David Hatcher

U74842093

Homework 4

1.

1. The most efficient data structure to use for this is a Min-Max heap as it will allow you to get the Maximum value and Minimum value in the heap currently in O(lgn), an array will also be useful for the return value as it will be easy to access the value nth lowest, and nth highest values from the sorted data.
2. Input: data, an unordered array  
   Input: n, the number of values in data  
   Output: dataOIS, a permutation of the input data array such that data[1] is the min of data, data[2] is the max of data, data[3] is the second smallest of data, data[4] is the second largest of data, so on and so forth.  
   Algorithm: Outside-In Sorting  
   data = MinMaxHeap.build(data)  
   dataOIS = Array(n)  
   for i = 1 to n step 1 do  
    if i % 2 = 1 then  
    dataOIS[i] = data.DeleteMin()  
    else  
    dataOIS[i] = data.DeleteMax()  
    end  
   end  
   return dataOIS
3. Line 1 takes a total of O(n), If we assume line 2 takes O(1) to build an empty array, and each iteration of the loop will take O(lgn) and the array will always run for a total of n iterations. Because of the invariability of this algorithm the worst-case, and best case, will always be O(nlgn).
4. Yes this Outside-In Sorting does exhibit the optimal substructure property. You can combine two of these arrays together by alternating what which value you’re looking at. Start with 3 pointers, one points to the first value in the first OIS array, one points to the first value in the second OIS array, and one points to the first value in a third, empty, array of n+m length. Then compare the first value of each array with each other, which ever one is smaller add to the third return array and increment the point to the next odd index for the array that had the smaller value. Continue this until both pointers are at the final index. Once you’re there you can do the same thing but backwards for the even indices.

2.

1. Input: n: size, of the Union-Find to initialize   
   Output: a Union-Find of size n where every element points to itself  
   Algorithm: Union-Find.Initialize  
   uf = Array(n)  
   Initialize uf to 1..n  
   min = array(n)  
   Initialize min to 1..n  
   size = Array(n)  
   Initialize size to 1  
   return (uf,size,min)  
     
   By adding in the min array as an n length array and initializing all of the values to 1..n then you currently have an array min that can be used to access the smallest value in each set currently since each set is just a single value.
2. Because each initialization of the arrays are just O(n) and the creation of each array is O(1) then we would end up with O(3n) = O(n)
3. Input: (uf,size,min): the Union-Find to modify  
   Input: a: index of one element to union  
   Input: b: index of another element to union  
   Output: modified uf that efficiently merges the trees containing a and b  
   Algorithm: uf.Union  
   ra = uf.Find(a)  
   rb = uf.Find(b)  
   if size[ra] > size[rb] then  
    Swap ra and rb  
   end  
   if min[ra] < min[rb] then  
    min[rb] = min[ra]  
   end  
   uf[ra] = rb  
   size[rb] = size[ra] + size[rb]
4. Both of the first two lines using the Find algorithm have a O(inverse ackermann(n))  
   The swap is O(1)  
   and the line with min[rb] = min[ra] is O(1)  
   both of the last two lines have O(1) as they are just value assignments and not functions or loops  
   Thus the entire algorithm has a complexity of:  
   f(n) = O(inverse Ackermann(n)) + O(inverse Ackermann(n)) + O(1) + O(1) + O(1) + O(1)  
   f(n) = O(inverse Ackermann(n))