ABBOTTABAD UNIVERSITY OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

Name: uzair Raza

Class: 3D(BSCS)

Submitted to: Mr. Jamal Abdul Ahad

Assignment: 1

Date: 12/10/2024

Roll no: 14829

Github:

https://github.com/Uzairraza0011/DSA.git

Chapter 1 exercise Question & Answer

1.1-1 Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.

ANS:

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.

1.1-2 Other than speed, what other measures of efficiency might you need to consider in a real-world setting

ANS:

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.

1.1-3 Select a data structure that you have seen, and discuss its strengths and limitations.

ANS:

Data Structure: Array

Strengths:

1. **Fast Access**: You can quickly access any item using its index (O(1) time).
2. **Memory Efficiency**: Uses less memory overall since it stores items in adjacent memory.
3. **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.
2. **Slow Insertions/Deletions**: Adding or removing items (especially in the middle) can be slow, as it requires shifting elements (O(n) time).
3. **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.

ANS:

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).

1.1-5 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.

ANS:

Best Solution Required:

Sorting Transactions in Banking: In a bank, it's really important to sort transactions 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.

1.1-6 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.

ANS:

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 satellites. 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, new information 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.

ANS:

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.

(2) 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.

(3) 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?

ANS:

• 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) =3.84<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?

ANS:

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

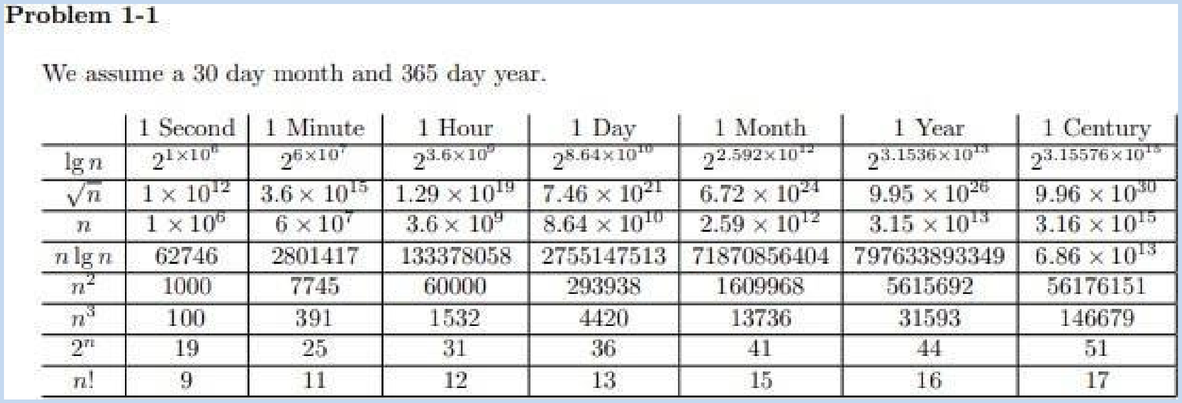
8100>5121.

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.



Problem 1-1

We assume a 30 day month and 365 day year.

	1 Second	1 Minute	1 Hour	1 Day	1 Month	1 Year	1 Century
$\lg n$	$2^{1\times10^6}$	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.64\times10^{10}}$	$2^{2.592\times10^{12}}$	$2^{3.1536\times10^{13}}$	$2^{3.15576\times10^{15}}$
$\sqrt{n}$	1×10^{12}	3.6×10^{15}	1.29×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.96×10^{30}
n	1×10^{6}	6×10^{7}	3.6×10^{9}	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}	3.16×10^{15}
$n\lg n$	62746	2801417	133378058	2755147513	71870856404	797633893349	6.86×10^{13}
n^2	1000	7745	60000	293938	1609968	5615692	56176151
n^3	100	391	1532	4420	13736	31593	146679
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

2.1-1

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

ANS:

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]

ANS:

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.

ANS:

```
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.

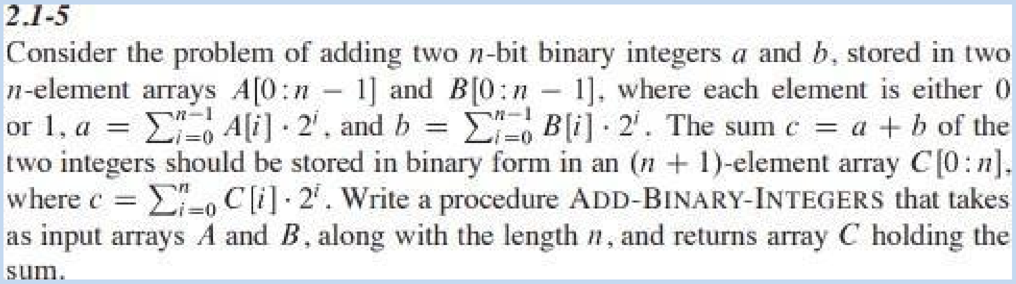
Write pseudocode for linear search, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

ANS:

```
LINEAR-SEARCH (A, n, x)
  for i = 1 to n
if A[i] == x
return i
return NI
```

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



2.1-5

Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays $A[0:n-1]$ and $B[0:n-1]$, where each element is either 0 or 1, $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$, and $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0:n]$, where $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

ANS:

```
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 ca

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.

ANS:

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

Given function:

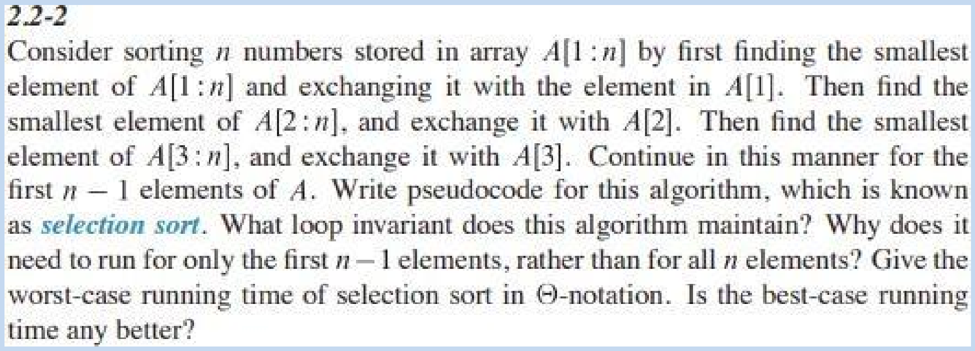
f(n)=n^3/1000+100n^2−100n+3

Steps:

1. **Find the biggest term**:
    - **n^3** is the biggest term because it grows faster than **n^2, n** and any constant.
2. **Ignore the constants**:
    - Ignore the constants like **1/1000,100 and -100**They don't matter for **Big-O**, only the biggest term does.
3. **Write the Big-O notation**:
    - The biggest term **is n^3**, so the function **is O (n^3).**

Final answer:

O(n3)



2.2-2

Consider sorting n numbers stored in array $A[1:n]$ by first finding the smallest element of $A[1:n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2:n]$, and exchange it with $A[2]$. Then find the smallest element of $A[3:n]$, and exchange it with $A[3]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the worst-case running time of selection sort in Θ-notation. Is the best-case running time any better?

Ans:

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 answers

ANS:

```
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.

Ans:

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).

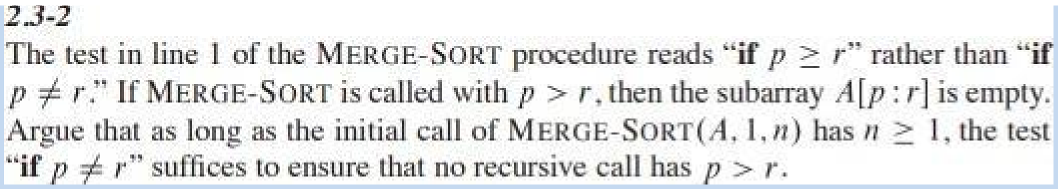
2.3-1 Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence h3; 41; 52; 26; 38; 57; 9; 49.

ANS:

1. **Step 1: Split the array** into two halves:
   - Left half: `[3, 41, 52, 26]`
   - Right half: `[38, 57, 9, 49]`
2. **Step 2: Keep splitting** each half until every part has only one element:
   - Left half `[3, 41, 52, 26]` becomes `[3], [41], [52], [26]`
   - Right half `[38, 57, 9, 49]` becomes `[38], [57], [9], [49]`
3. **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]`
   - 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]`
   - Then, merge `[38, 57]` and `[9, 49]` to become `[9, 38, 49, 57]`
4. **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]`.


2.3-2
The test in line 1 of the MERGE-SORT procedure reads "**if** $p \geq r$" rather than "**if** $p \neq r$." If MERGE-SORT is called with $p > r$, then the subarray $A[p:r]$ is empty. Argue that as long as the initial call of MERGE-SORT$(A, 1, n)$ has $n \geq 1$, the test "**if** $p \neq r$" suffices to ensure that no recursive call has $p > r$.

ANS:

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

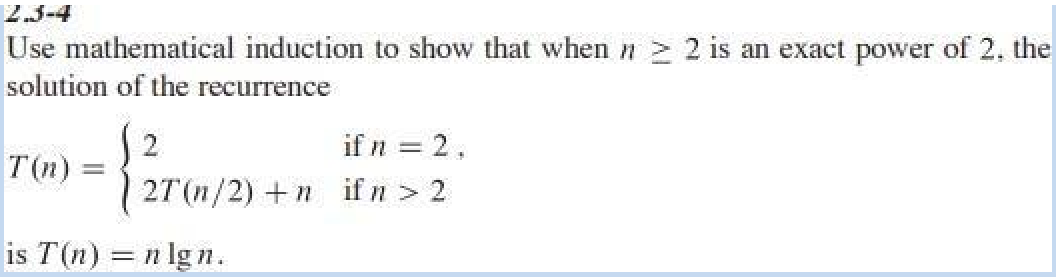
1. **Splitting**: The algorithm splits the array into two parts, ensuring `p` is never greater than `r` in recursive calls.
2. **Stopping**: When `p = r`, it means there's only one element left, so recursion stops.
3. **Initial Call**: If `MERGE-SORT(A, 1, n)` has `n ≥ 1`, then all recursive calls will 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.

ANS:

The loop invariant for the **MERGE** procedure states that at the start of each iteration, the elements merged into `A` from sub arrays `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 sub array is fully merged, allowing any remaining elements from the other sorted sub-array to be added directly to `A`. Thus, the **MERGE** procedure correctly combines two sorted sub arrays into one sorted array.

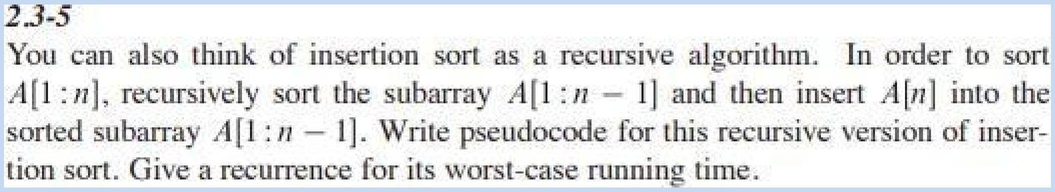

2.3-4
Use mathematical induction to show that when $n \geq 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2 , \\ 2T(n/2) + n & \text{if } n > 2 \end{cases}$$

is $T(n) = n \lg n$.

ANS:
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)


2.3-5
You can also think of insertion sort as a recursive algorithm. In order to sort $A[1:n]$, recursively sort the subarray $A[1:n-1]$ and then insert $A[n]$ into the sorted subarray $A[1:n-1]$. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

ANS:

```
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 sub array



INSERT(A, n)
```

```
1. key = A[n]              // The element to be inserted
2. i = n - 1
3. while i > 0 and A[i] > key
4.     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

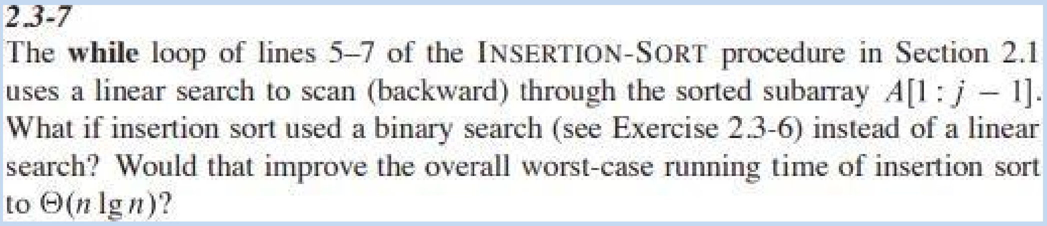
Let T(n) be the worst-case running time of the recursive insertion sort for an array of size nnn. The recurrence relation can be expressed as follows:

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 sub array being searched is already sorted, the searching algorithm can check the midpoint of the sub array against v and eliminate half of the sub array from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the sub array each time. Write pseudo code, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is big O log n.
ANS:

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.



2.3-7
The **while** loop of lines 5–7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1:j-1]$. What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

ANS:
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)

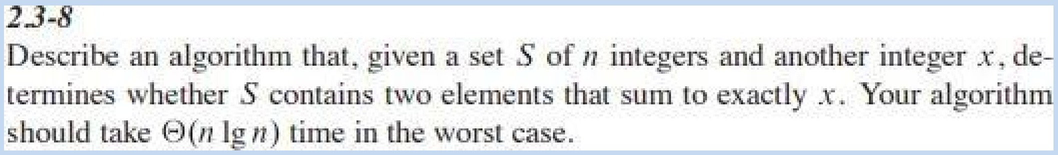
Thus, the overall complexity is:

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.

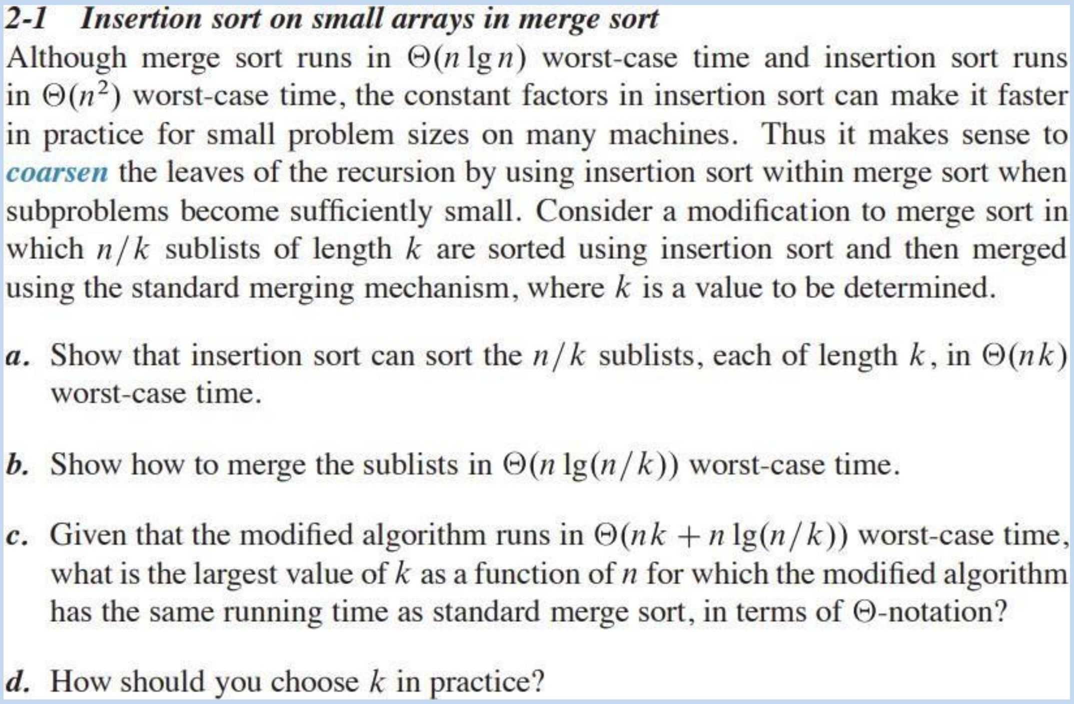


2.3-8
Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum to exactly x. Your algorithm should take $\Theta(n \lg n)$ time in the worst case.

ANS:

```
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
```



2-1 Insertion sort on small arrays in merge sort

Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

a. Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

b. Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

d. How should you choose k in practice?

ANS (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).

ANS(b):

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:

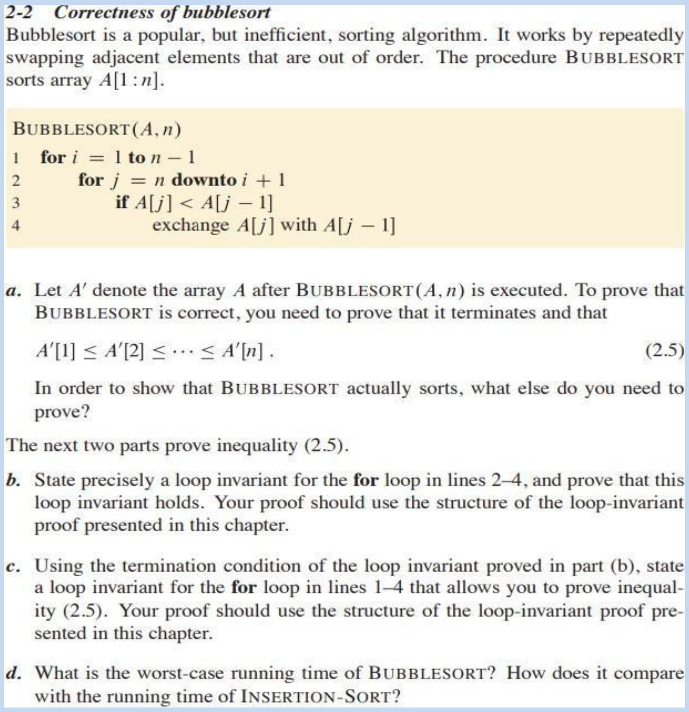
   Θ(log(n/k))

ANS(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.

