
2. 出队列的时候，若栈2为空，则栈1依次出栈存到栈2。
3. 栈2出栈一个值，即为出队列的值。

关于两个队列处理一个栈的方法：
1.入栈：判断队列A和队列B哪个不为空，则把元素放入队列
2.出栈：判断哪个队列不为空，则出队列到另一个队列，直至留下一个元素出队列并出栈。

- 代码实现

```
class Solution
{
public:
    void push(int node) {
        stack1.push(node);
    }

```

```
    int pop() {
        if(stack2.empty()){
            while(!stack1.empty()){
                stack2.push(stack1.top());
                stack1.pop();
            }
        }
        int a = stack2.top();
        stack2.pop();
        return a;
    }

private:
    stack<int> stack1;
    stack<int> stack2;
};
```

20 包含min函数的栈【1】【辅助栈的栈顶保存当前最小值】

- **题目**
  定义栈的数据结构，请在该类型中实现一个能够得到栈中所含最小元素的min函数（时间复杂度应为O（1））。
- **思路**

1. 定义栈A，栈B，其中B为辅助栈，栈顶保存最小值
2. 入栈：A每个值都入栈。B对每次入栈的值判断，如果小于栈顶值，就入栈（为空时也入栈）。
3. 出栈：如果A与B的栈顶值相同，则AB都出栈；否则A出栈。

- 代码实现

```
class Solution {
public:
    void push(int x) {
        s1.push(x);
        if(s2.empty())
            s2.push(x);
        else if(x <= s2.top())
```

```
            s2.push(x);
    }
    void pop() {
        if(s1.top()==s2.top()){
            s1.pop();
            s2.pop();
        }
        else
            s1.pop();
    }
    int top() {
        return s1.top();
    }
    int min() {
        return s2.top();
    }
private:
    stack <int> s1,s2;
};
```

21 栈的压入与弹出序列

- **题目**
  输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否可能为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列1,2,3,4,5是某栈的压入顺序，序列4,5,3,2,1是该压栈序列对应的一个弹出序列，但4,3,5,1,2就不可能是该压栈序列的弹出序列。（注意：这两个序列的长度是相等的）
- **思路**
  加入一个辅助栈，循环把压入顺序A放入辅助栈，每压入一个判断辅助栈的栈顶与出栈序列比较，若相等辅助栈一直pop()，并且加个判断辅助栈是否为空的限制，防止为空时读栈顶错误。最后根据辅助栈是否为空判断弹出序列能匹配栈的压入
- 代码实现

```
class Solution {
public:
    bool IsPopOrder(vector<int> pushV,vector<int> popV) {
        stack<int> s;
```

```
        int len = pushV.size();
        for(int i = 0,j = 0;i < len;i++){
            s.push(pushV[i]);
            while(!s.empty()&&s.top()==popV[j]){
                s.pop();
                j++;
            }
        }
        return s.empty();
    }
};
```

29 最小的k个数【2】【堆】

- **题目**
  输入n个整数，找出其中最小的K个数。例如输入4,5,1,6,2,7,3,8这8个数字，则最小的4个数字是1,2,3,4。
- **基础知识**

```
#include <algorithm>

vector <int> res = {1,3,5,4,2,7};
/* 建大顶堆 */
make_heap(res.begin(), res.end()); //默认建大顶堆
//打印结果：7 4 5 3 2 1 （7位堆顶值）
//make_heap(res.begin(), res.end(), greater<int>());  小顶堆，注意第三个参数 greater<int>() （需要头文件functional），
/- 添加元素 -/
res.push_back(8);
push_heap(res.begin(), res.end());
//打印结果：8 4 7 3 2 1 5
/- 删除元素 -/
pop_heap(res.begin(), res.end());
res.pop_back();
//打印结果：7 4 5 3 2 1
/- 排序 -/
sort_heap(res.begin(), res.end());
//打印结果：1 2 3 4 5 7
```

- **思路**
  大堆还是小堆的选择很重要，不是寻找最小的k个元素就要选择小堆，而且恰恰相反。寻找最小的k个数，其实就是寻找第k个大的元素，即寻找k个数中最大的，不断调整堆，堆得元素个数是k，堆顶是最大值，遍历完初始数组后，堆中存在的元素即使我们所要寻找的k个最小元素。
  建立k长度的大顶堆，遍历数组剩余的值，与堆顶比较，若小于堆顶，则堆顶弹出，压入该值。时间复杂度为nlogk
  该题还可以通过快排，挑选出最小的k个。nlogn
- 代码实现

```
class Solution {
public:
    vector<int> GetLeastNumbers_Solution(vector<int> input, int k) {
        int len = input.size();
        if(len < k || len == 0 || k <= 0)
            return {};
        vector<int> res(input.begin(), input.begin() + k);
        make_heap(res.begin(), res.end());  //无第三个参数默认建立大顶堆
        for(int i = k; i < len; ++i){
            if(res[0] > input[i]){
                pop_heap(res.begin(), res.end()); //把堆顶值换到数组最后位置，第二大的值换到堆顶
                res.pop_back();
                res.push_back(input[i]);
                push_heap(res.begin(), res.end()); //重新调整堆
            }
        }
        sort_heap(res.begin(), res.end()); //从小到大排序
        return res;
    }
};
```

63 数据流的中位数【2】【优先队列】

- **题目**
  如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么

中位数就是所有数值排序之后中间两个数的平均值。我们使用Insert()方法读取数据流，使用GetMedian()方法获取当前读取数据的中位数。

- **基础知识**

  priority_queue是优先队列，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先队列具有最高级先出的行为特征。

