

**NAME: SAJEEL AFSAR**

**ROLL no: 14798**

**SUBJECT: DSA**

**CLASS: BS CS 3D**

**DATE: 25/10/2024**

**:ASSIGEMENT:**

**Chapter 1**

**Exsercie Question & ANSWERwer**

* + 1. **Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.**

**ANSWER:**

**Example Requiring Sorting:**

**Library Book Organization:**

In a library, books are sorted by title, author, or genre. This helps customer find books easily and keeps the library organized. Example Requiring Shortest Distance: Navigation for Driving: When using a GPS app, it finds the shortest route from your current location to your destination. This helps you get there faster by avoiding traffic and taking the best roads.

* + 1. **Other than speed, what other measures of efûciency might you need to consider in a real-world setting**

ANSWER:

In a real-world setting, other than speed, you might consider:

1. **Resource Usage:**

How much memory or power the system uses.

**2. Scalability:**

How well the system handles more users or data.

**3.Cost:**

The financial cost of running the system.

**4. Reliability:**

How often the system fails or crashes.

**5. Maintainability:**

How easy it is to update or fix the system.

* + 1. **Select a data structure that you have seen, and discuss its strengths and limitations.**

ANSWER:

**Data Structure:**

**Array Strengths:**

1. **Fast Access:**

You can quickly access any item using its index (O(1) time).

1. **Memory Efficiency:**

Uses less memory overall since it stores items in adjacent memory.

1. **Simple to Use:**

Easy to understand and implement.

**Limitations:**

1. **Fixed Size:**

Once created, the size cannot change. You need to know the size in advance.

1. **Slow Insertions/Deletions:**

Adding or removing items (especially in the middle) can be slow, as it requires shifting elements (O(n) time).

1. **Memory Waste:**

If the array is too large for the data it holds, memory can be wasted.

**1.1-4 How are the shortest-path and traveling-salesperson problems given above similar? How are they different.**

ANSWER:

**Similarities:**

• Both problems involve finding the best route through a set of points (like cities or locations).

• They aim to minimize the total distance or cost of traveling.

**Differences:**

• Shortest Path Problem: Finds the shortest route between two specific points (e.g., from point A to point B).

• Traveling Salesperson Problem (TSP): Requires visiting every point exactly once and returning to the starting point (e.g., visiting multiple cities in one trip).

* + 1. **Suggest a real-world problem in which only the best solution will do. Then come up with one in which approximately the best solution is good enough.**

**ANSWER:**

Best Solution Required:

Sorting TrANSWERactions in Banking:

In a bank, it's really important to sort trANSWERactions correctly by date or amount when preparing reports. If the sorting goes wrong, it can cause mistakes with money, lead to audits, or even legal problems. So, getting the sorting right (the best solution) is very important to ensure everything is accurate and trustworthy.

**Approximately Best Solution is Good Enough:**

**Image Compression for Websites:**

When building a website, you want to make images smaller so they load faster. You don’t need to get the perfect file size; you just need the images to look good enough. As long as the quality of the images is okay for visitors, a close-to-optimal solution works well. This way, you balance loading speed and image quality without stressing over finding the perfect size.

* + 1. **Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.**

**ANSWER:**

Complete input Available:

Sometimes, meteorologists can collect all the information they need ahead of time. This includes things like temperature, humidity, wind speed, and air pressure from weather stations and s atellites. With all this data, they can make a detailed weather forecast for the week. input Comes in Over Time:

Other times, especially when the weather is changing quickly, newinformation comes in during the forecast period. For instance, real-time radar can show sudden storms or shifts in wind. Meteorologists then have to update their forecasts immediately as this new data arrives.

**1.2-1 Give an example of an application that requires algorithmic content at the Application level, and discuss the function of the algorithms involved.**

**ANSWER:**

**Example Application:**

**Online Shopping Platform**

1. **Product Search:**

When you search in an online store, algorithms quickly find the best matches from many products by using your search words, categories, and preferences. This helps you find what you want faster and easier.

1. **Recommendation System:**

Algorithms analyze your browsing and purchase history to suggest products you might like. For example, if you buy a book, the algorithm may recommend other books in the same genre.

1. **Inventory Management:**

Algorithms help track stock levels, predict demand, and automatically reorder items when they run low, ensuring that popular products are always available.

**1.2-2 Suppose that for inputs of size n on a particular computer, insertion sort runs in 8n2 steps and merge sort runs in 64 n log n steps. For which values of n does insertion sort beat merge sort?**

**ANSWER:**

• Insertion Sort:8n^2.

• Merge Sort: 64 n log n.

We need to find the values of n for which:

8n^2<64n log n

Step 1: Simplify the Inequality

Dividing both sides by 8 gives us

n^2 < 8 n log n

Step 2: Rearrange the Inequality

n^2/n log n< 8

n/ log n < 8

Step 3: Solve the Inequality

Now, we need to find values of n that satisfy this inequality.

Testing Values

Let's test some small values of n:

For n=2:

2/log (2)=2.89<8(true)

For n=3:

3/log (3) =2.73<8 (true)

For n=16

16/log (16) =3.848<8 (true)

For n=32:

32/log (32) = 9.23>8 (false)

When n=32 the insertion beat merge on this value

**1.2-3 What is the smallest value of n such that an algorithm whose running time is 100n2 runs faster than an algorithm whose running time is 2 n on the same machine?**

**ANSWER:**

To find the smallest value of n such that an algorithm with a running time of **100n^2** runs faster than an algorithm with a running time of **2^n** we need to solve the following inequality:

**100n^2<2^n**

Let's test small values of n to find when **100n**^2 becomes less than **2^n:**

For n=1:

100(1^2) =200 or (2^1) =2

200>2.

For n=9:

100(9^2) = 8100 or (2^9) =512

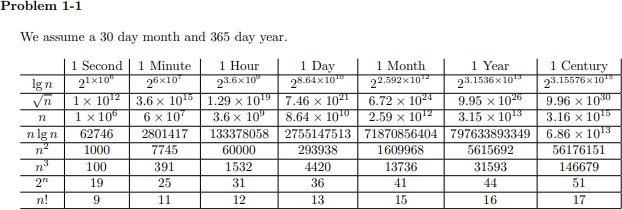
8100>5121.

For n=15:

100(15^2) = 22500 and (2^15) =32768

22500<32768

When n=15 then 100n^2 is become less than 2^n.



### 2.1-1

**Using Figure 2.2 as a model, illustrate the operation of I NSERTION-SORT on an array initially containingthe sequence 31; 41; 59; 26; 41; 58.**

**ANSWER:**

Step-by-Step Insertion Sort:

First Iteration (i = 1):

Current element: 41

Compare 41 with 31. Since 41 is greater than 31, it remains in place. Array after this step: [31, 41, 59, 26, 41, 58]

Second Iteration (i = 2):

Current element: 59

Compare 59 with 41. Since 59 is greater than 41, it remains in place. Array after this step: [31, 41, 59, 26, 41, 58]

Third Iteration (i = 3):

Current element: 26

Compare 26 with 59. Since 26 is smaller, move 59 to the right.

Compare 26 with 41. Since 26 is smaller, move 41 to the right.

Compare 26 with 31. Since 26 is smaller, move 31 to the right. Insert 26 in the correct position (at the start).

Array after this step: [26, 31, 41, 59, 41, 58] Fourth Iteration (i = 4):

Current element: 41

Compare 41 with 59. Since 41 is smaller, move 59 to the right. Compare 41 with 41. Since it's equal, leave it in place.

Array after this step: [26, 31, 41, 41, 59, 58] Fifth Iteration (i = 5):

Current element: 58

Compare 58 with 59. Since 58 is smaller, move 59 to the right. Compare 58 with 41. Since 58 is greater, leave it in place.

Array after this step: [26, 31, 41, 41, 58, 59]

**2.1-2 Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array A [1: n]. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in A [1: n]**

**ANSWER:**

A **loop invariant** for SUM-ARRAY is that before each iteration **i, sum** holds the sum of the first i−1 elements of the array.