ANS (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 confident 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.



2-2 *Correctness of bubblesort*

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order. The procedure BUBBLESORT sorts array $A[1:n]$.

```
BUBBLESORT(A, n)
1  for i = 1 to n − 1
2      for j = n downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

a. Let A' denote the array A after BUBBLESORT(A, n) is executed. To prove that BUBBLESORT is correct, you need to prove that it terminates and that

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n] . \qquad (2.5)$$

In order to show that BUBBLESORT actually sorts, what else do you need to prove?

The next two parts prove inequality (2.5).

b. State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop-invariant proof presented in this chapter.

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1–4 that allows you to prove inequality (2.5). Your proof should use the structure of the loop-invariant proof presented in this chapter.

d. What is the worst-case running time of BUBBLESORT? How does it compare with the running time of INSERTION-SORT?

ANS(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.

ANS(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 sub array, 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.

ANS(C):

The for loop in lines 1 through 4 maintain the following loop invariant: At the start of each iteration the sub array 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.

ANS(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 $\Theta(n^2)$. This is the same as that of insertion sort; however, bubble sort also has best-case running time $\Theta(n^2)$ whereas insertion sort has best-case running time $\Theta(n)$.

2-3 *Correctness of Horner's rule*

You are given the coefficents $a_0, a_1, a_2, \ldots, a_n$ of a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n , \end{aligned}$$

and you want to evaluate this polynomial for a given value of x. ***Horner's rule*** says to evaluate the polynomial according to this parenthesization:

$$P(x) = a_0 + x\Big(a_1 + x\left(a_2 + \cdots + x(a_{n-1} + x a_n)\cdots\right)\Big) .$$

The procedure HORNER implements Horner's rule to evaluate $P(x)$, given the coefficients $a_0, a_1, a_2, \ldots, a_n$ in an array $A[0:n]$ and the value of x.

```
HORNER(A, n, x)
1  p = 0
2  for i = n downto 0
3      p = A[i] + x · p
4  return p
```

a. In terms of Θ-notation, what is the running time of this procedure?

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare with HORNER?

c. Consider the following loop invariant for the procedure HORNER:

At the start of each iteration of the **for** loop of lines 2–3,

$$p = \sum_{k=0}^{n-(i+1)} A[k+i+1] \cdot x^k .$$

Interpret a summation with no terms as equaling 0. Following the structure of the loop-invariant proof presented in this chapter, use this loop invariant to show that, at termination, $p = \sum_{k=0}^{n} A[k] \cdot x^k$.

2-4 *Inversions*

Let $A[1:n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

b. What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

d. Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

ANS(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).

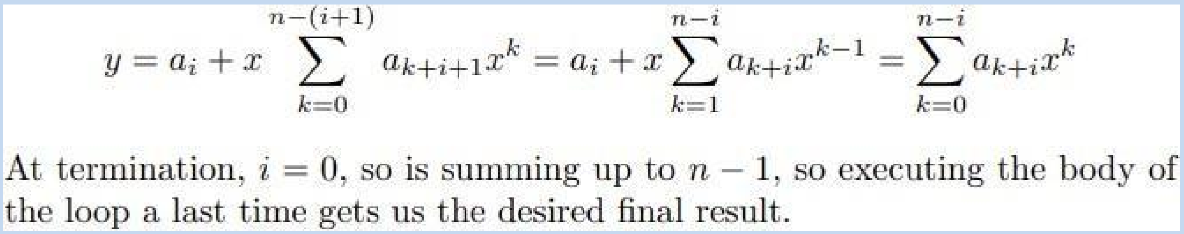
ANS(b):

```
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

ANS(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,

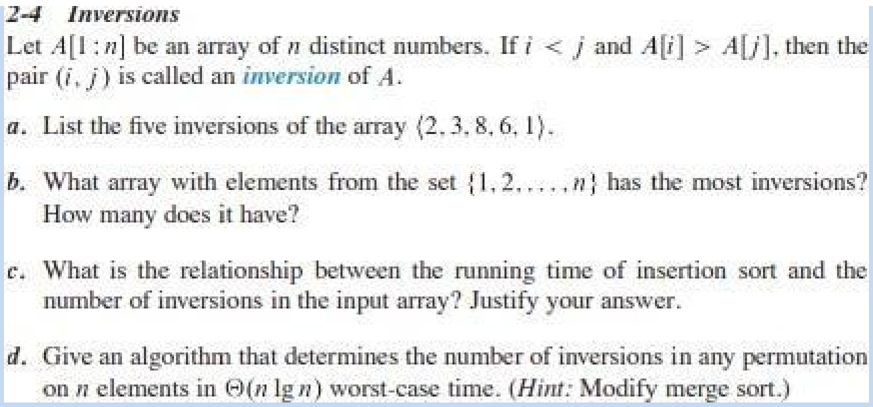


$$y = a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k = a_i + x \sum_{k=1}^{n-i} a_{k+i} x^{k-1} = \sum_{k=0}^{n-i} a_{k+i} x^k$$

At termination, $i = 0$, so is summing up to $n - 1$, so executing the body of the loop a last time gets us the desired final result.

ANS(D):

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


2-4 *Inversions*

Let $A[1:n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

b. What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

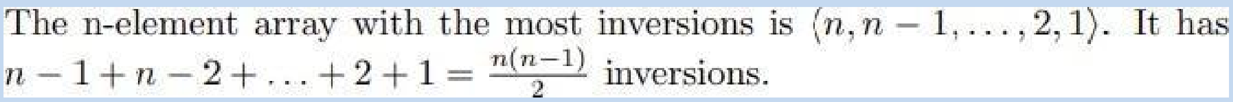
c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

d. Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

ANS(a):

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

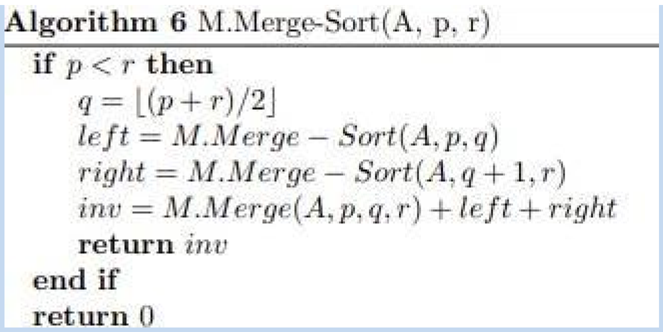
ANS(b):


The n-element array with the most inversions is $\langle n, n-1, \ldots, 2, 1\rangle$. It has $n-1+n-2+\ldots+2+1 = \frac{n(n-1)}{2}$ inversions.

ANS(c):

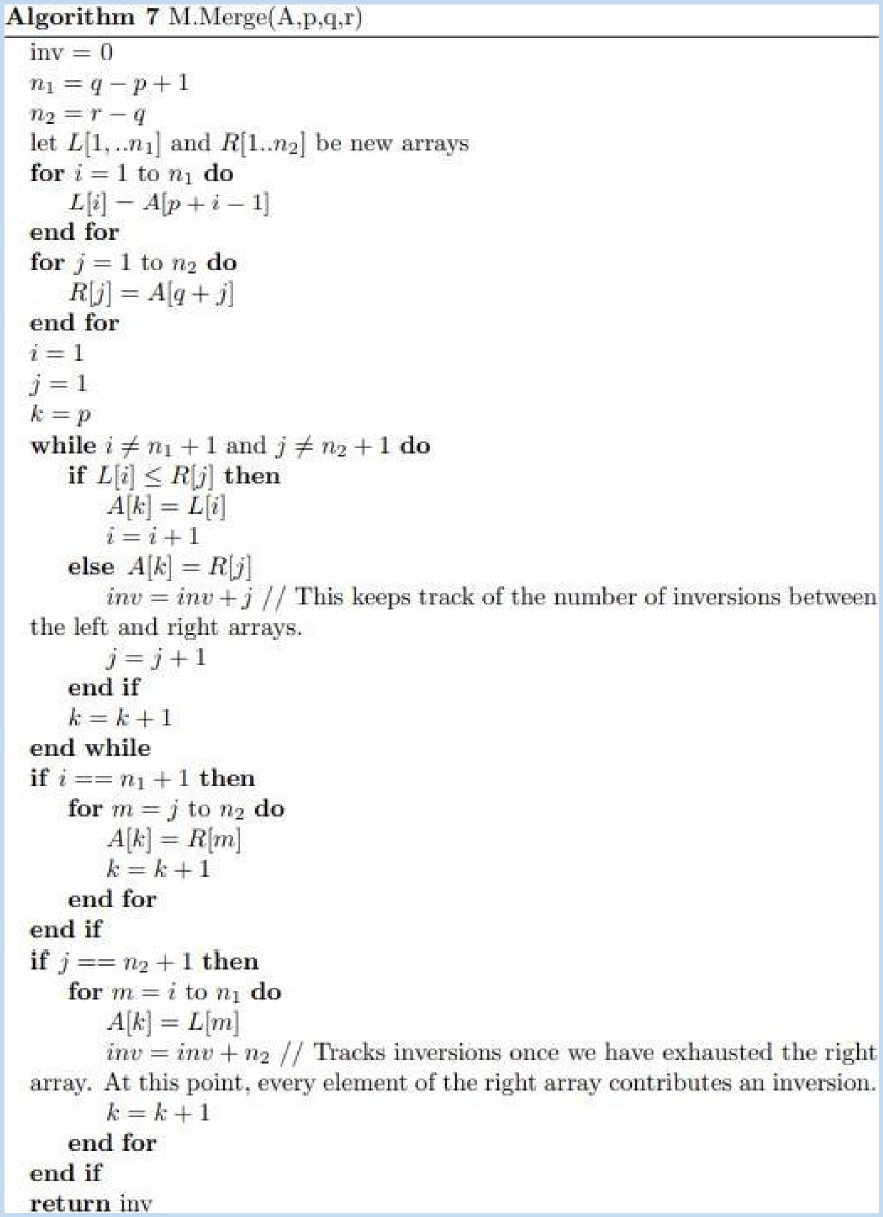
The running time of insertion sort is a constant times the number of inversions. Let P I(i) denote the number of j < i such that A[j] > A[i]. Then n i=1 I(i) equals the number of inversions in A. Now consider the while loop on lines 5-7 of the insertion sort algorithm. The loop will execute once for 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.

ANS(D):



Algorithm 6 M.Merge-Sort(A, p, r)

```
if $p < r$ then
    $q = \lfloor (p+r)/2 \rfloor$
    $left = M.Merge - Sort(A, p, q)$
    $right = M.Merge - Sort(A, q+1, r)$
    $inv = M.Merge(A, p, q, r) + left + right$
    return $inv$
end if
return 0
```



Algorithm 7 M.Merge(A,p,q,r)

```
inv = 0
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1, ..n_1]$ and $R[1..n_2]$ be new arrays
for $i = 1$ to $n_1$ do
    $L[i] - A[p + i - 1]$
end for
for $j = 1$ to $n_2$ do
    $R[j] = A[q + j]$
end for
$i = 1$
$j = 1$
$k = p$
while $i \neq n_1 + 1$ and $j \neq n_2 + 1$ do
    if $L[i] \leq R[j]$ then
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $inv = inv + j$ // This keeps track of the number of inversions between
the left and right arrays.
        $j = j + 1$
    end if
    $k = k + 1$
end while
if $i == n_1 + 1$ then
    for $m = j$ to $n_2$ do
        $A[k] = R[m]$
        $k = k + 1$
    end for
end if
if $j == n_2 + 1$ then
    for $m = i$ to $n_1$ do
        $A[k] = L[m]$
        $inv = inv + n_2$ // Tracks inversions once we have exhausted the right
array. At this point, every element of the right array contributes an inversion.
        $k = k + 1$
    end for
end if
return inv
```

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

ANS:

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

ANS:

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.

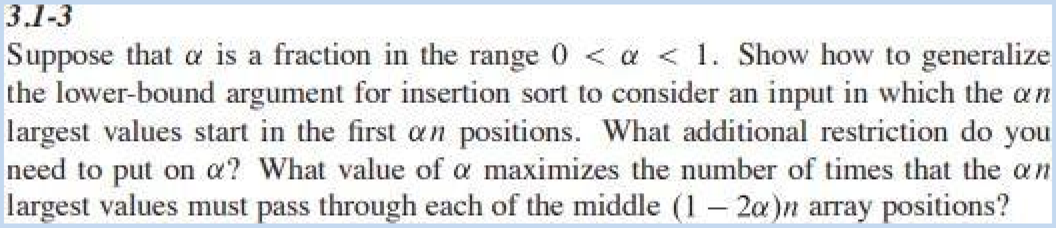
Total Comparisons:

The total number of comparisons is:

T(n)=n(n−1)/2

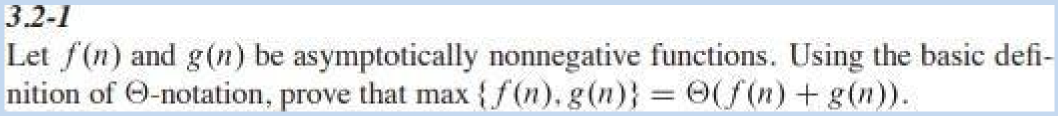
Time Complexity:

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.


3.1-3
Suppose that α is a fraction in the range $0 < \alpha < 1$. Show how to generalize the lower-bound argument for insertion sort to consider an input in which the αn largest values start in the first αn positions. What additional restriction do you need to put on α? What value of α maximizes the number of times that the αn largest values must pass through each of the middle $(1 - 2\alpha)n$ array positions?

ANS:
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.


3.2-1
Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.

ANS:

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:

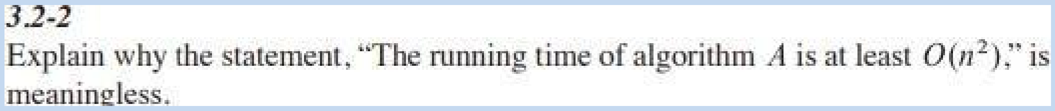
max(f(n), g(n)) =Ω(f(n)+g(n))

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))

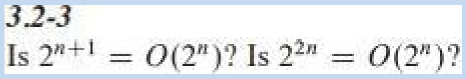
Thus, max(f(n), g(n))=Θ(f(n)+g(n))



3.2-2

Explain why the statement, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

ANS:
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)."



3.2-3

Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

ANS:
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:

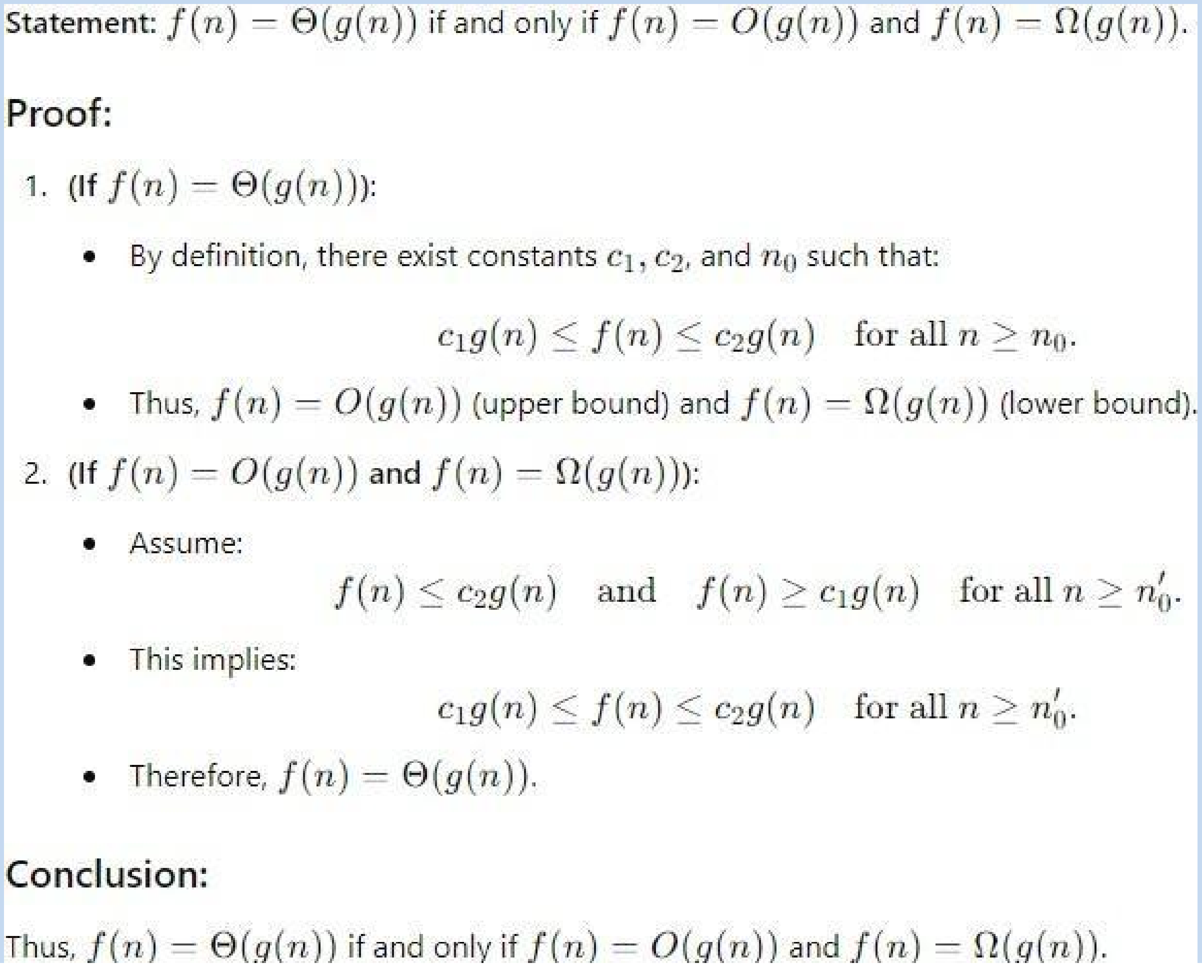
2^2n=(2n) ^2

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 answer is:

- 2n+1=O(2^n)
- 2^2n≠O(2^n)

3.2-4 Prove Theorem 3.1



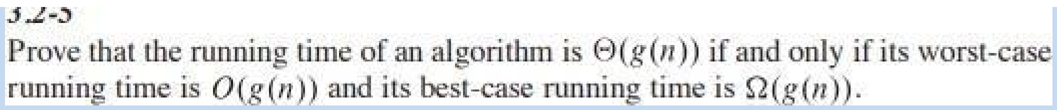
Statement: $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Proof:

1. (If $f(n) = \Theta(g(n))$):
   - By definition, there exist constants c_1, c_2, and n_0 such that:
   $$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \text{for all } n \geq n_0.$$
   - Thus, $f(n) = O(g(n))$ (upper bound) and $f(n) = \Omega(g(n))$ (lower bound).
2. (If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$):
   - Assume:
   $$f(n) \leq c_2 g(n) \quad \text{and} \quad f(n) \geq c_1 g(n) \quad \text{for all } n \geq n_0'.$$
   - This implies:
   $$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \text{for all } n \geq n_0'.$$
   - Therefore, $f(n) = \Theta(g(n))$.

