Assignment 1

Name Princess alishah

Roll number : 14666

Class: Bscs 3 A

Submitted to: Sir jamal abdul ahad

Date: 30 october 2024

Exercise 1.1

QUESTION : # 01

Q1.Describe your own real world example that requires sorting. Describe one that
Requires finding the shortest distance between two points

Answer:

1. **Sorting Example:**
    - In e-commerce, sorting plays a crucial role. Think of Amazon or any online store where items are sorted by price, popularity, or relevance. When a customer searches for a product, sorting helps show the most relevant or top-rated products first, making it easier for them to find what they want.
    - This sorting saves time and improves user experience, directly impacting sales.
2. **Finding Distance Example:**
    - GPS navigation systems use the distance formula to calculate the shortest route between two points on a map. For instance, if you're using Google Maps to travel from your house to a friend's place, it calculates the distance between the two points and gives you the quickest route. This is essential in logistics, where companies need to find the shortest or fastest paths to deliver goods efficiently.

Q2 Answer:

In real-world settings, efficiency goes beyond just speed. Key measures of efficiency include:

1. Cost-Effectiveness: Minimizing costs while maximizing output, crucial in manufacturing and business operations.
2. Energy Efficiency: Reducing energy consumption, especially in data centers, industries, or even home appliances.
3. Resource Utilization: Making optimal use of available resources (e.g., raw materials or staff) to reduce waste.
4. Accuracy: Ensuring precision in results, especially in tasks like medical diagnostics or financial forecasting.
5. Scalability: The ability to handle larger tasks or more users without compromising performance, which is vital for software and networks.

Q3 Answer:

Data Structure: Array

1. Strengths:
   - Fast access to elements using index positions, making it efficient for reading specific items.
   - Simple structure, easy to implement and understand.
   - Fixed size is useful when the data size is known in advance, like storing days of the week.

2. Limitations:
   - Fixed size makes it inflexible, meaning you can't resize it once it's full.
   - Inserting or deleting elements is inefficient, as it requires shifting other elements.
   - Not suitable for scenarios needing frequent size changes or complex relationships between data points.

Q4 Answer:

Similarities

- - Both problems involve finding paths between points and aim to optimize distance or cost.
- - Both are commonly solved using algorithms, like Dijkstra's algorithm for shortest paths and heuristic methods for the traveling salesperson problem.
-

Differences

- - In the shortest path problem, the goal is to find the shortest route from a starting point to a destination, often without visiting all points.
- - The traveling salesperson problem requires finding a minimum-cost route that visits all points exactly once and returns to the start, which is more complex and time-consuming due to more combinations.

Q5 Answer:

Real-World Problem

Delivery Route Optimization: Companies like Amazon or FedEx need to find the best route for delivery drivers to minimize travel time and fuel costs, ensuring packages arrive on time. Finding the exact best route for many destinations is crucial for efficiency but can be extremely complex.

Approximate Solution

Instead of calculating the exact shortest route for every delivery stop, companies use approximation algorithms like the nearest-neighbor method. This approach allows drivers to take an efficient route by choosing the closest next stop each time, which isn't the absolute best route but still significantly reduces time and fuel costs.

Q5 Answer:

Real-World Problem

Project Management: In project management, teams often need to allocate resources and set timelines. Sometimes, all the project requirements and resources are known at the start, allowing for a detailed plan. Other times, new requirements or changes emerge during the project, requiring adjustments to the plan as more information becomes available.

Approximate Solution

When all information is available, project managers can create a comprehensive timeline and resource allocation plan. When information arrives over time, they can use agile methodologies, allowing for iterative planning and adjustments based on the latest input. This approach helps teams remain flexible and responsive to changes while still moving forward with the project.

Exercise 1.2

Question 1:

Application Example

Recommendation Systems: Many online platforms, like Netflix or Amazon, use recommendation systems to suggest products or content to users based on their preferences and behaviors.

Algorithm Function

A common algorithm used in these systems is collaborative filtering. This algorithm analyzes user interactions and preferences, identifying patterns and similarities between users. For example, if User A likes certain movies that User B also likes, the system might recommend movies that User B has seen but User A has not. This algorithm helps enhance user experience by personalizing suggestions, increasing engagement and satisfaction with the platform

Question 2

To determine for which value of n insertion sort runs faster than merge sort, we need to compare the two time complexities.

