“Abbottabad University Of Science And Technology”

➢ **Submitted by:**

Nabiha Tahir

➢ **Submitted to:**

Sir Jamal Abdul Ahad

➢ **Department:**

BScs 3A

➢ **Subject:**

Data Structure and Algorithms

➢ **Semester:**

3rd (A)

➢ **Roll No:**

14662

➢ **Assignment No:**

01

➢ **Submitted Date:**

31/10/2024

Chapter No 01

The Role of Algorithms in Computing

Exercise 1

<u>Question:</u>

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

<u>Answer:</u>

Sorting Example:

Imagine you're a librarian tasked with organizing a newly acquired collection of books. You want to arrange them on the shelves alphabetically by author's last name. To do this efficiently, you'd likely sort the books first. This process involves comparing each book's author's last name to others and placing them in the correct order.

Shortest Distance Example:

Consider a delivery driver planning their route for the day. They have a list of addresses to visit. To minimize the total distance traveled, they need to find the most efficient route. This involves calculating the shortest distance between each pair of addresses. By using algorithms like Dijkstra's algorithm, the driver can determine the optimal path, saving time and fuel.

<u>Question:</u>

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

<u>Answer:</u>

Resource Utilization:

- **Memory Usage:** Efficient algorithms minimize the amount of memory required to process data, especially important for large datasets or resource-constrained systems.
- **CPU Usage:** Optimal algorithms reduce the number of computations and operations, leading to lower CPU usage and improved performance.
- **Network Bandwidth:** In network-based applications, efficient algorithms minimize data transfer, reducing network congestion and latency.

Accuracy and Correctness:

- **Error Rate:** Algorithms should produce accurate results with minimal errors or mistakes.
- **Robustness:** Algorithms should be able to handle unexpected inputs or edge cases without crashing or producing incorrect results.

Scalability:

- **Performance under Load:** As the amount of data or the complexity of the problem increases, the algorithm should maintain acceptable performance.
- **Ability to Handle Growth:** The algorithm should be adaptable to changes in data size or complexity.

Maintainability and Readability:

- **Code Clarity:** Well-written code is easier to understand, debug, and modify, reducing maintenance costs.
- **Modularity:** Breaking down the algorithm into smaller, reusable components improves maintainability and facilitates testing.

Energy Efficiency:

- **Power Consumption:** In battery-powered devices or energy-constrained environments, algorithms that minimize power consumption are essential.

Security and Privacy:

- **Data Protection:** Algorithms should be designed to protect sensitive data from unauthorized access or breaches.
- **Privacy-Preserving Techniques:** In certain applications, algorithms may need to be designed to preserve user privacy.

By considering these factors in addition to speed, we can develop algorithms that are not only fast but also reliable, efficient, and suitable for real-world applications.

Question:

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

Answer:

Strengths:

- **Efficient Lookup:** Hash tables provide constant-time average-case lookup, insertion, and deletion operations. This makes them ideal for scenarios where quick access to elements is crucial.
- **Flexible Key Types:** Hash tables can store elements with various key types, including strings, integers, and custom objects.
- **No Predefined Order:** Elements in a hash table are not stored in any particular order, which can be beneficial for certain applications where order is not a priority.

Limitations:

- **Hash Collisions:** When two different keys hash to the same index, a collision occurs. While effective collision resolution techniques like chaining or open addressing can mitigate this issue, collisions can still impact performance, especially in cases of high load factors.
- **Memory Overhead:** Hash tables often require additional memory for storing the hash table itself, the hash function, and potential collision resolution structures.
- **Unordered Access:** If you need to access elements in a specific order, hash tables are not the best choice. You'll need to use additional data structures or sort the elements after retrieval.
- **Sensitivity to Hash Function:** The choice of a good hash function is crucial for the performance of a hash table. A poorly designed hash function can lead to frequent collisions and degraded performance.

Real-world Applications:

- **Dictionaries and Sets:** Hash tables are commonly used to implement dictionaries and sets in programming languages.
- **Database Indexing:** Hash indexes are used to speed up database queries by directly accessing data based on a key value.
- **Caching:** Hash tables are used to store frequently accessed data in memory, reducing the need for expensive disk or network operations.
- **Routing Tables:** In networking, hash tables are used to efficiently map IP addresses to corresponding routing information.

Question:

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

Answer:

Similarities between Shortest-Path and Traveling-Salesperson Problems:

- **Graph-based:** Both problems are fundamentally graph-based, involving a set of nodes (vertices) connected by edges.
- **Optimization:** Both problems aim to find the optimal path or tour within a graph, minimizing a certain cost function (e.g., distance, time, or cost).
- **Complexity:** Both problems are NP-hard, meaning that finding exact solutions for large instances can be computationally expensive.

Differences between Shortest-Path and Traveling-Salesperson Problems:

Start and End Points:

    - **Shortest-Path:** Typically involves finding the shortest path between a specific source and destination node.

- **Traveling-Salesperson:** Requires finding the shortest cycle that visits every node exactly once and returns to the starting node.

Path Constraints:

- **Shortest-Path:** The path can be any sequence of edges that connects the source and destination.
- **Traveling-Salesperson:** The path must be a Hamiltonian cycle, visiting every node exactly once.

