Analysis of Merge and Quick Sort

## Name: Brendan Yeong

## Student ID: 005025797

## Course: MSCS532

## Assignment 2

**Merge Sort**

**Problem Description**

Merge sort is one of the most basic divides and conquer sort algorithms. It can solve the problem of sorting a large array or list in order consistenly.

**Key Steps**

* Divide: Divide the list into two halves when only individual elements are left, and it can no longer be divided
* Conquer: Recursively sort both halves.
* Merge: Merge the two small, sorted halves into a new list in sorted order till all the halves are sorted and merged.

**Time complexity analysis**

* **Worst-case:**
* **Best-case :**
* **Average-case:**

1. Dividing the array/list:
   * At each recursive level it is dividing the array/list into two sub-problems. This step takes a time as it is doing arithmetic calculations.
2. Recursive steps:
   * At each depth, the recursive steps of a Merge Sort is where is the array or list size and 2 is because it is split in half.
3. Merging step:
   * At each recursive level, the merging steps, we are taking two sorted halves and comparing them and proceeds to merge them which takes time, which can be simplified to
4. Total complexity:
   * Since there are recursive calling steps, for each step it divides, , and merges, . Therefore, the time complexity is .

**The recurrences relation**The recurrences relation for Merge Sort is where

* is the array size time complexity of size .
* is the recursion of breaking down into sub-problems
* is the merging steps which executes in linear time.

**Substitution method**  
We can assume from its run-time of to where is a constant.  
We can sub the guess into the recurrence equation and solve it from there.

Thus,

**Recursion-tree method**Since merge sort divides the array in half, we can say that it has a branching factor of 2. So for each level it splits into 2 branches.

1. Each depth, has a number of children nodes where each node has a subarray or sub-problem size of where is the input array size.
   * If we have sub-problems at a depth of .
   * If we have sub-problems at a depth of 4.
2. At each depth the merging time complexity is in levels.

**Master method**we can set . Next, we calculate . Using the master method if , then .  
Which gives us Therefore Merge Sort has an optimal time complexity of .

**Merge sort code analysis results**

**A graph with numbers and colored dots

Description automatically generated**

**A black background with numbers

Description automatically generated**

**Theoretical analysis**

The Merge sorts algorithm gives one of the best and efficient algorithms, as its time complexity on best-case, average-case, worst-case are all . If we look at the graph the execution time is almost following the function of and all three types of input are fairly close to one and another.

**Practical implication**

It is mostly used to sort very large datasets, and it is stable. Another practical use is external sorting since it is more efficient to sort datasets that are too large to fit entirely in memory, example disk-based or RAM memory sorting.

**Explanation**

Merge Sort might underperform compared to Quick Sort in practice is because of the additional memory overhead and the less efficient memory access patterns. The need to copy elements into temporary arrays can slow down the process, especially in systems where memory allocation and deallocation are costly.

**Quick Sort**

**Problem Description**

Quick sort is the quickest divide and conquer sort-based algorithms. It solves the problem of sorting a using a pivot element, partitioning the array around the pivot where if it is less than the pivot, it comes before else it comes after.

**Key Steps**

* **Divide:** Partition the array into two sub-arrays. Sort the elements where if it is smaller or equal to the pivot it will be placed in the left sub-array, if greater the element will be placed into the right sub-array. Resulting the pivot is in the correct position (middle).
* **Conquer:** Recursively call the quick sort method to sort each of the subarrays, left side and the right side
* **Combine:** The left and right sub-array array are then Recursively step into using the steps above until all of sub-array has reached an element of one or zero.

**Time complexity analysis**

* **Worst-case:**
* **Best-case :**
* **Average-case:**

**Worst-case**: This case occurs when the partitioning produces uneven sub-problems such as, one with elements and the other with elements. It can also occur in an input array that is already completely sorted. Resulting the following:

**Best-case:** This case occurs when the partitions are evenly split to produce two sub-problems, each of the size of to give both the left side and right side the same number of elements. Which will result in a run-time of

**Average-case:** This case occurs when the partitions have a mix of “good” and “bad” splits. However, the average-case runtime of it is much closer to the best case than it is to the worst case. The runtime of the average case:

**The recurrences relation**Using the **substitution method** to prove the worst-case recurrence with, is equal to .

Let’s assume that and

Therefore .

Using the **master theorem,** we can solve the best-case where the partition are split into two subproblems, each of the size of exactly. We can describe the recurrence to be,

The same way we solve merge sort this falls into case 2 of the master theorem, if , then . Where , and gives us

Therefore .

Using the **Recursion-tree method** we can solve the average-case with the recurrence of,

We can make the partition to an even amount of 99 to 1 split.

Since it is a split evenly with the left side of the tree is with a depth of and right side with a depth of . Since we know that the right side has the most cost . After simplified, .

**Quick sort code analysis results**

**A graph with colored dots and numbers

Description automatically generated**

**A black background with white numbers

Description automatically generated**

**Theoretical analysis**

Quick sort is one of the most fastest sort algorithm, with a best-case (random datasets) of and an average-case (descending sorted dataset) of It also sort in a very efficient manner because it sorts in place, requiring lesser memory than merge sort. However, its worst-case (ascending sorted dataset) can degrade to and its run-time can be worse than merge sort.

**Practical implication**

In practice, Quick sort is faster than Merge Sort due to sorting in place and better cache performance, making this algorithm suitable for situation with limited memory.

From the previous class discussion I learned that quick-sort is one of the best to further improve into hybrid-sort algorithms and it is in most of the general-purpose sorting in many libraries like dual-pivot quick sort in Java or TimSort in Python. The only drawback is that it is unstable as it does not preserve the order of the equal elements.

**Explanation**

In my experiment quick sort does prove itself to be the superior practical performance, mainly due to its in-place sorting. As it is in-place sorting, it reduces memory usage and leverages better cache locality. As mentioned because of modern approaches, it is often used to develop modern hybrid sort methods to mitigate the worst-case scenario, making Quick Sort very efficient in practice.

Reference

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. The MIT Press.

Joshi, V. (2017, June 12). *Making sense of merge sort [part 2]*. Medium. https://medium.com/basecs/making-sense-of-merge-sort-part-2-be8706453209

Mahad. (2023, November 8). *A deep dive into merge algorithms: Unraveling the magic of external sorting with a sample...* Medium. <https://medium.com/@mbanaee61/a-deep-dive-into-merge-algorithms-unraveling-the-magic-of-external-sorting-with-a-sample-69dd2abf4316#:~:text=External%20sorting%20is%20a%20technique%20used%20to,storage%20and%20the%20limited%20main%20memory%20(RAM)>.

UMBC. (n.d.). https://redirect.cs.umbc.edu/~nam1/TA/HWSols/hw6sol.pdf