1. Consider two asymptotically positive functions $f_1(n)$ and $f_2(n)$. Assume that $f_1(n) = o(f_2(n))$, or equivalently, $f_2(n) = \omega(f_1(n))$. Compare the functions $f_1(f_2(n))$ and $f_2(f_1(n))$ asymptotically. Can you make a universal statement on which grows faster, or whether they are asymptotically equivalent? If not, then give examples of all the three possibilities. Generalise your examples by a single statement summarising the observations.

2. **Solution:**
   All three possibilities occur. For the case, when $f_1(f_2(n)) = \theta(f_2(f_1(n)))$, we simply take $f_1, f_2$ to be functions, which are inverses of each other.
   For example, $f_1(n) = \sqrt{n}$ and $f_2(n) = n^2$. Also possible is $f_1(n) = \log n$ and $f_2(n) = 2^n$.

   This immediately suggests, that if we take a function which grows faster or slower than the inverse, we will succeed in producing examples for the other two cases. $f_1(n) = n^2$ and $f_2(n) = 2^n$. Here, $f_1(f_2(n)) = (2^n)^2 = 2^{2n} = o(2^{n^2}) = \theta(f_2(f_1(n)))$.

   Take $f_1(n) = \log^* n$ and $f_2(n) = \log n$. Here, $f_1(f_2(n)) = \log^* \log n = 1 + \log^* n = \omega(\log \log^* n) = \theta(f_2(f_1(n)))$.

3. (a) Analyse the running time of your algorithm for computing a treap (in the earlier assignment) from a given set of input data, each with an ordered pair of key elements.
   (b) Try and modify the algorithm to improve the running time.
   (c) Also incorporate functionality to insert or delete records dynamically into the data-structure. Give efficient algorithms for these requirements and analyse the performance.
4. (a) Binary Search is an algorithm which takes as input an array of \( n \) elements, in sorted order and a value, and returns the index of the array whose key value matches the input value, if one exists. It works by comparing the input value with the element in the middle of the array, and if it is not a match, then if the key value is larger than the array entry, it searches in the right half of the array, otherwise it searches in the left half, recursively. Write the recurrence relation for the running time of this algorithm, and solve the recurrence.

(b) Now consider a 2-dimensional \( n \times n \) array. In each row the elements are in increasing order, and in each column also the elements are in increasing order. Develop an algorithm similar to binary search, and argue that it is correct. Write an expression for its running time as a recurrence relation and solve the recurrence to get an estimate on the time required for this search algorithm.

(c) Generalise this idea to a \( k \)-dimensional array, with size \( n \) on each dimension.

(d) How long does it take given a set of \( n \) numbers, in arbitrary (unsorted) order to place in these data-structures (2 or higher dimensional arrays) satisfying the requirements of relative values? Is this better or worse than sorting the elements into a long single dimensional array, asymptotically and in absolute terms?

5. Solution:

(a) The recurrence relation in this case is \( T(n) = T(\frac{n}{2}) + \theta(1) \). The solution is \( T(n) = \theta(\log n) \).

(b) We compare with the element right in the geometric centre of the array. That is we look at position \( M[\frac{n}{2}, \frac{n}{2}] \) and compare with the input key. If it is not a match, then if the key element is less than the array entry, we search all elements except those with both coordinates higher than the searched position. We thus zero in on a \( \frac{3}{4} \) fraction of the original size. This is the best we can do, because if we select any other element, then an adversary could always make us search an even larger space by giving the appropriate response. Thus, in the first iteration, we reduce the problem size to three-quarters of its original size. The shape
is, however, no longer rectangular, and finding a geometrically
centred element may be hard. Moreover, the splitting ratio could
grow, and also the shapes can get complicated in more iterations.
Thus, we solve three independent subproblems of size one-fourth
of the original problem each, as the element being sought could
be in any of these portions. Thus the recurrence for the running
time is $T(n) = 3T\left(\frac{n}{4}\right) + \theta(1)$. The running time is $T(n) = \theta(n \log 3)$
for an array of $n^2$ elements. This is a sublinear polynomial but
clearly bigger than the $\log(n^2)$ time of the standard binary search
algorithm.

(c) Applying the same idea, on the first step we reduce the original
problem of size $n$ to a $2^k - 1$ subproblems of size $\frac{n}{2}$ each. Thus,
$T(n) = (2^k - 1)T\left(\frac{n}{2}\right) + \theta(1)$. This yields a running time of $T(n) =
\theta(n \log(2^k - 1))$, for an array of $n^k$ elements.

(d) At the time of creating this array in the 2-dimensional case, we
need to sort $n$ lists of $n$ elements each. Then we only need to
sort one more list containing the first element of each of the pre-
viously sorted lists. Thus the sorting time is $(n + 1)n \log n$, which
is around $n^2 \log n$, which is a saving in the constant factors in
the logarithmic term, since we have $\log n$, as against the usual
$\log n^2$ term. So, we have an asymptotic loss in the search phase
but a huge gain in the constant factors of building up the data-
structure. This might make sense in a sensitive application, where
update operations are much more frequent than search oper-
ations. Surveillance data is typically like this, where lots of things
are recorded, but a search is made only in exceptional circum-
stances.

6. A majority element of a multiset of $m$ elements is any element which
occurs at least $\lceil \frac{m}{2} \rceil$ times. Does every multiset have a majority ele-
ment? How many majority elements can a multiset have? If a given
multiset has a majority element, show that it can be found in $O(m)$
time.

7. **Solution:**

No. For example, any multiset on at least 3 elements with all elements
distinct does not have a majority element. An even cardinality multiset
can have 0, 1 or 2 majority elements. An odd cardinality multiset can
have either 0 or 1 majority elements.
If the set has even cardinality, then scan it and keep track to see if there are at least three distinct elements. This can be done with one linear scan keeping two temporary variables to copy the first two distinct elements seen. If only two are seen, then either one or both of them are majority elements. This can be resolved with one more linear pass, or by keeping count on the first pass itself.

Except in the above case, whatever the parity of the cardinality of the set, there is at most one majority element. Scan the elements of the array from left to right, keeping the last element seen in a temporary variable. As long as the same element is being seen just scan ahead. The moment there is a distinct element, throw it out along with the last occurrence of the repeating element. If one of those two is a majority element of the original set, then it continues to be a majority element of the reduced set. Continue scanning from the last element before the one thrown out and the first element after the distinct element was found again keeping track for change of element. At the end of this pass we are left either with an empty array or exactly one candidate majority element. Scan the array keeping count of the number of occurrences of the candidate element and then give the answer yes, if and only if it occurs at least half the number of times as the size of the array.