Abbottabad university of science and technology

Abbottabad (AUST)

Name :	Saqib Hafeez
Roll No :	14805
Class :	BS CS 3D
Section :	D
Subject :	Data structure and algorithm
Assignment No :	01
Submitted to :	Mr. Jamal Abdul Ahad
Date :	31 10 2024
Department :	Department of Computer Science

Exercises

1.1-1

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

ANSWER:

1. **Sorting:** One real-world example of a scenario that requires sorting is in the field of e-commerce. When a customer searches for a product on an e-commerce website, the search results need to be sorted in a specific order based on certain criteria such as price, popularity, or relevance. The sorting algorithm ensures that the most relevant products are displayed at the top of the search results, making it easier for the customer to find the desired product.

2. **Convex Hull:** One real-world example of a scenario that requires computing a convex hull is in the field of image processing. In image processing, the convex hull of a set of points is used to identify the boundary of an object in an image. For example, in the field of facial recognition, the convex hull of a set of facial feature points (such as the eyes, nose, and mouth) can be used to identify the boundary of a face in an image. By computing the convex hull of these feature points, the algorithm can accurately detect and isolate the face in the image, even if the face is partially obscured or rotated.

1.1-2

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

ANSWER:

Memory

This is probably the most obvious one other than speed. Not only we want to utilize available memory efficiently, we might want to reduce number of memory accesses, avoid leaking memories etc.

One example is learning based approaches often needs to work with very large amount of data. If the dataset is larger than available system RAM, we will have to design our algorithm to be able to run out of core.

Power Consumption

This is particularly relevant when we are developing our algorithm for portable devices like smartphones or smartwatches etc. In such sometimes we want to settle for relatively less power-hungry algorithm algorithm that does the job good enough instead of going with more power consuming complex algorithms that produce better results.

One example is anything to do with GPUs. Usually personal computers have GPUs that can consume 75-300 watts, whereas smartphone GPUs have a typical power budget of merely 1 Watt.

1.1-3

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

ANSWER:

Strengths of Linked List

1. Simpler addition and removal of elements, O(1)O(1) time complexity
2. Does not need contiguous memory space
3. New element can be easily inserted in any location

LIMITATIONS OF LINKED LIST

1. Accessing an element by index or by value means traversing the list, O(n)O(n) time complexity
2. Additional memory is required for storing the address (pointer) of the next/previous element.

STRENGTHS OF ARRAY

1. Accessing any element by index is simple, O(1)O(1) time complexity
2. No additional memory required to store address

LIMITATIONS OF ARRAY

1. Addition or removal of elements from any index but the last means re-arranging the whole list, O(n)O(n) time complexity
2. Accessing an element by value means traversing the list, O(n)O(n) time complexity
3. Needs contiguous memory

USE CASES:

- **Implementing stacks and queues:** Linked lists are commonly used to implement stacks and queues due to their efficient insertion and deletion operations at the beginning or end of the list.
- **Creating dynamic data structures:** Linked lists can be used to create more complex data structures like graphs, trees, and hash tables.
- **Implementing algorithms:** Linked lists are used in various algorithms, such as linked list reversal, palindrome detection, and cycle detection.

1.1-4

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

ANSWER:

They are similar in that they both deal with a map and care about the distance between points (cities and roads for example). Additionally, each problem seeks to produce a route that minimizes distance. They differ in that the shortest-path problem has a defined beginning point and end point with no stipulations on which or how many points must be traversed from beginning to end. The travelling salesman problem instead cares about hitting every destination along the way while ending at the original beginning point - this difference complicates the problem greatly.

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.

ANSWER:

Problem Requiring an Exact Solution:

Medical Diagnosis: A medical diagnosis system must accurately identify a patient's condition to ensure correct treatment. An incorrect diagnosis could lead to serious health consequences or even death. Therefore, only the exact and best solution is acceptable in this scenario.

Problem Requiring an Approximate Solution:

Route Optimization for Delivery Trucks: A delivery company aims to optimize routes for its fleet of trucks to minimize fuel consumption and delivery time. While an exact solution would be ideal, finding the absolute best route for every delivery is computationally expensive and often unnecessary. An approximate solution that is close to optimal can provide significant benefits in terms of efficiency and cost savings.

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.

ANSWER:

Real-World Problem: Weather Forecasting

In weather forecasting, the input data often includes historical weather patterns, current atmospheric conditions, and real-time satellite imagery.

