1. Write pseudocode that exactly represents your implementation of merge sort in this assignment, using CLRS pseudocode conventions. You must use the function signatures that are given in this assignment. Your pseudocode must include merge, mergesort, and sortIntegers.

```
Merge(A, temp, l,  m, r)
1       lp = l
2       rp = r
3       cur = l
4       while lp <= m and rp <= r
5               if A[lp] <= A[rp]
6                       temp[cur] = A[lp]
7                       cur ++
8                       lp ++
9               else
10                      tmp[cur] = A[rp]
11                      cur ++
12                      lp ++
13              while lp <= m
14                      temp[cur] = A[rp]
15                      cur ++
16                      rp ++
17              while rp <= r
18                      temp[cur] = A[lp]
19                      cur ++
20                      lp ++
21              cur = l
```

```
22              while cur <= r
23                      A[cur] = temp[cur]
24                      cur ++
```

```
Merge-Sort(A, temp, l, r)
1       if l >= r
2               return NIL
3       m = (l + r) / 2
5       Merge-Sort(A, temp, l, m)
6       Merge-Sort(A, temp, m+1, r)
7       Merge(A, temp, l, m, r)
```

```
Sort-Integers(A, size)
1       tmp[size]
2       Merge-Sort(A, tmp, 0, size-1)
```

2. State what you think the worst-case big-O complexity and the best-case big-O complexity of merge sort is, and what data conditions create the best and worst case. Explain why you think so.

best case and worst case is both O(n*log n)

best case condition is when array is sorted, best case is O(n*log n) because in this case merge sort recursively divides input array into two half until one element left and then merge, still need to do log2(n) level of recursive call, each level is o(n) work

worst case is when array is sorted in reverse order, and merge sort recursively divides input array into two half until one element left and then merge need do do log2(n) level of recursive call, each level is o(n) work

3. Write the number of operations per line for your pseudocode for merge, mergesort, and sortIntegers. Do your best here.

Merge:

line 1-3: 1 operation

line 4: n+1 operation

line 5-13: n/2 operation

line 14-16: n*(m-l+1) operations

line 17-20: n*(r-m) operations

line 21: n operation

line 22-24: n*(r-l+1) operations

Merge-Sort:

line 1-4: 1 operation

line 5-6: 2*T(n/2) operation

line 7: n operation

Sort-Integers:

line1: 1 operation

line2: n*logn operation

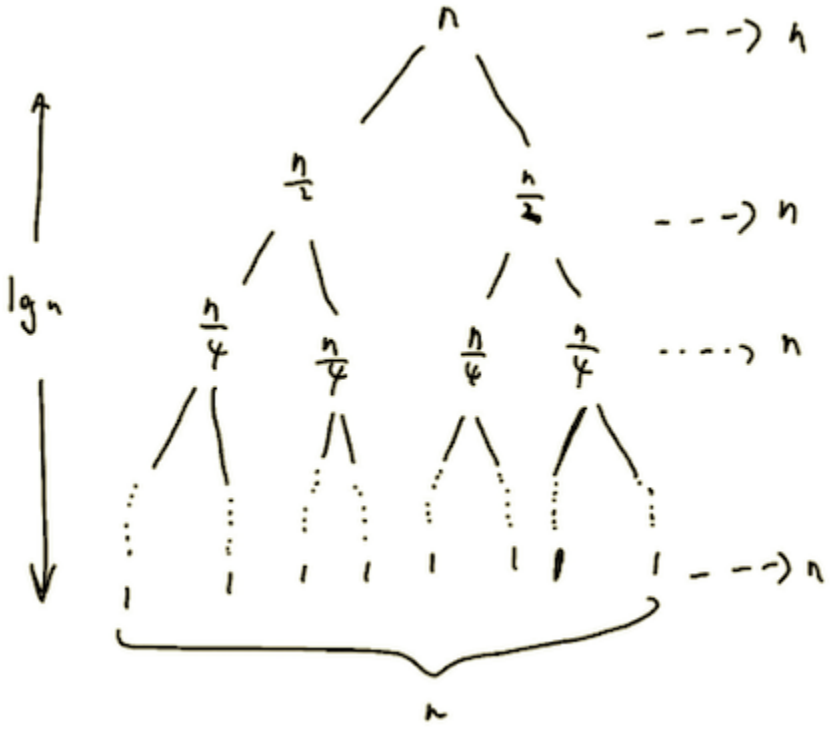
4. State the T(n) recurrence equation for your implementation of mergesort. Explain why you think the operations in (3) create this T(n) recurrence equation: specifically, make sure that you explain what is contributed by merge, by

