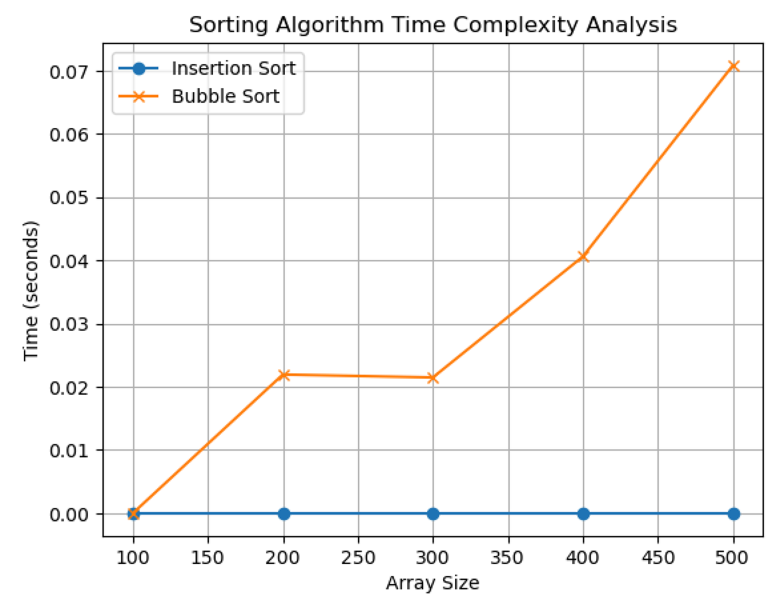
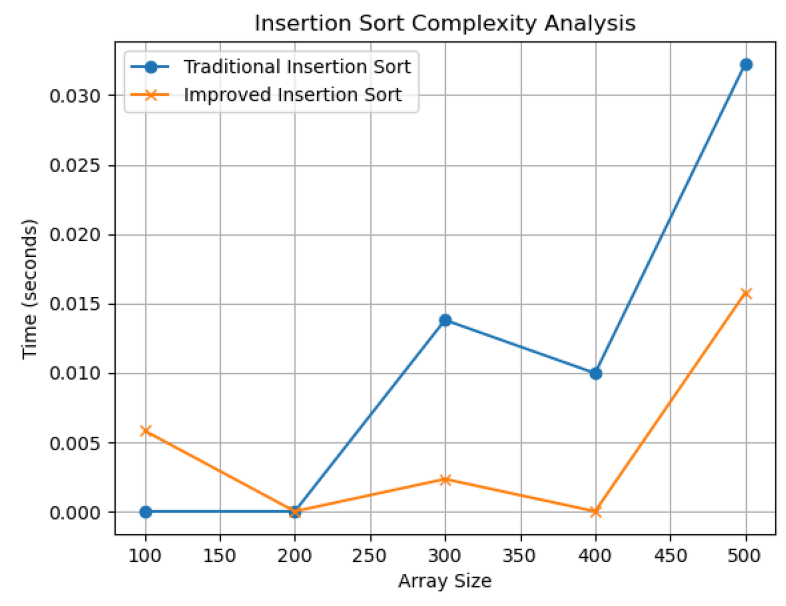
In [1]:

```python
import numpy as np
import matplotlib.pyplot as plt
import time

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

def generate_k_nearly_sorted_array(n, k):
    arr = np.arange(n)
    for i in range(k):
        idx1, idx2 = np.random.randint(0, n), np.random.randint(0, n)
        arr[idx1], arr[idx2] = arr[idx2], arr[idx1]
    return arr

sizes = [100, 200, 300, 400, 500]
insertion_times = []
bubble_times = []

for size in sizes:
    arr = generate_k_nearly_sorted_array(size, 10)  # k = 10
    start_time = time.time()
    insertion_sort(arr.copy())
    insertion_times.append(time.time() - start_time)

    arr = generate_k_nearly_sorted_array(size, 10)
    start_time = time.time()
    bubble_sort(arr.copy())
    bubble_times.append(time.time() - start_time)

# Plotting the results
plt.plot(sizes, insertion_times, label='Insertion Sort', marker='o')
plt.plot(sizes, bubble_times, label='Bubble Sort', marker='x')
plt.xlabel('Array Size')
plt.ylabel('Time (seconds)')
plt.title('Sorting Algorithm Time Complexity Analysis')
plt.legend()
plt.grid()
plt.show()
```



