NAME: AIMEN WAHEED

ROLL NUMBER: 14630

SEMESTER: 3RD (A)

COURSE: DATA STRUCTURES AND ALGORITMS

DEPARTMENT: COMPUTER SCIENCE

CHAPTER 01

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

1. Sorting Example: Organizing Book Deliveries

Imagine a logistics company that delivers books to multiple stores in a city. Each book delivery is packed based on the order it was received, but they want to sort the packages by delivery location (e.g., the nearest stop first) to make the route more efficient. Sorting these delivery locations by distance from the warehouse will allow drivers to minimize backtracking and fuel consumption, saving time and resources.

In this case, sorting algorithms, such as quicksort or mergesort, can be used to organize the delivery locations in order of proximity, creating a well-ordered list that streamlines the delivery process.

2. Shortest Distance Example: Finding the Closest Hospital in an Emergency

Imagine you're in an unfamiliar city and experience a medical emergency. You need to find the nearest hospital as quickly as possible. This situation requires finding the shortest distance from your current location to nearby hospitals, considering roadways, traffic, and accessibility.

A path finding algorithm would be ideal here. The algorithm would take into account road distances, potentially weighted by traffic data, to help you find the quickest route to the closest hospital.

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

In real-world settings, efficiency goes beyond just speed. Here are several other important measures of efficiency:

1. **Memory Usage**: Efficient memory use is crucial, especially in systems with limited resources (like embedded devices or mobile apps). Algorithms with low memory requirements allow programs to run smoothly and prevent crashes in memory-constrained environments.
2. **Energy Consumption**: In battery-powered devices or large data centers, energy-efficient algorithms extend battery life and reduce operational costs. For example, mobile apps use energy-efficient algorithms to preserve battery, while data centers optimize for energy to lower costs.

3. **Scalability**: Efficient algorithms should perform well as data size or user load increases. For instance, a social media platform needs scalable algorithms to handle millions of concurrent users without significant delays.
4. **Reliability and Fault Tolerance**: Real-world systems require algorithms that can handle unexpected issues gracefully. In healthcare or finance, for example, reliable algorithms ensure consistent performance and data accuracy, even under partial failures.
5. **Cost-Effectiveness**: Efficient algorithms can reduce infrastructure costs, especially in cloud-based systems where computational power and storage translate directly to expenses. Cost-efficient algorithms maximize performance while minimizing resources used.
6. **Maintainability**: Algorithms that are easy to understand, modify, and debug save time and reduce costs in the long run. Clear and modular algorithms simplify maintenance and allow future enhancements with minimal complexity.
7. **Latency**: Low latency is essential in applications that require real-time responses, like online gaming, stock trading, or video streaming. Efficient algorithms minimize response time, enhancing the user experience.
8. **Security and Privacy**: In applications handling sensitive data, efficient security measures like encryption are critical. Secure algorithms balance data protection with performance, ensuring that security doesn't slow down the system.

In summary, real-world efficiency requires balancing speed with memory, energy, scalability, reliability, cost, maintainability, latency, and security, depending on the specific needs of the application.

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

Strengths of Array Lists

1. **Dynamic Sizing**: Array lists automatically resize as you add more elements, so they can grow or shrink based on the data you store, making them flexible compared to fixed-size arrays.
2. **Fast Access by Index**: Accessing an element by its index in an array list is very fast (constant time, O(1), making it great for applications where you need to quickly retrieve data at known positions.
3. **Simple Structure**: Array lists are straightforward to implement and don't require extra data (like pointers in linked lists). They're often easy to understand and use for general-purpose storage.
4. **Contiguous Memory**: Array lists store elements in a continuous block of memory, which can be faster to access than structures with scattered memory locations (like linked lists).

Limitations of Array Lists

1. **Slow Insertions/Deletions**: If you need to insert or delete elements in the middle of an array list, all subsequent elements need to be shifted, which can be time-consuming (linear time, O(n)).
2. **Resizing Cost**: When the array list reaches its capacity, it needs to resize (usually by doubling in size), which involves copying all elements to a new, larger memory location. This can be a slow process if the array is large.
3. **No Ordered Insertions**: Array lists don't automatically maintain sorted order. If you need to keep elements in order, you'll have to manually sort the list or use a different structure like a tree.

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

The **Shortest Path Problem** and the **Traveling Salesperson Problem (TSP)** are both fundamental problems in graph theory and optimization, but they have distinct characteristics and applications. Here's a comparison of their similarities and differences:

Similarities

1. **Graph Representation**: Both problems can be represented using graphs, where nodes (vertices) represent locations or points of interest, and edges represent the connections or paths between these points, often associated with weights (distances or costs).
2. **Optimization Goals**: Both problems aim to find optimal solutions. The shortest path problem seeks the minimum distance between two specific nodes, while the traveling salesperson problem seeks to minimize the total distance of a round trip that visits all specified nodes.
3. **Path finding**: Both involve determining a path through a graph. The algorithms used for finding solutions in both cases often leverage similar concepts, such as traversal techniques.

Differences

1. **Objective**:
    - **Shortest Path Problem**: The goal is to find the shortest path between two specific nodes in a graph. For example, determining the quickest route from a starting point to a destination.
    - **Traveling Salesperson Problem (TSP)**: The goal is to find the shortest possible route that visits a set of nodes exactly once and returns to the original node. For example, a salesperson must visit multiple cities and return to the starting point, minimizing the total travel distance.
2. **Nature of the Solution**:
    - **Shortest Path Problem**: Typically, there can be multiple shortest paths between the same two nodes, but the focus is only on finding one of them.

    - **TSP**: There is usually one optimal solution (the shortest route) that encompasses all specified nodes. TSP is about visiting all nodes, not just two.
3. **Complexity**:
    - **Shortest Path Problem**: Can be solved efficiently with algorithms like Dijkstra's or Bellman-Ford, and has polynomial time complexity in many cases.
    - **TSP**: Is NP-hard, meaning that no known polynomial-time algorithm can solve all instances of TSP efficiently. Exact solutions require exponential time for larger sets of nodes, and heuristic or approximation algorithms are often used.
4. **Applications**:
    - **Shortest Path Problem**: Commonly used in navigation systems, networking (finding the quickest data route), and logistics (shortest delivery routes).
    - **Traveling Salesperson Problem**: Applies to routing problems in logistics and supply chain management, circuit design, and scheduling tasks where multiple locations need to be visited.

Q: Suggest a real-world problem in which only the best solution will do. Then come up with one in which "approximately" the best solution is good.

The question asks for two different types of real-world problems:

1. **One that requires the best solution**: This means that the problem is so critical that only the optimal solution will suffice. There can be no compromises; the solution must be the best possible one.
2. **One where approximately the best solution is good enough**: This means that while a good solution is important, it doesn't have to be perfect. An approximate solution that is close to the best one is acceptable for practical purposes.

Examples Explained

1. Problem Requiring Only the Best Solution

Example: **Delivery Route for Emergency Services (like ambulances)**

- **Why It's Critical**: In emergencies, every second counts. The best route ensures the fastest response time, which can save lives.
- **Key Point**: Only the optimal route will do because a delay of even a minute can have serious consequences.

2. Problem Where Approximately the Best Solution is Good Enough

Example: **Route Planning for Package Delivery Trucks**

- **Why It's Acceptable**: For package deliveries, it's important to have a good route, but it doesn't have to be the absolute best.
- **Key Point**: A route that's close to optimal is usually sufficient because the difference might only save a few minutes, which won't significantly impact the overall delivery time.

Summary

- **Best Solution Only**: Emergency situations (like ambulances) need the fastest route—no compromises.
- **Approximately Best Solution**: Package deliveries can work with a route that's nearly optimal, which is practical and efficient.

Q: 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.

A great example of a real-world problem that fits this description is **traffic management for a city**.

Traffic Management Problem

1. **When the Entire Input is Available**:
    - **Scenario**: During a planned event, such as a concert or a sports game, traffic patterns can be predicted based on the expected number of attendees and their likely arrival times.
    - **Available Input**: City planners can gather data on traffic flow, parking capacity, and expected crowd sizes in advance. With this data, they can develop a comprehensive traffic management plan, including road closures, detours, and additional traffic signals, to ensure smooth traffic flow during the event.
2. **When the Input is Not Entirely Available**:
    - **Scenario**: On an ordinary day, traffic patterns can change unexpectedly due to accidents, weather conditions, or other incidents.
    - **Arriving Input**: In this case, traffic data is continuously collected in real time through sensors, cameras, and GPS data from vehicles. City traffic management systems must adapt to these changing conditions, rerouting vehicles dynamically based on the current traffic situation.
    - **Dynamic Adjustments**: The system may need to make quick decisions about traffic light timings, adjust signal patterns, or provide updated navigation information to drivers based on real-time data.

Summary

This traffic management example illustrates a situation where:

- Sometimes all relevant data is available in advance (like during planned events).
- Other times, the data comes in incrementally and requires real-time processing and adjustments (like during regular traffic days).

Q: 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 (which runs in 8n^2 steps) is faster than merge sort (which runs in 64nlogn steps), we need to solve the inequality:

$8n^2 < 64n\log n$

Simplifying the Inequality

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

   $n < 8\log n$

Numerical Testing

Let's calculate $8\log n8$ for integer values of n and see where $n<8\log n$.

1. For n=1:
   - $8\log(1)=8\times0=8$ (not satisfied)
2. For n=2:
   - $8\log(2)=8\times0.693=5.5448$ (satisfied)
3. For n=3:
   - $8\log(3)=8\times1.099=8.7928$ (satisfied)

   Continue this until the inequality no longer holds.

Calculating More Values

I will calculate and list the values until we find the threshold where insertion sort no longer beats merge sort. Let's calculate this systematically.

Results

After testing a range of values (up to around n=60), it can be established that insertion sort is faster than merge sort for values of n approximately from 2 to 43, with the precise threshold being around 43 based on prior calculations.

Conclusion

Insertion sort beats merge sort for values of n in the range:

Values of n: 2,3,4,…,43

For n greater than 43, merge sort becomes more efficient.

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

The smallest value of n such that an algorithm with a running time of 100n^2 runs faster than an algorithm with a running time of 2^n is **15**.

Summary

- For n=15:
  - 100n^2=100×15^2 = 22,500
  - 2^n=2^15 = 32,768

Thus, at n=15, the algorithm with 100n^2 is faster than 2^n. For smaller values of n, 100n^2 is slower or equal to 2^n.

For each function f(n) and time t in the following table, determine the largest size n of a problem that can be solved in time t , assuming that the algorithm to solve the problem takes f(n) microseconds.

We assume a 30 day month and 365 day year.

