CISC-365

2009

Lab # 3

Week of September 28

Many algorithms depend on data being in a particular order (either ascending or descending). The goal of this lab is simple: to implement a particular sorting algorithm in the language of your choice.

You probably already know that any sorting algorithm that compares elements of the set to each other (what we call "comparison-based sorting") has an OMEGA(n * log n) lower bound on its running time, and of course there are well-known algorithms that run in O(n * log n) time. Nonetheless, if you ask someone to quickly implement a sorting algorithm, the odds are very high that they will implement bubble-sort or selection-sort or another O(n^2) algorithm. This may be due to the undeniable fact that these inefficient algorithms are easy to understand, easy to remember, and easy to code.

However, as sophisticated algorithmists we aren't going to settle for second best. There are three classic O(n * log n) sorting algorithms: QuickSort, MergeSort, and HeapSort. QuickSort can be implemented very efficiently, but is prone to subtle errors in coding that can lead to incorrect behaviour in some cases. MergeSort is relatively easy to code but typically runs quite slowly because it spends a lot of time copying data back and forth between arrays.

That leaves HeapSort, which is often overlooked because it involves a binary tree, and people in the "real world" seem to be afraid of implementing tree data structures. Fortunately we can use an array or other sequential linear structure to implement the tree.

Definition: A **min-heap** is a binary tree of values in which each value is less than or equal to its children (if any). For example

```
        3
       / \
      7    4
     / \     \
   10   34    5
```

is a min-heap,

but

```
        3
      /   \
    7       4
   / \
 10    2
```

is not, because the 7 is greater than one of its children.

Note that a min-heap always has its smallest value at the top of the tree. This property is crucial to the use of a min-heap for sorting.

First let us consider the problem of heapifying a set of values. It's actually very easy. We start with the first value in the set, as the root of the tree. For each remaining value, we add it as a new leaf at the bottom of the tree, then swap it with its parent if it is smaller than its parent. We continue to push the new value up the tree until it reaches a point where it is not smaller than its parent.

For example, suppose the 2 has just been added to the second tree shown above. It is smaller than its parent (7), so the two values would switch places. The 2 is smaller than its new parent (3), so the 2 would move up again by swapping with the 3. It has reached the top of the tree so it cannot go any higher. The tree now looks like

```
        2
      /   \
    3       4
   / \
 10   7
```

and we see that it is a well-formed min-heap again.

Once we have the entire set stored in a min-heap, we can extract all the values in ascending order by repeatedly applying the following steps:

1. Output the value at the top of the heap
2. Replace that value with the ***smaller*** of its children (or with a null value if there are no children).
3. Whichever child was just "promoted", replace that value by the smaller of ***its*** children, and so on.

For example, given the min-heap shown above, we start by outputting 2. Then we replace it by moving up the smaller of its children (3), and then we move up the smaller of 3's children (7). The min-heap now looks like

```
         3
       /   \
     7       4
   /
 10
```

In the next iteration, the 3 is output and the min-heap ends up as

```
         4
       /
     7
   /
 10
```

and so on.

Now for the trick: storing the heap in an array (in fact, we can use the array that initially contains the set, if we choose). In the array we can treat position 0 as the root of the tree, and then for each position i, we can treat positions 2*i + 1 and 2*(i+1) as position i's children. Similarly, for each position j (j > 0), we can treat position Floor((j-1)/2) as position j's parent.

For example, position 0's children are positions 1 and 2. Position 1's children are positions 3 and 4, and position 2's children are positions 5 and 6. Position 9's parent is position 4, etc.

With this crafty ploy, we have no need for fancy-pants dynamic structures and we run no risk of accidentally lost links or infinite loops of pointers.

Thus creating the min-heap looks “something like this”:

```
for (int x = 1; x < n; x++)
    temp = x
    while (temp > 0) && (A[temp] < A[temp's parent])
        swap A[temp] with A[temp's parent]
        temp = temp's parent
```

So there you have it. Put the pieces together and implement HeapSort. On the course webpage you will find a data file for this lab containing several lines. The first line contains a single integer, indicating the number of sets to be sorted. Subsequent lines occur in pairs. The first line in each pair contains a single integer, indicating the number of integers on the following line. The second line in each pair contains a randomly ordered set of **positive** integers. Your program should output each of the sets in non-decreasing order.