Merge Sort

By:

Dustin Smith

Merge sort is an efficient, general purpose comparison based sorting algorithm. Merge sort is a divide and conquer algorithm.

1. Divide the unsorted list into n sublists each containing one element.

2. Repeatedly merge sublists to produce new sorted sublists until there is on sorted sublist.

What would be the time complexity of this algorithm? Similar to a binary search, we are splitting the array in half by the mid point. However, unlike a binary search, merge sort has us operating on both splits. That is, we have $T(n) = 2 \cdot T(n/2)$ for the sort plus the $T(n) = 2 \cdot n$ for the merge operation. The weak version of the Master Theorem is

$$T(n) = \begin{cases} a \cdot T(n/b) + n^c, & n > 1 \\ d, & n = 1 \end{cases} \rightarrow T(n) = \begin{cases} \theta(n^c), & \log_b a < c \\ \theta(n^c \log_b n), & \log_b a = c \\ \theta(n^{\log_b a}), & \log_b a > c \end{cases}$$

We have that $a = 2$, $b = 2$, and $c = 1$. Therefore, the time complexity is $\theta(n \cdot \log_2 n)$. We need $\theta(n)$ space for the subarrays and $\theta(n)$ auxiliary space for the initial array.

```python
def merge_sort(arr: List[int]) -> List[int]:
  n = len(arr)
  if n == 1:
    return arr

  mid = n // 2

  left = merge_sort(arr[:mid])
  right = merge_sort(arr[mid:])

  return merge(left, right)


def merge(left: List[int], right: List[int]) -> List[int]:
  final = []
  i, j = 0, 0

  while i < len(left) and j < len(right):
    if left[i] < right[j]:
      final.append(left[i])
      i += 1
    else:
      final.append(right[j])
      j += 1

  final.extend(left[i:])
  final.extend(right[j:])

  return final
```