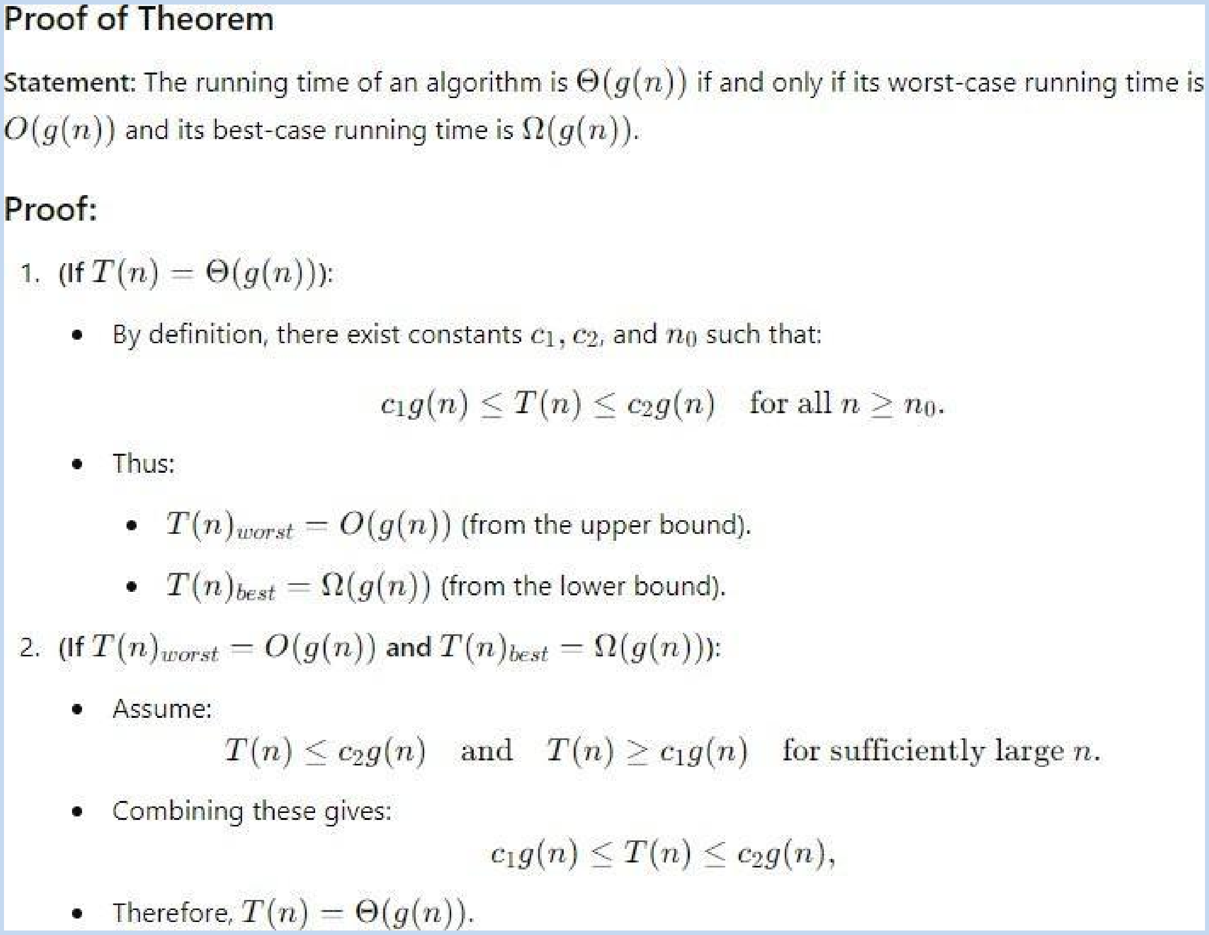
Conclusion:

Thus, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.


3.2-5

Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

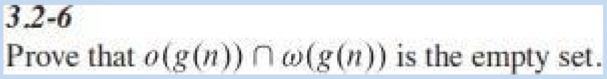
ANS:


Proof of Theorem

Statement: The running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

Proof:

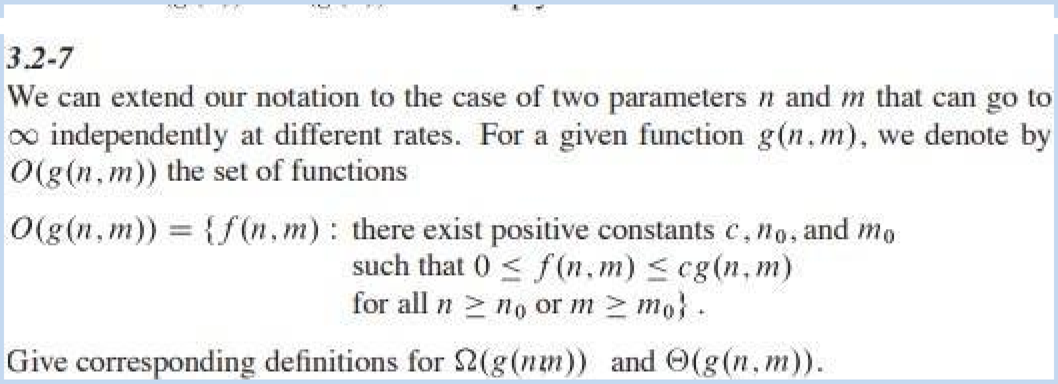
1. **(If $T(n) = \Theta(g(n))$):**
   - By definition, there exist constants c_1, c_2, and n_0 such that:
   $$c_1 g(n) \le T(n) \le c_2 g(n) \quad \text{for all } n \ge n_0.$$
   - Thus:
     - $T(n)_{worst} = O(g(n))$ (from the upper bound).
     - $T(n)_{best} = \Omega(g(n))$ (from the lower bound).
2. **(If $T(n)_{worst} = O(g(n))$ and $T(n)_{best} = \Omega(g(n))$):**
   - Assume:
   $$T(n) \le c_2 g(n) \quad \text{and} \quad T(n) \ge c_1 g(n) \quad \text{for sufficiently large } n.$$
   - Combining these gives:
   $$c_1 g(n) \le T(n) \le c_2 g(n),$$
   - Therefore, $T(n) = \Theta(g(n))$.


3.2-6

Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

ANS:

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


3.2-7

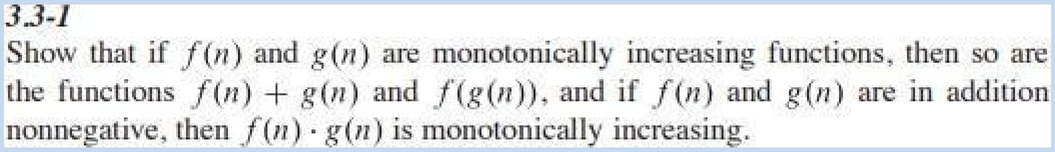
We can extend our notation to the case of two parameters n and m that can go to ∞ independently at different rates. For a given function $g(n, m)$, we denote by $O(g(n, m))$ the set of functions

$$O(g(n,m)) = \{f(n,m) : \text{there exist positive constants } c, n_0, \text{ and } m_0 \text{ such that } 0 \le f(n,m) \le cg(n,m) \text{ for all } n \ge n_0 \text{ or } m \ge m_0\} .$$

Give corresponding definitions for $\Omega(g(nm))$ and $\Theta(g(n, m))$.

ANS:

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}

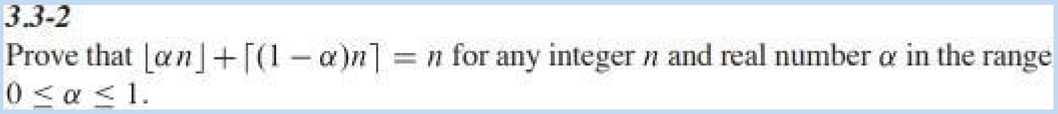

3.3-1

Show that if $f(n)$ and $g(n)$ are monotonically increasing functions, then so are the functions $f(n) + g(n)$ and $f(g(n))$, and if $f(n)$ and $g(n)$ are in addition nonnegative, then $f(n) \cdot g(n)$ is monotonically increasing.

ANS:

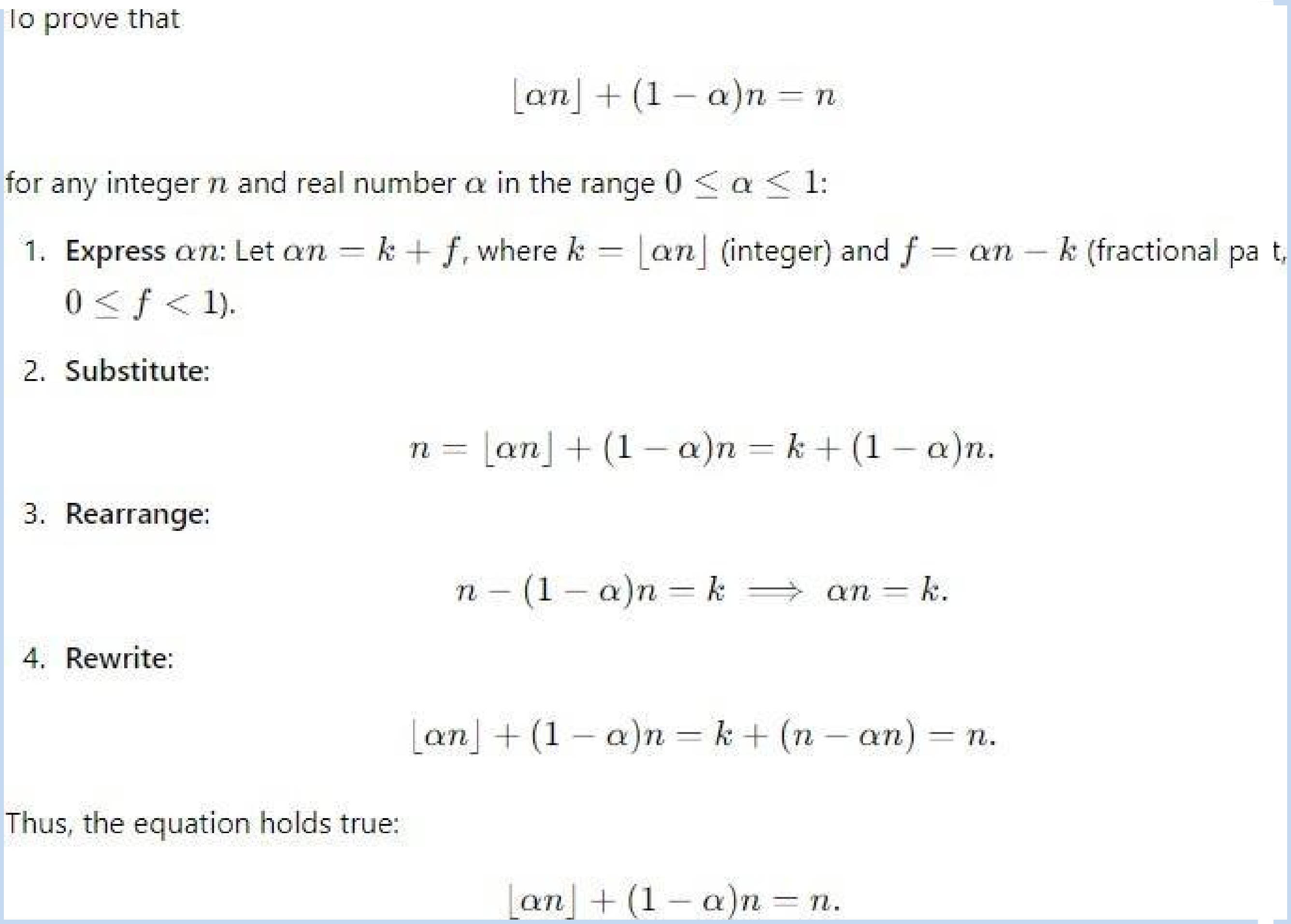
Let n1 < n2 be arbitrary. From f and g being monatomic increasing, we know f(n1) < f(n2) and g(n1) < g(n2). So f(n1) + g(n1) < f(n2) + g(n1) < f(n2) + g(n2).

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).


3.3-2

Prove that $\lfloor \alpha n \rfloor + \lceil (1-\alpha)n \rceil = n$ for any integer n and real number α in the range $0 \le \alpha \le 1$.

ANS:



To prove that

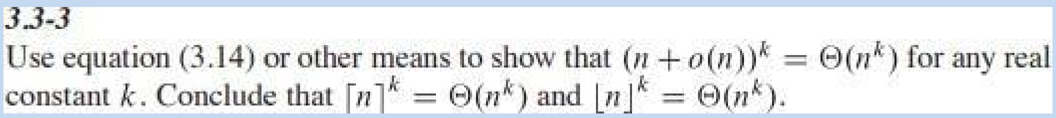
$$\lfloor \alpha n \rfloor + (1-\alpha)n = n$$

for any integer n and real number α in the range $0 \le \alpha \le 1$:

1. **Express αn:** Let $\alpha n = k + f$, where $k = \lfloor \alpha n \rfloor$ (integer) and $f = \alpha n - k$ (fractional pa t, $0 \le f < 1$).
2. **Substitute:**
$$n = \lfloor \alpha n \rfloor + (1-\alpha)n = k + (1-\alpha)n.$$
3. **Rearrange:**
$$n - (1-\alpha)n = k \implies \alpha n = k.$$
4. **Rewrite:**
$$\lfloor \alpha n \rfloor + (1-\alpha)n = k + (n - \alpha n) = n.$$

Thus, the equation holds true:

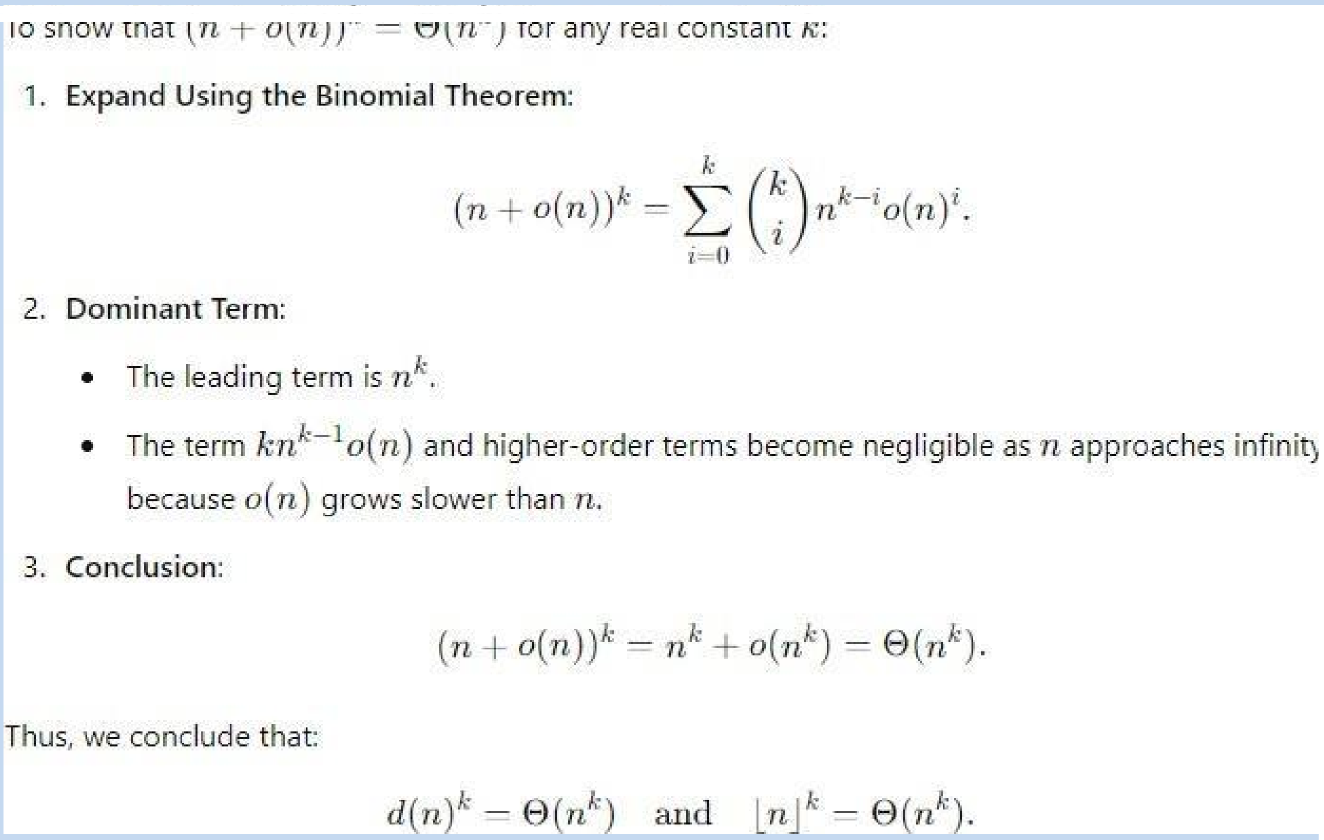
$$\lfloor \alpha n \rfloor + (1-\alpha)n = n.$$



3.3-3

Use equation (3.14) or other means to show that $(n + o(n))^k = \Theta(n^k)$ for any real constant k. Conclude that $\lceil n \rceil^k = \Theta(n^k)$ and $\lfloor n \rfloor^k = \Theta(n^k)$.