* + **Initialization:** Before the first iteration, **sum = 0**, which is the sum of zero elements.
  + **Maintenance:** In each iteration, sum is updated by adding **A[i]** so after each iteration, sum

holds the sum of the first **i** elements.

* + **Termination:** After **n** iterations, **sum** contains the sum of all **n** elements, and the procedure returns the correct sum.

**2.1-3 Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.**

**ANSWER:**

**DECREASING-INSERTION-SORT(A)**

**for i = 1 to length(A) - 1 key = A[i]**

**j = i - 1**

**while j >= 0 and A[j] < key A[j + 1] = A[j]**

**j = j - 1**

**A[j + 1] = key**

**2.1-4 Consider the searching problem: Input: A sequence of n numbers (a1; a2; : : : ; an) stored in array A[1:n] and a value x. Output: An index i such that x equals A[i] or the special value NIL if x does not appear in A.**

**Write pseudocode for linear search, which scANSWER through the array from beginning to end, looking for**

1. **Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.**

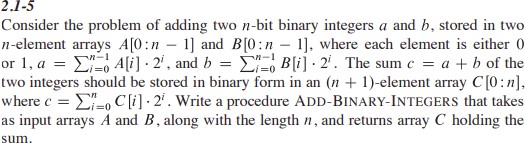
**ANSWER:**

LINEAR-SEARCH (A, n, x)

for i = 1 to n if A[i] == x return i return NIL

Explanation:

* + Start at the first element of the array.
  + Compare each element **A[i]** with the value **x.**
  + **If A[i]=X** return the index **i**.
  + If the loop finishes without finding x, return NIL.



**ANSWER:**

**ADD-BINARY (A, B, n)**

**C [0] = 0 // Initialize carry for i = 0 to n-1**

**sum = A[i] + B[i] + C[i] // Sum of bits and carry C [i + 1] = sum mod 2 // Store result bit**

**C[i] = sum div 2 // Update carry**

**return C**

**2.2-1 Express the function n^3/1000 +100n^2 - 100n + 3 in terms of ‚-notation.**

**ANSWER:**

To express the function in Big-O (‚-notation) in an easier way, follow these simple steps: Given function:

f(n)=n^3/1000+100n^2−100n+3

Steps:

### Find the biggest term:

* + **n^3** is the biggest term because it grows faster than **n^2, n** and any constant.

### Ignore the constants:

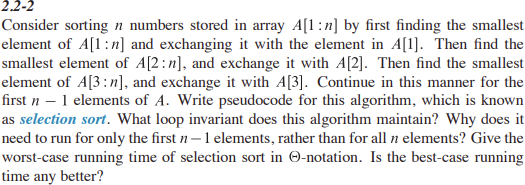
* + Ignore the constants like **1/1000,100 and -100**They don’t matter for **Big-O**, only the biggest term does.

### Write the Big-O notation:

* + The biggest term **is n^3**, so the function **is O (n^3).**

Final ANSWERwer:

O(n3)



**ANSWER:**

Selection Sort Algorithm

Selection sort repeatedly selects the smallest element from the unsorted portion of the array and swaps it with the element at the beginning of the unsorted portion.

**SELECTION-SORT(A)**

**for i = 1 to n - 1 min\_index = i for j = i + 1 to n**

**if A[j] < A[min\_index] min\_index = j**

**exchange A[i] with A[min\_index]**

Worst-Case Running Time:

The worst-case time complexity of selection sort is O(n^2), where:

* + The outer loop runs n−1n−1 times.
  + The inner loop runs for a decreasing number of elements, specifically n−1, n−2 ..., 1 comparison.
  + The total number of comparisons is ∑i=n−1=2n(n−1), which is O(n2)O(n^2)O(n2).

Best-Case Running Time:

The best-case running time is also O(n^2). Unlike some sorting algorithms (like insertion sort), selection sort performs the same number of comparisons regardless of the input order. Hence, even if the array is already sorted, it still has to perform the inner loop for every element, resulting in O(n^2) comparisons.

**2.2-3**

Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case

Using ‚-notation, give the average-case and worst-case running times of linear search. Justify your ANSWERwers

**ANSWER:**

**LINEAR-SEARCH (A, n, x)**

**for i = 1 to n if A[i] == x return i return NIL**

Worst-Case Analysis:

In the worst case, the element is either at the last position or is not in the array at all. In both cases, the search will have to check all n elements.

Thus, in the worst case, **n elements are checked**.

* + In O-notation, the worst-case running time is als O(n).

Average-Case Analysis:

If we assume that the element we are searching for is equally likely to be any element in the array, the number of comparisons depends on where the target is located:

* + If the target is at index 1, only 1 comparison is needed.
  + If the target is at index 2, 2 comparisons are needed, and so on.
  + If the target is at index nnn (the last element), nnn comparisons are needed.

On average, the position of the target will be somewhere in the middle of the array. Thus, the average number of comparisons is the mean of the sequence 1,2,3,…,n.

### 2.2-4 How can you modify any sorting algorithm to have a good best-case running time. ANSWER:

To improve the best-case running time of any sorting algorithm, **check if the array is already sorted** before running the algorithm. If it’s already sorted, you can skip the sorting process entirely, reducing the best-case running time to O(n).

**2.3-1 Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence h3; 41; 52; 26; 38; 57; 9; 49.**

### ANSWER:

1. **Step 1: Split the array** into two halves:

o Left half: [3, 41, 52, 26]

o Right half: [38, 57, 9, 49]

1. **Step 2: Keep splitting** each half until every part has only one element:

o Left half [3, 41, 52, 26] becomes [3], [41], [52], [26]

o Right half [38, 57, 9, 49] becomes [38], [57], [9], [49]

1. **Step 3: Merge the parts** back together while sorting them:
   * [3] and [41] merge to become [3, 41]
   * [52] and [26] merge to become [26, 52]

o Now, [3, 41] and [26, 52] merge to become [3, 26, 41, 52]

* + Similarly, merge [38] and [57] to become [38, 57]
  + Merge [9] and [49] to become [9, 49]

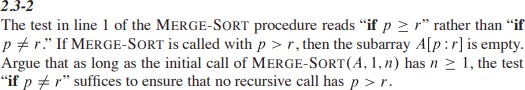
o Then, merge [38, 57] and [9, 49] to become [9, 38, 49, 57]

### Step 4: Final merge:

* + Merge [3, 26, 41, 52] with [9, 38, 49, 57] to get the sorted array:

[3,9,26,38,41,49,52,57]

**Result**: The sorted array is [3, 9, 26, 38, 41, 49, 52, 57].



**ANSWER:**

In the **MERGE-SORT** algorithm, the condition if p ≠ r is enough because:

1. **Splitting**: The algorithm splits the array into two parts, ensuring p is never greater than r

in recursive calls.

1. **Stopping**: When p = r, it meANSWER there’s only one element left, so recursion stops.
2. **Initial Call**: If MERGE-SORT(A, 1, n) has n ≥ 1, then all recursive calls will also satisfy

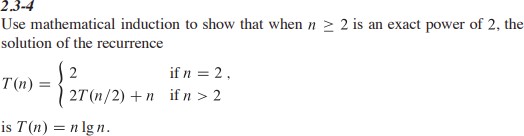
p ≤ r.

**Conclusion**: The check if p ≠ r ensures recursion stops correctly, and the splitting method prevents any invalid calls where p > r.