In [3]:

```python
import numpy as np
import matplotlib.pyplot as plt
import time

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

def improved_insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        left, right = 0, i - 1
        while left <= right:
            mid = left + (right - left) // 2
            if arr[mid] < key:
                left = mid + 1
            else:
                right = mid - 1
        for j in range(i, left, -1):
            arr[j] = arr[j - 1]
        arr[left] = key
    return arr

def generate_reverse_sorted_array(n):
    return list(range(n, 0, -1))

sizes = [100, 200, 300, 400, 500]
traditional_times = []
improved_times = []

for size in sizes:
    arr = generate_reverse_sorted_array(size)
    start_time = time.time()
    insertion_sort(arr.copy())
    traditional_times.append(time.time() - start_time)

    arr = generate_reverse_sorted_array(size)
    start_time = time.time()
    improved_insertion_sort(arr.copy())
    improved_times.append(time.time() - start_time)

plt.plot(sizes, traditional_times, label='Traditional Insertion Sort', marker='o')
plt.plot(sizes, improved_times, label='Improved Insertion Sort', marker='x')
plt.xlabel('Array Size')
plt.ylabel('Time (seconds)')
plt.title('Insertion Sort Complexity Analysis')
plt.legend()
plt.grid()
plt.show()
```



In [7]:

```python
import heapq

def merge_sorted_lists(lists):
    min_heap = []

    for i in range(len(lists)):
        if lists[i]:
            heapq.heappush(min_heap, (lists[i][0], i, 0))
    sorted_output = []

    while min_heap:
        value, list_index, element_index = heapq.heappop(min_heap)
        sorted_output.append(value)  # Add it to the output

        if element_index + 1 < len(lists[list_index]):
            next_value = lists[list_index][element_index + 1]
            heapq.heappush(min_heap, (next_value, list_index, element_index + 1))

    return sorted_output

sorted_lists = [
    [19, 22, 36, 43],
    [15, 25, 35, 45],
    [20, 22, 36, 48],
    [32, 33, 39, 50],
    [10, 18, 22, 28]
]

result = merge_sorted_lists(sorted_lists)
print(result)
```

```
[10, 15, 18, 19, 20, 22, 22, 22, 25, 28, 32, 33, 35, 36, 36, 39, 43, 45, 48, 50]
```

In [9]:

```python
import heapq

def find_k_largest_elements(arr, k):
    min_heap = []

    for num in arr:
        if len(min_heap) < k:
            heapq.heappush(min_heap, num)
        elif num > min_heap[0]:
            heapq.heappop(min_heap)
            heapq.heappush(min_heap, num)

    k_largest = list(min_heap)
    k_largest.sort(reverse=True)

    return k_largest


arr = [11, 23, 42, 91, 30, 22, 50]
k = 3
result = find_k_largest_elements(arr, k)
print(result)
```

```
[91, 50, 42]
```

In [11]:

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x > pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x < pivot]
    return quicksort(left) + middle + quicksort(right)

def find_k_largest_elements(arr, k):
    sorted_arr = quicksort(arr)
    return sorted_arr[:k]

arr = [11, 23, 42, 91, 30, 22, 50]
k = 3
result = find_k_largest_elements(arr, k)
print(result)
```

```
[91, 50, 42]
```

In [13]:

```python
def activity_selection(activities):
    activities.sort(key=lambda x: x[1])

    selected_activities = [activities[0]]
    last_finish_time = activities[0][1]

    for i in range(1, len(activities)):
        if activities[i][0] >= last_finish_time:
            selected_activities.append(activities[i])
            last_finish_time = activities[i][1]

    return selected_activities

activities = [(1, 3), (2, 4), (3, 5), (0, 6), (5, 7), (8, 9), (5, 9)]
print("The maximum number of activities that can be performed are:", len(activity_selection(activities)))
```

```
The maximum number of activities that can be performed are: 4
```

In [19]:

```python
def find_pair_with_sum(arr, target_sum):
    seen = set()
    for num in arr:
        complement = target_sum - num
        if complement in seen:
            return (complement, num)
        seen.add(num)
    return None

arr = [8, 7, 3, 2, 1, 5]
target_sum = 10
result = find_pair_with_sum(arr, target_sum)
print(result)
```

```
(7, 3)
```

In [ ]:

In [17]:

```python
import math
import heapq

def introsort(arr):
    max_depth = 2 * math.log2(len(arr))
    _introsort(arr, 0, len(arr) - 1, int(max_depth))

def _introsort(arr, start, end, max_depth):
    if start >= end:
        return

    if max_depth == 0:
        heapq_sort(arr, start, end)
    else:
        pivot = partition(arr, start, end)
        _introsort(arr, start, pivot - 1, max_depth - 1)
        _introsort(arr, pivot + 1, end, max_depth - 1)

def partition(arr, start, end):
    pivot = arr[end]
    i = start
    for j in range(start, end):
        if arr[j] <= pivot:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
    arr[i], arr[end] = arr[end], arr[i]
    return i

def heapq_sort(arr, start, end):

    heap = arr[start:end+1]
    heapq.heapify(heap)
    for i in range(start, end + 1):
        arr[i] = heapq.heappop(heap)

# Example usage:
arr = [10, 3, 76, 34, 23, 32]
```

```python
introsort(arr)
print("Sorted array:", arr)
```

```
Sorted array: [3, 10, 23, 32, 34, 76]
```

In [ ]: