UTTARANCHAL UNIVERSITY

(Established vide Uttaranchal University Act, 2012, Uttarakhand Act No. 11 of 2013)
Premnagar-248007, Dehradun, Uttarakhand, INDIA

ASSIGNMENT COVER PAGE

Name of Student:	VISHAL
Batch:	JULY-2023
Program:	BACHELOR OF COMPUTER APPLICATIONS
Subject & Code:	NUMERICAL ANALYSIS & OBCA 232
Semester:	3RD SEMESTER
Learner ID:	2313020052

NECESSARY INSTRUCTIONS

1. Cover Page must be filled in Capital Letters. All Fields of the Form are compulsory to be filled.
2. The assignment should be written / computer typed on A4 size paper and it should be neat and clearly readable.
3. The cover page should be stapled at the front of each and every assignment.
4. Incomplete Assignments will not be accepted.

Q 1. Given a binary search algorithm with floating-point numbers [1.23, 3.45, 5.67, 7.89, 9.01, 10.23, 12.34], analyze the impact of rounding errors in the computation. Propose a method to minimize these errors.

Analyzing Rounding Errors in Binary Search with Floating-Point Numbers

Binary search is a fundamental algorithm widely used to efficiently locate elements in a sorted list. However, when dealing with floating-point numbers, the algorithm may encounter challenges due to rounding errors. These errors arise because computers approximate floating-point numbers rather than storing them precisely. In this essay, we will explore the causes and impacts of these rounding errors in binary search and propose practical methods to minimize them.

Understanding Rounding Errors

Floating-point numbers are a way of representing real numbers in a computer. They are stored using a finite number of bits based on the IEEE 754 standard, which means not all real numbers can be represented exactly. This limitation can result in **rounding errors**, particularly during arithmetic operations like addition, subtraction, and division.

For instance, when calculating the mid-point in binary search, a rounding error might occur because the result of dividing by 2 is not always an exact value in binary representation. Over time, even small errors can accumulate, potentially affecting the accuracy of the algorithm.

How Binary Search Works

Binary search is an efficient algorithm that operates by repeatedly dividing the search range into halves. Consider the example list `[1.23, 3.45, 5.67, 7.89, 9.01, 10.23, 12.34]`. To find a specific value, the algorithm follows these steps:

1. Start with two pointers: `low` (pointing to the first index) and `high` (pointing to the last index).

Compute the mid-point using the formula:
mid = low + (high - low) / 2

2. Compare the value at `mid` with the target.
3. Adjust the `low` or `high` pointer based on the comparison.
4. Repeat until the target is found or the range becomes invalid.

In the context of floating-point numbers, errors can occur during mid-point calculation or comparison due to the imprecision of these numbers.

Effects of Rounding Errors

1. **Incorrect Mid-Point Calculation**:
   Rounding errors may result in slightly incorrect mid-point values. If the computed index doesn't align perfectly with the actual data, the algorithm might choose the wrong range for the next iteration.

2. **Comparison Challenges**:
   Floating-point numbers may not match exactly during comparisons. For example, even if the value `7.89` exists in the list, rounding errors might make a direct equality check fail.

3. **Increased Iterations**:
   Small inaccuracies can cause the algorithm to perform additional iterations, reducing its efficiency. In extreme cases, it might fail to find the correct value altogether.

Strategies to Minimize Rounding Errors

1. Use Integer Indices

To avoid errors during mid-point calculation, perform all index computations using integers. For example:

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

This ensures that the mid-point calculation is precise, as integers do not suffer from rounding issues. Comparisons can then be performed on the floating-point values at the calculated indices.

2. Implement a Tolerance for Comparison

Instead of relying on exact equality, introduce a small margin of error (epsilon) when comparing floating-point values. For example:

```
if (abs(array[mid] - target) < epsilon) {
   // Consider it a match
}
```

Here, `epsilon` is a tiny value like `0.000001`, which accounts for minor differences in representation.

3. Normalize the Data

Scale the floating-point numbers to integers by multiplying them with a constant factor (e.g., 1000). Perform the binary search on these scaled values, reducing the impact of floating-point imprecision.

4. Use Higher Precision

Using higher-precision data types, such as `double` or `long double`, reduces the likelihood of rounding errors. These data types provide more bits for storing fractional values, making calculations more accurate.

5. Avoid Repeated Operations

Repeated arithmetic operations can accumulate rounding errors. Minimize these by consolidating calculations into fewer steps and avoiding unnecessary recalculations.

Effectiveness of These Solutions

These strategies collectively enhance the robustness of binary search when working with floating-point numbers. Using integer indices ensures the core logic remains precise, while tolerance-based comparisons and data normalization accommodate the inherent limitations of floating-point arithmetic. Employing higher-precision data types provides an additional safeguard, particularly in applications requiring high accuracy.

Conclusion

While rounding errors are an inherent limitation of floating-point arithmetic, their impact on binary search can be effectively managed. By carefully choosing how computations are performed and incorporating strategies like integer-based indexing, tolerance for comparisons, and higher precision, developers can significantly reduce errors and maintain the algorithm's efficiency and accuracy. This not only ensures reliable performance but also underscores the importance of understanding data representation in computer systems.