2. Here are the pseudocodes for the given algorithms:

a. To check if a given number is Palindrome or not:

BEGIN

READ number to check as n

SET s = 0

SET t = n

WHILE n != 0 DO

r = n % 10

s = s \* 10 + r

n = n / 10

ENDWHILE

IF s == t THEN

DISPLAY n is palindrome

ELSE

DISPLAY n is not a palindrome

ENDIF

END

b. To find the sum of the digits of a number:

BEGIN

READ number as n

SET sum = 0

WHILE n > 0 DO

digit = n % 10

sum = sum + digit

n = n / 10

ENDWHILE

DISPLAY sum

END

c. Iterative Binary Search on an array of n numbers and a given element:

BEGIN

READ array of size n and element to search as A, x respectively.

SET low = 0, high = n - 1.

WHILE low <= high DO

SET mid = (low + high) / 2.

IF A[mid] == x THEN

RETURN mid.

ELSE IF A[mid] < x THEN

low = mid + 1.

ELSE

high = mid - 1.

ENDIF

ENDWHILE

RETURN -1.

END

d. To convert a given decimal number to the corresponding binary number:

BEGIN

READ decimal number as N.

SET B\_Number = 0.

SET count = 0.

WHILE N != 0 DO

SET rem = N % 2.

SET c = pow(10, cnt).

SET B\_Number += rem \* c.

N /= 2.

cnt += 1.

ENDWHILE

DISPLAY B\_Number.

END

e. To check if a number can be represented as the sum of two prime numbers or not:

BEGIN

READ number as N.

IF N < 2 THEN

DISPLAY "Invalid input".

STOP.

ENDIF

FOR i FROM 2 TO N/2 DO

IF isPrime(i) THEN

IF isPrime(N-i) THEN

DISPLAY "Yes".

STOP.

ENDIF

ENDIF

ENDFOR

DISPLAY "No".

END

FUNCTION isPrime(n: integer) -> boolean:

BEGIN

IF n <= 1 THEN RETURN false. ENDIF

FOR i FROM 2 TO sqrt(n) DO

IF n % i == 0 THEN RETURN false. ENDIF

ENDFOR

RETURN true.

END

Please note that these pseudocodes are for reference only and may require modifications based on the programming language used for implementation. Let me know if you need any further assistance!

3. Here are the answers to the given questions:

a. The order of growth of 𝑓(𝑛) is O(n^2). - The order of growth of 𝑔(𝑛) is O(n). - Therefore, 𝒇(𝒏) ≠ 𝑶(𝒈(𝒏)) and 𝒈(𝒏) ≠ 𝑶(𝒇(𝒏)).

b. The order of growth of 𝑓(𝑛) is O(n^(1/2)). - The order of growth of 𝑔(𝑛) is O(n^2). - Therefore, 𝒇(𝒏) ≠ 𝑶(𝒈(𝒏)) and 𝒈(𝒏) ≠ 𝑶(𝒇(𝒏)).

c. The order of growth of 𝑓(𝑛) is O(n). - The order of growth of 𝑔(𝑛) is O(n^(3/2)). - Therefore, g(n)=O(f(n)) and f(n)=O(g(n)).

d. The order of growth of f(n) is O(n log n). - The order of growth of g(n) is O(n^(3/2)). - Therefore, f(n)=O(g(n)).

e. The order of growth of f(n) is O(log^2 n). - The order of growth of g(n) is O(log n). - Therefore, f(n)=O(g(n)).

4. Here are the answers to the given questions:

a. 2𝑛^2 + 1 = 𝑂(𝑛^2) - The highest order term in 2𝑛^2 + 1 is 2𝑛^2. - Therefore, 2𝑛^2 + 1 = 𝑂(𝑛^2). - The statement is **TRUE**.

b. 𝑛^2(1 + √𝑛) = 𝑂(𝑛^2) - The highest order term in 𝑛^2(1 + √𝑛) is 𝑛^2. - Therefore, 𝑛^2(1 + √𝑛) = 𝑂(𝑛^2). - The statement is **TRUE**.

c. 𝑛^2(1 + √𝑛) = 𝑂(𝑛^2 log 𝑛) - The highest order term in 𝑛^2(1 + √𝑛) is 𝑛^2. - The highest order term in log n is log n. - Therefore, 𝑛^2(1 + √𝑛) ≠ 𝑂(𝑛^2 log 𝑛). - The statement is **FALSE**.

d. 3𝑛^2 + √𝑛 = 𝑂(𝑛 + 𝑛√𝑛 + √𝑛) - The highest order term in 3𝑛^2 is 𝑛^2. - The highest order term in 𝑛√𝑛 is 𝑛^(3/2). - Therefore, 3𝑛^2 + √𝑛 = 𝑜(𝑛^(3/2)). - The statement is **FALSE**.

e. √𝑛 log 𝑛 = 𝑜(𝑛) - The highest order term in √n log n is n^(1/2) \* log n. - Therefore, √n log n ≠ O(n). - The statement is **FALSE**.

f. 𝜇(n) ∈ O(n) - Since the function μ(n) takes values between −1 and 1, it can be bounded by a constant function. - Therefore, μ(n) ∈ O(1). - The statement is **TRUE**.

g. n ∈ O(n log n) - Since n ≤ n log n for all n > 0, we can say that n ∈ O(n log n). - The statement is **TRUE**.

h. n log n ∈ O(n^2) - Since n log n ≤ n \* n for all n > 0, we can say that n log n ∈ O(n^2). - Therefore, n log n ∈ O(n^2). - The statement is **TRUE**.

i. 2^n ∈ Ω(6^(logn)) - We can write 6^(logn) as (e(log6))logn = e^(log6 \* logn). - We can write 2^n as e^(n \* log 2). - Therefore, we need to show that there exist positive constants c and k such that: e^(n \* log 2) ≥ c \* e^(log6 \* logn) => e^(n \* (log 2 / log6)) ≥ c \* e^(logn) => e^(n \* (log 6 / log 2)) ≤ c => (6/4)^n ≤ c => c >= (3/4)^n We can choose c = (3/4)^n and k = 0 to satisfy the above condition. Therefore, we can say that 2^n ∈ Ω(6^(logn)). The statement is **TRUE**.

j. lg^3(n) ∈ o(sqrt(n))

* We know that lg(n) < sqrt(n) for all sufficiently large values of n.
* Therefore, lg^3(n) < sqrt(n)^3 for all sufficiently large values of n.
* Hence, lg^3(n) ∈ o(sqrt(n)).
* The statement is **TRUE**.

5. Here are the answers to the given questions:

a. 𝑓(𝑛) = √𝑛, 𝑔(𝑛) = log(𝑛 + 3) - The highest order term in 𝑓(𝑛) is √n. - The highest order term in 𝑔(𝑛) is log(n). - Therefore, 𝑓(𝑛) = 𝚯(𝒈(𝒏)).

b. 𝑓(𝑛) = 𝑛√𝑛, 𝑔(𝑛) = 𝑛^2 − 𝑛 - The highest order term in 𝑓(𝑛) is n^(3/2). - The highest order term in 𝑔(𝑛) is n^2. - Therefore, 𝒇(𝒏) = 𝚯(𝒈(𝒏)).

c. 𝑓(𝑛) = 2𝑛 − 𝑛^2, 𝑔(𝑛) = 𝑛^4 + 𝑛^2 - The highest order term in 𝑓(𝑛) is −n^2. - The highest order term in 𝑔(𝑛) is n^4. - Therefore, 𝒇(𝒏) = 𝚯(𝒈(𝒏)).

d. 𝑓(𝑛) = 𝑛^2 + 3𝑛 + 4, 𝑔(𝑛) = 6𝑛^2 - The highest order term in 𝑓(𝑛) is n^2. - The highest order term in 𝑔(𝑛) is n^2. - Therefore, 𝒇(𝒏) = Θ(g(n)).

e. 𝑓(𝑛) = 𝑛 + 𝑛√𝑛, 𝑔(𝑛) = 4𝑛 log(𝑛^2 + 1) - The highest order term in f(n) is n^(3/2). - The highest order term in g(n) is n log(n^2+1). - Therefore, f(n)=Ω(g(n)).

