* 1. The algorithm is intended to remove half the elements in a list. More specifically it will delete the smallest 25% and greatest 25%, leaving the middle 50% values in the list. L will keep the values from the 1st quartile to the 3rd quartile, and doesn’t specify anything of their ordering.
  2. Yes, assuming the intention of the algorithm in part a is as well. Since the removal code isn’t included we can safely assume it is correct. So the algorithm must be called n / 4 times to remove the n / 2 elements with 2 elements removed each time. AKA 4^(k - 1) recursive calls occur.  
       
     Proof by Induction:   
     We know the size of L is 4^k and 4^(k - 1) recursive calls are made, removing 2 elements each. When the algorithm completes there will be (4^k / 2) elements left in the list L.  
     Base Case: n = 4.  
     L = [4, 8, 2, 5]  
     Run with the Alg1(L, 4)  
     It removes the smallest and largest leaving L = [4, 5]  
     Then the algorithm ends because n – 2 = 2 is not > = 2. 2 is not greater than 2 and [4, 5] result does correctly remove the smallest and largest quarter of the array.  
       
     (4^k) – 4^(k-1)\*2 = (4^k) / 2  
     Base Case: k = 1  
     (4^1) – 4^(1-1)\*2 = (4^1) / 2  
      2 = 2  
     The base case is correct in removing the elements.  
     So I assume k = x is true, now I’ll prove for k = x + 1:  
       
     4^(x + 1) – 4^(x + 1 – 1)2 = (4^(x + 1)) / 2

4(4^x) – (4^x)2 = (4^(x + 1)) / 2

4(4^x) – 4(4^(x-1))2 = (4^(x + 1)) / 2

4((4^x) – (4^(x-1))2) = (4^(x + 1)) / 2

So (4^x) – (4^(x – 1))\*2 = (4^x) / 2, sub it into the left side

Therefore (4^(x + 1)) / 2 = (4 ^ (x + 1)) / 2

The case holds for x + 1. Therefore, it is proven correct by induction 😊

* 1. The time complexity is O(n2) with the assumption a doubly linked list is in use. Each iteration you step through the remaining items in the list, saving the smallest and largest values. After seeing all elements in L, remove them 2 \* O(1) for a linked list. This operation occurs n / 4 times, due to how many levels to the recursive call there are. Half of the elements are removed and each iteration removes 2. Therefore the expected runtime will be O(n2).
  2. Yes, I will sort it using an O(n log n) algorith. Then I will simply return the sorted array from positions n/4 to position 3n/4.

**private** **static** **boolean** isHeap(**int**[] arr, **int** i) {

**int** left = 2 \* i + 1;

**int** right = 2 \* i + 2;

// parent was a leaf

**if**(i >= arr.length)

**return** **true**;

// check for a left child and if it violates heap order

**if**(left < arr.length && arr[i] > arr[left])

**return** **false**;

// check for a right child and if it violates heap order

**if**(right < arr.length && arr[i] > arr[right])

**return** **false**;

// check that both children are a heap as well

**return** *isHeap*(arr, left) && *isHeap*(arr, right);

}

* 1. The time complexity of this algorithm is O(n). Each node of the heap is only run through the function once, and the operations of this are all constant. It satisfies the heap condition by default since the array is in heap-order, and we just check if the parent is less than their children and also that these children are heaps themselves.
  2. Establishing the correctness:  
     So all the possible versions of a heap include: empty, 1 node, and multiple nodes. The algorithm must state an empty and 1 node array to be heaps regardless. Then the more specific cases where there are 2+ nodes. In these cases the root / parents must always be less than both of their children or if they only have 1 then less than that for a min-heap. Then both sub-nodes from the root / parent will recurse and must satisfy the same heap conditions listed above.  
     So as long as the root is smaller than it’s children and all it’s children are also heaps then it will be correct. The algorithm coded will exit when it reaches a leaf and check left and right children individually as to not cause any conflicts when parents only have 1 child (left child).
  3. I would use Quicksort because finding a good pivot is essential for it to sort fast, and the median is the best pivot. O( n log n )  
     With this median algorithm, quicksort will
  4. No, because she doesn’t even check each value. The median could be the last value in the array, she would have to check at least every element i.e. O( n ). She would have to specify the use of a specific data structure, which she does not.   
     https://en.wikipedia.org/wiki/Median\_of\_medians#Proof\_of\_O(n)\_running\_time
     1. T(n) = 4 T(n / 4) + n / 4  
        a = 4, b = 4, f(n) = n / 4  
        Case 2: f(n) = Θ( ) = Θ(n) Which is true since   
        Therefore T(n) =
     2. T(n) = 4 T(n / 4) + O(1)  
        a = 4, b = 4, f(n) = O(1)  
        Case 1: f(n) = O() ϵ > 0  
         f(n) = O(n)  
        Therefore T(n) =
     3. T(n) = 8 T(n / 2) + n!  
        a = 8, b = 2, f(n) = n!  
        Case 3: Ω(n^(log2(8) + ϵ)) = Ω(n^(3 + ϵ)) for ϵ > 0  
        Check: a \* f(n / b) <= c \* f(n)  
         for some constant c < 1 and all sufficiently large n  
        8 \* (n / 2)! <= 0.5 \* n!  
         as n grows large, 0.5 \* n! will outgrow 8 \* (n / 2)!  
        Therefore T(n) = Θ(f(n)) = Θ(n!)
     4. T(n) = 8 T(n / 2) + log(n)  
        Case 1: f(n) is O(log n)

