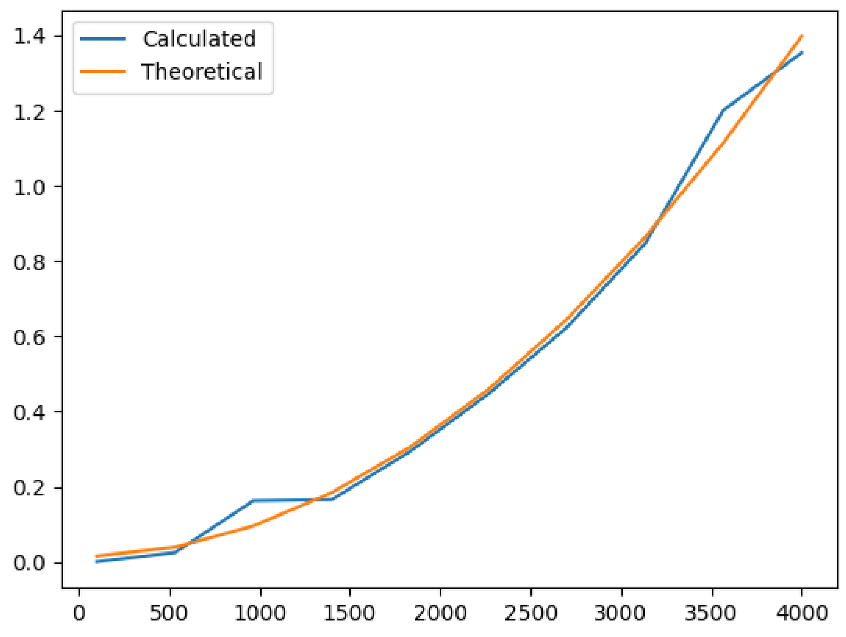
Below are 9 sorting algorithms, along with a short explanation to them, my views on their time complexities (comparatively), their codes, and their graphs.

Insertion Sort

Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is efficient for small lists but is generally slower than more advanced algorithms such as Quick Sort or Merge Sort for large lists. The time complexity of Insertion Sort is O(n^2) in the worst-case scenario.

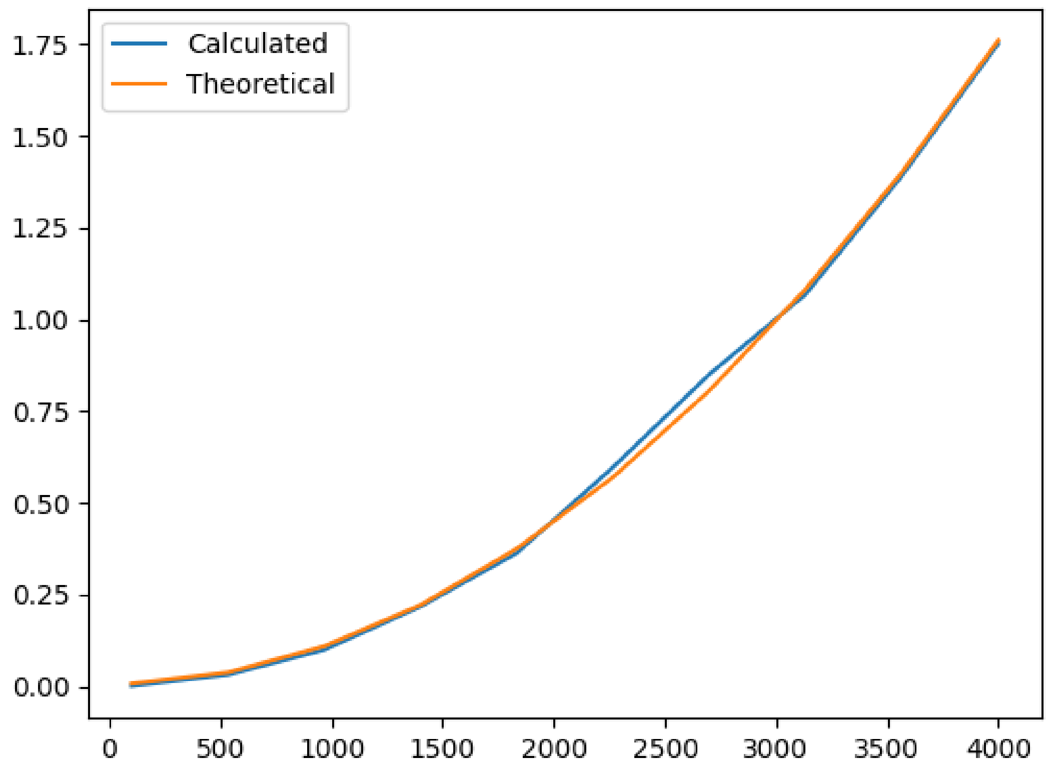
```python
# This function sorts an array using the insertion sort algorithm
def insertion_sort(lst):

    for step in range(1, len(lst)):
        key = lst[step]
        j = step - 1

        # move the elements of the sorted sub-list to create space for the key
element
        while j >= 0 and key < lst[j]:
            lst[j + 1] = lst[j]
            j = j - 1

        # insert the key element into the sorted sub-list
        lst[j + 1] = key
```



