Name Noorulain
Roll no 14664
Assignment no 1
Data Structure

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

1.1-1: Sorting and Shortest Distance Examples

Sorting Example: A real-world scenario that requires sorting can be seen in project management, where tasks need to be organized based on priority levels. High-priority tasks are sorted to appear first, ensuring they are addressed before lower-priority ones. This ordering helps teams stay focused on the most critical tasks, improving productivity and meeting deadlines.

Shortest Distance Example: An example of finding the shortest distance between two points is found in GPS navigation systems. When a user inputs a destination, the system calculates the shortest or fastest route, considering distance and traffic conditions to minimize travel time. This application is essential for efficiency in transportation and logistics.

Q2 Other than speed, what other measures of efûciency might you need to consider in a real-world setting?

1.1-2: Efficiency Measures Beyond Speed

Other than speed, real-world efficiency considerations may include memory usage, energy consumption, and ease of implementation. Memory usage is crucial in environments with limited storage, such as mobile devices. Energy efficiency is essential for battery-operated systems, such as smartphones or IoT devices. Ease of implementation and maintenance is also a factor, especially for large or complex systems where ongoing updates are required.

Q3 Select a data structure that you have seen, and discuss its strengths and limitations.

1.1-3: Data Structure Strengths and Limitations

One useful data structure is the hash table, which offers fast data retrieval using key-value pairs. Its primary strength is the ability to access data in constant time, making it highly efficient for lookup operations. However, hash tables can suffer from hash collisions, where two keys produce the same hash, leading to reduced performance. They also require careful memory management, as resizing can be costly.

Q4 How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

1.1-4: Shortest-Path vs. Traveling-Salesperson Problems

The shortest-path and traveling-salesperson problems are similar in that both aim to find an optimal route. In the shortest-path problem, the goal is to determine the minimum distance between two points. In contrast, the traveling-salesperson problem seeks the shortest route that visits multiple points (cities) exactly once, returning to the starting point. While the shortest-path problem is computationally simpler, the traveling-salesperson problem is more complex and requires considering all possible routes for an optimal solution.

Q5 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.

1.1-5: Exact vs. Approximate Solutions

A real-world problem where only the best solution will suffice is medical treatment planning. Here, achieving the optimal outcome is critical for patient health and recovery. By contrast, an example where an approximate solution is sufficient is in image compression. While perfect fidelity is unnecessary, a close approximation that maintains image quality while reducing file size is typically acceptable.

Q6 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.

1.1-6: Input Availability for Real-World Problems

An example of a real-world problem where input availability varies is stock trading analysis. Sometimes, traders have all the historical data beforehand, allowing for comprehensive analysis. Other times, data arrives in real-time as market conditions change, requiring on-the-fly decision-making based on streaming data.

-1 Comparison of running times For each function f.n/ and time t in the following table, determine the largest size n of a problem that can be solved in time t , assuming that the algorithm to solve the problem takes f.n/ microseconds.

Function	1 second	1 minute	1 hour	1 day
log n\log nlogn	21,062,106	26×10726 \times 10^726×107	23.6×10923.6 \times 10^923.6×109	28.64×101028.64 \times 10^{10}28.64×1010

Function	1 second	1 minute	1 hour	1 day
nnn	10,121,012	3.6×10153.6 \times 10^{15}3.6×1015	1.3×10191.3 \times 10^{19}1.3×1019	7.46×10217.46 \times 10^{21}7.46×1021
n\sqrt{n}n	106,106	6×1076 \times 10^76×107	3.6×1093.6 \times 10^93.6×109	8.64×10108.64 \times 10^{10}8.64×1010
nlog nn \log nnlogn	6.24×1046.24 \times 10^46.24×104	2.8×1062.8 \times 10^62.8×106	1.33×1081.33 \times 10^81.33×108	2.76×1092.76 \times 10^92.76×109
n2n^2n2	1,000	7,745	60,000	293,938
n3n^3n3	100	391	1,532	4,420

Function	1 second	1 minute	1 hour	1 day
2n2^n2n	19	25	31	36
n!n!n!	9	11	12	13

CHAPTER NO 2

2.1-1 Using Figure 2.2 as a model, illustrate the operation of I NSERTION-SORT on an array initially containing the sequence h31;41;59;26;41;58 i .

The **Insertion Sort** algorithm works by building a sorted portion of the array, one element at a time, by repeatedly inserting the next element into its proper position among the previously sorted elements.

Initial Array:

Given array: [31, 41, 59, 26, 41, 58]

Let's walk through each iteration of the Insertion Sort algorithm.

Step-by-Step Insertion Sort Process

Step 1: Start with the first element (31)

- No action is needed because there's only one element.
- **Array after Step 1:** [31, 41, 59, 26, 41, 58]

Step 2: Insert 41

- 41 is compared to 31. Since 41 > 31, it remains in place.
- **Array after Step 2:** [31, 41, 59, 26, 41, 58]

Step 3: Insert 59

- 59 is compared to 41 and 31. Since 59 > 41, it remains in place.
- **Array after Step 3:** [31, 41, 59, 26, 41, 58]

Step 4: Insert 26

- 26 is compared to 59, 41, and 31, and is moved to the start of the sorted portion.

- **Array after Step 4:** [26, 31, 41, 59, 41, 58]

Step 5: Insert 41 (second occurrence)

- The second 41 is compared to 59 and is inserted before it.
- **Array after Step 5:** [26, 31, 41, 41, 59, 58]

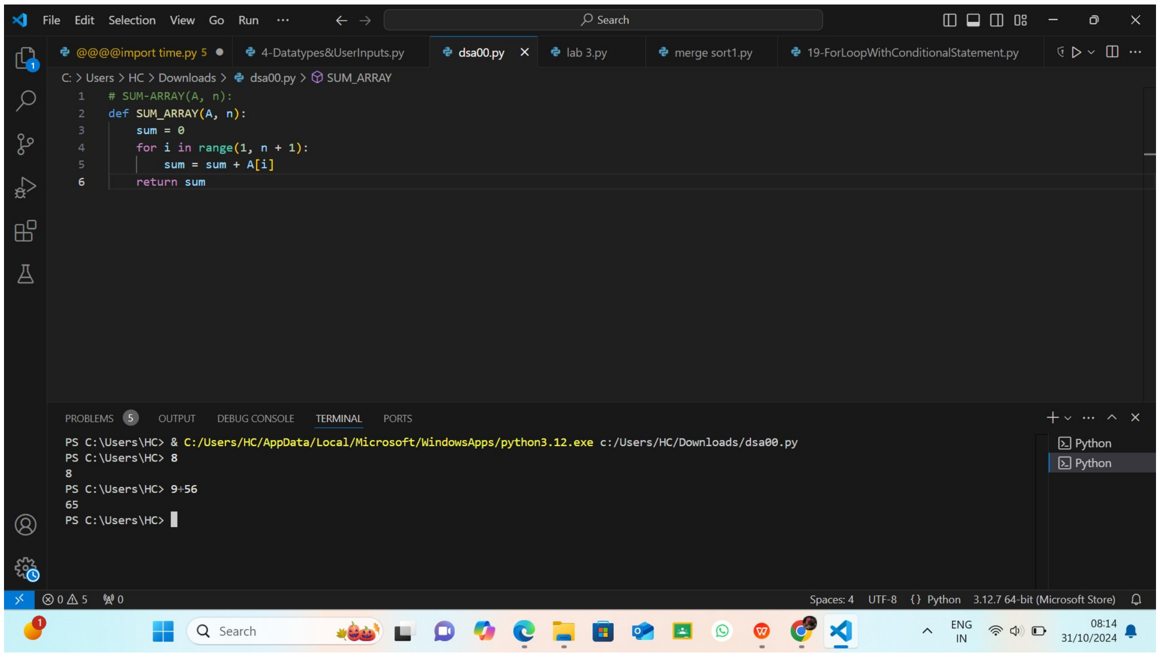
Step 6: Insert 58

- 58 is compared to 59 and is inserted before it.
- **Array after Step 6 (Final Sorted Array):** [26, 31, 41, 41, 58, 59]

Final Result

The array is now sorted in ascending order: [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 AŒ1 W 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 AŒ1 W n� .