**2.3-3 State a loop invariant for the while loop of lines 12-18 of the MERGE procedure. Show how to use it, along with the while loops of lines 20-23 and 24-27, to prove that the MERGE procedure is correct.**

**ANSWER:**

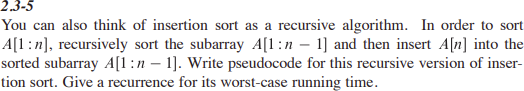
The loop invariant for the **MERGE** procedure states that at the start of each iteration, the elements merged into A from subarrays L and R are sorted, and all elements in A[p...k-1] are in sorted order. Initially, no elements have been merged, so the invariant holds. During each iteration, we compare the current elements from L and R, adding the smaller one to A and moving the corresponding index forward, which keeps the merged portion sorted. The loop continues until one subarray is fully merged, allowing any remaining elements from the other sorted subarray to be added directly to A. Thus, the **MERGE** procedure correctly combines two sorted subarrays into one sorted array.



**ANSWER:**

Since n is a power of two, we may write n = 2k . If k = 1, T(2) = 2 = 2 log(2). Suppose it is true for k, we will show it is true for k + 1. T(2k+1) = 2T (2^ k+1/ 2 )+ 2k+1 = (2T) 2^ k + 2^k+1 = 2(2^3 log(2^k )) + 2^k+1

= k2^ k+1 + 2^k+1 = (k + 1)2^k+1 = 2^k+1 log(2^k+1) = n log(n)



**ANSWER:**

INSERTION-SORT(A, n)

1. if n ≤ 1
2. return
3. INSERTION-SORT(A, n - 1) // Sort the first n - 1 elements
4. INSERT(A, n) // Insert the nth element into the sorted subarray

INSERT(A, n)

1. key = A[n] // The element to be inserted 2. i = n - 1

1. while i > 0 and A[i] > key
2. A[i + 1] = A[i] // Shift elements to the right 5. i = i - 1

6. A[i + 1] = key // Place the key in the correct position

Recurrence for Worst-Case Running Time

Let T(n) be the worst-case running time of the recursive insertion sort for an array of size nnn. The recurrence relation can be expressed as follows:

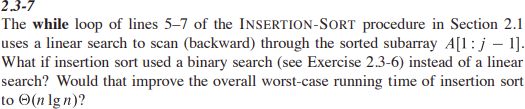
**T(n)= {c if n=1, T(n−1) +O(n) if n>1T(n)**

**2.3-6**

**Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is big O log n.**

**ANSWER:**

A binary search wouldn’t improve the worst-case running time. Insertion sort has to copy each element greater than key into its neighboring spot in the array. Doing a binary search would tell us how many how many elements need to be copied over, but wouldn’t rid us of the copying needed to be done.



### ANSWER:

Using binary search in insertion sort speeds up finding the insertion point from O(n) to O(log n). However, the time to shift elements remains O(n)

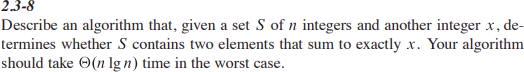
Thus, the overall complexity is:

T(n)=O (n log n) +O(n)

=O(n^2)

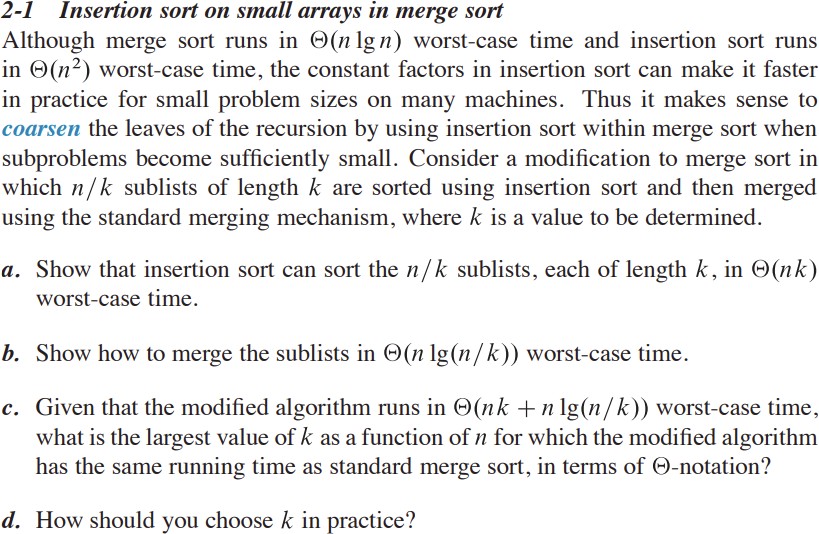
Conclusion

### So, insertion sort still has a worst-case time complexity O(n^2), even with binary search.



**ANSWER:**

1. FUNCTION find\_pair\_with\_sum(S, n, x)
2. SORT(S) // Sort the set S in O(n log n) time
3. left = 0 // Initialize left pointer
4. right = n - 1 // Initialize right pointer
5. WHILE left < right DO
6. current\_sum = S[left] + S[right] // Calculate current sum
7. IF current\_sum == x THEN
8. RETURN TRUE // Pair found
9. ELSE IF current\_sum < x THEN
10. left = left + 1 // Move left pointer to the right
11. ELSE
12. right = right - 1 // Move right pointer to the left
13. RETURN FALSE // No pair found



**ANSWER (a):**

a. The time for insertion sort to sort a single list of length k is Θ (k ^2), so, n/k of them will take time Θ( n /k k ^2 ) = Θ(nk).

**ANSWER(b):**

Here’s a simpler and shorter explanation:

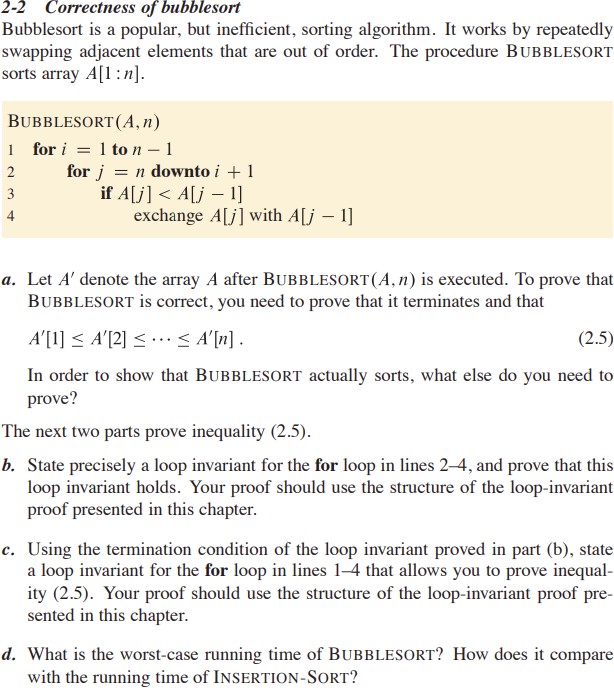
1. **Coarseness k**: We can start merging arrays when they are at most size k.
2. **Merge Depth**: The depth of the merge process is log2(n/k). This tells us how many levels we need to merge to get one sorted array.
3. **Time per Level**: Each level of merging takes O(n)time because we have to look at all elements.
4. **Total Time**: So, the total merging time is: Θ(log(n/k))

**ANSWER(c):**

. Viewing k as a function of n, as long as k(n) ∈ O(lg(n)), it has the same asymptotic. In particular, for any constant choice of k, the asymptotic are the same.

**ANSWER (D):**

. If we optimize the previous expression using our calculus 1 skills to get k, we have that c1n− nc^2/ k = 0 where c1 and c2 are the confidents of nk and n log(n/k) hidden by the asymptotic notation. In particular, a constant choice of k is optimal. In practice we could find the best choice of this k by just trying and timing for various values for sufficiently large n.



### ANSWER(a):

We need to prove that A0 contains the same elements as A, which is easily seen to be true because the only modification we make to A is swapping its elements, so the resulting array must contain a rearrangement of the elements in the original array.

**ANSWER(B):**

The for loop in lines 2 through 4 maintains the following loop invariant: At the start of each iteration, the position of the smallest element of A[i..n] is at most j. This is clearly true prior to the first iteration because the position of any element is at most A.length. To see that each iteration maintains the loop invariant, suppose that j = k and the position of the smallest

element of A[i..n] is at most k. Then we compare A[k] to A[k − 1]. If A[k] < A[k − 1] then A[k − 1] is not the smallest element of A[i..n], so when we swap A[k] and A[k − 1] we know that the

smallest element of A[i..n] must occur in the first k − 1 positions of the subarray, the maintaining the invariant. On the other hand, if A[k] ≥ A[k − 1] then the smallest element can’t be A[k]. Since we do nothing, we conclude that the smallest element has position at most k − 1. Upon termination, the smallest element of A[i..n] is in position i.

**ANSWER(C):**

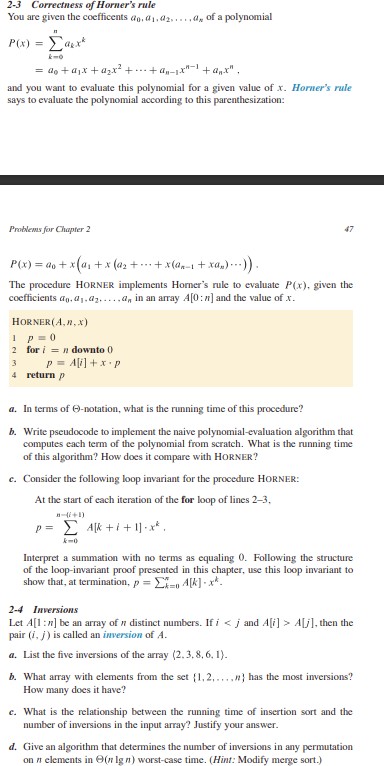
The for loop in lines 1 through 4 maintain the following loop invariant: At the start of each iteration the subarray A[1..i − 1] contains the i − 1 smallest elements of A in sorted order. Prior to the first iteration i = 1, and the first 0 elements of A are trivially sorted. To see that each iteration maintains the loop

invariant, fix i and suppose that A[1..i − 1] contains the i − 1 smallest elements of A in sorted order. Then we run the loop in lines 2 through 4. We showed in part b that when this loop terminates, the smallest

element of A[i..n] is in position i. Since the i − 1 smallest elements of A are already in A[1..i − 1], A[i] must be the i th smallest element of A. Therefore A[1..i] contains the i smallest elements of A in sorted order, maintaining the loop invariant. Upon termination, A[1..n] contains the n elements of A in sorted order as desired.

**ANSWER(D):**

The i the iteration of the for loop of lines 1 through 4 will cause n − i iterations of the for loop of lines 2 through 4, each with constant time execution, so the worst-case running time is Θ(n 2 ). This is the same as that of insertion sort; however, bubble sort also has best-case running time Θ(n 2 ) whereas insertion sort has best-case running time Θ(n).



### ANSWER(a):

If we assume that the arithmetic can all be done in constant time, then since the loop is being executed n times, it has runtime Θ(n).

**ANSWER(b):**

y = 0

for i=0 to n do yi = x

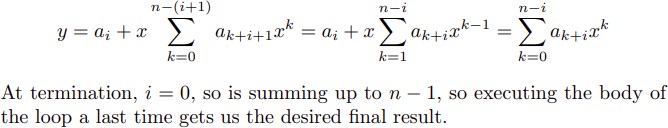
for j=1 to n yi = yix end for

y = y + aiyi end for

This code has runtime Θ(n 2 ) because it has to compute each of the powers of x. This is slower than Horner’s rule

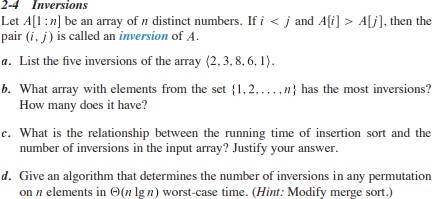
**ANSWER(C):**

Initially, i = n, so, the upper bound of the summation is −1, so the sum evaluates to 0, which is the value of y. For preservation, suppose it is true for an i, then,



**ANSWER(D):**

We just showed that the algorithm evaluated Σn ,k=0akx^k . This is the value of the polynomial evaluated at x.



**ANSWER(a):**

. The five inversions are (2, 1), (3, 1), (8, 6), (8, 1), and (6, 1).

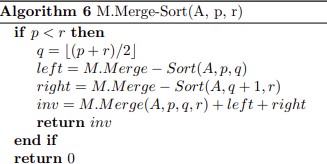
**ANSWER(b):**

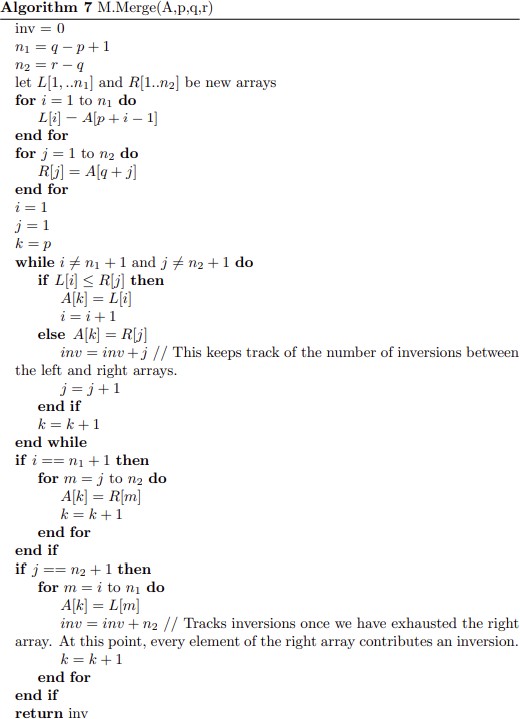


**ANSWER(c):**

The running time of insertion sort is a constant times the number of inversions. Let P I(i) denote the number of j < i such that A[j] > A[i]. Then n i=1 I(i) equals the number of inversions in A. Now consider the while loop on lines 5-7 of the insertion sort algorithm. The loop will execute once for each element of A which has index less than j is larger than A[j]. Thus, it will execute I(j) times. We reach this while loop once for each iteration of the P for loop, so the number of constant time steps of insertion sort is n j=1 I(j) which is exactly the inversion number of A.

**ANSWER(D):**





**3.1-1 Modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3.**

**ANSWER:**

To modify the lower-bound argument for Insertion Sort for input sizes not a multiple of 3, we simply note that the number of comparisons in the worst case is:

### T(n)=n(n−1)/2

This holds for any size **n** regardless of whether it's a multiple of **3**. The time complexity remains

### O(n^2).

**3.1-2 Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm from Exercise 2.2-2**

**ANSWER:**

To analyze the running time of **Selection Sort** using reasoning similar to Insertion Sort, let's break it down step-by-step:

Selection Sort Process:

1. Find the smallest element in the array and swap it with the first element.
2. Find the second smallest element and swap it with the second element.
3. Continue this process for all elements.

Comparisons in Selection Sort:

For an array of size n:

1. In the first pass, it makes n−1n-1n−1 comparisons to find the smallest element.
2. In the second pass, it makes n−2n-2n−2 comparisons to find the second smallest element.
3. This continues until the last pass, where it makes 1 comparison.

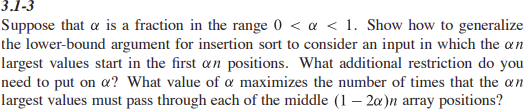
Total Comparisons:

The total number of comparisons is:

### T(n)=n(n−1)/2

Time Complexity:

Since T(n)=n(n−1)/2 the time complexity of Selection Sort is O(n^2) similar to Insertion Sort. However, unlike Insertion Sort, the number of swaps in Selection Sort is O(n), since it performs exactly one swap per pass.



ANSWER:

The lower-bound argument for Insertion Sort with αn\alpha nαn largest values in the first αn\alpha nαn positions shows that the number of comparisons is proportional to α(1−2α)nTo maximize the number of comparisons, α=1/4

The additional restriction is that αn\alpha nαn must be an integer.