ANS:



To show that $(n + o(n))^k = \Theta(n^k)$ for any real constant k:

1. **Expand Using the Binomial Theorem:**

$$(n + o(n))^k = \sum_{i=0}^{k} \binom{k}{i} n^{k-i} o(n)^i.$$

2. **Dominant Term:**
   - The leading term is n^k.
   - The term $kn^{k-1}o(n)$ and higher-order terms become negligible as n approaches infinity because $o(n)$ grows slower than n.

3. **Conclusion:**

$$(n + o(n))^k = n^k + o(n^k) = \Theta(n^k).$$

Thus, we conclude that:

$$d(n)^k = \Theta(n^k) \quad \text{and} \quad \lfloor n \rfloor^k = \Theta(n^k).$$

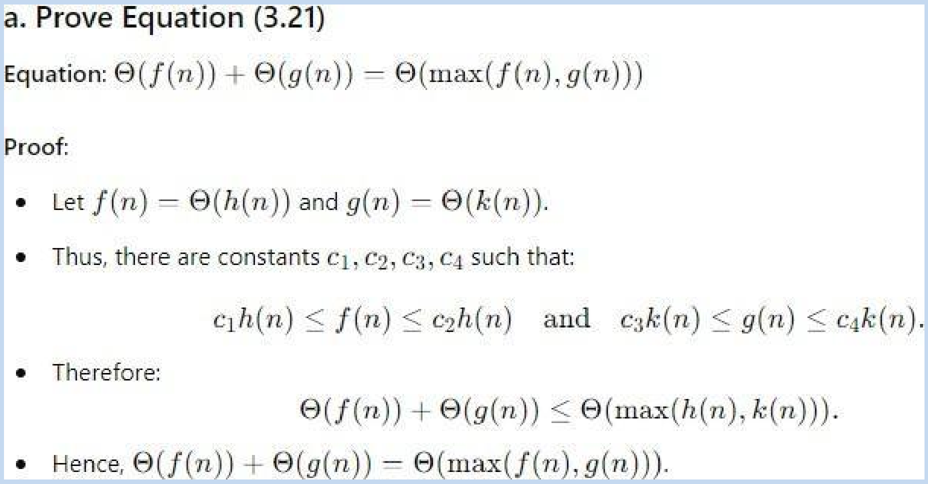
3.3-4 Prove the following:

a. Equation (3.21).

b. Equations (3.26)3(3.28).

c. lg.,.n// D ,.lg n/.

ANS(a):



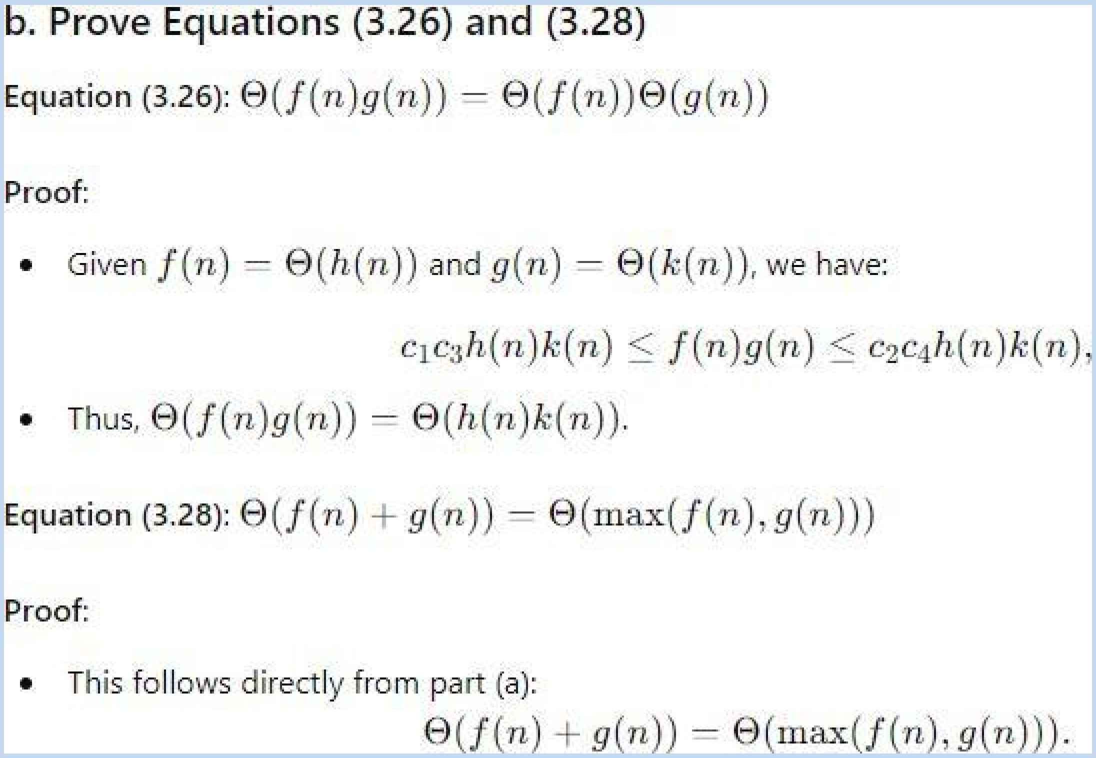
a. Prove Equation (3.21)

Equation: $\Theta(f(n)) + \Theta(g(n)) = \Theta(\max(f(n), g(n)))$

Proof:

- Let $f(n) = \Theta(h(n))$ and $g(n) = \Theta(k(n))$.
- Thus, there are constants c_1, c_2, c_3, c_4 such that:
$$c_1h(n) \le f(n) \le c_2h(n) \quad \text{and} \quad c_3k(n) \le g(n) \le c_4k(n).$$
- Therefore:
$$\Theta(f(n)) + \Theta(g(n)) \le \Theta(\max(h(n), k(n))).$$
- Hence, $\Theta(f(n)) + \Theta(g(n)) = \Theta(\max(f(n), g(n)))$.

ANS(b):



b. Prove Equations (3.26) and (3.28)

Equation (3.26): $\Theta(f(n)g(n)) = \Theta(f(n))\Theta(g(n))$

Proof:

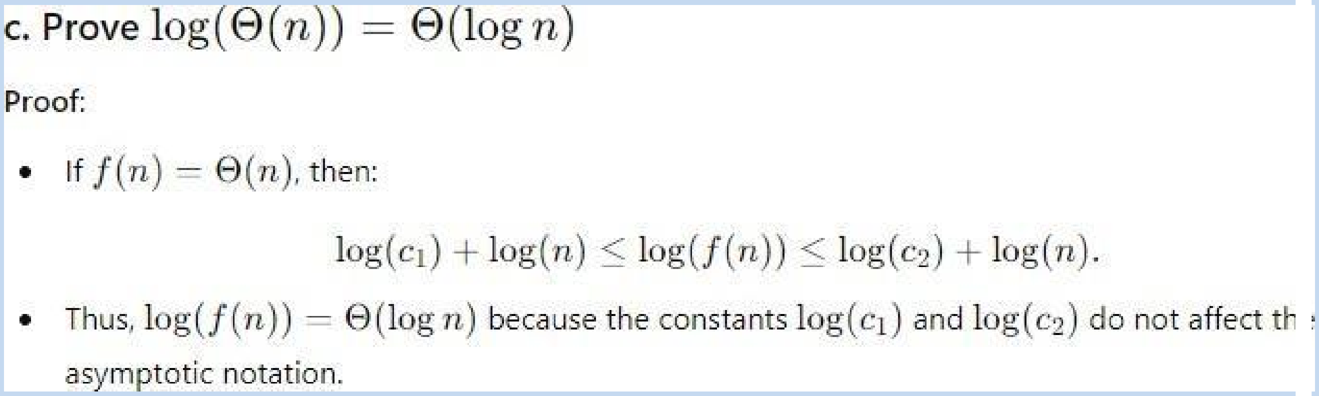
- Given $f(n) = \Theta(h(n))$ and $g(n) = \Theta(k(n))$, we have:
$$c_1c_3h(n)k(n) \le f(n)g(n) \le c_2c_4h(n)k(n),$$
- Thus, $\Theta(f(n)g(n)) = \Theta(h(n)k(n))$.

Equation (3.28): $\Theta(f(n) + g(n)) = \Theta(\max(f(n), g(n)))$

Proof:

- This follows directly from part (a):
$$\Theta(f(n) + g(n)) = \Theta(\max(f(n), g(n))).$$

ANS(C):

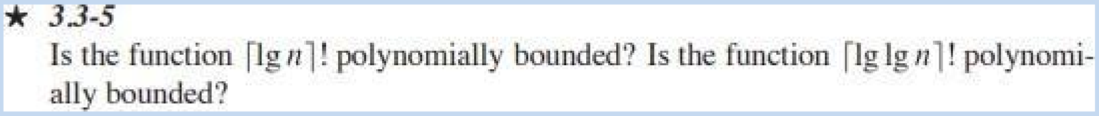

c. Prove $\log(\Theta(n)) = \Theta(\log n)$

Proof:

- If $f(n) = \Theta(n)$, then:

$$\log(c_1) + \log(n) \leq \log(f(n)) \leq \log(c_2) + \log(n).$$

- Thus, $\log(f(n)) = \Theta(\log n)$ because the constants $\log(c_1)$ and $\log(c_2)$ do not affect th asymptotic notation.


★ *3.3-5*
Is the function $\lceil \lg n \rceil!$ polynomially bounded? Is the function $\lceil \lg \lg n \rceil!$ polynomially bounded?

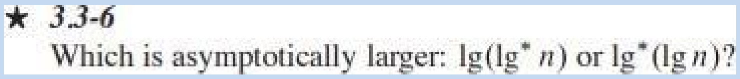
ANS:

1. Is log(n) polynomial bounded?:

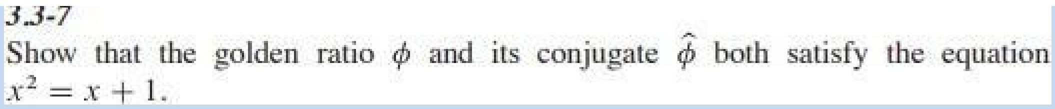
Yes, log(n) is polynomially bounded. For any k>0, log(n)≤n0.5 sufficiently large n.

2. 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.

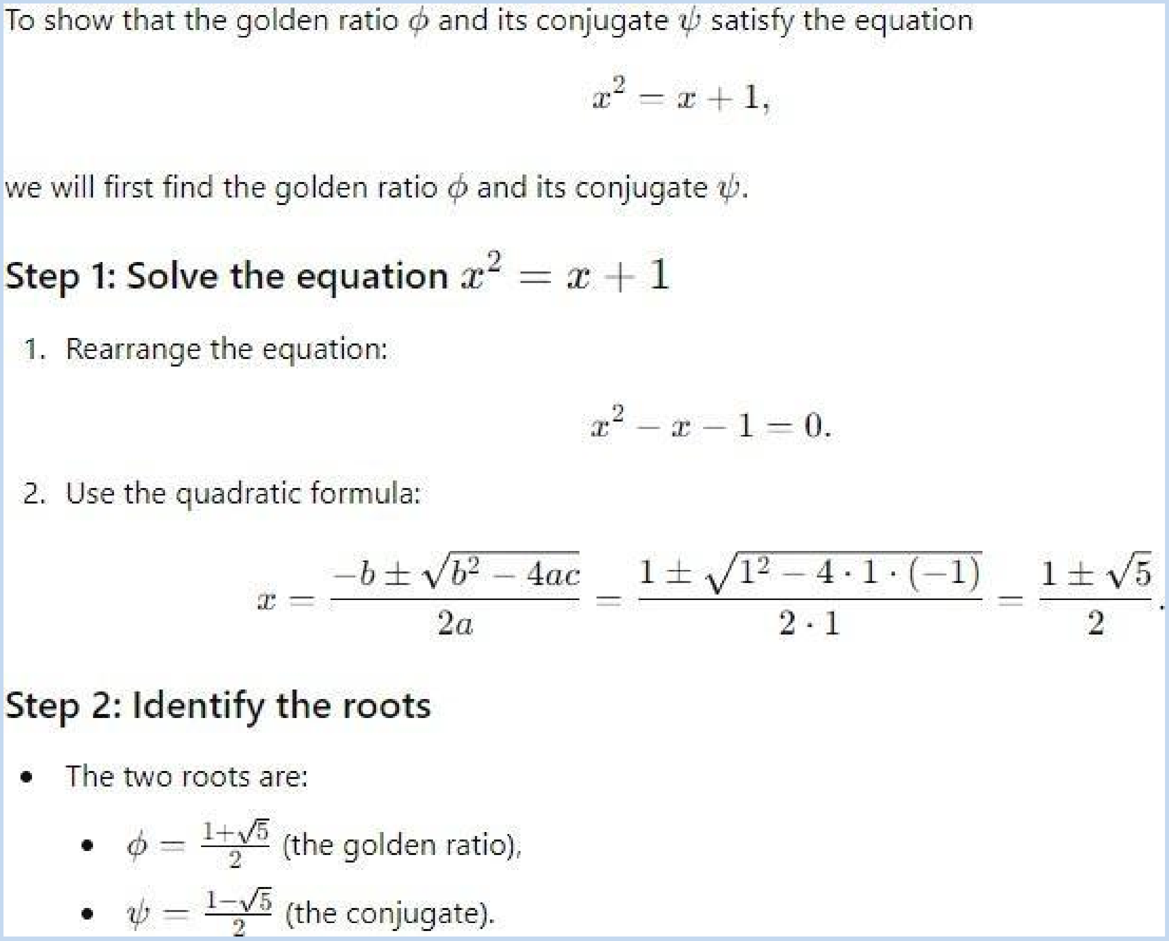

★ *3.3-6*
Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?

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



3.3-7
Show that the golden ratio ϕ and its conjugate $\hat{\phi}$ both satisfy the equation $x^2 = x + 1$.

ANS:



To show that the golden ratio ϕ and its conjugate ψ satisfy the equation

$$x^2 = x + 1,$$

we will first find the golden ratio ϕ and its conjugate ψ.

Step 1: Solve the equation $x^2 = x + 1$

1. Rearrange the equation:

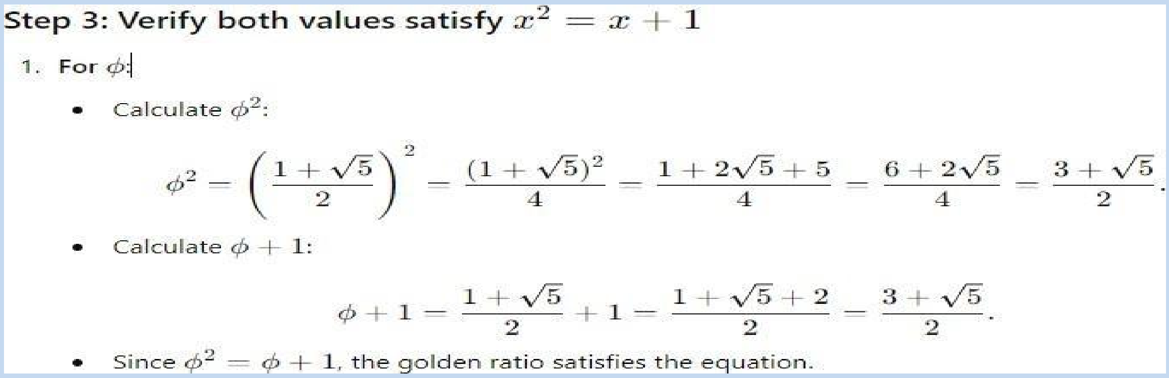
$$x^2 - x - 1 = 0.$$

2. Use the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{1 \pm \sqrt{1^2 - 4 \cdot 1 \cdot (-1)}}{2 \cdot 1} = \frac{1 \pm \sqrt{5}}{2}.$$

Step 2: Identify the roots

- The two roots are:
  - $\phi = \frac{1+\sqrt{5}}{2}$ (the golden ratio),
  - $\psi = \frac{1-\sqrt{5}}{2}$ (the conjugate).