- **Scenario 1: Long-Term Forecasts:** For long-term forecasts (e.g., seasonal or annual predictions), the historical data is generally available in advance, allowing for extensive analysis and modeling to predict future trends.
- **Scenario 2: Short-Term Forecasts:** For short-term forecasts (e.g., hourly or daily), the input data is constantly changing. As new observations are collected, weather

models are updated to provide the most accurate predictions. This requires real-time data processing and analysis to incorporate the latest information.

This example demonstrates how the availability of input data can vary depending on the specific problem and time frame. While historical data may be sufficient for long-term predictions, short-term forecasting often relies on real-time data to ensure accuracy.

1.2-1

Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

ANSWER:

Algorithmic Applications

Example: Recommendation Systems

One common application requiring algorithmic content at the application level is recommendation systems. These systems are used by various platforms, including e-commerce websites, streaming services, and social media platforms.

Algorithm Function:

- **Data Collection and Analysis:** The system gathers data on user preferences, behavior, and interactions with the platform.
- **Pattern Recognition:** Algorithms identify patterns and correlations within this data to understand user interests and preferences.
- **Content Matching:** Based on the identified patterns, the system suggests relevant content, products, or services to the user.
- **Personalization:** The recommendations are tailored to individual users, improving their experience and engagement.

Common Algorithms:

- **Collaborative Filtering:** Recommends items based on similarities between users or items.

- **Content-Based Filtering:** Suggests items based on their attributes and the user's preferences for those attributes.
- **Hybrid Approaches:** Combine collaborative and content-based filtering for more accurate recommendations.

Examples of Platforms:

- **Netflix:** Recommends movies and TV shows based on viewing history.
- Amazon: Suggests products based on purchase history and browsing behavior.
- **Spotify:** Recommends songs and playlists based on listening habits.
- **YouTube**: Suggests videos based on watch history and search queries.

1.2-2

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

ANSWER:

- Insertion Sort:8n^2.
- Merge Sort: 64 n log n.

We need to find the values of n for which:

8n^2<64n log n

Step 1: Simplify the Inequality

Dividing both sides by 8 gives us

n^2 < 8 n log n

Step 2: Rearrange the Inequality

n^2/n log n< 8

n/ log n < 8

Step 3: Solve the Inequality

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

To find the range of any for which this holds, we can evaluate the inequality at several values:

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)

Conclusion

From these evaluations, insertion sort is faster than merge sort for any values roughly from 1 up to around 32. Therefore, insertion sort beats merge sort for:

$$n<32$$

This gives a practical range where insertion sort would be more efficient than merge sort.

1.2-3

What is the smallest value of n such that an algorithm whose running time is 100n^2 runs faster than an algorithm whose running time is 2 n on the same machine?

ANSWER:

To find the smallest value of n where 100n^2 runs faster than 2^n, we need to solve the inequality:

100n^2 < 2^n

This inequality is difficult to solve algebraically. However, we can try different values of n to find the smallest one that satisfies the inequality.

- **For n = 10:** 100 * 10^2 = 10000, 2^10 = 1024
- **For n = 11:** 100 * 11^2 = 12100, 2^11 = 2048
- **For n = 12:** 100 * 12^2 = 14400, 2^12 = 4096
- **For n = 13:** 100 * 13^2 = 16900, 2^13 = 8192

We can see that for n = 13, 100n^2 is larger than 2^n. Therefore, the smallest value of n for which 100n^2 runs faster than 2^n is 13.

Problems

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 f. n/ microseconds.

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

Exercises

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; 58i.

ANSWER:

initial Sequence:

31, 41, 59, 26, 41, 58

Steps:

1. **First Iteration (i = 1):**
    - The first element (31) is already sorted.
2. **Second Iteration (i = 2):**
    - Key = 41
    - Compare Key with the element to its left (31).
    - Since Key (41) is greater than 31, the elements are already in the correct order.
3. **Third Iteration (i = 3):**
    - Key = 59
    - Compare Key with the element to its left (41).
    - Since Key (59) is greater than 41, the elements are already in the correct order.
4. **Fourth Iteration (i = 4):**
    - Key = 26
    - Compare Key with the element to its left (59).
    - Since Key (26) is less than 59, swap the elements:
        - 59, 41, 26, 41, 58
    - Compare Key (26) with the element to its left (41).
    - Since Key (26) is less than 41, swap the elements:
        - 59, 26, 41, 41, 58
    - Compare Key (26) with the element to its left (31).
    - Since Key (26) is less than 31, swap the elements:
        - 26, 31, 41, 41, 58
5. **Fifth Iteration (i = 5):**
    - Key = 41
    - Compare Key with the element to its left (41).
    - Since Key (41) is equal to the element to its left, the elements are already in the correct order.
6. **Sixth Iteration (i = 6):**
    - Key = 58
    - Compare Key with the element to its left (41).
    - Since Key (58) is greater than 41, the elements are already in the correct order.

