1. Theorem: A has a Ω(log n) lower bound on the number of calls to bicompare(i,j,x) that A must make in the worst-case.

Proof:

Let S be a sorted array.

There is a group of decision trees: T1, T2, T3… that describe the sequences of comparisons S makes on inputs with n elements.

Look at Tn:

The height of Tn is the number of comparisons for n elements in the worst case.

Let the active region of the algorithm start at size n, and let the number of probes through the sorted array to find x be k.

Since we know that S is a sorted array, that 0 ≤ i < j ≤ length[S]−1, and that for each comparison the algorithm must determine whether x is either greater than j, less than i, or between i and j, so there are at most three potential outcomes, and the decision tree would have a height of at least log\_3 n.

let k= log\_3 n

3^k = n

k log\_2 3 = log\_2 n

log\_3 n = log\_2 n / log\_2 3

Here, Tn has height of at least log\_3 n.

log\_3 n is in Ω(log n), therefore, we have a lower bound of Ω(log n).

2.

1. Proof: Partial sort is called several times where every value of h must be of the form h = 2^i\*3^j. With this information, we see that the last call will always be h = 2^0\*3^0, which means h = 1 and Insertion Sort is running on subsets of one. In this case, MysterySort is simply using Insertion Sort to sort every element one last time. Hence, MysterySort always produces the correct input.

b)

For pos 🡨 h to length[A]-1 do

j 🡨 pos – h

while (j ≥ 0 and A[j] > A[j+h]) do

swap A[j] and A[j+h]

j 🡨 j-h

endwhile

endfor

MysterySort(A, B, n): //where A is array, B is array for h values, and n is length array

Set i,j to 0

Let B be an array to store h values

While 2^i ≤ n do

While 2^i \* 3^j ≤ n do

Push to B new 2^i\*3^j value

Endwhile

i 🡨 i + 1

j 🡨 0

Endwhile

For index 🡨 0 to length[B]-1:

PartialSort(A,B[index])

Endfor

EndMysterSort