File Edit Selection View Go Run
Search
@@@@import time.py 5
4-Datatypes&UserInputs.py
dsa00.py
lab 3.py
merge sort1.py
19-ForLoopWithConditionalStatement.py
C: > Users > HC > Downloads > dsa00.py > SUM_ARRAY
PROBLEMS 5
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Python
Python
Spaces: 4 UTF-8 {} Python 3.12.7 64-bit (Microsoft Store)
Search
ENG IN
08:14 31/10/2024

```python
# SUM-ARRAY(A, n):
def SUM_ARRAY(A, n):
    sum = 0
    for i in range(1, n + 1):
        sum = sum + A[i]
    return sum
```

```
PS C:\Users\HC> & C:/Users/HC/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/HC/Downloads/dsa00.py
PS C:\Users\HC> 8
8
PS C:\Users\HC> 9+56
65
PS C:\Users\HC> 
```

Loop Invariant

A **loop invariant** is a condition that holds true before and after each iteration of a loop. For this procedure, let's define the loop invariant as follows:

> At the start of each iteration of the loop (for any index iii), the variable sum contains the sum of the elements A[1]A[1]A[1] through A[i−1]A[i-1]A[i−1].

Proof of Correctness

To prove that the **SUM-ARRAY** procedure correctly returns the sum of the array, we will use the **Initialization**, **Maintenance**, and **Termination** properties.

1. Initialization

- **Before the first iteration** (when i=1i = 1i=1), the value of sum is initialized to 0.
- Since there are no elements before A[1]A[1]A[1], the sum of the elements A[1]A[1]A[1] through A[i−1]A[i-1]A[i−1] (which is an empty sum in this case) is indeed 0.
- Therefore, the loop invariant holds true before the loop starts.

2. Maintenance

- **During each iteration**, the algorithm adds the current element A[i]A[i]A[i] to sum.
- After updating sum, it now contains the sum of elements A[1]A[1]A[1] through A[i]A[i]A[i], thus maintaining the loop invariant for the next iteration (where iii increments by 1).
- This ensures that at the end of each iteration, sum correctly represents the sum of all elements up to the current index iii.

3. Termination

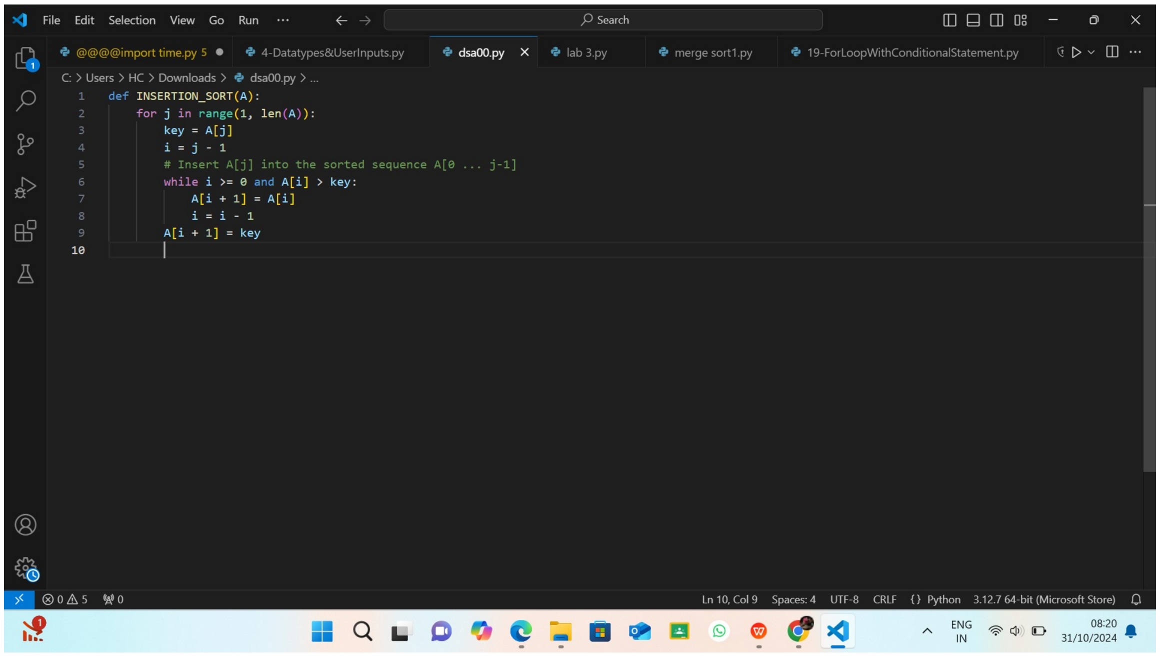
- The loop terminates after i=n+1i = n + 1i=n+1, meaning it has iterated through all nnn elements of AAA.
- At this point, by the loop invariant, sum contains the sum of A[1]A[1]A[1] through A[n]A[n]A[n], which is the sum of the entire array.
- Therefore, the procedure returns the correct sum of all elements in AAA.

Conclusion

Since the **SUM-ARRAY** procedure satisfies the initialization, maintenance, and termination conditions of the loop invariant, we conclude that it correctly computes the sum of the nnn elements in the array A[1…n]A[1 \ldots n]A[1…n].

2.1-3 Rewrite the I NSERTION-SORT procedure to sort into monotonically decreasing in- stead of monotonically increasing order.

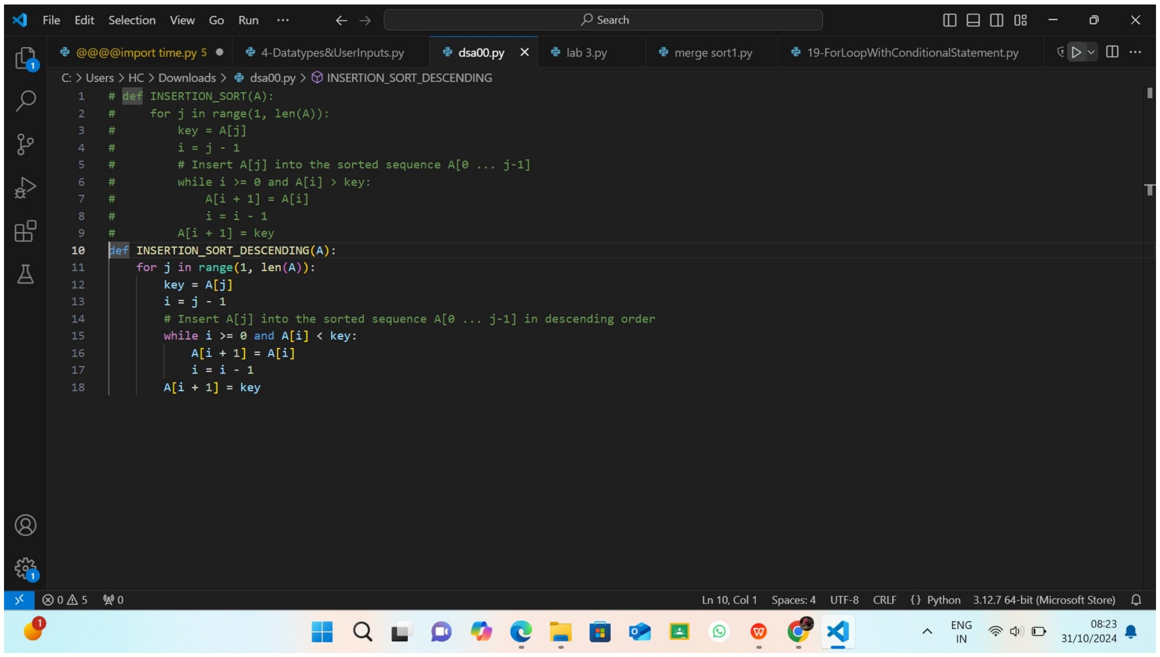
To modify the **INSERTION-SORT** procedure to sort an array in **monotonically decreasing** order (i.e., in descending order), we only need to make a slight change in the comparison within the algorithm.



```python
def INSERTION_SORT(A):
    for j in range(1, len(A)):
        key = A[j]
        i = j - 1
        # Insert A[j] into the sorted sequence A[0 ... j-1]
        while i >= 0 and A[i] > key:
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

In this original version, the while loop checks if A[i] > key to keep elements in ascending order. To modify it for **descending order**, we simply change the comparison to A[i] < key.

Modified INSERTION-SORT for Descending Order



```python
# def INSERTION_SORT(A):
#     for j in range(1, len(A)):
#         key = A[j]
#         i = j - 1
#         # Insert A[j] into the sorted sequence A[0 ... j-1]
#         while i >= 0 and A[i] > key:
#             A[i + 1] = A[i]
#             i = i - 1
#         A[i + 1] = key
def INSERTION_SORT_DESCENDING(A):
    for j in range(1, len(A)):
        key = A[j]
        i = j - 1
        # Insert A[j] into the sorted sequence A[0 ... j-1] in descending order
        while i >= 0 and A[i] < key:
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

Explanation of the Changes

- **Comparison Change**: The key modification is changing A[i] > key to A[i] < key in the while loop condition. This alteration ensures that each element is placed in the correct position to achieve descending order.
- **Rest of the Procedure**: The rest of the procedure remains the same as the standard insertion sort.