Selection Sort

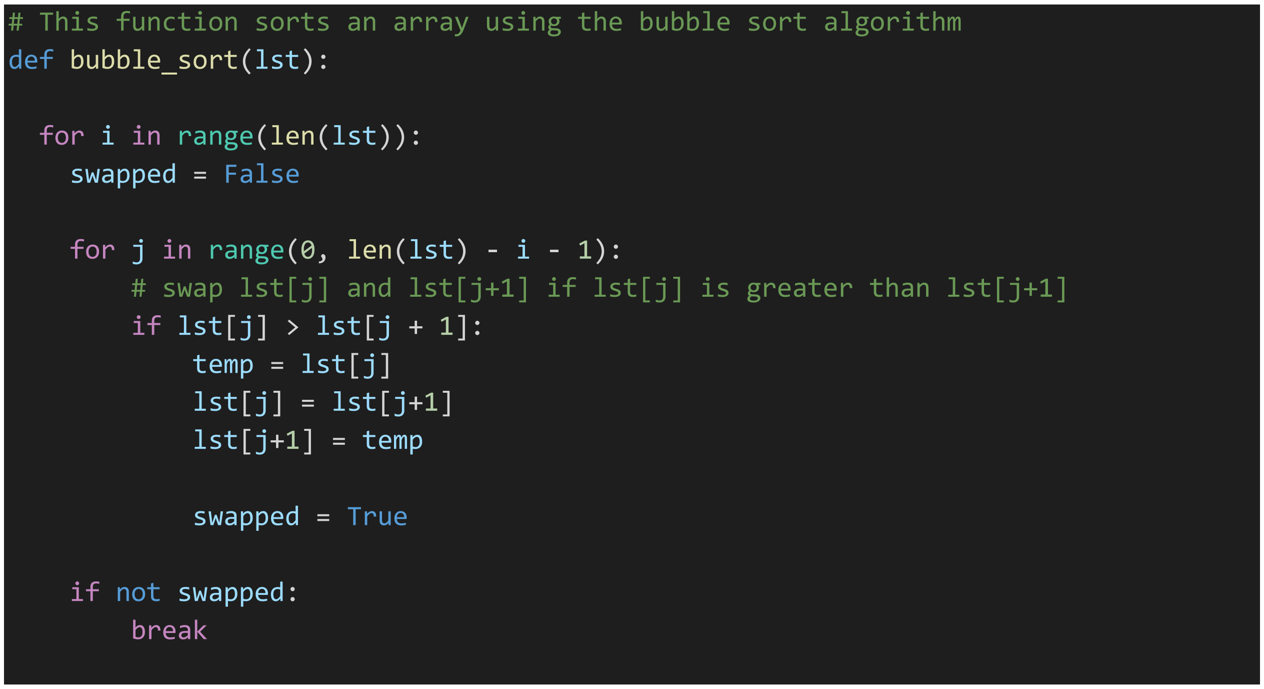
Selection Sort is another simple sorting algorithm that works by selecting the minimum element from an unsorted list in each iteration and placing it at the beginning of the list. It has the same time complexity as Insertion Sort, which is O(n^2) in the worst-case scenario.

```python
def selection_sort(lst):
    size = len(lst)

    # traverse through all array elements
    for step in range(size):
        min_idx = step

        # find the minimum element in remaining unsorted array
        for i in range(step + 1, size):
            if lst[i] < lst[min_idx]:
                min_idx = i

        # swap the found minimum element with the first element
        lst[step], lst[min_idx] = lst[min_idx], lst[step]
```