Final Sorted Sequence:

26, 31, 41, 41, 58, 59

Explanation:

The INSERTION-SORT algorithm works by iterating through the array and inserting each element into its correct sorted position. It compares the current element (Key) with the elements to its left. If Key is less than the element to its left, the elements are swapped. This process continues until Key is in its correct sorted position.

2.1-2

Consider the procedure SUM-ARRAY on the facing page. It computes the sum of

the n numbers in array A[1:n]. State a loop invariant for this procedure, and use

its initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in A[1:n].

ANSWER:

Procedure SUM-ARRAY(A, n)

```
1 sum = 0
2 for i = 1 to n
3       do sum = sum + A[i]
4 return sum
```

Loop Invariant

Loop Invariant: At the start of each iteration of the loop (line 2), sum contains the sum of the elements A[1..i-1].

Initialization

Before the first iteration (i = 1): sum is initialized to 0. Since there are no elements before A[1], the sum is correctly 0.

Maintenance

Assume the loop invariant holds at the start of the ith iteration: sum contains the sum of A[1..i-1].

During the ith iteration:

sum is updated to sum + A[i]. This adds the current element A[i] to the existing sum, which is the sum of A[1..i-1].

Therefore, at the start of the next iteration (i = i+1), sum will contain the sum of A[1..i], maintaining the loop invariant.

Termination

- **After the loop terminates (i = n+1):** sum contains the sum of A[1..n], which is the desired result.

Conclusion

The loop invariant is established at the beginning, maintained throughout the loop, and holds true at the end. Therefore, the SUM-ARRAY procedure correctly returns the sum of the numbers in the array A[1:n].

2.1-3

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

ANSWER:

DECREASING-INSERTION-SORT(A)

```
for i = 1 to length(A) - 1
key = A[i]
j = i - 1
while j >= 0 and A[j] < key
A[j + 1] = A[j]
```

```
j = j - 1
A[j + 1] = key
```

The key difference is in the comparison condition in line 5. Instead of comparing A[i] with key to see if it's greater, we now compare if it's less. This ensures that the elements are inserted in decreasing order.

2.1-4

Consider the searching problem:

Input: A sequence of n numbers ha 1 ; a 2 ; : : : ; a n i stored in array AOE1 W nc 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.

ANSWER:

Pseudocode:

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

Explanation:

he LINEAR-SEARCH algorithm finds the index of a given value x in an array A by checking each element of the array one by one. If it finds the value, it returns the index. If it doesn't find the value, it returns 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.

ANSWER:

```
def add_binary_integers (A, B, n):
  carry = 0
  C = [0] * (n + 1)
  for i in range(n - 1, -1, -1):
    C[i + 1] = (A[i] + B[i] + carry) % 2
    carry = (A[i] + B[i] + carry) // 2

  C[0] = carry
  return C
```

Explanation:

1. **Initialize `carry`:** Set carry to 0 to store any carry-over from previous additions.
2. **Create `C` array:** Create a new array C of size n+1 to store the sum.
3. **Iterate through arrays:** Iterate from the least significant bit (index n-1) to the most significant bit (index 0).

4. **Calculate current bit:** Calculate the current bit of the sum C[i+1] by taking the modulo 2 of the sum of the corresponding bits in A and B plus the carry.
5. **Update `carry`:** Calculate the carry-over for the next iteration by dividing the sum of the corresponding bits in A and B plus the carry by 2.
6. **Store carry:** Store the final carry-over in C[0], which is the most significant bit of the sum.
7. **Return `C`:** Return the C array, which now contains the sum of A and B in binary form.

This implementation effectively adds two binary integers by simulating the process of adding numbers in base 2, considering the carry-over at each step.

Exercises

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

ANSWER:

When we're dealing with big-O notation, we're mostly interested in the fastest-growing term in a function. This term will dominate the others as n gets very large.

In the function n^3/1000 + 100n^2 - 100n + 3, the term with the highest exponent is n^3. So, as n gets bigger and bigger, the n^3 term will become much larger than the other terms.

Therefore, we can say that the function is O(n^3). This means that its growth rate is roughly the same as n^3.