Step 3: Verify both values satisfy $x^2 = x + 1$

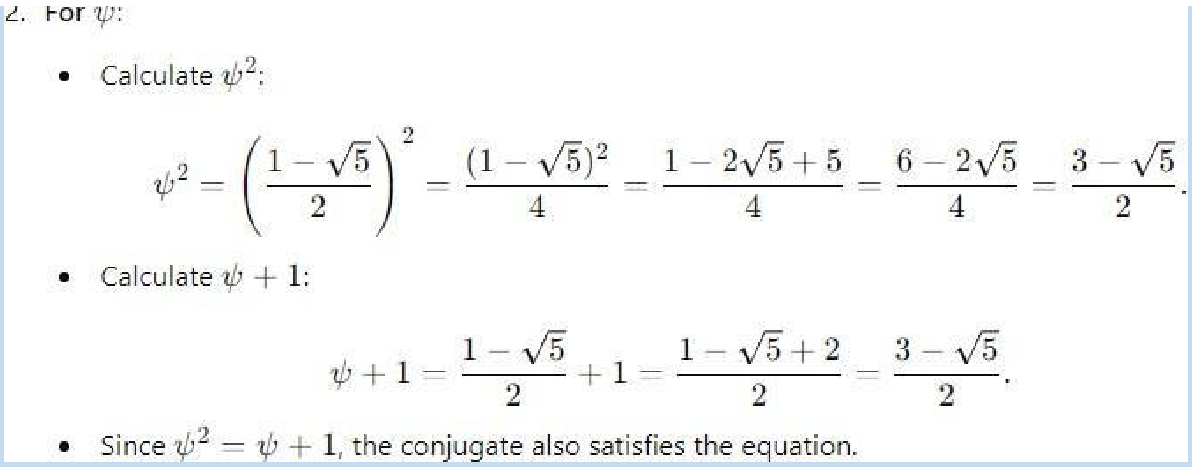
1. For ϕ:
   - Calculate ϕ^2:

$$\phi^2 = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{(1+\sqrt{5})^2}{4} = \frac{1+2\sqrt{5}+5}{4} = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2}.$$

   - Calculate $\phi + 1$:

$$\phi + 1 = \frac{1+\sqrt{5}}{2} + 1 = \frac{1+\sqrt{5}+2}{2} = \frac{3+\sqrt{5}}{2}.$$

   - Since $\phi^2 = \phi + 1$, the golden ratio satisfies the equation.


2. For ψ:

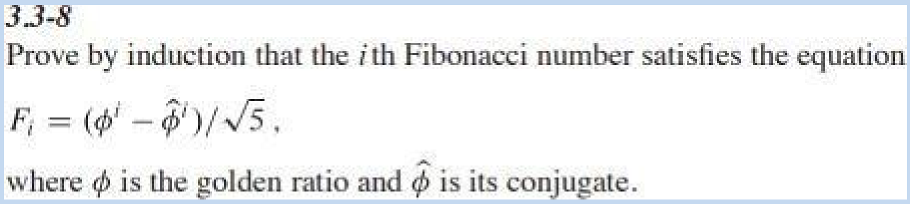
- Calculate ψ^2:

$$\psi^2 = \left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{(1-\sqrt{5})^2}{4} = \frac{1-2\sqrt{5}+5}{4} = \frac{6-2\sqrt{5}}{4} = \frac{3-\sqrt{5}}{2}.$$

- Calculate $\psi + 1$:

$$\psi + 1 = \frac{1-\sqrt{5}}{2} + 1 = \frac{1-\sqrt{5}+2}{2} = \frac{3-\sqrt{5}}{2}.$$

- Since $\psi^2 = \psi + 1$, the conjugate also satisfies the equation.

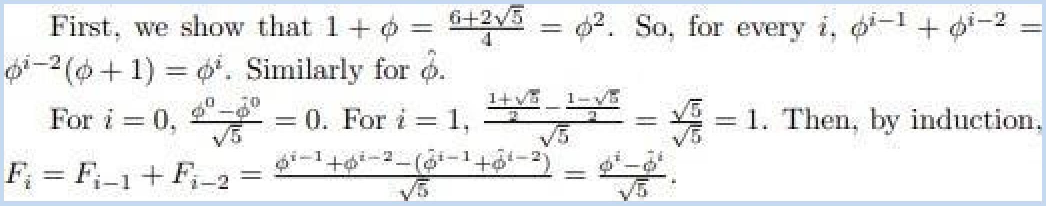

3.3-8

Prove by induction that the ith Fibonacci number satisfies the equation

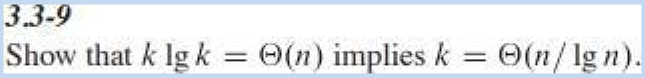
$$F_i = (\phi^i - \hat{\phi}^i)/\sqrt{5},$$

where ϕ is the golden ratio and $\hat{\phi}$ is its conjugate.

ANS:

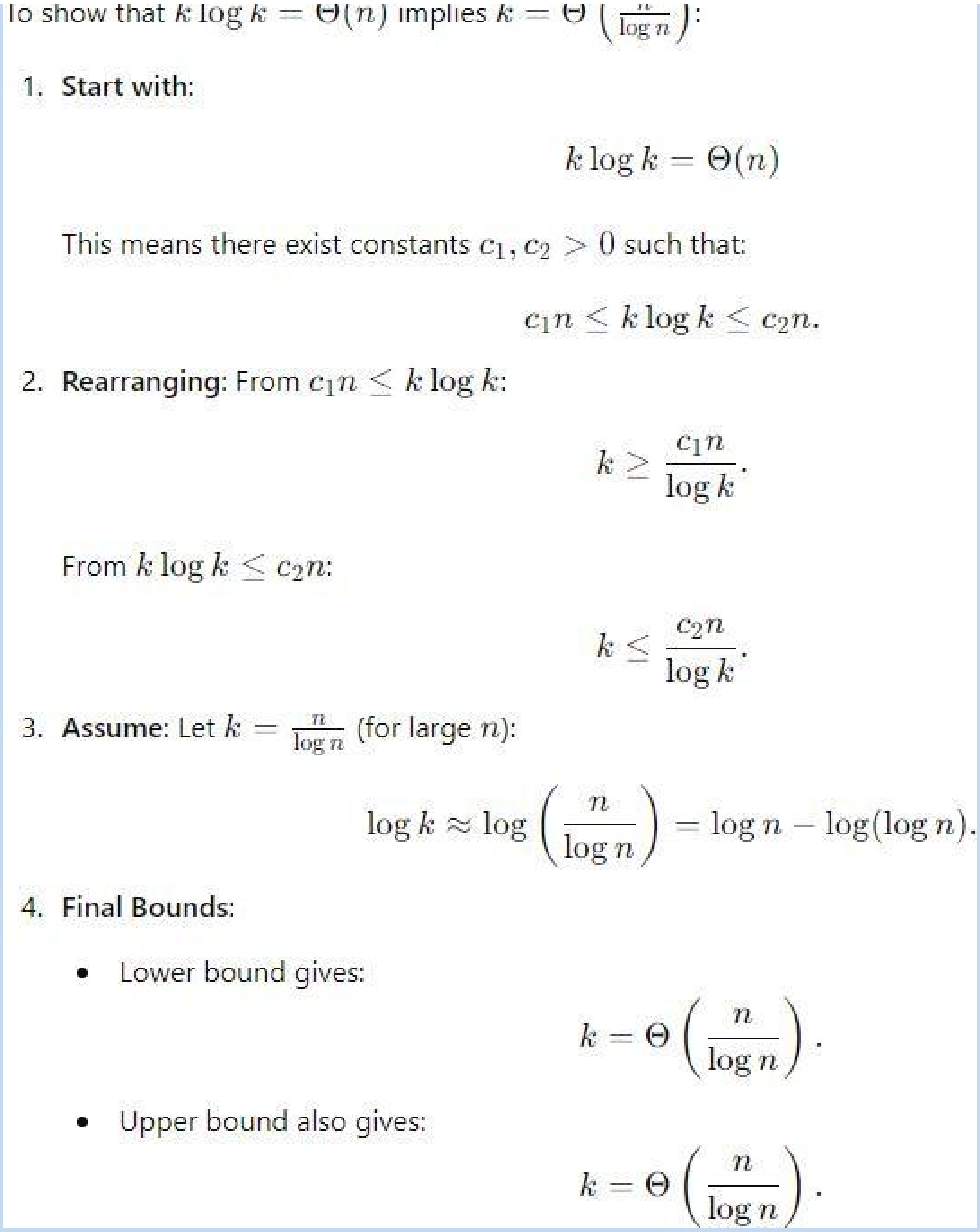

First, we show that $1+\phi = \frac{6+2\sqrt{5}}{4} = \phi^2$. So, for every i, $\phi^{i-1} + \phi^{i-2} = \phi^{i-2}(\phi+1) = \phi^i$. Similarly for $\hat{\phi}$.

For $i=0$, $\frac{\phi^0 - \hat{\phi}^0}{\sqrt{5}} = 0$. For $i=1$, $\frac{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}}{\sqrt{5}} = \frac{\sqrt{5}}{\sqrt{5}} = 1$. Then, by induction, $F_i = F_{i-1} + F_{i-2} = \frac{\phi^{i-1}+\phi^{i-2}-(\hat{\phi}^{i-1}+\hat{\phi}^{i-2})}{\sqrt{5}} = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$.


3.3-9

Show that $k \lg k = \Theta(n)$ implies $k = \Theta(n/\lg n)$.

ANS:



To show that $k \log k = \Theta(n)$ implies $k = \Theta\left(\frac{n}{\log n}\right)$:

1. **Start with:**

$$k \log k = \Theta(n)$$

   This means there exist constants $c_1, c_2 > 0$ such that:

$$c_1 n \leq k \log k \leq c_2 n.$$

2. **Rearranging**: From $c_1 n \leq k \log k$:

$$k \geq \frac{c_1 n}{\log k}.$$

   From $k \log k \leq c_2 n$:

$$k \leq \frac{c_2 n}{\log k}.$$

3. **Assume**: Let $k = \frac{n}{\log n}$ (for large n):

$$\log k \approx \log\left(\frac{n}{\log n}\right) = \log n - \log(\log n).$$

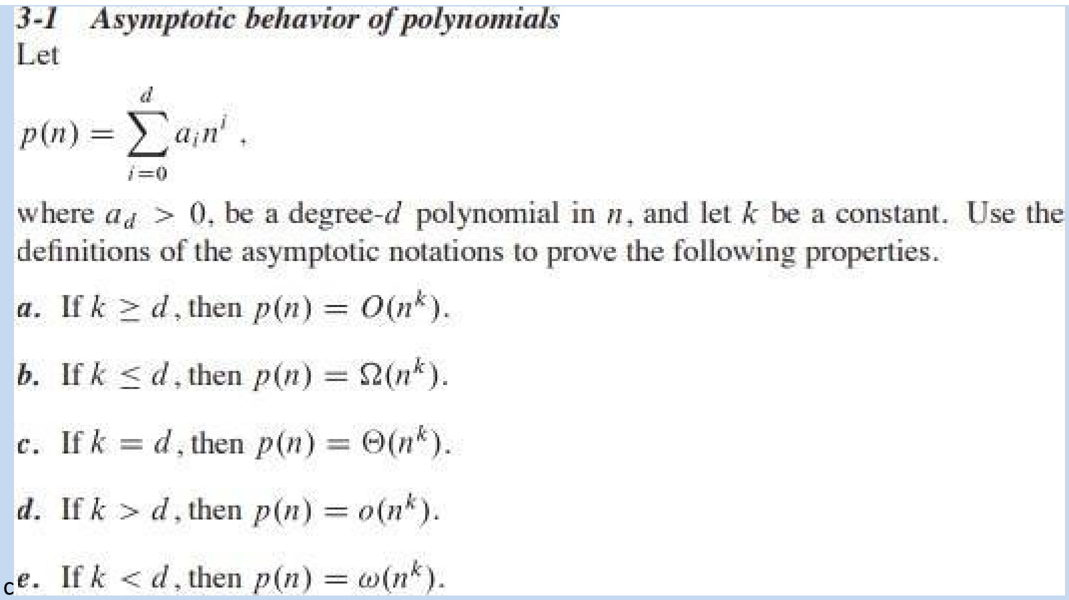
4. **Final Bounds:**

   - Lower bound gives:

$$k = \Theta\left(\frac{n}{\log n}\right).$$

   - Upper bound also gives:

$$k = \Theta\left(\frac{n}{\log n}\right).$$


3-1 Asymptotic behavior of polynomials

Let

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

where $a_d > 0$, be a degree-d polynomial in n, and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties.

a. If $k \geq d$, then $p(n) = O(n^k)$.

b. If $k \leq d$, then $p(n) = \Omega(n^k)$.

c. If $k = d$, then $p(n) = \Theta(n^k)$.

d. If $k > d$, then $p(n) = o(n^k)$.

c ***e.*** If $k < d$, then $p(n) = \omega(n^k)$.

ANS(a): a. If $<$, then ( ) = ( ).

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

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

Now, if $<$, we can find a constant such that:

() ≤ ≤ ′ for some ′ > 0 and sufficiently large

where ′ = ⁻. Thus, we can conclude that () = ().

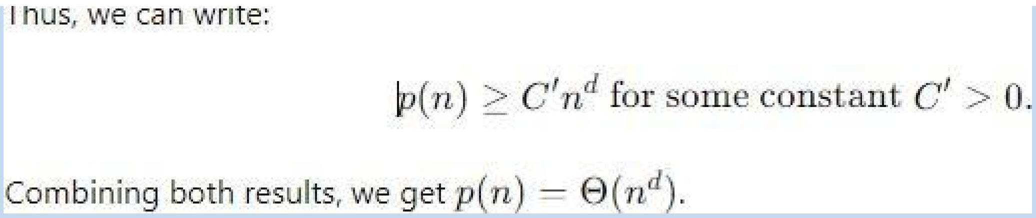
ANS(b):

If = , then () = Θ().

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

() ≥ for sufficiently large .

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

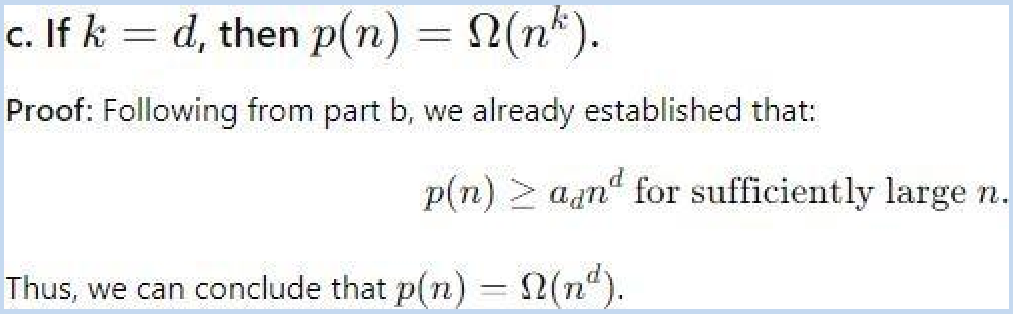


Thus, we can write:

$$p(n) \geq C'n^d \text{ for some constant } C' > 0.$$

Combining both results, we get $p(n) = \Theta(n^d)$.

ANS(C):



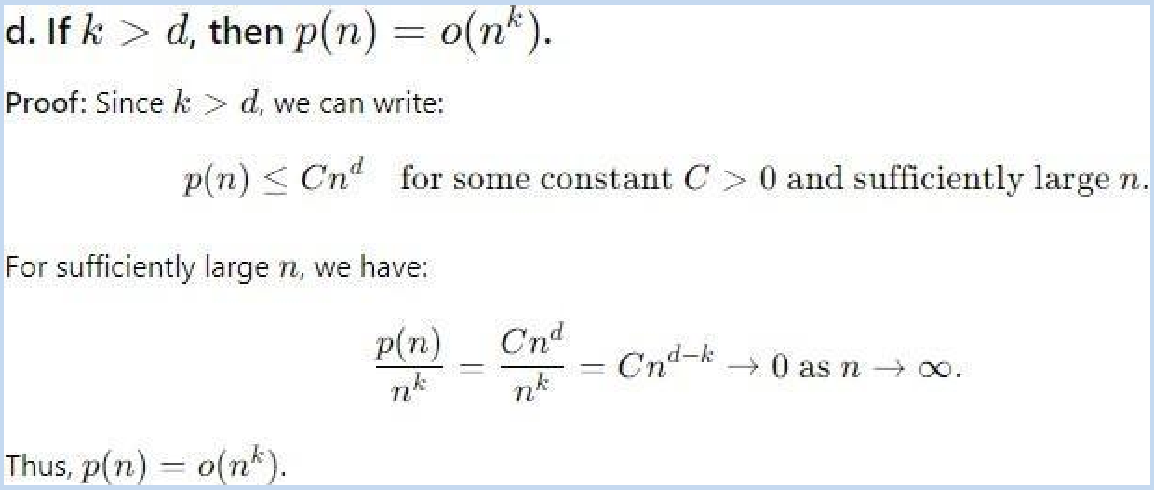
c. If $k = d$, then $p(n) = \Omega(n^k)$.

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

$$p(n) \geq a_d n^d \text{ for sufficiently large } n.$$

Thus, we can conclude that $p(n) = \Omega(n^d)$.

ANS(D):



d. If $k > d$, then $p(n) = o(n^k)$.

