# *ASSIGNMANT NO 1*


*BY*

*ZAIN ALI ASIF (14875)*

*BSCS 3D*



***Submitted to: Sir Jamal***

***Date: 30-10-2024***

***Subject: DSA***


***DEPARTMENT OF COMPUTER SCIENCE ABOTTABAD***
***UNIVERSITY OF SCIENCE AND TECHNOLOGY***

# CHAPTER NO 1

# THE ROLE OF ALGORITHMS IN COMPUTING

## EXERCISES

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

---

## Answer:

**Example Requiring Sorting**

**Scenario: Organizing a Book Collection**

Imagine you have a large collection of books at home. Over time, you've accumulated various genres, including fiction, non-fiction, fantasy, and biographies. To make it easier to find and access these books, you decide to organize your collection.

**Sorting Process:**

1. **Identifying Sorting Criteria**: You choose to sort the books by genre first, then by author within each genre, and finally by the title within each author's works.
2. **Collecting Books**: You gather all the books from different rooms and stack them on a table.
3. **Categorizing**: You create separate piles for each genre. For example:
    - **Fiction**: Novels by authors like J.K. Rowling and George Orwell.
    - **Non-Fiction**: Biographies and self-help books by authors like Michelle Obama and Dale Carnegie.
    - **Fantasy**: Books like "The Hobbit" and "A Game of Thrones".
4. **Sorting within Categories**: Once the genres are established, you sort the books within each genre alphabetically by the author's last name. For instance, in the fiction category, you would place George Orwell's works before J.K. Rowling's.
5. **Final Arrangement**: Finally, you arrange all the books on the shelves according to the sorted order.

**Outcome**: This sorting process makes it much easier to locate any book in your collection. You can quickly find a specific author's work or browse a particular genre without having to sift through an unorganized pile of books.

**Example Requiring Finding the Shortest Distance**

**Scenario: Delivering Packages in a City**

Consider a delivery driver working for an e-commerce company. The driver needs to deliver packages to various locations across a city. To optimize their route, they need to find the shortest distance between their starting point (the warehouse) and the various delivery addresses.

**Finding Shortest Distance Process:**

1. **Mapping the Route**: The driver starts by identifying all the delivery addresses on a map. Each address represents a point on the map.
2. **Using Graph Theory**: The city can be represented as a graph, where intersections are nodes (vertices) and the roads connecting them are edges. Each edge has a weight representing the distance between intersections.
3. **Applying Algorithms**: To find the shortest distance, the driver can use algorithms like Dijkstra's algorithm or the A* algorithm. These algorithms systematically explore the paths and calculate the shortest route from the starting point to each delivery location.
4. **Calculating the Route**: The driver inputs the warehouse location and the list of delivery addresses into a route optimization software. The software processes the data and calculates the most efficient route based on the distances.
5. **Resulting Route**: The driver receives a suggested route that minimizes the total distance traveled. For example, instead of delivering to Address A, B, and C in a random order, the software may suggest going to Address B first, then A, and finally C, resulting in reduced fuel consumption and time spent driving.

**Outcome**: By finding the shortest distance, the delivery driver can ensure timely deliveries while saving on operational costs. This efficient route planning is essential for maintaining customer satisfaction and optimizing delivery operations.

---

This structured approach to sorting and distance calculation demonstrates the practical applications of these concepts in everyday situations, illustrating their importance in organizing information and optimizing processes.

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

---

## Answer:

When assessing efficiency in real-world settings, it’s crucial to consider a variety of measures beyond just speed. These measures include:

1. **Cost Efficiency**: This evaluates the expenses incurred relative to the output produced. It's vital for organizations to ensure that resources are used economically. For instance, in a manufacturing environment, the goal is to lower production costs while maintaining high quality. By analyzing costs associated with labor, materials, and overhead, companies can identify areas to optimize spending without compromising output.
2. **Resource Utilization**: This measure examines how effectively resources such as materials, labor, and equipment are used in a process. High resource utilization indicates that inputs are being efficiently transformed into outputs. For example, in a logistics operation, maximizing the use of delivery trucks while ensuring they are not overloaded can lead to reduced transportation costs and increased operational efficiency.
3. **Quality**: Quality is a critical aspect of efficiency that shouldn’t be overlooked. A process may be fast but if it results in a high volume of defective products, it is considered inefficient. In software development, for instance, delivering a product quickly at the expense of quality can lead to additional costs for bug fixes and rework. Striking the right balance between speed and quality is essential for long-term success.
4. **Flexibility**: The ability of a system to adapt to changes—such as fluctuating customer demands or unexpected disruptions—is another vital measure of efficiency. Flexible processes can quickly adjust without incurring significant costs. For example, a manufacturing facility that can easily switch between different product lines in response to market trends can better meet consumer needs.
5. **Sustainability**: Sustainability measures the environmental impact of processes and practices. Efficient operations should aim to minimize waste and energy consumption while utilizing renewable resources when possible. For instance, companies adopting green technologies not only lower costs but also enhance their reputation and contribute to environmental preservation.
6. **Throughput**: This metric refers to the amount of product or service produced within a specified time frame. High throughput signifies that a process effectively generates output without unnecessary delays. In a restaurant setting, for example, a high throughput of meals prepared during peak hours indicates efficient kitchen operations.
7. **Lead Time**: Lead time measures the total duration from the initiation of a process until its completion. Reducing lead time enhances customer satisfaction by ensuring quicker delivery of products or services. For instance, in e-commerce, minimizing lead times for order fulfillment is critical for maintaining competitive advantage and customer loyalty.
8. **Employee Productivity**: Evaluating employee productivity provides insights into how effectively staff contribute to organizational goals. High productivity often reflects efficient processes and effective management practices. Companies can enhance productivity through training programs and by providing tools that empower employees to perform better.
9. **Customer Satisfaction**: Ultimately, efficiency should be reflected in customer satisfaction levels. Efficient processes that meet or exceed customer expectations regarding quality, delivery times, and overall service can lead to repeat business and positive referrals. Measuring customer feedback and implementing improvements based on their suggestions can significantly boost satisfaction.

10. **Risk Management**: This involves evaluating potential risks associated with processes and taking steps to mitigate them. Efficient operations should aim to minimize risks related to safety, compliance, and operational disruptions. For example, a construction firm that prioritizes safety protocols can prevent accidents and ensure that projects remain on schedule.

By considering these diverse measures of efficiency, organizations can gain a comprehensive understanding of their operational effectiveness and identify areas for improvement. Balancing these factors ensures that processes are not only rapid but also cost-effective, high-quality, and sustainable, leading to long-term success.

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

---

## Answer:

One commonly used data structure is the **Array**. Arrays are collections of items stored at contiguous memory locations and can be accessed using indices. They are foundational in computer science and programming, used in various applications.

### *Strengths of Arrays:*

1. **Fast Access**: Arrays provide constant-time access to elements, O(1), due to their contiguous memory allocation. This means that if you know the index of the element, you can retrieve it quickly without having to traverse other elements.
2. **Memory Efficiency**: Arrays have a low overhead compared to other data structures, such as linked lists. Since the elements are stored in contiguous memory locations, there is no need for additional pointers or references, making arrays memory efficient for storing data.
3. **Ease of Use**: Arrays are simple to implement and use. Most programming languages offer built-in support for arrays, making it easy to declare, initialize, and manipulate them. Their straightforward structure allows developers to quickly grasp their functionality.
4. **Cache Performance**: Due to their contiguous memory allocation, arrays tend to have better cache performance. When elements are stored close together, it minimizes cache misses, resulting in faster data access during operations.
5. **Predictable Size**: When you declare an array, you typically define its size, which can help with memory management. This predictability can be beneficial when dealing with fixed-size datasets.

### *Limitations of Arrays:*

1. **Fixed Size**: One of the significant limitations of arrays is their fixed size. Once an array is created, its size cannot be changed. This can lead to wasted memory if the array is

larger than necessary or insufficient capacity if the array is too small. To resize an array, a new array must be created, and the elements copied, which can be time-consuming.

2. **Inefficient Insertions and Deletions**: Inserting or deleting elements in an array can be inefficient. If an element is added or removed, the subsequent elements must be shifted, leading to a time complexity of O(n) for these operations. This makes arrays less suitable for dynamic datasets where frequent insertions and deletions occur.
3. **Limited Flexibility**: Arrays are not as flexible as some other data structures, such as linked lists or dynamic arrays. They do not support operations like dynamic resizing or automatic memory allocation when new elements need to be added.
4. **Type Homogeneity**: Most traditional arrays require elements to be of the same data type, which can limit their versatility. This homogeneity can make it challenging to store mixed data types without using more complex data structures.
5. **Poor Support for Sparse Data**: When dealing with sparse datasets, where most of the elements are unused or null, arrays can be inefficient. They allocate memory for all elements regardless of whether they are used, leading to waste. In such cases, alternative structures like sparse arrays or hash tables may be more efficient.

*Conclusion:*

In summary, arrays are a fundamental data structure with distinct strengths, such as fast access and memory efficiency. However, their limitations, including fixed size and inefficient insertions and deletions, must be considered when choosing the appropriate data structure for specific applications. Depending on the requirements of a project, other data structures like linked lists, dynamic arrays, or hash tables may offer better performance and flexibility.

**Question: How are the shortest-path and traveling-salesperson problems similar? How are they different?**

---

## Answer:

The **Shortest-Path Problem** and the **Traveling Salesperson Problem (TSP)** are both fundamental problems in graph theory and optimization. They share certain similarities but also have key differences.

*Similarities:*

1. **Graph Representation**: Both problems can be represented using graphs, where vertices (nodes) represent locations, and edges (connections) represent the paths between those locations. Each edge may have a weight that signifies distance, cost, or time.
2. **Optimization Goals**: Both problems involve finding an optimal solution regarding minimizing distance or cost. In the shortest-path problem, the goal is to find the minimum

distance between two specific points, while in the TSP, the goal is to find the minimum distance for a round trip that visits all specified points.

3. **Applications**: Both problems have practical applications in fields such as logistics, transportation, networking, and robotics. For example, both can be used to optimize delivery routes or network routing protocols.
4. **Algorithms**: Various algorithms can solve both problems, such as Dijkstra's algorithm for shortest paths and approximation algorithms or heuristic methods (like genetic algorithms or nearest-neighbor algorithms) for the TSP. This highlights the computational aspect of finding efficient solutions in both scenarios.