O(log n) <= O(n) <= O() <= O(n^(log2(8) – ϵ)) = O(n^(3-2)) where ϵ = 2 > 0  
Therefore T(n) = Θ()

* + 1. T(n) = 4 T(n / 2) – n  
       a = 4, b = 2, f(n) = n \* -1  
       -1 is the constant and the master theorem states this must be positive.  
       Therefore the master theorem is not applicable for this recurrence.
  1. Solve using iteration:   
     T(n) = T(n – 1) + 4n  
      = T(n – 2) + 4(n - 1) + 4n  
      = T(n – 3) + 4(n - 2) + 4(n – 3) + 4n  
      = 4 (n(n+1))/2 = 2 (n(n+1)) = 2n^2 + 2n = O()
  2. T(n) – 2 T(n - 1) + 4  
     Step 1: Guess T(n) <= c \* n  
     Step 2: Assume that T(k) <= ck for k < n for some constant c > 1  
     T(n) = 2 T(n – 1) + 4 <= 2c(n – 1) + 4 <= 2cn – 2c + 4 <= 2 (cn – c + 2)  
      <= 2(cn) since -c <= 2 for c > 1  
      <= 2 O(n)  
     T(n) = O(n)
  3. The time complexity of T(n) = 4 T(n) + O(log n)  
     This recurrence is infinite since the problem isn’t getting smaller, T(n) is repeated forever.
  4. Using iteration to solve this. Considering each level breaks down. There’s no need for substitution attempt to guess an upper bound. An estimated upper bound may be too lose, so we’ll use iteration.  
     T(n) = T(n / 4) + T(3n/4) + n  
     I will discard the T(n/4) since it’s the non dominant. For the recurrence tree, there will be levels, each level is O(n) work.   
     Making the total complexity O(n \* ) = O(n log n)

binarySearch (A, key, left, right)

if right >= left

mid = left + (right - left) / 2

if A[mid] == key

return mid

elif A[mid] > key

return binarySearch(A, key, left, mid - 1)

else

return binarySearch(A, key, mid + 1, right)

else

return -1

* 1. The time complexity of this algorith is O(log n).  
     The recurrence: a = 1, b = 2, d = 0  
     T(n) = 1 \* T() + O(1)  
     Using the simplified Master Theorem   
     We are therefore in Case B and thus get:  
     T(n) = O(nd \* log n) = O(log n)
  2. The resulting time complexity of the algorithm using a doubly linked list is increased to O(n). Which means it’s no better than just stepping through the list. It runs in O(n) time because you start by walking n/2 elements to the middle of the array, then check if you found it, otherwise walk n/4 elements back or forwards, and this sum will have a maximum value of n steps, the rest of the operations occur in constant time.
  3. If the array passed in each function call was copied each time, the time complexity would be:  
     T(n) = 1 \* T() + O(n / 2), where O(n / 2) used to be O(1) and represents the copying of the half the previous elements, still apart of the search. This accounts for all the recursive copying only. Yes it changes the time complexity, which is now O(n) = linear time. There are sub problems that each cost linear time, rather O(n / 2). You can apply the Master’s Theorem to this and use Case 3: you will get T(n) = Θ(n).