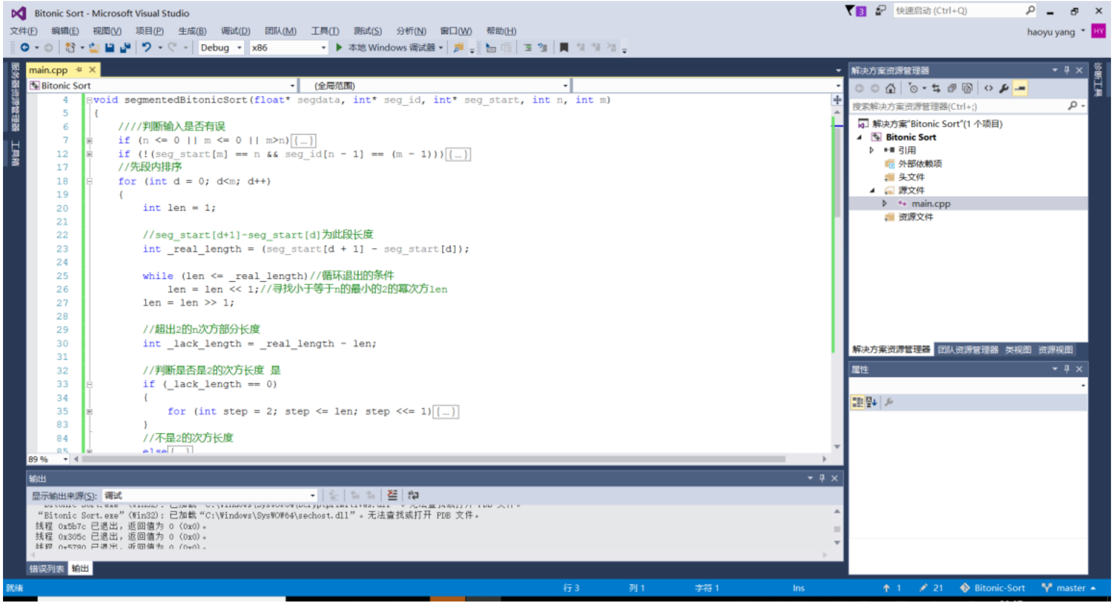
分段双调排序实现

最终代码截图：



最终结果：所有挑战均完成

a. 算法描述

双调排序的基本概念有两个：双调序列和 Batcher 定理。

参考资料（https://blog.csdn.net/jiange_zh/article/details/49533477）

双调序列：双调序列是由两个单调性相反的非严格单调序列构成的序列（**非严格指的是可以出现重复元素，或者NaN不参与排序**）。比如（23，10，8，3，5，7，7，8）。当序列满足以下两种情况时，它是双调序列：

（1） 存在一个ak$(1 \le k \le n)$，使得$a1 \ge .. \ge ak \le .. \le an$成立。

（2） 序列能偶循环位移满足条件（1）。

Batcher 定理：将任意一个长为 2n 的双调序列 A 分为等长的两半 X 和 Y，将 X 中的元素与 Y 中的元素一一按原序比较，即$a[i]$与$a[i+n]$ $(i < n)$比较，将较大者放入 MAX 序列，较小者放入 MIN 序列。则得到的 MAX 和 MIN 序列仍然是双调序列，并且 MAX 序列中的任意一个元素不小于 MIN 序列中的任意一个元素。

读过（https://blog.csdn.net/u014226072/article/details/56840243）

（https://blog.csdn.net/hanshuning/article/details/49132089）两位笔者的代码和思路后，总结算法基本过程主要有两步，首先将输入的无序的序列转化成双调序列，然后将得到的双调序列进行双调合并即可得到最终解。

具体思路：对于任意两个元素x，y，无论x ≥ y 或 x ≤ y，序列（x，y）均为双调序列。因此任何无序的序列都是由若干个二元有序的双调序列连接而成的。于是，对于一个无序序列我们按照递增和递减顺序合并相邻的双调序列，按照双调序列的定义，通过连接递增和递减序列得到的序列是双调的。最终，我们可以将若干个只有 2 个元素的双调序列合并成 1 个有 n 个元素的双调序列。

