Abbottabad University Of Science And Technology

Department Of Computer Science

Assignment # 01

Name
Zamir Ali

Roll no #
14876

Section
3rd D

Subject
Data Structure And Algorithms

Submitted to
Jamal Abdul ahad

The role of Algorithms in computing

Exercises

Question 1

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

Scenario:

Online Store Sorting Products by Price Imagine you are browsing an online store that sells electronics. The store allows you to sort products by price, either from lowest to highest or from highest to lowest. This is a classic use case for sorting, where you want to present the products in an organized way to help customers make better purchasing decisions.

In this example:

Sorting criteria:

Price (either ascending or descending)

Why sorting is important:

It helps users quickly find products within their budget or compare expensive and cheap products.This can be implemented using sorting algorithms like Merge Sort or Quick Sort for efficient sorting, especially when the number of products is large.

Code:

```
def merge(arr, left_half, right_half):
    i = j = k = 0
    while i < len(left_half) and j < len(right_half):
        if left_half[i]['price'] < right_half[j]['price']:
            arr[k] = left_half[i]
            i += 1
        else:
            arr[k] = right_half[j]
            j += 1
        k += 1
```

```
    while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1

    while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1
def merge_sort(products):
    if len(products) > 1:
        mid = len(products) // 2
        left_half = products[:mid]
        right_half = products[mid:]
        merge_sort(left_half)
        merge_sort(right_half)
        merge(products, left_half, right_half)
if__name__== "_main_"
    products = [
        {"name": "Laptop", "price": 999.99},
        {"name": "Smartphone", "price": 499.99},
        {"name": "Headphones", "price": 199.99},
        {"name": "Tablet", "price": 299.99}
    ]
    print("Original products:")
    for product in products:
        print(f"{product['name']} - ${product['price']}")
```

```
merge_sort(products)
print("\nSorted products by price (low to high):")
for product in products:
    print(f"{product['name']} - ${product['price']}")
```

```
def merge(arr, left_half, right_half):
    i = j = k = 0
    while i < len(left_half) and j < len(right_half):
        if left_half[i]['price'] < right_half[j]['price']:
            arr[k] = left_half[i]
            i += 1
        else:
            arr[k] = right_half[j]
            j += 1
        k += 1

    while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1

    while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1
def merge_sort(products):
    if len(products) > 1:
        mid = len(products) // 2
        left_half = products[:mid]
        right_half = products[mid:]

        merge_sort(left_half)
        merge_sort(right_half)
        merge(products, left_half, right_half)
```

```
if __name__ == "__main__":
    products = [
        {"name": "Laptop", "price": 999.99},
        {"name": "Smartphone", "price": 499.99},
        {"name": "Headphones", "price": 199.99},
        {"name": "Tablet", "price": 299.99}
    ]
    print("Original products:")
    for product in products:
        print(f"{product['name']} - ${product['price']}")
    merge_sort(products)
    print("\nSorted products by price (low to high):")
    for product in products:
        print(f"{product['name']} - ${product['price']}")
```

Output:

```
Original products:
Laptop - $999.99
Smartphone - $499.99
Headphones - $199.99
Tablet - $299.99

Sorted products by price (low to high):
Headphones - $199.99
Tablet - $299.99
Smartphone - $499.99
Laptop - $999.99
```

Question 2

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

In a real-world setting, efficiency is not just about speed. Other important measures of efficiency to consider include:

- **Memory Usage**

Why it matters:

Some algorithms are fast but use a lot of memory. In systems with limited resources, such as mobile devices or embedded systems, memory efficiency is critical.

Example:

When processing large datasets, algorithms like Merge Sort require additional space for the sorted sub-arrays, whereas algorithms like Quick Sort can be more memory-efficient due to in-place sorting.

- **Energy Consumption**

Why it matters:

In mobile devices, IoT, or battery-operated systems, minimizing energy consumption is vital to prolong battery life.

Example:

Algorithms that reduce CPU usage or avoid frequent disk accesses can help save power in portable devices.

- **Scalability**

Why it matters:

The ability of an algorithm or system to handle increasing amounts of work efficiently as the problem size grows.

Example:

In a web server, an algorithm that works well for 100 users but crashes or slows down with 1,000 users is not scalable. Efficient algorithms must scale to handle larger inputs or user loads without significant degradation in performance.

- **I/O Efficiency**

Why it matters:

In applications dealing with large amounts of data stored on disk, minimizing input/output (I/O) operations can significantly improve performance.

Example:

Database query optimization focuses on reducing the number of disk reads/writes, as these operations are much slower than memory access.

- **Latency**

Why it matters:

In real-time systems (like stock trading platforms or video games), it's not just the overall speed but how quickly the system can respond to an individual request (i.e., low-latency responses).

Example:

A self-driving car must react quickly to obstacles in real-time, where even a slight delay could lead to accidents.

- **Concurrency and Parallelism**

Why it matters:

Efficient handling of multiple tasks simultaneously is crucial in multi-core processors, cloud computing, and distributed systems.

Example:

An algorithm that can efficiently divide work among multiple processors can complete tasks faster, especially when handling large datasets in parallel computing.

- **Accuracy or Precision**

Why it matters:

Some algorithms may sacrifice accuracy for speed, but in applications like scientific computing, machine learning, or financial modeling, accuracy is crucial.

Example:

In data compression, the balance between compression speed and the accuracy of the reconstructed data is critical (lossless vs. lossy compression).

- **Ease of Implementation and Maintenance**

Why it matters:

An algorithm might be highly efficient but extremely complex to implement or maintain. In real-world projects, simpler algorithms are often preferred if the gains in performance from a more complex algorithm are not significant.

Example:

A less efficient but simpler sorting algorithm like Insertion Sort may be preferred in small-scale projects over more complex algorithms like Quick Sort or Heap Sort.

- **Security**

Why it matters:

Efficiency must not come at the expense of security. Some algorithms might be fast but vulnerable to attacks.

Example:

Cryptographic algorithms must be both secure and efficient, ensuring that even with strong encryption, the time to encrypt/decrypt data is reasonable.

- **Cost**

Why it matters:

In a business setting, the cost of implementing and running an algorithm (in terms of resources like cloud storage or processing power) needs to be weighed against its performance.

Example:

Using a more resource-intensive algorithm on a cloud service could increase operational costs. For example, a company might choose a less efficient algorithm if it saves on cloud infrastructure expenses.

Question 3

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

Let’s consider the Hash Table as a data structure, which is widely used in many real-world applications like databases, caching, and indexing.