Proof: Since $k > d$, we can write:

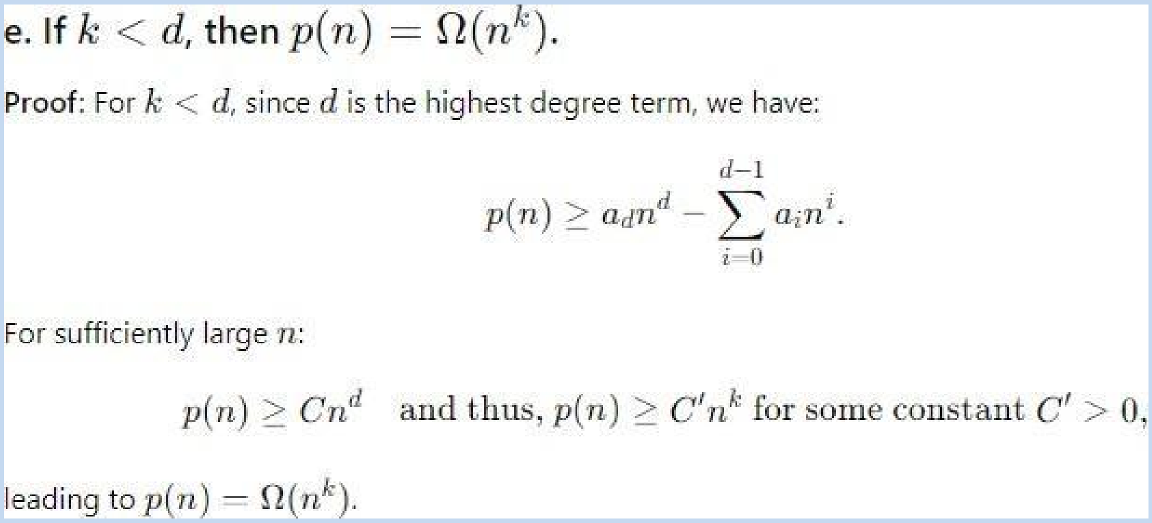
$$p(n) \leq Cn^d \quad \text{for some constant } C > 0 \text{ and sufficiently large } n.$$

For sufficiently large n, we have:

$$\frac{p(n)}{n^k} = \frac{Cn^d}{n^k} = Cn^{d-k} \to 0 \text{ as } n \to \infty.$$

Thus, $p(n) = o(n^k)$.

ANS(e):



e. If $k < d$, then $p(n) = \Omega(n^k)$.

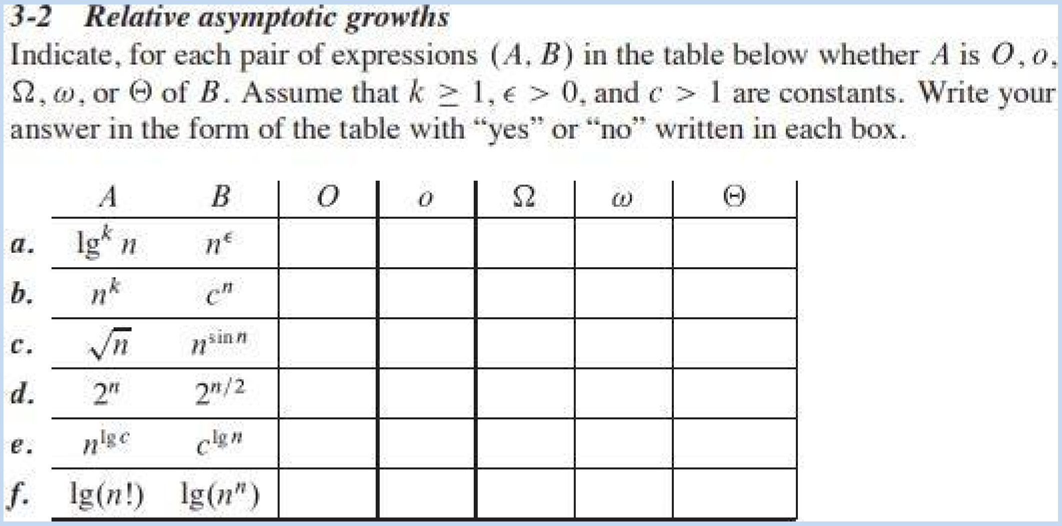
Proof: For $k < d$, since d is the highest degree term, we have:

$$p(n) \geq a_d n^d - \sum_{i=0}^{d-1} a_i n^i.$$

For sufficiently large n:

$$p(n) \geq Cn^d \quad \text{and thus, } p(n) \geq C'n^k \text{ for some constant } C' > 0,$$

leading to $p(n) = \Omega(n^k)$.

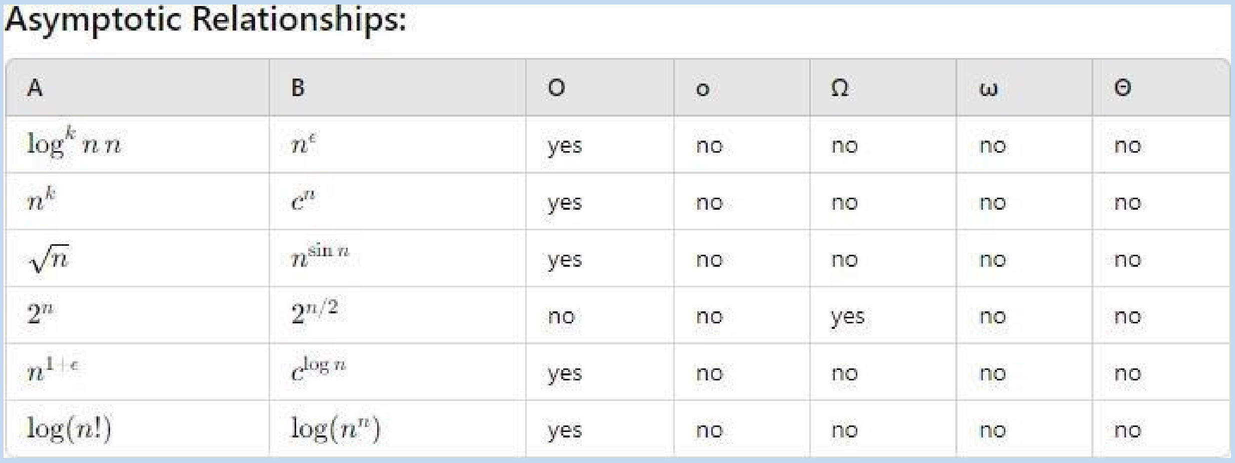


3-2 Relative asymptotic growths

Indicate, for each pair of expressions (A, B) in the table below whether A is O, o, Ω, ω, or Θ of B. Assume that $k \geq 1$, $\epsilon > 0$, and $c > 1$ are constants. Write your answer in the form of the table with "yes" or "no" written in each box.

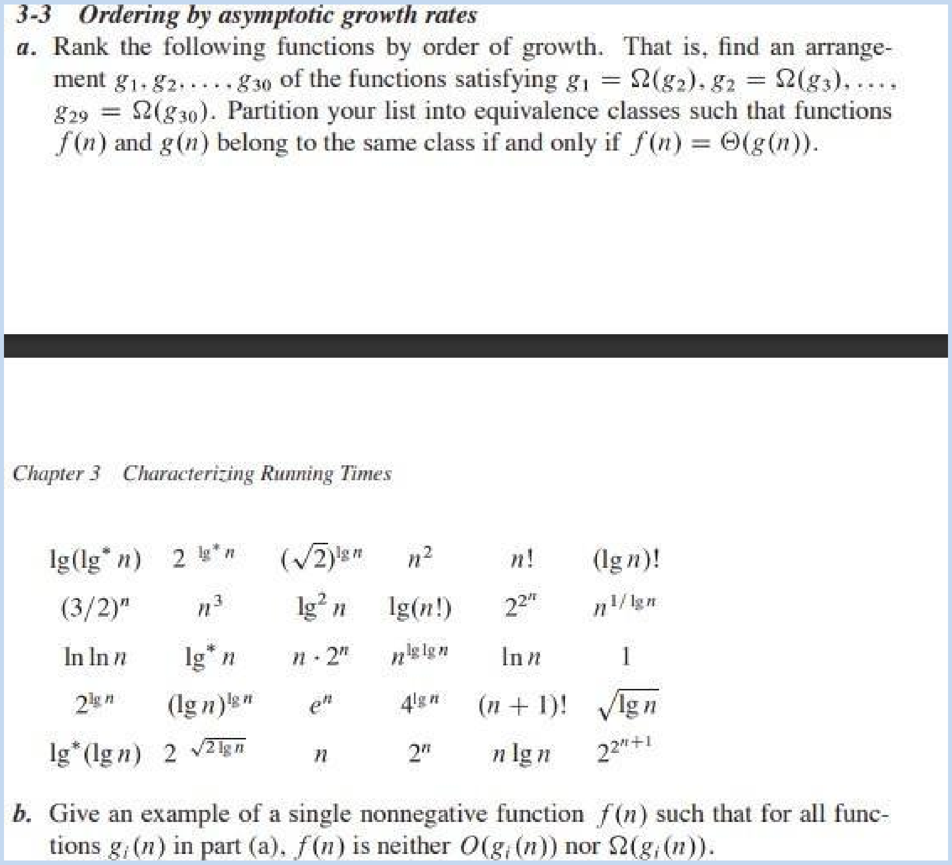
	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

ANS:



Asymptotic Relationships:

A	B	O	o	Ω	ω	Θ
$\log^k n\, n$	n^ϵ	yes	no	no	no	no
n^k	c^n	yes	no	no	no	no
$\sqrt{n}$	$n^{\sin n}$	yes	no	no	no	no
2^n	$2^{n/2}$	no	no	yes	no	no
$n^{1+\epsilon}$	$c^{\log n}$	yes	no	no	no	no
$\log(n!)$	$\log(n^n)$	yes	no	no	no	no



3-3 Ordering by asymptotic growth rates

a. Rank the following functions by order of growth. That is, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions satisfying $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that functions $f(n)$ and $g(n)$ belong to the same class if and only if $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(3/2)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

b. Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.

ANS(A):

$2^{2^{n+1}}$	
2^{2^n}	
$(n+1)!$	
$n!$	
$n2^n$	
e^n	
2^n	
$\left(\frac{3}{2}\right)^n$	
$(\lg(n))!$	
$n^{\lg(\lg(n))}$	$\lg(n)^{\lg(n)}$
n^3	
n^2	$4^{\lg(n)}$
$n\lg(n)$	$\lg(n!)$
$2^{\lg(n)}$	n
$(\sqrt{2})^{\lg(n)}$	
$2^{\sqrt{2\lg(n)}}$	
$\lg^2(n)$	
$\ln(n)$	
$\sqrt{\lg(n)}$	
$\ln(\ln(n))$	
$2^{\lg^*(n)}$	
$\lg^*(n)$	$\lg^*(\lg(n))$
$\lg(\lg^*(n)$	
1	$n^{1/\lg(n)}$

ANS(b):

If we define the function

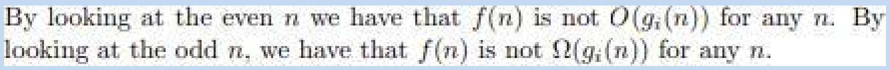
$$f(n) = \begin{cases} g_1(n)! & n \mod 2 = 0 \\ \frac{1}{n} & n \mod 2 = 1 \end{cases}$$

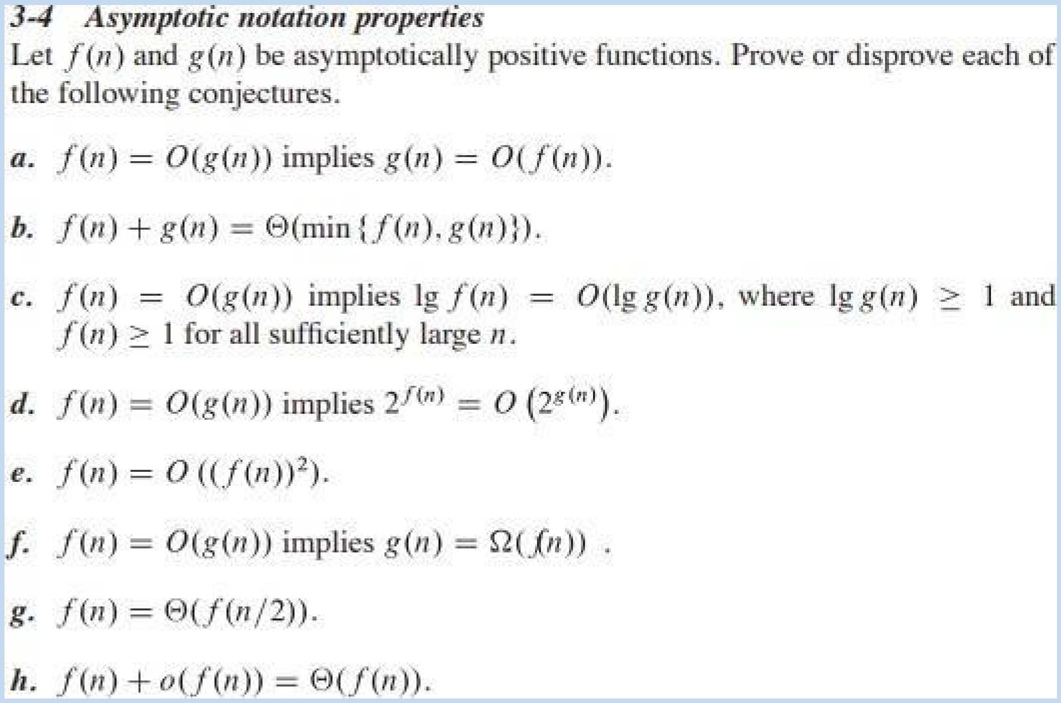
Note that f(n) meets the asymptotically positive requirement that this chapter puts on the functions analyzed. Then, for even n, we have

$$\begin{aligned} \lim_{n\to\infty} \frac{f(2n)}{g_i(2n)} &\geq \lim_{n\to\infty} \frac{f(2n)}{g_1(2n)} \\ &= \lim_{n\to\infty} (g_1(2n)-1)! \\ &= \infty \end{aligned}$$

And for odd n, we have

$$\begin{aligned} \lim_{n\to\infty} \frac{f(2n+1)}{g_i(2n+1)} &\leq \lim_{n\to\infty} \frac{f(2n+1)}{1} \\ &= \lim_{n\to\infty} \frac{1}{2n+1} \\ &= 0 \end{aligned}$$


By looking at the even n we have that $f(n)$ is not $O(g_i(n))$ for any n. By looking at the odd n, we have that $f(n)$ is not $\Omega(g_i(n))$ for any n.


3-4 *Asymptotic notation properties*

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.

a. $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

b. $f(n) + g(n) = \Theta(\min\{f(n), g(n)\})$.

c. $f(n) = O(g(n))$ implies $\lg f(n) = O(\lg g(n))$, where $\lg g(n) \geq 1$ and $f(n) \geq 1$ for all sufficiently large n.

d. $f(n) = O(g(n))$ implies $2^{f(n)} = O\left(2^{g(n)}\right)$.

e. $f(n) = O\left((f(n))^2\right)$.

f. $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

ANs(a):

a. **False**
b. **False**
c. **True**
d. **True**
e. **False**
f. **False**
g. **True**
h. **True**

3-5 Manipulating asymptotic notation

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove the following identities:

a. $\Theta(\Theta(f(n))) = \Theta(f(n))$.

b. $\Theta(f(n)) + O(f(n)) = \Theta(f(n))$.

c. $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$.

d. $\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$.

e. Argue that for any real constants $a_1, b_1 > 0$ and integer constants k_1, k_2, the following asymptotic bound holds:

$$(a_1 n)^{k_1} \lg^{k_2}(a_2 n) = \Theta(n^{k_1} \lg^{k_2} n) .$$

★ ***f.*** Prove that for $S \subseteq \mathbb{Z}$, we have

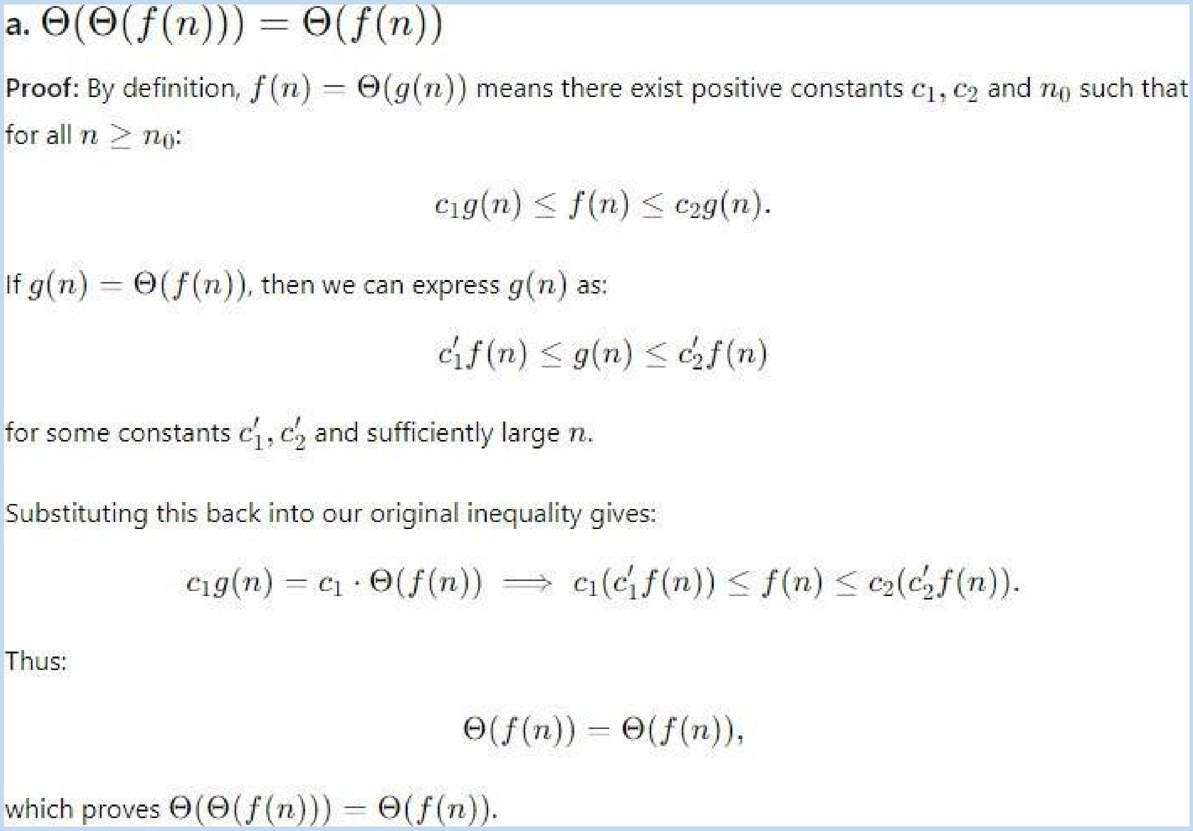
$$\sum_{k \in S} \Theta(f(k)) = \Theta\left(\sum_{k \in S} f(k)\right) ,$$

assuming that both sums converge.