Solution Approaches:

- **Shortest-Path:** Algorithms like Dijkstra's algorithm and Bellman-Ford algorithm are commonly used to find optimal solutions.
- **Traveling-Salesperson:** Exact solutions for large instances are often impractical. Approximation algorithms and heuristics, such as the nearest neighbor algorithm and the Christofides algorithm, are used to find near-optimal solutions.

Question:

A real-world problem in which only the best solution will do. Then come up with one in which “approximately” the best solution is good enough

Answer:

Real-world problem requiring the best solution:

- **Medical Diagnosis:** In a critical medical situation, such as diagnosing a rare disease, it's imperative to have the most accurate diagnosis. A suboptimal solution could lead to incorrect treatment, potentially endangering the patient's life.

Real-world problem where an approximately best solution is good enough:

- **Route Optimization for Delivery Trucks:** While finding the absolute shortest route for a fleet of delivery trucks is ideal, it might not be practical due to traffic conditions, road closures, or other unforeseen factors. An approximately optimal route, which is close to the best possible, can still significantly improve efficiency and reduce costs.

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:

Real-world problem with both types of input:

Network Routing:

- **Entire input available:** In static network routing, the network topology (nodes and links) is known in advance. Algorithms like Dijkstra's algorithm can be used to calculate the shortest paths between any two nodes.
- **Input arrives over time:** In dynamic network routing, the network topology can change due to link failures, congestion, or new nodes being added. In this case, routing protocols like OSPF and BGP continuously gather information about network changes and update routing tables accordingly.

This scenario demonstrates how, depending on the specific network configuration and traffic patterns, both types of input can be relevant. Static analysis can be used for initial route planning, while dynamic algorithms are essential to adapt to real-time changes and ensure optimal routing.

Exercise 2

Question:

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

Answer:

Example: Recommendation Systems

Recommendation systems are ubiquitous in modern applications, from suggesting products on e-commerce websites to recommending movies on streaming platforms. These systems rely heavily on algorithms to analyze user behavior and preferences to provide personalized recommendations.

Key Algorithms Involved:

1. **Collaborative Filtering:**
    - **User-based Collaborative Filtering:** This algorithm recommends items to a user based on the ratings of similar users. It calculates similarity between users and recommends items that similar users have rated highly.
    - **Item-based Collaborative Filtering:** This approach recommends items to a user based on their similarity to items the user has already rated highly. It calculates similarity between items and recommends items that are similar to items the user has liked.
2. **Content-Based Filtering:**
    - This algorithm recommends items to a user based on their descriptions and attributes. It analyzes the content of items a user has previously interacted with and recommends similar items. For example, if a user has watched many sci-fi movies, the system might recommend other sci-fi movies.
3. **Hybrid Approaches:**

    - Many recommendation systems combine collaborative filtering and content-based filtering to achieve more accurate and robust recommendations. Hybrid approaches can leverage the strengths of both techniques to provide better results.

Function of the Algorithms:

- **Understanding User Preferences:** Algorithms analyze user behavior, such as past purchases, ratings, and browsing history, to identify patterns and preferences.
- **Item Similarity:** Algorithms calculate the similarity between items based on their features, content, or user ratings.
- **Prediction:** Algorithms predict user ratings or preferences for items they haven't interacted with.
- **Ranking:** Algorithms rank the predicted items to present the most relevant recommendations to the user.
- **Personalization:** Algorithms tailor recommendations to individual users, taking into account their unique preferences and behavior.

By effectively employing these algorithms, recommendation systems can significantly enhance user experience and drive business growth.

Question:

Suppose that for inputs of size n on a particular computer, insertion sort runs in 8n2 steps and merge sort runs in 64 n lg n steps. For which values of n does insertion sort beat merge sort?

Answer:

To find the values of n for which insertion sort beats merge sort, we need to find when:

8n^2 < 64n lg n

Dividing both sides by 8n:

n < 8 lg n

This inequality is true for small values of n. By trying different values, we can find that:

- For n = 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, and 43, the inequality holds.

Therefore, insertion sort beats merge sort for input sizes up to **n = 43**.

Question:

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?

<u>Answer:</u>

To find the smallest value of n where 100n^2 is less than 2^n, we can't solve this equation algebraically. However, we can use a trial-and-error approach or a computational tool to find the approximate solution.

By trying different values of n, we can find that:

- For n = 14, 100n^2 ≈ 19600 and 2^n ≈ 16384.
- For n = 15, 100n^2 ≈ 22500 and 2^n ≈ 32768.

Therefore, the smallest value of n for which 100n^2 is less than 2^n is **n = 15**.

Chapter No 02

Getting Started

Exercise 1

<u>Question:</u>

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

<u>Answer:</u>

1. Initial state

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

2. Iteration 1

- The first element, 31, is considered sorted.

3. Iteration 2

- The second element, 41, is compared with the first element, 31.
- Since 41 is greater than 31, it remains in its position.

4. Iteration 3

- The third element, 59, is compared with the second element, 41.
- Since 59 is greater than 41, it remains in its position.