Strengths of a Hash Table:

- **Fast Lookups (O(1) Average Time Complexity):**

Hash tables provide constant-time complexity for searching, inserting, and deleting elements on average. This is because the data is stored in a way that allows for direct indexing using a hash function.

Example:

In a dictionary-like structure (such as Python's dict), you can quickly retrieve a value associated with a key without needing to search through all the entries.

- **Efficient Insertions and Deletions:**

Insertions and deletions are also efficient in hash tables. In contrast to data structures like arrays or linked lists, where insertions and deletions can take linear time in the worst case, hash tables are designed to minimize such inefficiencies.

- **Flexible Key-Value Pair Storage:**

Hash tables allow you to store data in key-value pairs, making them perfect for cases where fast access to data based on a specific key is needed.

Example:

In caching systems, a hash table is used to store recently accessed data, and each data item can be fetched quickly using its unique key.

- **Widely Used in Real-World Applications:**

Hash tables are found in applications like caches (e.g., LRU Cache), databases (indexing), symbol tables in compilers, and routing tables in networking.

Limitations of a Hash Table:

- **Collisions Can Degrade Performance:**

Although the average lookup time is O(1), collisions (where two keys hash to the same index) can cause multiple elements to be stored in the same location (bucket), requiring additional work to search through them. In the worst case, a hash table can degrade to O(n) lookup time.

- **Collision Handling Techniques:**

Methods like chaining (storing multiple elements at the same index) or open addressing (finding an alternative empty slot) are used to handle collisions, but these add complexity.

- **Requires a Good Hash Function:**

The efficiency of a hash table heavily depends on the quality of the hash function. A poorly designed hash function can cause too many collisions, leading to inefficient data access.

Example:

If the hash function distributes keys unevenly, it will result in certain buckets being overloaded while others are empty, reducing the efficiency of the table.

- **Memory Overhead:**

Hash tables typically require more memory than other data structures like arrays or linked lists because they need to allocate space for the hash table itself, often with extra slots (to reduce collisions), which can lead to memory overhead.

Example:

Hash tables tend to allocate more space than required initially to reduce the chance of collisions, which may lead to wasted space if the data is sparse.

- **No Ordering of Elements:**

Hash tables do not maintain any order of elements. If you need a data structure that can return elements in a sorted or specific order, a hash table is not suitable.

Example:

If you need to iterate over elements in sorted order (like in a priority queue or for range queries), hash tables are not the right choice.

- **Fixed Size in Some Implementations:**

In some hash table implementations, you need to define the size of the hash table in advance. If the table becomes too full, performance can degrade, and resizing the table is costly.

Example:

Expanding the size of a hash table often involves rehashing all elements, which is an expensive operation.

- **Inefficient for Small Datasets:**

Hash tables are overkill for small datasets where simpler data structures (like arrays or linked lists) can perform equally well or better without the overhead of hashing and managing collisions.

- **When to Use a Hash Table:**

Use a hash table when you need fast access to data based on unique keys and don't care about ordering.

Avoid a hash table when you need sorted data, or when you're working with small datasets where simpler structures like arrays or trees can be more efficient.

Question 4

These two problems are classic in graph theory and optimization, being the Shortest-Path Problem due its simplicity (as mentioned above) as well for it is swift to solve by optimal algorithms that run on polynomial time complexity), but each one stands out from another by a set of characteristics and possible applications.

Similarities:

These types of problems can be represented in terms of graphs, where each node represents a location such as city or point and edges represent lines or roads between them.

Optimization Objectives:

Both seek an optimum distance solution.

Finding the Shortest-Path to Two Nodes

The objective in TSP is to determine the shortest possible path that visits each node exactly once and returns to the point of origin.

Same or Similar Algorithm:

Solution Algorithms of these two problems are graph traversal algorithms. For instance, Dijkstra's algorithm is a classical solution to the shortest path problem, while numerous heuristics and optimizations approaches — e.g., branch-and-bound method, dynamic programming or genetic algorithms are commonly adopted for TSP.

Differences:

Problem Definition:

Single-Pair Shortest-Route Issue: searches for the best path among one pair of nodes. Like, for example, finding the most efficient path from point A to point B.

The Traveling Salesman Problem — you need to visit a set of locations (nodes) exactly once before returning to the starting point, which requires forming a full circuit. This is a more difficult problem as it has multiple destinations.

Complexity:

Shortest-Path Problem: Polynomial-time solveable in general. The problem is solvable by algorithm similar to Dijkstra's and Bellman-Ford, What I really meant was for larger graph.

Traveling Salesman Problem: NP-hard, which means that no polynomial-time algorithm can efficiently solve all instances of TSP. Exact solutions can are only scalable up to very small datasets, and larger instances often require heuristic algorithms.

Objective:

Shortest-Path Problem: The aim is to reduce the distance (or cost) between two nodes in particular.

Traveling Salesman Problem: It aims to minimize the total distance (cost) of visiting a number of nodes while also requiring that you return to your starting node, which adds another level of complexity.

Conclusion:

In brief whereas both are essentially graphs and optimization over distances, Shortest-path seeks to locate a route from point A to B; the TSP wants us visit multiple points but in an efficient complete circuit way. That difference in focus and the resulting complexity is one of the most important distinctions between these two problems.

Question 5

Suggest a real-world problem in which only the best solution will do. Then come up with one in which

Real-world problem to be solved with the Best Solution

Issue:

Air Traffic Management Routing aircraft safely and efficiently is essential to air traffic control. And, even a tiny mistake in the numbers or judgement can cause major things to happen like crashes and injuries.

Why We Need The Best Solution

Safety must come first, and this requires absolute precise routing for takeoffs and landings.

We adjust the schedules of every flight by considering weather conditions, how our aircraft are performing and whether or not we have open gates at each airport so that all flights can land safely.

A Real-World "Approximately the Best Solution is Good Enough" Problem

Problem:

E-commerce Logistic An e-commerce company owns a fleet of delivery vehicles that they use to deliver their products and it wants to optimize the routes for these deliveries such as (optimal path), so customers can receive quickly while minimizing transportation costs.

Why the Best Solution to Anything is About Good Enough:

- While obviously the most efficient routes can save on time and fuel, going a little bit out of one's way may not dramatically decrease overall efficiency.
- If a courier has quite many deliveries at once, an approximate answer might be quick to compute whilst still satisfying the customers.
- Heuristic methods (as the Nearest Neighbor or Genetic Algorithms) are able to get good routes with out such a computation effort

Question 6

Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.