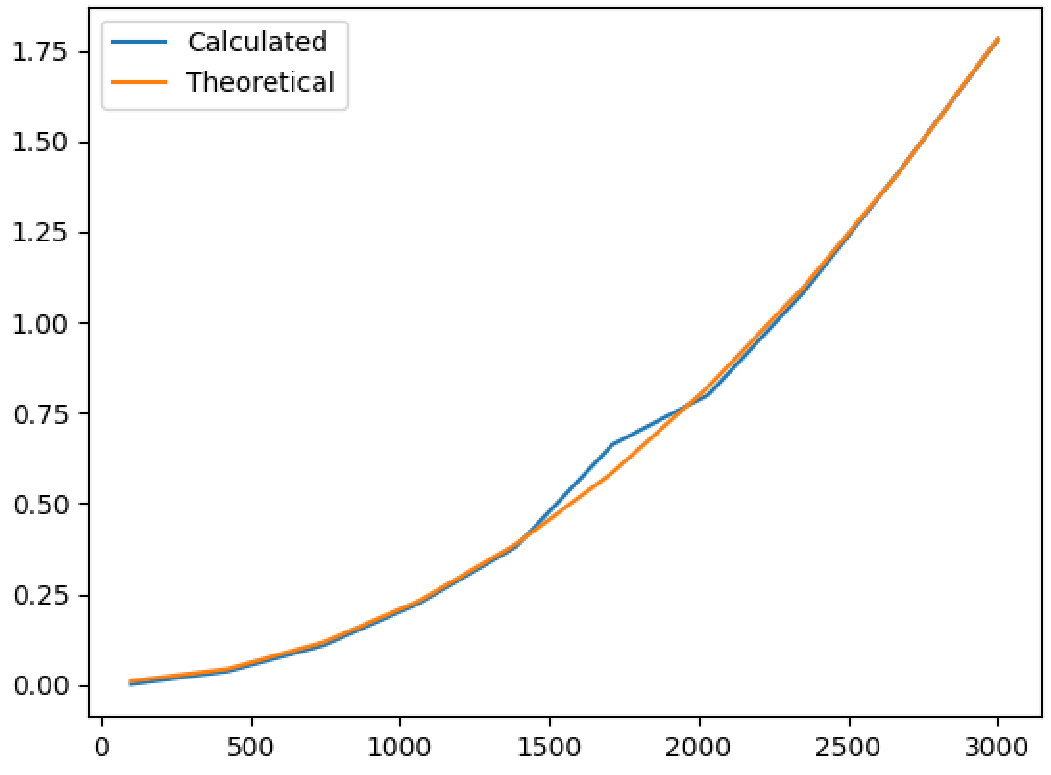


Bubble Sort

Bubble Sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. It has a worst-case time complexity of O(n^2), but its performance can be improved by detecting whether any swaps were made in the previous pass, which allows for early termination of the algorithm.



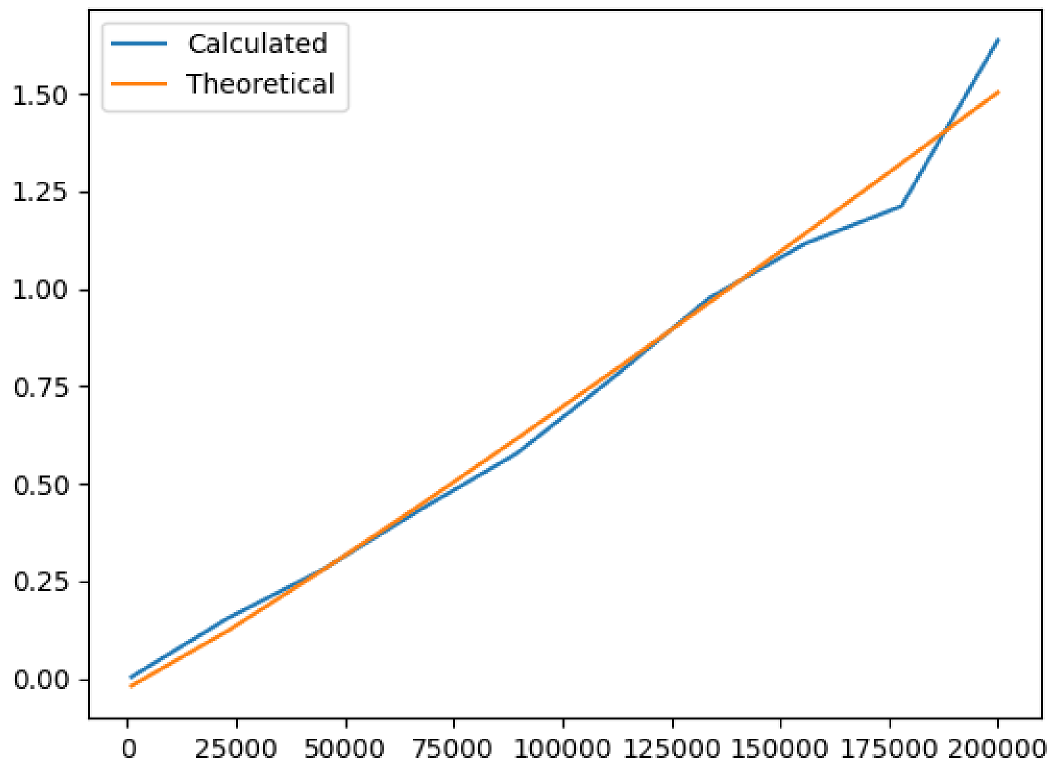
```python
# This function sorts an array using the bubble sort algorithm
def bubble_sort(lst):

  for i in range(len(lst)):
    swapped = False

    for j in range(0, len(lst) - i - 1):
        # swap lst[j] and lst[j+1] if lst[j] is greater than lst[j+1]
        if lst[j] > lst[j + 1]:
            temp = lst[j]
            lst[j] = lst[j+1]
            lst[j+1] = temp

            swapped = True

    if not swapped:
        break
```



Quick Sort

Quick Sort is a popular and efficient sorting algorithm that uses a divide-and-conquer approach. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The time complexity of Quick Sort is O(n log n) in the average case and O(n^2) in the worst-case scenario.