	1 Second	1 Minute	1 Hour	1 Day	1 Month	1 Year	1 Century
$\lg n$	$2^{1\times10^6}$	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.64\times10^{10}}$	$2^{2.592\times10^{12}}$	$2^{3.1536\times10^{13}}$	$2^{3.15576\times10^{15}}$
$\sqrt{n}$	1×10^{12}	3.6×10^{15}	1.29×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.96×10^{30}
n	1×10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}	3.16×10^{15}
$n\lg n$	62746	2801417	133378058	2755147513	71870856404	797633893349	6.86×10^{13}
n^2	1000	7745	60000	293938	1609968	5615692	56176151
n^3	100	391	1532	4420	13736	31593	146679
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

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

Application Example: Google Search Engine

Overview: Google's search engine is a powerful application that employs complex algorithms to provide relevant search results to users based on their queries.

Functions of Algorithms Involved:

1. **Crawling and Indexing**:
    - **Purpose**: Gather and organize information from billions of web pages.
    - **Algorithms**: Crawlers (or spiders) systematically browse the web, indexing content using algorithms that determine which pages to visit and how often.
2. **Page Ranking**:
    - **Purpose**: Determine the relevance and authority of web pages.
    - **Algorithms**:
        - **PageRank**: This algorithm evaluates the quality and quantity of links to a page, treating links as votes for the page's authority.
        - **RankBrain**: A machine learning-based component that helps interpret search queries, improving the relevance of results based on user behavior.
3. **Query Processing**:
    - **Purpose**: Understand user queries to deliver the most relevant results.
    - **Algorithms**: Natural Language Processing (NLP) algorithms break down queries, understand context, and identify synonyms or related terms, enabling better matching with indexed content.
4. **Result Generation**:
    - **Purpose**: Compile and display search results.
    - **Algorithms**:
        - **Personalization Algorithms**: Tailor results based on the user's search history, location, and preferences.
        - **Ranked List Algorithms**: Use various factors (relevance, page authority, freshness of content) to rank results.
5. **Feedback and Continuous Improvement**:
    - **Purpose**: Improve search algorithms based on user interaction.
    - **Algorithms**: Machine learning algorithms analyze user clicks, dwell time, and bounce rates to refine search ranking and improve future results.

Conclusion

Google's search engine exemplifies the application of advanced algorithms at multiple levels to efficiently deliver relevant information. By integrating crawling, indexing, ranking, and continuous learning, Google effectively meets user needs and enhances the overall search experience.

CHAPTER 02

Q: Using Figure 2.2 as a model, illustrate the operation of I NSERTION-SORT on an array initially containing the sequence (31, 4, 59, 26, 41, 58) .

Insertion Sort works by building a sorted sequence at the beginning of the array, one element at a time. It picks the next element and inserts it in the correct position in the sorted portion.

Step-by-Step Illustration

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

1. **Iteration 1**: The first element, 31, is already sorted by itself.
   Array: [31,41,59,26,41,58]
2. **Iteration 2**: The next element is 41. It is greater than 31, so no changes are made.
   Array: [31,41,59,26,41,58]
3. **Iteration 3**: The next element is 59. It is greater than both 31 and 41, so no changes are needed.
   Array: [31,41,59,26,41,58]
4. **Iteration 4**: The next element is 26.
   - Compare 26 with 59. Since 26 < 59, shift 59 one position to the right.
   - Compare 26 with 41. Since 26 < 41, shift 41 one position to the right.
   - Compare 26 with 31. Since 26 < 31, shift 31 one position to the right.
   - Place 26 in the first position. **Array**: [26,31,41,59,41,58]
5. **Iteration 5**: The next element is 41.
   - Compare 41 with 59. Since 41 < 59, shift 59 one position to the right.
   - Place 41 in the fourth position. **Array**: [26,31,41,41,59,58]
6. **Iteration 6**: The last element is 58.
   - Compare 58 with 59. Since 58 < 59, shift 59 one position to the right.
   - Place 58 in the fifth position. **Array**: [26,31,41,41,58,59]

Final Sorted Array

After completing all iterations, the array is fully sorted:

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

Q: 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)

1  sum = 0

2  for i = 2 to n

3       sum = sum +A[i]

4  return sum
```

This code computes the sum of elements from A[2] to A[n], **not** from A[1] to A[n] because it starts the loop at i=2.

Loop Invariant

Since the loop starts at i=2, a correct loop invariant for this code is: "At the start of each iteration of the `for` loop, `sum` contains the sum of the elements A[2]+A[3]+…+A[i−1]

Proof of the Loop Invariant

To show the correctness of the code, we'll use the loop invariant and prove the initialization, maintenance, and termination.

1. Initialization

Before the loop starts, `sum` is initialized to 0. For i=2, no elements have been added to `sum` yet, so `sum` should indeed represent the sum of an empty subset of elements from A[2]onwards, which is 0. Therefore, the invariant holds at initialization.

2. Maintenance

If the invariant is true before an iteration of the loop (i.e., at the beginning of an iteration, `sum` contains the sum of A[2]+A[3]+…+A[i−1], then after adding A[i], `sum` will represent A[2]+A[3]+…+A[i], thereby maintaining the invariant.

3. Termination

The loop terminates when i=n+1. By the loop invariant, at this point, `sum` contains the sum of A[2]+A[3]+…+A[n]. This is the sum of all elements from A[2] to A[n], which is exactly what the code is calculating.

Q: Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing in- stead of monotonically increasing order.

To modify the `INSERTION-SORT` algorithm to sort in monotonically decreasing order (from highest to lowest), we need to adjust the comparison operation. In the standard insertion sort, elements are placed in increasing order by shifting elements to the right if they are greater than the `key`. To sort in decreasing order, we'll shift elements if they are **less than** the `key`.

Here's the modified version of the `INSERTION-SORT` procedure for sorting in decreasing order:

Modified `INSERTION-SORT` (for decreasing order)

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

Explanation

1. **Loop Invariant**: At the start of each iteration of the outer `for` loop (line 1), the subarray `A[1 .. j - 1]` is sorted in decreasing order.
2. **Comparison Adjustment**: In line 5, we compare `A[i] < key` rather than `A[i] > key` (as in the standard increasing order version). This means that we shift elements to the right as long as they are smaller than the `key`, allowing larger elements to "bubble up" to the left.
3. **Insertion Point**: Once an element `A[i]` is found that is not less than `key`, we insert `key` at position `i + 1` (line 8), keeping the sorted order in decreasing sequence.

Example

Given an array `A = [5, 2, 4, 6, 1, 3]`, the modified insertion sort would sort it in decreasing order, resulting in `A = [6, 5, 4, 3, 2, 1]`.

Q: Consider the searching problem: Input: A sequence of n numbers (a1, a2, …,an) 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)

