5.1)

2.)a.) Algorithm MinMax( minval, maxval)
//Finds the values of the smallest and largest elements in a given subarray
//Input: A portion of array [0 − 1] between indices and ( ≤ )
//Output: The values of the smallest and largest elements in []
//assigned to minval and maxval, respectively
if =
minval ← []; maxval ← []
else if − = 1
if [] ≤ []
minval ← []; maxval ← []
else minval ← []; maxval ← []
else // − 1
MinMax ( b( + )2c minval, maxval)
MinMax( b( + )2c + 1 minval2, maxval2 )
if minval2 minval
minval ← minval2
if maxval2 maxval
maxval ← maxval2

b.) Given the assumption that the array size is a power of 2, denoted as n = 2^k, we can derive the recurrence relation for the number of comparisons, denoted as C(n), as follows:

C(n) = 2C(n/2) + 2 for n > 2, C(2) = 1, C(1) = 0.

Solving this recurrence relation using backward substitution for n = 2^k, where k ≥ 1, yields the following sequence:

C(2^k) = 2C(2^(k-1)) + 2
= 2[2C(2^(k-2)) + 2] + 2 = 2^2C(2^(k-2)) + 2^2 + 2
= 2^2[2C(2^(k-3)) + 2] + 2^2 + 2 = 2^3C(2^(k-3)) + 2^3 + 2 + 2
= ...
= 2^iC(2^(k-i)) + 2^i + 2^(i-1) + ... + 2

Continuing this process, we eventually reach:

= 2^(k-1)C(2) + 2^(k-1) + ... + 2 = 2^(k-1) + 2^k - 2 = 3/2 * n - 2.

c.)This algorithm conducts approximately 25% fewer comparisons – specifically, 1.5 times n as opposed to 2 times n in the brute-force method. It's worth noting that this reduction in comparisons is achieved by halting recursive calls when n equals 2; otherwise, this advantage would be lost. Indeed, in terms of comparison count, the algorithm is optimal. However, in

practice, it may not necessarily outperform the brute-force approach due to potential overhead associated with recursion.

4.) In the second scenario, where the solution's complexity is expressed as Θ(n^d log n), variations in the base of the logarithm would merely affect the function by a constant factor, rendering it insignificant. However, in the third case, where the solution's complexity is denoted as Θ(n log^β α), the logarithm is integral to the function's exponent. Therefore, it must be specified, as functions of n exhibit varying growth rates for different values of α.

5.) a. In equation a, the recurrence relation is given as $T(n) = 4T(n/2) + n$. Here, the constants are $\alpha = 4$, $\beta = 2$, and $\delta = 1$. As $\alpha > \beta\delta$, we determine that T(n) belongs to $\Theta(n \log_2 4)$, which simplifies to $\Theta(n^2)$.

b. In equation b, the recurrence relation is given as $T(n) = 4T(n/2) + n^2$. Here, the constants are $\alpha = 4$, $\beta = 2$, and $\delta = 2$. As $\alpha$ equals $\beta\delta$, T(n) is in $\Theta(n^2 \log n)$.

c. In equation c, the recurrence relation is given as $T(n) = 4T(n/2) + n^3$. Here, the constants are $\alpha = 4$, $\beta = 2$, and $\delta = 3$. As $\alpha < \beta\delta$, T(n) is in $\Theta(n^3)$.

5.2.)
8.) Algorithm NegBeforePos([0 − 1])
//Puts negative elements before positive (and zeros, if any) in an array
//Input: Array [0 − 1] of real numbers
//Output: Array [0 − 1] in which all its negative elements precede
nonnegative
← 0; ← − 1
while ≤ do // would suffice
if [] 0 //shrink the unknown section from the left
← + 1
else //shrink the unknown section from the right
swap([] [])
← − 1

5.3.)
2.) The algorithm is incorrect because it returns 0 for any binary tree; in particular, it returns 0 instead of 1 for the one-node binary tree.

Algorithm LeafCounter ()
//Computes recursively the number of leaves in a binary tree
//Input: A binary tree
//Output: The number of leaves in
if = ∅ return 0 //empty tree
else if = ∅ and = ∅ return 1 //one-node tree

else return LeafCounter ()+ LeafCounter () //general case

5.) a. Preorder: a b d e c f
b. Inorder: d b e a c f
c. Postorder: d e b f c a

5.4)
9.) The recurrence relation governing the number of multiplications in Pan's algorithm is expressed as M(n) = 143640M(n/70) for n > 1, with M(1) = 1. Solving it for n = 70k or applying the Master Theorem leads to M(n) belonging to Θ(n^π), where π is calculated as the logarithm base 70 of 143640 divided by the natural logarithm of 70, which is approximately 2.795. This value is slightly lower than the exponent in Strassen's algorithm, denoted as σ, calculated as the logarithm base 2 of 7, which is approximately 2.807.