Real-World Problem: Ride-Hailing Services (like Uber)

1. When All Input is Available:

- **Pre-Scheduled Rides:**

  Example:

  - A person books a ride to the airport for 6 AM tomorrow.
  - Here, the ride details (pickup time and location) are known in advance. The app can plan ahead and assign drivers accordingly.

2. When Input Arrives Over Time:

- **On-Demand Rides:**

  Example:

  - A person requests a ride right now (e.g., at 5 PM).

- In this case, the app gets ride requests one at a time as people ask for rides. The app must quickly find available drivers for each new request while managing other rides already in progress.

Summary:

- **Pre-Scheduled:**

  All ride details are known ahead of time.

- **On-Demand:**

  Ride requests come in real-time, and the system must react immediately.

Question 7

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

Example Application: Online Recommendation Systems

Overview:

Online recommendation systems are widely used in platforms like Netflix, Amazon, and Spotify to suggest products, movies, or music to users based on their preferences and behavior. These systems require algorithmic content at the application level to analyze data and generate recommendations.

Algorithmic Components:

1. **Collaborative Filtering**:

   - This technique analyzes user behavior and preferences to recommend items. For example, if User A and User B have similar tastes, the system can recommend items that User B liked to User A.

   ### Algorithmic Content:

   Algorithms like Matrix Factorization (e.g., Singular Value Decomposition) are used to decompose user-item interaction matrices to identify patterns and similarities.

2. **Content-Based Filtering:**

This approach recommends items based on the attributes of items that a user has liked in the past. For example, if a user liked action movies, the system will recommend other action movies.

Algorithmic Content:

Algorithms that utilize techniques such as TF-IDF (Term Frequency-Inverse Document Frequency) for text analysis or cosine similarity to measure item similarity based on features.

3. **Hybrid Methods:**

Many systems combine both collaborative and content-based filtering to improve accuracy and address the limitations of each approach.

Algorithmic Content:

Algorithms that integrate results from both methods and may use machine learning techniques to optimize recommendations based on user feedback.

Importance:

User Experience:

By providing personalized recommendations, these systems enhance user satisfaction and engagement.

Business Impact:

Effective recommendations can lead to increased sales, higher user retention, and improved customer loyalty.

Example in Practice:

Netflix:

Uses complex algorithms to analyze user viewing history, ratings, and behaviors to suggest new shows and movies. For instance, if you often watch thrillers, Netflix will recommend other thrillers that similar users have enjoyed.

Conclusion:

Online recommendation systems exemplify applications that heavily rely on algorithmic content to process large datasets, derive insights, and deliver personalized experiences. The effectiveness

of these systems hinges on sophisticated algorithms that continuously learn and adapt to user preferences. If you have more questions or need further examples, feel free to ask.

Question 8

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

To determine for which values of n insertion sort beats merge sort, we need to compare their running times:

1. **Insertion Sort**:$8n^2$
2. **Merge Sort**: 64nlog_2 n

We want to find the values of n for which:

$$8n^2<64n log_2$$

Step 1: Simplify the Inequality

Divide both sides by 8n (assuming n>0):

$$n<8log_2n$$

Step 2: Analyze the Inequality

To solve the inequality $n<8log_2n$

$$\frac{n}{log_2n}<n$$

Step 3: Find Values of n

This inequality indicates that we need to check values of n to see where $\frac{n}{log_2n}$ is less than 8.

1. **Try small values of n**:

   For n=1:

   $\frac{1}{log_2 1}$ is undefined(since $log_2\ 1 = 0$

   For n=2:

$$\frac{2}{log_2 2} = \frac{2}{1} = 2 < 8$$

For n=4:

$$\frac{4}{log_2 4} = \frac{4}{2} = 2 < 8$$

For n=8:

$$\frac{8}{log_2 8} = \frac{8}{3} \approx 2.67 < 8$$

For n=16:

$$\frac{16}{log_2 16} = \frac{16}{4} \approx 4 < 8$$

For n=32:

$$\frac{32}{log_2 32} = \frac{32}{5} \approx 6.4 < 8$$

For n=64:

$$\frac{64}{log_2 64} = \frac{64}{6} \approx 10.67 < 8$$

2. **Continue Checking Larger Values**:

For n=50:

$$log_2 50 \approx 5.64 \Rightarrow \frac{50}{5.64} \approx 8.86 < 8$$

For n=40:

$$log_2 40 \approx 5.32 \Rightarrow \frac{40}{5.32} \approx 7.51 < 8$$

Question 9

What is the smallest value of n such that an algorithm whose running time is 100n2 runs faster than an algorithm whose running time is 2 n on the same machine?

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

Step 1: Evaluate Both Sides for Small Values of n

Let's check values of n to see when the inequality holds.

For n=1:

$$100\,(1^2) = 100 \;\; and \;\; 2^1 = 2 \;\Rightarrow\; 100 < 2 \;\; (false)$$

For n=2:

$$100\,(2^2) = 400 \;\; and \;\; 2^2 = 4 \Rightarrow\; 400 < 4 \;\; (false)$$

For n=3:

$$100\,(3^2) = 900 \;\; and \;\; 2^3 = 8 \Rightarrow\; 900 < 8 \;\; (false)$$

For n=4:

$$100\,(4^2) = 1600 \;\; and \;\; 2^4 = 16 \;\Rightarrow\; 1600 < 16\;(false)$$

For n=5:

$$100\,(5^2) = 2500 \;\; and \;\; 2^5 = 32 \;\Rightarrow\; 2500 < 32\;(false)$$

For n=6:

$$100\,(6^2) = 3600 \;\; and \;\; 2^6 = 64 \;\Rightarrow\; 3600 < 64 \;\; (false)$$

For n=7:

$$100\,(7^2) = 4900 \;\; and \;\; 2^7 = 128 \;\Rightarrow\; 4900 < 128\;(false)$$

For n=8:

$$100\,(8^2) = 6400 \;\; and \;\; 2^8 = 256 \;\Rightarrow\; 6400 < 256\;(false)$$

For n=9:

$$100\,(9^2) = 8100 \quad and \quad 2^9 = 512 \;\Rightarrow\; 8100 < 512\;(false)$$

For n=10:

$$100\,(10^2) = 10000 \quad and \quad 2^{10} = 128 \;\Rightarrow\; 1000 < 1024\;(false)$$

For n=11:

$$100\,(11^2) = 12100 \quad and \quad 2^{11} = 2048 \;\Rightarrow\; 12100 < 1024\;(false)$$

For n=12:

$$100\,(12^2) = 14400 \quad and \quad 2^{12} = 4096 \Rightarrow\; 14400 < 4096\;(false)$$

For n=13:

$$100\,(13^2) = 16900 \quad and \quad 2^{13} = 9192 \;\Rightarrow\; 16900 < 9192\;(false)$$

For n=14:

$$100\,(14^2) = 19600 \quad and \quad 2^{14} = 16384 \;\Rightarrow\; 19600 < 16384\;(false)$$

For n=15:

$$100\,(15^2) = 22500 \quad and \quad 2^{15} = 32768 \;\Rightarrow\; 22500 < 32768\;(false)$$

The smallest value of n such that $100n^2 < 2^n$ is 15.