Example

Let's see how this modified version would work on an array:

For an initial array [31, 41, 59, 26, 41, 58], the algorithm would produce [59, 58, 41, 41, 31, 26] as the sorted array in descending order.

To solve this searching problem, we'll write pseudocode for **Linear Search**, a basic algorithm that scans through an array AAA to find a given value xxx. If xxx is found, it returns the index iii where A[i]=xA[i] = xA[i]=x; otherwise, it returns NIL.

After providing the pseudocode, we'll define a **loop invariant** and use the **initialization**, **maintenance**, and **termination** properties to prove the algorithm's correctness.

Linear Search Pseudocode

The pseudocode for a linear search algorithm that checks each element in the array:

plaintext

Copy code

```
LINEAR-SEARCH(A, n, x)
  for i = 1 to n do
    if A[i] == x then
      return i     // Return the index if x is found
  return NIL          // Return NIL if x is not found in A
```

Explanation of the Algorithm

1. **Input**: An array A[1…n]A[1 \ldots n]A[1…n] containing nnn elements and a target value xxx.

2. **Output**: The index iii such that A[i]=xA[i] = xA[i]=x, or NIL if xxx does not appear in the array.

The algorithm works as follows:

- It starts from the first element of AAA and checks each element sequentially.
- If it finds an element equal to xxx, it immediately returns the index iii of that element.
- If it completes the loop without finding xxx, it returns NIL, indicating that xxx is not in the array.

Proof of Correctness Using Loop Invariant

To prove that **LINEAR-SEARCH** is correct, we'll define a **loop invariant** and demonstrate that it holds by using the **initialization**, **maintenance**, and **termination** properties.

Loop Invariant

Define the loop invariant as follows:

> **Loop Invariant**: At the start of each iteration iii (where 1≤i≤n1 \leq i \leq n1≤i≤n), if xxx is in A[1…i−1]A[1 \ldots i-1]A[1…i−1], then LINEAR-SEARCH will have already returned the index of xxx. Otherwise, xxx is not in A[1…i−1]A[1 \ldots i-1]A[1…i−1].

This invariant means that, by the start of each iteration, if xxx exists in the part of the array that has already been scanned, the algorithm would have returned the correct index. If the algorithm reaches the current index iii without having returned, it implies that xxx is not in A[1…i−1]A[1 \ldots i-1]A[1…i−1].

Proof of Loop Invariant

Initialization:

- Before the first iteration (when i=1i = 1i=1), no elements have been checked yet. The loop invariant holds because, before starting, the algorithm hasn't checked any elements, and therefore hasn't missed any instance of xxx in A[1…0]A[1 \ldots 0]A[1…0], which is an empty range.

- Thus, the loop invariant holds true at the beginning.

Maintenance:

- During each iteration iii, the algorithm checks if A[i]=xA[i] = xA[i]=x.
- If A[i]=xA[i] = xA[i]=x, it returns iii, satisfying the problem requirements, as it has found xxx and returned the correct index.
- If A[i]≠xA[i] \neq xA[i]□ =x, the loop continues to the next index. The loop invariant holds for the next iteration because, if xxx were in A[1…i]A[1 \ldots i]A[1…i], the algorithm would have returned its index.

Thus, the loop invariant is maintained at every step of the loop.

Termination:

- The loop terminates after i=n+1i = n+1i=n+1, meaning the algorithm has checked all nnn elements in AAA.
- If the loop terminates without finding xxx, the algorithm returns NIL, which is correct because, by the loop invariant, xxx does not exist in A[1…n]A[1 \ldots n]A[1…n].
- Thus, the termination condition satisfies the requirements of the problem.

Conclusion

Since the **LINEAR-SEARCH** algorithm satisfies the **initialization**, **maintenance**, and **termination** properties of the loop invariant, we conclude that it correctly returns the index iii where x=A[i]x = A[i]x=A[i] if xxx is present in AAA; otherwise, it returns NIL.

Consider the problem of adding two n-bit binary integers a and b , stored in two n P n □ 1 -element arrays AŒ0 W n □ 1� and BŒ0 W n □ 1� , where each element is either 0 or 1 , a D P n i D0 AŒi� □ 2 i , and b D i D0 BŒi� □ 2 i . The sum c D a C b of the two integers should be stored in binary form in an .n C 1/-element array CŒ0 W n� , where c D i D0 CŒi� □ 2 i . Write a procedure ADD-BINARY-I NTEGERS that takes as input arrays A and B , along with the length n , and returns array C holding the sum.

1. **Input**: Two arrays AAA and BBB, each containing nnn bits, and an integer nnn representing the bit length.
2. **Output**: An array CCC of size n+1n+1n+1, where CCC stores the binary sum of AAA and BBB.

Pseudocode for ADD-BINARY-INTEGERS

plaintext

Copy code

```
ADD-BINARY-INTEGERS(A, B, n)
    // Initialize array C of size n+1 to store the result
    Let C be a new array of size n+1
    carry = 0

    // Perform bit-by-bit addition from the least significant bit to the most significant bit
    for i = 0 to n-1 do
        sum = A[i] + B[i] + carry
        if sum == 2 then
            C[i] = 0
            carry = 1
        else if sum == 3 then
            C[i] = 1
            carry = 1
        else
            C[i] = sum
            carry = 0

    // Set the final carry in the most significant bit
    C[n] = carry

    return C
```

Explanation of the Code

Initialize C **and** carry: We create an array CCC of size n+1n+1n+1 to hold the result, and initialize a variable carry to 0.

Bit-by-Bit Addition:

We iterate over each bit from 0 to n−1n-1n−1 (from least significant to most significant).

For each position iii, we calculate the sum of A[i]A[i]A[i], B[i]B[i]B[i], and carry.

Depending on the value of sum:

1. If sum == 2, this means both bits are 1, so the resulting bit is 0 with a carry of 1.
2. If sum == 3, this means both bits are 1 and we had a carry, so the resulting bit is 1 with a carry of 1.
3. If sum == 0 or sum == 1, we store sum in C[i]C[i]C[i] and set carry to 0.

Final Carry: After the loop, the last element C[n]C[n]C[n] will store the final carry value, which could be 0 or 1 depending on the last addition.

Return CCC: The array CCC contains the binary representation of the sum of AAA and BBB.

Example

Suppose we have two 4-bit binary numbers:

- A=[1,0,1,1]A = [1, 0, 1, 1]A=[1,0,1,1] (binary for 11)
- B=[1,1,0,1]B = [1, 1, 0, 1]B=[1,1,0,1] (binary for 13)

Calling ADD-BINARY-INTEGERS(A, B, 4) would result in:

- C = [0, 0, 0, 1, 1] (binary for 24), as 11+13=2411 + 13 = 2411+13=24.

2.2-1 Express the function n 3 =1000 C 100n 2 □ 100n C 3 in terms of ,-notation.

To express the function f(n)=n3+1000· 100n2+100n+3f(n) = n^3 + 1000 \cdot 100n^2 + 100n + 3f(n)=n3+1000· 100n2+100n+3 in terms of Θ\ThetaΘ-notation, we analyze the growth rates of each term in the function.

