1. Let $T(n) = T(an) + T(bn) + n$ for some constants $0 < a \leq b < 1$.

The cost at each level of the recurrence tree is the sum of contributions from each node at that level:

Level 0: $T(n)$
Level 1: $T(an) + T(bn)$
Level 2: $T(a^2n) + T(abn) + T(b^2n)$
Level 3: $T(a^3n) + T(a^2bn) + T(ab^2n) + T(b^3n)$
Level 4: $T(a^4n) + T(a^3bn) + T(a^2b^2n) + T(ab^3n) + T(b^4n)$

Using the pattern observed, we can generalize the cost at the kth level: $\sum_{i=0}^{i=k} (T(a^i \cdot b^{(k-i)} \cdot n)$

- shortest branch of recurrence tree: minimum exponent of n only occurs when we only choose $T(bn)$ at each step, so the exponent of n after k steps will be b^k.
- it is the opposite for the longest branch, which is the maximum exponent of n when we choose $T(an)$ at each step, so it would be a^k.
- hence the total cost at all levels would be $\sum_{k=0}^{k=\log_a(n)} \left(\sum_{i=0}^{i=k} (T(a^i \cdot b^{(k-i)} \cdot n) \right)$

a) $T(n) = \Theta(n)$ if $a+b < 1$

according to the constraint $a+b<1$, then the sum $a^i \cdot b^{k-i}$ decreases as k increases, and the terms $T(a^i \cdot b^{k-i} \cdot n)$ become smaller & smaller. The total cost will be dominated by the first few levels because there is only a finite number of levels in the $\log_a(n)$ tree. The total cost must be $\Theta(n)$, and $T(n) = \Theta(n)$.

b) $T(n) = \Theta(n \log n)$ if $a+b=1$

Due to the constraint $a+b=1$, the sum $a^i \cdot b^{k-i}$ would remain constant for all k:

$a^i \cdot b^{k-i}$
$= a^i \cdot b^k \cdot b^{-i}$
$= a^i \cdot (a^{-1})^i$
$= 1^i = 1$

$\Rightarrow$ since the sum for each level of k is constant and there are $\log_a(n)$ levels in the tree, the total cost will be $\Theta(n \log n)$, hence $T(n) = \Theta(n \log n)$.

2. Given k sorted arrays, each having n elements. Combine sorted arrays to form one array with kn elements. Sorted arrays are called $A_1, A_2, A_3, \ldots, A_k$.

a) Iteratively merge A_1 & $A_2 \rightarrow B_1$. Then B_1 & $A_3 \rightarrow B_2$. Repeat till kn sized sorted array. Time complexity?

- merge A_1 (n elements) & A_2 (n elements) $\rightarrow B_1$ ($2n$ elements) $\rightarrow O(n)$
- merge B_1 ($2n$ elements) & A_3 (n elements) $\rightarrow B_2 = O(2n) \rightarrow O(n)$
- merge B_2 ($3n$ elements) & A_4 (n elements) $\rightarrow B_3 = O(3n) \rightarrow O(n)$

⋮

- continue process until all arrays are merged, last merge takes $O(kn)$ time since the previous merge has $(k-1)n$ elements, and A_k has n elements.

The time complexity overall for the iterative merge algorithm is the sum of merge operations $= O(n) + O(2n) + O(3n) + \ldots + O(kn)$

$= O(n \cdot (1+2+3+\ldots+k))$

arithmetic series formula

$1+2+3+\ldots+k = \frac{k(k+1)}{2}$

Hence $T(kn) = O(n \cdot \frac{k(k+1)}{2})$.

The time complexity for the iterative merge algorithm is $O(nk^2)$.

b) Use divide & conquer to find a more efficient algorithm to merge the k arrays. Time complexity?

The approach above works fine for $k \leq 2$. For $k > 2$, divide k arrays into 2 groups and recursively merge them using the same divide & conquer approach. Then merge both sorted groups using a merge procedure, taking advantage of the fact that both groups are already sorted. Starting at $k > 2$, denote the time complexity of merging k arrays of size n each: $T(kn) = 2 \cdot T(\frac{kn}{2}) + O(kn)$.

recursively merge each group

split k arrays in 2 groups

time for merge

$= 2 \cdot (2 \cdot T(\frac{kn}{4}) + O(\frac{kn}{2})) + O(kn) = 4 \cdot T(\frac{kn}{4}) + O(kn) + O(kn)$

$= 4 \cdot (2 \cdot T(\frac{kn}{8}) + O(\frac{kn}{4})) + O(kn) = 8 \cdot T(\frac{kn}{8}) + O(kn) + O(kn) + O(kn)$

$\vdots$

$T(kn) = 2^{\log k} \cdot T(\frac{kn}{2^{\log k}}) + O(kn) + O(kn) + \dots \log k \text{ terms}$

$O(1)$ ↑ $2^{\log k}$ ↘ k

$= k \cdot T(n) + O(kn) \cdot \log k$

$= O(k) + O(kn \log k)$

$= O(kn \log k)$

3. sorted arrays A & B → both $\frac{n}{2}$ elements. find $\frac{n}{2}$th smallest element in A∪B.

a) Give an $O(n)$ time algorithm to solve this problem. Justify correctness & the time taken by your algorithm.