Insertion Sort: 8^n

Merge Sort: $64n \log n$

We need to find n such that:

$8^n < 64n \log n$

First, we can simplify the comparison. Notice that 64 can be rewritten as 8^2:

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

This leads to:

$n < 2n \log n$

Dividing both sides by n (assuming $n > 0$):

$1 < 2 \log n$

Dividing by 2 gives:

$0.5 < \log n$

Now we can convert the logarithmic inequality back to an exponential form:

$n > 2^{0.5}$

Calculating this gives:

$n > \sqrt{2} \approx 1.41$

Since n must be an integer, we can start testing integer values.

1. For n = 1:
   - Insertion Sort: 8^1 = 8
   - Merge Sort: 64 * 1 * log(1) = 0
   - Insertion Sort is not faster.

2. For n = 2:
   - Insertion Sort: 8^2 = 64
   - Merge Sort: 64 * 2 * log(2) = 64
   - Insertion Sort is not faster.

3. For n = 3:

- Insertion Sort: 8^3 = 512
- Merge Sort: 64 * 3 * log(3) ≈ 64 * 3 * 1.585 ≈ 303.36
- Insertion Sort is not faster.

4. For n = 4:
- Insertion Sort: 8^4 = 4096
- Merge Sort: 64 * 4 * log(4) = 64 * 4 * 2 = 512
- Insertion Sort is faster.

Thus, insertion sort starts beating merge sort when n = 4.

Exercise 2.1

Question 1

Insertion Sort Operation on the Array [31, 41, 59, 26, 41, 58]

Initial Array:

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

Step 1:
- The first element (31) is considered sorted.
- Current Array:

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

Step 2:
- Consider the second element (41).
- It is already in the correct position.
- Current Array:

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

Step 3:
- Consider the third element (59).
- It is also in the correct position.
- Current Array:

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

Step 4:
- Consider the fourth element (26).
- Compare with 59, 41, and 31:
  - 26 < 59: shift 59 right.
  - 26 < 41: shift 41 right.
  - 26 < 31: shift 31 right.
- Insert 26 in the first position.
- Current Array:

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

Step 5:
- Consider the fifth element (41).
- Compare with 59:
  - 41 < 59: shift 59 right.
- Insert 41 in its correct position.
- Current Array:
\[
[26, 31, 41, 41, 59, 58]
\]

Step 6:
- Consider the sixth element (58).
- Compare with 59:
  - 58 < 59: shift 59 right.
- Insert 58 in its correct position.
- Final Sorted Array:
[26, 31, 41, 41, 58, 59]

Final Result:
The array is now sorted:

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

Question 2

Loop Invariant:At the start of each iteration of the loop, the variable `sum` contains the sum of the elements in the array A from index 1 to the current index `i`.

Initialization

- Before the first iteration (when `i = 1`), `sum` is initialized to 0.
- The invariant holds true because the sum of an empty set of numbers (i.e., no elements considered yet) is 0.

Maintenance

- Assume the loop invariant holds true at the beginning of the `i`th iteration, where `sum` contains the sum of elements A[1] to A[i].
- In the `i`th iteration, the procedure adds A[i] to `sum`.
- After this addition, `sum` now contains the sum of elements A[1] to A[i] plus A[i], which is the same as the sum of elements A[1] to A[i + 1].
- Thus, the loop invariant continues to hold after this iteration.

Termination

- The loop terminates when `i` exceeds `n`.
- At this point, the loop invariant states that `sum` contains the sum of elements from A[1] to A[n].
- Therefore, when the loop ends, `sum` will equal the total sum of all elements in the array A[1:n].

Conclusion

Since the loop invariant is initialized correctly, maintained throughout the loop, and leads to a correct conclusion upon termination, we can conclude that the sum procedure correctly computes the sum of the numbers in A[1:n].