mergesort, and by sortIntegers. The T(n) recurrence equation does not have coefficients for *n*, but it does have coefficients for *T* as necessary.

T(n) = 2*T(n/2) + n

Because MergeSort function recursively divide input data into two parts of size n/2, which explains 2*T(n/2) part, and then call Merge Function to merge these parts together, which takes O(n) work to merge. So recurrence equation is T(n) = 2*T(n/2) + n

5. Draw the recurrence tree for your recurrence equation in 4. Label the height of the tree in terms of *n*, the width of the tree in terms of *n.* Label the amount of work done at every level of the tree, and explain in English sentences which part of the algorithm is doing the work at every level (indicate which line of code is doing the work).



Explain:

At each level, of recurrence tree, we divide the input into two parts, in line 5-6 of Merge-Sort function, we do recursive call to itself on left and right subarray, each have size of n/2.

Then after dividing until that each sub-array only contains one element, in line 7 of Merge-Sort function we merge subarrays, the work of merging is n and each level.

6. Prove that the height of the tree is *lg n* .

Proof: Suppose input array has size of n, at each level we divide input array into two parts until that the subarray has size 1. After one recursive call, size of input is n/2, then n/4, after k-th recursive call, each subarray size is n/(2**k).

Since we stop while subarray has size 1, we have n/(2**k) = 1

-> 2**k = n

-> k = log_2(n)

-> as a result we prove that height of the tree is lg n

7. Solve the recurrence equation in 4 using the substitution method. Show your work, and use English sentences to explain what each step is doing.

Since T(n) = 2T(n/2) + n, we can replace n with n/2 and get:

-> T(n/2) = 2T(n/4) + n/2

We substitute T(n/2) with 2T(n/4) + n/2 to T(n) = 2T(n/2) + n and get:

-> T(n) = 2(2T(n/4) + n/2) + n

-> T(n) = 4*T(n/4) + 2n

We substitute n/2 with n/4 to T(n/2) = 2T(n/4) + n/2 and get:

-> T(n/4) = 2T(n/8) +n/4

We substitute T(n/4) with 2T(n/8) +n/4 to T(n) = 4*T(n/4) + 2n and get:

-> T(n) = 8*T(n/8) + 3n

…

At kth iteration: we have T(n) = 2^k * (n/2^k) + k*n

let n = 2^k, then we have n/2^k = 2^k/2^k = 1, so T(n/2^k) = T(1) is constant

we can simplify T(n) as T(n) = 2^k * T(1) + k *n

since T(1) is constant

we can simplify T(n) = 2^k + k*n

since we let n = 2^k, we have log_2(n) = k

we can simply T(n) = 2^(log_2(n)) + log_2(n)) * n

we simplify T(n) = n + log2(n) * n

Since n*log2(n) grows faster than n because log2(n) > 1 when n > 0

We can drop the term n when computing time complexity,

T(n) = log2(n) * n and proved that T(n) is O(n*log n)

//

8. Solve the recurrence equation in 4 using the master method. Show your work, and use English sentences to explain what each step is doing.

In master method, T(n) = a*T(n/b) + O(n^d)
since a = 2, b = 2, d = 1
we have log_b(a) = 1
we have d = log_b(a)
the we have T(n) = O(n^d * log(n)) = O(n * log(n))
//