Assignment 6 - key

1. What is the running time of insertion sort if all elements are equal?

O(*N*), because the *while* loop terminates immediately. Of course, accidentally changing the test to include equalities raises the running time to quadratic for this type of input.

2. Show how heap sort processes the input 142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102

The input is read in as

142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102

The result of the heapify is

879, 811, 572, 434, 543, 123, 142, 65, 111, 242, 453, 102

879 is removed from the heap and placed at the end. We'll place it in italics to signal that it is not part of the heap. 102 is placed in the hole and bubbled down, obtaining

811, 543, 572, 434, 453, 123, 142, 65, 111, 242, 102, *879*

Continuing the process, we obtain

572, 543, 142, 434, 453, 123, 102, 65, 111, 242, *811*, *879*

543, 453, 142, 434, 242, 123, 102, 65, 111, *572*, *811*, *879*

453, 434, 142, 111, 242, 123, 102, 65, *543*, *572*, *811*, *879*

434, 242, 142, 111, 65, 123, 102, *453*, *543*, *572*, *811*, *879*

242, 111, 142, 102, 65, 123, *434*, *453*, *543*, *572*, *811*, *879*

142, 111, 123, 102, 65, *242*, *434*, *453*, *543*, *572*, *811*, *879*

123, 111, 65, 102, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*

111, 102, 65, *123*, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*

102, 65, *111*, *123*, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*

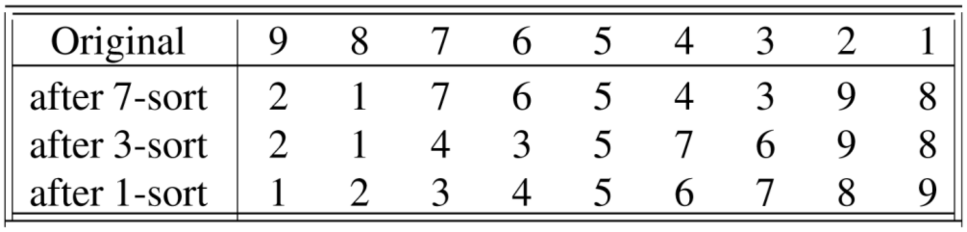
65, *102*, *111*, *123*, *142*, *242*, *434*, *453*, *543*, *572*, *811*, *879*

3. Sort 3, 1, 4, 1, 5, 9, 2, 6 using merge sort

First the sequence {3, 1, 4, 1} is sorted. To do this, the sequence {3, 1} is sorted. This involves sorting {3} and {1}, which are base cases, and merging the result to obtain {1, 3}. The sequence {4, 1} is likewise sorted into {1, 4}. Then these two sequences are merged to obtain {1, 1, 3, 4}. The second half is sorted similarly, eventually obtaining {2, 5, 6, 9}. The merged result is then easily computed as {1, 1, 2, 3, 4, 5, 6, 9}.

4. Show the result of running Shellsort on the input 9, 8, 7, 6, 5, 4, 3, 2, 1 using the increments {1,3,7}.

(Show steps)



Original	9	8	7	6	5	4	3	2	1
after 7-sort	2	1	7	6	5	4	3	9	8
after 3-sort	2	1	4	3	5	7	6	9	8
after 1-sort	1	2	3	4	5	6	7	8	9

5. A sorting algorithm is stable if elements with equal keys are left in the same order as they occur in the input. Which of the sorting algorithms in Chapter 7 are stable and which are not? Explain.

Some Sorting Algorithms are stable by nature, such ***as Insertion Sort, Merge Sort, Count Sort****. Comparison based stable sorts such as Merge Sort and Insertion Sort, maintain stability by ensuring that-*

Element $A[j]$ *comes before* $A[i]$ *if and only if* $A[j] < A[i]$*, here i, j are indices and* $i < j$*. Since* $i < j$*, the relative order is preserved if* $A[i] = A[j]$*, i.e.* $A[i]$ *comes before* $A[j]$*. Other non-comparison based sorts such as Counting Sort maintain stability by ensuring that the Sorted Array is filled in a reverse order so that elements with equivalent keys have the same relative position. Some sorts such as Radix Sort depend on another sort, with the only requirement that the other sort should be stable.*

Quick Sort, Heap Sort are not stable *by nature but they can be made stable by also taking the position of the elements into consideration. This change may be done in a way which does not compromise a lot on the performance and takes some extra space, possibly* $\Theta\ (N)$*.*
Any comparison based sorting algorithm which is not stable by nature can be modified to be stable by changing the key comparison operation so that the comparison of two keys considers position as a factor for objects with equal keys.

6. Prove that any comparison-based algorithm to sort 4 elements requires 5 comparisons. Give the algorithm to sort 4 elements in 5 comparisons.

$\lceil \log 4! \rceil = 5$.