*Differences:*

1. **Problem Definition**:
   - **Shortest-Path Problem**: The objective is to find the shortest path between two specific nodes in a graph. For example, finding the shortest route from point A to point B.
   - **Traveling Salesperson Problem (TSP)**: The objective is to find the shortest possible route that visits a set of cities (nodes) exactly once and returns to the starting city. It requires a complete tour of all points.
2. **Complexity**:
   - **Shortest-Path Problem**: This problem can generally be solved in polynomial time, with algorithms like Dijkstra's providing efficient solutions for graphs with non-negative edge weights.
   - **Traveling Salesperson Problem (TSP)**: TSP is NP-hard, meaning that there is no known polynomial-time solution for it. As the number of cities increases, the number of possible routes grows factorially, making it computationally challenging to solve optimally.
3. **Constraints**:
   - **Shortest-Path Problem**: Typically has fewer constraints, often focusing solely on the path from one point to another without needing to visit all nodes.
   - **Traveling Salesperson Problem (TSP)**: Requires visiting each node exactly once, making it more constrained and complex. Additionally, TSP can have variations, such as allowing visits to cities more than once or not requiring a return to the starting point.
4. **Nature of Solutions**:
   - **Shortest-Path Problem**: Solutions are often unique for given start and end points and can vary based on the weights of edges in the graph.
   - **Traveling Salesperson Problem (TSP)**: Solutions can vary widely based on the order in which cities are visited, and there can be many equally optimal routes, particularly in symmetrical TSP scenarios (where the distance from A to B is the same as from B to A).

*Conclusion:*

In summary, while the shortest-path problem and the traveling-salesperson problem share common ground in graph representation, optimization goals, and applications, they differ significantly in problem definition, computational complexity, constraints, and the nature of solutions. Understanding these similarities and differences is crucial for selecting appropriate algorithms and approaches when tackling these problems in practical scenarios.

**Question: 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 the Best Solution: Medical Diagnosis*

**Scenario**: In the field of medicine, diagnosing a patient's condition accurately is critical. For example, if a patient presents with symptoms of a serious illness such as cancer, obtaining the best possible diagnosis is essential for effective treatment.

- **Why Only the Best Solution Will Do**: Accurate diagnosis directly affects patient outcomes. Misdiagnosis can lead to inappropriate treatment, delayed recovery, and even life-threatening consequences. In this scenario, healthcare professionals rely on comprehensive testing, imaging, and clinical assessments to arrive at the most precise diagnosis possible.
- **Implications**: The consequences of an inaccurate diagnosis could be severe, ranging from unnecessary surgeries to the progression of an untreated illness. Thus, in this situation, only the best solution—an accurate and timely diagnosis—will suffice.

## *Problem Where Approximately the Best Solution is Good Enough: Delivery Route Optimization*

**Scenario**: In logistics and transportation, companies often need to optimize delivery routes for their vehicles to minimize travel time and costs.

- **Why Approximately the Best Solution is Good Enough**: While the goal is to find the most efficient route, the reality is that many factors (traffic, weather conditions, road closures) can affect delivery times. Therefore, a solution that is "close enough" to the optimal route can still be very effective in achieving timely deliveries.
- **Implications**: Companies can use heuristic or approximation algorithms (like the nearest neighbor algorithm) to determine efficient routes without needing to find the absolute best path. This approach balances computational efficiency with the need for effective service, ensuring that deliveries are made on time while being responsive to real-world variables.

**Question: 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: Scheduling Deliveries in a Logistics Company*

**Overview**: A logistics company manages the delivery of goods to various locations, requiring effective scheduling to optimize routes and ensure timely deliveries. This problem can vary significantly based on the availability of information regarding deliveries.

## *Scenario 1: Entire Input Available in Advance*

In some situations, the logistics company receives all the delivery requests for a particular day in advance. This can occur, for example, when:

- **Customers place orders for delivery well ahead of time**, allowing the company to have a complete list of delivery addresses, time windows, and special requirements.
- **Input Characteristics**: With all the data available upfront, the company can analyze the delivery requests and create an optimal route for its drivers using algorithms like the Vehicle Routing Problem (VRP). The entire dataset allows for comprehensive planning, ensuring that routes are efficient and meet customer expectations for timely delivery.
- **Advantages**: In this scenario, the logistics team can minimize fuel costs, reduce delivery times, and maximize the number of deliveries made in a day by carefully scheduling routes based on the full information set.

## *Scenario 2: Input Arrives Over Time*

Conversely, there are instances when delivery requests come in sporadically throughout the day, such as during peak hours or when last-minute orders are placed. This can happen when:

- **Customers place urgent orders**, requiring immediate delivery, which disrupts previously planned schedules.
- **Input Characteristics**: As new orders arrive, the logistics company must adapt its routing in real-time. This requires a more dynamic scheduling approach, where the company might need to reevaluate and modify routes based on current traffic conditions, driver locations, and new delivery requests.
- **Challenges**: In this scenario, drivers may already be on the road with an initial route, and introducing new deliveries can lead to inefficiencies if not managed well. The company must balance the need to accommodate urgent requests while minimizing disruptions to planned routes.
- **Advantages of Flexibility**: Utilizing adaptive algorithms or real-time tracking systems can help the logistics company make quick adjustments to routes based on incoming data. This flexibility can enhance customer satisfaction by fulfilling urgent requests while maintaining overall operational efficiency.

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

---

## Answer:

*Application: Online Recommendation Systems*

**Overview**: Online recommendation systems are widely used in various industries, particularly in e-commerce (like Amazon), streaming services (like Netflix), and social media (like Facebook). These systems analyze user preferences and behaviors to suggest products, movies, music, or other content that users are likely to enjoy or purchase.

*Function of Algorithms in Recommendation Systems:*

1. **Collaborative Filtering**:
    - **Description**: This algorithm makes recommendations based on the preferences and behaviors of similar users. It operates under the principle that if two users have a history of agreeing on certain items (e.g., movies), they are likely to agree on others.
    - **Types**:
        - **User-based Collaborative Filtering**: This approach finds users with similar preferences to the target user and recommends items they liked. For instance, if User A and User B both enjoyed the same movies, the system may recommend movies liked by User B to User A.
        - **Item-based Collaborative Filtering**: This method analyzes the relationships between items. If a user likes a specific item (e.g., a movie), the algorithm looks for items that are similar to that one, based on what other users who liked the same item have also liked.
    - **Function**: By analyzing user interaction data (ratings, clicks, purchase history), collaborative filtering helps in personalizing recommendations, increasing user engagement, and enhancing the overall user experience.
2. **Content-Based Filtering**:
    - **Description**: This algorithm recommends items based on the characteristics of the items themselves and the user's past interactions. It evaluates the features of items that a user has liked in the past and suggests similar items.
    - **Function**: For example, if a user frequently watches action movies, the content-based algorithm would recommend other action movies, considering factors such as genre, director, actors, and keywords.
    - **Implementation**: This approach often involves natural language processing (NLP) to analyze item descriptions and extract relevant features, helping to build a profile of the user's preferences.
3. **Matrix Factorization**:
    - **Description**: This is a more advanced technique used to improve recommendation accuracy. Matrix factorization methods, such as Singular Value Decomposition (SVD), decompose the user-item interaction matrix into lower-dimensional matrices.

- **Function**: This allows the system to capture latent factors (hidden patterns) that explain user preferences and item characteristics. For instance, it can identify that users who enjoy romantic comedies also tend to like action-adventure films, even if they haven't explicitly rated them.
- **Benefits**: Matrix factorization helps address the sparsity problem in recommendation systems (where many items have few ratings) and improves the quality of recommendations.

4. **Hybrid Methods**:
   - **Description**: Many modern recommendation systems use a combination of collaborative filtering, content-based filtering, and matrix factorization to leverage the strengths of each approach while mitigating their weaknesses.
   - **Function**: For example, a hybrid system can use content-based filtering to initially recommend items to a new user (who may not have enough historical data for collaborative filtering) and gradually incorporate collaborative filtering as more data is collected over time.
   - **Advantages**: This approach leads to more accurate, diverse, and robust recommendations, improving user satisfaction and retention.

**For which values of nnn does insertion sort beat merge sort given that insertion sort runs in 8n28n^28n2 steps and merge sort runs in 64nlog⁡n64n \log n64nlogn steps?**

To determine the values of nnn for which insertion sort outperforms merge sort, we need to solve the inequality:

8n2<64nlog⁡n8n^2 < 64n \log n8n2<64nlogn

## Step 1: Simplifying the Inequality

First, we can simplify the inequality by dividing both sides by 8n8n8n (assuming n>0n > 0n>0):

n<8log⁡nn < 8 \log nn<8logn

## Step 2: Finding the Intersection Points

To find the values of nnn for which this inequality holds, we can analyze the functions f(n)=nf(n) = nf(n)=n and g(n)=8logng(n) = 8 \log ng(n)=8logn. We will find the intersection points by setting:

n=8lognn = 8 \log nn=8logn

## Step 3: Solving for nnn

1. **Graphical Method**: You can plot both functions f(n)f(n)f(n) and g(n)g(n)g(n) to visually find the intersection points. This can be done using graphing software or a calculator.

2. **Numerical Method**: Alternatively, we can try some values of nnn:
    - For n=1n = 1n=1:

      1<8log⁡(1)(since log⁡(1)=0)⇒1<0(False)1 < 8 \log(1) \quad \text{(since } \log(1) = 0\text{)} \Rightarrow 1 < 0 \quad \text{(False)}1<8log(1)(since log(1)=0)⇒1<0(False)

    - For n=2n = 2n=2:

      2<8log⁡(2)⇒2<8×0.693≈5.544(True)2 < 8 \log(2) \quad \Rightarrow 2 < 8 \times 0.693 \approx 5.544 \quad \text{(True)}2<8log(2)⇒2<8×0.693≈5.544(True)

    - For n=3n = 3n=3:

      3<8log⁡(3)⇒3<8×1.099≈8.792(True)3 < 8 \log(3) \quad \Rightarrow 3 < 8 \times 1.099 \approx 8.792 \quad \text{(True)}3<8log(3)⇒3<8×1.099≈8.792(True)

    - For n=4n = 4n=4:

      4<8log⁡(4)⇒4<8×1.386≈11.088(True)4 < 8 \log(4) \quad \Rightarrow 4 < 8 \times 1.386 \approx 11.088 \quad \text{(True)}4<8log(4)⇒4<8×1.386≈11.088(True)

    - For n=8n = 8n=8:

      8<8log⁡(8)⇒8<8×3≈24(True)8 < 8 \log(8) \quad \Rightarrow 8 < 8 \times 3 \approx 24 \quad \text{(True)}8<8log(8)⇒8<8×3≈24(True)

    - For n=16n = 16n=16:

      16<8log⁡(16)⇒16<8×4≈32(True)16 < 8 \log(16) \quad \Rightarrow 16 < 8 \times 4 \approx 32 \quad \text{(True)}16<8log(16)⇒16<8×4≈32(True)

    - For n=20n = 20n=20:

      20<8log⁡(20)⇒20<8×4.3219≈34.5752(True)20 < 8 \log(20) \quad \Rightarrow 20 < 8 \times 4.3219 \approx 34.5752 \quad \text{(True)}20<8log(20)⇒20<8×4.3219≈34.5752(True)

    - For n=32n = 32n=32:

      32<8log⁡(32)⇒32<8×5≈40(True)32 < 8 \log(32) \quad \Rightarrow 32 < 8 \times 5 \approx 40 \quad \text{(True)}32<8log(32)⇒32<8×5≈40(True)

    - For n=64n = 64n=64:

$$64 < 8 \log(64) \quad \Rightarrow 64 < 8 \times 6 \approx 48 \quad \text{(False)}$$

From this analysis, we see that insertion sort outperforms merge sort for nnn values up to around 43 or 44. To pinpoint this more accurately, you could calculate further or use numerical methods to find the exact transition point.