Question 3:

Insertion Sort Procedure for Monotonically Decreasing Order

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

        # Move elements of arr[0..i-1] that are less than key
        while j >= 0 and arr[j] < key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

 Example usage
array = [31, 41, 59, 26, 41, 58]
insertion_sort_decreasing(array)
print(array)
```

Explanation:
1. The algorithm starts from the second element and iterates through the array.
2. For each element (key), it compares it with the elements in the sorted part of the array (to the left).
3. It shifts elements that are less than the key to the right.
4. Finally, it inserts the key in its correct position to maintain the order.

Exercise 2.2

Question 1

The function ({n^3}{1000} + 100n^2 - 100n + 3) can be expressed in terms of scientific notation (e notation) as follows:

f(n) = 1.0 \times 10^{-3} n^3 + 1.0 \times 10^{2} n^2 - 1.0 \times 10^{2} n + 3.0
\]

In e notation, this can be represented as:

\[
f(n) = 1.0e-3 \cdot n^3 + 1.0e2 \cdot n^2 - 1.0e2 \cdot n + 3.0
\]

This representation maintains clarity and consistency for pasting into a word processor.

Question 2:

Pseudocode for Selection Sort

```plaintext
function selection_sort(A):
    n = length(A)
    for i from 1 to n - 1 do:
        min_index = i
        for j from i + 1 to n do:
            if A[j] < A[min_index] then:
                min_index = j
        exchange A[i] and A[min_index]
```

Loop Invariant

Loop Invariant: At the start of each iteration of the outer loop, the subarray A[1:i] contains the smallest elements in sorted order.

xplanation of the Loop Invariant

Initialization Before the first iteration (when `i = 1`), the subarray A[1:1] is trivially sorted (contains one element).

Maintenance:If the invariant holds for the first `i` elements, after finding the minimum element in A[i:n] and swapping it with A[i], the first `i + 1` elements are now sorted.

Termination: When the loop terminates (after the last iteration when `i = n - 1`), the entire array A[1:n] is sorted.

Why Only the First n - 1 Elements?

The algorithm only needs to run for the first n - 1 elements because, after placing the smallest n - 1 elements in their correct positions, the last element (A[n]) is automatically in the correct position as it is the only remaining element.

Worst-Case Running Time

The worst-case running time of selection sort in big O notation is:

O(n^2)

Best-Case Running Time

The best-case running time of selection sort is also:

O(n^2)

This is because selection sort always scans the remaining elements to find the minimum, regardless of the initial order of the elements. Thus, the best case does not provide any performance improvement over the worst case.

Question 3:

Linear Search: Linear search checks each element of the array sequentially until it finds the target element or reaches the end of the array.

Average Case Analysis

- Average Case: When searching for an element that is equally likely to be any element in the array, if the array has `n` elements, the average number of elements checked can be calculated as follows:

  - If the element is in the array, on average, it will be found after checking half of the elements. This can be represented mathematically as:

  $$
  \text{Average checks} = \frac{1 + 2 + 3 + \ldots + n}{n} = \frac{n + 1}{2} \approx \frac{n}{2}
  $$

  - Therefore, on average, about $\frac{n}{2}$ elements need to be checked.

Worst Case Analysis

- Worst Case:The worst-case scenario occurs when the target element is either the last element in the array or not present at all. In both cases, all `n` elements must be checked. Thus:

  $$

$$
\text{Worst checks} = n
$$

Running Times in Big O Notation

- Average Case Running Time: The average-case running time of linear search, based on the average number of checks, is:

$$
O\left(\frac{n}{2}\right) = O(n)
$$

- Worst Case Running Time: The worst-case running time is:

$$
O(n)
$$

Justification

- In linear search, regardless of the position of the target element, each element is examined at most once.
- In the average case, we are considering that the element can be anywhere, leading to the average checks being about half of the total elements.
- The worst-case time complexity remains $O(n)$ because in the worst scenario, all elements are checked before arriving at a conclusion. Thus, both the average-case and worst-case running times of linear search are $O(n)$.

Question 4

1. Adaptive Algorithms

Use adaptive algorithms like **Insertion Sort**, which runs in O(n)O(n)O(n) for already sorted input.

2. Hybrid Approaches

Implement hybrid algorithms like **Timsort**, combining the strengths of different sorting methods to optimize performance on partially sorted data.

3. Early Exit Conditions

Add checks to terminate the algorithm early if no swaps are made, as seen in **Bubble Sort** or **Selection Sort**.

4. Parallelization

Leverage parallel processing in algorithms like **Parallel Merge Sort** for large datasets, particularly when they are partially sorted.

5. Specialized Sorts

Use **Bucket Sort** or **Radix Sort** for specific cases, achieving linear time under ideal conditions.

6. Smart Pivot Selection

Optimize pivot selection in **Quick Sort** to improve performance by reducing unnecessary comparisons.