Question 10

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.

time	Log n	√n	n	n log n	n^2	n^3	2^n	n!
1 second	999,999	1.00e+12	1,000,000	62,746	1,000	99	19	9
1 minute	60,000,000	3.60e+15	60,000,000	2,801,417	7,745	391	25	11
1 hour	3.60e+9	1.30e+19	3.60e+19	133,378,058	60,000	1,532	31	12
1 day	8.64e+10	7.46e+21	8.64e+10	2,755,147,513	293,938	4,420	36	13
1 month	2.59e+12	6.72e+24	2.59e+12	71,870,856,404	1,609,968	13,736	41	15
1 year	3.15e+13	9.95e+26	3.15e+13	797,633,893,349	5,615,692	31,593	44	16
1 century	3.15e+15	9.95e+30	3.15e+15	68,610,956,750,570	56,156,922	146,654	51	17

Chapter 2

Analyzing Algorithms

Exercises

illustrate the operation of I NSERTION-SORT on an array initially containing the sequence {31, 41,59, 26,41, 58}.

```
def insertion_sort(arr):


    for i in range(1, len(arr)):
        key = arr[i]
        j = i -
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
```

```
    arr[j + 1] = key

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

insertion_sort(array)

print("Sorted array:", array)
```

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key


array = [31, 41, 59, 26, 41, 58]
insertion_sort(array)
print("Sorted array:", array)
```

Question 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].

SUM-ARRAY(A, n):

```
    sum = 0

    for i from 1 to n do

        sum = sum + A[i]

    return sum
```

Loop Invariant:

At the start of each iteration of the loop, the variable `sum` contains the sum of the elements in the subarray A[1…i−1].

Initialization:

- Before the loop begins, sum is initialized to 0 (line 1). At this point, for i=1, the subarray A[1…0] (an empty subarray) has a sum of 0. Thus, the invariant holds true before the first iteration.

Maintenance:

- Assume that the invariant holds at the start of iteration iii (i.e., sum contains the sum of the elements in A[1…i−1]).
- During the i^{th}iteration (line 3), we execute sum = sum + A[i].
- After this operation, sum now contains the sum of the elements in A[1…i], because it adds the i^{th}element to the sum of the previous elements.
- Thus, the invariant continues to hold for the next iteration.

Termination:

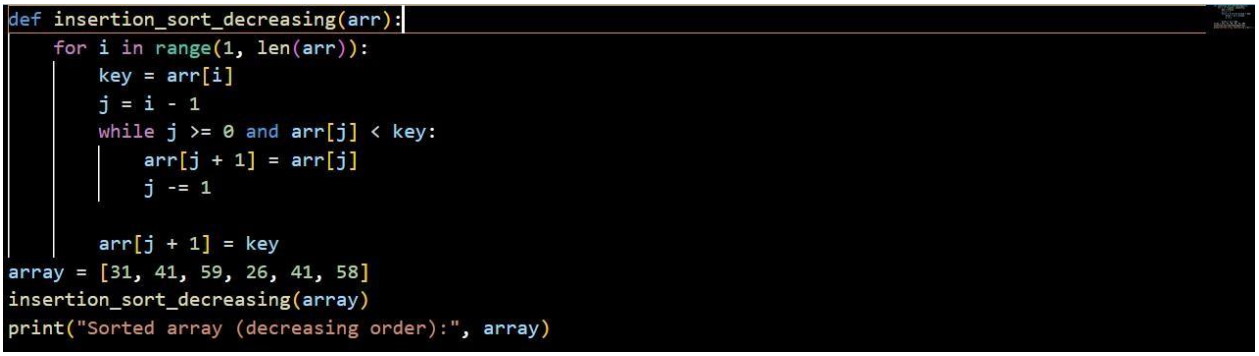
- The loop terminates when iii exceeds n. At this point, the loop has iterated for i=1 to n.
- By the loop invariant, when the loop exits, sum contains the sum of the elements in A[1…n].
- Therefore, upon termination, sum correctly represents the total sum of the array A[1…n].

Question 3

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

```
def insertion_sort_decreasing(arr):

    for i in range(1, len(arr)):

    key = arr[i]

    j = i - 1

        while j >= 0 and arr[j] < key:

       arr[j + 1] = arr[j]

       j -= 1

    arr[j + 1] = key

array = [31, 41, 59, 26, 41, 58]
```

```
insertion_sort_decreasing(array)

print("Sorted array (decreasing order):", array)
```



```
def insertion_sort_decreasing(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] < key:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key
array = [31, 41, 59, 26, 41, 58]
insertion_sort_decreasing(array)
print("Sorted array (decreasing order):", array)
```

Question 4

Consider the searching problem:

Input:

A sequence of n numbers $\{a_1, a_2, \ldots\ldots a_n\}$ stored in array A[1 : n]and a value x.

Output:

An index i such that x equals A[i] or the special value NIL if x does not appear in A.

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

LINEAR-SEARCH(A, n, x):

```
for i = 1 to n do
    if A[i] == x then
        return i
return NIL
```

Loop Invariant

At the start of each iteration of the loop, if **x** is in the array **A[1…n],** then it is located in **A[1…i−1]** or the search has not yet checked **A[i].**

Properties of the Loop Invariant

Initialization:

Before the first iteration (when i=1), the invariant holds because the algorithm has not yet checked any elements, so it correctly states that if x is in the array, it hasn't been found in A[1…0] (which is empty). Thus, the invariant holds at initialization.

Maintenance:

- Assume the invariant holds at the beginning of the i^{th}iteration (i.e., if x is in the array, it is in A[1…i−1] or has not yet been checked).
- In this iteration, we check if A[i]==x

If true, we return iii, and the search is successful.

If false, the invariant still holds because we have now confirmed that xxx is not in A[i] and we proceed to check the next element A[i+1].

- Thus, after this iteration, if xxx is in the array, it must be in A[1…i] or the search continues to A[i+1]

Termination:

- The loop terminates when iii exceeds n. At this point, we have checked all elements in the array.
- If the loop exits without returning an index, it means x was not found in any of the elements A[1…n], so we return NIL.
- By the invariant, we can conclude that if x exists in the array, it will have been found; otherwise, we correctly return NIL.

Question 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 = $\sum_{i=0}^{n-1} A[i].\ 2^i$, and b $\sum_{i=0}^{n-1} B[i].\ 2^i$. 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 = $\sum_{i=0}^{n-1} C[i].\ 2^i$ 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.

