Quiz

- In a binary search, what is the maximum number of comparisons needed to find a target value in a sorted array of size 16?

A. 4
B. 8
C. 16
D. 32

- Hint: Think about how binary search divides the array into halves and the number of times this division happens.

Analyzing Merge Sort Time Complexity

- Base Case:
  - `if (left < right) {`
  - Time: O(1).
  - It's a single comparison that runs in constant time.
- Calculate Midpoint:
  - `int mid = left + (right - left) / 2;`
  - Time: O(1).
  - Arithmetic operations and assignment take constant time.

Analyzing Merge Sort Time Complexity

- Recursive Calls:
  - `merge_sort(prices, left, mid);`
  - `merge_sort(prices, mid + 1, right);`
  - Time: This is where the recursion happens, dividing the array into smaller subarrays. Each recursive call breaks the array into two halves until size 1.
  - Total time: O(log n) recursion depth.
- Merging:
  - `merge(prices, left, mid, right);`
  - Time: O(n).Merging two halves involves comparing all elements in the subarrays, which takes linear time relative to the number of elements in the range [left, right].

Analyzing Merge Sort Time Complexity

- Temporary Arrays (Copying):

```
vector<double> L(n1), R(n2);
for (int i = 0; i < n1; i++) L[i] = prices[left + i];
for (int j = 0; j < n2; j++) R[j] = prices[mid + 1 + j];
```

  - Time: O(n1 + n2) ≈ O(n).
  - Both loops iterate through the left and right subarrays to copy elements.

Analyzing Merge Sort Time Complexity

- Merge While Loop:

```
while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            prices[k] = L[i];
            i++;
        } else {
            prices[k] = R[j];
            j++;
        }
        k++;
    }
```

  - Time: O(n).
  - Each element in L and R is compared and copied into the main array.

Analyzing Merge Sort Time Complexity

- Remaining Elements in L or R:

```
while (i < n1) {
        prices[k] = L[i];
        i++;
        k++;
    }

while (j < n2) {
        prices[k] = R[j];
        j++;
        k++;
   }
```

  - Time: O(n).
  - After one subarray is exhausted, the remaining elements in the other subarray are copied into prices.

Analyzing Merge Sort Time Complexity

- C++ Overall Analysis:
  - Lines like if (left < right) or calculating mid take constant time.
  - Recursion (merge_sort) divides the array in O(log n) steps.
  - Merging at each recursion level processes all n elements → O(n).
- Overall time complexity: O(n log n).

Merge sort Space Complexity analysis

- C++ Implementation:
  - Uses temporary arrays (L and R) for merging, requiring O(n) extra space.
- Python Implementation:
  - Slicing (prices[:mid] and prices[mid:]) creates new subarrays, also requiring O(n) space.
- Overall Space Complexity: O(n).
- C++ is more memory-efficient and faster because it avoids creating new arrays during recursion (uses indices instead).
- Python’s slicing adds an extra overhead, but the time complexity remains the same because the constant factors don’t change the overall growth rate.