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?

ANSWER:

Selection Sort in Pseudocode

```
selection_sort(arr):
  n = length(arr)

  for i from 0 to n-2:
    min_index = i

    for j from i+1 to n-1:
      if arr[j] < arr[min_index]:
        min_index = j

    swap(arr[i], arr[min_index])
```

Explanation:

1. **Outer loop:** Iterates through the array from index 0 to n-2.
2. **Inner loop:** Iterates through the unsorted part of the array, starting from index i+1.
3. **Comparison:** Compares the current element with the minimum element found so far.
4. **Update minimum index:** If the current element is smaller, updates the minimum index.
5. **Swap:** Swaps the current element with the minimum element.

Loop Invariant:

- **Before iteration i:** The subarray A[0..i-1] is in sorted order.
- **After iteration i:** The subarray A[0..i] is in sorted order.

Why only n-1 elements?

After the first n-1 iterations, the last element will automatically be in its correct position since the smallest elements have been moved to the beginning of the array. So, there's no need to process the last element in the loop.

Worst-case running time:

In the worst case, the inner loop needs to examine all n-i elements for each iteration i. So, the total number of comparisons is approximately:

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

This is proportional to n^2, so the worst-case running time is **O(n^2)**.

Best-case running time:

The best case occurs when the array is already sorted. In this case, the inner loop only needs to examine one element for each iteration. However, the outer loop still needs to iterate n-1 times. Therefore, the best-case running time is also **O(n^2)**.

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?

ANSWER:

Response:

Average Case:

If the element being searched for is equally likely to be any element in the array, then on average, we'll need to check half of the elements.

- For an array of size n, the average number of comparisons is (n+1)/2.

Worst Case:

In the worst case, the element is not in the array. We'll need to check all n elements before concluding that it's not there.

- The worst-case number of comparisons is n.

Running Times using Big-O Notation:

- **Average Case:** O(n)
- **Worst Case:** O(n)

Justification:

The average case and worst-case running times are both O(n) because the number of comparisons grows linearly with the size of the array in both scenarios. The constant factors (1/2 for average case) don't affect the asymptotic behavior.

2.2-4

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

ANSWER:

To achieve a good best-case running time in sorting algorithms, you can:

1. **Check if the array is already sorted** before performing the sort.
2. **Use adaptive algorithms** like Tim sort, which perform better on partially sorted data.
3. **Implement early stopping** in algorithms like Bubble Sort if no swaps occur in a pass.
4. **Use hybrid approaches**, combining algorithms (e.g., Insertion Sort for small subarrays in Merge Sort).

These strategies can significantly improve the best-case performance of your sorting algorithm.

Exercises

2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence [3; 41; 52; 26; 38; 57; 9; 49].

ANSWER:

Merge Sort Example

Given array: [3, 41, 52, 26, 38, 57, 9, 49]

Step 1: Divide the array into halves.

- Left half: [3, 41, 52, 26]
- Right half: [38, 57, 9, 49]

Step 2: Recursively sort each half.

- **Left half:**
  - Divide further: [3, 41] and [52, 26]
  - Sort each half: [3, 41] and [26, 52]
  - Merge: [26, 3, 41, 52]
- **Right half:**
  - Divide further: [38, 57] and [9, 49]
  - Sort each half: [38, 57] and [9, 49]
  - Merge: [9, 38, 49, 57]

Step 3: Merge the sorted halves.

- **Merge** [26, 3, 41, 52] and [9, 38, 49, 57]: [9, 26, 38, 41, 49, 52, 57]

Final sorted array**:** [9, 26, 38, 41, 49, 52, 57]

Note: The merge sort algorithm is a divide-and-conquer algorithm that works by recursively dividing the array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays

to get the final sorted array.

Sources and related content

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 $p > r$.

ANSWER:

Analyzing the MERGE-SORT Procedure

Understanding the Test:

The if p ≤ r test in the MERGE-SORT procedure ensures that the subarrays being merged are not empty. If p > r, the subarray A[p..r] is empty, and there's no need to merge it.

Ensuring Correctness:

To prove that the if p ≤ r test is sufficient to prevent recursive calls with p > r, we'll use induction:

Base Case:

- When n = 1, the subarray A[1..n] has only one element.
- The initial call of MERGE-SORT will have p = 1 and r = 1.
- Since p ≤ r, the base case holds.

Inductive Hypothesis:

- Assume that for any subarray of size k, where 1 ≤ k < n, the if p ≤ r test prevents recursive calls with p > r.