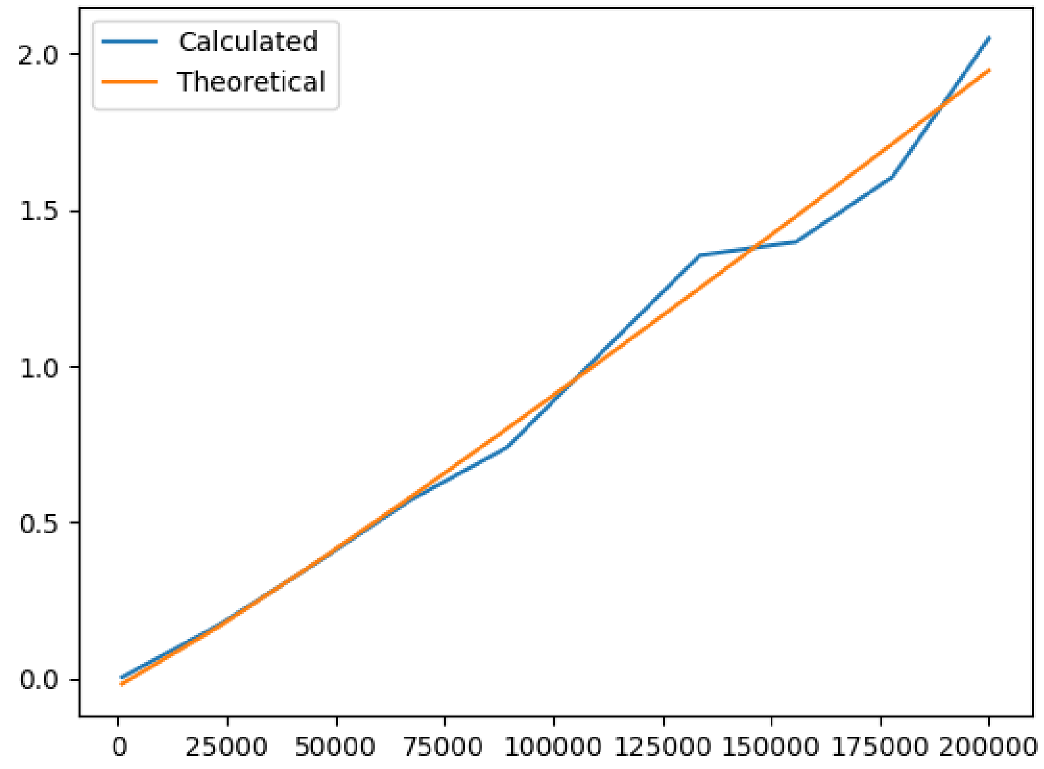
```python
# This function is used by the quick_sort function to partition the array
# around a pivot and returns the partition index
def partition(lst, low, high):
    pivot = lst[high]

    i = low - 1

    for j in range(low, high):
        if lst[j] <= pivot:
            i = i + 1

            # swap lst[i] and lst[j]
            (lst[i], lst[j]) = (lst[j], lst[i])

    # swap lst[i+1] and lst[high]
    (lst[i + 1], lst[high]) = (lst[high], lst[i + 1])

    return i + 1

# This function is called by the quick_sort function recursively to sort the
array
def quick_sort_aux(lst, low, high):
    if low < high:

        # partition the array around a pivot
        pi = partition(lst, low, high)

        # sort the left sub-array
        quick_sort_aux(lst, low, pi - 1)

        # sort the right sub-array
        quick_sort_aux(lst, pi + 1, high)

# This function sorts an array using the quick sort algorithm
def quick_sort(lst):
    quick_sort_aux(lst, 0, len(lst)-1)
```



Merge Sort

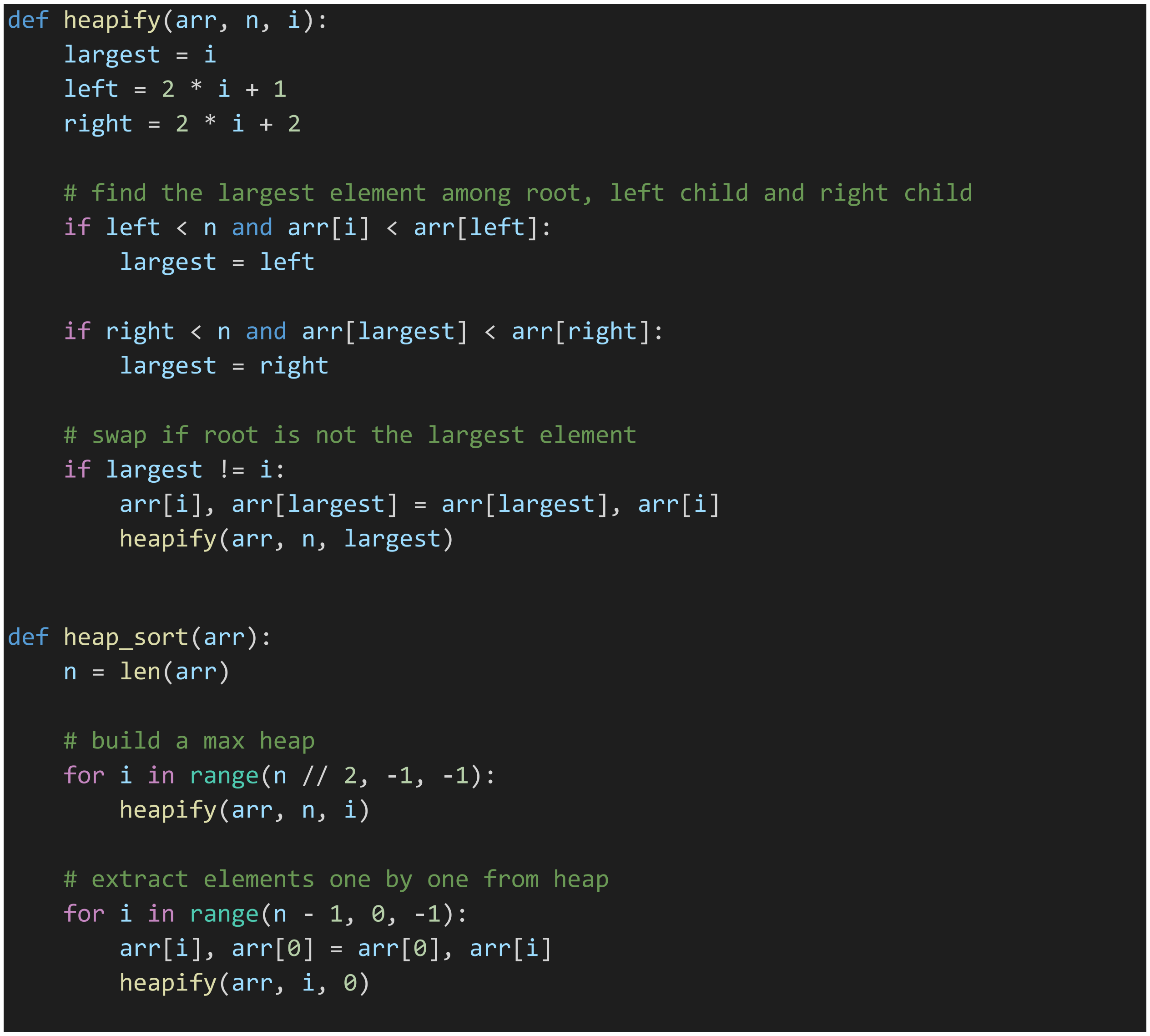
Merge Sort is another popular and efficient sorting algorithm that uses a divide-and-conquer approach. It works by recursively dividing the array into two halves, sorting each half, and then merging them back together. The time complexity of Merge Sort is O(n log n) in the worst-case scenario.