```
ADD-BINARY-INTEGERS(A, B, n):
    #Initialize the result array C with size n + 1 (for possible carry)
    C = array of size (n + 1
    carry = 0 #Initialize carry to 0
    #Iterate through each bit from least significant to most significant
    for i from 0 to n - 1 do
        #Calculate the sum of the current bits and the carry
        sum = A[i] + B[i] + carry
        #C[i] will be the least significant bit of the sum
        C[i] = sum mod 2
        carry = sum
    C[n] = carry
    return C
```

Explanation:

Initialization:

- Create an array C of size n+1to store the result.
- Initialize a variable carry to 0, which will hold the carry from the addition of bits.

Loop through bits:

Loop from 0 to n−1 to process each bit of arrays A and B:

- Compute the sum of the corresponding bits A[i] and B[i], along with any carry from the previous addition.
- Store the least significant bit of the result in C[i].
- Update the carry for the next iteration using integer division by 2.

Final carry:

- After the loop, check if there is a carry left and store it in C[n].

Return result:

- Finally, return the array C, which now contains the binary sum of A and B.

Example Usage:

If you want to add the binary numbers A=[1,0,1] (which represents the number 5 in decimal) and B=[1,1,0] (which represents the number 6 in decimal), you can call the function as follows:

A = [1, 0, 1] // 5 in binary

B = [1, 1, 0] // 6 in binary

n = 3

C = ADD-BINARY-INTEGERS(A, B, n)

// C will contain the result: [0, 0, 1, 1] which represents 11 in binary

Question 6

Express the function $n^3/1000$ C+$100n^2$ - 100n + 3 in terms of , Θ-notation.

To express the function f(n)=$\frac{n^3}{1000}$+$100n^2$−100n+3in term of Θ-notation ,we need to identify the dominant term as n grow larger:

Analyzing the Function:

1. **Identify the Growth of Each Term**:
   - The term $\frac{n^3}{1000}$ grows as O(n^3).
   - The term $100n^2$ grows as O(n^2).
   - The term −100ngrows as O(n).
   - The constant 3 is O(1).
2. **Dominant Term**:

   As n becomes very large, $\frac{n^3}{1000}$will dominate the growth of the function since it is a cubic term, which grows faster than the quadratic or linear terms.

Conclusion:

Thus, the function f(n) can be expressed in Θ-notation as:

$$f(n)=\Theta(n^3)$$

This notation indicates that f(n) grows at the same rate as $n^3$for large n.

Question 7

Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? Using ,-notation, give the average-case and worst-case running times of linear search. Justify your answers

Linear Search Analysis

Linear search is a simple algorithm that checks each element in an array sequentially until the desired element is found or the end of the array is reached.

Average Case

1. **Average Number of Comparisons**:

    - If the array has n elements, the desired element could be in any position with equal likelihood.
    - On average, the element will be found after checking about half of the elements.
    - Therefore, the average number of comparisons is:

$$\text{Average comparisons}=\frac{n+1}{2}\approx\frac{n}{2}$$

Average-Case Running Time:

- Using Big O notation, the average-case running time is: O(n)
- Justification: Even though it's $\frac{n}{2}$ comparisons on average, we express it in terms of Big O as O(n) because we focus on the highest-order term as n grows large.

Worst Case

Worst Number of Comparisons:

- The worst-case scenario occurs when the desired element is not present in the array or is the last element.
- In both cases, all n elements must be checked.
- Therefore, the worst number of comparisons is:

Worst comparisons = n

Worst-Case Running Time:

- The worst-case running time is: O(n)
- Justification: Since the search must check every element in the worst case, the time complexity is linear with respect to the number of elements.

Question 8

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

Insertion Sort Optimization

Best Case:

Insertion Sort already has a best-case time complexity of O(n) when the input array is already sorted or nearly sorted. You can improve it further by adding a check at the beginning to see if the array is already sorted.

Implementation:

Before proceeding with the sort, iterate through the array once to check if each element is less than or equal to the next element. If this condition holds for the entire array, you can conclude it's sorted and return early.

Merge Sort Modification

Best Case:

Merge Sort typically has a time complexity of O(nlogn), but you can improve its best-case performance by recognizing if subarrays are already sorted.

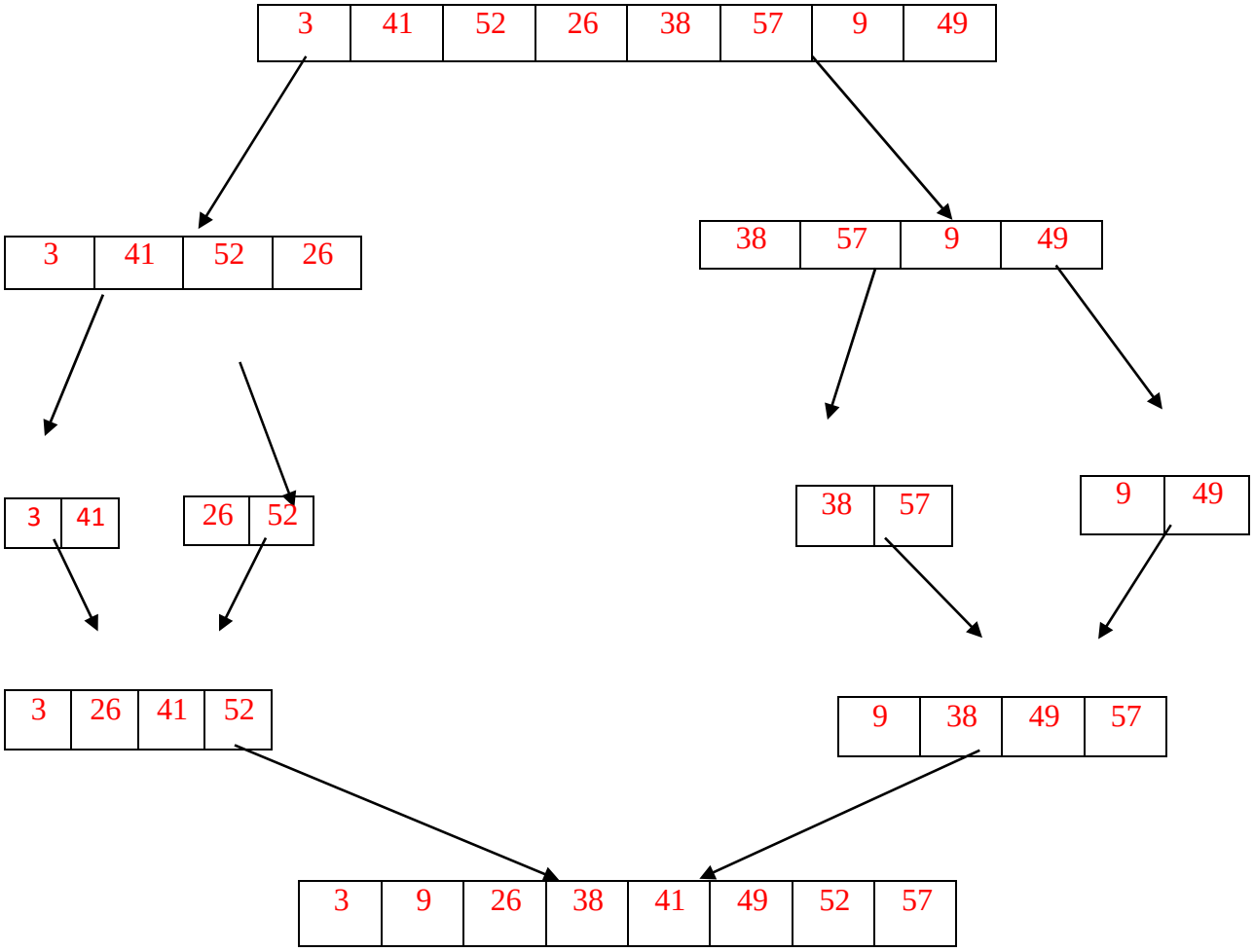
Implementation:

Before merging two sorted subarrays, check if the last element of the first subarray is less than or equal to the first element of the second subarray. If true, the entire array is already sorted, and no further action is necessary.

Question 9

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}.

Merge sorting



Question 10

The test in line 1 of the MERGE-SORT procedure reads r, if P >= r rather then p≠ r if merge sort is called with p > r then subarray A[p:r] is empty is empty. Argue that as long as the initial call of MERGE-SORT.(A, 1,n) has n ≥ 1, the test p≠r suffices to ensure that no recursive call has p > r.