## Step 4: Conclusion

Insertion sort beats merge sort for nnn values approximately in the range $n < 43$. As nnn increases beyond this point, merge sort becomes more efficient than insertion sort.

## Final Answer

Insertion sort is faster than merge sort for values of nnn approximately in the range $n < 43$.

**What is the smallest value of nnn such that an algorithm whose running time is 100n2100n^2100n2 runs faster than an algorithm whose running time is 2n2^n2n on the same machine?**

To find the smallest value of nnn such that the algorithm with a running time of 100n2100n^2100n2 is faster than the one with a running time of 2n2^n2n, we need to solve the inequality:

$$100n^2 < 2^n$$

## Step 1: Analyzing the Inequality

We want to find the smallest integer nnn where 100n2100n^2100n2 is less than 2n2^n2n. Since 2n2^n2n grows exponentially, we can expect it to surpass 100n2100n^2100n2 for sufficiently large nnn.

## Step 2: Testing Values of nnn

We'll test integer values of nnn:

- **For $n = 1$**:

  $$100(1^2) = 100 \quad \text{and} \quad 2^1 = 2 \quad \Rightarrow \quad 100 < 2 \quad \text{(False)}$$

- **For n=2n = 2n=2**:

  $100(22)=400 \text{and} 22=4 \Rightarrow 400<4 \text{(False)} 100(2^2) = 400 \quad \text{and} \quad 2^2 = 4 \quad \Rightarrow \quad 400 < 4 \quad \text{(False)} 100(22)=400 \text{and} 22=4 \Rightarrow 400<4 \text{(False)}$

- **For n=3n = 3n=3**:

  $100(32)=900 \text{and} 23=8 \Rightarrow 900<8 \text{(False)} 100(3^2) = 900 \quad \text{and} \quad 2^3 = 8 \quad \Rightarrow \quad 900 < 8 \quad \text{(False)} 100(32)=900 \text{and} 23=8 \Rightarrow 900<8 \text{(False)}$

- **For n=4n = 4n=4**:

  $100(42)=1600 \text{and} 24=16 \Rightarrow 1600<16 \text{(False)} 100(4^2) = 1600 \quad \text{and} \quad 2^4 = 16 \quad \Rightarrow \quad 1600 < 16 \quad \text{(False)} 100(42)=1600 \text{and} 24=16 \Rightarrow 1600<16 \text{(False)}$

- **For n=5n = 5n=5**:

  $100(52)=2500 \text{and} 25=32 \Rightarrow 2500<32 \text{(False)} 100(5^2) = 2500 \quad \text{and} \quad 2^5 = 32 \quad \Rightarrow \quad 2500 < 32 \quad \text{(False)} 100(52)=2500 \text{and} 25=32 \Rightarrow 2500<32 \text{(False)}$

- **For n=6n = 6n=6**:

  $100(62)=3600 \text{and} 26=64 \Rightarrow 3600<64 \text{(False)} 100(6^2) = 3600 \quad \text{and} \quad 2^6 = 64 \quad \Rightarrow \quad 3600 < 64 \quad \text{(False)} 100(62)=3600 \text{and} 26=64 \Rightarrow 3600<64 \text{(False)}$

- **For n=7n = 7n=7**:

  $100(72)=4900 \text{and} 27=128 \Rightarrow 4900<128 \text{(False)} 100(7^2) = 4900 \quad \text{and} \quad 2^7 = 128 \quad \Rightarrow \quad 4900 < 128 \quad \text{(False)} 100(72)=4900 \text{and} 27=128 \Rightarrow 4900<128 \text{(False)}$

- **For n=8n = 8n=8**:

  $100(82)=6400 \text{and} 28=256 \Rightarrow 6400<256 \text{(False)} 100(8^2) = 6400 \quad \text{and} \quad 2^8 = 256 \quad \Rightarrow \quad 6400 < 256 \quad \text{(False)} 100(82)=6400 \text{and} 28=256 \Rightarrow 6400<256 \text{(False)}$

- **For n=9n = 9n=9**:

  $100(92)=8100 \text{and} 29=512 \Rightarrow 8100<512 \text{(False)} 100(9^2) = 8100 \quad \text{and} \quad 2^9 = 512 \quad \Rightarrow \quad 8100 < 512 \quad \text{(False)} 100(92)=8100 \text{and} 29=512 \Rightarrow 8100<512 \text{(False)}$

- **For n=10n = 10n=10**:

  100(102)=10000and210=1024⇒10000<1024(False)100(10^2) = 10000 \quad \text{and} \quad 2^{10} = 1024 \quad \Rightarrow \quad 10000 < 1024 \quad \text{(False)}100(102)=10000and210=1024⇒10000<1024(False)

- **For n=11n = 11n=11**:

  100(112)=12100and211=2048⇒12100<2048(False)100(11^2) = 12100 \quad \text{and} \quad 2^{11} = 2048 \quad \Rightarrow \quad 12100 < 2048 \quad \text{(False)}100(112)=12100and211=2048⇒12100<2048(False)

- **For n=12n = 12n=12**:

  100(122)=14400and212=4096⇒14400<4096(False)100(12^2) = 14400 \quad \text{and} \quad 2^{12} = 4096 \quad \Rightarrow \quad 14400 < 4096 \quad \text{(False)}100(122)=14400and212=4096⇒14400<4096(False)

- **For n=13n = 13n=13**:

  100(132)=16900and213=8192⇒16900<8192(False)100(13^2) = 16900 \quad \text{and} \quad 2^{13} = 8192 \quad \Rightarrow \quad 16900 < 8192 \quad \text{(False)}100(132)=16900and213=8192⇒16900<8192(False)

- **For n=14n = 14n=14**:

  100(142)=19600and214=16384⇒19600<16384(False)100(14^2) = 19600 \quad \text{and} \quad 2^{14} = 16384 \quad \Rightarrow \quad 19600 < 16384 \quad \text{(False)}100(142)=19600and214=16384⇒19600<16384(False)

- **For n=15n = 15n=15**:

  100(152)=22500and215=32768⇒22500<32768(True)100(15^2) = 22500 \quad \text{and} \quad 2^{15} = 32768 \quad \Rightarrow \quad 22500 < 32768 \quad \text{(True)}100(152)=22500and215=32768⇒22500<32768(True)

## Conclusion

After testing these values, we find that the smallest nnn for which 100n2<2n100n^2 < 2^n100n2<2n is:15

# *CHAPTER NO 2:*
# *GETTING STARTED*

**Chapter 2 Exercises**

---

**2.1 Exercises**

---

**2.1-1**
**Question:** Using Figure 2.2 as a model, illustrate the operation of **INSERTION-SORT** on an array initially containing the sequence ⟨31,41,59,26,41,58⟩\langle 31, 41, 59, 26, 41, 58 \rangle⟨31,41,59,26,41,58⟩.
**Answer:**
To demonstrate **INSERTION-SORT**, we proceed by moving each element into its correct position in the sorted subarray.

1. **Initial Array:** ⟨31,41,59,26,41,58⟩\langle 31, 41, 59, 26, 41, 58 \rangle⟨31,41,59,26,41,58⟩
2. **Step 1:** 31 and 41 are already in order. ⟨31,41,59,26,41,58⟩\langle 31, 41, 59, 26, 41, 58 \rangle⟨31,41,59,26,41,58⟩
3. **Step 2:** 59 is already greater than 41, so no change. ⟨31,41,59,26,41,58⟩\langle 31, 41, 59, 26, 41, 58 \rangle⟨31,41,59,26,41,58⟩
4. **Step 3:** Insert 26. Shift 59, 41, and 31 to the right. Result: ⟨26,31,41,59,41,58⟩\langle 26, 31, 41, 59, 41, 58 \rangle⟨26,31,41,59,41,58⟩
5. **Step 4:** Insert 41. Shift 59 to the right. Result: ⟨26,31,41,41,59,58⟩\langle 26, 31, 41, 41, 59, 58 \rangle⟨26,31,41,41,59,58⟩
6. **Step 5:** Insert 58. Shift 59 to the right. Result: ⟨26,31,41,41,58,59⟩\langle 26, 31, 41, 41, 58, 59 \rangle⟨26,31,41,41,58,59⟩

Final sorted array: ⟨26,31,41,41,58,59⟩\langle 26, 31, 41, 41, 58, 59 \rangle⟨26,31,41,41,58,59⟩.

---

**Question:**
Consider the procedure **SUM-ARRAY** that computes the sum of the nnn numbers in an array

A[1…n]A[1 \dots n]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]A[1 \dots n]A[1…n].

The pseudocode for **SUM-ARRAY** is as follows:

```
bash
Copy code
SUM-ARRAY(A, n)
1  sum = 0
2  for i = 1 to n
3       sum = sum + A[i]
4  return sum
```

---

**Solution:**

To prove the correctness of the **SUM-ARRAY** procedure, we will define a loop invariant and then analyze it through the initialization, maintenance, and termination properties. This approach will show that the procedure correctly computes the sum of the elements in A[1…n]A[1 \dots n]A[1…n].

## Loop Invariant

The **loop invariant** for the procedure can be defined as follows:

- **Loop Invariant**: At the start of each iteration of the loop, the variable `sum` holds the sum of the elements of A[1…i−1]A[1 \dots i-1]A[1…i−1], where `i` is the current loop counter.

This invariant asserts that as the loop progresses, `sum` accumulates the sum of elements from the beginning of the array up to, but not including, the current element A[i]A[i]A[i].

## Proof Using Initialization, Maintenance, and Termination

1. **Initialization**:
   Before the loop begins, `i = 1`, and `sum` is initialized to 0 (line 1). According to our loop invariant, `sum` should hold the sum of elements A[1…0]A[1 \dots 0]A[1…0], which is an empty sum. The sum of no elements is indeed zero, which satisfies our invariant. Therefore, the loop invariant holds at initialization.
2. **Maintenance**:
   Assuming that the loop invariant holds at the start of a given iteration, we need to show that it continues to hold at the next iteration.
   - During each loop iteration (line 3), the current element A[i]A[i]A[i] is added to `sum`.
   - Then, `i` is incremented by 1 as the loop proceeds.

