**Week 7 Tutorial Questions**

1. Write the **binary search** as a recursive algorithm.

int binarySearch(int arr[], int l, int r, int x)

{

   if (r >= l)

   {

        int mid = l + (r - l)/2;

        // If the element is present at the middle itself

        if (arr[mid] == x)  return mid;

        // If element is smaller than mid, then it can only be present

        // in left subarray

        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray

        return binarySearch(arr, mid+1, r, x);

   }

1. Prove that the binary search algorithm has a worst-case of O(lg n).
2. Suppose we are comparing implementations of the **insertion sort** and **merge sort** on the same machine. For inputs of size n, insertion sort runs in 8n^2, while merge sort runs in 64nlgn. For which value of n does insertion sort beat merge sort.

**Answer:**

43 = insertion sort(14792) < mergesort(14933)

44= insertion sort(15488) > mergesort(15373)

n=43

**COMBINING ALGORITHMS (maybe useful for assignment)**

1. **[COMPLEX] – MERGE SORT + INSERTION SORT**

Although merge sort runs in O(*nlgn*) worst-case time and insertion sort runs in O(*n^2*) worst case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Therefore it makes sense to **coarsen** the leaves of the recursion (reduce the size of the tree) by using insertion sort with merge sort when sub-problems become sufficiently small. Consider a modification to merge sort in which *n/k* sublists of length *k* are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

* 1. Show that insertion sort can sort *n/k* sub-lists, each of length, *k*, in O(*nk*).

**Answer:** Each sublist takes O(k^2). There are n/k sublists.=> (k^2)\*(n/k)=nk

* 1. Show how the amended merge sort algorithm runs in O(*nlg(n/k*)).

**log(n/k)** = Recursive part of the algorithm

**n**  = merge part of the algorithm - requires O(n) work at each level since we are still working on n elements even if they are partitioned into sub-lists.

=> nlog(n/k) NOTE: This does not include the insertion sort

**Modified Merge Sort**

*If n = k*

*Return*

*Else*

*mid = (low+high)/2*

*Merge\_Sort(A, low, mid)*

*InsertionSort(A,low,mid)*

*Merge\_Sort(A, mid+1, high)*

*InsertionSort(A,mid+1,high)*

*Merge(A, low, mid, high)*

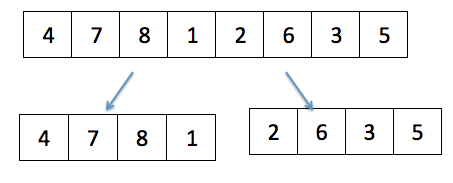
**(a)**

For example:

**n** (number in original array of numbers): 8

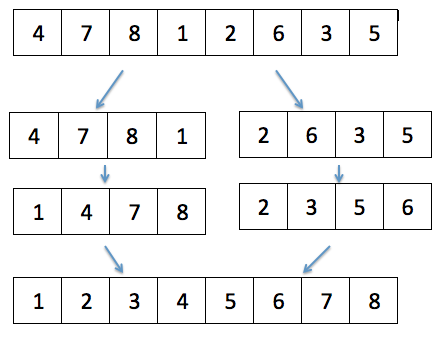
**k** (number in sub lists): 4

n/k (number of sub lists): 2



**k^2** (insertion sort) **\* n/k** (number of sub lists) = **nk** = 32

**(b)**



**n**

**Log ?**

The amended algorithm does not include the insertion sort at the moment.

Normal merge sort = **n log n** = 8 log 8 = 8\*3 = 24

Amended merge sort = **n log ?** = 8 log ? =

Log 8 = 3 means that the divide and conquer occurs 3 times.

In this case, the divide and conquer only occurs 1 time =>

Log ? = 1

?=2 = n/k

Log n/k = 1

The amended merge sort is **nlog (n/k)**

1. **[COMPLEX]** – **INSERTION SORT + LINEAR SEARCH**

The insertion sort algorithm uses a linear search to scan backwards through the sorted sub array. Could the **binary search** be used to improve the overall worst-case running time of the insertion sort to O(nlg(n))? Write this algorithm.