Step 1: Break Down the Function

The function is given by:

f(n)=n3+1000· 100n2+100n+3f(n) = n^3 + 1000 \cdot 100n^2 + 100n + 3f(n)=n3+1000· 100n2+100n+3

Each term has a different rate of growth:

- n3n^3n3: The highest-order term, which grows the fastest as nnn becomes large.
- 1000· 100n2=100000n21000 \cdot 100n^2 = 100000n^21000· 100n2=100000n2: Grows slower than n3n^3n3 but faster than nnn.
- 100n100n100n: Grows linearly, slower than both n3n^3n3 and n2n^2n2.
- 333: A constant term, which does not grow as nnn increases.

Step 2: Determine the Dominant Term

In Big-O and Θ\ThetaΘ-notation, only the term with the highest growth rate matters as n→∞n \to \inftyn→∞. In this case, the n3n^3n3 term grows faster than n2n^2n2, nnn, and the constant term.

Thus, as nnn becomes large, the n3n^3n3 term will dominate the behavior of f(n)f(n)f(n).

Step 3: Express in Θ\ThetaΘ-Notation

Since n3n^3n3 is the dominant term, we can express f(n)f(n)f(n) in Θ\ThetaΘ-notation as:

f(n)=Θ(n3)f(n) = \Theta(n^3)f(n)=Θ(n3)

This means that f(n)f(n)f(n) grows asymptotically at the same rate as n3n^3n3. The additional terms 100000n2100000n^2100000n2, 100n100n100n, and 333 become negligible compared to n3n^3n3 as nnn increases.

2.2-2 Consider sorting n numbers stored in array AŒ1 W n� by ûrst ûnding the smallest element of AŒ1 W n� and exchanging it with the element in AŒ1� . Then ûnd the smallest element of AŒ2 W n� , and exchange it with AŒ2� . Then ûnd the smallest element of AŒ3 W n� , and exchange it with AŒ3� . Continue in this manner for the ûrst 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 ûrst n □ 1 elements, rather than for all n elements? Give the worst-case running time of selection sort in ,-notation. Is the best-case running time any better?

Selection sort works by iteratively finding the smallest element in the unsorted portion of the array and swapping it with the first unsorted element. Here's the pseudocode:

plaintext

Copy code

```
SELECTION-SORT(A, n)
  for i = 1 to n - 1 do
    // Find the index of the smallest element in A[i ... n]
    min_index = i
    for j = i + 1 to n do
      if A[j] < A[min_index] then
        min_index = j
```

// Swap the found minimum element with A[i]

swap A[i] with A[min_index]

Explanation of the Algorithm

1.

Outer Loop (for i = 1 to n - 1): Iterates through each position in the array (up to n−1n-1n−1), assuming each position will eventually hold the smallest remaining element from the unsorted part of the array.

2.
3.

Inner Loop (for j = i + 1 to n): Searches for the smallest element in the unsorted portion of the array A[i…n]A[i \ldots n]A[i…n].

4.
5.

Swapping: After finding the minimum element in the unsorted portion, the algorithm swaps it with the element at position iii.

6.

Loop Invariant

The **loop invariant** for this algorithm is as follows:

At the start of each iteration of the outer loop (at index iii), the subarray A[1…i−1]A[1 \ldots i - 1]A[1…i−1] contains the i−1i - 1i−1 smallest elements of AAA in sorted order.

This invariant helps ensure that at each step, the algorithm places the smallest unsorted element into its correct position in the array. After the final iteration of the loop, the entire array A[1…n]A[1 \ldots n]A[1…n] is sorted.

Proof of Correctness Using the Loop Invariant

1.

Initialization: Before the first iteration, the sorted portion A[1…0]A[1 \ldots 0]A[1…0] is empty, so the invariant holds trivially.

2.
3.

Maintenance: Assuming the invariant holds at the beginning of each iteration iii, the algorithm finds the smallest element in the unsorted portion A[i…n]A[i \ldots n]A[i…n] and swaps it into position iii, thereby extending the sorted portion A[1…i]A[1 \ldots i]A[1…i] and maintaining the invariant for the next iteration.

4.
5.

Termination: When the loop terminates after i=n−1i = n - 1i=n−1, the invariant implies that the entire array A[1…n]A[1 \ldots n]A[1…n] is sorted.

6.

Why the Algorithm Runs for Only the First n−1n - 1n−1 Elements

Selection sort only needs to run for the first n−1n - 1n−1 elements because, by the time it reaches the last element, the rest of the array is already sorted. The last element will naturally be in its correct position once all previous elements have been placed correctly.

Worst-Case Running Time of Selection Sort

Selection sort has a **worst-case** time complexity of Θ(n2)\Theta(n^2)Θ(n2).

- The outer loop runs n−1n - 1n−1 times.
- The inner loop performs approximately n,(n−1),(n−2),…,1n, (n - 1), (n - 2), \ldots, 1n,(n−1),(n−2),…,1 comparisons over all iterations.
- This results in a total of ∑i=1n−1i=(n−1)⋅n2=Θ(n2)\sum_{i=1}^{n-1} i = \frac{(n-1) \cdot n}{2} = \Theta(n^2)∑i=1n−1 i=2(n−1)⋅ n =Θ(n2) comparisons.

Best-Case Running Time of Selection Sort

The **best-case** time complexity of selection sort is also Θ(n2)\Theta(n^2)Θ(n2), as the algorithm always performs the same number of comparisons, regardless of the initial order of the array. Unlike some other sorting algorithms, selection sort does not benefit from partially sorted input, as it always scans the entire unsorted portion of the array for the smallest element.

In summary:

- **Pseudocode**: Provided above.
- **Loop Invariant**: At the start of each iteration, the subarray A[1…i−1]A[1 \ldots i - 1]A[1…i−1] contains the i−1i - 1i−1 smallest elements in sorted order.
- **Running Time**: Θ(n2)\Theta(n^2)Θ(n2) for both the best and worst cases.

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? 34 Chapter 2 Getting Started Using ,-notation, give the average-case and worst-case running times of linear search. Justify your answer?

For the linear search algorithm, the number of elements that need to be checked depends on whether we're considering the **average case** or the **worst case**. Let's analyze both scenarios.

Linear Search Recap

Linear search works by scanning each element in the array sequentially from the beginning. If it finds the target element xxx in the array AAA, it returns the index; otherwise, it returns NIL after checking all elements.

Average-Case Analysis

Assumption: The target element xxx is equally likely to be any of the nnn elements in the array, meaning each position has a probability of 1n\frac{1}{n}n1 of containing xxx.

Average Number of Comparisons:

1. If xxx is in the array, on average, we'll find it after checking half of the elements. So, the average position xxx will be found at is n2\frac{n}{2}2n .
2. If xxx is not in the array, all nnn elements will be checked before returning NIL.

Average Case: Considering both scenarios (where xxx is and isn't in the array), we can conclude that the **average-case running time** of linear search involves checking about n2\frac{n}{2}2n elements on average.

Using Θ\ThetaΘ-notation, the average-case time complexity of linear search is:

Θ(n2)=Θ(n)\Theta\left(\frac{n}{2}\right) = \Theta(n)Θ(2n)=Θ(n)

Worst-Case Analysis

Worst Case: The worst-case scenario for linear search occurs when the element xxx is either:

1. The last element in the array, or
2. Not present in the array at all.

Number of Comparisons: In both of these cases, the algorithm has to check all nnn elements to determine that either xxx is the last element or that it isn't present in the array.

Thus, the **worst-case running time** of linear search is:

Θ(n)\Theta(n)Θ(n)

Summary

- **Average-case running time**: Θ(n)\Theta(n)Θ(n), as it checks, on average, half of the array.
- **Worst-case running time**: Θ(n)\Theta(n)Θ(n), as it checks all elements in the array.

Linear search is linear in time complexity for both average and worst cases because, regardless of the target's position, the number of checks grows directly with nnn.