5. Iteration 4

- The fourth element, 26, is compared with the third element, 59.
- Since 26 is less than 59, it is compared with the second element, 41.
- Since 26 is less than 41, it is compared with the first element, 31.
- Since 26 is less than 31, it is inserted at the beginning of the array.

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

6. Iteration 5

- The fifth element, 41, is compared with the fourth element, 59.
- Since 41 is less than 59, it is compared with the third element, 41.
- Since 41 is equal to 41, it remains in its position.

7. Iteration 6

- The sixth element, 58, is compared with the fifth element, 41.
- Since 58 is greater than 41, it is compared with the fourth element, 59.
- Since 58 is greater than 59, it remains in its position.

8. Final state

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

Explanation:

The insertion sort algorithm works by iterating through the array, starting from the second element. It compares each element with the elements to its left and shifts them to the right until it finds the correct position to insert the current element. This process continues until the entire array is sorted.

Visual Representation:

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

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

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

<u>Question:</u>

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 = 1 to n

Sum = sum + A[i]

Return sum
```

Answer:

Loop Invariant: At the start of each iteration of the loop, `sum` stores the sum of the elements in the subarray `A[1:i-1]`.

Proof of Correctness:

Initialization:

- Before the loop starts, `i = 1` and `sum = 0`.
- The subarray `A[1:0]` is empty, so its sum is 0.
- Therefore, the invariant holds initially.

Maintenance:

- At the start of each iteration, assume the invariant holds, i.e., `sum = A[1] + A[2] + ... + A[i-1]`.
- In the loop, `sum` is updated as `sum = sum + A[i]`.
- This adds the next element `A[i]` to the previous sum, maintaining the invariant for the next iteration.

Termination:

- The loop terminates when `i = n+1`.
- At this point, the invariant states that `sum = A[1] + A[2] + ... + A[n]`.
- This is exactly the sum of all elements in the array `A[1:n]`.

Therefore, the SUM-ARRAY procedure correctly returns the sum of the elements in the array `A[1:n]`.

This proof demonstrates the use of loop invariants to establish the correctness of algorithms. It's a powerful technique for reasoning about iterative algorithms.

Question:

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

Answer:

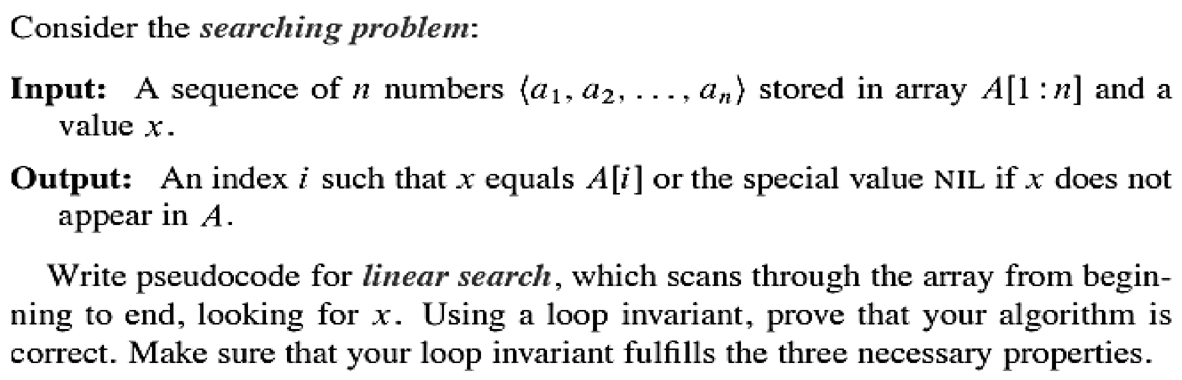
```
INSERTION-SORT (A)
 For j = 2 to length [A]
    Key = A[j]
 // Insert A[j] into the sorted sequence A [1..j-1]
   i = j - 1
```