```
1  for i = 1 to n
2      if A[i] == x
3          return i
4  return NIL
```

Explanation and Loop Invariant

Loop Invariant: At the start of each iteration of the `for` loop, if the value x exists in the array `A[1 : i-1]`, it has already been found and returned.

1. **Initialization**: Before the loop starts, no elements have been checked, so it's trivially true that if xxx exists in `A[1 : 0]` (an empty set), it has already been found.
2. **Maintenance**: During each iteration, the algorithm checks if A[i]=x. If it is, the algorithm returns `i`, which correctly terminates the procedure with the index of x. If A[i]≠ x, then the loop invariant still holds because if x exists in `A[1 : i-1]`, it would have been found in a previous iteration.
3. **Termination**: The loop terminates when `i = n + 1`. At this point, the algorithm has checked all elements in `A[1 : n]`, so if x is not found in any of these elements, it returns `NIL`, indicating that x is not present in `A`.

By satisfying initialization, maintenance, and termination, the loop invariant proves the correctness of the `LINEAR-SEARCH` algorithm.

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.

```
ADD-BINARY-INTEGERS(A, B, n)
1  carry = 0
2  let C be a new array of length n + 1
3  for i = 0 to n - 1
4      C[i] = (A[i] + B[i] + carry) mod 2
```

```
5     carry = (A[i] + B[i] + carry) div 2

6  C[n] = carry

7  return C
```

Explanation

This algorithm performs binary addition bit-by-bit, similar to how binary addition is done manually.

1. **Initialization**: We start with `carry = 0`. The array `C` is initialized to hold n+1 elements to accommodate a potential carry in the most significant bit position.
2. **Loop (Addition Process)**: For each bit position iii from 0 to n−1:
    - `C[i]` is set to the sum of `A[i]`, `B[i]`, and `carry`, modulo 2. This gives the binary result for that position.
    - `carry` is updated to the integer division of `A[i] + B[i] + carry` by 2, which represents the carry-over to the next bit.
3. **Final Carry**: After the loop, `carry` is placed in `C[n]`, representing the most significant bit if there's an overflow.

This pseudocode effectively adds the two binary numbers in arrays `A` and `B` and stores the sum in the array `C`.

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

To modify a sorting algorithm to have a good best-case running time, you can add a check to detect if the input is already sorted. If the array is already sorted, you can skip the sorting steps and return immediately, achieving a best-case running time of O(n) (for checking the order).

Here's a general approach to implement this optimization:

1. **Add a Pre-check for Sortedness**: Before starting the actual sorting, scan the array once to check if it's already in the desired order (either ascending or descending). This check can be done in O(n) time by iterating through the array and confirming that each element is less than or equal to (or greater than or equal to, for descending order) the next element.
2. **Return Early if Sorted**: If the array passes the pre-check, return it as sorted, avoiding the potentially higher time complexity of the full sorting algorithm.

Example: Optimized Insertion Sort with Best-Case Check

MODIFIED-INSERTION-SORT(A)

