Editorial-Approach-W3A2: Exploring Linear & Binary Search, Sorting Methods, and Complexity Trade-offs

---

Question 1

Which of the following best describes an algorithm?

- **A)** A programming language
- **B)** A step-by-step procedure for solving a problem
- **C)** A type of data structure
- **D)** A mathematical equation

Approach:

1. **Recall definition**: Think about what an algorithm fundamentally is.
2. **Differentiate**: Compare it to programming languages, data structures, and math equations—these are different concepts.
3. **Identify the universal property**: An algorithm must be a systematic, stepwise method to solve a given problem.

Question 2

What is the time complexity of linear search in the worst case?

- **A)** O(1)
- **B)** O(log n)
- **C)** O(n)
- **D)** O(n^2)

Approach:

1. **Recall how linear search works**: It checks each element one by one.
2. **Worst case scenario**: Consider if the target element is at the end or not in the list at all—how many elements do you end up examining?
3. **Match with Big-O notation**: Decide which complexity fits the scenario of scanning the entire list.

Question 3

Which of the following is TRUE about selection sort?

- **A)** It's the fastest sorting algorithm for all input sizes

- **B)** It has a best-case time complexity of O(n)
- **C)** It's an in-place sorting algorithm
- **D)** It's a stable sorting algorithm

Approach:

1. **Recall selection sort's characteristic**: It selects the smallest (or largest) element and places it in its correct position each pass.
2. **Consider memory usage**: Check if it needs extra space proportional to the array size.
3. **Check definitions**: Distinguish between stability, best-case time complexity, and typical performance for selection sort.

Question 4

What is the primary advantage of binary search over linear search?

- **A)** It works on unsorted lists
- **B)** It has a worst-case time complexity of O(log n)
- **C)** It's easier to implement
- **D)** It uses less memory

Approach:

1. **Contrast approaches**: Linear search checks elements one by one, binary search divides the list repeatedly.
2. **Efficiency factor**: Consider how many comparisons each method might need for large lists.
3. **Precondition**: Remember if binary search requires the data to be sorted or not.

Question 5

In bubble sort, after the first pass, which element is guaranteed to be in its correct position?

- **A)** The smallest element
- **B)** No element is guaranteed
- **C)** The middle element
- **D)** The largest element

Approach:

1. **Recall bubble sort mechanism**: In each pass, pairs of adjacent elements are compared and swapped if out of order.
2. **Visualize the first pass**: Notice which element "bubbles" to its final position.
3. **Identify the final position after one complete pass**: Either the largest or smallest ends up fixed, depending on the implementation (ascending or descending order).

Question 6

What is the space complexity of linear search?

- **A)** O(1)
- **B)** O(log n)
- **C)** O(n)
- **D)** O(n^2)

Approach:

1. **Consider memory usage**: During a linear search, how many extra data structures are created or how much additional space is used beyond a few variables?
2. **Big-O space complexity**: Compare that memory usage to the size of the list.

Question 7

Which of the following is NOT a characteristic of selection sort?

- **A)** It divides the input into a sorted and an unsorted region
- **B)** It repeatedly selects the smallest element from the unsorted region
- **C)** It has a time complexity of O(n log n) in all cases

Approach:

1. **Recall selection sort steps**: Sort region grows one element at a time by selecting an extreme (smallest/largest).
2. **Compare known complexities**: What is selection sort's typical time complexity for best, average, and worst cases?
3. **Spot the incorrect statement**: Identify which claim contradicts the known behavior or performance of selection sort.

Question 8

In selection sort, after the first pass, which element is guaranteed to be in its correct position?

- **A)** The largest element
- **B)** The smallest element
- **C)** The middle element
- **D)** No element is guaranteed

Approach:

1. **Recall the selection sort process**: On each pass, it finds the minimum (or maximum) from the unsorted segment and places it in the sorted segment.
2. **Focus on the first pass**: Which element moves into its final position after identifying the extremum in the entire list?
3. **Distinguish from bubble sort**: Remember that bubble sort "bubbles up" the largest element, while selection sort "selects" the smallest (in a typical ascending implementation).

Question 9

What is the primary advantage of selection sort over bubble sort?

- **A)** Selection sort is stable
- **B)** Selection sort has a better average-case time complexity
- **C)** Selection sort performs fewer swaps
- **D)** Selection sort works better on partially sorted arrays

Approach:

1. **Compare the mechanics**: Bubble sort may swap on nearly every comparison, while selection sort specifically searches for a minimal element and swaps only once per pass.
2. **Count swaps**: Think about how many swaps each algorithm performs in the worst case.
3. **Stability vs. swaps**: Recognize which algorithm is stable and which one has fewer swaps.