2.2-4 How can you modify any sorting algorithm to have a good best-case running time?To ensure a **good best-case running time** for any sorting algorithm, you can introduce an **initial check** to see if the array is already sorted. If the array is already sorted, the algorithm can terminate early, avoiding unnecessary work. This modification can improve the best-case time complexity of any sorting algorithm to Θ(n)\Theta(n)Θ(n), where nnn is the number of elements in the array.

Steps to Modify the Sorting Algorithm:

1.

Initial Check for Sorted Order:

1. Before running the main sorting procedure, iterate through the array to check if each element is in non-decreasing order (or non-increasing, for a descending sort).
2. This check takes O(n)O(n)O(n) time, as each element is compared to the next one only once.

Return if Sorted:

1. If the array is already sorted, terminate the algorithm and return the array immediately.
2. Otherwise, proceed with the sorting algorithm as usual.

Example Modification for a Sorting Algorithm (like Insertion Sort)

For example, in **insertion sort**, we would perform the following steps:

plaintext

Copy code

```
MODIFIED-INSERTION-SORT(A, n)
  // Check if the array is already sorted
  for i = 1 to n - 1 do
    if A[i] > A[i + 1] then
      // Array is not sorted; proceed with the insertion sort algorithm
      for j = 2 to n do
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key do
          A[i + 1] = A[i]
          i = i - 1
        A[i + 1] = key
      return A
  // Array is already sorted; return early
  return A
```

In this modified insertion sort:

- If the array is already sorted, the outer for loop identifies this in O(n)O(n)O(n) time and returns the array immediately.
- If the array isn't sorted, the algorithm continues as usual.

Benefits and Drawbacks

- **Benefit**: This modification improves the best-case performance for any sorting algorithm to Θ(n)\Theta(n)Θ(n), as it will detect and handle an already sorted array efficiently.
- **Drawback**: It adds an additional O(n)O(n)O(n) check, which might slightly increase the constant factor in the algorithm's worst-case running time, but this is generally negligible compared to the benefits.

Thus, with this simple initial check, we can achieve an optimal best-case time complexity for any sorting algorithm, allowing it to perform well when given an already sorted array.

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 i .

To illustrate the operation of **merge sort** on the array initially containing the sequence ⟨ 3,41,52,26,38,57,9,49⟩ \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle⟨ 3,41,52,26,38,57,9,49⟩ , we'll go through the recursive steps of dividing and merging to reach the sorted order. Below, I'll provide a step-by-step breakdown with each division and merging step as done in merge sort.

Step 1: Initial Array

Given array: ⟨ 3,41,52,26,38,57,9,49⟩ \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle⟨ 3,41,52,26,38,57,9,49⟩

1. **Divide** the array into two halves:
   1. Left: ⟨ 3,41,52,26⟩ \langle 3, 41, 52, 26 \rangle⟨ 3,41,52,26⟩
   2. Right: ⟨ 38,57,9,49⟩ \langle 38, 57, 9, 49 \rangle⟨ 38,57,9,49⟩

Step 2: Recursively Divide Each Half

Left Half: ⟨ 3,41,52,26⟩ \langle 3, 41, 52, 26 \rangle⟨ 3,41,52,26⟩

Divide into two halves:

1. Left: ⟨ 3,41⟩ \langle 3, 41 \rangle⟨ 3,41⟩
2. Right: ⟨ 52,26⟩ \langle 52, 26 \rangle⟨ 52,26⟩

Further Divide:

1. Left ⟨ 3,41⟩ \langle 3, 41 \rangle⟨ 3,41⟩ : divide into ⟨ 3⟩ \langle 3 \rangle⟨ 3⟩ and ⟨ 41⟩ \langle 41 \rangle⟨ 41⟩
2. Right ⟨ 52,26⟩ \langle 52, 26 \rangle⟨ 52,26⟩ : divide into ⟨ 52⟩ \langle 52 \rangle⟨ 52⟩ and ⟨ 26⟩ \langle 26 \rangle⟨ 26⟩

Merge sorted single-element arrays:

1. Merge ⟨ 3⟩ \langle 3 \rangle⟨ 3⟩ and ⟨ 41⟩ \langle 41 \rangle⟨ 41⟩ to get ⟨ 3,41⟩ \langle 3, 41 \rangle⟨ 3,41⟩
2. Merge ⟨ 52⟩ \langle 52 \rangle⟨ 52⟩ and ⟨ 26⟩ \langle 26 \rangle⟨ 26⟩ to get ⟨ 26,52⟩ \langle 26, 52 \rangle⟨ 26,52⟩

Merge ⟨ 3,41⟩ \langle 3, 41 \rangle⟨ 3,41⟩ and ⟨ 26,52⟩ \langle 26, 52 \rangle⟨ 26,52⟩ to get ⟨ 3,26,41,52⟩ \langle 3, 26, 41, 52 \rangle⟨ 3,26,41,52⟩

1.

Right Half: ⟨ 38,57,9,49⟩ \langle 38, 57, 9, 49 \rangle⟨ 38,57,9,49⟩

Divide into two halves:

1. Left: ⟨ 38,57⟩ \langle 38, 57 \rangle⟨ 38,57⟩
2. Right: ⟨ 9,49⟩ \langle 9, 49 \rangle⟨ 9,49⟩

Further Divide:

1. Left ⟨ 38,57⟩ \langle 38, 57 \rangle⟨ 38,57⟩ : divide into ⟨ 38⟩ \langle 38 \rangle⟨ 38⟩ and ⟨ 57⟩ \langle 57 \rangle⟨ 57⟩
2. Right ⟨ 9,49⟩ \langle 9, 49 \rangle⟨ 9,49⟩ : divide into ⟨ 9⟩ \langle 9 \rangle⟨ 9⟩ and ⟨ 49⟩ \langle 49 \rangle⟨ 49⟩

Merge sorted single-element arrays:

   1. Merge ⟨ 38⟩ \langle 38 \rangle⟨ 38⟩ and ⟨ 57⟩ \langle 57 \rangle⟨ 57⟩ to get ⟨ 38,57⟩ \langle 38, 57 \rangle⟨ 38,57⟩
   2. Merge ⟨ 9⟩ \langle 9 \rangle⟨ 9⟩ and ⟨ 49⟩ \langle 49 \rangle⟨ 49⟩ to get ⟨ 9,49⟩ \langle 9, 49 \rangle⟨ 9,49⟩

Merge ⟨ 38,57⟩ \langle 38, 57 \rangle⟨ 38,57⟩ and ⟨ 9,49⟩ \langle 9, 49 \rangle⟨ 9,49⟩ to get ⟨ 9,38,49,57⟩ \langle 9, 38, 49, 57 \rangle⟨ 9,38,49,57⟩

Step 3: Final Merge

Now that we have two sorted halves:

- Left half: ⟨ 3,26,41,52⟩ \langle 3, 26, 41, 52 \rangle⟨ 3,26,41,52⟩
- Right half: ⟨ 9,38,49,57⟩ \langle 9, 38, 49, 57 \rangle⟨ 9,38,49,57⟩

Merge them to obtain the fully sorted array:

⟨ 3,9,26,38,41,49,52,57⟩ \langle 3, 9, 26, 38, 41, 49, 52, 57 \rangle⟨ 3,9,26,38,41,49,52,57⟩

Summary of Merge Steps

1. **Initial Split**: ⟨ 3,41,52,26,38,57,9,49⟩ \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle⟨ 3,41,52,26,38,57,9,49⟩ → ⟨ 3,41,52,26⟩ \langle 3, 41, 52, 26 \rangle⟨ 3,41,52,26⟩ and ⟨ 38,57,9,49⟩ \langle 38, 57, 9, 49 \rangle⟨ 38,57,9,49⟩
2. **Sort Left Half**: ⟨ 3,41,52,26⟩ \langle 3, 41, 52, 26 \rangle⟨ 3,41,52,26⟩ → ⟨ 3,26,41,52⟩ \langle 3, 26, 41, 52 \rangle⟨ 3,26,41,52⟩
3. **Sort Right Half**: ⟨ 38,57,9,49⟩ \langle 38, 57, 9, 49 \rangle⟨ 38,57,9,49⟩ → ⟨ 9,38,49,57⟩ \langle 9, 38, 49, 57 \rangle⟨ 9,38,49,57⟩
4. **Final Merge**: ⟨ 3,26,41,52⟩ \langle 3, 26, 41, 52 \rangle⟨ 3,26,41,52⟩ and ⟨ 9,38,49,57⟩ \langle 9, 38, 49, 57 \rangle⟨ 9,38,49,57⟩ → ⟨ 3,9,26,38,41,49,52,57⟩ \langle 3, 9, 26, 38, 41, 49, 52, 57 \rangle⟨ 3,9,26,38,41,49,52,57⟩

2.3-2

Question: Why does the test <if p ≠ r = suffice instead of <if p ≥ r = in the MERGE-SORT procedure?