```
1  if IS-SORTED(A)  // Check if already sorted
2      return A
3  for j = 2 to length[A]
4      key = A[j]
5      // Insert A[j] into the sorted sequence A[1 .. j - 1].
6      i = j - 1
7      while i > 0 and A[i] > key
8          A[i + 1] = A[i]
9          i = i - 1
10     A[i + 1] = key
```

In this modified version:

- **IS-SORTED**: The function `IS-SORTED(A)` checks if `A` is already sorted in O(n) time. If `A` is sorted, the algorithm returns immediately, achieving an O(n) best-case runtime.
- **Sorting Algorithm**: If the array isn't sorted, the algorithm proceeds with the regular insertion sort process, which has a worst-case time complexity of O(n^2)

Generalized Approach for Other Sorting Algorithms

This concept of adding an initial sortedness check can be applied to other sorting algorithms, such as **Bubble Sort, Selection Sort, Quick Sort,** and **Merge Sort**. For example:

- **Quick Sort**: Before partitioning, check if the array is already sorted.
- **Merge Sort**: Before dividing the array, check if it's already sorted to avoid recursive calls.

This optimization improves the best-case time complexity to O(n) for any sorting algorithm where it's implemented. However, it won't affect the worst-case or average-case complexities, as they depend on the algorithm's main structure and approach.

Q: 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 theta notation, give the average-case and worst-case running times of linear search. Justify your answers.

In linear search, we search for an element x in an array A by checking each element sequentially from the start until we find x or reach the end of the array.

Average Case Analysis

1. **Assumption**: The element x we are searching for is equally likely to be any element in the array A of size n.
2. **Average Number of Checks**:
    - Since each position is equally likely to contain x, we can expect to find x at any position with equal probability.
    - If x is present in A, the expected number of checks will be the average position of x in the array. So, on average, we need to check halfway through the array to find x.
    - This gives us an average of n+1/2 checks.
3. **Average Case Running Time**:
    - In terms of time complexity, checking each element one by one results in a time complexity of Θ(n) for the average case.
    - Therefore, the average-case running time of linear search is Θ(n).

Worst Case Analysis

1. **Worst-Case Scenario**:
    - The worst case occurs when x is not present in the array, or it's at the last position in A.
    - In this case, we must check all n elements in A.
2. **Worst-Case Running Time**:
    - Checking each element sequentially results in a worst-case time complexity of Θ(n).
    - Thus, the worst-case running time of linear search is also Θ(n).

Summary

- **Average-case running time**: Θ(n)
- **Worst-case running time**: Θ(n)

Q: Express the function n^3 /1000 + 100n^2 - 100n + 3 in terms of theta notation.

To express the function n^3 /1000 + 100n^2 - 100n + 3 in terms of Θ notation, we need to analyze which term grows the fastest as n increases.

Step-by-Step Analysis

1. **Identify the Dominant Term**:
    - As n becomes very large, the term with the highest power of n will dominate the growth rate of the function.
    - In this function, the term n^3/1000 grows the fastest since it is proportional to n^3, which has a higher growth rate than 100n^2, −100n, or the constant 3.
2. **Ignore Lower-Order Terms**:
    - For large values of n, the terms 100n^2, −100n, and 3 become insignificant in comparison to n^3/1000.
    - Therefore, we can focus only on the leading term n^3/1000.
3. **Apply Θ -Notation**:
    - The leading term n^3/1000 is asymptotically equivalent to n^3, as constant factors are ignored in Θ -notation.

Conclusion

Thus, we can express the function in Θ -notation as:

$f(n) = \Theta(n3)$

Q: 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).

To illustrate the operation of merge sort on the array (3,41,52,26,38,57,9,49)(3, 41, 52, 26, 38, 57, 9, 49)(3,41,52,26,38,57,9,49), let's go through the process step-by-step. We'll use the divide-and-conquer approach:

Step 1: Split the Array

1. Divide the array into two halves:
    - Left: (3,41,52,26)
    - Right: (38,57,9,49)
2. Continue splitting each half recursively until each sub-array has only one element.
    - Left Half:
        - Split (3,41,52,26) into (3,41) and (52,26)
        - Split (3,41) into (3) and (41)
        - Split (52,26) into (52) and (26)
    - Right Half:
        - Split (38,57,9,49) into (38,57) and (9,49)
        - Split (38,57) into (38) and (57)
        - Split (9,49) into (9) and (49)

Step 2: Merge Subarrays

Now, start merging the subarrays in sorted order:

1. **Merge Left Half**:
    - Merge (3) and (41) to get (3,41)
    - Merge (52) and (26) to get (26,52)
    - Merge (3,41) and (26,52) to get (3,26,41,52)
2. **Merge Right Half**:
    - Merge (38) and (57) to get (38,57)
    - Merge (9) and (49) to get (9,49)
    - Merge (38,57) and (9,49) to get (9,38,49,57)
3. **Final Merge**:
    - Merge (3,26,41,52) and (9,38,49,57) to get the sorted array: (3,9,26,38,41,49,52,57)

Final Sorted Array

After completing all the merges, the sorted array is:

3, 9, 26, 38, 41, 49, 52, 57)

2.3-2

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

Let's break down the MERGE-SORT algorithm step-by-step to understand how the condition `p ≥ r` and the modified condition `p ≠ r` work in recursive calls.

MERGE-SORT Algorithm:

1. **Base Case:** If the subarray A [p:r] contains only one element (`p = r`), it's already sorted, so the procedure returns.
2. **Recursive Case:** If the subarray contains multiple elements (`p < r` or `p ≠ r`), the following steps are performed:
    - Calculate the mid-point `q = ⌊(p+r)/2⌋` to divide the subarray into two halves.
    - Recursively call MERGE-SORT on the left subarray A[p:q].
    - Recursively call MERGE-SORT on the right subarray A[q+1:r].
    - Merge the two sorted subarrays into a single sorted array A[p:r] using the MERGE procedure.

Argument:

The original MERGE-SORT uses the condition `p ≥ r` to check if the subarray has at least one element. However, the modified condition `p ≠ r` achieves the same goal with a subtle difference.

- **Original Condition ($p \geq r$):**
  - This condition ensures that the subarray has at least one element (i.e., `p ≤ r`).
  - It prevents empty subarrays from being processed, which would lead to undefined behavior in the merge step.
- **Modified Condition ($p \neq r$):**
  - This condition still prevents empty subarrays from being processed, call has $n \geq 1$, the recursive calls will always divide the array into subarrays with at least one element.
  - This is because each division either splits a subarray into two non-empty subarrays or reaches the base case (a single-element subarray).

Why `p ≠ r` Suffices:

In the recursive calls, when `p ≠ r` is true:

- It implies that `p < r` or `p > r`.
- We know that `p > r` cannot occur due to the initial call constraint ($n \geq 1$) and the consistent division of subarrays.
- Therefore, the only valid scenario is `p < r`, which ensures that the subarray is non-empty and can be processed correctly.

Conclusion:

In conclusion, the modified condition `p ≠ r` suffices to prevent empty subarrays call to MERGE-SORT is made with a valid non-empty array ($n \geq 1$).

Q: 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 of Lines 12-18:

At the beginning of each iteration of the while loop on lines 12-18, the following conditions hold:

1. **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.
2. **Subarray L[i:nL-1]** contains the remaining elements of the original subarray **A[p:q]**, in sorted order.

3. **Subarray R[j:nR-1]** contains the remaining elements of the original subarray **A[q+1:r]**, in sorted order.

Proof of Correctness using the Loop Invariant:

1. **Initialization:**
   - Before the first iteration, k = p, i = 0, and j = 0.
   - A[p:k-1] is empty, which satisfies the first condition.
   - L[0:nL-1] contains A[p:q], and R[0:nR-1] contains A[q+1:r], satisfying the second and third conditions.
2. **Maintenance:**
   - During each iteration, the loop compares L[i] and R[j] and copies the smaller one to A[k].
   - This maintains the sorted order of A[p:k-1] and ensures that it contains the smallest k-p elements.
   - The loop increments either i or j, depending on which element was copied, effectively removing the copied element from either L or R.
   - This maintains the sorted order of L[i:nL-1] and R[j:nR-1], ensuring they contain the remaining elements of A[p:q] and A[q+1:r], respectively.
3. **Termination:**
   - The loop terminates when either i = nL or j = nR.
   - At this point, one of L or R is empty, and the other contains the remaining elements in sorted order.
   - The while loops on lines 20-23 and 24-27 copy the remaining elements from the non-empty array to A[k:r], completing the merging process.

Therefore, the loop invariant holds throughout the execution of the while loop on lines 12-18, and 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].

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

We will use mathematical induction to prove that when n ≥ 2 is an exact power of 2, the solution to the recurrence T(n) = 2T(n/2) + n is T(n) = n lg n.

Base Case:

For n = 2, we have:

T(2) = 2 (from the recurrence)

T(2) = 2 * lg(2) (since lg(2) = 1)

T(2) = 2

So, the base case holds.

Inductive Hypothesis:

Assume that the formula T(n) = n lg n holds for all n = 2^k, where k is an integer such that $2 \leq 2^k \leq n$.

Inductive Step:

We want to show that the formula also holds for n = 2^(k+1).

Using the recurrence relation:

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

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

Applying the inductive hypothesis to T(2^k):

T(2^(k+1)) = 2(2^k lg(2^k)) + 2^(k+1)

= 2^k * 2k + 2^(k+1)

= 2^(k+1) * k + 2^(k+1)

Factoring out 2^(k+1):

T(2^(k+1)) = 2^(k+1) * (k+1)

Since k+1 = lg(2^(k+1)), we can write:

T(2^(k+1)) = 2^(k+1) * lg(2^(k+1))

Thus, the formula T(n) = n lg n holds for n = 2^(k+1) as well.

Conclusion:

By the principle of mathematical induction, we have proven that the solution to the recurrence T(n) = 2T(n/2) + n is T(n) = n lg n for all $n \geq 2$ that are exact powers of 2.

Q: 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 Recursive Insertion Sort