Let the elements in the arrays be represented as a_1*,* a_2*,* a_3 *and* a_4

Step 1: Compare and exchange (if necessary) a_1 *and* a_2 *so that* $a_1 \geq a_2$*.*

Step 2: Compare and exchange (if necessary) a_3 *and* a_4*.*

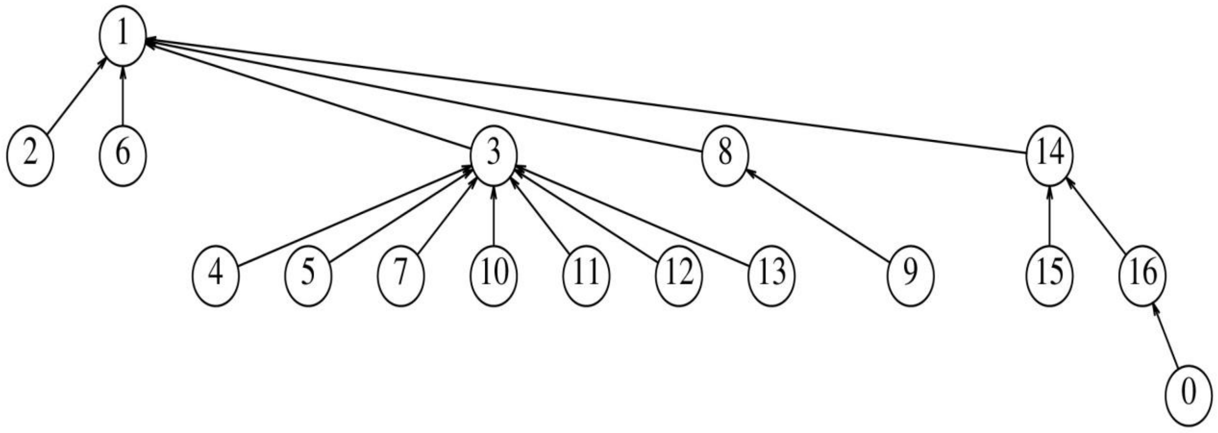
Step 3: Compare and exchange (if necessary) a_1 *and* a_3*.*

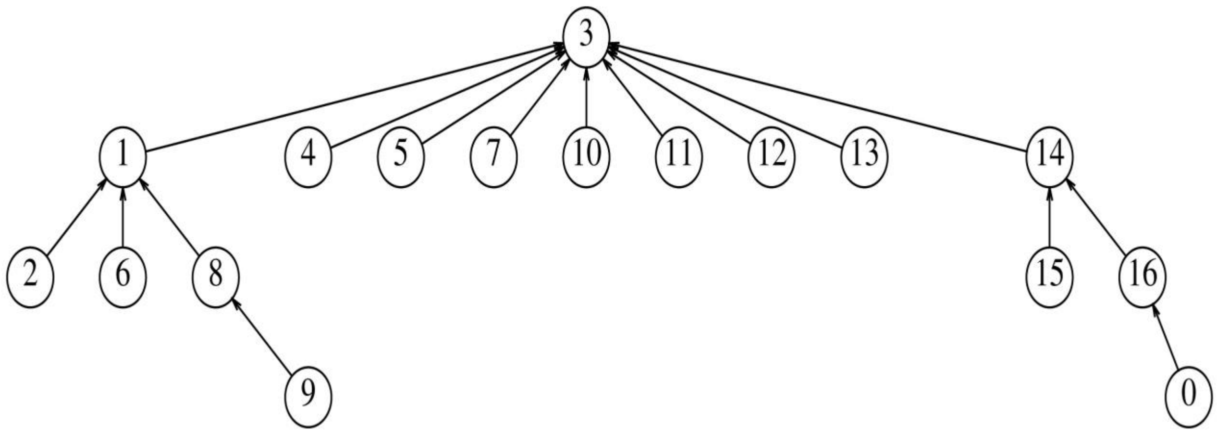
Step 4: Compare and exchange (if necessary) a_2 *and* a_4*.*

Step 5: Compare and exchange (if necessary) a_2 *and* a_3*.*

7. Show the result of the following sequence of instruction: union(1,2), union(3,4), union(3,5), union (1,6), union(3,7), union(8,9), union(1,8), union(3,10), union(3,11), union(3,12), union(3,13), union(14,15), union(16,0), union(14,16), union(1,3), union(1,14) where the union are
   a. Performed arbitrarily
   b. Performed by height
   c. Performed by size

We assume that unions operated on the roots of the trees containing the arguments. Also, in case of ties, the second tree is made a child of the first. Arbitrary union and union by height give the same answer (shown as the first tree) for this problem. Union by size gives the second tree.


1
2
6
3
8
14
4
5
7
10
11
12
13
9
15
16
0


3
1
4
5
7
10
11
12
13
14
2
6
8
15
16
9
0