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # recursively sort the left and right halves
        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        # merge the two halves together
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # add any remaining elements from left half
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        # add any remaining elements from right half
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

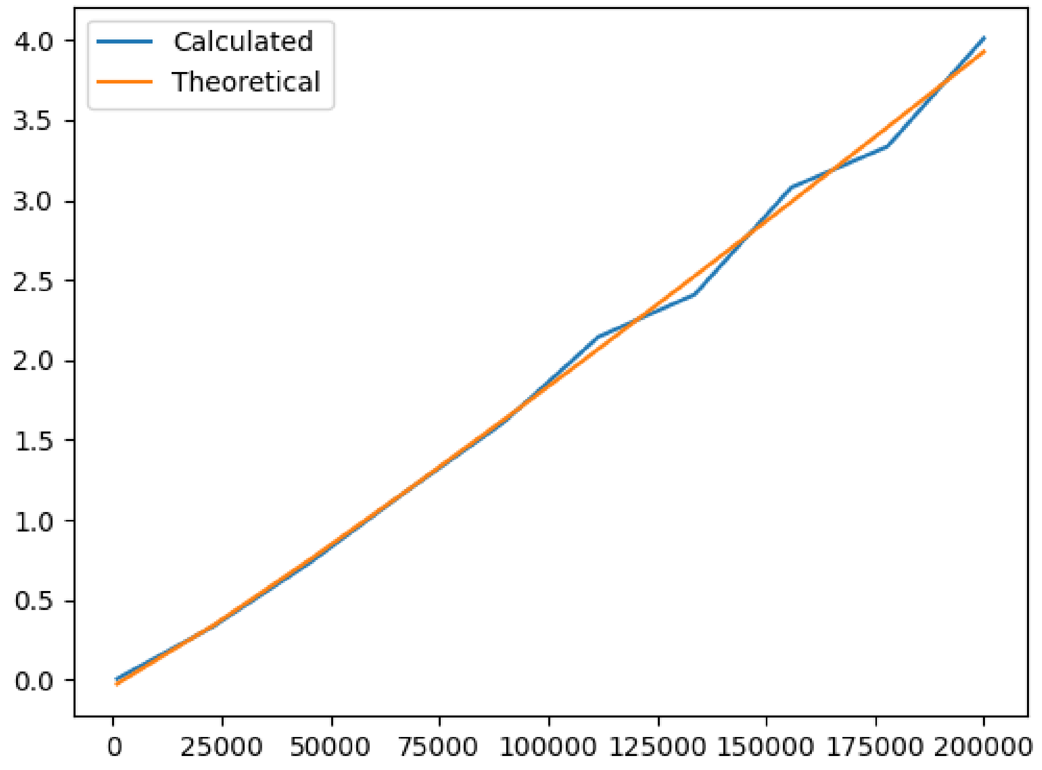


Heap Sort

Heap Sort is a comparison-based sorting algorithm that works by first building a heap from the input array and then repeatedly extracting the minimum element from the heap and placing it at the end of the sorted array. The time complexity of Heap Sort is O(n log n) in the worst-case scenario.