These modifications can enhance the best-case running time and overall efficiency of sorting algorithms.

Exercise 2.3

Question 1

Step 1: Dividing the Array

1. Initial Array: [34, 41, 52, 26, 38, 57, 9, 49]

2. Divide:
   - Split the array into two halves:
   - Left: [34, 41, 52, 26]
   - Right: [38, 57, 9, 49]

3. Further Divide Left Half:
   - Left: [34, 41]
   - Right: [52, 26]

4. Further Divide Right Half:
   - Left: [38, 57]
   - Right: [9, 49]

5. Divide Until Single Elements:
   - Left: [34] and [41]
   - Left: [52] and [26]
   - Left: [38] and [57]

- Left: [9] and [49]

Step 2: Merging and Sorting

Now, we will merge the subarrays back together in sorted order.

1 Merge [34] and 41]:
- Merged: [34, 41]

2. Merge [52] and [26]:
- Merged: [26, 52]

3. Merge [38] and [57]:
- Merged: [38, 57]

4. Merge [9] and [49]:
- Merged: [9, 49]

Step 3: Continue Merging

Now, we merge the sorted pairs.

1. Merge [34, 41] and [26, 52]:
- Compare elements:
  - 26 < 34 → Place 26
  - 34 < 41 → Place 34
  - 41 < 52 → Place 41
  - Remaining 52
- Merged:[26, 34, 41, 52]
2. Merge [38, 57] and [9, 49]:
- Compare elements:
  - 9 < 38 → Place 9
  - 38 < 49 → Place 38
  - Remaining 57
- Merged: \[9, 38, 49, 57 \]

Final Merge

Finally, we merge the two larger sorted arrays:

1. Merge [26, 34, 41, 52] and [9, 38, 49, 57]:
- Compare elements:
  - 9 < 26 → Place 9
  - 26 < 34 → Place 26

- 34 < 38 → Place 34
- 38 < 41 → Place 38
- 41 < 49 → Place 41
- Remaining 49 and 52, then 57

- Merged: [9, 26, 34, 38, 41, 49, 52, 57]

Sorted Array

The final sorted array is:

[9, 26, 34, 38, 41, 49, 52, 57]

This illustrates how Merge Sort works by recursively dividing the array and merging the sorted subarrays back together.

Question 2:

To state a loop invariant for the while loop of the MERGE procedure, we first need to clarify what the MERGE procedure generally does. The MERGE procedure combines two sorted subarrays into a single sorted array. Let's assume the relevant lines are as follows:

- Lines 123-18: This is the main while loop where elements from the two subarrays are compared and merged.
- Lines 203-23 and 243-27: These are additional while loops that handle any remaining elements in either subarray after the main loop has completed.

Loop Invariant for the While Loop (Lines 123-18)

Loop Invariant:

At the beginning of each iteration of the while loop (lines 123-18), the following conditions hold:
1. The elements in the left subarray (L) that have been merged into the output array are in sorted order.
2. The elements in the right subarray (R) that have been merged into the output array are in sorted order.
3. All elements in the output array are less than or equal to all elements in the remaining unmerged portions of both subarrays.

Proof of Correctness Using the Loop Invariant

Initialization

Before the first iteration of the while loop:
- The output array is empty.
- No elements from either subarray have been merged yet.
- Therefore, the invariant holds as there are no merged elements to contradict the sorted order.

Maintenance

Assuming the loop invariant holds at the beginning of an iteration:

- When an element is selected from either the left subarray (L) or the right subarray (R), it is placed in the output array.
- Since L and R are sorted, the selected element is guaranteed to be the smallest among the remaining elements of both subarrays.
- The merged part of the output array remains sorted because we always add the smallest element next, thus maintaining the invariant.

Termination
The while loop terminates when all elements from either the left subarray or the right subarray have been processed:
- At this point, all elements from one subarray are fully merged into the output array.
- The remaining elements in the other subarray are still sorted and are simply appended to the output array.
- Therefore, the output array contains all elements from both subarrays, sorted in ascending order.

Conclusion
Since the loop invariant holds through initialization, maintenance, and termination, we conclude that the MERGE procedure correctly merges two sorted subarrays into a single sorted array. Thus, the MERGE procedure is correct based on the loop invariant established.

Question 4

To determine whether a given set S of n integers contains two elements that sum to a specified integer x, we can use a combination of sorting and the two-pointer technique. This approach will have a worst-case time complexity of $O(n \log n)$.

Algorithm Steps

1. **Sort the Array:**
   - First, sort the array S using an efficient sorting algorithm like Merge Sort or Quick Sort. Sorting will take $O(n \log n)$ time.

2. **Initialize Two Pointers:**
   - Set two pointers:
     - left at the start of the sorted array (index 0).
     - right at the end of the sorted array (index $n - 1$).

3. **Iterate and Check for Sum:**
   - While $\text{left} < \text{right}$:
     - Calculate the current sum as $\text{sum} = S[\text{left}] + S[\text{right}]$.
     - If sum equals x:
       - Return true (the pair is found).
     - If $\text{sum} < x$:

- Increment the left pointer (move to the next larger number).
- If $\text{sum} > x$:
- Decrement the right pointer (move to the next smaller number).

4. Return Result:
- If the loop exits without finding a pair, return false.

Pseudocode

