COP4533 Final Project: Merge Sort

Pieter Alley (49806796) & Juan Carlos Plate (34182137)

Merge Sort

Merge Sort is an efficient, general-purpose, and comparison-based divide-and-conquer sorting algorithm. It's efficient and general purpose in that in its worst case, it runs in a guaranteed $O(n\log n)$ time, on par with the best of the classic sorting algorithms; comparison-based in that elements are directly compared to insert into the sorted array; and divide-and-conquer in that algorithm recursively sorts subarrays, effectively breaking up the problem into subproblems until a base case is reached.

The divide-and-conquer nature of this algorithm makes it a suitable candidate for parallel processing, further improving on the efficiency of such an algorithm. The default sorting algorithm for Python, TimSort, uses a hybrid of merge sort and insertion sort. Beyond sorting, Merge Sort has applications in polynomial multiplication, genomic sequencing, and comparative ranking systems.

Implementation

Source code is better found in the Jupyter notebook or in the Python mergesort.py file.

```python
# Helper function for merging two already sorted halves of the array:
def merge(left, right):

  # initializing the resulting sorted array and the indices of interest:
  result = []
  i = 0
  j = 0

  # looping through the elements of both arrays to compare and place in sorted array:
  while i < len(left) and j < len(right):
    if left[i] <= right[j]:
      result.append(left[i])
      i += 1
    else:
      result.append(right[j])
      j += 1

  # the remaining elements can be assumed to be greater than all the elements already placed in the array,

  # looping through the remaining elements to clean up and add to sorted array:
  while i < len(left):
    result.append(left[i])
    i += 1
  while j < len(right):
    result.append(right[j])
    j += 1

  # resulting array is fully sorted.
  return result
```

```python
# General merge sort algorithm, recursively calls itself until the base case is reached.
def mergeSort(array):

  n = len(array)
  # base case of only one element in the array:
  if n <= 1:
    return array

  # setting the midpoint:
  if n%2 == 0:
      mid = int(n/2)
  else:
      mid = int(n/2)+1

  # splitting the array down the middle and recursively sorting the halves:
  left = mergeSort(array[0:mid])
  right = mergeSort(array[mid:n])

  # merging and returning the sorted halves of the array:
  return merge(left, right)
```

As seen in the source code above, merge sort is implemented using two different functions. The first function is merge sort which recursively splits the original input in half until each array is of size one. In order to do this, at each recursive call we find the midpoint of the input array and split the left side and right side into two different arrays. These subproblems are then passed into the merge function.

The merge function takes in two sorted halves of the array. It then loops through the elements of both arrays to compare and place the values in a final sorted array. Once one array is exhausted the remaining elements in the other array are added to the end of the result array. Finally we return the result array and our original input is now sorted.

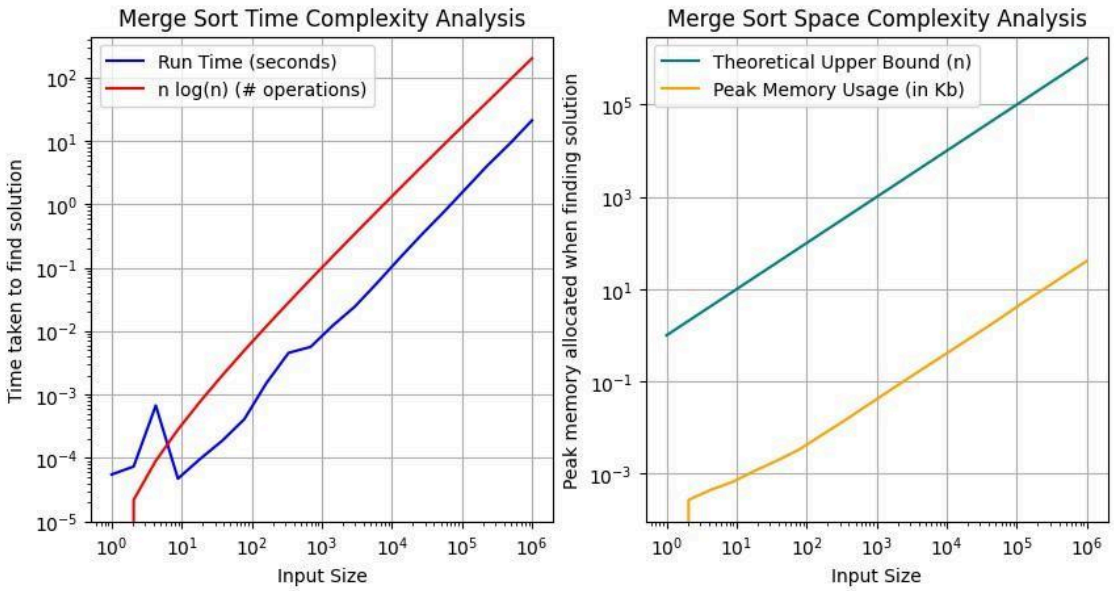
Experimental Analysis

The above implementation theoretically runs in $O(n\log n)$ time, even more interesting is that this complexity is maintained for best-case and average-case test cases. This is due to the fundamental nature of this divide-and-conquer algorithm, the algorithm will break down an array and then remerge it even if it's already sorted.

In terms of space complexity, merge sort is not done in-place, but it doesn't necessarily make an exorbitant amount of copies when making comparisons. In the worst case space complexity, this algorithm would be O(n).

Our experimental analysis reflects this, as the below plots show the theoretical time complexity plotted against their corresponding experimental values. It's not entirely accurate to compare run time with number of operations and peak memory usage with theoretical array size, but the plots exhibit the correlation well. As input size grows, the plot conforms to the theoretical upper-bound.

It's also interesting to note that the run time exceeds the theoretical upper bound for the smallest input sizes, this is likely a product of other function calls for testing memory allocation happening during testing.



Key Takeaways

From this project we learned that merge sort is a very useful and reliable sorting algorithm. The divide-and-conquer structure guarantees that the time complexity will always be $O(n\log n)$ no matter what input it takes. Merge sort is very useful for very large datasets that need to be sorted.

This algorithm does have some tradeoffs, specifically the temporary space it uses to sort the subproblems and its recursive nature. However, its guaranteed time complexity and ease of being scaled for larger and larger inputs makes it an extremely useful and powerful tool to sort data.