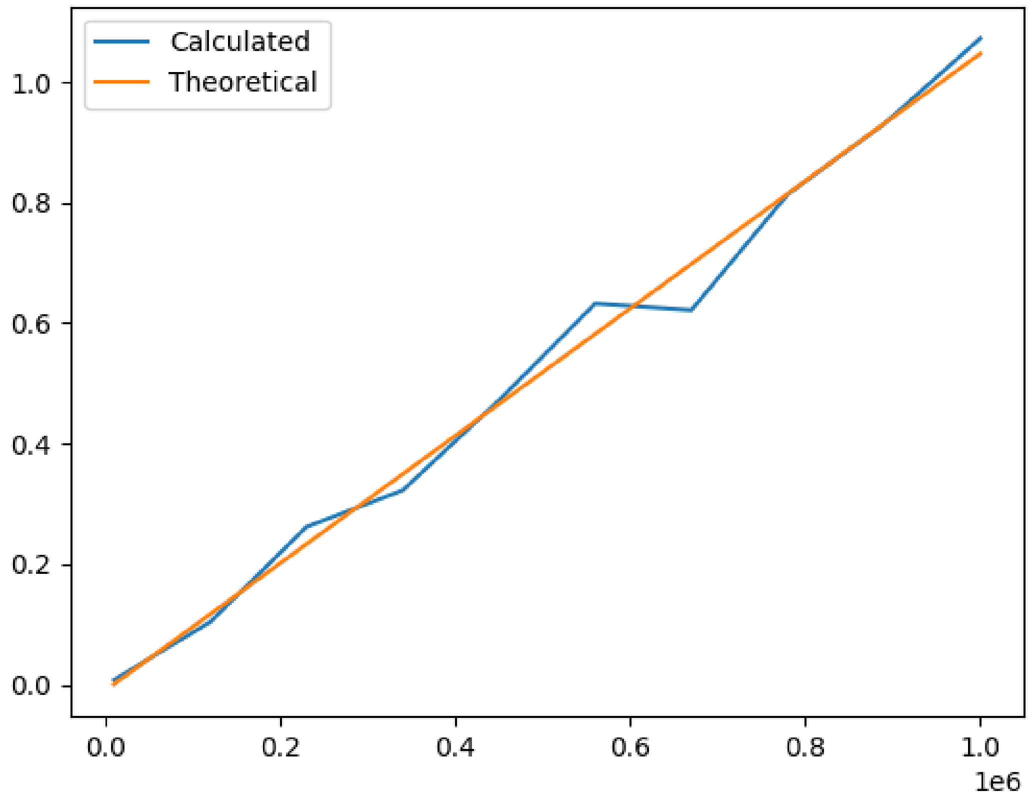
```python
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    # find the largest element among root, left child and right child
    if left < n and arr[i] < arr[left]:
        largest = left

    if right < n and arr[largest] < arr[right]:
        largest = right

    # swap if root is not the largest element
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def heap_sort(arr):
    n = len(arr)

    # build a max heap
    for i in range(n // 2, -1, -1):
        heapify(arr, n, i)

    # extract elements one by one from heap
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```



Counting Sort

Counting Sort is a non-comparison-based sorting algorithm that works by counting the number of occurrences of each element in the array and then using this information to determine the sorted order of the array. It has a time complexity of O(n + k), where k is the range of values in the input array.

```python
# This function sorts an array using the counting sort algorithm
def counting_sort(arr):
    size = len(arr)
    output = [0] * size

    count = [0] * 10

    # count the frequency of each element in the array
    for i in range(0, size):
        count[arr[i]] += 1

    # modify the count array to store the actual position of each element in the output array
    for i in range(1, 10):
        count[i] += count[i - 1]

    # fill the output array with the elements of the input array in the order determined by the count array
    i = size - 1
    while i >= 0:
        output[count[arr[i]] - 1] = arr[i]
        count[arr[i]] -= 1
        i -= 1

    # copy the output array back to the input array
    for i in range(0, size):
        arr[i] = output[i]
```

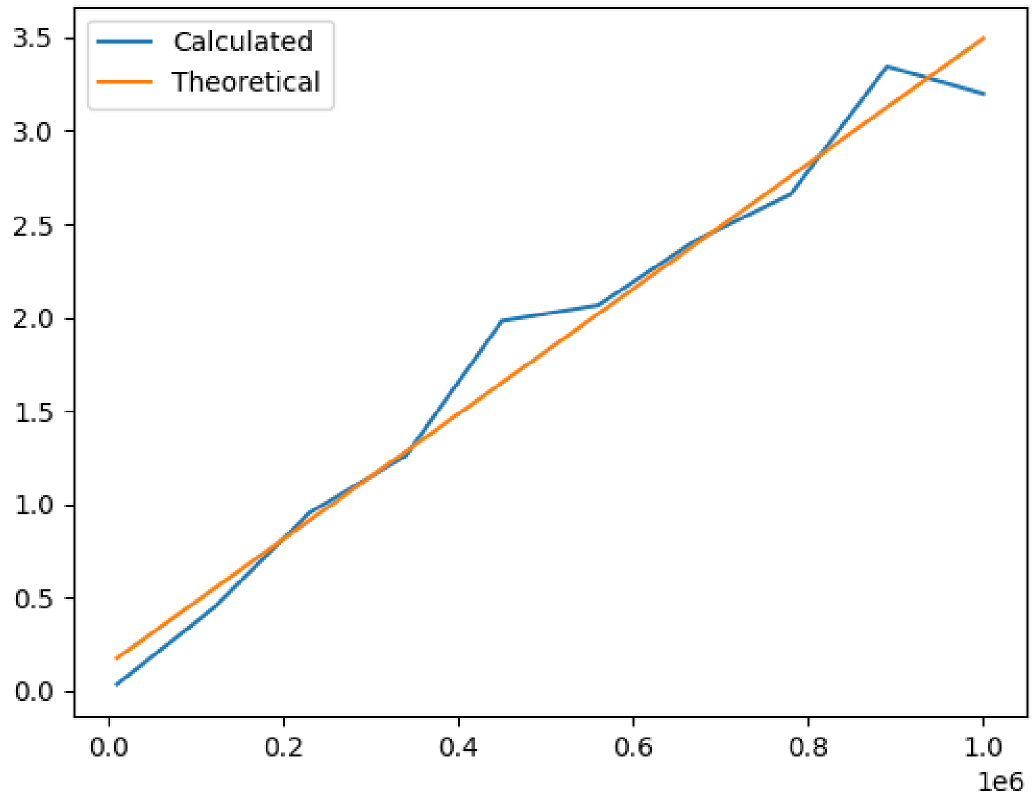


Radix Sort

Radix Sort is another non-comparison-based sorting algorithm that works by sorting the input array one digit at a time, from the least significant digit to the most significant digit. It has a time complexity of O(d * (n + k)), where d is the number of digits in the maximum value in the input array and k is the range of values.

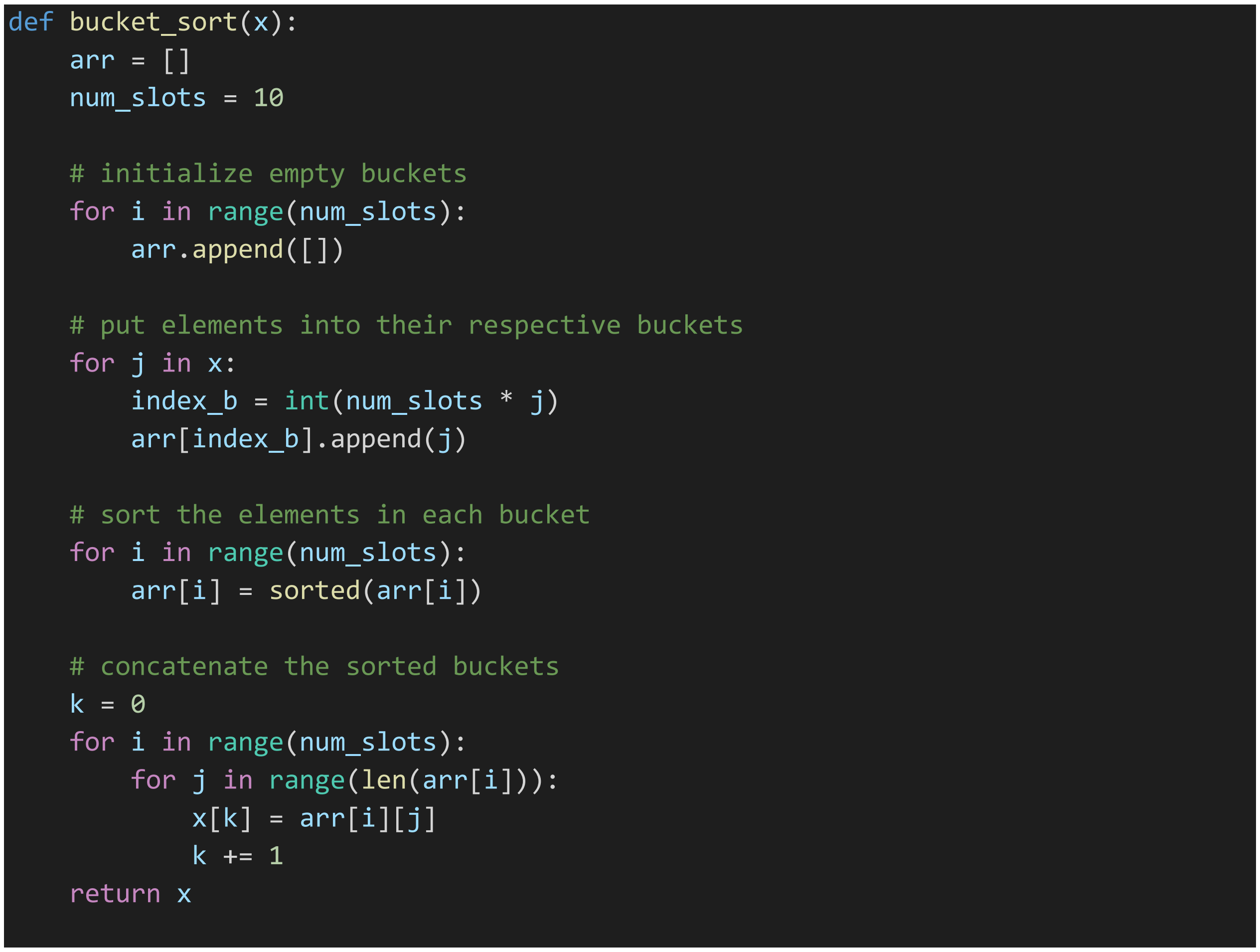
```
# Define the counting sort algorithm which will be used as a subroutine in the
radix sort algorithm.
def counting_sort(lst, place):
    # Get the size of the input list and initialize a list of zeros with the same
size
    # as the input list, which will hold the sorted output.
    size = len(lst)
    output = [0] * size
```