The essence of this algorithm is a merge step similar to the one in merge sort, merging while keeping track of the number of elements already merged until the $\frac{n}{2}$th number is reached:

1. Initialize a pointer for array A as well as array B pointing to the first element of each array.
2. Initialize a counter for elements already merged
3. As long as the count is less than $\frac{n}{2}$:
   - compare elements at the pointers of arrays A & B.
   -
4. The $\frac{n}{2}$th smallest element is the current element at the position of the pointer that was just moved in the last step.

Each step moves the pointer of the smaller element. Both arrays are already sorted, the smaller elements come first, and the pointers increment through in ascending order. When the $\frac{n}{2}$th smallest element is reached, the pointer that was last moved points to it. Since both arrays are iterated through at the same time, and n represents the total number of elements in both A & B, the time complexity of this algorithm is $O(n)$.

b) solve this problem in $O(\log n)$ time using divide & conquer

1. compare the medians from both arrays A & B.
   - if they are equal, the $\frac{n}{2}$ th smallest element is found.
   - if median of B > median of A, the $\frac{n}{2}$ th smallest element is either in:
     - elements > median of A
     - elements $\leq$ median of B
   - if median of A > median of B, the $\frac{n}{2}$ th smallest element is either in:
     - elements > median of B
     - elements $\leq$ median of A
2. Recursively apply the above step until the median from both arrays A & B are equal, which means the $\frac{n}{2}$ th smallest element is found.

The time complexity of this algorithm is $O(\log n)$, because at each step, the problem size is reduced by roughly half.

4. An array of n elements is said to have a majority element if more than half of its entries are the same. Provide an algorithm that outputs the majority element on an input array, or input none if that is the case.

a) Show that the frequencies of all elements can be $O(n^2)$.

1. Initialize majority element & counter variables
2. For each element in the array:
   - initialize a counter to keep track of the frequency of the current element
3. For each element in the array:
   - increment the counter if the element is equal to the current element

- if the loop counter is greater than the first counter, update the majority element to the current element and the first counter to the loop counter.

4. After the loop terminates, check if the majority element $> \frac{n}{2}$. if not, then there is no majority element.

This algorithm iterates through the array and computes the frequency of each element with the use of a nested loop. By doing so, it identifies the majority element if it exists. The time complexity is $O(n^2)$ because the outer loop runs n times and the inner loop runs n times.

b) Show that the frequencies of all elements can actually be computed in $O(n \log n)$ time.

1. Divide the array into equal halves
2. Recursively compute the frequencies of all elements in both halves of the array
3. merge the frequencies of elements from either half
4. Check if each element merged has a count greater than $\frac{n}{2}$. If it does, then it is the majority element.

This approach reduces the problem size by half at each step, dividing the time complexity by 2. Dividing the array takes $O(1)$ time. Recursively computing frequencies on both halves takes $O(n \log n)$. merging frequencies takes $O(n)$ time. The overall time complexity of this algorithm is $\underline{O(n \log n)}$.

5. Consider an array to have n integers. The middle half would have ranks in the range $[\frac{n}{4}, \frac{3n}{4}]$ (both inclusive)

a) Provide an $O(n \log n)$ algorithm that returns the middle half.

1. Sort the array using a comparision based sorting algorithm with $O(n \log n)$ complexity, such as merge sort or quick sort.
2. Use the range of $[\frac{n}{4}, \frac{3n}{4}]$ and get the middle half.

The sorting algorithm takes $O(n \log n)$. It takes $O(\frac{n}{2})$ $= O(n)$ time to return the half. Therefore, the overall time complexity must be $O(n \log n)$.

b) Prove there is a more efficient algorithm that returns the middle half in $O(n)$ time.

1. Use quickselect to find the $\frac{n}{4}$th element
2. Use quickselect to find the $\frac{3n}{4}$th element
3. traverse the array to find the middle half of the array by collecting elements $\geq \frac{n}{4}$ & $\leq \frac{3n}{4}$.

Quickselect finds a specific element in $O(n)$ time. By finding the $\frac{n}{4}$ & $\frac{3n}{4}$ elements it sets the lower and upper bounds, and traversing the array between these bounds takes $O(n)$ time. The time complexity is dominated by the two Quickselect algorithms and the array traversal, which takes $O(n)$ time.

6. (Extra Credit) Given an array of n integers, a pair of elements (a_i, a_j) are out of order if $i < j$ & $a_i \geq a_j$.

a) Provide an $O(n^2)$ algorithm that counts the number of out of order pairs in the given array.

1. Initialize a counter variable to keep track of the number of out of order pairs.
2. For each element a_i in the array:
   - For each element a_j after a_i:
     - increment the count if $a_i \geq a_j$
3. Return the count

The time complexity of this algorithm is $O(n^2)$ because the outer loop runs n times, and the inner loop runs $n-1$ times worst case.

b) Use divide & conquer to make a more efficient algorithm The goal running time of your algorithm is

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

1. Divide the given array into 2 halves and initialize left & right indices
2. If the left and right indeces are equal, there are no out of order pairs
3. Calculate a middle index as $\frac{left + right}{2}$
4. Recursively count the out of order pairs on the left side of the array
5. Recursively count the out of order pairs on the right side of the array
6. Merge both halves and count the out of order pairs across both halves

This divide & conquer method divides the problem size into a smaller subproblem size of $\frac{n}{2}$, which makes up the $2T(\frac{n}{2})$ portion, and merging the out of order halves across both halves makes up the $O(n \log n)$ portion, effectively making the running time of the whole algorithm $T(n) = 2T(\frac{n}{2}) + O(n \log n)$