By the end of the iteration, `sum` now represents the sum of the elements A[1…i]A[1 \dots i]A[1…i]. Therefore, at the start of the next iteration, `sum` correctly holds the sum of elements A[1…i−1]A[1 \dots i-1]A[1…i−1], maintaining the loop invariant.

3. **Termination**:
   The loop terminates when `i = n + 1`. At this point, according to our loop invariant, `sum` holds the sum of all elements A[1…n]A[1 \dots n]A[1…n], which is the required result.

**Question:**
Consider the procedure **SUM-ARRAY** that computes the sum of the nnn numbers in an array A[1…n]A[1 \dots n]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]A[1 \dots n]A[1…n].

The pseudocode for **SUM-ARRAY** is as follows:

```
bash
Copy code
SUM-ARRAY(A, n)
1  sum = 0
2  for i = 1 to n
3      sum = sum + A[i]
4  return sum
```

---

**Solution:**

To prove the correctness of the **SUM-ARRAY** procedure, we will define a loop invariant and then analyze it through the initialization, maintenance, and termination properties. This approach will show that the procedure correctly computes the sum of the elements in A[1…n]A[1 \dots n]A[1…n].

## Loop Invariant

The **loop invariant** for the procedure can be defined as follows:

- **Loop Invariant**: At the start of each iteration of the loop, the variable `sum` holds the sum of the elements of A[1…i−1]A[1 \dots i-1]A[1…i−1], where `i` is the current loop counter.

This invariant asserts that as the loop progresses, `sum` accumulates the sum of elements from the beginning of the array up to, but not including, the current element A[i]A[i]A[i].

## Proof Using Initialization, Maintenance, and Termination

1. **Initialization**:
   Before the loop begins, `i = 1`, and `sum` is initialized to 0 (line 1). According to our loop invariant, `sum` should hold the sum of elements A[1…0]A[1 \dots 0]A[1…0], which is an

empty sum. The sum of no elements is indeed zero, which satisfies our invariant. Therefore, the loop invariant holds at initialization.

2. **Maintenance**:
   Assuming that the loop invariant holds at the start of a given iteration, we need to show that it continues to hold at the next iteration.
   - During each loop iteration (line 3), the current element A[i]A[i]A[i] is added to `sum`.
   - Then, `i` is incremented by 1 as the loop proceeds.

   By the end of the iteration, `sum` now represents the sum of the elements A[1…i]A[1 \dots i]A[1…i]. Therefore, at the start of the next iteration, `sum` correctly holds the sum of elements A[1…i−1]A[1 \dots i-1]A[1…i−1], maintaining the loop invariant.

3. **Termination**:
   The loop terminates when `i = n + 1`. At this point, according to our loop invariant, `sum` holds the sum of all elements A[1…n]A[1 \dots n]A[1…n], which is the required result.

**Question:**
Rewrite the **INSERTION-SORT** procedure to sort an array in **monotonically decreasing** order instead of **monotonically increasing** order.

---

**Solution:**

To modify the **INSERTION-SORT** algorithm to sort an array in decreasing order, we only need to adjust the comparison condition in the inner loop. The original **INSERTION-SORT** sorts in increasing order by moving elements that are greater than the current element one position to the right. To achieve a decreasing order, we should instead move elements that are **less than** the current element one position to the right.

Here is the pseudocode for the modified **INSERTION-SORT**:

```
css
Copy code
INSERTION-SORT-DECREASING(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 ... j-1]
4      i = j - 1
5      while i > 0 and A[i] < key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```

## Explanation of Each Step:

1. **Outer Loop** (`for j = 2 to A.length`):

- The outer loop starts at the second element (index `j = 2`) and iterates through each element in the array. The purpose of this loop is to insert each element `A[j]` into the correct position in the sorted portion of the array from `A[1]` to `A[j-1]`.

2. **Initialize `key`** (`key = A[j]`):
   - `key` holds the value of the current element `A[j]` that we need to insert into the sorted portion of the array.
3. **Inner Loop Condition** (`while i > 0 and A[i] < key`):
   - The inner `while` loop finds the correct position for `key` by shifting elements that are **less than `key`** to the right.
   - The condition `A[i] < key` ensures that `key` will be placed in the correct position for a decreasing order, since elements less than `key` will be shifted to the right, making room for `key` in the sorted sequence.
4. **Shift Elements** (`A[i + 1] = A[i]`):
   - Inside the `while` loop, each element `A[i]` that is less than `key` is shifted one position to the right (i.e., to `A[i + 1]`), making room for `key` to be placed in its correct position.
5. **Insert `key` in Correct Position** (`A[i + 1] = key`):
   - After the inner loop exits, `key` is placed in its correct position within the sorted portion of the array.

## Example

Let's take an example to illustrate how this works.

Given array: `A = [5, 2, 9, 3, 7]`

1. **j = 2**: `key = 2`
   - Compare `A[1] = 5` with `key = 2` (no shift needed).
   - Array: `[5, 2, 9, 3, 7]`
2. **j = 3**: `key = 9`
   - Compare `A[2] = 2` with `key = 9` → Shift `2`.
   - Compare `A[1] = 5` with `key = 9` → Shift `5`.
   - Insert `key` at `A[1]`.
   - Array: `[9, 5, 2, 3, 7]`
3. **j = 4**: `key = 3`
   - Compare `A[3] = 2` with `key = 3` → Shift `2`.
   - Insert `key` at `A[3]`.
   - Array: `[9, 5, 3, 2, 7]`
4. **j = 5**: `key = 7`
   - Compare `A[4] = 2` with `key = 7` → Shift `2`.
   - Compare `A[3] = 3` with `key = 7` → Shift `3`.
   - Compare `A[2] = 5` with `key = 7` → Shift `5`.
   - Insert `key` at `A[2]`.
   - Final Array: `[9, 7, 5, 3, 2]`

**Question:**
Consider the **searching problem**:

**Input**: A sequence of nnn numbers ⟨a1,a2,…,an⟩\langle a_1, a_2, \dots, a_n \rangle⟨a1,a2,…,an⟩ stored in an array A[1…n]A[1 \dots n]A[1…n] and a value xxx.
**Output**: An index iii such that x=A[i]x = A[i]x=A[i] or a special value `NIL` if xxx does not appear in AAA.

Write pseudocode for **linear search**, which scans through the array from beginning to end, looking for xxx. Using a **loop invariant**, prove that the algorithm is correct. Ensure that the loop invariant fulfills the three necessary properties (initialization, maintenance, and termination).

---

**Solution:**

The **Linear Search** algorithm iterates over each element in the array, comparing it to xxx. If it finds xxx in the array, it returns the index of the element. If the loop completes without finding xxx, it returns `NIL`.

## Pseudocode for Linear Search

```
css
Copy code
LINEAR-SEARCH(A, x)
1  for i = 1 to A.length
2      if A[i] == x
3          return i
4  return NIL
```

## Explanation of the Pseudocode

1. The `for` loop iterates from index `1` to `n` (where `n` is the length of array `A`).
2. Inside the loop, it checks if `A[i]` equals `x`.
    - If it finds xxx at index iii, it returns iii.
3. If the loop completes and no element equals xxx, the function returns `NIL` to indicate that xxx is not present in the array.

## Loop Invariant

To prove the correctness of this **Linear Search** algorithm, let's define a loop invariant and demonstrate that it fulfills the three properties: initialization, maintenance, and termination.

- **Loop Invariant**: At the start of each iteration of the loop, for each index k<ik < ik<i, either `A[k]` $\neq$ `x` or the algorithm has already returned the correct index kkk.

This invariant ensures that if xxx is found, the algorithm will return its index immediately, and if the loop terminates, no index contains xxx.

## Proof Using Initialization, Maintenance, and Termination

1. **Initialization**:
   Before the loop begins (when `i = 1`), no elements have been checked. The invariant holds because there are no elements with indices less than `i = 1`, so the condition is trivially satisfied.
2. **Maintenance**:
   Assume the loop invariant holds at the start of an iteration for some `i`. During this iteration:
   - If `A[i] == x`, the algorithm immediately returns `i`, which is the correct index.
   - If `A[i] ≠ x`, the loop proceeds to the next iteration with `i` incremented by 1.

   Since `A[i]` was checked and did not match xxx, the invariant remains true for all indices k<i+1k < i+1k<i+1.

3. **Termination**:
   The loop terminates when `i = n + 1`, meaning all indices from `1` to `n` have been checked. At this point:
   - If xxx was found in `A`, it would have been returned during the loop.
   - If xxx was not found, the algorithm reaches line 4 and returns `NIL`.

**Question:**
Consider the problem of adding two nnn-bit binary integers aaa and bbb, stored in two nnn-element arrays A[0…n−1]A[0 \dots n-1]A[0…n−1] and B[0…n−1]B[0 \dots n-1]B[0…n−1], where each element is either 0 or 1. Define a=∑i=0n−1A[i]·2ia = \sum_{i=0}^{n-1} A[i] \cdot 2^ia=∑i=0n−1A[i]·2i and b=∑i=0n−1B[i]·2ib = \sum_{i=0}^{n-1} B[i] \cdot 2^ib=∑i=0n−1 B[i]·2i. The sum c=a+bc = a + bc=a+b of the two integers should be stored in binary form in an (n+1)(n+1)(n+1)-element array C[0…n]C[0 \dots n]C[0…n], where c=∑i=0nC[i]·2ic = \sum_{i=0}^{n} C[i] \cdot 2^ic=∑i=0nC[i]·2i. Write a procedure **ADD-BINARY-INTEGERS** that takes as input arrays AAA and BBB, along with the length nnn, and returns array CCC holding the sum.

---

**Solution:**

To add two binary integers stored in arrays AAA and BBB, we can perform binary addition from the least significant bit (index 0) to the most significant bit (index n−1n-1n−1). This approach is similar to how addition is done manually, where we add corresponding bits and handle carries.

## Pseudocode for ADD-BINARY-INTEGERS

The following pseudocode calculates the binary sum of two nnn-bit numbers represented by arrays AAA and BBB, and stores the result in array CCC.