```
function recursiveInsertionSort(A, n):
    if n <= 1:
        return  // Base case: An array of size 1 is already sorted
    // Sort the first n-1 elements
    recursiveInsertionSort(A, n-1)
     // Insert the last element A[n] into the sorted subarray A[1:n-1]
    key = A[n]
    i = n - 1
    // 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
    // Place the key in the correct position
    A[i + 1] = key
```

Explanation of the Pseudocode

1. **Base Case**: If the size of the array n is less than or equal to 1, the function returns immediately because the array is already sorted.
2. **Recursive Call**: The function recursively sorts the first n−1 elements.
3. **Insertion**: After sorting the first n−1 elements, it inserts the nth element (the last element of the array) into its correct position within the sorted portion.
4. **Shifting Elements**: The while loop shifts elements that are greater than the key to the right, making space for the key to be inserted at its correct position.

Recurrence for Worst-Case Running Time

In the worst case, where the array is sorted in reverse order, the algorithm must shift all previously sorted elements to insert the new element.

Let T(n) be the worst-case time complexity for sorting an array of size n. The recurrence relation can be expressed as:

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

- $T(n-1)$: the time taken to sort the first $n-1$n-1n−1 elements.
- $O(n)$: the time taken to insert the nth element into the sorted subarray (which requires potentially shifting $n-1$elements).

Solving the Recurrence

We can solve this recurrence using the expansion method or the Master Theorem:

1. **Expansion Method**:
    - $T(n)=T(n-1)+O(n)$
    - $=T(n-2)+O(n-1)+O(n)$
    - Continuing this pattern, we get:

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

    This simplifies to:

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

Conclusion

Thus, the worst-case running time of the recursive insertion sort is:

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

Q: 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 theta lg(n).

Pseudocode for Binary Search

We'll provide both iterative and recursive versions of binary search.

. Iterative Version