```python
    # Initialize a list of zeros to hold the count of each digit from 0 to 9.
    count = [0] * 10

    # Loop through the input list, counting the occurrence of each digit in the
place value.
    for i in range(0, size):
        index = lst[i] // place # Get the digit in the current place value
        count[index % 10] += 1 # Increment the count for that digit

    # Calculate the running total of the counts for each digit.
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Starting from the end of the input list, place each element into the sorted
output list
    # based on its position in the running total of counts for its digit.
    i = size - 1
    while i >= 0:
        index = lst[i] // place # Get the digit in the current place value
        output[count[index % 10] - 1] = lst[i] # Place the element in the correct
position in the output list
        count[index % 10] -= 1 # Decrement the count for that digit
        i -= 1

    # Copy the sorted output list back into the input list.
    for i in range(0, size):
        lst[i] = output[i]

# Define the radix sort algorithm which will sort the list by repeatedly calling
the counting sort
# algorithm on each digit in the numbers in the list, from the least significant
to the most significant.
def radix_sort(lst):
    max_element = max(lst) # Find the maximum element in the input list

    place = 1 # Initialize the place value to sort by to the least significant
digit
    while max_element // place > 0: # Continue sorting until all digits have been
sorted
        counting_sort(lst, place) # Sort the list by the current place value
        place *= 10 # Move to the next place value
```

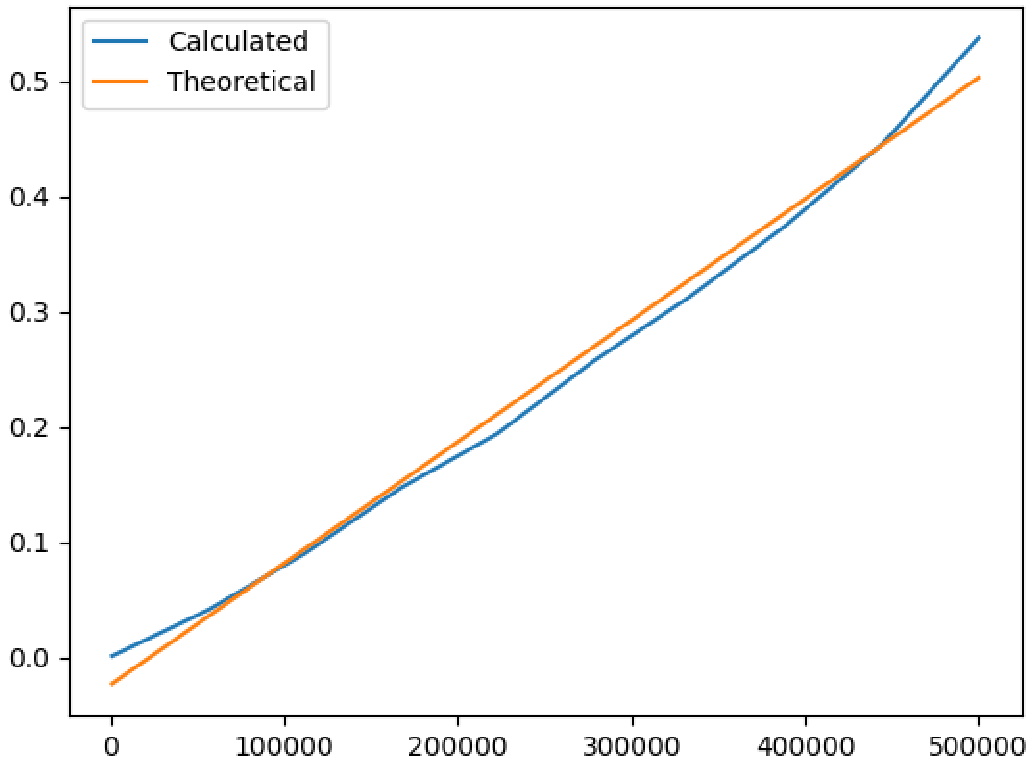


Bucket Sort

Bucket Sort is a non-comparison-based sorting algorithm that works by distributing the elements of an array into a number of "buckets" and then sorting each bucket individually using another sorting algorithm, such as Insertion Sort or Quick Sort. It has a time complexity of O(n + k), where k is the number of buckets.



```python
def bucket_sort(x):
    arr = []
    num_slots = 10

    # initialize empty buckets
    for i in range(num_slots):
        arr.append([])

    # put elements into their respective buckets
    for j in x:
        index_b = int(num_slots * j)
        arr[index_b].append(j)

    # sort the elements in each bucket
    for i in range(num_slots):
        arr[i] = sorted(arr[i])

    # concatenate the sorted buckets
    k = 0
    for i in range(num_slots):
        for j in range(len(arr[i])):
            x[k] = arr[i][j]
            k += 1
    return x
```


Calculated
Theoretical
0.5
0.4
0.3
0.2
0.1
0.0
0
100000
200000
300000
400000
500000