```
    While i > 0 and A[i] < key
      A [i+1] = A[i]
      i = i - 1
    A [i+1] = key
```

The key modification is in line 5: A[i] < key. This condition ensures that elements larger than the current key are shifted to the right, resulting in a decreasing order.

The rest of the algorithm remains the same, iterating through the array and inserting each element into its correct position within the sorted subarray.

Question:


Consider the *searching problem*:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ 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.

Answer:

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

Loop Invariant:

At the start of each iteration of the loop, the subarray A [1..i-1] does not contain x.

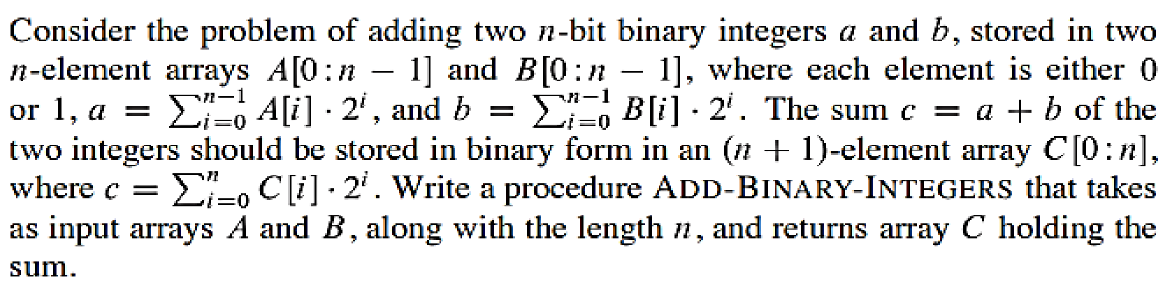
Proof of Correctness:

1. **Initialization:** Before the first iteration (i = 1), the subarray A[1..0] is empty, so it trivially doesn't contain x.
2. **Maintenance:** Assume the invariant holds at the start of the ith iteration. If A[i] == x, the algorithm returns i, which is correct. If A[i] ≠ x, the invariant still holds for the next iteration since x is not in A [1..i].

3. **Termination:** The loop terminates when either i > length [A] or A[i] == x. If i > length [A], the loop has checked all elements and x is not found, so NIL is returned correctly. If A[i] == x, the algorithm has found x and returns its index correctly.

Therefore, the loop invariant is correct, and the LINEAR-SEARCH algorithm correctly finds the index of x in the array A, or returns NIL if x is not present.

Question:



Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays $A[0:n-1]$ and $B[0:n-1]$, where each element is either 0 or 1, $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$, and $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0:n]$, where $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

Answer:

```
ADD-BINARY-INTEGERS (A, B, n)
 C = new array of size n + 1
 carry = 0
 for i = n - 1 to 0
     C[i + 1] = (A[i] + B[i] + carry) mod 2
     carry = (A[i] + B[i] + carry) / 2
 C[0] = carry
 return C
```

Explanation:

1. **Initialization:** We create a new array C of size n + 1 to store the result. We also initialize a variable `carry` to 0, which will keep track of any carry-over from the previous bit addition.
2. **Iterative Addition:** We iterate through the arrays A and B from the rightmost bit (index n - 1) to the leftmost bit (index 0). In each iteration, we calculate the sum of the corresponding bits in A and B, along with the current `carry`.
3. **Bit Addition:** The current bit of C is set to the remainder of the sum when divided by 2. This gives us the binary result of the addition for that bit.
4. **Carry-Over:** The `carry` for the next iteration is calculated by dividing the sum by 2. This effectively carries over any excess to the next higher bit.
5. **Final Carry:** After the loop, the final `carry` (if any) is placed in the leftmost bit of C (index 0).

6. **Return:** The function returns the array C containing the binary sum of A and B.

Example:

Let's consider adding the binary numbers 101 and 110:

A = [1, 0, 1]

B = [1, 1, 0]

i = 2: C [3] = (1 + 0 + 0) mod 2 = 1, carry = 0

i = 1: C [2] = (0 + 1 + 0) mod 2 = 1, carry = 0

i = 0: C [1] = (1 + 1 + 0) mod 2 = 0, carry = 1

C [0] = carry = 1

Therefore, the result is C = [1, 0, 1, 1], which represents the binary number 1011.

Exercise 2

Question:

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

Answer:

To express the function n^3/1000 + 100n^2 - 100n + 3 in terms of big-O notation, we need to find the dominant term, which is the term that grows the fastest as n gets larger.

In this case, the dominant term is n^3/1000. As n approaches infinity, this term will be significantly larger than the other terms.

Therefore, we can express the function in big-O notation as:

O (n^3)

This means that the function grows at most as fast as n^3.

Question:

Consider sorting n numbers stored in array $A[1:n]$ by first finding the smallest element of $A[1:n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2:n]$, and exchange it with $A[2]$. Then find the smallest element of $A[3:n]$, and exchange it with $A[3]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the worst-case running time of selection sort in Θ-notation. Is the best-case running time any better?

Answer:

```
SELECTION-SORT (A)

 for i = 1 to length[A] - 1

    min_index = i

     for j = i + 1 to length[A]

         if A[j] < A[min_index]