6. Here are the answers to the given questions:

a. Let c1 and c2 be positive constants such that f1(n) >= c1 \* g1(n) and f2(n) >= c2 \* g2(n). - Then, f1(n) + f2(n) >= c1 \* g1(n) + c2 \* g2(n). - Therefore, f1(n) + f2(n) = Ω(g1(n) + g2(n)). - The statement is **TRUE**.

b. If 𝑓1(𝑛) = Ω(𝑔1(𝑛)) and 𝑓2(𝑛) = Ω(𝑔2(𝑛)) then 𝑓1𝑛) + 𝑓2(𝑛) = Ω(min(𝑔1(𝑛), 𝑔2(𝑛))) - Let c1 and c2 be positive constants such that f1(n) >= c1 \* g1(n) and f2(n) >= c2 \* g2(n). - Then, f1(n) + f2(n) >= (c1 + c2) \* min(g1(n), g2(n)). - Therefore, f1(n) + f2(n) = Ω(min(g1(n), g2(n))). - The statement is **TRUE**.

c. If 𝑓1(𝑛) = O(𝑔1(𝑛)) and 𝑓2(𝑛) = O(𝑔2(𝑛)) then 𝑓1(𝑛) ∙ 𝑓2(𝑛) = O(𝑔1(𝑛) ∙ 𝑔2(𝑛)) - Let c1 and c2 be positive constants such that f1(n) <= c1 \* g1(n) and f2(n) <= c2 \* g2(n). - Then, f1(n)*f2(n)<=c1*c2\*g1(n)\*g2(n). - Therefore, f1(n)\*f2(n)=O(g1(n)\*g2(n)). - The statement is **TRUE**.

d. If 𝑓(𝑛) = O(𝑔(𝑛)), then 𝑓(𝑛)^k = O(g(n)k) - Let c be a positive constant such that f(n)<=c\*g(n). - Then, (f(n))k<=(c\*g(n))k=ck\*(g(n))^k. - Therefore, (f(n))k=O(g(n)k). - The statement is **TRUE**.

e. If 𝒇\_1 (n)=Θ(g\_1 (n)), 𝒇\_2 (n)=Θ(g\_2 (n)), … , 𝒇\_k (n)=Θ(g\_k (n)), where k∈I+, then prove that ∑\_(i=0)^k▒〖f\_i (n)k=Θ(max\_(j=0)k▒〖g\_j (n)^k〗〗

* Let cj be positive constants such that 0<=cj<=infinity.
* Then, for each i=0 to k, there exist positive constants ci and di such that ci*g\_i<=f\_i<=di*g\_i.
* Therefore, cik\*g\_ik<=f\_ik<=dik\*g\_i^k.

7. Let’s prove that 𝑓(𝑛) ∈ 𝜃(𝑛^𝑏).

We know that (n+a)^b = n^b \* (1 + a/n)^b.

Now, let’s consider the limit of (1 + a/n)^b as n approaches infinity.

Using L’Hopital’s rule, we can rewrite the limit as e^(b \* ln(1 + a/n)).

Since ln(1 + x) is approximately equal to x for small values of x, we can rewrite the limit as e^(b \* a/n).

Therefore, as n approaches infinity, (n+a)^b is approximately equal to n^b \* e^(b \* a/n).

Since e^(b \* a/n) approaches 1 as n approaches infinity, we can say that (n+a)^b is in the same order of growth as n^b.

Therefore, 𝑓(𝑛) ∈ 𝜃(𝑛^𝑏).

Hence proved!

8. Let’s prove that 𝑙g 𝑛 = 𝑂(√𝑛).

We know that the square root function grows slower than the logarithmic function. Therefore, we can say that √n is upper-bounded by log n.

Now, let’s consider the limit of log n / √n as n approaches infinity.

Using L’Hopital’s rule, we can rewrite the limit as ∞ / ∞.

By applying the rule again, we get the limit as 0.

Therefore, log n is in the same order of growth as √n.

Hence, 𝑙g 𝑛 = 𝑂(√𝑛).

Now, let’s prove that √n ≠ 𝑂(𝑙g 𝑛).

Assume that √n = O(log n).

Then, there exist positive constants c and k such that √n <= c \* log n for all n > k.

Squaring both sides, we get n <= c^2 \* log^2 n for all n > k.

This implies that n grows slower than log^2 n for all n > k.

However, this is not true because there exists a value of n such that n > k and log^2 n < n.

Therefore, our assumption is false and √n ≠ O(log n).

Hence proved!