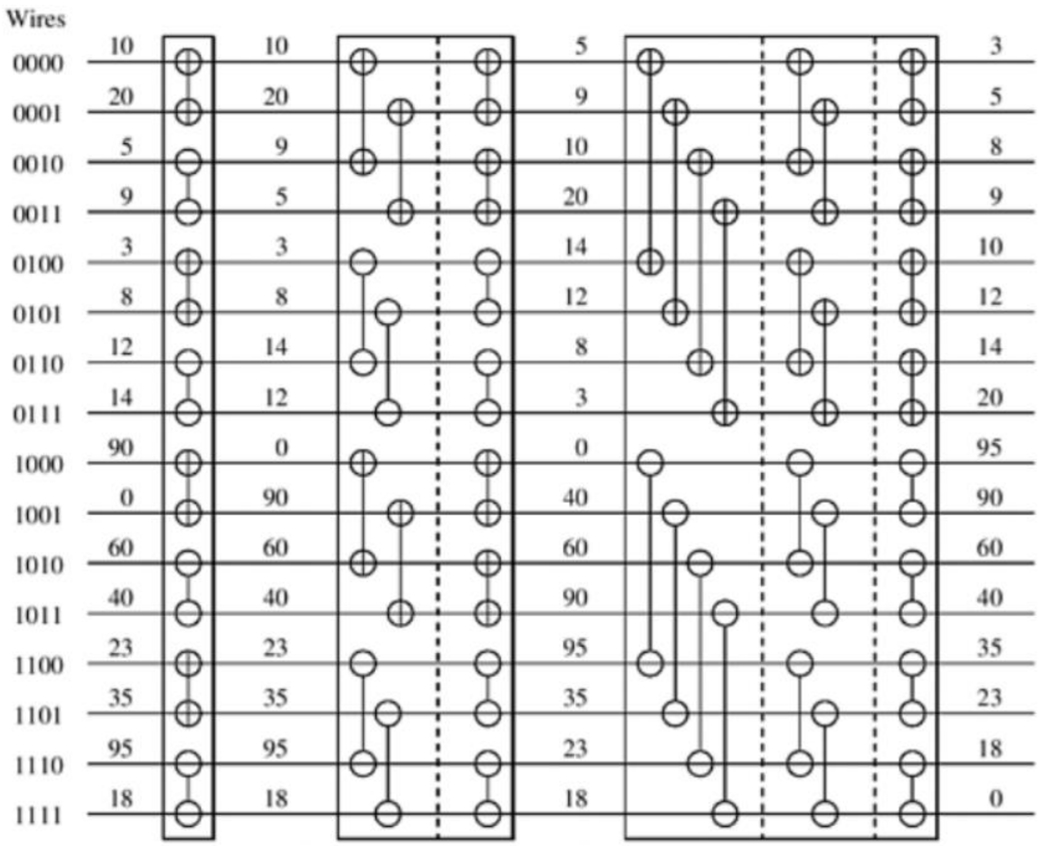


Figure 1 将无序的输入序列转换成双调序列

因为思路基本相同，我在（https://blog.csdn.net/hanshuning/article/details/49132089）笔者分享的代码上略加修改形成了第一版代码（主要函数）：

```
void segmentedBitonicSort(float* data, int* seg_id, int* seg_start, int n, int m)
```

```
{
    ////判断输入是否有误
    if (n <= 0 || m <= 0 || m>n)
    {
        cout << "Input error!n>m>0" << endl;
        return;
    }
    if (!(seg_start[m] == n && seg_id[n - 1] == (m - 1)))
    {
        cout << "Input error! seg_start[m]==n,seg_id[n-1]==(m-1)" << endl;
        return;
    }
    //先段内排序
    for (int d = 0; d<m; d++)
    {
        int len = 1;
        //seg_start[d+1]-seg_start[d]为此段长度
        while (len < seg_start[d + 1] - seg_start[d])//循环退出的条件
            len = len << 1;//寻找大于n的最小的2的幂次方len

        float Max = 999999;//作为填充数
        //vector<float> segdata(len);

        int _properly_divided = 1;//判断是否整出的参量
        int _real_length = (seg_start[d + 1] - seg_start[d]);

        vector<float> segdata(len);

        for (int i = 0; i < seg_start[d + 1] - seg_start[d]; i++)
        {
```

```
        segdata[i] = data[seg_start[d] + i];
    }
    //如果len > n,就说明数组的个数不够，要将个数填充到len个
    for (int i = seg_start[d + 1] - seg_start[d]; i < len; i++)
        segdata[i] = Max;



    ///////对整个数组进行排序
    for (int step = 2; step <= len; step <<= 1)
    {
        //////内部循环可任意交换
        for (int i = 0; i < len; i += step << 1)//1
        {
            ///前半部分升序排
            for (int step0 = step >> 1; step0 >0; step0 >>= 1)// 2
            {
                for (int j = 0; j < step; j += step0 << 1)//3
                {
                    for (int k = 0; k < step0; ++k)//4
                    {
                        if (i + j + k + step0 > _real_length)
                        {
                            continue;
                        }
                        if (segdata[i + j + k] > segdata[i + j + k + step0] ||
segdata[i + j + k] != segdata[i + j + k])  //交换数据使升序排列,同时判断二者之中是否有
NaN
                        {
                            //交换data
                            float T = segdata[i + j + k];
```

```
                        segdata[i + j + k] = segdata[i + j + k + step0];
                        segdata[i + j + k + step0] = T;
                    }
