Sure, let's document the algorithms, provide pseudo-code, analyze their time complexity, and compare them.

Recursive Algorithm

Pseudo-code:

```
function countTallestCandlesRecursive(candles, index, max_height):
    if index < 0:
        return 0

    count = countTallestCandlesRecursive(candles, index - 1, max_height)
    if candles[index] > max_height OR candles[index] == max_height:
        return count + 1
    else:
        return count

function birthdayCakeCandlesRecursive(candles_count, candles):
    max_height = 0
    for each candle in candles:
        if candle > max_height:
            max_height = candle

    return countTallestCandlesRecursive(candles, candles_count - 1, max_height)
```

Analysis:

- The recursive algorithm traverses the array once to find the maximum height.

- Then, it recursively counts the tallest candles while traversing the array again.

- Time Complexity: O(n), where n is the number of candles.

Non-Recursive (Iterative) Algorithm

Pseudo-code:

```
function birthdayCakeCandlesIterative(candles_count, candles):
    max_height = 0
    count = 0

    for each candle in candles:
        if candle > max_height:
            max_height = candle
            count = 1
        else if candle == max_height:
            count++

    return count
```

Analysis:

- The non-recursive algorithm iterates through the array once to find the maximum height and count the tallest candles simultaneously.
- Time Complexity: O(n), where n is the number of candles.

Divide and Conquer Algorithm

Pseudo-code:

```
```

```
function merge(arr, left, mid, right, max_height):
    // Merge step of Merge Sort modified to count tallest candles
    // Returns count of tallest candles during merge
    i = left
    j = mid + 1
    k = left
    count = 0

    while i <= mid AND j <= right:
        if arr[i] > arr[j]:
            arr[k++] = arr[i++]
            if arr[i - 1] == max_height:
                count++
        else:
            arr[k++] = arr[j++]
            if arr[j - 1] == max_height:
                count++

    // Copy remaining elements of left subarray
    while i <= mid:
        arr[k++] = arr[i++]
        if arr[i - 1] == max_height:
            count++

    // Copy remaining elements of right subarray
    while j <= right:
        arr[k++] = arr[j++]
        if arr[j - 1] == max_height:
            count++
```

```
    return count

function mergeSort(arr, left, right, max_height):
    // Merge sort with counting tallest candles
    if left < right:
        mid = left + (right - left) / 2
        count = 0

        // Count tallest candles in left and right halves recursively
        count += mergeSort(arr, left, mid, max_height)
        count += mergeSort(arr, mid + 1, right, max_height)

        // Merge two sorted halves and count tallest candles
        count += merge(arr, left, mid, right, max_height)

        return count
    else:
        return 0

function birthdayCakeCandlesDivideConquer(candles_count, candles):
    max_height = 0
    for each candle in candles:
        if candle > max_height:
            max_height = candle

    // Use Divide and Conquer (Merge Sort) to count tallest candles
    return mergeSort(candles, 0, candles_count - 1, max_height)
```

Analysis:

- The divide and conquer algorithm uses a modified version of the Merge Sort algorithm.

- It recursively divides the array into smaller subarrays until each subarray contains only one element.

- Then, it merges the subarrays while counting the tallest candles in the process.

- Time Complexity: $O(n \log n)$, where n is the number of candles.

Comparison:

- The recursive and non-recursive algorithms have a time complexity of $O(n)$, which is linear and efficient.

- The divide and conquer algorithm has a time complexity of $O(n \log n)$, which is slightly less efficient than the linear algorithms but still efficient for most practical purposes.

- For small input sizes, all algorithms perform similarly. However, as the input size grows, the divide and conquer algorithm may become slightly slower due to its higher time complexity compared to the linear algorithms. However, the difference may not be significant unless dealing with very large input sizes.