1 程序设计思路

对于一个最大堆（最大的元素在根部的）而言，可以通过依次弹射最大元的方法，实现排序，因此总体思路为，每一次都将最大元弹射出去排序，然后将剩下的数据重新调整成堆即可。

下面讨论细节：弹射操作非常容易，因为对于一个最大堆而言，其根节点就是最大元，至于如何将剩下的数据重新变成堆需要着重处理。可以先将数组末尾的元素安置到堆的根部去，但是这并不是堆（因为不符合堆序性），所以通过“下滤”操作，如果这个元素比其孩子还小，那么就与孩子进行交换，就好像“下沉”一样，如此不断与孩子比较，直到下不去为止。

流程：

1. 弹出最大元，最末位补充，即两者进行 swap。（因为最大的元素，进行递增排序后恰会排在末尾）
2. 调用 Downward_sift 函数把剩余的数据变成一个正确的堆，由于排除根节点外的元素，都已符合堆序性，因此仅需跟踪该根节点的下沉过程。
3. 再次重复前两步操作。

2 测试思路

其中 check 函数只需要对比使用标准库的方法和使用我自己的方法，生成的两个 vector 是否一致即可，可传入引用以避免复制。

1. 长度 1500000 的随机序列
2. 长度 1500000 的增序序列
3. 长度 1500000 的降序序列
4. 长度 1500000 的部分元素重复序列（只需要限制随机数产生的范围，自动就能得到含重复的元素的随机序列）

在测试时使用 chrono 进行计时，执行前后的时间作差，使用 print_result 函数打印出时间，包括 check 的结果即可。

注意，这里的计时统一用**建堆后开始到排序结束**的时间。

3 测试效果

可直接使用 make run 命令编译且运行。

输出形如：

```
 **Random Input**
 sort_heap: 0.157933
 my_sort: 0.15796
 check result: 1
**Increasing Input**
 sort_heap: 0.063219
 my_sort: 0.0688413
 check result: 1
**Decreasing Input**
 sort_heap: 0.0810255
```

```
 my_sort: 0.0808056
 check result: 1
**Random with Repeat Input**
 sort_heap: 0.152754
 my_sort: 0.157996
 check result: 1
```

在未加入-O2 优化时，如下（单位：second）：

表 1: 运行时间对比

输入序列	**my_sort time**	**std::sort_heap time**
随机	0.733434	1.02845
递增	0.53948	0.851013
递减	0.542546	0.865929
随机（含重复）	0.716736	1.0294

在编译时加入-O2 进行优化后，如下（单位：second）：

表 2: 运行时间对比 (-O2)

输入序列	**my_sort time**	**std::sort_heap time**
随机	0.1622	0.161694
递增	0.076275	0.0677776
递减	0.0800664	0.0843663
随机（含重复）	0.160439	0.156313

用 valgrind –leak-check=full ./test 进行测试，发现没有发生内存泄露

4 时间复杂度分析

由于每一次弹出最大元是 $\Theta(1)$ 的，而耗时的主要是“向下过滤”的过程，由于最糟糕的情况是向下过滤到底，即树高 h 那么多。而共操作 N 次，因此为 $\Theta(N \log N)$

实际上这和标准库的方法一样，因此时间复杂度一致。但，由于我写的 my_sort 函数要求输入必须**已经是个堆了，不会进行检查**，而 std::sort_heap 函数则会首先进行检查，如果不是堆会先建堆。这导致在“检查阶段”耗时，所以可见在表格 1无 O2 优化时，我的函数在各种情况下都更快 0.3 秒左右，有理由猜测是检查的耗时。