                }
            }
        }
        //后半部分降序排
        if (i + step < len)
        {  //////内部循环可任意交换
            for (int step0 = step >> 1; step0 >0; step0 >>= 1) //1
            {
                for (int j = 0; j < step; j += step0 << 1) //2
                {
                    for (int k = 0; k < step0; ++k) //3
                    {
                        if (i + step + j + step0 + k > _real_length)
                        {
                            continue;
                        }

                        if (segdata[i + step + j + k] < segdata[i + step + j
+ step0 + k]
                            || segdata[i + step + j + step0 + k] !=
segdata[i + step + j + step0 + k]) //交换数据使降序排列,同时判断二者之中是否有NaN
                        {
                            //交换data
                            float T = segdata[i + step + j + k];
                            segdata[i + step + j + k] = segdata[i + step +
j + k + step0];
                            segdata[i + step + j + k + step0] = T;
```

```
                        }
                    }
                }
            }
        }
    }
}
//赋值
for (int i = seg_start[d]; i < seg_start[d + 1]; i++)
{
    data[i] = segdata[i - seg_start[d]];
    if (data[i] == Max)
        data[i] = sqrt(-1.f);
}
}
}
```

经测试，代码运行正确。然而这段代码在解决序列长度非 2 的 n（n 自然数）次幂时，利用了补无穷的思想。实际需要的数组长度大于输入的数组长度，因此借助了 C++ STL 工具类 <vector> 动态的分配了数组大小，**不能**满足**内存高效**的挑战要求。

```
vector<float> segdata(len);

for (int i = 0; i < seg_start[d + 1] - seg_start[d]; i++)
{
    segdata[i] = data[seg_start[d] + i];
}
//如果len > n,就说明数组的个数不够，要将个数填充到len个
for (int i = seg_start[d + 1] - seg_start[d]; i < len; i++)
    segdata[i] = Max;
```

Figure 2 补无穷方法处理一般序列

经过很长时间的思考，我发现无法在这种方法的基础上避免动态分配内存。在网上查阅资料后，在简书中发现了一篇文章（https://www.jianshu.com/p/ea4a62fdaae9）。

任意双调排序主要思想：

1. 首先不断的二分，直到每组只剩下一个元素，然后开始归并。
2. 双调排序归并时**以不大于 n 的最大 2 的幂次方 2^k 为界限**，把 2^k..n 的元素分别与 0..(n-2^k) 的元素比较，然后再分别在0..2^k 和 2^k..n 这两段上分别应用比较网络。
3. 双调排序难以理解的地方就在于这个归并的过程，假设我们要把长度为 n 的序列 arr 升序排列，**由于二分时是把前 n/2 的序列降序排列后 n/2 的序列升序排列的，而 n-2^k < n/2，所以前 n-2^k 和后 n-2^k 个元素都大于中间的元素，当前 n-2^k 个元素和后 n-2^k 个元素比较时，会把序列中最大的 n-2^k 个元素放到最后 n-2^k 个位置上，也就是说比较后，2^k..n 的元素都比 0..2^k 的元素大（这一点是确定的，因为前半段递减，后半段递增，比较的时候相当于“极大值”与“极小值”的比较，有点田忌赛马的味道）**，这样在分别对这两段用同样的方法归并，最终得到完整的升序序列。

