

1. 设计一栈结构，使得出栈、入栈以及求栈的最小值均能在 $O(1)$ 时间内完成。

问题分析：

正常情况下入栈与出栈的时间复杂度都是 $O(1)$ ，但是求得栈的最小值则需要遍历整个栈，时间复杂度为 $O(n)$ 。那么我们该如何使求最小值的时间复杂度变为 $O(1)$ 呢。

因为出栈和入栈都是直接对最后一个元素进行操作，我们能否让最小值始终放在栈的最后呢？我们现在就是由此入手解决此问题。

算法思路

一个直观的想法可以是，设计两个栈，其中一个存放所有元素，另一个存放每次进入新元素后，所有元素中最小的元素。

图示：

4
2
1
7
6
5
8

Stack1

1
1
1
5
5
5
8

Stack2

在这种情况下，栈2的顶部就始终是最小值。

根据这样的想法，我们可以写出对应的算法。

```

void Push(x){
    Stack1.push(x);
    if(Stack2.empty())
        Stack2.push(x);    //如果2为空，则不用比较，直接入栈
    else
        Stack2.push((x<Stack2.top())?x:(Stack2.top())) //如果不为空，则压入较小者
}
void Pop(x){
    Stack1.pop(x);
    Stack2.pop(x);
}
int getmin(){
    cout<<Stack2.top()<<endl; //此时2中的顶部就是最小值
    return Stack2.top();
}

```

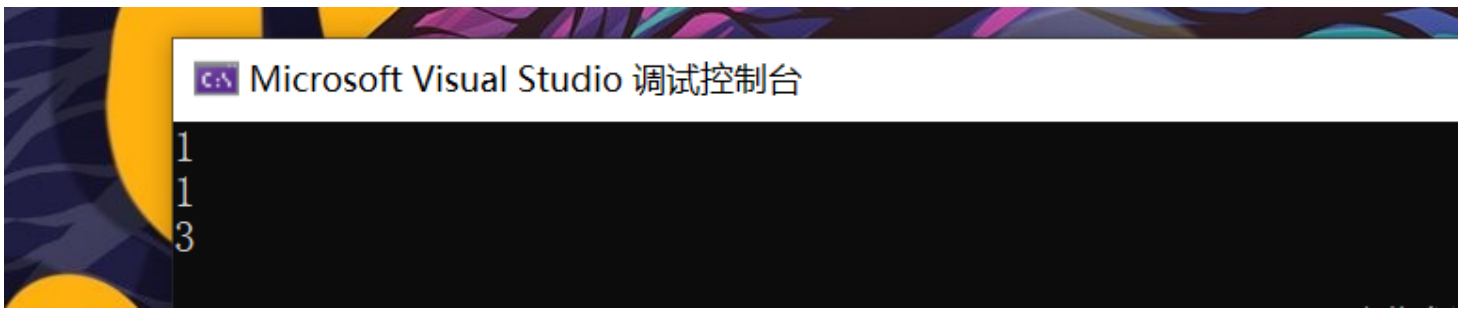
算法代码与分析

```
#include<iostream>
using namespace std;
#include<stack>
template<class T>
class Stack {
public:
    void Push(const T& x) {                //入栈
        Stack1.push(x);
        if (Stack2.empty())
            Stack2.push(x);
        else {
            Stack2.push((_minstack.top() > x) ? x : Stack2.top());
        }
    }
    void Pop() {                            //出栈
        Stack1.pop();
        Stack2.pop();
    }
    T GetMin() {                          //栈的最小值
        cout <<Stack2.top()<< endl;
        return Stack2.top();
    }
private:
    stack<T> Stack1;
    stack<T> Stack2;
};

void test() {
    Stack<int> s;
    s.Push(5);
    s.Push(3);
    s.Push(1);
    s.Push(7);
    s.Push(8);
    s.GetMin();
    s.Pop();
    s.GetMin();
    s.Pop();
    s.Pop();
    s.GetMin();
}

int main() {
    test();
    return 0;
}
```

运算结果:



分析：在这个结构体内，包含了两个小的栈，但是它们并不是分离的，而是一个整体。分别负责记录所有的元素和最小元素从而让出栈入栈以及求栈的最小值都达到 $O(1)$ 。