```
less
Copy code
ADD-BINARY-INTEGERS(A, B, n)
1   let C be a new array of size n + 1
```

```
2   carry = 0
3   for i = 0 to n - 1
4       sum = A[i] + B[i] + carry
5       C[i] = sum % 2          // Store the bit result of the addition
6       carry = sum // 2        // Update carry for the next higher bit
7   C[n] = carry                // Store the final carry in the last position
8   return C
```

## Explanation of Each Step:

1. **Initialize the Result Array `C`**:
   Create an array `C` of size n+1n+1n+1, which will store the resulting sum. This size accounts for a possible carry in the most significant bit.
2. **Initialize `carry` to 0**:
   Set `carry` to zero initially, as we start adding from the least significant bit.
3. **Iterate Over Each Bit Position `i`** (`for i = 0 to n - 1`):
   For each bit position iii:
   - **Compute the Sum** (`sum = A[i] + B[i] + carry`):
     Add the bits `A[i]` and `B[i]`, along with any carry from the previous bit.
   - **Store the Result Bit** (`C[i] = sum % 2`):
     Since each bit can only be 0 or 1, the expression `sum % 2` gives the result bit for the position iii. This result bit is stored in `C[i]`.
   - **Update the Carry** (`carry = sum // 2`):
     Update `carry` to `sum // 2`, which is 1 if the sum is 2 or 3 (indicating a carry) and 0 otherwise.
4. **Store the Final Carry in `C[n]`**:
   After the loop completes, if there is a carry left, it is stored in `C[n]`, the most significant position in `C`.
5. **Return `C`**:
   The array `C` now holds the binary representation of a+ba + ba+b in n+1n+1n+1 bits.

## Example

Let's work through an example:

Given:

- A=[1,0,1,1]A = [1, 0, 1, 1]A=[1,0,1,1] (binary for 13 in decimal)
- B=[1,1,0,1]B = [1, 1, 0, 1]B=[1,1,0,1] (binary for 11 in decimal)
- n=4n = 4n=4

**Steps:**

1. **Initialize**: `C = [0, 0, 0, 0, 0]`, `carry = 0`
2. **Iteration 1** (i = 0):
   - `sum = A[0] + B[0] + carry = 1 + 1 + 0 = 2`
   - `C[0] = sum % 2 = 2 % 2 = 0`

    - `carry = sum // 2 = 2 // 2 = 1`
    - `C = [0, 0, 0, 0, 0]`
3. **Iteration 2** (i = 1):
    - `sum = A[1] + B[1] + carry = 0 + 1 + 1 = 2`
    - `C[1] = sum % 2 = 2 % 2 = 0`
    - `carry = sum // 2 = 2 // 2 = 1`
    - `C = [0, 0, 0, 0, 0]`
4. **Iteration 3** (i = 2):
    - `sum = A[2] + B[2] + carry = 1 + 0 + 1 = 2`
    - `C[2] = sum % 2 = 2 % 2 = 0`
    - `carry = sum // 2 = 2 // 2 = 1`
    - `C = [0, 0, 0, 0, 0]`
5. **Iteration 4** (i = 3):
    - `sum = A[3] + B[3] + carry = 1 + 1 + 1 = 3`
    - `C[3] = sum % 2 = 3 % 2 = 1`
    - `carry = sum // 2 = 3 // 2 = 1`
    - `C = [0, 0, 0, 1, 0]`
6. **Final Carry**:
    - `C[4] = carry = 1`
    - Final `C = [0, 0, 0, 1, 1]` (binary for 24 in decimal)

**Question:**
Express the function f(n)=n31000+100n2+100n+3f(n) = \frac{n^3}{1000} + 100n^2 + 100n + 3f(n)=1000n3+100n2+100n+3 in terms of **Big-O notation**.

---

**Solution:**

To express f(n)=n31000+100n2+100n+3f(n) = \frac{n^3}{1000} + 100n^2 + 100n + 3f(n)=1000n3+100n2+100n+3 in Big-O notation, we need to determine the asymptotic upper bound of the function as nnn grows large. Here's a step-by-step solution:

## Step 1: Analyze Each Term's Growth Rate

The function has four terms:

1. n31000\frac{n^3}{1000}1000n3
2. 100n2100n^2100n2
3. 100n100n100n
4. 333

As n→∞n \to \inftyn→∞, terms with higher powers of nnn will grow faster than those with lower powers. Therefore:

- n3n^3n3 grows faster than n2n^2n2, nnn, or any constant.
- Thus, n31000\frac{n^3}{1000}1000n3 is the **dominant term** that determines the asymptotic behavior of f(n)f(n)f(n).

## Step 2: Simplify by Ignoring Lower-Order Terms

In Big-O notation, we only consider the term with the highest growth rate. The terms 100n2100n^2100n2, 100n100n100n, and 333 become insignificant relative to n3n^3n3 as nnn increases.

Therefore, we can simplify the expression to focus only on the dominant term:

f(n)≈n31000f(n) \approx \frac{n^3}{1000}f(n)≈1000n3

## Step 3: Ignore Constants in Big-O Notation

Big-O notation only captures the growth rate and ignores constant coefficients. Therefore, we remove the constant 11000\frac{1}{1000}10001 from the dominant term:

f(n)=O(n3)f(n) = O(n^3)f(n)=O(n3)

## Conclusion

The function f(n)=n31000+100n2+100n+3f(n) = \frac{n^3}{1000} + 100n^2 + 100n + 3f(n)=1000n3+100n2+100n+3 can be expressed in Big-O notation as:

f(n)=O(n3)f(n) = O(n^3)f(n)=O(n3)

**Question:**
Consider sorting nnn numbers stored in an array A[1…n]A[1 \dots n]A[1…n] by first finding the smallest element in A[1…n]A[1 \dots n]A[1…n] and exchanging it with the element in A[1]A[1]A[1]. Then, find the smallest element in A[2…n]A[2 \dots n]A[2…n] and exchange it with A[2]A[2]A[2]. Next, find the smallest element in A[3…n]A[3 \dots n]A[3…n] and exchange it with A[3]A[3]A[3]. Continue this process for the first n−1n - 1n−1 elements of AAA. 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−1n - 1n−1 elements, rather than for all nnn elements? Provide the worst-case running time of selection sort in Big-O notation. Is the best-case running time any better?

---

**Solution:**

## Pseudocode for Selection Sort

The **Selection Sort** algorithm works by repeatedly finding the smallest unsorted element in the array and swapping it with the element at the current position.