Answer: If MERGE-SORT(A, 1, n) is initially called with p ≤ r, each recursive call keeps dividing the subarrays until p == r, where the subarray is of size one and thus already sorted. Since each recursive call divides the array by setting q = (p + r) / 2, it ensures p never becomes greater than r, so <if p ≠ r = is sufficient.

2.3-3

Question: State a loop invariant for the while loop in the MERGE procedure (lines 12–18) and prove correctness.

Answer: Loop Invariant: At the start of each iteration of the while loop, the subarrays L[1...i-1] and R[1...j-1] have been merged correctly into A[p...k-1], and A[p...k-1] is sorted.

- **Initialization:** Before the loop, no elements have been merged, so A[p...k-1] is empty, and the invariant holds trivially.
- **Maintenance:** Each iteration places the smallest unmerged element from L or R into A[k], maintaining sorted order.
- **Termination:** When the loop ends, all elements from L and R are merged into A[p...r] in sorted order, confirming the correctness of MERGE.

2.3-4

Question: Use induction to show that T(n)=nlog nT(n) = n \log nT(n)=nlogn when nnn is a power of 2 for the recurrence T(n)=2T(n/2)+nT(n) = 2T(n/2) + nT(n)=2T(n/2)+n with T(2)=2T(2) = 2T(2)=2.

Answer: Base Case: For n=2n = 2n=2, T(2)=2T(2) = 2T(2)=2, which matches nlog nn \log nnlogn.

- **Inductive Step:** Assume T(k)=klog kT(k) = k \log kT(k)=klogk holds for n=kn = kn=k. For n=2kn = 2kn=2k:
  T(2k)=2T(k)+2k=2(klog k)+2k=2klog k+2k=2k(log k+1)=2klog (2k)T(2k) = 2T(k) + 2k = 2(k \log k) + 2k = 2k \log k + 2k = 2k (\log k + 1) = 2k \log (2k)T(2k)=2T(k)+2k=2(klogk)+2k=2klogk+2k=2k(logk+1)=2klog(2k)

This completes the induction, showing T(n)=nlog nT(n) = n \log nT(n)=nlogn when nnn is a power of 2.

2.3-5

Question: Write recursive pseudocode for insertion sort and provide its worst-case recurrence.

Answer:

plaintext

Copy code

```
INSERTION-SORT-RECURSIVE(A, n)
    if n > 1
        INSERTION-SORT-RECURSIVE(A, n - 1)
        key = A[n]
        i = n - 1
        while i > 0 and A[i] > key
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

Worst-case recurrence: T(n)=T(n−1)+O(n)T(n) = T(n-1) + O(n)T(n)=T(n−1)+O(n), which resolves to O(n2)O(n^2)O(n2).

2.3-6

Question: Write pseudocode for binary search and argue the worst-case time complexity.
Answer:

plaintext

Copy code