ANSWER:

Step-by-Step Proof:

Let f(n) and g(n) be asymptotically nonnegative functions (i.e., they are nonnegative for large n).

### Upper Bound:

max(f(n), g(n)) ≤f(n)+g(n))

his is because the maximum of two values is always less than or equal to their sum. So: max(f(n), g(n)) =O(f(n)+g(n))

### Lower Bound:

max(f(n), g(n)) ≥f(n)+g(n))>=f(n)+g(n)/2

This is because the maximum of two numbers is always at least half their sum. Therefore: max(f(n), g(n)) =Ω(f(n)+g(n))

This gives us the lower bound.

Since max(f(n), g(n)) is both O(f(n)+g(n) and Ω(f(n)+g(n)), we conclude: max(f(n), g(n)) =Θ(f(n)+g(n))

Thus, max(f(n), g(n))=Θ(f(n)+g(n))



ANSWER:

The statement "The running time of algorithm A is at least O(n^2) is meaningless because O(n^2) represents an **upper bound**, not a lower bound. The correct way to describe a lower bound is to use **Ω\Omega-notation**, as in "The running time is at least Ω(n^2)."



ANSWER:

1. Is 2^n+1=O(2n).

We know that:

2^n+1=2⋅2^2.

This shows that 2^n+1 just a constant multiple of 2^n, specifically 2^n+1=O(2^n )

2. Is 2^2n=O(2^n) We know that:

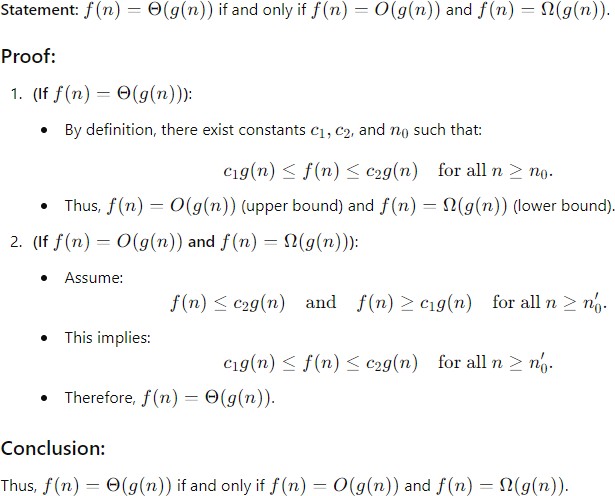
2^2n=(2n) ^2

This grows much faster than 2^n because squaring 2^n increases its growth rate significantly. Therefore, 2^n grows exponentially faster than 2^nand we cannot say that 2^2n=O(2^n)

thus, the correct ANSWERwer is:

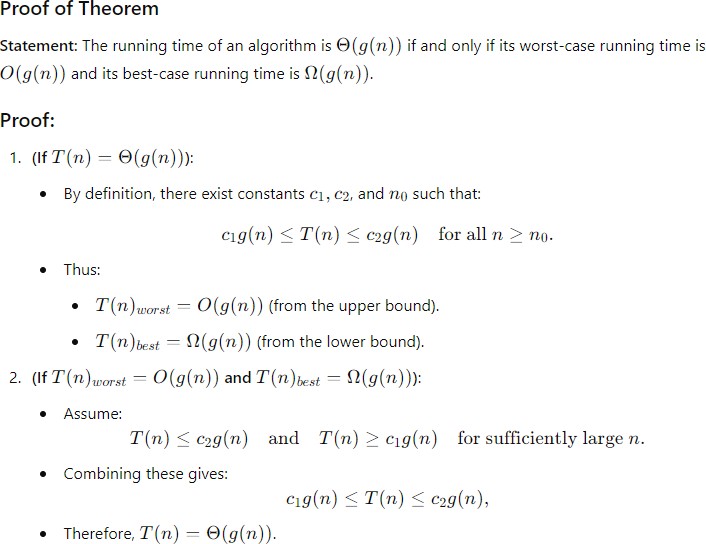
* 2n+1=O(2^n)
* 2^2n≠O(2^n)

### 3.2-4 Prove Theorem 3.1





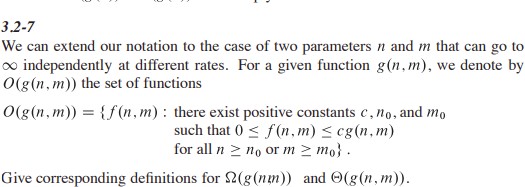
**ANSWER:**





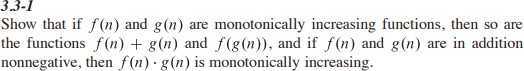
**ANSWER:**

**Suppose we had some f(n)** ∈ **o(g(n)) ∩ ω(g(n)). Then, we have 0 = limn→∞ f(n) g(n) = ∞**



**ANSWER:**

Ω(g(n, m)) = {f(n, m) : there exist positive constants c, n0, and m0 such that f(n, m) ≥ cg(n, m) for all n ≥ n0 or m ≥ m0}



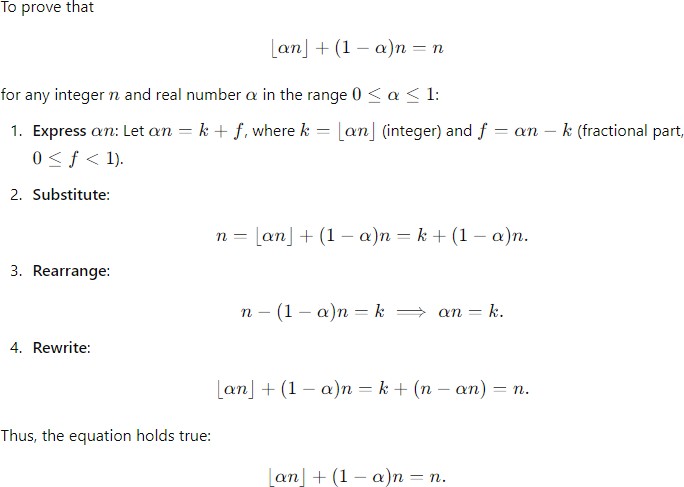
**ANSWER:**

Let n1 < n2 be arbitrary. From f and g being monatomic increasing, we know f(n1) < f(n2) and g(n1) < g(n2). So f(n1) + g(n1) < f(n2) + g(n1) < f(n2) + g(n2).

Since g(n1) < g(n2), we have f(g(n1)) < f(g(n2)). Lastly, if both are nonegative, then, f(n1)g(n1) = f(n2)g(n1) + (f(n2) − f(n1))g(n1) = f(n2)g(n2) + f(n2)(g(n2) − g(n1)) + (f(n2) − f(n1))g(n1) Since f(n1) ≥ 0, f(n2) > 0, so, the second term in this expression is greater than zero. The third term is nonnegative, so, the whole thing is< f(n2)g(n2).