Inductive Step:

- Consider a subarray of size n (where n > 1).
- MERGE-SORT will divide this subarray into two halves: A[p..q] and A[q+1..r].
- By the inductive hypothesis, the recursive calls on these subarrays will not have p > r.

- When merging the sorted halves, p will always be less than or equal to r because the subarrays are contiguous and have been sorted.

Conclusion:

Therefore, by strong induction, the if $p \leq r$ test is sufficient to ensure that no recursive call of MERGE-SORT has $p > r$, as long as the initial call has $n \geq 1$. This is because the test guarantees that the subarrays being merged are always non-empty, and the recursive calls on smaller subarrays will also adhere to this condition.

2.3-3

State a loop invariant for the **while** loop of lines 12318 of the MERGE procedure.
Show how to use it, along with the **while** loops of lines 20323 and 24327, to prove that the MERGE procedure is correct.

Loop Invariant for MERGE Procedure

Invariant: At the start of each iteration of the while loop (lines 12-18), the elements in:

- A[p…q] are sorted.
- A[q+1…r] are sorted.

- B[p…i] contains the smallest elements from both subarrays in sorted order.

Proving MERGE Correctness

1. **Initialization**: Before the loop starts, both subarrays are sorted, and BBB is empty. The invariant holds.
2. **Maintenance**: During each iteration, we compare the smallest elements of both subarrays and add the smaller one to BBB. This maintains the invariant.
3. **Termination**: The loop stops when one subarray is fully merged. The remaining elements of the other subarray (which are already sorted) are then copied to AAA.

Conclusion

At the end of the MERGE procedure, A[p…r] is sorted, proving that the MERGE procedure works

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

ANSWER:

Base Case

When n=2n=2, T(2)=2=2lg⁡2T(2)=2=2lg2. So, the solution holds for the initial step.

Inductive Step

Let’s assume that there exists a kk, greater than 1, such that T(2k)=(2k)=2klg2k. We must prove that the formula holds for k+1 too, i.e. T(2k+1)=2k+1lg2k+1.

From our recurrence formula,

T(2k+1) =2T(2k+1/2)+2k+1

=2T(2k)+2·2k

=2·2klg2k+2·2k

=2·2k(lg2k+1)

=2k+1(lg2k+lg2)

=2k+1lg2k+1

This completes the inductive step.

Since both the base case and the inductive step have been performed, by mathematical induction, the statement T(n)=n lg n holds for all n that are exact power of 2.

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.

ANSWER:

Pseudocode:

```
Recursive_Insertion_Sort(A, n):
    if n <= 1:
        return  # Base case: A[1] is already sorted

    recursive_Insertion_Sort(A, n-1)  # Sort the subarray A[1:n-1] recursively

    key = A[n]  # Element to be inserted

    i = n-1  # Start from the last element of the sorted subarray

    while i > 0 and A[i] > key:
        A[i+1] = A[i]  # Shift elements to the right
        i -= 1

    A[i+1] = key  # Insert the element at the correct position
```

Worst-case recurrence:

The worst-case occurs when the array is already sorted in descending order, and each element needs to be shifted to the beginning. In this case, the recurrence relation is:

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

- T(n): The time taken to sort an array of size n.
- T(n-1): The time taken to sort the subarray of size n-1.
- (n-1): The time taken to insert the nth element into the sorted subarray.

Explanation:

- The base case n <= 1 is trivially sorted.
- The recursive call recursive Insertion Sort(A, n-1) sorts the subarray A[1:n-1].
- The loop iterates n-1 times in the worst case, shifting each element one position to the right.

Overall running time:

Solving the recurrence relation using the master theorem or other techniques, we find that the worst-case running time of the recursive insertion sort is O(n^2). This is the same as the iterative version of insertion sort.

2.3-6

Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further Problems for Chapter 2 45
consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is ,.lg n/.

ANSWER:

Pseudocode (Iterative):

```
binarySearch(A, p, r, v):
    while p <= r:
        q = floor((p + r) / 2)
        if A[q] == v:
            return q
        else if A[q] < v:
            p = q + 1
        else:
            r = q - 1
    return -1  # Element not found
```

Pseudocode (Recursive):

```
binarySearchRecursive(A, p, r, v):
    if p > r:
        return -1  # Element not found

    q = floor((p + r) / 2)

    if A[q] == v:
        return q
    else if A[q] < v:
        return binarySearchRecursive(A, q+1, r, v)
    else:
        return binarySearchRecursive(A, p, q-1, v)
```

Worst-case running time analysis:

In the worst case, binary search repeatedly divides the subarray in half until it finds the target element or determines that it's not present. This process can be modeled using a recurrence relation:

$T(n) = T(n/2) + O(1)$

- $T(n)$: The time taken to search an array of size n.