```
css
Copy code
```

```
SELECTION-SORT(A, n)
1  for i = 1 to n - 1
2      min_index = i
3      for j = i + 1 to n
4          if A[j] < A[min_index]
5              min_index = j
6      swap A[i] with A[min_index]
```

## Explanation of the Pseudocode

1. **Outer Loop**:
   The outer loop runs from `i = 1` to `n - 1`, with each iteration moving the boundary of the sorted portion of the array one element forward.
2. **Find the Minimum**:
   For each position iii, the algorithm initializes `min_index` to iii. Then, the inner loop finds the smallest element in the subarray A[i+1…n]A[i+1 \dots n]A[i+1…n] and updates `min_index` accordingly.
3. **Swap Elements**:
   After finding the minimum element in the unsorted portion of the array, the algorithm swaps this element with `A[i]`, effectively growing the sorted portion of the array by one element.

## Loop Invariant

The loop invariant for the outer loop is as follows:

- **Loop Invariant**: At the start of each iteration of the outer loop (index iii), the subarray A[1…i−1]A[1 \dots i - 1]A[1…i−1] contains the smallest i−1i - 1i−1 elements of the original array, sorted in ascending order.

## Why the Algorithm Runs Only n−1n - 1n−1 Times

- The algorithm only needs to iterate through the first n−1n - 1n−1 elements because, after placing the smallest n−1n - 1n−1 elements in their correct positions, the nnn-th element will already be in its correct position as the largest remaining unsorted element.

## Worst-Case Running Time

Selection Sort's running time can be analyzed as follows:

1. For each iii, the algorithm performs a linear scan of the remaining unsorted elements to find the minimum element, taking n−in - in−i comparisons.
2. Summing up these comparisons gives: T(n)=(n−1)+(n−2)+⋯+1=n(n−1)2=O(n2)T(n) = (n - 1) + (n - 2) + \dots + 1 = \frac{n(n - 1)}{2} = O(n^2)T(n)=(n−1)+(n−2)+⋯+1=2n(n−1)=O(n2)

Thus, the **worst-case running time** of Selection Sort is O(n2)O(n^2)O(n2).

## Best-Case Running Time

Selection Sort's running time remains O(n2)O(n^2)O(n2) in the best case because it still needs to scan and compare each element to determine the minimum, regardless of the array's initial order.

Therefore, **best-case running time** is also O(n2)O(n^2)O(n2).

**Question:**
Consider linear search. How many elements of the input array need to be checked on **average**, assuming that the element being searched for is equally likely to be any element in the array? What about in the **worst case**?

---

**Solution:**

## Linear Search Overview

In a linear search, we scan through each element of the array sequentially until we either find the target element or reach the end of the array. This search method has different performance depending on the position of the element being searched for.

## Average-Case Analysis

Assuming:

- The array has nnn elements.
- The element we are searching for is equally likely to be at any position from the first to the last.

To calculate the average number of elements checked, consider the position of the target element:

- If the target is the **first** element, we need **1 check**.
- If it's the **second** element, we need **2 checks**.
- Similarly, if it's the **kkk-th** element, we need **kkk checks**.

If the element is equally likely to be at any position, the expected (average) number of checks can be calculated by averaging the positions:

Average number of checks=1+2+3+⋯+nn\text{Average number of checks} = \frac{1 + 2 + 3 + \dots + n}{n}Average number of checks=n1+2+3+⋯+n

The sum 1+2+3+⋯+n1 + 2 + 3 + \dots + n1+2+3+⋯+n can be simplified to n(n+1)2\frac{n(n + 1)}{2}2n(n+1), so we get:

Average number of checks=n(n+1)2n=n+12\text{Average number of checks} = \frac{\frac{n(n + 1)}{2}}{n} = \frac{n + 1}{2}Average number of checks=n2n(n+1)=2n+1

Therefore, on **average**, the linear search will check **n+12\frac{n + 1}{2}2n+1 elements**.

## Worst-Case Analysis

In the **worst case**:

- The target element is not present in the array, or it's the last element in the array.
- In this scenario, we need to check every element in the array, so we perform **nnn checks**.

Thus, the **worst-case number of checks** is **nnn**.

## Summary

- **Average case**: n+12\frac{n + 1}{2}2n+1 checks.
- **Worst case**: nnn checks.

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

---

**Solution:**

To improve the best-case running time of any sorting algorithm, we can incorporate a **pre-check** at the start of the algorithm to see if the array is already sorted. If the array is sorted, we can immediately return it without performing any sorting operations, which will make the best-case running time **O(n)** for all sorting algorithms.

## Detailed Explanation of the Modification

1. **Add a Pre-check**:
   - Before beginning the main sorting procedure, iterate through the array once to check if each element is less than or equal to the next element.
   - This requires one scan of the array, which takes O(n)O(n)O(n) time.
2. **Conditions for the Pre-check**:
   - If every element A[i]≤A[i+1]A[i] \leq A[i + 1]A[i]≤A[i+1] for all iii from 1 to n−1n-1n−1, then the array is already sorted in ascending order.
   - If the check confirms the array is sorted, terminate the algorithm and return the array as-is.
3. **Impact on Best-Case Running Time**:
   - With this modification, the algorithm will immediately finish in O(n)O(n)O(n) time if the array is already sorted, which becomes the **best-case running time**.

- This modification does not affect the average or worst-case running times, as the main sorting procedure will only be executed if the array is not sorted.

## Example: Applying This to Bubble Sort

Consider Bubble Sort, which normally has a best-case running time of O(n2)O(n^2)O(n2) due to its nested loop structure, even if the array is already sorted. By applying this pre-check:

1. Perform a single pass to check if the array is sorted in O(n)O(n)O(n) time.
2. If sorted, return immediately.
3. If not, proceed with the usual Bubble Sort algorithm.

With this pre-check, Bubble Sort achieves a best-case running time of **O(n)** while retaining its average and worst-case O(n2)O(n^2)O(n2) complexity.

## Summary

By adding a pre-check to verify if an array is already sorted, we can make the best-case running time for any sorting algorithm O(n)O(n)O(n). This optimization is especially useful in scenarios where data may already be sorted or nearly sorted, thus reducing unnecessary operations.

**Question:** 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⟩\langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle⟨3,41,52,26,38,57,9,49⟩.
**Answer:**
To illustrate **merge sort**, we will divide the array, sort each half, and then merge the sorted halves. Here are the steps:

1. **Initial 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⟩
2. **Divide Step 1:** Split into ⟨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⟩
3. **Divide Step 2:** Further split each half:
    - ⟨3,41⟩\langle 3, 41 \rangle⟨3,41⟩, ⟨52,26⟩\langle 52, 26 \rangle⟨52,26⟩
    - ⟨38,57⟩\langle 38, 57 \rangle⟨38,57⟩, ⟨9,49⟩\langle 9, 49 \rangle⟨9,49⟩
4. **Divide Step 3:** Split into individual elements:
    - ⟨3⟩\langle 3 \rangle⟨3⟩, ⟨41⟩\langle 41 \rangle⟨41⟩, ⟨52⟩\langle 52 \rangle⟨52⟩, ⟨26⟩\langle 26 \rangle⟨26⟩, ⟨38⟩\langle 38 \rangle⟨38⟩, ⟨57⟩\langle 57 \rangle⟨57⟩, ⟨9⟩\langle 9 \rangle⟨9⟩, ⟨49⟩\langle 49 \rangle⟨49⟩
5. **Merge Step 1:** Merge individual elements into sorted pairs:

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

6. **Merge Step 2:** Merge pairs into sorted quadruples:
   - ⟨3,26,41,52⟩\langle 3, 26, 41, 52 \rangle⟨3,26,41,52⟩, ⟨9,38,49,57⟩\langle 9, 38, 49, 57 \rangle⟨9,38,49,57⟩
7. **Merge Step 3:** Merge the final two sorted halves to get:
   - ⟨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⟩

Final 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⟩.

**Question:**
The test in line 1 of the **MERGE-SORT** procedure reads `<if p < r` rather than `<if p ≤ r`. If **MERGE-SORT** is called with p>rp > rp>r, then the subarray A[p…r]A[p \dots r]A[p…r] is empty. Argue that as long as the initial call of **MERGE-SORT(A, 1, n)** has n≥1n \geq 1n≥1, the test `<if p < r` suffices to ensure that no recursive call has p>rp > rp>r.

---

**Solution:**

## Understanding the MERGE-SORT Algorithm

The **MERGE-SORT** algorithm works by dividing an array into two halves, sorting each half recursively, and then merging the sorted halves. The key part of this algorithm is the condition that determines whether to continue sorting a subarray.

## Recursive Calls in MERGE-SORT

1. The initial call to **MERGE-SORT** is made with the parameters AAA, 111, and nnn.
2. The condition `if p < r` checks whether the subarray has more than one element. If ppp is not less than rrr, it indicates that the subarray is either empty or contains a single element.

## Analyzing the Condition `if p < r`

- When **MERGE-SORT** is called initially with MERGE−SORT(A,1,n)MERGE-SORT(A, 1, n)MERGE−SORT(A,1,n), we have p=1p = 1p=1 and r=nr = nr=n. Thus, p≤rp \leq rp≤r holds true when n≥1n \geq 1n≥1.
- As the algorithm proceeds, the array is divided:
  - The midpoint qqq is calculated as q=⌊(p+r)/2⌋q = \lfloor (p + r) / 2 \rfloorq=⌊(p+r)/2⌋.

- Subsequent recursive calls will look like MERGE−SORT(A,p,q)MERGE-SORT(A, p, q)MERGE−SORT(A,p,q) and MERGE−SORT(A,q+1,r)MERGE-SORT(A, q + 1, r)MERGE−SORT(A,q+1,r).

## Ensuring p≤rp \leq rp≤r in Recursive Calls

- When we call MERGE−SORT(A,p,q)MERGE-SORT(A, p, q)MERGE−SORT(A,p,q):
  - Since qqq is always the floor of the average of ppp and rrr, it guarantees that p≤qp \leq qp≤q (because ppp is the left endpoint).
- When we call MERGE−SORT(A,q+1,r)MERGE-SORT(A, q + 1, r)MERGE−SORT(A,q+1,r):
  - Here, q+1q + 1q+1 is incremented from qqq. Since q≤rq \leq rq≤r, it ensures q+1≤rq + 1 \leq rq+1≤r.

Thus, both recursive calls guarantee that ppp will never exceed rrr. Therefore, as long as the initial call is valid (with n≥1n \geq 1n≥1), all subsequent calls maintain the invariant p≤rp \leq rp≤r.

## Question:

**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.**

---

## Answer:

To analyze the correctness of the MERGE procedure, we need to establish loop invariants for the relevant while loops. Here's a structured approach:

### *1. Loop Invariant for the While Loop at Line 12318*

Assuming this loop is responsible for merging two sorted subarrays AAA and BBB into a single array CCC:

- **Invariant:** At the start of each iteration of the loop, the first iii elements of the merged array CCC are sorted and contain the smallest iii elements from the input arrays AAA and BBB.

### *2. Proving the Loop Invariant*

To demonstrate that this invariant holds, we follow three steps: initialization, maintenance, and termination.

**Initialization:**

- Before the first iteration of the loop (when i=0i = 0i=0), no elements have been added to CCC. Thus, the first 000 elements of CCC are trivially sorted and contain the smallest 000 elements from AAA and BBB. The invariant holds.

**Maintenance:**

- Assume the invariant holds at the beginning of the kthk^{th}kth iteration (the first kkk elements of CCC are sorted and contain the smallest kkk elements from AAA and BBB).
- During the kthk^{th}kth iteration, the loop compares A[j]A[j]A[j] and B[k]B[k]B[k], adding the smaller element to CCC and incrementing the appropriate index. After this addition, the first k+1k+1k+1 elements of CCC are still sorted and include the smallest k+1k+1k+1 elements from AAA and BBB. Therefore, the invariant is maintained for the next iteration.

**Termination:**

- The loop terminates when all elements from either AAA or BBB have been added to CCC. At this point, the first nnn elements of CCC (where nnn is the total number of elements in AAA and BBB) are sorted and include all elements from both arrays. Thus, the invariant guarantees that CCC is a sorted array containing all elements from AAA and BBB.

### *3. Using Other Loops to Prove Overall Correctness*

Assuming lines 20323 and 24327 contain additional while loops that also contribute to the MERGE procedure, we can establish invariants for those loops as well.

**While Loop at Line 20323:**

- **Invariant:** After this loop, all remaining elements from the input array AAA or BBB (whichever is not yet fully traversed) are copied to CCC in sorted order.

**While Loop at Line 24327:**

- **Invariant:** Ensures that any remaining elements (if any) are handled correctly, maintaining the sorted order of CCC.

### *4. Proving the Correctness of the MERGE Procedure*

To prove the overall correctness of the MERGE procedure, we combine the established invariants:

1. **Combine the Invariants:**
    - Show that each invariant preserves the correctness of the overall merge process:
        - The first loop (12318) ensures that the elements are merged in sorted order.
        - The second loop (20323) ensures that any remaining elements from one of the input arrays are added correctly.
        - The third loop (24327) handles any additional elements and guarantees the final sorted order.

2. **Conclusion:**
    - By establishing that each loop maintains its respective invariant and that these invariants collectively contribute to the sorted nature of the output CCC, we conclude that the MERGE procedure correctly merges two sorted lists into a single sorted list.

## Question:

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

T(n)={2if n=22T(n2)+nif n>2T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 2T\left(\frac{n}{2}\right) + n & \text{if } n > 2 \end{cases}T(n)={22T(2n)+nif n=2if n>2

is T(n)=nlg⁡nT(n) = n \lg nT(n)=nlgn.

---

## Answer:

To prove that T(n)=nlg⁡nT(n) = n \lg nT(n)=nlgn using mathematical induction, we will follow these steps:

### *1. Base Case:*

We start with the smallest power of 2 greater than or equal to 2, which is n=2n = 2n=2.

- For n=2n = 2n=2:

  T(2)=2T(2) = 2T(2)=2

  According to the formula T(n)=nlg⁡nT(n) = n \lg nT(n)=nlgn:

  T(2)=2lg⁡2=2⋅1=2T(2) = 2 \lg 2 = 2 \cdot 1 = 2T(2)=2lg2=2⋅1=2

Thus, the base case holds: T(2)=2lg⁡2T(2) = 2 \lg 2T(2)=2lg2.

### *2. Inductive Step:*

Assume that the formula holds for some kkk, where kkk is an exact power of 2 and k≥2k \geq 2k≥2. This means we assume:

T(k)=klg⁡kT(k) = k \lg kT(k)=klgk

Now we need to prove that it also holds for n=2kn = 2kn=2k (the next power of 2).

- For n=2kn = 2kn=2k:

T(2k)=2T(k)+2kT(2k) = 2T(k) + 2kT(2k)=2T(k)+2k

By the inductive hypothesis, substitute T(k)T(k)T(k):

T(2k)=2(klg⁡k)+2kT(2k) = 2(k \lg k) + 2kT(2k)=2(klgk)+2k

Simplifying this expression:

T(2k)=2klg⁡k+2kT(2k) = 2k \lg k + 2kT(2k)=2klgk+2k T(2k)=2k(lg⁡k+1)T(2k) = 2k(\lg k + 1)T(2k)=2k(lgk+1)

Using the property of logarithms lg⁡k+1=lg⁡k+lg⁡2=lg⁡(2k)\lg k + 1 = \lg k + \lg 2 = \lg(2k)lgk+1=lgk+lg2=lg(2k):

T(2k)=2klg⁡(2k)T(2k) = 2k \lg(2k)T(2k)=2klg(2k)

Since 2k=n2k = n2k=n, we can write:

T(n)=nlg⁡nT(n) = n \lg nT(n)=nlgn

*3. Conclusion:*

By the principle of mathematical induction, since the base case holds for n=2n = 2n=2 and the inductive step shows that if the formula holds for n=kn = kn=k, it also holds for n=2kn = 2kn=2k, we conclude that:

T(n)=nlgnT(n) = n \lg nT(n)=nlgn

holds for all n≥2n \geq 2n≥2 that are exact powers of 2.

## Question:

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

---

## Answer:

*Pseudocode for Recursive Insertion Sort:*