             min_index = j

     if min_index != i

        Exchange A[i] with A [min_index]
```

Loop Invariant:

At the start of each iteration of the outer loop, the subarray A[1..i-1] contains the smallest i-1 elements of the original array in sorted order.

Why only n-1 iterations?

After n-1 iterations, the largest element will be in the last position (A[n]). The remaining n-1 elements will be in the correct sorted order. Therefore, there's no need to process the last element.

Worst-case running time:

The worst-case running time of Selection Sort is **O(n^2)**. This occurs when the array is sorted in reverse order. In each iteration of the outer loop, the inner loop needs to scan the entire remaining unsorted portion of the array to find the minimum element.

Best-case running time:

The best-case running time is also **O (n^2)**. Even if the array is already sorted, the algorithm will still perform the same number of comparisons in the inner loop.

Therefore, Selection Sort has a worst-case and best-case running time of O (n^2).

Question:

Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? 34 Chapter 2 Getting Started Using ,Θnotation, give the average-case and worst-case running times of linear search. Justify your answers.

Answer:

Average Case:

In the average case, we assume that the element being searched for is equally likely to be at any position in the array. So, on average, we would need to check half of the elements.

Therefore, the average-case running time is **Θ(n/2)**, which is equivalent to **Θ(n)**.

Worst Case:

In the worst case, the element being searched for is either the first or the last element in the array. In this case, we would need to check all n elements.

Therefore, the worst-case running time is **Θ (n)**.

Justification:

- **Average Case:** The average number of comparisons is proportional to the size of the array, n. As n grows larger, the number of comparisons also grows linearly.
- **Worst Case:** In the worst case, the entire array needs to be scanned, leading to a linear relationship between the number of comparisons and the array size.

So, both the average-case and worst-case running times of linear search are **Θ (n)**.

Question:

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

Answer:

Modifying Sorting Algorithms for Better Best-Case Performance

While many sorting algorithms have inherent worst-case and average-case running times, we can often modify them to improve their best-case performance:

1. Early Termination:

- **Insertion Sort:** If the input array is already sorted, the inner loop of Insertion Sort will terminate early in each iteration, leading to a linear time complexity of O(n).
- **Quick Sort:** By using a more intelligent pivot selection strategy (e.g., median-of-three), we can reduce the likelihood of unbalanced partitions, which can lead to better best-case performance.

2. Input-Specific Optimizations:

- **Radix Sort:** If the input data has specific properties, like being limited to a certain range of values or having a specific distribution, Radix Sort can be optimized to take advantage of these characteristics.

- **Bucket Sort:** Similar to Radix Sort, Bucket Sort can be optimized for specific input distributions.

3. Hybrid Approaches:

- **Intro sort:** This hybrid sorting algorithm combines Quick Sort with Heap Sort. It starts with Quick Sort but switches to Heap Sort when the recursion depth exceeds a certain threshold. This can improve the worst-case performance while maintaining good average-case performance.

Key Considerations:

- **Trade-offs:** Improving the best-case performance often involves sacrificing some efficiency in other cases. For example, a more intelligent pivot selection strategy in Quick Sort might increase the average-case running time.
- **Input Distribution:** The effectiveness of these modifications depends heavily on the input distribution. For example, if the input is already sorted or nearly sorted, Insertion Sort can be very efficient.
- **Algorithm Choice:** The choice of sorting algorithm should be based on the specific use case, considering factors like input size, data distribution, and memory constraints.

By carefully considering these factors and applying appropriate modifications, we can often improve the best-case performance of sorting algorithms without significantly impacting their overall efficiency.

Exercise 3

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).

Answer:

Initial Array:

P

3 41 52 26 38 57 9 49

Divide 1:

P1

3 41 52 26

Divide 2:

P2