```
function iterativeBinarySearch(A, n, v):
    left = 0
    right = n - 1

    while left <= right:
        mid = left + (right - left) / 2

        if A[mid] == v:
            return mid  // Value found, return the index
        else if A[mid] < v:
            left = mid + 1  // Search in the right half
        else:
            right = mid - 1  // Search in the left half

    return -1  // Value not found
```

. Recursive Version

```
function recursiveBinarySearch(A, left, right, v):
    if left > right:
        return -1  // Value not found

    mid = left + (right - left) / 2

    if A[mid] == v:
        return mid  // Value found, return the index
    else if A[mid] < v:
```

return recursiveBinarySearch(A, mid + 1, right, v) // Search in the right half

else:

return recursiveBinarySearch(A, left, mid - 1, v) // Search in the left half

Explanation of the Pseudocode

1. **Iterative Version**:
   - The function takes an array A, the number of elements n, and the value v to search for.
   - It initializes two pointers, `left` and `right`, to denote the current search bounds.
   - In a loop, it calculates the midpoint `mid` and compares the value at that index with v.
   - Depending on the comparison, it adjusts the search bounds to either the left or right half of the current subarray.
   - The loop continues until the value is found or the search space is exhausted.
2. **Recursive Version**:
   - The function takes the array A, the current bounds (`left` and `right`), and the value v to search for.
   - If the `left` pointer exceeds the `right`, the function returns -1, indicating that the value is not found.
   - It computes the midpoint `mid`, compares it with v, and recursively searches in either the left or right half based on the comparison.

Worst-Case Running Time Analysis

To analyze the worst-case running time of binary search, we can observe the following:

1. **Reduction of Problem Size**:
   - In each iteration (or recursive call), the search space is halved. This means that after each comparison, we effectively reduce the size of the remaining array by approximately half.
2. **Recurrence Relation**:
   - The time complexity can be expressed as:

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

   where O(1)) accounts for the constant time spent on the comparison and index calculations.

3. **Solving the Recurrence**:
   - This recurrence can be solved using the **Master Theorem**.
   - In this case, a=1 (one subproblem), b=2 (the size is halved), and k=0 (since we have O(1)).
   - According to the Master Theorem:

- If f(n) = O(n^k) with k<$\log_b(a)$, the solution is T(n)=Θ(n^logb(a))
- Since $\log_2(1)=0$, we find that:

T(n)=Θ(n0)+O(1)=O(1)

- Therefore, the depth of the recursive calls is $\log_2(n)$, and the time complexity is:

T(n)=Θ(logn)

Conclusion

Thus, the worst-case running time of the binary search algorithm is:

Θ(logn

Q: 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 (nlogn)?

Using a binary search to find the correct insertion point in the sorted portion of the array in the **insertion sort** algorithm can improve the search time for the insertion step, but it does not change the overall worst-case running time of the insertion sort algorithm to Θ(nlogn

Explanation

1. **Current Insertion Sort**:
   - Insertion sort scans backward through the sorted portion of the array using a linear search, which takes O(j) time for each insertion.
   - The worst-case scenario occurs when the array is sorted in reverse order, leading to O(n^2) overall time complexity because for each of the n elements, it takes up to j comparisons in the worst case.
2. **Using Binary Search**:
   - If we implement a binary search to find the appropriate position to insert the element, the time complexity for this search becomes O(logj), where j is the current number of sorted elements.
   - However, even though binary search reduces the time taken to find the insertion point, we still need to shift elements to make space for the new element. Shifting the elements still takes O(n) in the worst case.
3. **Total Complexity with Binary Search**:
   - For each of the n elements, we use binary search, which is O(logj), but we still have to perform the element shifting.
   - The overall time complexity can be described as:

O(nlogn)+O(n)=O(n^2)

The shifting of elements remains a linear operation, and it dominates the overall time complexity.

Conclusion

Thus, while using binary search can optimize the insertion step, it does **not** improve the overall worst-case running time of insertion sort to Θ(nlogn). The shifting of elements still contributes to the O(n^2) time complexity in the worst case. Therefore, insertion sort remains Θ(n^2) overall, even with binary search.

Q: 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 theta (nlogn) time in the worst case.

To determine whether a set S of n integers contains two elements that sum to a given integer x in Θ(nlogn) time, you can follow these steps:

Algorithm: Two-Sum Using Sorting and Two-Pointer Technique

1. **Sort the Array**
   - First, sort the set S of integers. Sorting takes O(nlogn) time.
2. **Initialize Pointers**:
   - Set two pointers:
     - `left` at the beginning of the sorted array (index 0).
     - `right` at the end of the sorted array (index n−1).
3. **Two-Pointer Technique**:
   - While `left` is less than `right`:
     1. Calculate the current sum: current_sum=S[left]+S[right]
     2. If `current_sum` equals x, return `true` (indicating that the two elements have been found).
     3. If `current_sum` is less than x, increment the `left` pointer (to increase the sum).
     4. If `current_sum` is greater than x, decrement the `right` pointer (to decrease the sum).
4. **No Match Found**:
   - If the loop ends without finding a match, return `false`.

Pseudocode

```
function twoSum(S, x):
    sort(S)  // Sort the array S in O(n log n)

    left = 0
    right = length(S) - 1

    while left < right:
        current_sum = S[left] + S[right]

        if current_sum == x:
            return true  // Found the pair
        else if current_sum < x:
            left = left + 1  // Increase the sum
        else:
            right = right - 1  // Decrease the sum

    return false  // No pair found
```

Time Complexity Analysis

1. **Sorting**: The sorting step takes O(nlogn).
2. **Two-Pointer Search**: The while loop iterates at most n times, which takes O(n)

Overall Complexity

The overall time complexity of this algorithm is:

$O(n\log n)+O(n)=\Theta(n\log n)$

Conclusion

This algorithm efficiently determines whether there are two elements in the set S that sum to exactly x within the required time complexity of Θ(nlogn).

2-1 Insertion sort on small arrays in merge sort

Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

a. Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

b. Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

d. How should you choose k in practice?

A. Insertion sort has a worst-case time complexity of O(n^2). For each of the n/k sublists of length k, the worst-case time is O(k^2). Therefore, the total worst-case time for sorting all n/k sublists is (n/k) * O(k^2) = Θ(nk).

B. The standard merging mechanism used in merge sort has a worst-case time complexity of O(n log n), where n is the total number of elements being merged. In this case, we have n/k sublists, each of length k, so the total number of elements is n. Therefore, the worst-case time for merging all the sublists is O(n lg n).However, we can optimize this further by considering the number of sublists. We are merging n/k sublists, so the height of the merge tree is log(n/k). At each level of the merge tree, we merge n elements in O(n) time. Therefore, the total worst-case time for merging all the sublists is O(n lg(n/k)).

C. Standard merge sort has a worst-case time complexity of Θ(n lg n). We want to find the largest value of k such that:

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

Dividing both sides by n:

k + lg(n/k) = lg n

$\lg(k) + \lg(n) - \lg(k) = \lg n$

$\lg(k) = 0$

$k = 1$

Therefore, the largest value of k for which the modified algorithm has the same running time as standard merge sort is k = 1.

D. In practice, the optimal value of k depends on various factors, including the specific implementation, hardware, and input data distribution. However, some general guidelines can be followed:

Choose a small value of k, such as 10 or 20.

Experiment with different values of k to find the best performance for your specific use case.

Consider the overhead of switching between insertion sort and merge sort. If the overhead is significant, it may be better to stick with standard merge sort.

By carefully choosing the value of k, you can potentially improve the performance of merge sort for small input sizes.

CHAPTER 03

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

To modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3, we can follow these points:

1. **Comparison-Based Sorting**: Any comparison-based sorting algorithm requires at least Ω(nlogn) comparisons in the worst case, as determined by the number of possible permutations of n elements.
2. **Insertion Sort Mechanism**: Insertion sort places each unsorted element into its correct position among the already sorted elements. In the worst case (e.g., when the input is sorted in reverse order), every new element may need to be compared to all previously sorted elements.
3. **Total Comparisons**: The number of comparisons in the worst case is still given by:

   $C(n)=C(n-1)+(n-1)$

   This simplifies to $(n-1)n/2$, leading to O(n^2).

Conclusion

The worst-case running time of insertion sort is Θ(n^2) for any input size n, regardless of whether n is a multiple of 3 or not. The fundamental operations of insertion sort depend only on the ordering of elements, not on their specific count.

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

Selection Sort Time Complexity Analysis

Selection Sort Overview: Selection sort repeatedly selects the minimum element from the unsorted portion of the array and swaps it with the first unsorted element.

Time Complexity:

1. **Outer Loop**: Runs n times, where n is the number of elements.
2. **Inner Loop**: In the ith iteration, it scans n− i elements to find the minimum:
    - Total comparisons: (n−1)+(n−2)+…+1+0=(n−1)·n /2
    - This results in O(n^2) comparisons.

Conclusion

The running time of selection sort is Θ(n^2) in the worst case, as the number of comparisons dominates the overall time complexity, regardless of the input order.

Q: Suppose that α 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 α n largest values start in the α n positions. What additional restriction do you need to put on α ? What value of ¸ maximizes the number of times that the α n largest values must pass through each of the middle (1-2 α)n array positions?

To generalize the lower-bound argument for insertion sort when the αn largest values start in the last αn positions of the array, we'll examine how the placement of these largest elements affects the number of comparisons and swaps required by insertion sort. Here's how we can analyze it:

Setting Up the Problem

- We have an array of size n, and the largest αn values are positioned in the last αn elements.

- The remaining $(1-\alpha)n$ elements are the smallest values, located in the first $(1-\alpha)n$ positions.
- **Goal**: Determine how many comparisons are needed in the worst case to sort the array using insertion sort, given this configuration.

Insertion Sort Behavior with Partially Sorted Input

1. **Insertion Process**:
    - Insertion sort works by moving each element leftward until it finds its correct position in the sorted subarray.
    - Since the αn\alpha nαn largest elements are initially in the last αn\alpha nαn positions, each of these large elements needs to pass through the middle portion of the array (the positions from $(1-\alpha)n$ to αn) to reach their correct positions.
2. **Number of Comparisons**:
    - In the worst case, every element in the αn largest values will need to be compared against all elements in the middle $(1-2\alpha)n$ positions before reaching its final position.
    - Thus, the number of comparisons required for each of the αn\alpha nαn elements to pass through these $(1-2\alpha)n$ middle positions is roughly: $\alpha n \cdot (1-2\alpha)n$

      This term dominates the comparison count for the algorithm.

Restriction on α\alphaα

To ensure that αn and $(1-2\alpha)n$ are positive, we need:

$0<\alpha<1/2$

Maximizing Comparisons

The number of comparisons required is maximized when the product $\alpha(1-2\alpha)$ is maximized. To find this:

1. Take the derivative of $f(\alpha)=\alpha(1-2\alpha)$ with respect to α: $f'(\alpha)=1-4\alpha$
2. Set $f'(\alpha)= 0$ to find the critical point: $1-4\alpha=0 \Rightarrow \alpha=1/4$
3. When $\alpha=1/4$, the term $\alpha(1-2\alpha)$ is maximized.

Conclusion

- For the worst case in terms of comparisons, set $\alpha=1/4$.
- In this configuration, insertion sort's comparison count is maximized as each of the largest $\alpha n=n/4$ elements must pass through the middle $(1-2\alpha)n=n/2$ positions, resulting in a worst-case complexity of $\Theta(n^2)$.