```
plaintext
Copy code
function recursiveInsertionSort(A, n):
    if n <= 1:
        return  // Base case: array is already sorted
```

```
    // Recursively sort the first n - 1 elements
    recursiveInsertionSort(A, n - 1)

    // Insert the last element (A[n]) into the sorted subarray A[1 .. n - 1]
    insert(A, n)

function insert(A, n):
    key = A[n]
    j = n - 1

    // Move elements of A[1 .. n - 1], that are greater than key,
    // to one position ahead of their current position
    while j > 0 and A[j] > key:
        A[j + 1] = A[j]
        j = j - 1

    A[j + 1] = key
```

*Explanation of the Pseudocode:*

1. **Base Case:** The function `recursiveInsertionSort` checks if the size of the array nnn is less than or equal to 1. If so, it returns since the array is already sorted.
2. **Recursive Call:** The function calls itself with n−1n - 1n−1 to sort the first n−1n - 1n−1 elements of the array.
3. **Insertion Step:** After sorting the subarray, it calls the `insert` function to place the nthn^{th}nth element in its correct position within the sorted subarray.
4. **Insert Function:** The `insert` function uses a loop to shift elements greater than the key (the nthn^{th}nth element) to the right, creating a position for the key to be inserted.

*Recurrence for Worst-Case Running Time:*

In the worst case, the insertion of the last element into the sorted subarray will require shifting all n−1n - 1n−1 elements. The running time of the algorithm can be expressed with the following recurrence relation:

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

- T(n−1)T(n - 1)T(n−1) accounts for the time taken to sort the first n−1n - 1n−1 elements.
- O(n)O(n)O(n) accounts for the time taken to insert the nthn^{th}nth element into the sorted subarray, which could take up to n−1n - 1n−1 comparisons and shifts in the worst case.

*Solving the Recurrence:*

We can solve this recurrence using the method of iteration:

1. Unrolling the recurrence: T(n)=T(n−1)+O(n)T(n) = T(n - 1) + O(n)T(n)=T(n−1)+O(n) T(n−1)=T(n−2)+O(n−1)T(n - 1) = T(n - 2) + O(n - 1)T(n−1)=T(n−2)+O(n−1) Substituting: T(n)=(T(n−2)+O(n−1))+O(n)T(n) = (T(n - 2) + O(n - 1)) + O(n)T(n)=(T(n−2)+O(n−1))+O(n) =T(n−2)+O(n−1)+O(n)= T(n - 2) + O(n - 1) + O(n)=T(n−2)+O(n−1)+O(n)

Continuing to unroll, we can generalize this:

T(n)=T(n−k)+O(n)+O(n−1)+O(n−2)+…+O(n−k+1)T(n) = T(n - k) + O(n) + O(n - 1) + O(n - 2) + \ldots + O(n - k + 1)T(n)=T(n−k)+O(n)+O(n−1)+O(n−2)+…+O(n−k+1)

2. Setting k=n−1k = n - 1k=n−1: T(n)=T(1)+O(n)+O(n−1)+…+O(2)T(n) = T(1) + O(n) + O(n - 1) + \ldots + O(2)T(n)=T(1)+O(n)+O(n−1)+…+O(2)

The base case T(1)=O(1)T(1) = O(1)T(1)=O(1). Thus:

T(n)=O(1)+O(n+(n−1)+(n−2)+…+2)T(n) = O(1) + O(n + (n - 1) + (n - 2) + \ldots + 2)T(n)=O(1)+O(n+(n−1)+(n−2)+…+2)

The sum of the first nnn natural numbers is n(n−1)2\frac{n(n - 1)}{2}2n(n−1), leading to:

T(n)=O(1)+O(n2)=O(n2)T(n) = O(1) + O(n^2) = O(n^2)T(n)=O(1)+O(n2)=O(n2)

## Question:

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 vvv and eliminate half of the subarray from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is Θ(lg⁡n)\Theta(\lg n)Θ(lgn).

---

## Answer:

Binary search is a highly efficient searching algorithm that works on a sorted array. It operates by comparing the target value vvv with the midpoint element and, based on this comparison, eliminates half of the search range in each step.

### *Pseudocode for Binary Search*

Below is the recursive version of binary search pseudocode.

```
plaintext
Copy code
function binarySearch(A, v, low, high):
    if low > high:
        return -1  // v is not present in A

    mid = (low + high) / 2
```

```
    if A[mid] == v:
        return mid  // Found v at index mid
    else if A[mid] > v:
        return binarySearch(A, v, low, mid - 1)  // Search left half
    else:
        return binarySearch(A, v, mid + 1, high)  // Search right half
```

- **Input:** `A` is a sorted array, `v` is the value to search for, and `low` and `high` represent the indices of the current subarray being searched.
- **Output:** The index of `v` in `A` if found; otherwise, `-1` indicating `v` is not in the array.

*Iterative Version of Binary Search (Optional)*

An iterative version of binary search can be written as follows:

```
plaintext
Copy code
function binarySearch(A, v):
    low = 0
    high = length(A) - 1

    while low <= high:
        mid = (low + high) / 2

        if A[mid] == v:
            return mid  // Found v at index mid
        else if A[mid] > v:
            high = mid - 1  // Move to left half
        else:
            low = mid + 1  // Move to right half