3 41

Divide 3:

P3

3

Merge 1:

P4

3 41

Merge 2:

P5

3 26 41 52

Merge 3:

P6

3 9 26 38 41 49 52 57

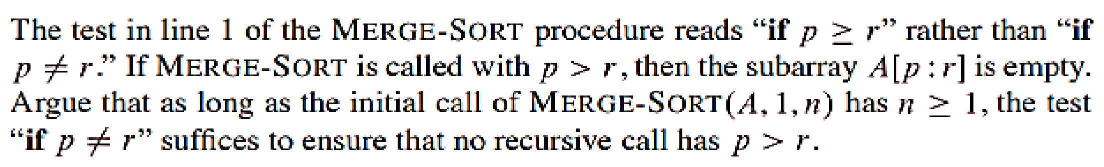
Final Sorted Array:

P

3 9 26 38 41 49 52 57

As you can see, the merge sort algorithm recursively divides the array into smaller subarrays until each subarray contains only one element. Then, it merges these sorted subarrays back together, resulting in a fully sorted array.

Question:



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

Answer:

Here's why the `if p ≠ r` condition is sufficient:

1. **Recursive Calls:** In each recursive call, the MERGE-SORT procedure divides the array into two subarrays, A [p..q] And A[q+1..r], where q = ⌊(p+r)/2⌋.
2. **Base Case:** The base case is reached when `p = r`, meaning the subarray contains only one element. In this case, the subarray is already sorted, so no further recursion is needed.

3. **Subarray Size:** With each recursive call, the size of the subarray either remains the same or is halved. Therefore, the values of `p` and `r` always approach each other.
4. **Condition Guarantee:** As long as the initial call has n ≥ 1, the if p ≠ r condition ensures that the subarray is not empty. If p = r, the recursion stops, and no further calls are made.

In conclusion:

Using the if p ≠ r condition as the base case guarantees that no recursive call will have p > r. This ensures the correct behavior of the MERGE-SORT algorithm.

Question:

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.

Answer:

Loop Invariant for Lines 12-18:

At the start of each iteration of the while loop (lines 12-18), the subarray A[p..k-1] contains the smallest `k-p` elements of the original subarrays A[p..q] and A[q+1..r], in sorted order.

Proof of Correctness Using Loop Invariants:

Initialization:

- Before the first iteration, k = p, and the subarray A[p..k-1] is empty. This trivially satisfies the invariant.

Maintenance:

- During each iteration, the smaller of A[i] and A[j] is copied to A[k], and k is incremented. This ensures that A[p..k-1] still contains the smallest k-p elements in sorted order.

Termination:

- The loop terminates when either i > q or j > r.
- If i > q, it means all elements from A[p..q] have been copied to A[p..k-1]. The remaining elements in A[q+1..r] are already in sorted order and are larger than the elements in A[p..k-1].
- If j > r, it means all elements from A[q+1..r] have been copied to A[p..k-1]. The remaining elements in A[p..q] are already in sorted order and are larger than the elements in A[p..k-1].
- Therefore, in either case, the final subarray A[p..r] contains the merged elements in sorted order.

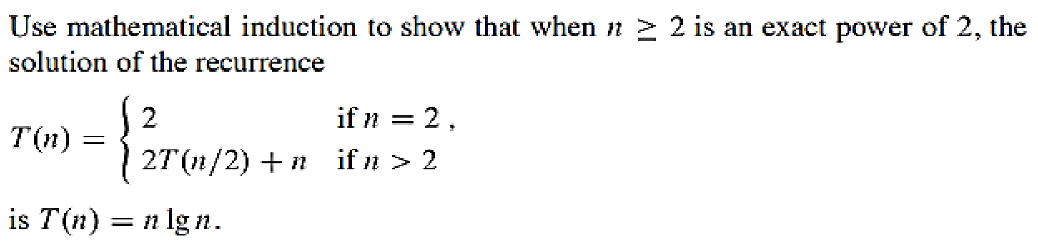
Loop Invariants for Lines 20-23 and 24-27:

These loops are simple copying loops that ensure that any remaining elements in A[p..q] or A[q+1..r] are copied to the correct positions in A[p..r]. The correctness of these loops can be proven similarly to the main loop.

Overall Correctness:

By combining the loop invariants for the three loops and the termination conditions, we can conclude that the MERGE procedure correctly merges the two sorted subarrays A[p..q] and A[q+1..r] into a single sorted subarray A[p..r].

<u>Question:</u>



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

$$T(n) = \begin{cases} 2 & \text{if } n = 2\,, \\ 2T(n/2) + n & \text{if } n > 2 \end{cases}$$

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

<u>Answer:</u>

Using Mathematical Induction to Prove T(n) = n lg n

Given:

- Recurrence relation:
- T(n) =
- { 2   if n = 2
- { 2T(n/2) + n   if n > 2
- Base case: n = 2
- Inductive hypothesis: Assume T(k) = k lg k for all k < n, where n is a power of 2.

To prove: T(n) = n lg n for all n ≥ 2 that are powers of 2.

Proof by Induction:

Base Case:

- When n = 2, T(2) = 2 = 2 lg 2, which satisfies the hypothesis.

Inductive Step:

- Assume T(k) = k lg k for all k < n, where n > 2 and n is a power of 2.
- We want to prove T(n) = n lg n.

Using the recurrence relation:

T(n) = 2T(n/2) + n

By the inductive hypothesis, we can substitute T(n/2) with (n/2) lg(n/2):

T(n) = 2[(n/2) lg(n/2)] + n

Simplifying:

T(n) = n lg(n/2) + n

Using the logarithmic property lg(a/b) = lg(a) - lg(b):

T(n) = n (lg n - lg 2) + n

Since lg 2 = 1:

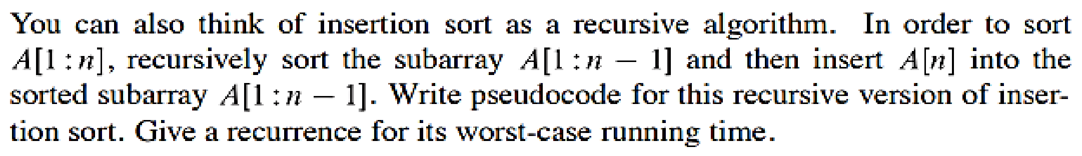
T(n) = n lg n - n + n

Simplifying further:

T(n) = n lg n

Therefore, by the principle of mathematical induction, T(n) = n lg n for all n ≥ 2 that are powers of 2.

<u>Question:</u>



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.

<u>Answer:</u>