2.改进冒泡排序使得在最好情况下可以在线性时间内完成。

问题分析：

在普通的冒泡排序中，相邻的两个数 $arr[i]$ 与 $arr[i+1]$ 比较，如果前者大于后者，则交换两数的数值。据此类比，在进行完一轮的排序后，这个数列的最后一位一定是最大的数字。像这样的排序 $n-1$ 次，即可实现整个数列从小到大的排序。但是用这种方法，不管起始数列如何，其时间复杂度一定是 $O(n^2)$ 。在前面这句话中，重点是“不管起始数列如何，一定是”，所以我们想要改良此算法，可以从起始数列入手。

算法思路（1）

题目中所给的“最好情况下”想必就是起始数列就是一个排列好的数列，无需我们进行交换。因此，我们可以设置一个标志位，如果在一轮循环中，没有进行数值的交换，则break循环结束。

那么此时我们的大概思路就是：

```

for(int i=0;i<n-1;i++){
    bool flag = 1;  //令标志位为1
    for(int j =0;j<n-1-i;j++){
        if(arr[j]>arr[j+1]){
            temp = arr[j];
            arr[j] = arr[j+1];    //元素的交换
            arr[j+1] = temp;
            flag = 0;
        }
        if(flag)
            break;    //如果没有发生交换，则说明当前序列已经排列完毕
    }
}

```

算法思路（2）

我们都知道，理论上每轮交换完数值后，最后一次交换位置往后都是标准顺序了，因此我们可以记录下每一次最后交换的位置，然后下一次遍历交换时，仅到上一次的标志位即可。

过程如下：

```

while (m > 0)
{
    for (k = j = 0; j < m; j++)
    {
        if (array[j] > array[j + 1])
        {
            temp = array[j];
            array[j] = array[j+1];
            array[j+1] = temp;
            k = j;    // 记录每次交换的位置
        }
    }
    m = k;          // 记录最后一个交换的位置
}

```

算法代码及分析：


(1)：

```

1      #include<iostream>
2      using namespace std;
3      int main() {
4          int arr[5] = {1, 2, 3, 4, 5};
5          int irr = 0;
6          for (int i = 0; i < 4; i++) {
7              int flag = 1;
8              for (int j = 0; j < 4 - i; j++) {
9                  if (arr[j] > arr[j + 1]) {
10                     irr = arr[j];
11                     arr[j] = arr[j + 1];
12                     arr[j + 1] = irr;
13                     flag = 0;
14                 }
15             }
16             if (flag)
17                 break;
18         }
19         for (int i = 0; i <= 4; i++) {
20             cout << arr[i] << endl;
21         }
22     }
23 }

```

结果展示:

 选择 Microsoft Visual Studio 调试控制台

```

1
2
3
4
5

```

C:\Users\lenovo\Desktop\Project2\Debug\Project2.exe
要在调试停止时自动关闭控制台，请启用“工具”菜单中的“调试”->“在调试停止时关闭控制台”
按任意键关闭此窗口

分析:


这种算法只考虑了没有交换这一种情况，也即只有遇到“完美序列”才可以起到简化时间的作用达到 $O(n)$ ，一旦遇到一般情况下的序列，其时间复杂度还是 $O(n^2)$ 。

(2)：

```
#include<iostream>
using namespace std;
int main() {
    int temp = 0;
    int array[5] = { 1,2,3,4,5 };
    int m = 4;
    int k, j;
    while (m > 0)
    {
        for (k = j = 0; j < m; j++)
        {
            if (array[j] > array[j + 1])
            {
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
                k = j; // 记录每次交换的位置
            }
        }
        m = k; // 记录最后一个交换的位置
    }

    for (int i = 0; i <= 4; i++) {
        cout << array[i] << endl;
    }
}
```

结果展示：

 Microsoft Visual Studio 调试控制台

```
1
2
3
4
5
```

分析：我个人认为这种方法是要比上一种方法好，因为这种方法对于任何序列都可以起到简化作用，即省去不必要的交换验证。并且遇到最好完美序列需要 $O(n)$ ，最差情况为 $O(n^2)$ 。

以上两种优化方式的比较：

第一种只对“完美序列”起到了优化作用，作用范围比较局限。

第二种方法则对除“最不完善序列”外的所有序列都可以达到优化作用，并且最好可以达到 $O(n)$ ，作用范围广泛且明显。