- T(n/2): The time taken to search the left or right half of the array.
- O(1): The time taken to compare the midpoint element with the target.

Using the master theorem, we can solve this recurrence to obtain:

T(n) = O(log n)

Therefore, the worst-case running time of binary search is O(log n).

2.3-7

The while loop of lines 537 of the I NSERTION-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 ,.n lg n/?

ANSWER:

No, using binary search instead of linear search in the insertion sort algorithm would not improve the overall worst-case running time to O(n lg n).Due to following reason:

1. **Binary search requires a sorted array:** Binary search is efficient because it can quickly eliminate half of the search space in each iteration. However, the subarray A[1: j - 1] in insertion sort is not necessarily sorted at the beginning of each iteration. Therefore, using binary search would introduce additional overhead to sort the subarray before searching, which would negate any potential gains from using binary search.
2. **Linear search is already efficient for small subarrays:** For small subarrays, linear search is often more efficient than binary search due to the overhead of setting up and maintaining the search boundaries. In insertion sort, the subarrays being searched are typically small, especially in the early stages of the algorithm. Therefore, the overhead of binary search might outweigh any benefits.
3. **Overall worst-case is still O(n^2):** Even if binary search were used efficiently for larger subarrays, the overall worst-case running time of insertion sort would still be O(n^2). This is because the outer loop of insertion sort iterates n times, and in the worst case, the inner loop can iterate up to n times as well, resulting in a quadratic time complexity.

Therefore, while binary search can be a more efficient searching algorithm for sorted arrays, it's not suitable for the specific context of insertion sort, where the subarrays are not always sorted and the overhead of using binary search might outweigh any potential benefits.

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 ,.n lg n/ time in the worst case.

ANSWER:

The algorithm for the two-sum problem involves:

1. **Sorting** the given set of integers.
2. **Using two pointers** to efficiently search for a pair of elements that add up to the target value.

This approach ensures that the algorithm runs in O(n log n) time, which is the required efficiency.

Problems

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?

ANSWER:

Analyzing the Modified Merge Sort Algorithm

a. Insertion Sort on Sub lists

- **Worst-case time for insertion sort:** O(k^2) for a sub list of length k.
- **For n/k sub lists:** O(k^2) * (n/k) = O(nk).

b. Merging Sub lists

- **Standard merging:** O(n) for merging two sorted lists of total length n.
- **For n/k sub lists:** O(n) * log(n/k) = O(n log(n/k)).

c. Overall Running Time and Choosing k

- **Modified algorithm:** O(nk + n log(n/k))
- **Standard merge sort:** O(n log n)

- **Setting the two running times equal:**
  - $nk + n \log(n/k) = n \log n$
  - $nk = n \log n - n \log(n/k)$
  - $nk = n \log n - n (\log n - \log k)$
  - $nk = n \log k$
  - $k = \log k$
- **Solving for k:**
  - This equation doesn't have a closed-form solution. However, we can observe that as n increases, k must also increase to maintain equality.
  - For practical purposes, we can choose a value for k that balances the overhead of insertion sort with the potential benefits of reducing the number of merge operations.

d. Choosing k in Practice

- **Experimentation:** The optimal value of k can vary depending on the specific implementation, data sets, and hardware.
- **Factors to consider:**
  - **Insertion sort overhead:** If insertion sort is relatively fast on your system, a larger value of k might be beneficial.
  - **Merge operation overhead:** If merging is expensive, a smaller value of k might be better.
  - **Data distribution:** If the data is already partially sorted, a smaller value of k might be sufficient.
- **A common approach:** Start with a moderate value of k (e.g., 16) and experiment with different values to find the best performance for your specific use case.

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

$$A'[1] \le A'[2] \le \cdots \le A'[n] . \tag{2.5}$$

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?

ANSWER:

a. Proving BUBBLESORT is Correct

1. **Termination**: BUBBLESORT finishes after a finite number of iterations.
2. **Sorted Order**: After execution, the array A’A'A’ satisfies A’[1]≤A’[2]≤…≤A’[n]A'.

b. Loop Invariant for Inner Loop

Invariant: At the start of each inner loop iteration, A[n−i+1:n]A[n - i + 1 : n]A[n−i+1:n] contains the iii largest elements in sorted order.

