The Introduction to Algorithms text has some exercises on page 106 (specifically 4.5-1). In addition, I have made up a few below.

Use the Master Theorem to find a, b, $f(n)$, and the complexity of a function. In all cases the function being analyzed that takes an array of size n as its parameter. Simplify all logarithms as far as possible.

Theorem 4.1 (Master theorem)

Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by

$$T(n) = aT(n/b) + f(n),$$

where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ∎

---

1. The function that takes an array of size n and does three recursive calls each on 25% of the array. The other work done in the function is a single for loop that runs through the input array one time and does one operation to each array element.

2. The function that takes an array of size n and does three recursive calls each on 25% of the array. The other work done in the function is all constant time.

3. The function that takes an array of size n and does four recursive calls each on 25% of the array. The other work done in the function is a single for loop that runs through the input array one time and does one operation to each array element.

4. The function that takes an array of size n and does four recursive calls each on 25% of the array. The other work done in the function is all constant time.

5. The function that takes an array of size n and does two recursive calls each on 25% of the array. The other work done in the function is a single for loop that runs through the input array one time and does one operation to each array element.

6. The function that takes an array of size n and does two recursive calls each on 25% of the array. The other work done in the function is all constant time.

7. The function that takes an array of size n and does four recursive calls each on 20% of the array. The other work done in the function is two *nested* for loops. The first runs through the entire array (i from 0 to $n-1$), and the second runs from i to the end of the array (j from i to $n-1$).

8. The function that takes an array of size n and does four recursive calls each on 20% of the array. The other work done in the function is a single for loop that runs through the input array one time and does one operation to each array element.

9. The function that takes an array of size n and does seven recursive calls each on 20% of the array. The other work done in the function is a single for loop that runs through the input array one time and does one operation to each array element.