    return -1  // v is not present in A
```

*Analysis of the Worst-Case Running Time of Binary Search*

In the worst case, binary search examines lg⁡nlg nlgn elements, where nnn is the number of elements in the array.

1. **Explanation of the Process**:
   - Each iteration of binary search reduces the search range by half. Given an initial array of nnn elements, binary search eliminates half of the elements at each step by checking the middle element.
   - This halving process continues until only one element remains or the target element vvv is found.
2. **Deriving the Running Time**:
   - At each step, binary search reduces the problem size from nnn to n2\frac{n}{2}2n, then n4\frac{n}{4}4n, and so forth.
   - This gives us a recurrence relation for the worst-case running time T(n)T(n)T(n) as: T(n)=T(n2)+O(1)T(n) = T\left(\frac{n}{2}\right) + O(1)T(n)=T(2n)+O(1)
   - Solving this recurrence shows that the number of divisions required to reach a single element is lg⁡nlg nlgn. Thus: T(n)=Θ(lg⁡n)T(n) = \Theta(\lg n)T(n)=Θ(lgn)

**Question:** The while loop of **insertion sort** in Section 2.1 uses a linear search to scan through the sorted subarray. If **insertion sort** used a binary search instead of a linear search, would that improve the worst-case running time to Θ(nlog⁡n)\Theta(n \log n)Θ(nlogn)?
**Answer:**
Using binary search within insertion sort would indeed reduce the number of comparisons needed to find the correct insertion point for each element, lowering it to Θ(log⁡n)\Theta(\log n)Θ(logn) per insertion instead of the linear Θ(n)\Theta(n)Θ(n) comparisons in the worst case.

However, while binary search can help identify where to place the element, it does not reduce the time required to shift elements to make space for the insertion. Even if the insertion point is determined in Θ(log⁡n)\Theta(\log n)Θ(logn) time, moving elements to insert the new element in its proper position still takes Θ(n)\Theta(n)Θ(n) time in the worst case, as it may require shifting nearly all elements in the subarray. For example, inserting a new smallest element would require shifting all elements one position to the right.

Thus, the overall time complexity remains dominated by the shifting operation, which, in the worst case, occurs nnn times with an average of O(n)O(n)O(n) shifts per insertion. As a result, the total time complexity for insertion sort remains Θ(n2)\Theta(n^2)Θ(n2), even if binary search is used to locate the insertion point.

In summary, using binary search within insertion sort does not improve the overall worst-case running time to Θ(nlog⁡n)\Theta(n \log n)Θ(nlogn); the worst-case time complexity remains Θ(n2)\Theta(n^2)Θ(n2) due to the shifting of elements.

# *CHAPTER NO 3*

# *CHARACTERIZING RUNNING TIMES*

## ***EXERCISE***

*3.1-1*

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

**Solution:**
In the lower-bound argument for insertion sort, the assumption that the input size is a multiple of 3 helps simplify calculations by dividing the array into three equal parts. When the input size nnn is not a multiple of 3, we can generalize the argument by dividing nnn into three segments of sizes approximately n/3n/3n/3, ensuring each part is as close to n/3n/3n/3 as possible. Then, follow the same logic: the largest values starting in one third must move through the middle third to reach the last third. Even if the groups aren't precisely equal, each element in the initial section must still pass through multiple positions, resulting in the same order of growth, Ω(n2)\Omega(n^2)Ω(n2), for the insertion sort's worst-case time.

*3.1-2*

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

**Solution:**
Selection sort has a worst-case running time of Θ(n2)\Theta(n^2)Θ(n2). In selection sort, for each iii-th element, the algorithm searches through the unsorted part of the array to find the smallest remaining element, which takes approximately n−in-in−i comparisons for the iii-th position. Summing this over all positions results in ∑i=1n−1(n−i)=n(n−1)/2\sum_{i=1}^{n-1} (n-i) = n(n-1)/2∑i=1n−1(n−i)=n(n−1)/2, which is Θ(n2)\Theta(n^2)Θ(n2). Thus, like insertion sort, selection sort also exhibits a quadratic time complexity for both its worst and average cases.

*3.1-3*

**Question:** Suppose that α\alphaα is a fraction in the range 0<α<10 < \alpha < 10<α<1. Show how to generalize the lower-bound argument for insertion sort to consider an input in which the αn\alpha nαn largest values start in the first αn\alpha nαn positions. What additional restriction do you need to put on α\alphaα? What value of α\alphaα maximizes the number of times that the αn\alpha nαn largest values must pass through each of the middle (1−2α)n(1 - 2\alpha) n(1−2α)n array positions?

**Solution:**
To generalize the argument, consider an array where the αn\alpha nαn largest elements are initially in the first αn\alpha nαn positions. Each element in this segment would need to move through the middle (1−2α)n(1 - 2\alpha)n(1−2α)n positions to reach its final sorted position in the last αn\alpha nαn positions. The worst case occurs when α=1/3\alpha = 1/3α=1/3, as this maximizes the product of the segments involved in movement, which results in α(1−2α)n2=(1/3)(1/3)n2=n2/9\alpha(1 - 2\alpha)n^2 = (1/3)(1/3)n^2 = n^2/9α(1−2α)n2=(1/3)(1/3)n2=n2/9, yielding a lower bound of Ω(n2)\Omega(n^2)Ω(n2).

### *3.2-5*

**Question:** Prove that the running time of an algorithm 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)).

**Solution:**
To establish that a running time T(n)=Θ(g(n))T(n) = \Theta(g(n))T(n)=Θ(g(n)), we need both upper and lower bounds that are asymptotically tight. If T(n)=O(g(n))T(n) = O(g(n))T(n)=O(g(n)), there exists a constant c1c_1c1 and n0n_0n0 such that T(n)≤c1g(n)T(n) \leq c_1 g(n)T(n)≤c1g(n) for n≥n0n \geq n_0n≥n0. Similarly, if T(n)=Ω(g(n))T(n) = \Omega(g(n))T(n)=Ω(g(n)), there exists a constant c2c_2c2 such that T(n)≥c2g(n)T(n) \geq c_2 g(n)T(n)≥c2g(n) for n≥n0n \geq n_0n≥n0. When both are true, T(n)T(n)T(n) is bounded above and below by g(n)g(n)g(n), establishing T(n)=Θ(g(n))T(n) = \Theta(g(n))T(n)=Θ(g(n)), meaning that the algorithm's running time grows as g(n)g(n)g(n) in both the best and worst cases.

### *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.

**Solution:**
By definition, f(n)=o(g(n))f(n) = o(g(n))f(n)=o(g(n)) implies that limn→∞f(n)/g(n)=0\lim_{n \to \infty} f(n)/g(n) = 0limn→∞f(n)/g(n)=0, meaning f(n)f(n)f(n) grows strictly slower than g(n)g(n)g(n). Conversely, f(n)=ω(g(n))f(n) = \omega(g(n))f(n)=ω(g(n)) implies limn→∞f(n)/g(n)=∞\lim_{n \to \infty} f(n)/g(n) = \inftylimn→∞f(n)/g(n)=∞, meaning f(n)f(n)f(n) grows strictly faster than g(n)g(n)g(n). It is impossible for f(n)f(n)f(n) to simultaneously grow both faster and slower than g(n)g(n)g(n) as nnn approaches infinity, thus the intersection o(g(n))∩ω(g(n))o(g(n)) \cap \omega(g(n))o(g(n))∩ω(g(n)) is indeed the empty set.

### *3.2-7*

**Question:** Give corresponding 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)) with two parameters nnn and mmm that can go to infinity independently at different rates.

**Solution:**
For a function f(n,m)f(n,m)f(n,m) with two parameters:

- f(n,m)=O(g(n,m))f(n,m) = O(g(n,m))f(n,m)=O(g(n,m)) means there exist constants ccc, n0n_0n0, and m0m_0m0 such that f(n,m)≤c·g(n,m)f(n,m) \leq c \cdot g(n,m)f(n,m)≤c·g(n,m) for all n≥n0n \geq n_0n≥n0 and m≥m0m \geq m_0m≥m0.
- f(n,m)=Ω(g(n,m))f(n,m) = \Omega(g(n,m))f(n,m)=Ω(g(n,m)) implies there exist positive 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 and m≥m0m \geq m_0m≥m0.
- f(n,m)=Θ(g(n,m))f(n,m) = \Theta(g(n,m))f(n,m)=Θ(g(n,m)) holds if 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)), implying f(n,m)f(n,m)f(n,m) is asymptotically bounded both above and below by g(n,m)g(n,m)g(n,m).

*3.3-1*

**Question:** 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 also nonnegative, show that f(n)·g(n)f(n) \cdot g(n)f(n)·g(n) is monotonically increasing.

**Solution:**
If f(n)f(n)f(n) and g(n)g(n)g(n) are monotonically increasing, then for any n1<n2n_1 < n_2n1<n2, we have f(n1)≤f(n2)f(n_1) \leq f(n_2)f(n1)≤f(n2) and g(n1)≤g(n2)g(n_1) \leq g(n_2)g(n1)≤g(n2). Then 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), so f(n)+g(n)f(n) + g(n)f(n)+g(n) is also monotonically increasing. For the composition f(g(n))f(g(n))f(g(n)), since g(n)g(n)g(n) is monotonically increasing and fff is also monotonically increasing, f(g(n1))≤f(g(n2))f(g(n_1)) \leq f(g(n_2))f(g(n1))≤f(g(n2)), so f(g(n))f(g(n))f(g(n)) is monotonically increasing. For the product f(n)·g(n)f(n) \cdot g(n)f(n)·g(n), if f(n),g(n)≥0f(n), g(n) \geq 0f(n),g(n)≥0, then 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), making f(n)·g(n)f(n) \cdot g(n)f(n)·g(n) monotonically increasing.

*3.3-2*

**Question:** Prove that ⌊αn⌋+⌈(1−α)n⌉=n\lfloor \alpha n \rfloor + \lceil (1 - \alpha) n \rceil = n⌊αn⌋+⌈(1−α)n⌉=n for any integer nnn and real number α\alphaα in the range 0≤α≤10 \leq \alpha \leq 10≤α≤1.

**Solution:**
Since ⌊αn⌋\lfloor \alpha n \rfloor⌊αn⌋ is the largest integer less than or equal to αn\alpha nαn and ⌈(1−α)n⌉\lceil (1 - \alpha) n \rceil⌈(1−α)n⌉ is the smallest integer greater than or equal to (1−α)n(1 - \alpha) n(1−α)n, their sum will equal nnn due to the properties of floors and ceilings. Essentially, the fractional parts of αn\alpha nαn and (1−α)n(1 - \alpha) n(1−α)n sum to 1, giving an integer result of nnn when combined with their respective floors and ceilings.

### 3.3-3

**Question:** Use equation (3.14) or other means to show that (n+o(n))k=Θ(nk)(n + o(n))^k = \Theta(n^k)(n+o(n))k=Θ(nk) for any real constant kkk. Conclude that ⌈n⌉k=Θ(nk)\lceil n \rceil^k = \Theta(n^k)⌈n⌉k=Θ(nk) and ⌊n⌋k=Θ(nk)\lfloor n \rfloor^k = \Theta(n^k)⌊n⌋k=Θ(nk).

**Solution:**
As o(n)o(n)o(n) grows slower than nnn, adding o(n)o(n)o(n) to nnn does not affect the asymptotic behavior. Expanding (n+o(n))k(n + o(n))^k(n+o(n))k via the binomial theorem, the dominant term remains nkn^knk, as all other terms grow slower than nkn^knk, resulting in Θ(nk)\Theta(n^k)Θ(nk). Similarly, both ⌈n⌉k\lceil n \rceil^k⌈n⌉k and ⌊n⌋k\lfloor n \rfloor^k⌊n⌋k are asymptotically equivalent to nkn^knk.

### 3.3-4

**Question:** Prove the following:

- (a) Equation (3.21).
- (b) Equations (3.26)–(3.28).
- (c) lg⁡(Θ(n))=Θ(lg⁡n)\lg(\Theta(n)) = \Theta(\lg n)lg(Θ(n))=Θ(lgn).

**Solution:**
Each of these statements formalizes properties of logarithmic and polynomial growth rates. For example, (a) can be derived by analyzing properties of sums, and (c) uses the property that asymptotic bounds are preserved under logarithmic transformations. Each property holds due to the regularity of logarithmic and polynomial growth rates under standard asymptotic manipulations.

### 3.3-5

**Question:** Is the function ⌈lg⁡n⌉\lceil \lg n \rceil⌈lgn⌉ polynomially bounded? Is the function ⌈lg⁡lg⁡n⌉\lceil \lg \lg n \rceil⌈lglgn⌉ polynomially bounded?

**Solution:**
The function ⌈lg⁡n⌉\lceil \lg n \rceil⌈lgn⌉ is not polynomially bounded because logarithmic growth is slower than any polynomial function of nnn. Similarly, ⌈lg⁡lg⁡n⌉\lceil \lg \lg n \rceil⌈lglgn⌉ is also not polynomially bounded since lg⁡lg⁡n\lg \lg nlglgn grows even slower than lg⁡n\lg nlgn.

### 3.3-6

**Question:** Which is asymptotically larger: lg⁡(lg⁡∗n)\lg(\lg^* n)lg(lg∗n) or lg⁡∗(lg⁡n)\lg^*(\lg n)lg∗(lgn)?

**Solution:**
lg⁡(lg⁡∗n)\lg(\lg^* n)lg(lg∗n) grows asymptotically larger than lg⁡∗(lg⁡n)\lg^* (\lg

n)lg∗(lgn). The iterated logarithm lg∗n\lg^* nlg∗n grows extremely slowly, so taking lg∗\lglg of it still grows slowly but remains larger than lg∗(lg∗n)\lg^* (\lg n)lg∗(lgn), which remains bounded due to the properties of lg∗\lg^*lg∗ applied to any logarithmic function.

### 3.3-7

**Question:** Show that the golden ratio ϕ\phiϕ and its conjugate ϕ^\hat{\phi}ϕ^ both satisfy the equation x2=x+1x^2 = x + 1x2=x+1.

**Solution:**
To solve for ϕ\phiϕ and ϕ^\hat{\phi}ϕ^, substitute x=1±52x = \frac{1 \pm \sqrt{5}}{2}x=21±5 into x2=x+1x^2 = x + 1x2=x+1. Expanding both sides confirms that 1±52\frac{1 \pm \sqrt{5}}{2}21±5 satisfies the equation, yielding the golden ratio ϕ≈1.618\phi \approx 1.618ϕ≈1.618 and its conjugate ϕ^≈−0.618\hat{\phi} \approx -0.618ϕ^≈−0.618.

### 3.3-8

**Question:** Prove by induction that the iii-th Fibonacci number satisfies the equation Fi=ϕi−ϕ^i5F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}Fi=5ϕi−ϕ^i, where ϕ\phiϕ is the golden ratio and ϕ^\hat{\phi}ϕ^ is its conjugate.

**Solution:**
Base case: For i=0i = 0i=0 and i=1i = 1i=1, this identity holds as F0=0F_0 = 0F0=0 and F1=1F_1 = 1F1=1. Inductive step: Assuming it holds for i=ki = ki=k and i=k−1i = k-1i=k−1, we use the recurrence Fk+1=Fk+Fk−1F_{k+1} = F_k + F_{k-1}Fk+1=Fk+Fk−1 and properties of ϕ\phiϕ and ϕ^\hat{\phi}ϕ^ to complete the induction, verifying the identity.

### 3.3-9

**Question:** Show that klg∗k=Θ(n)k \lg k = \Theta(n)klgk=Θ(n) implies k=Θ(nlg∗n)k = \Theta\left(\frac{n}{\lg n}\right)k=Θ(lgnn).

**Solution:**
Assume klg∗k=Θ(n)k \lg k = \Theta(n)klgk=Θ(n). Solving for kkk in terms of nnn, we approximate kkk as Θ(nlg∗n)\Theta\left(\frac{n}{\lg n}\right)Θ(lgnn) by rearranging terms and using iterative approximation methods. This relationship shows that kkk scales as nlg∗n\frac{n}{\lg n}lgnn, leading to k=Θ(nlg∗n)k = \Theta\left(\frac{n}{\lg n}\right)k=Θ(lgnn).