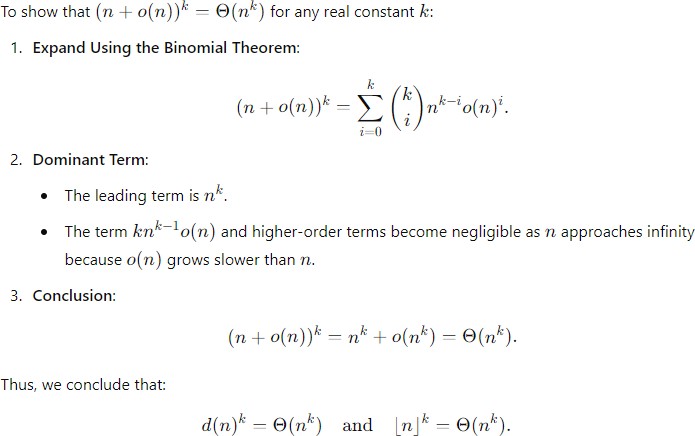


## **ANSWER:**





**ANSWER:**

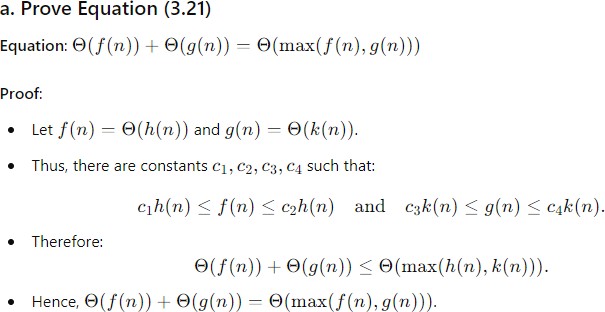


**3.3-4 Prove the following:**

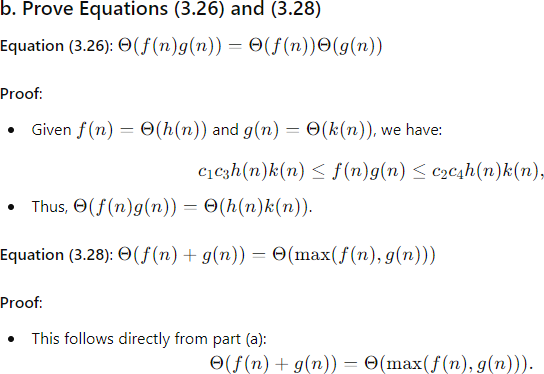
**a. Equation (3.21).**

**b. Equations (3.26)3(3.28).**

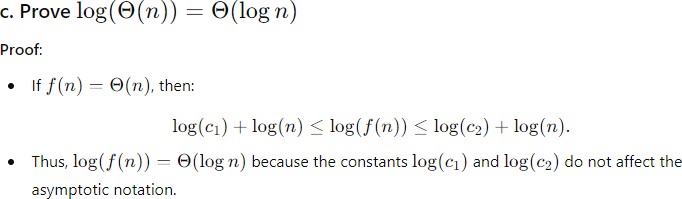
**c. lg.‚.n// D ‚.lg n/.**

**ANSWER(a):**

**ANSWER(b):**



**ANSWER(C):**





**ANSWER:**

# Is log(n) polynomial bounded?:

Yes, log(n) is polynomially bounded. For any k>0, log(n)≤n0.5 sufficiently large n.

# log(log(n)) polynomially bounded?:

Yes, log(log(n)) is polynomially bounded. For any k>0, log(log(n)) n^{0.1} is for sufficiently large n.

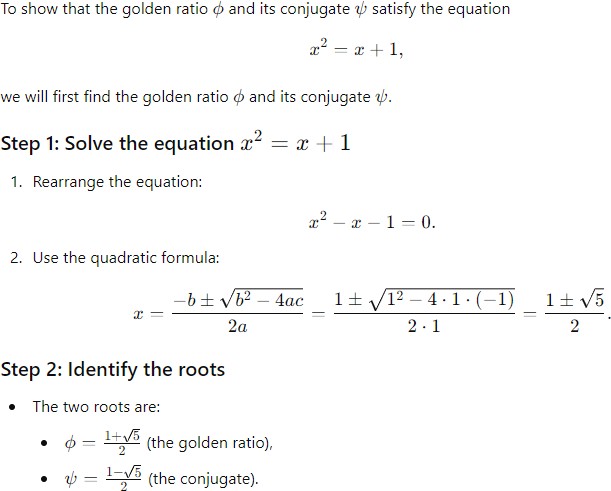


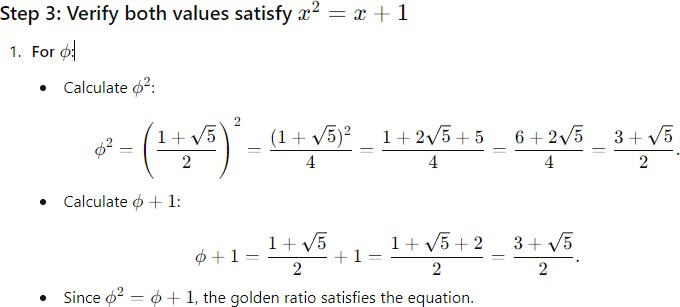
### ANSWER:

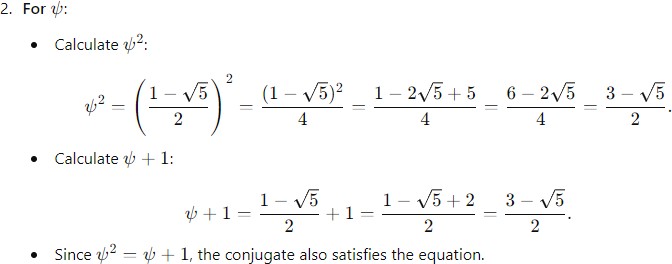
**Asymptotically**, log(log(n)) is larger than log(\sqrt\log(n)).

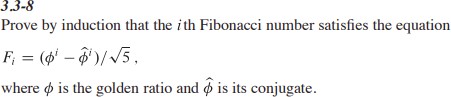


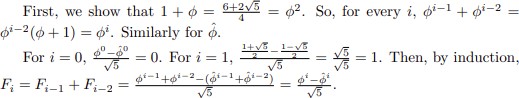
### ANSWER:





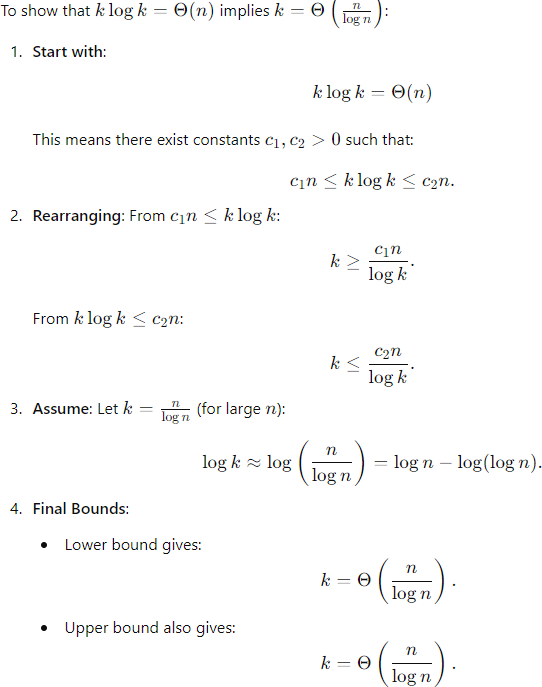


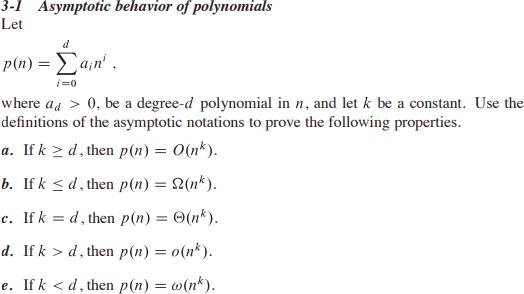


**ANSWER:**



### ANSWER:





**ANSWER(a):** a. If 𝑘 < 𝑑, then 𝑝(𝑛) = 𝑂(𝑛𝑘).

**Proof:** Since d is the highest degree of the polynomial, for large n, the term a n^ddominates the polynomial. Therefore, we can write:

(𝑛) ≤ 𝐶𝑛𝑑 for some constant 𝐶 > 0 and sufficiently large 𝑛.

Now, if 𝑘 < 𝑑, we can find a constant 𝑀 such that:

(𝑛) ≤ 𝐶𝑛𝑑 ≤ 𝐶′𝑛𝑘 for some 𝐶′ > 0 and sufficiently large 𝑛

where 𝐶′ = 𝐶𝑛𝑑−𝑘. Thus, we can conclude that (𝑛) = 𝑂(𝑛𝑘).