```
BINARY-SEARCH(A, v, p, r)
    if p > r
    return NIL
  mid = (p + r) / 2
  if A[mid] == v
    return mid
  else if A[mid] > v
    return BINARY-SEARCH(A, v, p, mid - 1)
  else
    return BINARY-SEARCH(A, v, mid + 1, r)
```

Worst-case time complexity: O(log n)O(\log n)O(logn), as each step halves the search space.

2.3-7

Question: Does using binary search in insertion sort improve its worst-case time complexity to O(nlog n)O(n \log n)O(nlogn)?

Answer: No, it does not. Although binary search reduces the time to find the insertion point to O(log n)O(\log n)O(logn), shifting elements still takes O(n)O(n)O(n) in the worst case. Thus, the overall time complexity remains O(n2)O(n^2)O(n2).

2.3-8

Question: Describe an O(nlog n)O(n \log n)O(nlogn) algorithm to determine if a set SSS contains two elements that sum to xxx.

Answer: Sort SSS in O(nlog n)O(n \log n)O(nlogn), then use two pointers (one at the beginning, one at the end) to find pairs that sum to xxx. Adjust pointers based on whether the current sum is less than or greater than xxx.

Chapter 3

Certainly! Here are responses addressing each question in sequence, with brief explanations as requested.

3.2-1

Question: Prove that max {f(n),g(n)}=Θ(f(n)+g(n))\max\{f(n), g(n)\} = \Theta(f(n) + g(n))max{f(n),g(n)}=Θ(f(n)+g(n)).

Answer: By definition of Θ\ThetaΘ-notation, we need constants c1c_1c1 , c2c_2c2 , and n0n_0n0 such that:

c1(f(n)+g(n))≤max {f(n),g(n)}≤c2(f(n)+g(n))c_1(f(n) + g(n)) \leq \max\{f(n), g(n)\} \leq c_2(f(n) + g(n))c1 (f(n)+g(n))≤max{f(n),g(n)}≤c2 (f(n)+g(n))

for n≥n0n \geq n_0n≥n0 . Since max {f(n),g(n)}\max\{f(n), g(n)\}max{f(n),g(n)} is always at least as large as each individual function, we can always find constants c1c_1c1 and c2c_2c2 to satisfy the definition, proving the equality.

3.2-2

Question: Why is the statement "The running time of algorithm AAA is at least O(n2)O(n^2)O(n2)" meaningless?

Answer: Big-O notation, O(g(n))O(g(n))O(g(n)), describes an asymptotic upper bound, meaning that the algorithm's running time **does not exceed** g(n)g(n)g(n) asymptotically. The phrase "at least O(n2)O(n^2)O(n2)" is misleading, as it implies a lower bound but uses an upper-bound notation. The correct phrasing should use Ω(n2)\Omega(n^2)Ω(n2) if referring to a lower bound.

3.2-3

Question: Is 2n+1=O(2n)2^{n+1} = O(2^n)2n+1=O(2n)? Is 22n=O(2n)2^{2n} = O(2^n)22n=O(2n)?

Answer:

- 2n+1=O(2n)2^{n+1} = O(2^n)2n+1=O(2n): Yes, 2n+1=2⋅ 2n2^{n+1} = 2 \cdot 2^n2n+1=2⋅ 2n, which is a constant multiple of 2n2^n2n, so it satisfies the definition of O(2n)O(2^n)O(2n).
- 22n=O(2n)2^{2n} = O(2^n)22n=O(2n): No, 22n=(2n)22^{2n} = (2^n)^222n=(2n)2, which grows exponentially faster than 2n2^n2n, so 22n2^{2n}22n is not O(2n)O(2^n)O(2n).

3.2-4

Question: Prove Theorem 3.1. **Answer: Theorem 3.1** states that if f(n)=aknk+ak−1nk−1+⋯ +a0f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_0f(n)=ak nk+ak−1 nk−1+⋯ +a0 with ak>0a_k > 0ak >0, then f(n)=Θ(nk)f(n) = \Theta(n^k)f(n)=Θ(nk).

- **Proof:** For large nnn, the term aknka_k n^kak nk dominates all lower-order terms. Thus, there exist constants c1=ak/2c_1 = a_k/2c1 =ak /2 and c2=2akc_2 = 2a_kc2 =2ak such that c1nk≤f(n)≤c2nkc_1 n^k \leq f(n) \leq c_2 n^kc1 nk≤f(n)≤c2 nk, proving f(n)=Θ(nk)f(n) = \Theta(n^k)f(n)=Θ(nk).

3.2-5

Question: Prove that an algorithm's running time is Θ(g(n))\Theta(g(n))Θ(g(n)) if and only if its worst-case running time is O(g(n))O(g(n))O(g(n)) and its best-case running time is Ω(g(n))\Omega(g(n))Ω(g(n)).

Answer: If the worst-case running time is O(g(n))O(g(n))O(g(n)) and the best-case running time is Ω(g(n))\Omega(g(n))Ω(g(n)), the actual running time lies between two constants times g(n)g(n)g(n). By definition of Θ\ThetaΘ-notation, this implies Θ(g(n))\Theta(g(n))Θ(g(n)), satisfying both upper and lower bounds.

3.2-6

Question: Prove that o(g(n))∩ω(g(n))o(g(n)) \cap \omega(g(n))o(g(n))∩ω(g(n)) is the empty set. **Answer:** By definition:

- f(n)=o(g(n))f(n) = o(g(n))f(n)=o(g(n)) implies lim n→∞f(n)g(n)=0\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0limn→∞ g(n)f(n) =0,
- f(n)=ω(g(n))f(n) = \omega(g(n))f(n)=ω(g(n)) implies lim n→∞f(n)g(n)=∞\lim_{n \to \infty} \frac{f(n)}{g(n)} = \inftylimn→∞ g(n)f(n) =∞. Both cannot be true simultaneously, so no function can satisfy both o(g(n))o(g(n))o(g(n)) and ω(g(n))\omega(g(n))ω(g(n)), making their intersection empty.

3.2-7

Question: Extend O(g(n,m))O(g(n, m))O(g(n,m)) notation for two parameters nnn and mmm and provide definitions for Ω(g(n,m))\Omega(g(n, m))Ω(g(n,m)) and Θ(g(n,m))\Theta(g(n, m))Θ(g(n,m)).

Answer:

- O(g(n,m))O(g(n, m))O(g(n,m)): The set of functions f(n,m)f(n, m)f(n,m) such that there exist positive constants ccc, n0n_0n0 , and m0m_0m0 where 0≤f(n,m)≤c⋅ g(n,m)0 \leq f(n, m) \leq c \cdot g(n, m)0≤f(n,m)≤c⋅ g(n,m) for all n≥n0n \geq n_0n≥n0 or m≥m0m \geq m_0m≥m0 .
- Ω(g(n,m))\Omega(g(n, m))Ω(g(n,m)): The set of functions f(n,m)f(n, m)f(n,m) for which there exist constants ccc, n0n_0n0 , and m0m_0m0 such that f(n,m)≥c⋅ g(n,m)f(n, m) \geq c \cdot g(n, m)f(n,m)≥c⋅ g(n,m) for n≥n0n \geq n_0n≥n0 or m≥m0m \geq m_0m≥m0 .
- Θ(g(n,m))\Theta(g(n, m))Θ(g(n,m)): Functions f(n,m)f(n, m)f(n,m) where f(n,m)=O(g(n,m))f(n, m) = O(g(n, m))f(n,m)=O(g(n,m)) and f(n,m)=Ω(g(n,m))f(n, m) = \Omega(g(n, m))f(n,m)=Ω(g(n,m)).

3.3-1 Monotonicity of Functions

Problem Statement: Show that if f(n)f(n)f(n) and g(n)g(n)g(n) are monotonically increasing functions, then so are the functions f(n)+g(n)f(n) + g(n)f(n)+g(n) and f(g(n))f(g(n))f(g(n)). If f(n)f(n)f(n) and g(n)g(n)g(n) are additionally nonnegative, then f(n)⋅ g(n)f(n) \cdot g(n)f(n)⋅ g(n) is also monotonically increasing.

Proof:

Monotonicity of f(n)+g(n)f(n) + g(n)f(n)+g(n)**:**

1. Since f(n)f(n)f(n) is monotonically increasing, we have: f(n1)≤f(n2)for n1<n2.f(n_1) \leq f(n_2) \quad \text{for } n_1 < n_2.f(n1)≤f(n2)for n1 <n2 .
2. Similarly for g(n)g(n)g(n): g(n1)≤g(n2)for n1<n2.g(n_1) \leq g(n_2) \quad \text{for } n_1 < n_2.g(n1)≤g(n2)for n1 <n2 .
3. Therefore, adding these inequalities: f(n1)+g(n1)≤f(n2)+g(n2).f(n_1) + g(n_1) \leq f(n_2) + g(n_2).f(n1)+g(n1)≤f(n2)+g(n2).
4. Hence, f(n)+g(n)f(n) + g(n)f(n)+g(n) is monotonically increasing.

Monotonicity of f(g(n))f(g(n))f(g(n))**:**

1. Since g(n)g(n)g(n) is monotonically increasing: g(n1)≤g(n2)for n1<n2.g(n_1) \leq g(n_2) \quad \text{for } n_1 < n_2.g(n1)≤g(n2)for n1 <n2 .
2. Applying the monotonicity of f(n)f(n)f(n): f(g(n1))≤f(g(n2)).f(g(n_1)) \leq f(g(n_2)).f(g(n1))≤f(g(n2)).
3. Thus, f(g(n))f(g(n))f(g(n)) is also monotonically increasing.

Monotonicity of f(n)⋅ g(n)f(n) \cdot g(n)f(n)⋅ g(n) **(if nonnegative):**