Proof:

- **Initialization**: The largest element is moved to A[n] in the first pass.
- **Maintenance**: Each pass moves the next largest element to its correct position.
- **Termination**: When finished, all elements are sorted.

c. Loop Invariant for Outer Loop

Invariant: At the start of each outer loop iteration, A[n−i+1:n] is sorted.

Proof:

- **Initialization**: Initially, nothing is sorted.
- **Maintenance**: The largest unsorted element is moved to its correct position each pass.
- **Termination**: After n−1 iterations, the entire array is sorted.

d. Worst-Case Running Time

- **BUBBLESORT**: O(n2)
- **INSERTION-SORT**: Also O(n2), but generally faster in practice due to fewer operations on partially sorted arrays.

In summary, both algorithms have O(n2) complexity, but INSERTION-SORT is typically more efficient.

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.)

ANSWER:

(a):

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

(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

(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.

(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.)

ANSWER:

Inversions in an Array

a. Inversions of the Array

The five inversions of the array [2, 3, 8, 6, 1] are:

- (2, 1)
- (3, 1)
- (8, 1)
- (6, 1)
- (8, 6)

b. Array with Most Inversions

The array with elements from the set {1, 2, ..., n} that has the most inversions is the one in descending order:

- [n, n-1, ..., 2, 1] The number of inversions in this array is:
- (n-1) + (n-2) + ... + 1 = n(n-1)/2

c. Relationship Between Insertion Sort and Inversions

The number of comparisons made by insertion sort is directly related to the number of inversions in the input array. Each time an element is shifted to its correct position, it is compared with elements that are greater than it. The number of such comparisons is equal to the number of inversions involving that element. Therefore, the more inversions in the array, the more comparisons insertion sort will make.

d. Algorithm to Count Inversions

Modified Merge Sort:

```
countInversions(A, p, r)
    if p < r
        q = floor((p + r) / 2)
        leftInversions = countInversions(A, p, q)
        rightInversions = countInversions(A, q+1, r)
        mergeInversions = mergeAndCountInversions(A, p, q, r)
        return leftInversions + rightInversions + mergeInversions

mergeAndCountInversions(A, p, q, r)
    # Merge A[p..q] and A[q+1..r]
    # While merging, count the number of elements in A[q+1..r] that are smaller than elements in A[p..q]
    # This count represents the number of inversions involving elements from the two subarrays
```

Explanation:

- The modified merge sort algorithm counts inversions while merging the subarrays.
- Whenever an element from the right subarray is smaller than an element from the left subarray, it indicates an inversion.

- The total number of inversions is the sum of inversions in the left subarray, right subarray, and the inversions that occur during merging.
- The worst-case running time of this algorithm is O(n log n), as it is based on the merge sort algorithm.

Exercises

3.1-1

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

ANSWER:

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

$T(n)=n(n-1)/2$

This holds for any size n regardless of whether it's a multiple of 3. The time complexity remains O(n^2).

3.1-2

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

ANSWER:

Selection Sort Overview

1. Repeatedly selects the smallest element from the unsorted portion of the array.
2. Swaps it with the first unsorted element.
3. Moves the boundary between sorted and unsorted parts.

Running Time Analysis

1. **Outer Loop**: Runs n−1n - 1n−1 times.
2. **Inner Loop**: In each iteration, it compares elements:
    - First iteration: n−1n - 1n−1 comparisons.
    - Second iteration: n−2n - 2n−2 comparisons.
    - ...
    - Last iteration: 1 comparison.

3. **Total Comparisons**:

   $(n-1)+(n-2)+\ldots+1=(n-1)n/2=O(n2)$

Conclusion

- **Overall Time Complexity**: O(n2).
- Selection Sort is inefficient for large arrays compared to faster algorithms like Merge Sort or Quick Sort.

  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?

-

ANSWER:

The lower-bound argument for Insertion Sort with αn\alpha nαn largest values in the first αn\alpha nαn positions shows that the number of comparisons is proportional to α(1−2α)n To 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 $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.

ANSWER:

Given:

- f(n) and g(n) are nonnegative functions.

To prove:

- max(f(n), g(n)) = Θ(f(n) + g(n))

Proof:

1. max(f(n), g(n)) is O(f(n) + g(n))

- The maximum of two numbers is always less than or equal to their sum. So,
  - max(f(n), g(n)) ≤ f(n) + g(n)
- Therefore, max(f(n), g(n)) is O(f(n) + g(n)).