Question 10

What is the primary difference between linear search and binary search?

- **A)** Linear search can only be used on sorted lists

- **B)** Binary search can be used on unsorted lists
- **C)** Linear search examines every element, while binary search eliminates half the remaining elements in each step
- **D)** Binary search is always faster than linear search, regardless of input size

Approach:

1. **Recall how linear vs. binary search operates**:
    - Linear: sequential check of each element.
    - Binary: repeated halving of a **sorted** data set.
2. **Identify the fundamental distinction** in how they narrow down the search space.
3. **Check assumptions**: Consider the necessity of sorted data for binary search.

Question 11

What is the best-case time complexity of bubble sort?

- **A)** O(1)
- **B)** O(n)
- **C)** O(n log n)
- **D)** O(n^2)

Approach:

1. **Think about bubble sort in the best case**: The best case typically occurs if the array is already sorted.
2. **Analyze the steps**: Even if it's sorted, bubble sort still does at least one pass of comparisons.
3. **Decide the Big-O**: Determine how many comparisons happen in that best-case scenario and how it translates to time complexity.

Case Study: Optimizing Search and Sort Operations at TechMart

TechMart, a rapidly growing e-commerce company specializing in electronics, is facing challenges with its search and sort operations. As a newly hired software engineer, you've been tasked with improving their existing algorithms to enhance efficiency and customer satisfaction.

Question 12

You're implementing a search function for TechMart's customer support team to find customer orders. Which of the following Python functions correctly implements a linear search for order IDs? Linear Search should return the position.

A)

```
def search_order(order_list, target_id):
    for i in range(len(order_list)):
        if order_list[i] == target_id:
            return i
    return -1
```

B)

```
def search_order(order_list, target_id):
    for order in order_list:
        if order == target_id:
            return True
    return False
```

C)

```
def search_order(order_list, target_id):
    for order in order_list:
        if order == target_id:
            return order
    return -1
```

D)

```
def search_order(order_list, target_id):
    return target_id in order_list
```

Approach:

1. **Check correctness of a "linear search"**: A correct linear search typically iterates over the list and compares each element to target_id.
2. **Identify required return value**: We need the **index** where the target is found (or -1 if not found).
3. **Inspect each option**:
    - Do they return the index or just a boolean?

- Do they return the order itself or -1?

Question 13

In the context of sorting product lists, what does it mean for an algorithm to be "in-place"?

- **A)** The algorithm sorts the products without using any extra space
- **B)** The algorithm maintains the relative order of products with equal prices
- **C)** The algorithm works only on arrays of products, not on linked lists
- **D)** The algorithm has a time complexity of O(n log n) for sorting products

Approach:

1. **Definition check**: An "in-place" algorithm manipulates data directly in the original array, using constant additional space.
2. **Compare each option**: Which option specifically mentions extra space usage versus stability or complexity?
3. **Recall formal definition**: Confirm that "in-place" focuses on minimal auxiliary space requirements.

Question 14

TechMart wants to implement binary search for their product catalog. What prerequisite must be met before they can use this algorithm?

- **A)** The product catalog must be sorted
- **B)** The catalog must have an odd number of products
- **C)** The catalog must be stored in contiguous memory locations

Approach:

1. **Binary search fundamentals**: Repeatedly comparing the middle element with the target requires a certain property of the data.
2. **Check each listed condition**: Is the number of products relevant? Is contiguity always required at a conceptual level?
3. **Most crucial requirement**: Recall that binary search depends on a specific ordering to cut the search space in half correctly.

Question 15

Which of the following Python functions correctly implements binary search for a sorted product catalog?

A)

```
def binary_search_product(catalog, target_id):
    left, right = 0, len(catalog) - 1
    while left <= right:
        mid = (left + right) // 2
        if catalog[mid] == target_id:
            return mid
        elif catalog[mid] < target_id:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

B)

```
def binary_search_product(catalog, target_id):
    return catalog.index(target_id)
```

C)

```
def binary_search_product(catalog, target_id):
    for i in range(len(catalog)):
        if catalog[i] == target_id:
            return i
    return -1
```

D)

```
def binary_search_product(catalog, target_id):
    return target_id in catalog
```

Approach:

1. **Recall how binary search operates**: Use left and right pointers; repeatedly calculate the mid.
2. **Check each function**:
    - Does it manually track left, right, and mid?

- Does it just do a linear scan or use a built-in function?

3. **Assess correctness**: Confirm whether the search range is updated appropriately and a comparison is made with the middle element to halve the search space.