1 Roughly how many levels does the recursion tree of $MergeSort$ have, as a function of the length n of the input array?

1. A constant number
2. $log_2 n$
3. $\sqrt{n}$
4. n

(2) we continously invoke the recursive call dividing the problem by 2 into we get empty arrays or arrays of 1. This directly leads to the definition of the base-2 logarithm hence $log_2 n$ is the level of the recursion tree.

2 What is the pattern? Fill in the blanks in the following statement: at each level $j = 0, 1, 2 \ldots$ of the recursion tree, there are [blank] subproblems, each operating on a subarray of length [blank].

1. 2^j and 2^j, respectively
2. $\frac{n}{2^j}$ and $\frac{n}{2^j}$, respectively
3. 2^j and $\frac{n}{2^j}$, respectively
4. $\frac{n}{2^j}$ and 2^j, respectively

(3) is the correct answer. In the base case when $j = 0$ we have $2^0 = 1$ sub problems and $\frac{n}{2^0} = n$ length subarray. the next level will have $2^1 = 2$ subproblems as the array has been split into a lower half and upper half and these arrays are of length $\frac{n}{2^1} = \frac{n}{2}$. Thinking inductively we conclude that the number of subproblems and lengths of the subarray in level $j = i$ are 2^i and $\frac{n}{2^i}$ respectively.

3 Suppose we run $MergeSort$ on the following input array: [5, 3, 8, 9, 1, 7, 0, 2, 6, 4]. Fast forward to the moment after the two outermost recursive callse complete, but before the final $Merge$ step. Thinking of the two 5-element output arrays of the recursive calls as a glued together 10-element array, which number if in the $7th$ position?

the left-half subarray is [1, 3, 5, 8, 9] and the right half subarray is [0, 2, 4, 6, 7]. gluing the subarrays we get [1, 3, 5, 8, 9, 0, 2, 4, 6, 7]. The 7th element is 2.

4 Consider the following modification to the $MergeSort$ algorithm: divide the input array into thirds recursively sort each thig, and finally combine the results using a three-way $Merge$ subroutine. What is the running time of this algorithm as a function of the length n of the input array, ignoring constant factors and lower-order terms?

1. n
2. $n \cdot logn$
3. $n \cdot (logn)^2$
4. $n^2 \cdot logn$

(2) is the correct answer. The recurrence of the three way mergesort is of the form of $T(n) = 3 \cdot T(\frac{n}{3}) + O(n)$. We apply the $Akra - Bazzi$ formula $T(n) = \Theta(n^p(1 + \int_1^n \frac{g(u)}{u^{p+1}} du))$ where p satisfies $3 \cdot (\frac{1}{3})^p = 1$. In this case $p = 1$ and $g(u) = n$. Substituting we obtain

$$\begin{aligned} \Theta(n^1 \cdot (1 + \int_1^n \frac{u}{u^2} \cdot du) &= \Theta(n^1 \cdot (1 + \int_1^n \frac{u}{u^2} \cdot du) \\ &= \Theta(n + n \cdot lnn) \\ &= \Theta(n \cdot lnn) \\ &= \Theta(n \cdot logn) \end{aligned}$$

5 Suppose you are given k sorted arrays, each with n elements, and you want to combine them into a single array of $k \cdot n$ elements. One approach is to use the $Merge$ subroutine from Section 1.4.5 repeatedly, first merging the first two arrays, then merging the result with the third arrays, then with the fourth array, and so on until you merge in the kth and final input array. What is the running time taking by this successive merging algorithm, as a function of k and n, ignoring constant factors and lower-order terms?

1. $n \cdot logk$
2. $n \cdot k$
3. $n \cdot k \cdot logk$
4. $n \cdot k \cdot logn$
5. $n \cdot k^2$
6. $n^2 \cdot k$

(5) is correct.

$$\begin{aligned} T(1)+T(2)+T(3)+\ldots+T(k-1) &= (n+n)+(n+2\cdot n)+(n+3\cdot n)+\ldots+(n+n\cdot(k-1)) \\ &= 2\cdot n+3\cdot n+4\cdot n+\ldots+k\cdot n \\ &= n\sum_{i=1}^{k} i-n \\ &= \frac{n\cdot k(k-1)}{2}-n \\ &\leq n\cdot k^2 \end{aligned}$$

6 Consider again the problem of merging k sorted length n arrays into a single length-kn array. Consider the algorithm that first divides the k arrays into $\frac{k}{2}$ pairs of arrays, and uses the $Merge$ subroutine to combine each pair, resulting in $\frac{k}{2}$ sorted length $2\cdot n$ arrays. The algorithm repeats this step until there is only one length-$k\cdot n$ sorted array. What is the running time of this procedure, as a function of k and n, ignoring constant factors and lower-order terms?

1. $n\cdot logk$
2. $n\cdot k$
3. $n\cdot k\cdot logk$
4. $n\cdot k\cdot logn$
5. $n\cdot k^2$
6. $n^2\cdot k$

(3) is correct. We first use the divide and conquer principle to continously divide the problem in half. We have $O(log_2 k)$ operations. From this we do the merge as state above we merge $\frac{k}{2}$ arrays of size $2\cdot n$. Hence we have the equation $\sum_{i=1}^{\frac{k}{2}} 2\cdot n = 2\cdot n\cdot\sum_{i=1}^{\frac{k}{2}} 1 = n\cdot k$ operations. Hence we have $O(log_2 k\cdot(n\cdot k)) = O(log_2 k\cdot n\cdot k)$ operations.

7 You are given as input an unsorted array of n distinct numbers, where n is a power of 2. Give an algorithm that identifies the second largest number in the array, and that uses at most $n+log_2 n-2$ comparisons.

Find the largest element by comparing pairwise elements. Continue doing so until you find the largest element. This step requires $n-1$ comparisons. The largest was compared to by $log_2(n)$ elements so if we do the same method call to

this array we will get $log_2(n)-1$ comparisons. thus in total we have $n+log_2(n)-2$ comparisons.