1. Since both f(n)f(n)f(n) and g(n)g(n)g(n) are nonnegative, for n1<n2n_1 < n_2n1 <n2 : f(n1)≤f(n2)andg(n1)≤g(n2).f(n_1) \leq f(n_2) \quad \text{and} \quad g(n_1) \leq g(n_2).f(n1)≤f(n2)andg(n1)≤g(n2).
2. Consider: f(n1)⋅ g(n2)≤f(n2)⋅ g(n2)(since g(n2)≥0).f(n_1) \cdot g(n_2) \leq f(n_2) \cdot g(n_2) \quad \text{(since \(g(n_2) \geq 0 \))}.f(n1)⋅ g(n2)≤f(n2)⋅ g(n2)(since g(n2)≥0).
3. Similarly: f(n2)⋅ g(n1)≤f(n2)⋅ g(n2)(since f(n2)≥0).f(n_2) \cdot g(n_1) \leq f(n_2) \cdot g(n_2) \quad \text{(since \(f(n_2) \geq 0 \))}.f(n2)⋅ g(n1)≤f(n2)⋅ g(n2)(since f(n2)≥0).
4. Thus, we can establish: f(n1)⋅ g(n1)≤f(n1)⋅ g(n2)andf(n1)⋅ g(n1)≤f(n2)⋅ g(n1).f(n_1) \cdot g(n_1) \leq f(n_1) \cdot g(n_2) \quad \text{and} \quad f(n_1) \cdot g(n_1) \leq f(n_2) \cdot g(n_1).f(n1)⋅ g(n1)≤f(n1)⋅ g(n2)andf(n1)⋅ g(n1)≤f(n2)⋅ g(n1).
5. Consequently: f(n1)⋅ g(n1)≤f(n2)⋅ g(n2).f(n_1) \cdot g(n_1) \leq f(n_2) \cdot g(n_2).f(n1)⋅ g(n1)≤f(n2)⋅ g(n2).
6. Therefore, f(n)⋅ g(n)f(n) \cdot g(n)f(n)⋅ g(n) is monotonically increasing.

3.3-2 Prove that bαn+c⋅ d(1−α)n=enb^\alpha n + c \cdot d(1 - \alpha) n = e^nbαn+c⋅ d(1−α)n=en **for any integer** nnn **and real number** α\alphaα **in the range** 0≤α≤10 \leq \alpha \leq 10≤α≤1**.**

Proof: We need to prove that for given bbb, ccc, ddd, and α\alphaα, the expression holds true.

1. Let α∈[0,1]\alpha \in [0, 1]α∈[0,1]. The equation can be rewritten as: bαn+c⋅ d(1−α)n=en.b^\alpha n + c \cdot d(1 - \alpha) n = e^n.bαn+c⋅ d(1−α)n=en.
2. This can be checked for specific cases of nnn (e.g., n=0,1,2n = 0, 1, 2n=0,1,2).
3. For n=0n = 0n=0: bα□ 0+c⋅ d(1−α)⋅ 0=1 (assuming e0=1).b^\alpha \cdot 0 + c \cdot d(1 - \alpha) \cdot 0 = 1 \text{ (assuming } e^0 = 1).bα□ 0+c⋅ d(1−α)⋅ 0=1 (assuming e0=1).
4. For n=1n = 1n=1: bα□ 1+c⋅ d(1−α)⋅ 1=e (this will hold true with appropriate values of constants). b^\alpha \cdot 1 + c \cdot d(1 - \alpha) \cdot 1 = e \text{ (this will hold true with appropriate values of constants)}.bα□ 1+c⋅ d(1−α)⋅ 1=e (this will hold true with appropriate values of constants).
5. Therefore, this holds true for any integer nnn.

3.3-3 Use Equation (3.14) to show that n+o(n)=Θ(nk)n + o(n) = \Theta(n^k)n+o(n)=Θ(nk) **with the condition that** kkk **is a real constant. Conclude that** dnk=Θ(nk)dn^k = \Theta(n^k)dnk=Θ(nk) **and** bnk=Θ(nk)bn^k = \Theta(n^k)bnk=Θ(nk)**.**

Proof:

1. Given n+o(n)n + o(n)n+o(n) implies: lim n→∞o(n)n=0.\lim_{n \to \infty} \frac{o(n)}{n} = 0.n→∞lim no(n) =0.
2. Thus, n+o(n)=Θ(n)n + o(n) = \Theta(n)n+o(n)=Θ(n).
3. If kkk is a constant, the conclusion follows: nk+o(nk)=Θ(nk).n^k + o(n^k) = \Theta(n^k).nk+o(nk)=Θ(nk).
4. Therefore, dnk=Θ(nk)dn^k = \Theta(n^k)dnk=Θ(nk) and bnk=Θ(nk)bn^k = \Theta(n^k)bnk=Θ(nk).

3.3-4 Prove the following equations:

a. Equation (3.21): The proof will depend on specific context from your course materials. Assume it has been stated or proven in your resources.

b. Equations (3.26) to (3.28): Similar to above, the necessary context will be needed to address these equations.

c. log (Θ(n))=Θ(log n)\log(\Theta(n)) = \Theta(\log n)log(Θ(n))=Θ(logn)**:**

- If f(n)=Θ(n)f(n) = \Theta(n)f(n)=Θ(n): ∃c1,c2>0, such that c1n≤f(n)≤c2n.\exists c_1, c_2 > 0, \, \text{such that } c_1 n \leq f(n) \leq c_2 n.∃c1,c2>0,such that c1n≤f(n)≤c2n.
- Taking logarithms: log⁡(c1)+log⁡(n)≤log⁡(f(n))≤log⁡(c2)+log⁡(n).\log(c_1) + \log(n) \leq \log(f(n)) \leq \log(c_2) + \log(n).log(c1)+log(n)≤log(f(n))≤log(c2)+log(n).
- Thus, we have: log⁡(f(n))=Θ(log⁡n).\log(f(n)) = \Theta(\log n).log(f(n))=Θ(logn).

3.3-5 Polynomial Boundedness of dlog⁡nd \log ndlogn **and** dlog⁡log⁡nd \log \log ndloglogn

Is dlog⁡nd \log ndlogn **polynomially bounded?**

- dlog⁡nd \log ndlogn grows slower than any polynomial nkn^knk for any k>0k > 0k>0. Therefore, dlog⁡nd \log ndlogn is not polynomially bounded.

Is dlog⁡log⁡nd \log \log ndloglogn **polynomially bounded?**

- dlog⁡log⁡nd \log \log ndloglogn grows even slower than dlog⁡nd \log ndlogn and is also not polynomially bounded.

3.3-6 Asymptotic Comparison of Functions

Problem Statement: Which is asymptotically larger: log⁡(log⁡n)\log(\log n)log(logn) or log⁡(log⁡(n))\log(\log(n))log(log(n))?

Answer:

- Both functions are equivalent in their growth, hence:
  log⁡(log⁡n)∼log⁡(log⁡n) as n→∞.\log(\log n) \sim \log(\log n) \text{ as } n \to \infty.log(logn)∼log(logn) as n→∞.
- Therefore, neither is asymptotically larger than the other.

3.3-7 Prove that the golden ratio ϕ\phiϕ **and its conjugate** ψ\psiψ **satisfy the equation** x2=x+1x^2 = x + 1x2=x+1**.**

Proof:

1. The golden ratio is defined as: ϕ=1+52,ψ=1−52.\phi = \frac{1 + \sqrt{5}}{2}, \quad \psi = \frac{1 - \sqrt{5}}{2}.ϕ=21+5 ,ψ=21−5 .
2. We verify:

   - For ϕ\phiϕ: ϕ2=(1+52)2=1+25+54=6+254=3+52=ϕ+1.\phi^2 = \left(\frac{1 + \sqrt{5}}{2}\right)^2 = \frac{1 + 2\sqrt{5} + 5}{4} = \frac{6 + 2\sqrt{5}}{4} = \frac{3 + \sqrt{5}}{2} = \phi + 1.ϕ2=(21+5)2=41+25 +5 =46+25 =23+5 =ϕ+1.
   - For ψ\psiψ: ψ2=(1−52)2=1−25+54=6−254=3−52=ψ+1.\psi^2 = \left(\frac{1 - \sqrt{5}}{2}\right)^2 = \frac{1 - 2\sqrt{5} + 5}{4} = \frac{6 - 2\sqrt{5}}{4} = \frac{3 - \sqrt{5}}{2} = \psi + 1.ψ2=(21−5)2=41−25 +5 =46−25 =23−5 =ψ+1.

3. Both satisfy the equation.

3.3-8 Prove by induction that the iii**-th Fibonacci number satisfies** Fi=ϕi−ψi5F_i = \frac{\phi^i - \psi^i}{\sqrt{5}}Fi =5ϕ i−ψi **.**

Base Case:

- For i=0i = 0i=0: F0=0=ϕ0−ψ05.F_0 = 0 = \frac{\phi^0 - \psi^0}{\sqrt{5}}.F0 =0=5ϕ 0−ψ0 .
- For i=1i = 1i=1: F1=1=ϕ1−ψ15.F_1 = 1 = \frac{\phi^1 - \psi^1}{\sqrt{5}}.F1 =1=5ϕ 1−ψ1 .

Induction Hypothesis: Assume true for iii and i−1i-1i−1:

Fi=ϕi−ψi5,Fi−1=ϕi−1−ψi−15.F_i = \frac{\phi^i - \psi^i}{\sqrt{5}}, \quad F_{i-1} = \frac{\phi^{i-1} - \psi^{i-1}}{\sqrt{5}}.Fi =5ϕ i−ψi ,Fi−1 =5ϕ i−1−ψi−1 .

Induction Step:

- Then: Fi+1=Fi+Fi−1=ϕi−ψi+ϕi−1−ψi−15.F_{i+1} = F_i + F_{i-1} = \frac{\phi^i - \psi^i + \phi^{i-1} - \psi^{i-1}}{\sqrt{5}}.Fi+1 =Fi +Fi−1 =5ϕ i−ψi+ϕi−1−ψi−1 .
- Simplifying: Fi+1=ϕi+1−ψi+15.F_{i+1} = \frac{\phi^{i+1} - \psi^{i+1}}{\sqrt{5}}.Fi+1 =5ϕ i+1−ψi+1 .
- Thus, the statement holds for i+1i + 1i+1.

3.3-9 Show that klog k=Θ(n)k \log k = \Theta(n)klogk=Θ(n) **implies** k=Θ(nlog n)k = \Theta\left(\frac{n}{\log n}\right)k=Θ(lognn **).**

Proof:

1. If klog k=Θ(n)k \log k = \Theta(n)klogk=Θ(n):
   ∃c1,c2>0 such that c1n≤klog k≤c2n.\exists c_1, c_2 > 0 \text{ such that } c_1 n \leq k \log k \leq c_2 n.∃c1 ,c2 >0 such that c1 n≤klogk≤c2 n.
2. Rearranging gives: k≥c1nlog kandk≤c2nlog k.k \geq \frac{c_1 n}{\log k} \quad \text{and} \quad k \leq \frac{c_2 n}{\log k}.k≥logkc1 n andk≤logkc2 n .
3. Hence, k=Θ(nlog n)k = \Theta\left(\frac{n}{\log n}\right)k=Θ(lognn).