9. Given A [1 … 60] = {10, 11, … , 70}, we can perform a binary search to find the elements 10, 40, 50, and 70.

The binary search algorithm works by dividing the search interval in half at each step.

Let’s consider the case of searching for the element 10.

The initial search interval is [1, 60].

The middle element of the interval is A[30] = 40. Since 10 < 40, we can eliminate the right half of the interval and continue searching in the left half [1, 29].

The middle element of this interval is A[15] = 25. Since 10 < 25, we can eliminate the right half of the interval and continue searching in the left half [1, 14].

The middle element of this interval is A[8] = 18. Since 10 < 18, we can eliminate the right half of the interval and continue searching in the left half [1, 7].

The middle element of this interval is A[4](https://cs.stackexchange.com/questions/83024/number-of-comparisons-in-binary-search) = 14. Since A[4](https://cs.stackexchange.com/questions/83024/number-of-comparisons-in-binary-search) = 10, we have found the element we were looking for.

Therefore, a total of **4 comparisons** were performed to find the element 10.

Similarly, for finding elements 40 and 70, a total of **4 comparisons** will be performed in each case.

For finding element 50, a total of **5 comparisons** will be performed.

Here is a recursion tree for finding element 50:

[1,60]

/ \

[1,29] [31,60]

/ \

[1,14] [16,29]

/ \

[1,7] [9,14]

/ \

[5,7] [10,14]

|

[10]

As we can see from the recursion tree above, a total of **5 comparisons** were performed to find element 50.

The stack space used by the binary search algorithm is proportional to the height of the recursion tree. In this case, since all elements are present in the array A and no element needs to be inserted or deleted from it during search operation, all recursive calls will be made on intervals that are either left or right halves of their parent intervals. Therefore, at any point in time during execution of binary search algorithm for this problem statement there will be at most log2(60) + 1 = **7 recursive calls** on stack.

10. Algorithm A solves the problem P by dividing it into five sub-problems of half the size, recursively solving each sub-problem and then combining the solution in linear time. The time complexity of Algorithm A can be expressed as T(n) = 5T(n/2) + O(n). Using the Master Theorem, we can see that the time complexity of Algorithm A is O(n^2.32).

Algorithm B solves the problem P by recursively solving two sub-problems of size n-1 and then combining the solutions in constant time. The time complexity of Algorithm B can be expressed as T(n) = 2T(n-1) + O(1). Using the Master Theorem, we can see that the time complexity of Algorithm B is O(2^n).

Algorithm C solves the problem P by dividing it into nine sub-problems each of size n/3, recursively solving each sub-problem and then combining the solutions in O(n^2) times. The time complexity of Algorithm C can be expressed as T(n) = 9T(n/3) + O(n^2). Using the Master Theorem, we can see that the time complexity of Algorithm C is O(n^2.81).

Therefore, Algorithm B has the worst time complexity among all three algorithms. Hence, I would choose either Algorithm A or Algorithm C depending on other factors such as space complexity, ease of implementation, etc.

11. The given algorithm is a recursive function that divides the input size n into three sub-problems of size n/3 and then recursively solves each sub-problem. The function then performs a loop that runs n times.

Let T(n) be the time complexity of the function.

The time complexity of the recursive part of the function can be expressed as T(n/3) + T(n/3) + T(n/3) = 3T(n/3).

The time complexity of the loop is O(n).

Therefore, the time complexity of the function can be expressed as T(n) = 3T(n/3) + O(n).

Using the Master Theorem, we can see that the time complexity of the function is O(n log n).

The space complexity of the function is proportional to the height of the recursion tree. Since each node in the recursion tree has three children, the height of the tree is log3n. Therefore, the space complexity of the function is O(log n).

Hence, the time complexity of the given algorithm is O(n log n) and its space complexity is O(log n).

12. The given algorithm is a loop that runs n times and divides n by 2 at each iteration until n becomes less than or equal to 1.

Let T(n) be the time complexity of the function.

The loop runs n times, so the time complexity of the loop is O(n).

The value of n is divided by 2 at each iteration, so the number of iterations required to reach a value less than or equal to 1 is log2n.

Therefore, the time complexity of the function can be expressed as T(n) = O(n log n).

The space complexity of the function is proportional to the number of variables used in the function. In this case, only one variable Temp is used. Therefore, the space complexity of the function is O(1).

