1. T(n) = T(n/2) + n; T(1) = 1 can be rewritten as T(n) = 1 \* T(n / 2) + 1 \* n1 where a=1, b=2, c=1, k=1, d=1. With our master formula 1 < 21 then **T(n) = O(n2)**.
2. (a). In the worst case isPrime method will check all numbers between 1 to n-1 with in each step is O(1) which is the algorithm is running time of o(n).

public static int isPrime(int n) {

    return isPrime(n, n - 1);

}

public static int isPrime(int n, int d) {

    if(d == 1) { return 1; }

    if(n == 1 || n % d == 0) { return 0; }

    return isPrime(n, d - 1);

}

(b). T(n) = T(n – 1) + c, T(1) = 4; which is T(n) = O(N)

(c). T(b) = O(2b)

1. On this algorithm I am using 2 recursive helper functions gcd, and isPrime. As we studied in class and above on problem2 both of these functions runs o(n). Also the while loop that extracts the power of 2 is o(log(n)). So, the running time of our algorithm is o(n) + o(n) + o(log(n)) which is o(n).

public static boolean problem3(int m, int n) {

    int d = gcd(m, n);

    while (d % 2 == 0) {

        d /= 2;

    }

    if(d > 1 && isPrime(d) == 1) {

        return true;

    }

    return false;

}

1. (A). Merge: I(i): The i smallest elements of A union B occur in S in sorted order

I(0) is true, obviously. If I(i) is true, the smallest element x that remains in A union B is placed at the end of S. Because A and B were already sorted, x is larger than all elements of S so far.

MergeSort: Valid Recursion. Base case when L has size 0 or 1. Self-calls reduce input size by ½ so they lead to base case. Base Case Correct. Lists of length 0 or 1 are already sorted Recursive Steps Correct. Assuming MergeSort is correct for lists of length < n, when we run MergeSort on a list L of length n, partition step produces sublists L1, L2 of smaller length and so MergeSort correctly sorts each. Then merging combines them into a single sorted list, which is returned.

(B). T(1) =d, T(n) <= 2\*T(n/2) + 2n then the runtime of the algorithm is O(n\*log n) by the master formula

(C) The comparison between 2 sort algorithm results the MergeSort is much faster.

With ARRAY\_SIZES = {10000, 20000, 40000, 10000}:

86 ms -> MergeSort

168 ms -> LibrarySort

ARRAY\_SIZES = {100000, 200000, 400000, 100000}:

734 ms -> MergeSort

2617 ms -> LibrarySor

public int[] sort(int[] arr) {

    if(arr.length == 1) {

        return arr;

    }

    // Partition

    int p1Length = arr.length / 2;

    int p2Length = arr.length - p1Length;

    int[] p1 = new int[p1Length];

    int[] p2 = new int[p2Length];

    for(int i = 0; i < p1Length; i++) {

        p1[i] = arr[i];

    }

    for(int i = 0; i < p2Length; i++) {

        p2[i] = arr[p1Length + i];

    }

    // Recursion

    p1 = sort(p1);

    p2 = sort(p2);

    // Merge

    int p1i = 0;

    int p2i = 0;

    for(int i = 0; i < arr.length; i++) {

        if(p1i == p1Length || (p2i < p2Length && p1[p1i] > p2[p2i]) ) {

            arr[i] = p2[p2i];

            p2i ++;

        } else {

            arr[i] = p1[p1i];

            p1i ++;

        }

    }

    return arr;

}