双调排序分治的核心理论：**任意的正整数都能表示成 2 的幂指数和的形式，其实就是所有的正整数都能用二进制表示。**因为序列在不断地拆分成 2 的幂指数的长度。

Figure 3 任意双调排序主要思想

简单的概括就是利用多次分段的排序和归并以达到整体排序。因此，我推倒第一版代码重新开始编写。将输入分为标准（指序列长度为 2 的 n 次幂）和非标准两种情况。标准情况直接求解，非标准情况以**小于序列长度的最大的 2 的幂次方**（记作len）作为参考将输入序列分为前后两部分，通过三次排序运算求解。（源码见附件）

b. 尝试过和完成了的加分挑战

所有挑战均尝试并完成

1. 不递归：segmentedBitonicSort函数中不进行任何直接或间接递归。
2. 不调用函数：segmentedBitonicSort函数中不调用任何除标准库函数以外的任何其他函数。
3. 内存高效：算法中所有的数值交换均在初始数据数组中完成，没有任何动态分配以及使用 STL 类。
4. 可并行：排序算法中所有 for 循环头嵌套时没有任何上下依赖关系，因此 for 循环内部的循环顺序可以随意改变。
5. 不需内存：segmentedBitonicSort函数没有调用任何函数，不使用全局变量，所有局部变量均是 int，float 或指针型，没有使用 new 关键字。

6. 绝对鲁棒：输入数据包含NaN时不影响其他数据排序，且NaN个数不变。通过判断 NaN！ = NaN 实现。

```
|| segdata[i + j + k + seg_start[d]] != segdata[i + j + k + seg_start[d]]
```

Figure 4 **鲁棒性处理**

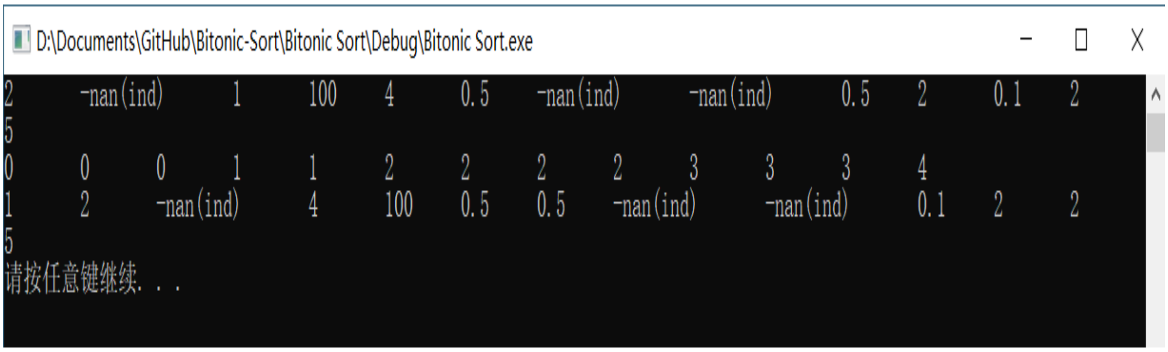
c. 可以独立运行的源代码

源代码和执行程序见附件

PS :Linux 或 mac 系统没有System（“pause”）函数且需要导入cmath库以支持 sqrt 函数。

d. 测试数据

```
float data[13] = { 2,sqrt(-1.f) - 100 , 1, 100, 4, 0.5,sqrt(-1.f), sqrt(-1.f), 0.5, 2, 0.1, 2, 5 };
int seg_id[13] = { 0, 0, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4};
int seg_start[6] = { 0,3,5,9,12,13};
int n = 13;
int m = 5;
```



```
D:\Documents\GitHub\Bitonic-Sort\Bitonic Sort\Debug\Bitonic Sort.exe
2       -nan(ind)       1       100     4       0.5     -nan(ind)       -nan(ind)       0.5     2       0.1     2
5
0       0       0       1       1       2       2       2       2       3       3       3       4
1       2       -nan(ind)       4       100     0.5     0.5     -nan(ind)       -nan(ind)       0.1     2       2
5
请按任意键继续. . .
```