Hence, the time complexity of the given algorithm is O(n log n) and its space complexity is O(1).

13. The given algorithm is a recursive function that divides the input size n into sub-problems of size floor(sqrt(n)) and then recursively solves each sub-problem. The function then adds 1 to the result of the recursive call.

Let T(n) be the time complexity of the function.

The time complexity of the recursive part of the function can be expressed as T(sqrt(n)).

Therefore, the time complexity of the function can be expressed as T(n) = T(sqrt(n)) + O(1).

Using the Master Theorem, we can see that the time complexity of the function is O(log log n).

The space complexity of the function is proportional to the height of the recursion tree. Since each node in the recursion tree has one child, the height of the tree is log2logn. Therefore, the space complexity of the function is O(log log n).

Hence, the time complexity of the given algorithm is O(log log n) and its space complexity is O(log log n).

14. The given algorithm is a recursive function that divides the input size n by 2 at each iteration until n becomes 1. The function then performs a loop that runs n^3 times.

Let T(n) be the time complexity of the function.

The time complexity of the recursive part of the function can be expressed as T(n/2).

Therefore, the time complexity of the function can be expressed as T(n) = T(n/2) + O(n^3).

Using the Master Theorem, we can see that the time complexity of the function is O(n^3).

The space complexity of the function is proportional to the height of the recursion tree. Since each node in the recursion tree has one child, the height of the tree is log2n. Therefore, the space complexity of the function is O(log n).

Hence, the time complexity of the given algorithm is O(n^3) and its space complexity is O(log n).

15. Here is the recursive version of the LinearSearch algorithm:

int LinearSearchRecursive(int A[], int n, int el, int i) {

if (i > n) {

return -1;

}

if (A[i] == el) {

return i;

}

return LinearSearchRecursive(A, n, el, i + 1);

}

The recurrence relation for the time complexity function can be expressed as T(n) = T(n-1) + O(1).

Using the Master Theorem, we can see that the time complexity of the recursive version of LinearSearch algorithm is O(n).

The space complexity of the recursive version of LinearSearch algorithm is proportional to the height of the recursion tree. Since each node in the recursion tree has one child, the height of the tree is n. Therefore, the space complexity of the recursive version of LinearSearch algorithm is O(n).

Therefore, we can see that both iterative and recursive versions of LinearSearch algorithm have same time complexity but different space complexities. The iterative version has a space complexity of O(1), while the recursive version has a space complexity of O(n).

16. a. The given recurrence relation is:

T(n) = 1 if n = 2; 1 if n = 4; T(n/2) + 2T(n/4) + theta(n^2) if n > 4

We can solve this recurrence relation using the Master Theorem.

The recurrence relation can be expressed as:

a = 1, b = 2, d = 2

Since logb a = log2 1 = 0 < d, the recurrence relation falls under Case 1 of the Master Theorem.

Therefore, the solution to the recurrence relation is T(n) = theta(n^log2 2) = theta(n).

Hence, the time complexity of the given recurrence relation is O(n).

Since there is no recursive call in the function, the space complexity of the function is O(1).

Therefore, the time complexity of the given algorithm is O(n) and its space complexity is O(1).

b. The given recurrence relation is:

T(n) = n^(1/3) \* T(n^(2/3)) + O(n)

We can solve this recurrence relation using the Master Theorem.

The recurrence relation can be expressed as:

a = 1, b = n^(2/3), d = 1/3

Since logb a > d, the recurrence relation falls under Case 3 of the Master Theorem.

Therefore, the solution to the recurrence relation is T(n) = theta(n^(log3 2)).

Hence, the time complexity of the given algorithm is O(n^(log3 2)) and its space complexity is O(log n).

c. The given recurrence relation is:

T(n) = sqrt(n) \* T(sqrt(n)) + log n

We can solve this recurrence relation using the Master Theorem.

The recurrence relation can be expressed as:

a = sqrt(n), b = n^(1/2), d = 1

Since logb a < d, the recurrence relation falls under Case 1 of the Master Theorem.

Therefore, the solution to the recurrence relation is T(n) = theta(sqrt(n) \* log n).

Hence, the time complexity of the given algorithm is O(sqrt(n) \* log n) and its space complexity is O(log n).

d.