```
RECURSIVE-INSERTION-SORT (A, n)
  if n > 1
      RECURSIVE-INSERTION-SORT (A, n - 1)
      INSERTION (A, n)
```

Explanation:

1. **Base Case:** If the array has only one element (n = 1), it's already sorted, so we do nothing.
2. **Recursive Call:** We recursively sort the subarray A[1:n-1].
3. **Insertion:** We insert the last element A[n] into its correct position within the already sorted subarray A[1:n-1]. This can be done using a similar approach to the iterative version of insertion sort, where we compare A[n] with elements in the sorted subarray and shift larger elements to the right.

Worst-Case Running Time Recurrence:

Let T(n) be the worst-case running time of the recursive insertion sort on an array of size n.

- **Base Case:** T(1) = c, where c is a constant time for the base case.
- **Recursive Case:** T(n) = T(n-1) + c', where c' is the time taken for the insertion step. In the worst case, the insertion step might require comparing A[n] with all n-1 elements in the sorted subarray.

Therefore, the recurrence relation for the worst-case running time is:

T(n) =

{ c if n = 1

{ T(n-1) + c' (or c'n) if n > 1

Analysis of the Recurrence:

Solving this recurrence using techniques like the substitution method or the master theorem, we can show that the worst-case running time of the recursive insertion sort is O (n^2). This is the same as the iterative version of insertion sort.

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 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 ϴ(lg n)

Answer:

```
BINARY-SEARCH (A, p, r, x)
 while p ≤ r
   q = floor((p+r)/2)
   if A[q] == x
     return q
   else if A[q] < x
     p = q + 1
   else
```

```
        r = q - 1
    return NIL
```

Explanation:

1. **Initialize:** Set `p` and `r` to the leftmost and rightmost indices of the subarray, respectively.
2. **Check Midpoint:** Calculate the midpoint `q` of the subarray.
3. **Compare:** Compare `A[q]` with the target value `x`.
4. **Adjust Search Space:**
    - If A[q] == x, return q as the index.
    - If A[q] < x, the target value must be in the right half of the subarray. Update p to q+1.
    - If A[q] > x, the target value must be in the left half of the subarray. Update r to q-1.
5. **Repeat:** Continue the process until the subarray is empty (i.e., `p > r`), indicating that the target value is not present.

Worst-Case Running Time Analysis:

In the worst case, the binary search algorithm halves the search space in each iteration. This can be represented by the following recurrence relation:

$T(n) = T(n/2) + c$

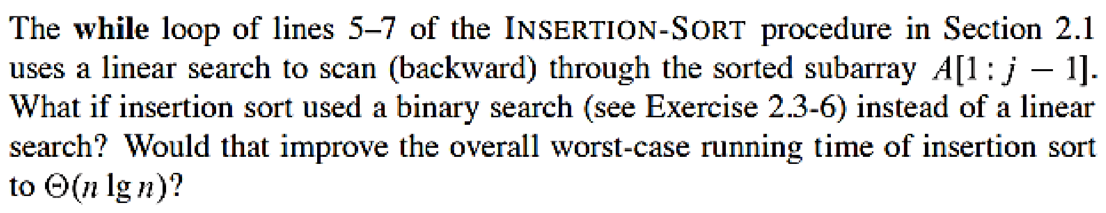
where c is a constant representing the time taken for comparisons and other operations in each iteration.

Using the Master Theorem, we can solve this recurrence to get a worst-case running time of **Θ (lg n)**.

Justification:

The binary search algorithm repeatedly halves the search space, which leads to a logarithmic number of iterations. This is because each iteration reduces the search space by a factor of 2. Therefore, the worst-case running time is proportional to the logarithm of the input size, resulting in a **Θ(lg n)** time complexity.

<u>Question:</u>



The **while** loop of lines 5–7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1:j-1]$. What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

<u>Answer:</u>

Improved Insertion Step:

- **Binary Search:** Using binary search to find the correct position for the current element within the sorted subarray would take O(log n) time in the worst case.
- **Linear Search:** The original linear search takes O(n) time in the worst case.

Overall Worst-Case Running Time:

- **Insertion Sort:** The overall worst-case running time of insertion sort is still O(n^2). This is because, even with the improved insertion step, the outer loop of insertion sort still iterates n-1 times, and in each iteration, the inner loop (whether linear or binary search) can potentially take O(n) or O(log n) time, respectively.

Why O(n^2) persists:

- **Shifting Elements:** The key bottleneck in insertion sort is the shifting of elements to make space for the current element. Even with binary search to find the correct position, shifting elements still takes linear time in the worst case.

Conclusion:

While using binary search within the insertion step can improve the performance of each individual insertion, it doesn't fundamentally change the quadratic nature of the overall algorithm. The worst-case running time of insertion sort remains O(n^2).

To achieve an O (n log n) worst-case running time, we would need to use a fundamentally different sorting algorithm like Merge Sort or Quick Sort.

Question:

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 ~~Θ~~(n lg n) time in the worst case.

Answer:

Algorithm:

1. **Sort the set S:** Sort the set S in ascending order. This can be done using a sorting algorithm like Merge Sort or Quick Sort, which have a worst-case time complexity of O(n log n).
2. **Two-Pointer Technique:**

- Initialize two pointers, low and high.
- Set low to the first element of the sorted array and high to the last element.

3. **Iterative Comparison:**

    - While low is less than high:

        - Calculate the sum of the elements at `low` and `high`.
        - If the sum is equal to `x`, return true (we've found a pair).
        - If the sum is less than `x`, increment `low` to consider larger numbers.
        - If the sum is greater than `x`, decrement `high` to consider smaller numbers.
4. **If no pair is found:**
    - If the loop completes without finding a pair, return false.

Time Complexity Analysis:

- Sorting the set S takes O(n log n) time.
- The two-pointer technique takes O(n) time in the worst case, as we iterate through the sorted array at most once.

Therefore, the overall time complexity of the algorithm is O(n log n), dominated by the sorting step.

Example:

Let's consider the set S = {1, 2, 3, 4, 5} and x = 7.

1. Sort S: {1, 2, 3, 4, 5}
2. Initialize low = 0 and high = 4.
3. Calculate the sum: 1 + 5 = 6. Since 6 < 7, increment low.
4. New low = 1, high = 4.
5. Calculate the sum: 2 + 5 = 7. Since 7 == 7, return true.

This algorithm efficiently finds the pair of elements that sum to x by leveraging the sorted order of the set and the two-pointer technique.

Chapter No 03

Characterizing Running Times

Exercise 1

Question:

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

Answer:

The original lower bound argument for insertion sort, which assumes input sizes are multiples of 3, is a simplification. To handle input sizes that are not necessarily multiples of 3, we need to consider the worst-case scenario for any input size.

Worst-Case Scenario:

The worst-case scenario for insertion sort occurs when the input array is sorted in reverse order. In this case, each element needs to be compared with all preceding elements and shifted into its correct position.

Lower Bound Argument:

Let's consider an input array of size n. In the worst case, the first element needs to be compared with 0 elements, the second element with 1 element, the third element with 2 elements, and so on, up to the nth element, which needs to be compared with n-1 elements.

Therefore, the total number of comparisons in the worst case is:

$0 + 1 + 2 + ... + (n-1)$

This is an arithmetic series, and its sum can be calculated using the formula:

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

Asymptotically, this is **Ω (n^2)**.

Conclusion:

The lower bound for insertion sort remains **Ω (n^2)**, regardless of whether the input size is a multiple of 3 or not. The worst-case scenario, where the input is reverse-sorted, leads to quadratic time complexity.

Question:

Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm.

Answer:

Selection Sort: A Closer Look

How it works:

1. **Find the Minimum:** Scans the entire unsorted array to find the minimum element.
2. **Swap:** Swaps the minimum element with the first element.
3. **Repeat:** Repeats steps 1 and 2 for the remaining unsorted subarray.

Worst-Case Analysis:

- **First Iteration:** Scans n elements to find the minimum.
- **Second Iteration:** Scans n-1 elements.
- **Third Iteration:** Scans n-2 elements.
- **...**
- **Last Iteration:** Scans 2 elements.

Total Comparisons:

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

This is an arithmetic series, and its sum can be calculated as:

n (n+1)/2

Asymptotic Analysis:

As n grows larger, the dominant term in the above expression is n^2. Therefore, the worst-case running time of selection sort is **O (n^2)**.

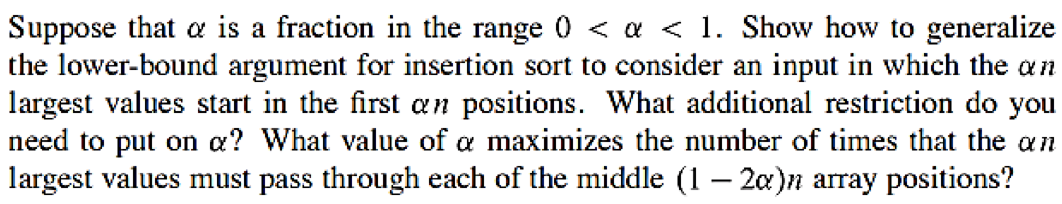
Lower Bound:

Similar to insertion sort, the lower bound for selection sort is also **Ω (n^2)**. In the worst case, each element needs to be compared with all other elements to find the minimum.

Conclusion:

Selection sort has a worst-case time complexity of **Θ (n^2)**. It's a simple algorithm to understand and implement, but its quadratic time complexity makes it inefficient for large datasets.

Question:



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

Answer:

Generalizing the Lower-Bound Argument for Insertion Sort

Understanding the Problem:

We're asked to consider a specific input scenario for insertion sort: the first αn elements of the input array are the largest elements. We want to find a lower bound on the number of comparisons in this case and determine the value of α that maximizes the number of comparisons for the middle elements of the array.

Analysis:

1. **Insertion of the Largest Elements:**
    - The first αn elements, being the largest, will be placed directly in their final positions without any comparisons.
2. **Insertion of the Remaining Elements:**

- For the remaining $(1-\alpha)n$ elements, each element will need to be compared with, on average, $\alpha n/2$ elements from the sorted portion. This is because, on average, half of the sorted portion will be larger than the current element.

Calculating the Total Number of Comparisons:

Total Comparisons = $(1-\alpha)n * (\alpha n/2)$

To maximize this expression, we can take the derivative with respect to α and set it to zero:

$d/d\alpha\ [(1-\alpha)n * (\alpha n/2)] = 0$

Solving this equation, we find that the maximum occurs when $\alpha = 1/2$.

Interpretation:

- **Maximum Comparisons:** When $\alpha = 1/2$, half of the elements are the largest and half are the smallest. This leads to the maximum number of comparisons for the middle elements, as each middle element will need to be compared with approximately half of the larger elements.
- **Lower Bound:** With this optimal α value, the lower bound on the number of comparisons becomes:

Total Comparisons = $(1/2)n * (1/4)n = (1/8)n^2$

This is still **$\Omega(n^2)$**, indicating that insertion sort remains inefficient in this specific scenario.

Conclusion:

Even when the input array is partially sorted, insertion sort can still exhibit quadratic behavior in the worst case. By carefully analyzing the input distribution, we can identify scenarios that lead to the maximum number of comparisons and, thus, the worst-case performance.