```plaintext
function hasPairWithSum(S, n, x):
    sort(S)  // Sort the array S
    left = 0
    right = n - 1

    while left < right:
        sum = S[left] + S[right]
        if sum == x:
            return true
        else if sum < x:
            left = left + 1
        else:
            right = right - 1

    return false
```

Explanation

- Sorting the array takes $O(n \log n)$ time.
- The two-pointer technique iterates through the array at most n times, taking $O(n)$ time.
- Therefore, the overall time complexity is $O(n \log n)$ due to the sorting step, which satisfies the requirement.

Conclusion

This algorithm efficiently determines whether there are two elements in the set S that sum to the given integer x while maintaining a worst-case time complexity of $O(n \log n)$.

Exercise 3.1

Question 1

To modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily multiples of 3, we can follow a similar reasoning used in the original argument while adjusting it for arbitrary input sizes. The key idea is to establish a lower bound based on the structure of the input and the behavior of the algorithm.

Lower-Bound Argument for Insertion Sort

1. Understanding Insertion Sort:
   - Insertion sort works by building a sorted array incrementally. It takes each element from the unsorted part and inserts it into the correct position in the sorted part.
   - The worst-case time complexity of insertion sort is (O(n^2)) when the input is sorted in reverse order.

2. Analyzing Input Size:
   - Let (n) be the number of elements in the input array.
   - We want to establish a lower bound for any sorting algorithm, including insertion sort, when the input size (n) is not a multiple of 3.

3. Considering Arbitrary Sizes:
   - For any integer (n), we can express (n) in terms of a quotient and a remainder when divided by 3. Let (n = 3k + r \), where (r) can be 0, 1, or 2. Here, (k) is the number of complete sets of 3, and (r) is the leftover elements.
   - This means the input can be split into groups of 3 and a smaller group containing either 0, 1, or 2 elements.

4. Establishing the Lower Bound:
   - Insertion sort requires comparisons and potentially shifts for each element inserted into the sorted part.
   - The number of comparisons needed to sort (n) distinct elements is (Omega(n log n)) in the average case. In the worst case (reverse sorted), it can be as high as (n(n-1)/2).
   - In a scenario where elements need to be compared to maintain sorted order, even if we take groups of size up to 3, each element needs to be compared with others to ensure proper placement.

5. Generalizing the Argument:
   - Whether (n) is a multiple of 3 or not, the fundamental nature of comparison-based sorting still holds. Therefore, we can conclude:
   - The insertion sort, like any comparison-based sorting algorithm, must perform at least (Omega(n log n)) comparisons on average for any input size, including those not multiples of 3.

- The specific behavior of insertion sort, where each element may need to be compared against many others in the worst case, solidifies this lower bound.

Conclusion

The lower-bound argument for insertion sort remains valid regardless of whether the input size is a multiple of 3. The inherent properties of comparison-based sorting and the way insertion sort functions ensure that the time complexity reflects the necessary operations on any arbitrary input size (n). Thus, we can assert that insertion sort has a worst-case lower bound of (Omega(n^2)) and an average-case lower bound of (Omega(n log n)), independent of the size of (n).

Question 2

To analyze the running time of the selection sort algorithm without using mathematical terms, let's break down how the algorithm works and how that impacts its efficiency.

How Selection Sort Works

1. Process Overview:
   - Selection sort sorts an array by repeatedly finding the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element.
   - It continues this process until all elements are sorted.

2. Steps Involved:
   - First, the algorithm looks through the entire array to find the smallest element.
   - Once found, it swaps this smallest element with the first element of the array.
   - Then, it moves to the next position and repeats the process for the remaining unsorted elements.
   - This continues until the array is fully sorted.

Analyzing Running Time

1. **Finding the Smallest Element:**
   - In each pass through the array, the algorithm examines every element to find the smallest one.
   - This means it has to look at all the elements that have not been sorted yet.
