4(i) Time complexity

Sort function in bottom up approach involves two for loops. The first for loop represents size of the sub array considered. Size of the subarray can be represented as 2^x

Subarrays considered at each interval are of length 2^{size} where x>=1 and maximum length of the sub array considered can be n.

Therefore, we have $2^{size} = n$

taking log on both sides we have

size * log2 = logn

⇨ $size = \frac{logn}{log2}$

⇨ $size = \log_2 n$

Therefore first for loops executes for $\log_2 n$ times

Second for loop represents the total no of merges for each subarray size considered. If we have an array of size 8 and subarray size = 2, then no of merges = 4,
Similarly, for subarray of size = 4, no of merges = 2.
Further for subarray of size = 8, no of merges = 1

From the above the explanation we can generalize and say:
no of merges = $\frac{n}{2^x}$, for any level x>=1

Where, n is the total array length
x is the size of the sub array considered for merge

Total number of merges = $n[1/2+1/2^2 + 1/2^3 + 1/2^4 + \ldots\ldots + 1/2^{\log_2 n}]$ -> Equation 1

We can solve the geometric series using the formula
$s_n = \frac{a(1- r^n)}{(1-r)}$

now by substituting a =1/2 and r = 1/2 , we have
$s_n = \frac{½ (1-1/2^{\log_2 n})}{(1-1/2)} = (1-1/2^{\log_2 n}) = (1-1/n)$ (using $2^{\log_2 n} = n^{\log_2 2} = n$)

Therefore, total number of merges = n (n-1)/n = (n-1)

Hence, the execution time of the two for loops in sort function = nlogn – logn

The time taken by the remaining steps is constant. Let that be C

Now we have T(n) = = nlogn – logn + C

By ignoring all the smaller terms and considering the largest term cost, We have
T(n) = O(nlogn)

4(ii):

Advantages of bottom-up merge sort over top-down merge sort:

Though both top-down and bottom-up merge sort involves the same worst-case time complexity, bottom-up approach for large input size runs faster than the top-down approach, as it involves loop iterations.

Top-down approach involves recursion which uses stack frames to keep track of return address. Setting up and tearing down stack frames is expensive. For large inputs, we might encounter stack overflow. Compilers of few functional programming languages optimize tail recursions but compliers for languages like JAVA and Python, tail call optimization is not supported.

So, while working with large data set and non-functional programming languages like Java or Python, bottom-up approach would be a better choice for merge sort.