★ ***g.*** Show that for $S \subseteq \mathbb{Z}$, the following asymptotic bound does not necessarily hold, even assuming that both products converge, by giving a counterexample:

$$\prod_{k \in S} \Theta(f(k)) = \Theta\left(\prod_{k \in S} f(k)\right) .$$

ANS(a):



a. $\Theta(\Theta(f(n))) = \Theta(f(n))$

Proof: By definition, $f(n) = \Theta(g(n))$ means there exist positive constants c_1, c_2 and n_0 such that for all $n \geq n_0$:

$$c_1 g(n) \leq f(n) \leq c_2 g(n).$$

If $g(n) = \Theta(f(n))$, then we can express $g(n)$ as:

$$c_1' f(n) \leq g(n) \leq c_2' f(n)$$

for some constants c_1', c_2' and sufficiently large n.

Substituting this back into our original inequality gives:

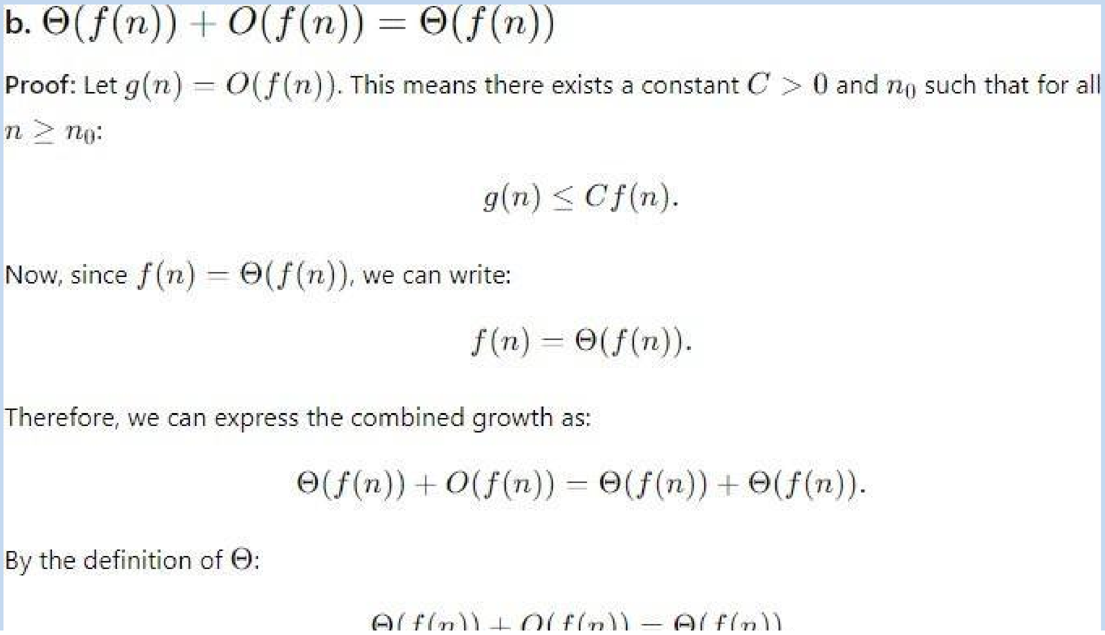
$$c_1 g(n) = c_1 \cdot \Theta(f(n)) \implies c_1(c_1' f(n)) \leq f(n) \leq c_2(c_2' f(n)).$$

Thus:

$$\Theta(f(n)) = \Theta(f(n)),$$

which proves $\Theta(\Theta(f(n))) = \Theta(f(n))$.

ANS(b):



b. $\Theta(f(n)) + O(f(n)) = \Theta(f(n))$

Proof: Let $g(n) = O(f(n))$. This means there exists a constant $C > 0$ and n_0 such that for all $n \geq n_0$:

$$g(n) \leq C f(n).$$

Now, since $f(n) = \Theta(f(n))$, we can write:

$$f(n) = \Theta(f(n)).$$

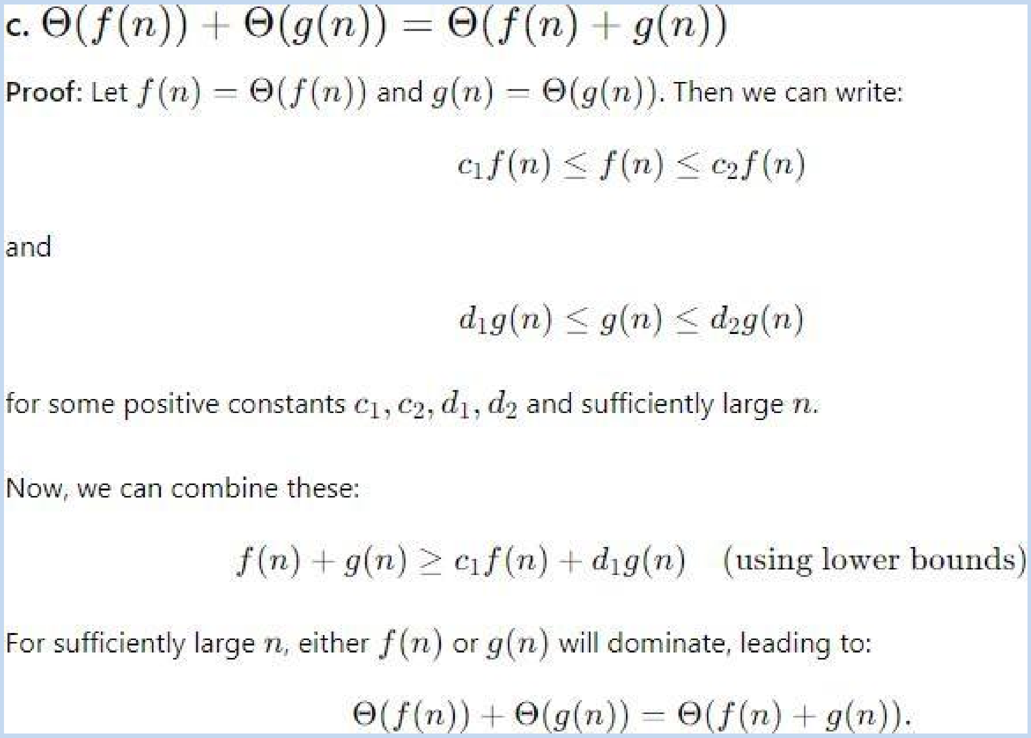
Therefore, we can express the combined growth as:

$$\Theta(f(n)) + O(f(n)) = \Theta(f(n)) + \Theta(f(n)).$$

By the definition of Θ:

$$\Theta(f(n)) + O(f(n)) = \Theta(f(n))$$

ANS(C):



c. $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$

Proof: Let $f(n) = \Theta(f(n))$ and $g(n) = \Theta(g(n))$. Then we can write:

$$c_1 f(n) \leq f(n) \leq c_2 f(n)$$

and

$$d_1 g(n) \leq g(n) \leq d_2 g(n)$$

for some positive constants c_1, c_2, d_1, d_2 and sufficiently large n.

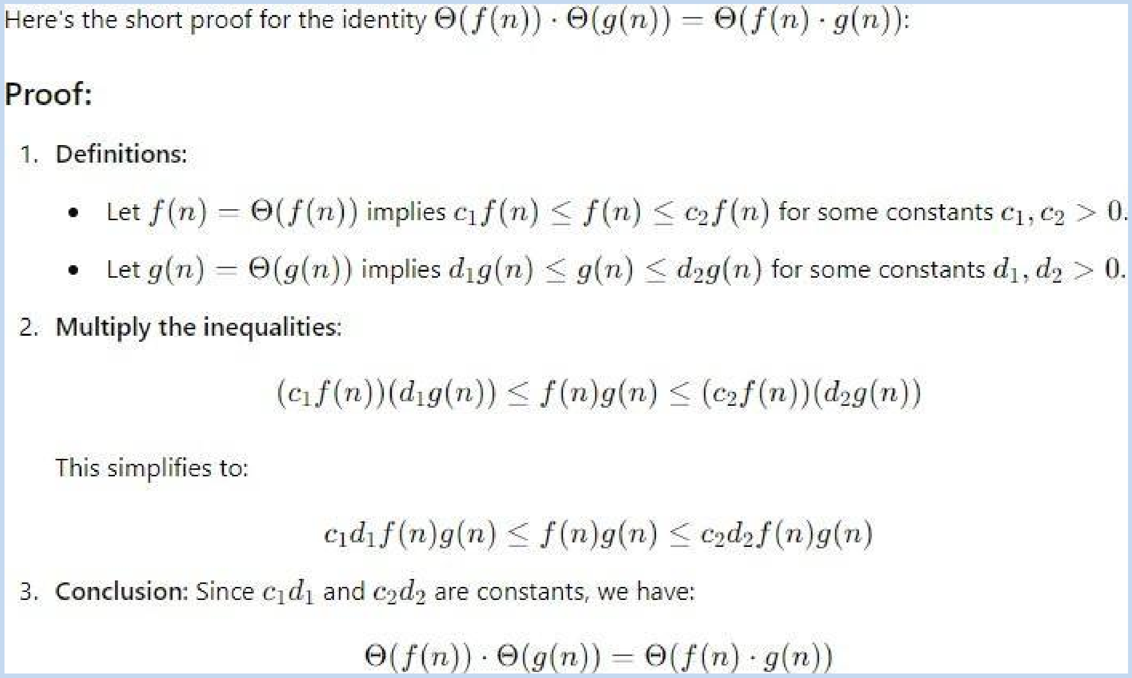
Now, we can combine these:

$$f(n) + g(n) \geq c_1 f(n) + d_1 g(n) \quad \text{(using lower bounds)}$$

For sufficiently large n, either $f(n)$ or $g(n)$ will dominate, leading to:

$$\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n)).$$

ANS(D):



Here's the short proof for the identity $\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$:

Proof:

1. **Definitions:**
   - Let $f(n) = \Theta(f(n))$ implies $c_1 f(n) \leq f(n) \leq c_2 f(n)$ for some constants $c_1, c_2 > 0$.
   - Let $g(n) = \Theta(g(n))$ implies $d_1 g(n) \leq g(n) \leq d_2 g(n)$ for some constants $d_1, d_2 > 0$.
2. **Multiply the inequalities:**

   $$(c_1 f(n))(d_1 g(n)) \leq f(n)g(n) \leq (c_2 f(n))(d_2 g(n))$$

   This simplifies to:

   $$c_1 d_1 f(n) g(n) \leq f(n) g(n) \leq c_2 d_2 f(n) g(n)$$
3. **Conclusion:** Since $c_1 d_1$ and $c_2 d_2$ are constants, we have:

   $$\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$$

ANS(e):

e. Asymptotic Bound

Claim: For any real constants $a_1, b_1 > 0$ and integer constants k_1, k_2, the following asymptotic bound holds:

$$(a_1 n^{k_1} \log^{k_2}(a_2 n)) = \Theta(n^{k_1} \log^{k_2} n).$$

Starting with the left-hand side:

$$a_1 n^{k_1} \log^{k_2}(a_2 n) = a_1 n^{k_1} (\log(a_2) + \log(n))^{k_2}.$$

Using the Binomial Theorem:

$$(\log(a_2) + \log(n))^{k_2} = \sum_{j=0}^{k_2} \binom{k_2}{j} (\log(a_2))^{k_2 - j} (\log(n))^j$$

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

$$a_1 n^{k_1} \cdot \Theta(\log^{k_2}(n)).$$

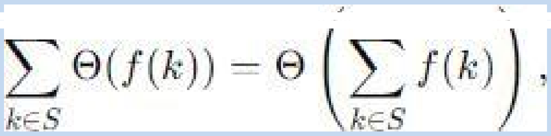
Conclusion:
Thus, we conclude:

$$(a_1 n^{k_1} \log^{k_2}(a_2 n)) = \Theta(n^{k_1} \log^{k_2}(n)).$$

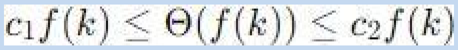
ANS(f):

Summation Identity

Claim: For S⊆ZS we have:

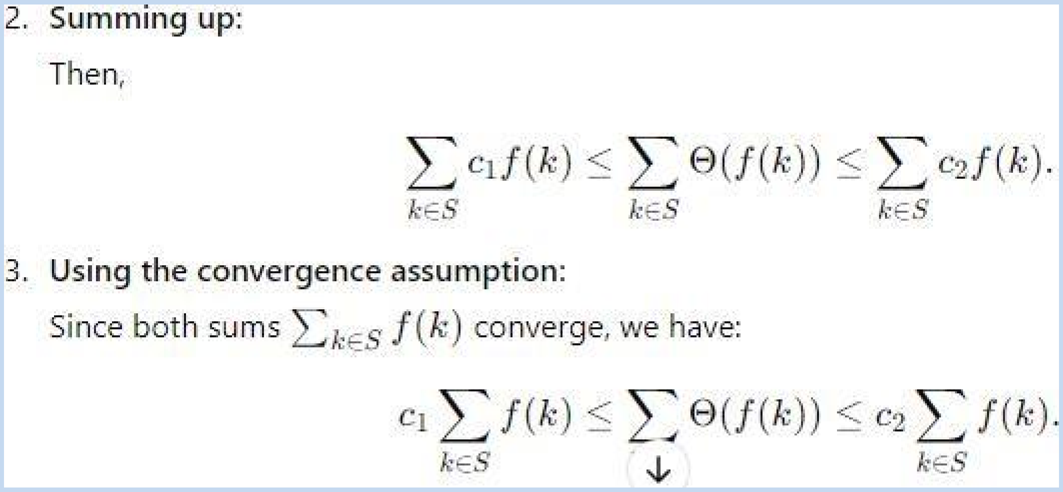


$$\sum_{k\in S}\Theta(f(k)) = \Theta\left(\sum_{k\in S} f(k)\right),$$



$$c_1 f(k) \leq \Theta(f(k)) \leq c_2 f(k)$$

for some constants c1,c2>0



2. **Summing up:**

   Then,

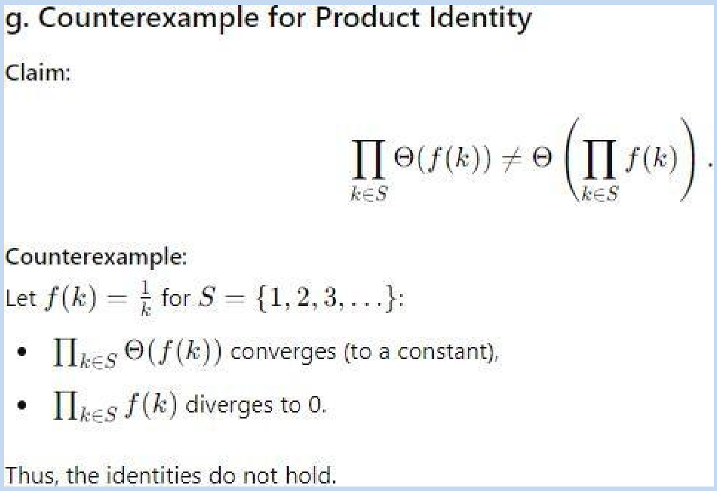
$$\sum_{k\in S} c_1 f(k) \leq \sum_{k\in S}\Theta(f(k)) \leq \sum_{k\in S} c_2 f(k).$$

3. **Using the convergence assumption:**

   Since both sums $\sum_{k\in S} f(k)$ converge, we have:

$$c_1\sum_{k\in S} f(k) \leq \sum_{k\in S}\Theta(f(k)) \leq c_2\sum_{k\in S} f(k).$$

ANS(g):



g. Counterexample for Product Identity

Claim:

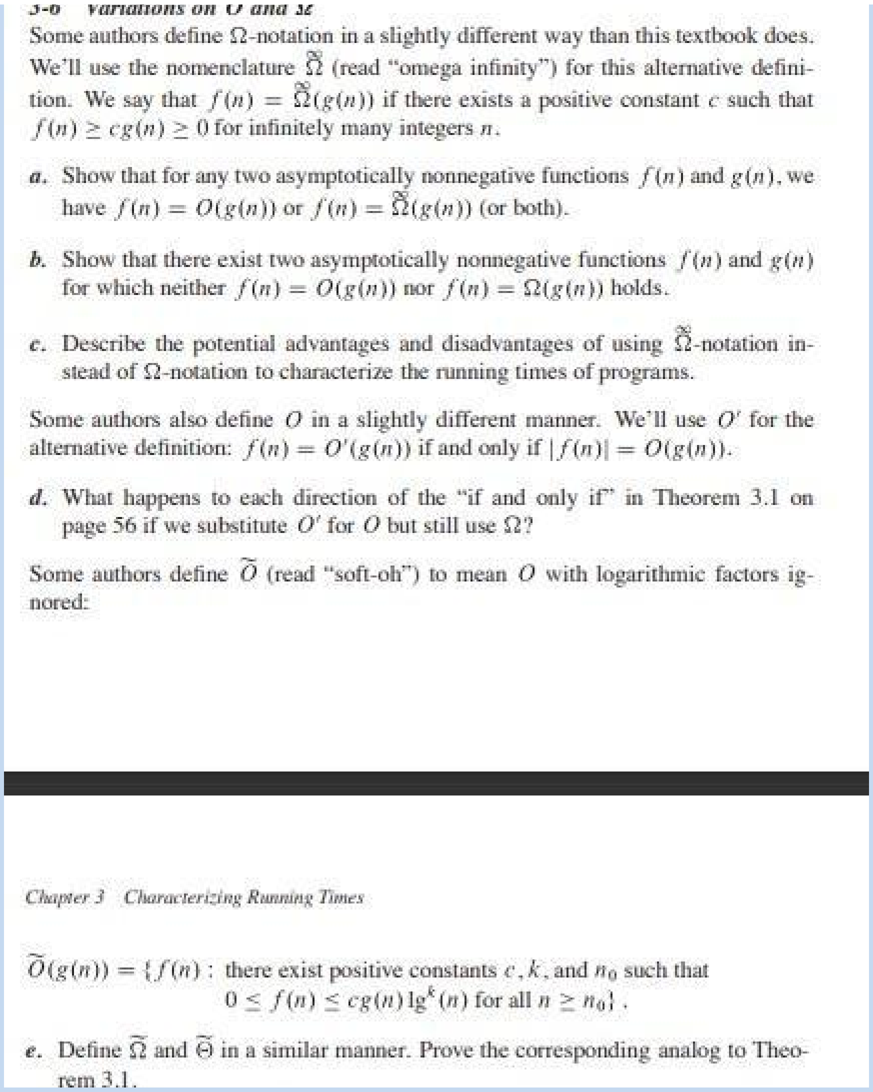
$$\prod_{k\in S}\Theta(f(k)) \neq \Theta\left(\prod_{k\in S} f(k)\right).$$

Counterexample:

Let $f(k) = \frac{1}{k}$ for $S = \{1, 2, 3, \ldots\}$:

- $\prod_{k\in S}\Theta(f(k))$ converges (to a constant),
- $\prod_{k\in S} f(k)$ diverges to 0.