Figure 5 测试数据结果 1

```
float data[11] = { 0,sqrt(-1.f) - 100 , 2, 100, 4, 0.5,sqrt(-1.f), sqrt(-1.f), 3, 0.1, 2 };
int seg_id[11] = { 0, 0, 0, 1, 1, 2, 2, 2, 2,3,3 };
int seg_start[5] = { 0,3,5,9,11 };
int n = 11;
```

```
int m = 4;
```

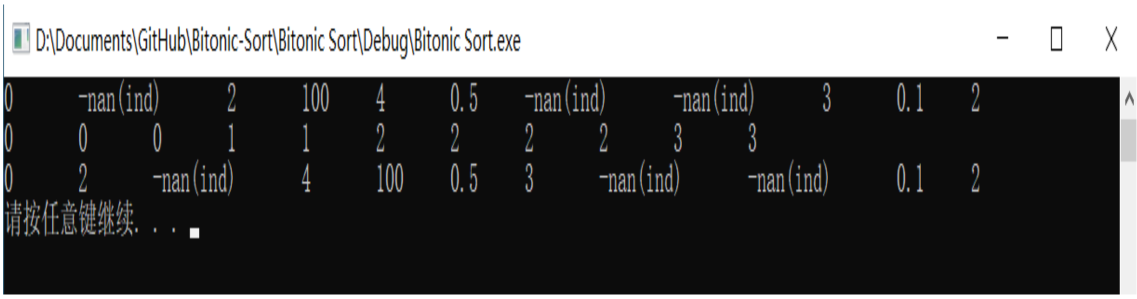


Figure 6 测试数据结果 2

```
float data[10] = { 0,sqrt(-1.f) - 100 , 2, 100, 4, 0.5, sqrt(-1.f), 3, 0.1, 2 };
int seg_id[10] = { 0, 0, 0, 1, 1, 1, 2, 2, 2, 2 };
int seg_start[4] = { 0,3,6,10 };
int n = 10;
int m = 3;
```

Figure 7 测试数据结果 3

e. 性能分析

时间复杂度：如果以串行方式运行，其复杂度为$O(n\log^2 n)$相对的，如果有n个可同时运行的线程，复杂度为$O(\log^2 n)$。

本程序中未使用递归，所以更节省时间和空间。内存高效，没有调用任何函数（包括C/C++标准库函数），没有使用全局变量，没有进行动态内存分配。可并行，segmentedBitonicSort涉及到的所有时间复杂度 O(n)以上的代码都写在 for 循 环中，而且每个这样的 for 循环内部的循环顺序可以任意改变，不影响程序结果。可以处理任意长度的输入序列。绝对鲁棒，在输入数据中包含NaN时（例如 sqrt(-1.f)），把NaN当作比任意数大的数进行排序，保证除NaN以外 的数据正确排序，NaN的个数保持不变。

不足：

（1） 算法复杂度较高，执行效率低。

（2） 时间仓促，代码有可精简之处。

（3） 初始化变量写死在文件中，用户调试体验较差。

f. 测试的起始和完成时间以及实际的使用时间

开始测试时间：2018 年 5 月 22 日 16：00 左右

完成时间：2018 年 5 月 23 日 1：19

邮件时间：2018 年 5 月 23 日 8：00

实际使用时间：7 到 8 小时之间

接到题目后，我便开始查找资料。虽然是第一次接触双调排序，但是好在算法比较经典，网上资料很丰富，大约 1h 得到了第一版的代码。之后研究如何处理任意数列排序使其避免动态分配内存。花了 2h 找不到解决办法，吃饭洗澡回来后，又和哥哥讨论了约 0.5h 依旧无果。决定改变思路，花了约 1h 查找资料并完成了新的方法。最后花了 3-4h 时间整理文档。