**Problem 1:**

elementEqualsIndex(A, start, end, m):

if A[⌈n/2⌉] == m + ⌈n/2⌉

return TRUE

if |A| ≤ 1,

return FALSE

if A[⌈n/2⌉] < m + ⌈n/2⌉,

return elementEqualsIndex (A, ⌈n/2⌉ + 1, n, m + ⌈n/2⌉)

else

return elementEqualsIndex (A, 0 , ⌈n/2⌉ − 1) , m)

The first call to this algorithm will be elementEqualsIndex(A, 0, n, 0). This algorithm is correct because if A[⌈n/2⌉] < m + ⌈n/2⌉, then none of the A[i] can be equal to i for all i < ⌈n/2⌉. This is because all the items are distinct and sorted. The same argument holds for the case when A[⌈n/2⌉] > ⌈n/2⌉ + m. This is almost like a tweak binary search, that doesn’t search for a specific value, but instead checks to see if any A[i] =i.

The running time is T (n) = T (n/2) + O(1), which by the Master Theorem discussed in the previous homework is **O(log n).**

**Problem 2:**

function binary\_search(A, l, r, v)

Input: Array A indexed from l to r with items sorted from smallest to largest, and item V to search for

Output: Location of item v in an array. If not found, -1 is returned.

if (l>r)

return (-1)

m = floor((l+r)/2)

if (A[m] == v)

return m

if (A[m] < v)

return binary\_search(A, m+1, r, v)

else

return binary\_search(A, l, m-1, v)

function find\_intersection(aBig, aSmall)

Input: Array aBig (larger size) and array aSmall (smaller size)

Output: Array result with the matching intersection elements.

result = []

for (i = 0; i< Asmall.size(); i++){

exists = binarySearch(Abig, 0, aBig.size()-1, aSmall[i]) //Search for element in bigger array

if (exists != -1) //If element exists in the larger array, push it to results

results.push aSmall[i]

}

return result

**Binary Search Algorithm:**

The binary search algorithm above is correct since it follows the principles of binary search. It checks to make sure the upper bound of the array is greater than the lower bound of the array, and if it is, it returns -1 (Not found). Otherwise it will divide the array in two pieces and checks to see if the element is found at m. If it is, it returns the location. Otherwise it checks to see if the current element is less than the search item and if it is, it recursively calls binary\_search from m+1 to the end of the array. Else If the current element is greater, it recursively calls binary\_search from the start to m-1.

The runtime of this algorithm O(log(n)) since the recursive relationship can be expressed as:

Which, according to the master’s thereom covered in the previous homework is

**Preprocessing and Query Algorithm:**

The running time of the above algorithm is This is because our binary search utilizes a run time of and we loop over the smaller array once so . Worst case scenario if both array are the same size which would give us , where n is the size of the large array. This is asymptotically better than

**Problem 3:**

B = new vector

C = new vector

function preprocess(A){

Input: array A where A is the list of items.

Output: B and C (global) are filled in .

B.resize(A.size, 0) // Initialize B to all 0s at same size of A. O(k)

for (i = 0; i< A.size; i++){

B[A[i]] = B[A[i]] + 1 //increment the count. O(n)

}

C = B

for (i = 1; i< C.size; i++){ // C will hold the count of all the previous counts

C[i] = B[i]+B[i-1] // O(k)

}

}

function count\_in\_range(a, b){

Input: start range: a and end range: b

Output: Number of n integers in that range

return C[b]- C[a] + b[a] // Requires O(1)

}

Assuming that our integers are stored in array A[1 · · · n] length[A] = n. We will use to addition arrays B and C at the same size of n. The algorithm will first initialize all elements of array B to 0. This will require O(k). Next, for each element of array A we will increment B[A[i]]. This will take O(n) time. Finally, making C[1] = B[1] and for each element starting from element we will compute C[l]=B[l]+B[l−1]. This step takes O(k)time again.

To answer any query, we simply return C[b] – C[a] + b[a] which requires O(1) time.

Thus our preprocessing takes O(n+k) and our query time is O(1).

**Problem 4:**

This is a very specific case since we know the array size. We initialize our result array of size [ with all zeros. We loop over the given array and for every element, a[i], we increment that index at result[a[i]] by 1. This will store the count of each element in our original array at the respective index. This algorithm is correct because we are not physically sorting the array, instead, we are using a different array to store the number of elements for each index. This is very similar to the category sorting technique described in class.

Using this algorithm, we have sorted our array in **O(n)** time and our we can obtain the sorted array by writing each element as many times as it occurred.