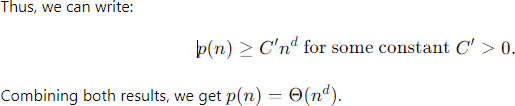
### ANSWER(b):

If 𝑘 = 𝑑, then (𝑛) = Θ(𝑛𝑘).

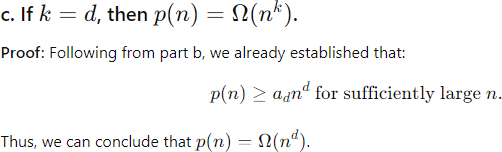
Proof: Following from part b, we already established that:

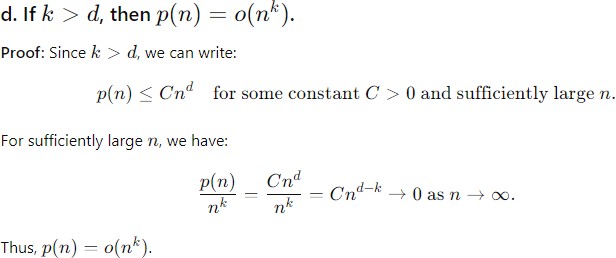
(𝑛) ≥ 𝑎𝑑𝑛𝑑 for sufficiently large 𝑛.

Thus, we can conclude that p(n)=Ω(nd)p(n) = Ω(n^d)p(n)*=*Ω(nd).

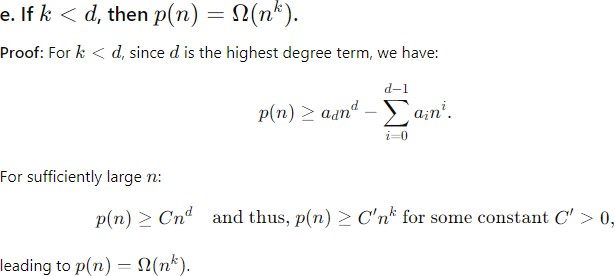


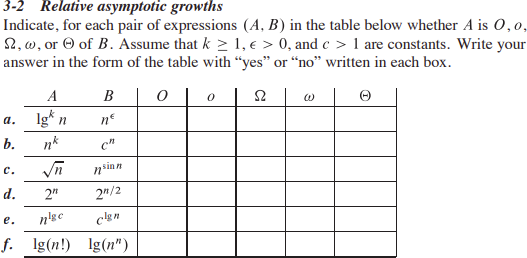
### ANSWER(C):

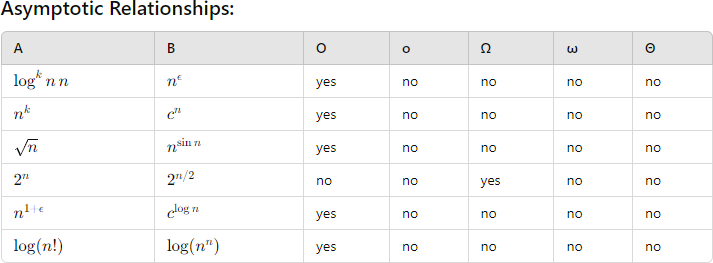


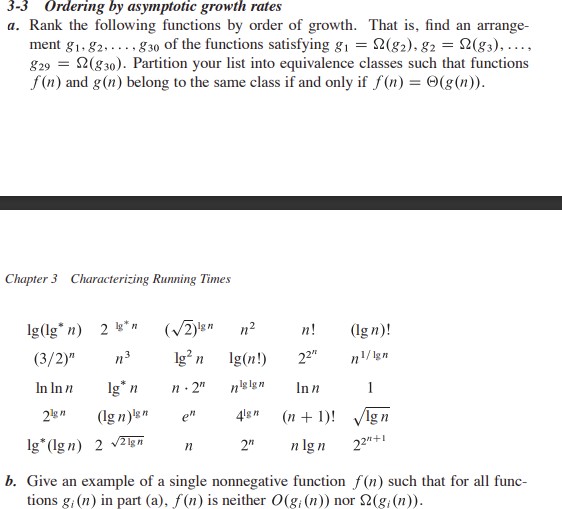
**ANSWER(D):**

### ANSWER(e):





**ANSWER:**



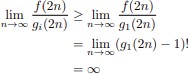
### ANSWER(A):

**ANSWER(b):**

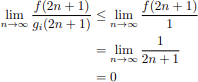
If we define the function



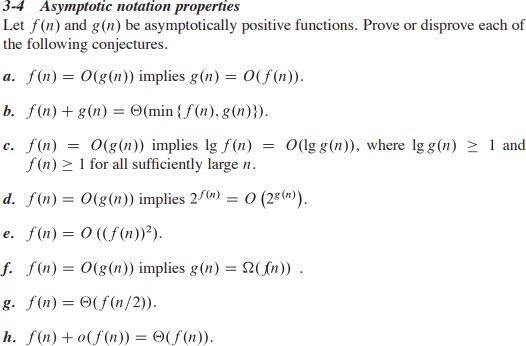
Note that f(n) meets the asymptotically positive requirement that this chapter puts on the functions analyzed. Then, for even n, we have



And for odd n, we have

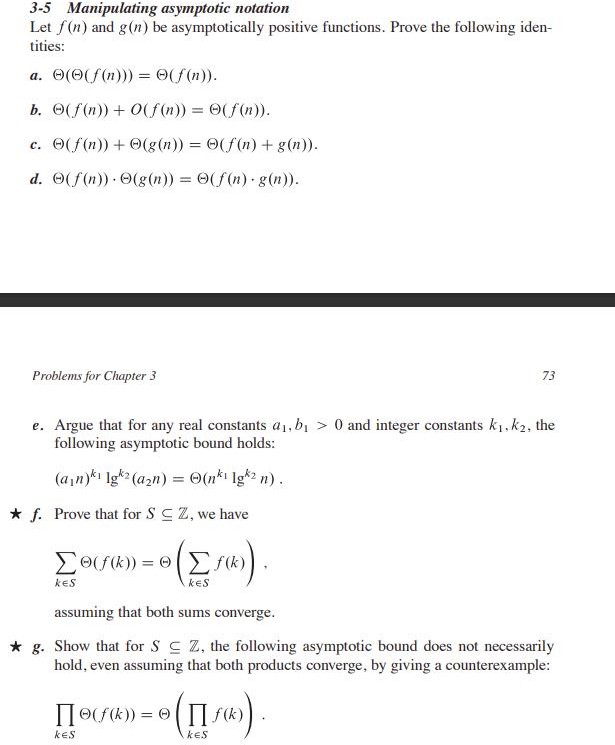






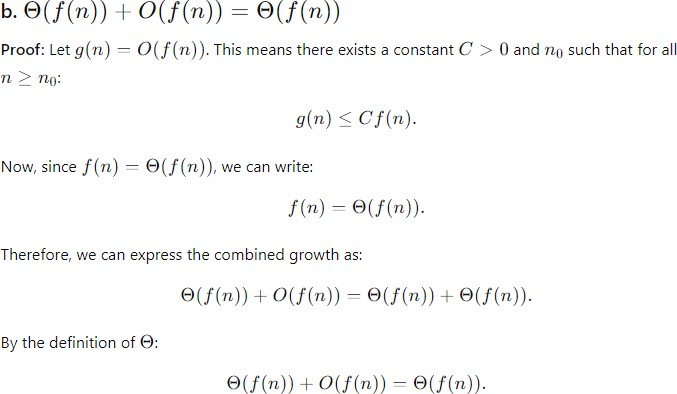
### ANSWER(a):

1. **False**
2. **False**
3. **True**
4. **True**
5. **False**
6. **False**
7. **True**
8. **True**