Initial Call:

- The initial call is `MERGE-SORT(A, 1, n)`.
- Here, p=1and r=n. Given that n≥1, the range of the subarray is valid.

Recursive Splitting:

- The `MERGE-SORT` algorithm divides the array into two halves. The midpoint q is calculated as:

$$q=\left\lfloor \frac{p+r}{2} \right\rfloor$$

It then recursively calls itself on the two halves:

- MERGE-SORT(A, p, q)
- MERGE-SORT(A, q + 1, r)

Analyzing the Recursive Calls

- **First Recursive Call**: MERGE-SORT(A, p, q)
  - Here, p remains the same (1) and q is less than or equal to r (which is n).
  - Since q is derived from the midpoint of p and r, q will always be less than or equal to r, ensuring $p \leq q \leq r$.

- **Second Recursive Call**: MERGE-SORT(A, q + 1, r)
  - Here, q+1will be greater than q but still less than or equal to r.
  - Hence, we have $q+1 \leq r$ because q can never exceed r.

Base Case

- The base case of the recursion is when p is not less than r (i.e., when $p \geq r$).
- If p equals r, we have a single-element subarray, which is trivially sorted.
- If p is greater than r (which cannot happen given the initial constraints), the call would not be made because the condition $p<r$ fails.

Question 11

State a loop invariant for the while loop of lines 12-18 of the MERGE procedure. Show how to use it, along with the while loops of lines 20-23 and 24-27, to prove that the MERGE procedure is correct.

Loop Invariant for the While Loop (Lines 12-18)

Loop Invariant:

At the start of each iteration of the loop (lines 12-18), the elements in the merged array (let's call it C) up to the current position contain the smallest elements from both subarrays A and B, sorted in non-decreasing order.

Using the Loop Invariant

Initialization:

Before the first iteration of the loop, both subarrays A and B are sorted. Initially, C is empty (or contains a valid initial state, such as only C[0] being filled with the smallest element). Hence, the invariant holds.

Maintenance:

Assume the invariant holds at the beginning of the current iteration:

- If A[i] <= B[j], then the next element in C is A[i]. After this assignment, the elements in C remain sorted because the next smallest element is added.
- If B[j] < A[i], then B[j] is assigned to C. The same reasoning applies: the order is maintained.
- The indices i and j are incremented appropriately, ensuring that the next iteration considers the correct elements from A and B.

Termination:

The loop terminates when one of the arrays is fully traversed. At this point, the elements in C consist of all elements from both A and B that have been compared, maintaining their sorted order due to our invariant.

Proving the Remaining While Loops (Lines 20-23 and 24-27)

After the main while loop (lines 12-18), the two additional while loops handle any remaining elements in either A or B.

While Loop :

Loop Invariant:

At the start of each iteration of this loop, all elements from A[i] to A[m] (where m is the end of A) have not yet been added to C, and all elements in C remain sorted.

Maintenance:

Each time an element from A is added to C, it is the next smallest element due to the fact that A is already sorted, and all previously added elements from C are also sorted. This maintains the order in C.

While Loop:

Loop Invariant:

At the start of each iteration of this loop, all elements from B[j] to B[n] (where n is the end of B) have not yet been added to C, and all elements in C remain sorted.

Maintenance:

Similar to the previous loop, each element from B added to C is the next smallest element, maintaining the sorted property of C.

Conclusion:

After executing all three loops:

- The merged array C contains all elements from A and B, sorted in non-decreasing order.
- The initial loop invariant holds throughout, proving that at termination, C is correctly merged and sorted.

Thus, the correctness of the **MERGE** procedure is established through the use of loop invariants.

Let me know if you have another question!

Question 12

Use mathematical induction to show that when n ≥ 2 is an exact power of 2,the solution of recurrence t(n) = 2 if n D 2; 2T.n=2/ C n if n > 2 is T(n)= n log n .

To prove that T(n)=n log n for the recurrence relation:

$$t(n) = \begin{cases} 2T(\frac{n}{2}) + C(n) & \text{if } n \geq 2 \\ 2 & \text{if } n=2 \end{cases}$$

using mathematical induction, we follow these steps:

Base Case

For n=2:

$$T(2)=2$$

We need to check if this is equal to 2log2:

$$2\log 2=2\cdot 1=2$$

The base case holds true.

Inductive Step

Assume the hypothesis holds for n=k where k is an exact power of 2 (i.e., $k=2^m$)

$$T(k)=k \log k$$

We need to prove it holds for n=2k:

$$T(2k) = 2T(\frac{2k}{2})+ C(2k) = 2T(k) + 2Ck$$

Using the inductive hypothesis T(k)=k log k:

$$T(2k)=2(k \log k)+2Ck$$

This simplifies to:

$$T(2k)=2k \log k+2Ck$$

We need to show that:

$$T(2k)=2k\log (2k)$$

Using the properties of logarithms:

$$\text{Log } (2k)=\log 2+\log$$

Thus:

$$2k\log(2k)=2k(\log 2+\log k)=2k\log 2+2k$$

Comparing T(2k):

$$T(2k)=2k\log k+2Ck$$

We need 2Ck to match 2klog2:

$$2k\log k+2Ck=2k\log 2+2k\log k$$

Question 13

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.

Pseudocode for insertion sort:

```
Function RecursiveInsertionSort(A, n):
    if n <= 1:
        return  // Base case: an array of size 1 is already sorted
    // Recursively sort the first n-1 elements
    RecursiveInsertionSort(A, n - 1)
    // Insert the nth element into the sorted subarray A[1...n-1]
    Insert(A, n)
Function Insert(A, n):
    key = A[n] // The element to be inserted
    i = n - 1  // Index of the last sorted element
    // Move elements of A[1...n-1], that are greater than key, to one position ahead
    while i > 0 and A[i] > key:
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key  // Place the key in its correct position
```

Recurrence for Worst-Case Running Time

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

$$T(n)=\begin{cases} o(1) & \text{if } n = 1 \\ T(n-1) + o(n) & \text{if } n > 1 \end{cases}$$

Explanation of the Recurrence:

Base Case:

When n=1, the running time is O(1)since no sorting is required.

Recursive Case:

For n>1, we first sort the first n−1 elements (taking time T(n−1), and then insert the nth element into its correct position. The insertion takes O(n) time in the worst case because it may require moving all previously sorted elements.

Solving the Recurrence

Using the recurrence relation:

Expand:

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

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

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

Base Case:

When $k=n-1$, we reach $T(1)=O$.

Sum:

$$T(n)=O(1)+O(2)+O(3)+\ldots+O(n)$$

This is the sum of the first n integers, which gives:

$$T(n)=O(n^2)$$

Question 14

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 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 log n.

Understanding Binary Search

Binary search is a highly efficient algorithm used to find a specific value in a **sorted** array. Unlike linear search, which checks each element one by one, binary search takes advantage of the fact that the array is sorted. It repeatedly divides the search space in half, allowing it to eliminate large portions of the array with each comparison.

How Binary Search Works

Here's the basic idea:

Start with the entire array:

Begin with the lowest index (low) set to 0 and the highest index (high) set to the last index of the array.

Find the middle:

Calculate the middle index of the current subarray. This is done by taking the average of low and high.

Compare the middle value:

- If the middle value is equal to the target value you're searching for, you've found your item!
- If the middle value is less than the target, then you know that the target must be in the upper half of the array. You can ignore the lower half.
- If the middle value is greater than the target, the target must be in the lower half, so you can ignore the upper half.

Repeat:

Adjust your low and high indices based on the comparisons and repeat the process until you either find the target or the subarray size becomes zero.

Pseudocode for Binary Search

Here's the pseudocode to illustrate how binary search works:

```
BinarySearch(array, target, low, high)

  while low <= high do

    mid = low + (high - low) / 2  // To avoid overflow
```

```
    if array[mid] == target then
        return mid              // Target found at index mid
    else if array[mid] < target then
        low = mid + 1           // Search in the upper half
    else
        high = mid - 1          // Search in the lower half
end while
return -1                       // Target not found
```

Question 15

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 og n/ time in the worst case.

Algorithm Overview

Sort the Array:

First, we sort the array SSS. Sorting takes O(nlogn) time.

Two-Pointer Technique:

After sorting, we can use two pointers to find the two elements that sum to x:

- Initialize one pointer at the beginning of the sorted array (let's call it left) and the other at the end of the array (let's call it right).
- While left is less than right:
  - Calculate the sum of the elements at the left and right pointers.
  - If the sum is equal to x, we have found the two elements.
  - If the sum is less than x, increment the left pointer to increase the sum.
  - If the sum is greater than x, decrement the right pointer to decrease the sum.
- If the pointers meet without finding a pair, there are no two elements that sum to x.

Problems

Insertion sort on small arrays in merge sort Although merge sort runs in ,.n lg n/ worst-case time and insertion sort runs in ,.n 2 / worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when sub problems become sufficiently small. Consider a modification to merge sort in which n=k subsists 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 subsists, each of length k , in ,Θ(nk) worst-case time.

Insertion Sort Time Complexity

When we divide the array of size n into k sublists, each sublist has a length of k. There are n/k such sublists.

The time complexity of insertion sort is O(k^2)for sorting one sublist of length k. Since there are n/k sublists, the total time to sort all sublists using insertion sort is:

$$T_{insertion} = \frac{n}{k} \cdot o(k^2) = 0\left(\frac{n.k^2}{k}\right) = o(nk)$$

b. Show how to merge the sub lists in ,.Θ(nlg($^n/_k$)) worst-case time.

Merging Sublists Time Complexity

To merge n/k sorted sublists, each of length k, we can use a standard merging mechanism. The merge step can be efficiently implemented using a min-heap (or priority queue). The total number of elements being merged is n.

In the worst case, merging n/k sublists (each containing k elements) takes:

1. Initializing the heap with n/k elements: O(n/k)
2. Merging all n elements: O(nlog(n/k))

Thus, the total time complexity for merging is:

$$T_{merge} = o\left(n \log\left(\frac{n}{k}\right)\right)$$

c. Given that the modified algorithm runs in , $\theta n\ lo(^n/_k)$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 ,-notation?

Overall Time Complexity of Modified Algorithm

The overall running time of the modified merge sort algorithm is given by:

$$T_{modified} = o(nk) + o(nlog\ (\frac{n}{k}))$$

To find the largest value of k such that $T_{modified}$ has the same running time as standard merge sort, which is O(nlog n), we set:

$$o(nk) + o\ (n \log\ (\frac{n}{k})) = o(nlogn)$$

For O(nk) to not exceed O(nlogn):

$$k \leq O(\log n)$$

Thus, the largest value of k that maintains the same asymptotic behavior as standard merge sort is:

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

d. How should you choose k in practice?

Practical Choice of k

In practice, the choice of k should strike a balance between the overhead of recursion and the efficiency of insertion sort. Common choices are:

1. **Empirical Testing:**

   Implement the algorithm and test with various values of k to measure performance across typical datasets.

2. **Small Values:**

   Often, k is chosen to be a small constant (e.g., 10 or 20) because insertion sort becomes efficient for very small arrays due to low overhead and constant factors.

3. **Adaptivity:**

Choose k based on the characteristics of the data. If the data is mostly sorted, a larger k might be more beneficial.

Chapter 3

Question 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 $O(n^2)$.

Question 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

To analyze the running time of Selection Sort using reasoning similar to Insertion Sort, let's

break it down step-by-step:

Selection Sort Process:

1. Find the smallest element in the array and swap it with the first element.

2. Find the second smallest element and swap it with the second element.

3. Continue this process for all elements.

Comparisons in Selection Sort:

For an array of size n:

1. In the first pass, it makes $n-1$comparisons to find the smallest element.

2. In the second pass, it makes n−2comparisons to find the second smallest element.

3. This continues until the last pass, where it makes 1 comparison.

Total Comparisons:

The total number of comparisons is:

T(n)=n(n−1)/2

Time Complexity:

Since T(n)=n(n−1)/2 the time complexity of Selection Sort is O(n^2) similar to Insertion Sort. However, unlike Insertion Sort, the number of swaps in Selection Sort is O(n), since it performs exactly one swap per pass.

Question 3

Suppose that ¸ is a fraction in the range 0 < α < 1 . Show how to generalize the lower-bound argument for insertion sort to consider an input in which the ¸n largest values start in the first α n positions. What additional restriction do you need to put on ¸ ? What value of α maximizes the number of times that the α n largest values must pass through each of the middle .1 - 2 α /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)nTo maximize the number of comparisons, α=1/4 The additional restriction is that αn\alpha nαn must be an integer.

Question 4

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

Step-by-Step Proof:

Let f(n) and g(n) be asymptotically nonnegative functions (i.e., they are nonnegative for large n).

Upper Bound:

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

his is because the maximum of two values is always less than or equal to their sum. So:

$$\max(f(n), g(n)) = O(f(n)+g(n))$$

Lower Bound:

$$\max(f(n), g(n)) \geq f(n)+g(n)) >= f(n)+g(n)/2$$

This is because the maximum of two numbers is always at least half their sum. Therefore:

$$\max(f(n), g(n)) = \Omega(f(n)+g(n))$$

This gives us the lower bound.

Since max(f(n), g(n)) is both O(f(n)+g(n) and Ω(f(n)+g(n)), we conclude:

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

Thus, max(f(n), g(n))=Θ(f(n)+g(n))

Question 5

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 O(n^2) represents an upper bound, not a lower bound. The correct way to describe a lower bound is to use Ω\Omega-notation, as in "The running time is at least Ω(n^2)."

Question 6

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

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)

Question 7

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

Big-O Notation:

- T(n)=O(g(n)) means there exist constants C>0 and $n_0$such that for all n ≥ n_0

$$T(n) \leq C \cdot g(n)$$

Big-Omega Notation:

- T(n)=Ω(g(n)) means there exist constants C′>0 and n_1 such that for all n ≥ n_1.

$$T(n) \geq C' \cdot g(n)$$

Theta Notation:

- T(n)=Θ(g(n)) means T(n) is both O(g(n)) and Ω(g(n)). Specifically, there exist constants C1,c_2 > 0 and n_2 such that for all n ≥ n_2:

$$C1 \cdot g(n) \leq T(n) \leq C2 \cdot g(n)$$

Proof

If T(n)=Θ(g(n))T(n) then T(n)=O(g(n))T(n) and T(n)=Ω(g(n))T(n)

Assuming T(n)=Θ(g(n)):

1. **By definition of Θ:**

There exist constants $c_1, c_2 > 0$ and n_2 such that for all $n \geq n_2$:

$$c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot g(n)$$

2. **This inequality implies**

Upper Bound (Big-O):

$$T(n) \leq c_2 \cdot g(n) \text{ for } n \geq n2 \implies T(n)=O(g(n))$$

Lower Bound (Big-Omega):

$$T(n) \geq c_1 \cdot g(n) \text{ for } n \geq n2 \implies T(n)=\Omega(g(n))$$

Thus, if $T(n)=\Theta(g(n))$ then $T(n)=O(g(n))$ and $T(n)=\Omega(g(n))T(n)$

If $T(n)=O(g(n))T(n)$ and $T(n)=\Omega(g(n))T(n)$ then $T(n)=\Theta(g(n))T(n)$

Now assume $T(n)=O(g(n))$ and $T(n)=\Omega(g(n))T(n)$

By O(g(n)):

- There exist constants $c_2 > 0$ and n_0 such that for all $n \geq n_0$:

$$T(n) \leq c_2 \cdot g$$

Establishing the Bounds

You defined $n^2=\max(n_0, n_1)$. For all $n \geq n_2$, the inequalities hold:

1. **From T(n)=O(g(n)):**

$$T(n) \leq c_2 \cdot g(n) \qquad \text{for all } n \geq n_0$$

Since $n_0 \leq n_2$ this holds for all $n_0 \leq n_2$

2. **From T(n)=Ω(g(n)):**

$$T(n) \geq c_1 \cdot g(n) \qquad \text{for all } n \geq n_1$$

Since $n_1 \leq n_2$ this holds for all $n \geq n_2$.

Combining the Results

Now, combining both inequalities for $n \geq n_2$:

$$c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot g(n)$$

https://github.com/Zamirali1/DSA-Assignment/upload/main