Thus, the identities do not hold.



3-6 Variations on O and Ω

Some authors define Ω-notation in a slightly different way than this textbook does. We'll use the nomenclature $\overset{\infty}{\Omega}$ (read "omega infinity") for this alternative definition. We say that $f(n) = \overset{\infty}{\Omega}(g(n))$ if there exists a positive constant c such that $f(n) \geq cg(n) \geq 0$ for infinitely many integers n.

a. Show that for any two asymptotically nonnegative functions $f(n)$ and $g(n)$, we have $f(n) = O(g(n))$ or $f(n) = \overset{\infty}{\Omega}(g(n))$ (or both).

b. Show that there exist two asymptotically nonnegative functions $f(n)$ and $g(n)$ for which neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds.

c. Describe the potential advantages and disadvantages of using $\overset{\infty}{\Omega}$-notation instead of Ω-notation to characterize the running times of programs.

Some authors also define O in a slightly different manner. We'll use O' for the alternative definition: $f(n) = O'(g(n))$ if and only if $|f(n)| = O(g(n))$.

d. What happens to each direction of the "if and only if" in Theorem 3.1 on page 56 if we substitute O' for O but still use Ω?

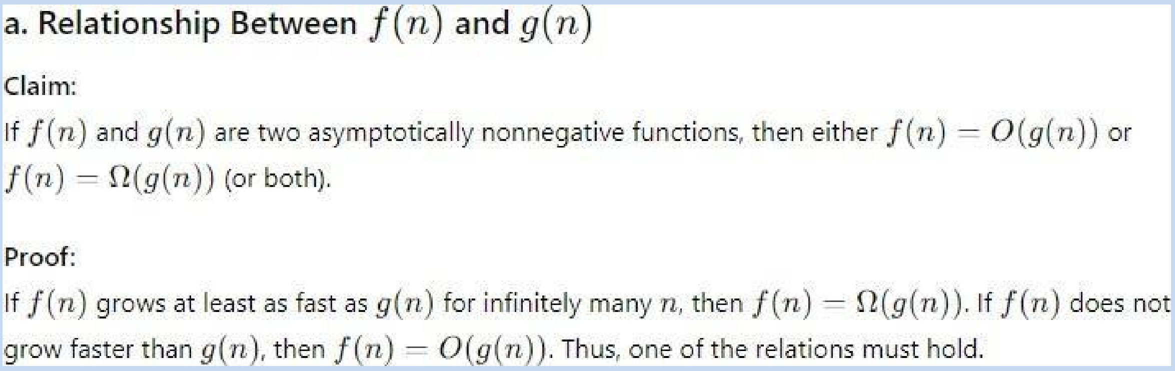
Some authors define $\widetilde{O}$ (read "soft-oh") to mean O with logarithmic factors ignored:

Chapter 3 Characterizing Running Times

$$\widetilde{O}(g(n)) = \{f(n) : \text{there exist positive constants } c, k, \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n)\lg^k(n) \text{ for all } n \geq n_0\}.$$

e. Define $\widetilde{\Omega}$ and $\widetilde{\Theta}$ in a similar manner. Prove the corresponding analog to Theorem 3.1.

ANS(a):



a. Relationship Between $f(n)$ and $g(n)$

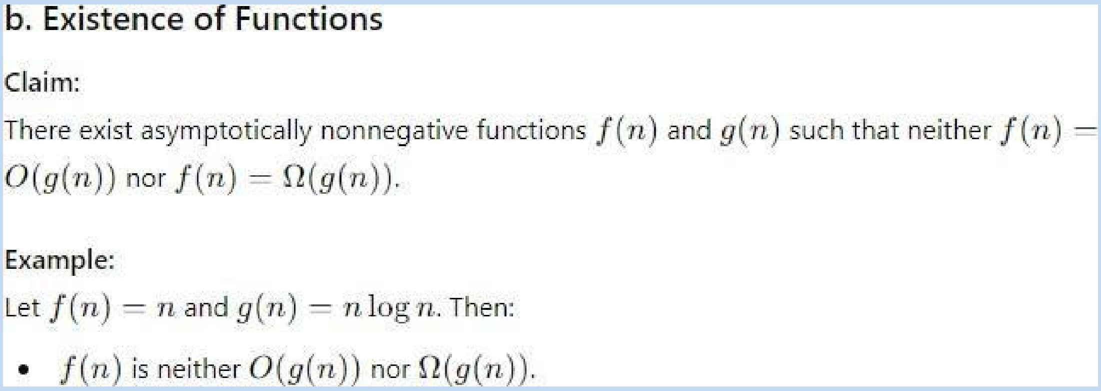
Claim:

If $f(n)$ and $g(n)$ are two asymptotically nonnegative functions, then either $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$ (or both).

Proof:

If $f(n)$ grows at least as fast as $g(n)$ for infinitely many n, then $f(n) = \Omega(g(n))$. If $f(n)$ does not grow faster than $g(n)$, then $f(n) = O(g(n))$. Thus, one of the relations must hold.

ANS(b):



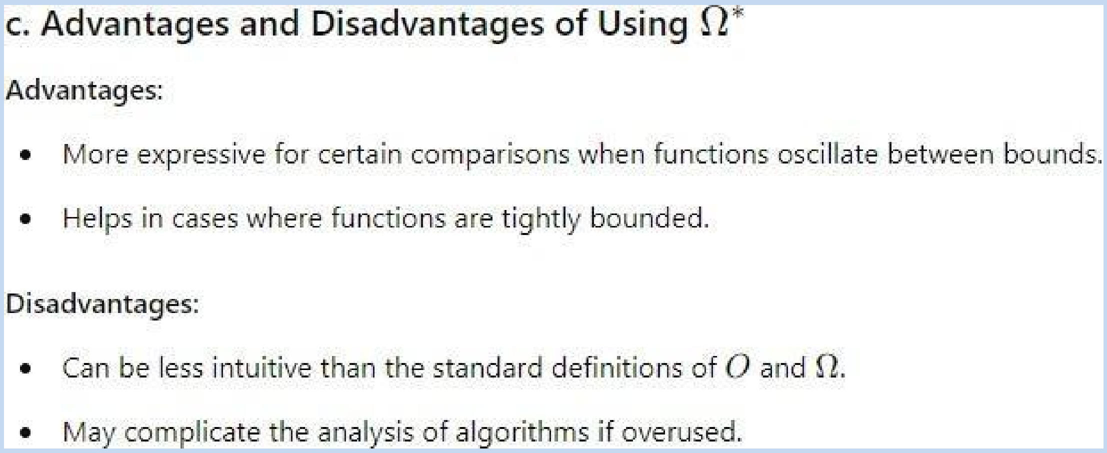
b. Existence of Functions

Claim:
There exist asymptotically nonnegative functions $f(n)$ and $g(n)$ such that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$.

Example:
Let $f(n) = n$ and $g(n) = n \log n$. Then:

- $f(n)$ is neither $O(g(n))$ nor $\Omega(g(n))$.

ANS(c):



c. Advantages and Disadvantages of Using Ω^*

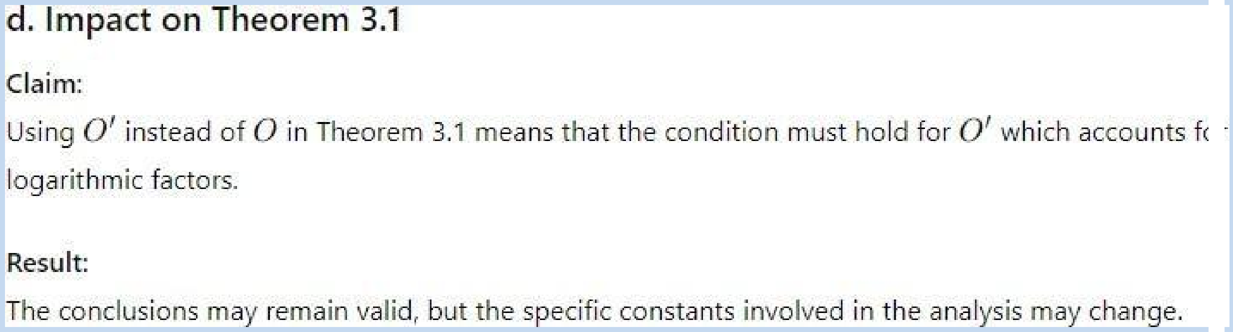
Advantages:

- More expressive for certain comparisons when functions oscillate between bounds.
- Helps in cases where functions are tightly bounded.

Disadvantages:

- Can be less intuitive than the standard definitions of O and Ω.
- May complicate the analysis of algorithms if overused.

ANS(d):



d. Impact on Theorem 3.1

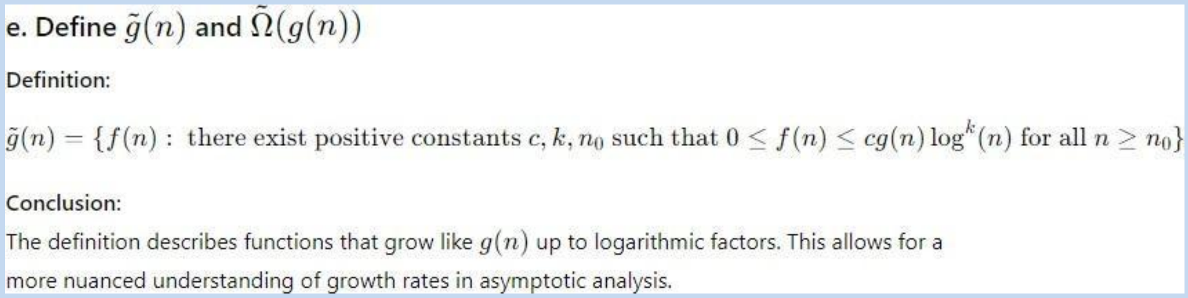
Claim:

Using O' instead of O in Theorem 3.1 means that the condition must hold for O' which accounts f
logarithmic factors.

Result:

The conclusions may remain valid, but the specific constants involved in the analysis may change.

ANS(e):



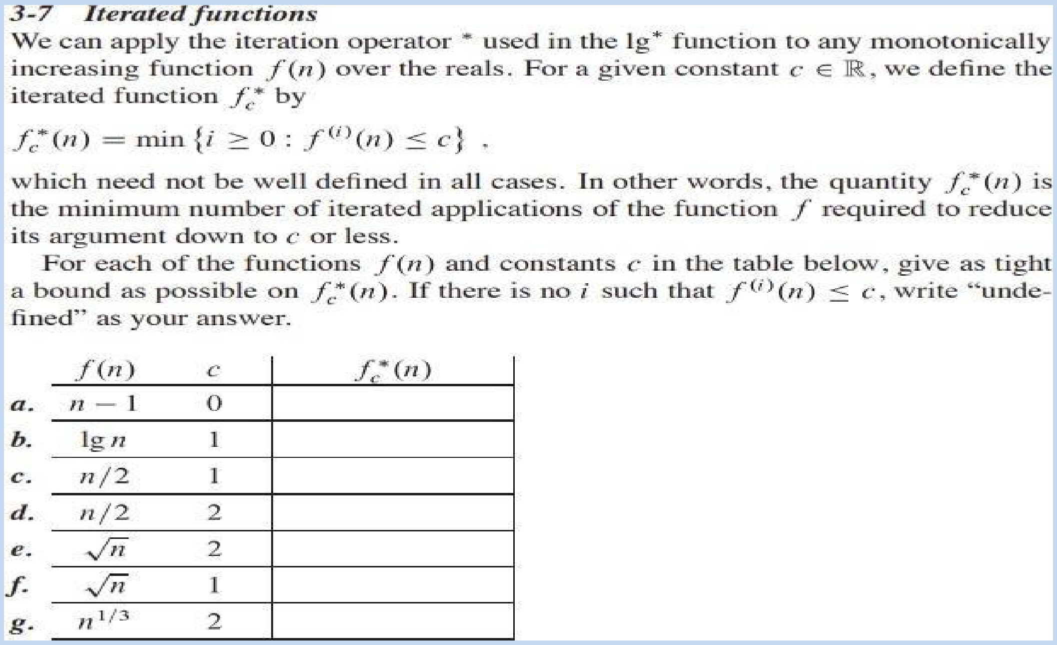
e. Define $\tilde{g}(n)$ and $\tilde{\Omega}(g(n))$

Definition:

$$\tilde{g}(n) = \{f(n) : \text{ there exist positive constants } c, k, n_0 \text{ such that } 0 \leq f(n) \leq cg(n)\log^k(n) \text{ for all } n \geq n_0\}$$

Conclusion:

The definition describes functions that grow like $g(n)$ up to logarithmic factors. This allows for a more nuanced understanding of growth rates in asymptotic analysis.



3-7 Iterated functions

We can apply the iteration operator * used in the $\lg^*$ function to any monotonically increasing function $f(n)$ over the reals. For a given constant $c \in \mathbb{R}$, we define the iterated function f_c^* by

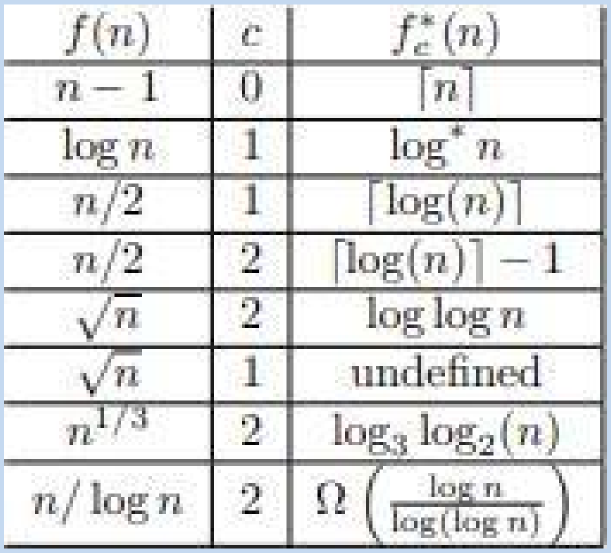
$$f_c^*(n) = \min\{i \geq 0 : f^{(i)}(n) \leq c\} ,$$

which need not be well defined in all cases. In other words, the quantity $f_c^*(n)$ is the minimum number of iterated applications of the function f required to reduce its argument down to c or less.

For each of the functions $f(n)$ and constants c in the table below, give as tight a bound as possible on $f_c^*(n)$. If there is no i such that $f^{(i)}(n) \leq c$, write "undefined" as your answer.

	$f(n)$	c	$f_c^*(n)$
a.	$n-1$	0	
b.	$\lg n$	1	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	

ANS:



$f(n)$	c	$f_c^*(n)$
$n-1$	0	$\lceil n \rceil$
$\log n$	1	$\log^* n$
$n/2$	1	$\lceil \log(n) \rceil$
$n/2$	2	$\lceil \log(n) \rceil - 1$
$\sqrt{n}$	2	$\log \log n$
$\sqrt{n}$	1	undefined
$n^{1/3}$	2	$\log_3 \log_2(n)$
$n/\log n$	2	$\Omega\left(\frac{\log n}{\log(\log n)}\right)$