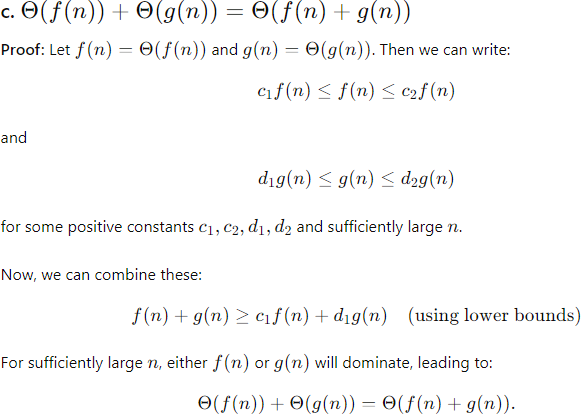


### ANSWER(a):

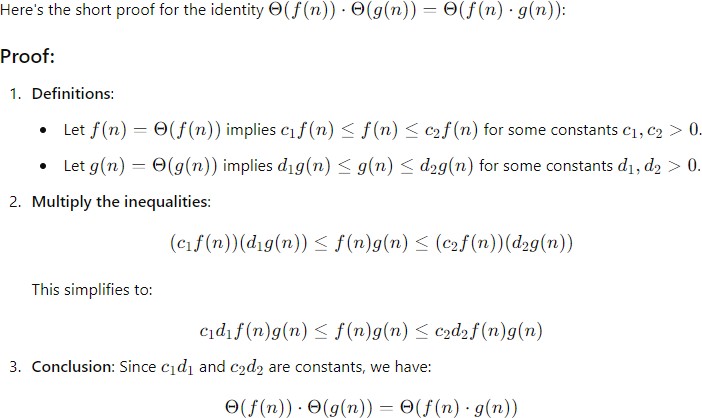
**ANSWER(b):**



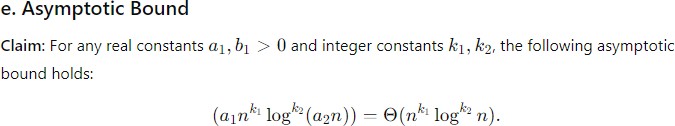
### ANSWER(C):



**ANSWER(D):**



### ANSWER(e):



Starting with the left-hand side:



Using the Binomial Theorem:



**Dominant Term:**

The dominant term when n is large is log^k2(n)



**Conclusion:**

Thus, we conclude:



**ANSWER(f):**

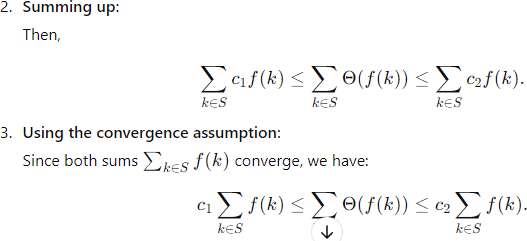
Summation Identity

**Claim:** For S⊆ZS we have:

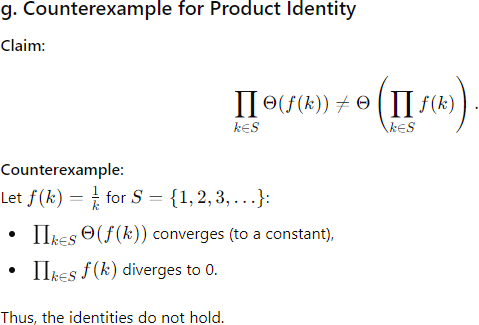


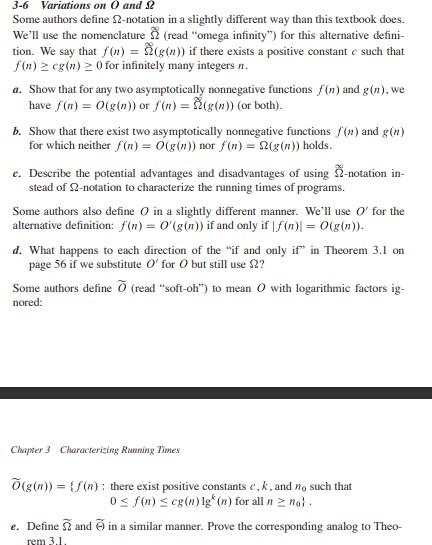


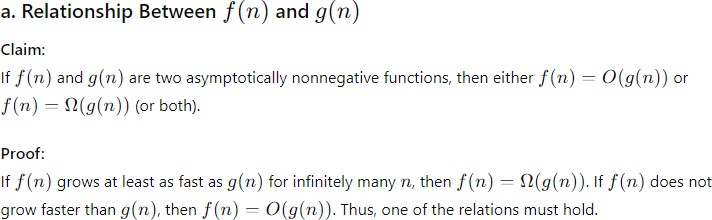
for some constants c1,c2>0



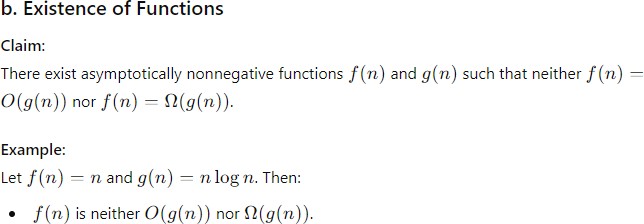
## **ANSWER(g):**



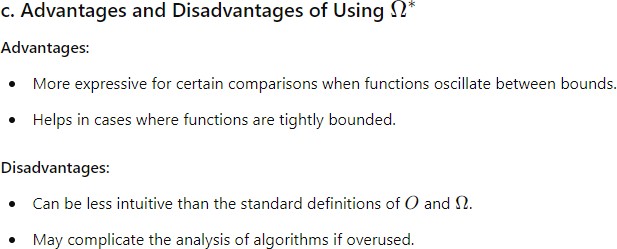


**ANSWER(a):**

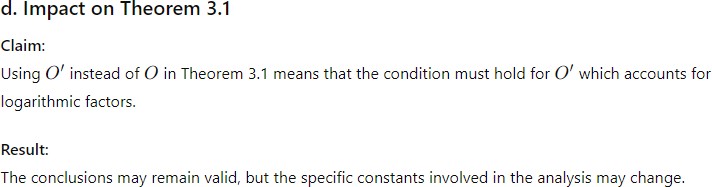
## ANSWER(b):



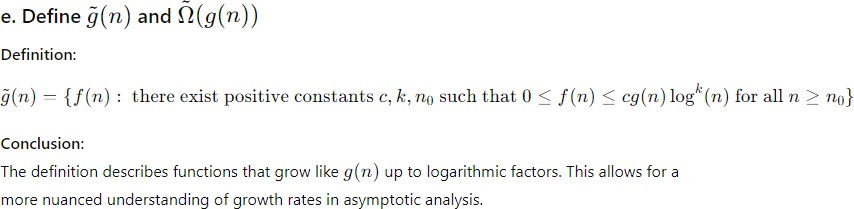
**ANSWER(c):**

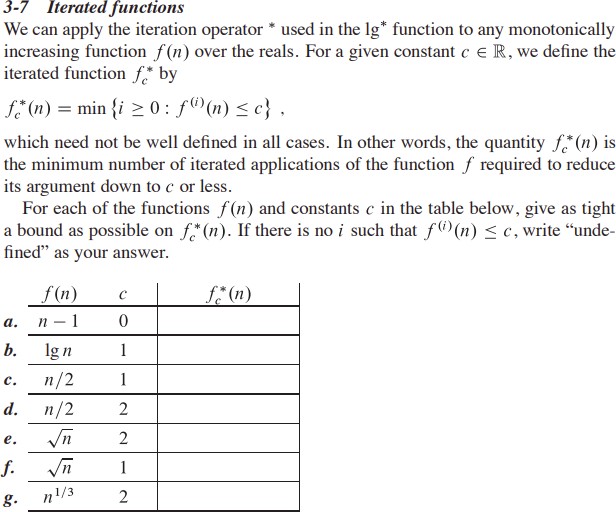


## **ANSWER(d):**



**ANSWER(e):**





## ANSWER:

