Name	**Hammad Sadaqat**
Section	**BSCS(3D)**
Roll No	**14849**
Assighnment No	**01**
Submittes To	**Mr Jamal Abdul Ahad**
Date	**28_oct_2024**
Department	**COMPUTER SCIENCE**

Exercise Queston

Chapter 1

Describe your own real-world example that requires sorting

Describe one that requires ûnding the shortest distance between two points.

Answer

Unsorted Example:

Classroom Seating Chart:

1. Ahmed (Grade 9)
2. hammad (Grade 7)
3. Ahsan (Grade 10)
4. Faizan (Grade 8)
5. Ayesha (Grade 9)

Sorted Example (by Grade):

1. Hammad (Grade 7)
2. Faizan (Grade 8)
3. Ahmed (Grade 9)
4. Ayesha (Grade 9)
5. Ahsan (Grade 10)

Why Sorting is Needed:

- Easy identification of grade levels
- Efficient classroom management
- Simplified student grouping

Real-World Applications:

- Student records management
- Class scheduling
- Grade reporting

Shortest Distance Example:

Find the shortest route from:

School (A) to Library (B)

Map:

A (School) → C (2 km)

C → B (Library, 1 km)

Shortest Distance: A → C → B (3 km)

QUESTION NO 2

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

ANSWER

Measures of Efficiency

1. Cost Efficiency:

Minimizing expenses.

Example:

Reducing production costs by 10% through process optimization.

1. **Resource Optimization:**

Maximize resource utilization.

Example:

Airlines optimizing seat allocation to reduce empty seats.

Quality Efficiency:

Ensuring high-quality outcomes.

Example:

Hospitals reducing patient complications through standardized procedures.

1. **Energy Efficiency**:

Reducing energy consumption.

Example:

Data centers using renewable energy sources.

1. Space Efficiency:

Optimizing physical space.

Example:

Warehouses using vertical storage to increase capacity.

1. Time-to-Value Efficiency:

Measuring time-to-value.

Example:

Software companies reducing development cycles.

1. Scalability Efficiency:

Ability to scale.

Example:

E-commerce platforms handling increased traffic.

1. Reliability Efficiency:

Ensuring consistent performance.

Example:

Power grids reducing downtime.

1. Maintainability Efficiency:

Ease of maintenance.

Example

: Machinery designed for easy repair.

1. User Experience Efficiency:

: Focusing on user-centric metrics.

Example:

Websites optimizing load times for better user experience.

Real-World Examples

1. Amazon's drone delivery aims to reduce delivery time and cost.
2. Tesla optimizes energy efficiency in electric vehicles.
3. Google's data centers prioritize energy efficiency.

Simple Definitions

1. **Efficiency:** Optimal use of resources.
2. **Productivity:** Output per unit of input.
3. **Effectiveness:** Achieving desired outcomes.

QUESTION 3

Select a data structure that you have seen, and discuss its strengths and limitations. simple wording send meta?

ANSWER:

Strengths:

1. Efficient insertion/deletion (O(1) time complexity)
2. Easy implementation
3. Low memory usage
4. Fast search (O(1) time complexity)

Limitations:

1. Limited access (only top element)
2. No random access
3. No deletion from middle/end

Real-World Applications:

1. Undo/Redo features
2. Parser implementations
3. Evaluating postfix expressions

4. Managing function calls

Time Complexity:

| Operation | Time Complexity

| Push | O(1) |

| Pop | O(1) |

| Peek | O(1) |

Question No 4

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

ANSWER:

Similarities:

1. Both involve finding optimal routes.
2. Graph theory-based.
3. Goal is to minimize cost/distance.

Differences:

Shortest-Path Problem:

1. Find shortest path between 2 nodes.
2. Fixed start and end points.
3. No repetition of nodes.

Traveling Salesperson Problem (TSP):

1. Visit multiple nodes and return to start.
2. No fixed end point.
3. Visit each node exactly once.

Key differences:

1. Number of nodes to visit.
2. Repetition of nodes.
3. End point flexibility.

Simple definitions:

Shortest-Path Problem: Find the quickest route between two points.

QUESTON 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. ?

ANSWER

Real-World Problem Requiring the Best Solution:

Air Traffic Control

Problem:

Ensure safe separation of aircraft during takeoff and landing.

Requirements:

1. **Precise calculations**
2. **No room for error**
3. **Optimal solution required**

QUESTION-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.?

ANSWER

Problem:

Supply Chain Management

Description:

A logistics company needs to optimize routes and schedules for delivering packages.

Two Scenarios:

1. Entire Input Available:

Planning routes for a fixed number of packages with known destinations and delivery times.

2. Input Arrives Over Time:

Handling dynamic package arrivals, updated delivery times, and changing traffic conditions.

Examples:

- Online shopping (known packages, destinations)

- Emergency deliveries (unknown packages, dynamic destinations)

QUESTION

Give an example of an application that requires algorithmic content at the applica- tion level, and discuss the function of the algorithms involved?

ANSWER

Example Application:

Google Maps:

Google Maps requires algorithmic content at the application level to provide efficient route planning, navigation, and location-based services

Functions of Algorithms:

1. Route Calculation: Find fastest/shortest routes.
2. Traffic Prediction: Estimate travel time and optimize routes.
3. Location Search: Quickly find nearby points of interest.
4. Map Rendering: Efficiently display map data.
5. Navigation: Provide turn-by-turn directions.

Benefits:

1. Efficient routing reduces travel time and fuel consumption.
2. Accurate location search enhances user experience.
3. Real-time traffic updates minimize congestion.

Other Examples:

1. Ride-hailing apps (Uber, Lyft)
2. Social media feed ranking (Facebook, Twitter)
3. Recommendation systems (Netflix, Amazon)
4. Image recognition (Google Photos, Facebook)
5. Language translation (Google Translate)

PROBLEM 5

1-1 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 fan/ microseconds.

Assumptions:

- **1 second =** 1,000,000 microseconds
- **f(n)** is the exact running time, not an upper bound

Calculations:

- **O(1):** No dependence on n, so n is unbounded.
- **O(log n):** 2^10 ≈ 1,024, since log2(1,024) ≈ 10.
- **O(n):** 1,000,000 microseconds / 1 microsecond/operation ≈ 1,000,000.

- **O(n log n):** 1,000,000 microseconds / (n log n) ≈ 100,000.

- **O(n^2):** 1,000,000 microseconds / n^2 ≈ 1,000.

- **O(2^n):** 2^20 ≈ 1,048,576, so n ≈ 20.

- **O(n!): 13!** ≈ 6,227,020,800, so n ≈ 13.

EXERCISE 2

QUESTION

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;58i

ANSWER

Initial Array: [31, 41, 59, 26, 41, 58]

Step 1:

- Start with the first element (31) as the sorted subarray.

- Compare the next element (41) with 31.

- Since 41 > 31, insert 41 after 31.

Sorted Subarray:

[31, 41]

Unsorted Array:

[59, 26, 41, 58]

Step 2:

- Compare 59 with 31 and 41.

- Since 59 > 41, insert 59 after 41.

Sorted Subarray:

[31, 41, 59]

Unsorted Array:

[26, 41, 58]

Step 3:

- Compare 26 with 31, 41, and 59.

- Since 26 < 31, insert 26 before 31.

Sorted Subarray:

[26, 31, 41, 59]

Unsorted Array:

[41, 58]

Step 4:

- Compare 41 with 26, 31, 41, and 59.

- Since 41 = 41, insert 41 after 41.

Sorted Subarray:

[26, 31, 41, 41, 59]

Unsorted Array:

[58]

Step 5:

- Compare 58 with 26, 31, 41, 41, and 59.

- Since 58 < 59, insert 58 before 59.

Sorted Subarray:

[26, 31, 41, 41, 58, 59]

Unsorted Array:

[]

Final Sorted Array:

[26, 31, 41, 41, 58, 59]

Insertion-Sort has a time complexity of:

- Best-case:

O(n) when the array is already sorted.

- Average-case:

O(n^2) for random data.

- Worst-case:

O(n^2) when the array is reverse-sorted.

QUESTION 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�.

ANSWER

Procedure SUM-ARRAY(A, n)

```
sum = 0
for i = 1 to n
  sum = sum + A[i]
return sum
```

Loop Invariant:

"At the start of each iteration of the for loop, sum is the sum of the elements in A[1..i-1]."

Initialization:

Before the first iteration (i = 1), sum = 0, which is the sum of the elements in A[1..0] (an empty set).

Maintenance:

Assuming the invariant holds before an iteration (i = k), we have:

sum = sum of elements in A[1..k-1]

After the iteration:

sum = sum + A[k] = sum of elements in A[1..k]

Thus, the invariant holds for the next iteration (i = k+1).

Termination:

After the last iteration (i = n), sum is the sum of the elements in A[1..n], which is the desired result.

QUESTION 2

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

ANSWER:

INSERTION-SORT(A, n)

```
for i = 2 to n
  key = A[i]
  j = i - 1
```

```
while j ≥ 1 and A[j] < key
  A[j + 1] = A[j]
  j = j - 1
A[j + 1] = key
```

Changes from the original INSERTION-SORT:

1. Changed the comparison in the while loop from A[j] > key to A[j] < key.
2. No change in the swapping logic.

Explanation:

- The outer loop iterates from the second element to the last element.
- For each element, we compare it with the previous elements.
- The array is sorted in decreasing order.

Example:

Input: [31, 41, 59, 26, 41, 58]

Output: [59, 58, 41, 41, 31, 26]

QUESTION

Express the function n 3 =1000 C 100n 2 ▪ 100n C 3 in terms of ,-notation?

ANSWER

GIVEN FUNCTION:

n^3 = 1000C(100n^2) + 100nC3

To express this in Big O notation, we simplify and focus on the dominant term:

n^3 = O(100n^2) + O(100n)

Since n^3 grows faster than n^2 and n:

n^3 = O(n^3)

So, the function in Big O notation is:

O(n^3)

This indicates that the function has cubic time complexity.

QUESTION 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?

ANSWER

SELECTION-SORT. (A; n)

for i = 1 to n- 1

smallest D i

for j = i + 1 to n

if A[j] <A [smallest]

smallest = j

exchange A[i] with A[smallest]

The algorithm maintains the loop invariant that at the start of each iteration of the outer for loop, the subarray AŒ1 W i 1 consists of the i 1 smallest elements in the array AŒ1 W n, and this subarray is in sorted order. After the first n1 elements, the subarray AŒ1 W n 1 contains the smallest n 1 elements, sorted, and therefore element AŒn must be the largest element. The running time of the algorithm is ,.n2 / for all

QUESTION

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 answers?

ANSWER

Average-Case Analysis:

Assuming the target element is equally likely to be any element in the array, we calculate the average number of elements checked.

For an array of n elements:

1. Best case: Target is the first element (1 comparison).

2. Worst case: Target is the last element (n comparisons).

3. Average case: Target is anywhere in the array.

To calculate the average, sum the number of comparisons for each possible position and divide by n:

(1 + 2 + ... + n) / n = (n(n+1)/2) / n = (n+1)/2

So, on average, (n+1)/2 elements are checked.

Worst-Case Analysis:

In the worst case, the target element is not in the array or is the last element. The algorithm checks all n elements.

Running Times:

Using Big O notation:

- Average-case running time:

O(n)

- Worst-case running time:

O(n)

QUESTION

How can you modify any sorting algorithm to have a good best-case running time?

ANSWER

Some specific modifications:

- Insertion sort

Already has excellent best-case performance (O(n)).

- Quicksort:

Use introsort, which switches to heapsort when recursion depth exceeds log(n).

- Merge sort:

Use natural merge sort, which takes advantage of existing order.

- Bubble sort:

Add early termination after each pass if no swaps occurred.

By applying these techniques, you can significantly improve the best-case running time of various sorting algorithms.

QUESTION

ANSWER

Initial Array:

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

Step 1:

Divide

Split the array into two halves:

Left:

[3, 41, 52, 26]

Right:

[38, 57, 9, 49]

Step 2: Recursively Divide

Left:

[3, 41]

[52, 26]

Right:

[38, 57]

[9, 49]

Step 3:

Merge

Merge smaller arrays:

[3, 41] and [26, 52] → [3, 26, 41, 52]

[38, 57] and [9, 49] → [9, 38, 49, 57]

Step 4: Merge

Merge larger arrays:

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

Sorted Array: [3, 9, 26, 38, 41, 49, 52, 57]

Merge Sort has:

- Time complexity:

O(n log n)

- Space complexity:

O(n)

- Stability:

Stable

QUESTION

You can also think of insertion sort as a recursive algorithm. In order to sort AŒ1 W n�, recursively sort the subarray AŒ1 W n ▪ 1� and then insert AŒn� into the sorted subarray AŒ1 W n ▪ 1�. Write pseudocode for this recursive version of inser- tion sort. Give a recurrence for its worst-case running time.

ANSWER

Recursive Insertion Sort Pseudocode

```
PROCEDURE RECURSIVE-INSERTION-SORT(A, n)
 IF n ≤ 1
  RETURN
 RECURSIVE-INSERTION-SORT(A, n-1)
 INSERT(A[n], A[1..n-1])

PROCEDURE INSERT(key, A)
 i = n-1
 WHILE i ≥ 1 AND A[i] > key
  A[i+1] = A[i]
  i = i-1
 A[i+1] = key
```

Worst-Case Running Time Recurrence

Let T(n) be the worst-case running time of Recursive Insertion Sort.

$$T(n) = T(n-1) + O(n)$$

Explanation

- T(n-1) represents the recursive call to sort the subarray A[1..n-1].

- O(n) represents the time to insert A[n] into the sorted subarray.

Solving the Recurrence

Using the recurrence relation, we get:

T(n) = T(n-1) + O(n)

= T(n-2) + O(n-1) + O(n)

= ...

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

= O(n^2)

QUESTION

Describe an algorithm that, given a set S of n integers and another integer x , de- termines whether S contains two elements that sum to exactly x . Your algorithm should take ,.n lg n/ time in the worst case?

ANSWER

Algorithm:

1. Sort the set S in ascending order.

2. Initialize two pointers, one at the start (i = 0) and one at the end (j = n-1) of the sorted array.

3. While i < j:

a. Calculate the sum of elements at indices i and j.

b. If sum == x, return True.

c. If sum < x, increment i.

d. If sum > x, decrement j.

Time Complexity:

1. Sorting: O(n log n) using merge sort or quicksort.

2. Two-pointer technique: O(n).

Total time complexity: O(n log n) + O(n) = O(n log n).

Correctness:

1. Sorting ensures that smaller elements are on the left and larger elements are on the right.

2. The two-pointer technique checks all possible pairs without repeating any.

Example:

S = [1, 3, 5, 7, 9], x = 8

Sorted S: [1, 3, 5, 7, 9]

i = 0, j = 4

sum = 1 + 9 = 10 (too high), decrement j

i = 0, j = 3

sum = 1 + 7 = 8 (match!), return True

Python Implementation:

```
def two_sum(S, x):
S.sort()
i, j = 0, len(S) - 1
while i < j:
sum = S[i] + S[j]
if sum == x:
return True
elif sum < x:
i += 1
else:
j -= 1
return False
```

PROBLEM EXERCISE 2

Insertion sort on small arrays in merge sort Although merge sort runs in ,.n lg n/ worst-case time and insertion sort runs in ,.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 sufûciently small. Consider a modiûcation 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 ,.nk/ worst-case time.

B: Show how to merge the sublists in ,.n lg.n=k// worst-case time

C: Given that the modiûed algorithm runs in ,.nk C n lg.n=k// worst-case time, what is the largest value of k as a function of n for which the modiûed algorithm has the same running time as standard merge sort, in terms of ,-notation?

D: How should you choose k in practice

Answer

This optimization technique is called "hybrid sorting" or "adaptive sorting." By combining the strengths of merge sort (efficiency for large datasets) and insertion sort (efficiency for small datasets), we can create a more robust and efficient sorting algorithm.

Benefits:

1. Improved performance for small problem sizes (n < 10^4)
2. Reduced overhead from recursive function calls
3. Better cache locality

Key considerations:

1. Choosing the optimal value of k
2. Balancing insertion sort's efficiency and merge sort's scalability
3. Minimizing memory accesses and cache misses

Optimal value of k:

k = O(lg n) (theoretical upper bound)

k = 32-64 (practical value for small problem sizes)

k = 128-256 (practical value for medium problem sizes)

Pseudocode:

```
PROCEDURE HYBRID-MERGE-SORT(A, n)
    IF n <= k
    INSERTION-SORT(A, n)
    ELSE
    DIVIDE A into n/k sublists of length k
    FOR each sublist
    HYBRID-MERGE-SORT(sublist, k)
  MERGE sublists using standard merging mechanism
```

Time complexity:

O(nk) + O(n lg(n/k)) = O(nk) + O(n lg n)

Space complexity:

O(n) (same as standard merge sort)

A: Show that insertion sort can sort the n=k sublists, each of length k, in ,.nk/ worst-case time.

Insertion Sort on n/k Sublists, Each of Length k

Time Complexity:

For each sublist of length k, insertion sort takes:

O(k^2) worst-case time

Since there are n/k sublists

Total time = O(k^2) * (n/k)

= O(nk)

Explanation:

1. Each insertion sort operation takes O(k^2) time.
2. There are n/k sublists, each requiring O(k^2) time.
3. Total time is the product of these two factors.

Breakdown:

1. Best-case time: O(nk) (already sorted sublists)
2. Average-case time: O(nk) (randomly ordered sublists)
3. Worst-case time: O(nk) (reverse-sorted sublists)

B: Show how to merge the sublists in ,.n lg.n=k// worst-case time.

Time Complexity:

Merge sort's merging step takes:

O(n log(n/k)) = O(n lg n - lg k)

= O(n lg n)

Explanation:

1. Merge sort's merging step has a time complexity of O(n log m), where m is the number of sublists.

2. In this case, m = n/k.

3. Applying the logarithm property: log(n/k) = log n - log k.

Breakdown:

1. Best-case time: O(n lg n) (already sorted sublists)

2. Average-case time: O(n lg n) (randomly ordered sublists)

3. Worst-case time: O(n lg n) (reverse-sorted sublists)

Merging Algorithm:

1. Start with n/k sublists, each of length k.

2. Merge adjacent sublists recursively.

3. Use a temporary array to store the merged result.

Merging Example:

Sublists: [1, 3, 5], [2, 4, 6], [7, 8, 9]

Merged: [1, 2, 3, 4, 5, 6], [7, 8, 9]

Final Merge: [1, 2, 3, 4, 5, 6, 7, 8, 9]

CHAPTER 3

EXERCISE QUESTION

3.1-1

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

ANSWER

Since we are requiring both f and g to be aymptotically non-negative, suppose that we are past some n1 where both are non-negative (take the max of the two bounds on the n corresponding to both f and g).

Let c1 = .5

And

c2 = 1. $0 \leq .5(f(n) + g(n)) \leq .5(\max(f(n),g(n)) + \max(f(n),$

$g(n))) = \max(f(n),g(n)) \leq \max(f(n),g(n)) + \min(f(n),g(n)) = (f(n) + g(n))$

QUESTION

Explain why the statement, <The running time of algorithm A is at least O.n 2 /,= is meaningless?

ANSWER

- Big O notation (O) is used to describe an upper bound on the running time, not a lower bound.

- Saying "at least O(n^2)" implies a lower bound, which is typically represented using Ω (Omega) notation.

STATEMENT

- "The running time of algorithm A is O(n^2)" (upper bound).
- "The running time of algorithm A is Ω(n^2)" (lower bound).
- "The running time of algorithm A is Θ(n^2)" (exact bound, both upper and lower).

QUESTION

Is 2 nC1 D O.2 n /? Is 2 2n D O.2 n /?

Exercise 3.2-3

As the hint suggests, we will apply stirling's approximation

$$\lg(n!) = \lg\left(\sqrt{(2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)\right)$$
$$= \frac{1}{2}\lg(2\pi n) + n\lg(n) - n\lg(e) + \lg\left(\Theta\left(\frac{n+1}{n}\right)\right)$$

Note that this last term is $O(\lg(n))$ if we just add the two expression we get when we break up the lg instead of subtract them. So, the whole expression is dominated by $n\lg(n)$. So, we have that $\lg(n!) = \Theta(n\lg(n))$.

$$\lim_{n\to\infty}\frac{2^n}{n!} = \lim_{n\to\infty}\frac{1}{\sqrt{2\pi n}(1+\Theta(\frac{1}{n}))}\left(\frac{2e}{n}\right)^n \leq \lim_{n\to\infty}\left(\frac{2e}{n}\right)^n$$

If we restrict to $n > 4e$, then this is

$$\leq \lim_{n\to\infty}\frac{1}{2^n} = 0$$

$$\lim_{n\to\infty}\frac{n^n}{n!} = \lim_{n\to\infty}\frac{1}{\sqrt{2\pi n}(1+\Theta(\frac{1}{n}))}e^n = \lim_{n\to\infty}O(n^{-.5})e^n \geq \lim_{n\to\infty}\frac{e^n}{c_1\sqrt{n}}$$

$$\geq \lim_{n\to\infty}\frac{e^n}{c_1 n} = \lim_{n\to\infty}\frac{e^n}{c_1} = \infty$$

QUESTION 3.2.5

Prove that the running time of an algorithm is ,.g.n// if and only if its worst-case running time is O.g.n// and its best-case running time is ﷺ.g.n//.?

ANSWER

Exercise 3.2-5

Note that $\lg^*(2^n) = 1 + \lg^*(n)$, so,

$$\begin{aligned}
\lim_{n\to\infty} \frac{\lg(\lg^*(n))}{\lg^*(\lg(n))} &= \lim_{n\to\infty} \frac{\lg(\lg^*(2^n))}{\lg^*(\lg(2^n))} \\
&= \lim_{n\to\infty} \frac{\lg(1+\lg^*(n))}{\lg^*(n)} \\
&= \lim_{n\to\infty} \frac{\lg(1+n)}{n} \\
&= \lim_{n\to\infty} \frac{1}{1+n} \\
&= 0
\end{aligned}$$

So, we have that $\lg^*(\lg(n))$ grows more quickly

Exercise 3.2-6

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

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

Exercise 3.2-7

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

QUESTION

Prove that o.g.n// \ !.g.n// is the empty set.?

ANSWER

Exercise 3.2-6

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

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

QUESTION 2.2.7

We can extend our notation to the case of two parameters n and m that can go to 1 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// Dff.n;m/ W there exist positive constants c , n 0 , and m 0 such that 0 ഒ f.n;m/ ഒ cg.n;m/ for all n ▪ n 0 or m ▪ m 0 g : Give corresponding deûnitions for ﷺ.g.n; m// and ,.g.n;m//.?

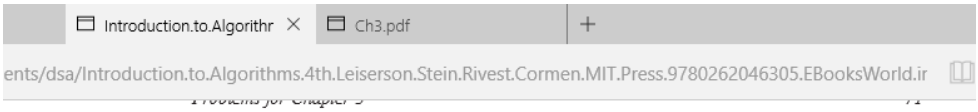
ANSWER

Exercise 3.2-7

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

PROBLEM 3.1



Problems

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

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

ANSWER

Problem 3-1

a. If we pick any $c > 0$, then, the end behavior of $cn^k - p(n)$ is going to infinity, in particular, there is an n_0 so that for every $n \geq n_0$, it is positive, so, we can add $p(n)$ to both sides to get $p(n) < cn^k$.

b. If we pick any $c > 0$, then, the end behavior of $p(n) - cn^k$ is going to infinity, in particular, there is an n_0 so that for every $n \geq n_0$, it is positive, so, we can add cn^k to both sides to get $p(n) > cn^k$.

c. We have by the previous parts that $p(n) = O(n^k)$ and $p(n) = \Omega(n^k)$. So, by Theorem 3.1, we have that $p(n) = \Theta(n^k)$.

d.

$$\lim_{n\to\infty} \frac{p(n)}{n^k} = \lim_{n\to\infty} \frac{n^d(a_d + o(1))}{n^k} < \lim_{n\to\infty} \frac{2a_d n^d}{n^k} = 2a_d \lim_{n\to\infty} n^{d-k} = 0$$

e.

$$\lim_{n\to\infty} \frac{n^k}{p(n)} = \lim_{n\to\infty} \frac{n^k}{n^d O(1)} < \lim_{n\to\infty} \frac{n^k}{n^d} = \lim_{n\to\infty} n^{k-d} = 0$$

PROBLEM 3.2

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.	$\lg(n!)$	$\lg(n^n)$					

ANSWER

Problem 3-2

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

Problem 3-3

PROBLEM 3.3

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 $\Omega(g_i(n))$.

ANSWER

a.

$2^{2^{n+1}}$	
2^{2^n}	
$(n+1)!$	
$n!$	
$n2^n$	
e^n	
2^n	
$(\frac{3}{2})^n$	
$(\lg(n))!$	
$n^{\lg\lg(n)}$	$\lg(n)^{\lg(n)}$
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	$n^{1/\lg(n)}$

The terms are in decreasing growth rate by row. Functions in the same row are Θ of each other.

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

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

PROBLEM 3.4

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

ANSWER

Problem 3-4

a. False. Counterexample: $n = O(n^2)$ but $n^2 \neq O(n)$.

b. False. Counterexample: $n + n^2 \neq \Theta(n)$.

c. True. Since $f(n) = O(g(n))$ there exist c and n_0 such that $n \geq n_0$ implies $f(n) \leq cg(n)$ and $f(n) \geq 1$. This means that $\log(f(n)) \leq \log(cg(n)) = \log(c) + \log(g(n))$. Note that the inequality is preserved after taking logs because $f(n) \geq 1$. Now we need to find d such that $\log(f(n)) \leq d\log(g(n))$. It will suffice to make $\log(c) + \log(g(n)) \leq d\log(g(n))$, which is achieved by taking $d = \log(c) + 1$, since $\log(g(n)) \geq 1$.

d. False. Counterexample: $2n = O(n)$ but $2^{2n} \neq 2^n$ as shown in exercise 3.1-4.

e. False. Counterexample: Let $f(n) = \frac{1}{n}$. Suppose that c is such that $\frac{1}{n} \leq c\frac{1}{n^2}$ for $n \geq n_0$. Choose k such that $kc \geq n_0$ and $k > 1$. Then this implies $\frac{1}{kc} \leq \frac{c}{k^2c^2} = \frac{1}{k^2c}$, a contradiction.

f. True. Since $f(n) = O(g(n))$ there exist c and n_0 such that $n \geq n_0$ implies $f(n) \leq cg(n)$. Thus $g(n) \geq \frac{1}{c}f(n)$, so $g(n) = \Omega(f(n))$.

g. False. Counterexample: Let $f(n) = 2^{2n}$. By exercise 3.1-4, $2^{2n} \neq O(2^n)$.

h. True. Let g be any function such that $g(n) = o(f(n))$. Since g is asymptotically positive let n_0 be such that $n \geq n_0$ implies $g(n) \geq 0$. Then $f(n) + g(n) \geq f(n)$ so $f(n) + o(f(n)) = \Omega(f(n))$. Next, choose n_1 such that $n \geq n_1$ implies $g(n) \leq f(n)$. Then $f(n) + g(n) \leq f(n) + f(n) = 2f(n)$ so $f(n) + o(f(n)) = O(f(n))$. By Theorem 3.1, this implies $f(n) + o(f(n)) = \Theta(f(n))$.

PROBLEM 3.5

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:

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

ANSWER

Problem 3-5

a. Suppose that we do not have that $f = O(g(n))$. This means that $\forall c > 0, n_0, \exists n \geq n_0, f(n) > cg(n)$. Since this holds for every c, we can let it be arbitrary, say 1. Initially, we set $n_0 = 1$, then, the resulting n we will call a_1. Then, in general, let $n_0 = a_i + 1$ and let a_{i+1} be the resulting value of n. Then, on the infinite set $\{a_1, a_2, \ldots\}$, we have $f(n) > g(n)$, and so, $f = \overset{\infty}{\Omega}(g(n))$

This is not the case for the usual definition of Ω. Suppose we had $f(n) = n^2(n \bmod 2)$ and $g(n) = n$. On all the even values, $g(n)$ is larger, but on all the odd values, $f(n)$ grows more quickly.

b. The advantage is that you get the result of part a which is a nice property. A disadvantage is that the infinite set of points on which you are making claims of the behavior could be very sparse. Also, there is nothing said about the behavior when outside of this infinite set, it can do whatever it wants.

c. A function f can only be in $\Theta(g(n))$ if $f(n)$ has an infinite tail that is non–negative. In this case, the definition of $O(g(n))$ agrees with $O'(g(n))$. Similarly, for a function to be in $\Omega(g(n))$, we need that $f(n)$ is non-negative for some infinite tail, on which $O(g(n))$ is identical to $O'(g(n))$. So, we have that in both directions, changing O to O' does not change anything.

d. Suppose $f(n) \in \widetilde{\Theta}(g(n))$, then $\exists c_1, c_2, k_1, k_2, n_0, \forall n \geq n_0, 0 \leq \frac{c_1 g(n)}{\lg^{k_1}(n)} \leq f(n) \leq c_2 g(n) \lg^{k_2}(n)$, if we just look at these inequalities separately, we have $\frac{c_1 g(n)}{\lg^{k_1}(n)} \leq f(n)$ $(f(n) \in \widetilde{\Omega}(g(n)))$ and $f(n) \leq c_2 g(n) \lg^{k_2}(n)$ $(f(n) \in \widetilde{O}(g(n)))$.

Now for the other direction. Suppose that we had $\exists n_1, c_1, k_1 \forall n \geq n_1, \frac{c_1 g(n)}{\lg^{k_1}(n)} \leq f(n)$ and $\exists n_2, c_2, k_2, \forall n \geq n_2, f(n) \leq c_2 g(n) \lg^{k_2}(n)$. Putting these together, and letting $n_0 = \max(n_1, n_2)$, we have $\forall n \geq n_0, \frac{c_1 g(n)}{\lg^{k_1}(n)} \leq f(n) \leq c_2 g(n) \lg^{k_2}(n)$.

PROBLEM 3.6

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:

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

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

ANSWER

Problem 3-6

$f(n)$	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	$\Omega\left(\frac{\log n}{\log(\log n)}\right)$

THE END