  > 和queue不同的就在于我们可以自定义其中数据的优先级，让优先级高的排在队列前面,优先出队。
  > 优先队列具有队列的所有特性，包括队列的基本操作，只是在这基础上添加了内部的一个排序，它本质是一个堆实现的。

```
//升序队列
priority_queue <int,vector<int>,greater<int> > q;
//降序队列
priority_queue <int,vector<int>,less<int> >q;

//greater和less是std实现的两个仿函数（就是使一个类的使用看上去像一个函数。其实现就是类中实现一个operator()，这个类就有了类似函数的行为，就是一个仿函数类了）
```

- **思路**
- 代码实现

```
class Solution {
    priority_queue<int, vector<int>, less<int> > p; //降序，保存较小的值
    priority_queue<int, vector<int>, greater<int> > q; //升序，保存较大的值
public:
    void Insert(int num){
        p.empty() || num <= p.top() ? p.push(num) : q.push(num);
        //实现 q <= p << q+1
        if(p.size() == q.size() + 2) //相当于p.size()-q.size()<=1
            q.push(p.top()), p.pop();
        if(p.size() + 1 == q.size()) //相当于q.size() - p.size() <=0
            p.push(q.top()), q.pop();
    }
    double GetMedian(){
```

```
        return p.size() == q.size() ? (p.top() + q.top()) / 2.0 :
p.top();
    }
};
```

64 滑动窗口的最大值【3】【deque】

- **题目**
  给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及给定滑动窗口的大小3，那么一共存在6个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}；
  针对数组{2,3,4,2,6,2,5,1}的滑动窗口有以下6个：{[2,3,4],2,6,2,5,1}，{2,[3,4,2],6,2,5,1}，{2,3,[4,2,6],2,5,1}，{2,3,4,[2,6,2],5,1}，{2,3,4,2,[6,2,5],1}，{2,3,4,2,6,[2,5,1]}。
- **思路**
  首先得理解题意，给的滑动窗口大小3表示放三个元素，最大值为当前活动窗口中的最大值。
  采用一个双向队列deque
  以数组{2,3,4,2,6,2,5,1}为例，来细说整体思路。
  数组的第一个数字是2，把它存入队列中。第二个数字是3，比2大，所以2不可能是滑动窗口中的最大值，因此把2从队列里删除，再把3存入队列中。第三个数字是4，比3大，同样的删3存4。此时滑动窗口中已经有3个数字，而它的最大值4位于队列的头部。（在队列里存入的是数字在数组里的下标）
  第四个数字2比4小，但是当4滑出之后它还是有可能成为最大值的，所以我们把2存入队列的尾部。下一个数字是6，比4和2都大，删4和2，存6。就这样依次进行，**最大值永远位于队列的头部。**
- 代码实现

```
class Solution {
public:
    vector<int> maxInWindows(const vector<int>& num, unsigned int
size){
        if(!size || num.size() < size)
            return{};
        vector<int> res;
        deque<int> s;
        for(unsigned int i = 0; i < num.size(); ++i){
```

```
            //从后面依次弹出队列中小于等于当前num的元素（同时也能保证队列首元素为当前窗口最大值下标）
            while(s.size() && num[s.back()]<=num[i])
                s.pop_back();
            //移出不在滑动窗口内的队首元素坐标，弹出
            if(s.size() && i-s.front()+1>size)
                s.pop_front();
            s.push_back(i);//把每次滑动的num下标加入队列
            //当滑动窗口首地址i大于等于size时才开始写入窗口最大值（直到滑动窗口满了才开始统计）
            if(i+1 >= size) //这里注意一个细节，用的是无符号整形，切记保证不要和负数进行比较
                res.push_back(num[s.front()]);
        }
        return res;
    }
};
```

快速排序

- **思想**
  快速排序使用分治的思想，通过一趟排序将待排序列分割成两部分，其中一部分记录的关键字均比另一部分记录的关键字小。之后分别对这两部分记录继续进行排序，以达到整个序列有序的目的。
- **思路**
  1) 选择基准：在待排序列中，按照某种方式挑出一个元素，作为 “基准”（ pivot ）；
  2) 分割操作：以该基准在序列中的实际位置，把序列分成两个子序列。此时，在基准左边的元素都比该基准小，在基准右边的元素都比基准大；
  3) 递归地对两个序列进行快速排序，直到序列为空或者只有一个元素；

其中选择基准的方法如下：1、固定基准元 2、随机基准元 3、三数取中

- 代码实现

```
public static void QsortMedianOfThree(int[] arr, int low, int high) {
            if (low >= high) return; //递归出口
            PartitionMedianOfThree(arr, low, high); //三数取中
```

```
            int partition = Partition(arr, low, high); //将 >= x 的元素交换到右边区域，将 <= x 的元素交换到左边区域
            QsortMedianOfThree(arr, low, partition - 1);
            QsortMedianOfThree(arr, partition + 1, high);
        }

/// <summary>
        /// 三数取中确定基准元，将确定好的基准元与第一个数交换，无返回值
        /// </summary>
        public static void PartitionMedianOfThree(int[] arr, int low, int high)  {
            int mid = low + (high + -low) / 2;
            if (arr[mid] > arr[high])
                Swap(arr, mid, high);
            if (arr[low] > arr[high])
                Swap(arr, low, high);
            if (arr[mid] > arr[low])
                Swap(arr, mid, low); //将中间大小的数与第一个数交换
        }
```