   - In the first pass, it checks all elements, in the second pass, it checks all but one, and so on.

2. **Number of Passes:**

- The selection sort will make as many passes as there are elements in the array minus one.
- For example, if you have 10 elements, it will need to make 9 passes.

3. **Total Comparisons:**
- Since the algorithm checks each unsorted element in every pass, the number of comparisons adds up.
- In the first pass, it checks 10 elements, in the second pass 9 elements, and so on, until the last pass, where it checks just 1 element.

4. **Impact of Input Size:**
- As the size of the array increases, the number of comparisons grows significantly.
- This means that if you double the size of the array, the time taken to sort it can increase quite a bit because it checks so many more elements.

5. **Best and Worst Case:**
- Regardless of whether the input is already sorted or in reverse order, the selection sort will always perform the same number of comparisons. This is because it systematically checks all elements to find the smallest one each time.
- Therefore, the efficiency does not improve with better input arrangements; it will always take a similar amount of time.

Conclusion

In summary, selection sort is not the fastest sorting method because it has to go through the entire array multiple times, checking many elements in each pass. The number of comparisons increases as the number of elements grows, leading to longer running times. This consistent behavior across different input situations contributes to the perception that selection sort can be inefficient, especially for larger datasets.

Question 3

1. **Fraction of Largest Values**
- Imagine we have a set of values, and a portion of these values is sorted at the start of the list. This portion contains some of the largest values.
- The remaining values are mixed in with these largest values, and they are not necessarily sorted.

2. **Behavior of Insertion Sort**
   - Insertion sort processes the array by taking each value and placing it into its correct position among the already sorted values.
   - When a value is placed, it may need to compare with several other values to find its appropriate spot.

1. **Impact of Initial Arrangement**
   - Since a fraction of the largest values starts at the beginning, insertion sort will have to deal with moving these values into their proper places as the algorithm progresses.
   - As it processes each element, those large values will likely be compared with many smaller values that follow them.

2. **Movement Through the Array**
   - When a large value is inserted, it has to traverse through the positions that are initially filled with smaller values. This means it will often be pushed back through multiple positions.
   - The more large values that are present at the beginning, the more comparisons and movements will be required as the algorithm tries to correctly position them.

3. **Restriction on the Fraction**
   - For the argument to hold effectively, there needs to be a restriction on the fraction representing the portion of the largest values.
   - This restriction ensures that the number of large values does not exceed a certain threshold, as having too many could lead to a situation where insertion sort's efficiency significantly degrades.

1. **Optimal Fraction Value**
   - To maximize the number of times these largest values pass through the middle sections of the array, the fraction should be balanced.
   - If too few large values are at the start, their movement will be less, but if too many are at the beginning, it can lead to fewer movements as some may already be near their positions.

2. **Finding the Best Balance**
   - The ideal situation occurs when there is just enough of a mix so that as the algorithm processes the array, it maximizes the interaction between the largest values and the smaller ones.

- This creates a scenario where the largest values frequently have to navigate through a significant number of smaller values, leading to a high number of comparisons.

In summary, by considering a fraction of the largest values at the beginning of the input, we can see that insertion sort will require more effort to sort the array. The additional restriction on this fraction ensures that the algorithm still has to do considerable work without becoming overwhelmingly inefficient. The optimal value for this fraction maximizes the number of necessary comparisons, demonstrating the insertion sort's reliance on the arrangement of input values.

The end