2. max(f(n), g(n)) is Ω(f(n) + g(n))

- The maximum of two numbers is always greater than or equal to their average. So,
  - max(f(n), g(n)) ≥ (f(n) + g(n)) / 2
- Since (f(n) + g(n)) / 2 is a constant multiple of f(n) + g(n), max(f(n), g(n)) is Ω(f(n) + g(n)).

Conclusion:

- From 1 and 2, we conclude that max(f(n), g(n)) is both O(f(n) + g(n)) and Ω(f(n) + g(n)).
- Therefore, 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.

ANSWER:

The statement "The running time of algorithm A is at least O(n^2)" is meaningless because it's redundant.

Here's a breakdown:

1. **O(n^2) is an upper bound:** O(n^2) means that the algorithm's running time is bounded above by a constant multiple of n^2 for sufficiently large n. In other words, it's a worst-case scenario.
2. **"At least" implies a lower bound:** Saying "at least O(n^2)" implies that the running time is always greater than or equal to a constant multiple of n^2.
3. **Redundancy:** Since O(n^2) is already an upper bound, saying it's "at least" O(n^2) is redundant. It's like saying "the temperature is at least hot" when "hot" already implies a minimum temperature.

A more meaningful statement would be:

- **"The running time of algorithm A is O(n^2)"** (This indicates the worst-case scenario.)
- **"The running time of algorithm A is Ω(n^2)"** (This indicates a lower bound, meaning the algorithm's running time is at least proportional to n^2 in the best case.)

By understanding the nuances of big O, big Omega, and big Theta notation, we can avoid making such redundant statements.

3.2-3

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

ANSWER:

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

We know that:

2^n+1=2·2^2.

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

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

We know that:

2^2n=(2n) ^2

This grows much faster than 2^n because squaring 2^n increases its growth rate significantly.

Therefore, 2^n grows exponentially faster than 2^nand we cannot say that 2^2n=O(2^n)

thus, the correct answer is:

- 2n+1=O(2^n)
- 2^2n≠O(2^n)

3.2-4

Prove Theorem 3.1.

ANSWER:

Theorem 3.1:

For any two functions f(n) and g(n), we have f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).

This theorem is a fundamental concept in the analysis of algorithms. It establishes a relationship between the big-Θ notation, which denotes an asymptotically tight bound, and the big-O and big-Ω notations, which represent asymptotic upper and lower bounds, respectively.

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 $\Omega(g(n))$.

ANSWER:

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:
     $$c_1 g(n) \leq T(n) \leq c_2 g(n) \quad \text{for all } n \geq n_0.$$
   - 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:
     $$T(n) \leq c_2 g(n) \quad \text{and} \quad T(n) \geq c_1 g(n) \quad \text{for sufficiently large } n.$$
   - Combining these gives:
     $$c_1 g(n) \leq T(n) \leq c_2 g(n),$$
   - Therefore, $T(n) = \Theta(g(n))$.

3.2-6

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

ANSWER:

Proof: Assume f(n) ∈ o(g(n)) ∩ g(n). Then there exist c, n0, c1, c2, n1 such that:

- f(n) < c * g(n) for all n ≥ n0
- c1 * g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n1 Choose c = c1/2. For n ≥ max(n0, n1), we have:
- f(n) < c * g(n)
- f(n) ≥ c1 * g(n) = 2c * g(n) Contradiction. Therefore, o(g(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 $\Theta(g(n,m))$.

ANSWER:

To provide the definitions for Ω(g(n, m)) and Θ(g(n, m)) in the context of two independent parameters n and m, let's first recall the definitions for the single-parameter case:

- **Ω(g(n)):** f(n) is in Ω(g(n)) if there exist positive constants c and n0 such that f(n) ≥ cg(n) for all n ≥ n0.
- **Θ(g(n)):** f(n) is in Θ(g(n)) if there exist positive constants c1, c2, and n0 such that c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0.

Now, we can extend these definitions to the two-parameter case:

Ω(g(n, m)): f(n, m) is in Ω(g(n, m)) if there exist positive constants c, n0, and m0 such that f(n, m) ≥ cg(n, m) for all n ≥ n0 **or** m ≥ m0.

Θ(g(n, m)): f(n, m) is in Θ(g(n, m)) if there exist positive constants c1, c2, n0, and m0 such that c1g(n, m) ≤ f(n, m) ≤ c2g(n, m) for all n ≥ n0 **or** m ≥ m0.