Synthesis Assignment 1
Erdun E
September 20, 2024
Dr. Alan Jamieson

CS 5800: Algorithm
Synthesis Assignment 1

1 Asymptotic Analysis

1.1 Big O

1.11 Explanation

Big-O notation is a mathematical concept to describe the efficiency of algorithms in computer science, specially to describe the time-complexity and space-complexity. Big-O notation allows people to discuss in the abstract that how the performance of an algorithm changes when the size grows. For example:

Time-complexity: it refers to the relationship between the execution time of algorithms and input size n.

- O(1): constant time. Whatever the input size, the execution time is always the same.
- O(n): linear time. When the input size increases, the execution time increases linearly as well.
- O(n^2): quadratic time. When the input size increases, the execution time increases quadratically.
- O(logn): logarithmic time. When the input size increases, the execution time increases logarithmically.

Space-complexity: it refers to the relationship between the required memory space and the size of the inputs by an algorithm during execution. The pattern is similar to the time complexity.

Big O is an upper bound notation that describes the maximum resource consumption of an algorithm in the worst case. This is particularly important for measuring the performance of an algorithm under the most unfavorable conditions.

1.12 Exercises

Analyze the time complexity of the Insertion Sort algorithm for the following array:

[7,3,5,2,9,1,4]

1.13 Solution

Insertion Sort is a sorting method that organizes an array by putting one item in the right spot at a time. Imagine you have a messy list of numbers. You go through the list and take each number, then place it in the correct order among the numbers you've already sorted. It's like sorting playing cards by picking one card at a time and inserting it into the right place in your hand. The specific step is as follows:

Initial Array

- Array is [7,3,5,2,9,1,4]

Iteration 1

- Start with [7]

Iteration 2

- Compare 3 with 7, then insert 3 before 7, result as [3,7]

Iteration 3

- Compare 5 with 3 and 7 and insert 5 between 3 and 7, result as [3,5,7]

Iteration 4

- Insert 2 at the beginning of the array, result as [2,3,5,7]

Iteration 5

- Insert 9 at the end of the array, result as [2,3,5,7,9]

Iteration 6

- Insert 1 at the beginning of the array, result as [1,2,3,5,7,9]

Iteration 7

- Insert 4 between 3 and 5, result as [1,2,3,4,5,7,9]

Time Complexity: is $O(n^2)$

Best Case is O(n)

- The best case happens when the list is already sorted. In this case, when adding new items, only need to check each one once, which makes it really quick—only takes O(n) time.

Worst Case is $O(n^2)$

- The worst case is when the list is sorted backward. In this case, you have to compare each new item with every other item already in the right order, which takes a lot longer about $O(n^2)$ time.

Average Case is$O(n^2)$

- On average, end up comparing about half of the items. This means it usually takes around $O(n^2)$ time, which is pretty slow compared to the best case.

Space Complexity: is O(1)

- Insertion Sort is a way to sort an array without needing extra space. It only uses a little bit of extra memory for a couple of variables to keep track of the current position and the number we're trying to place in the right spot. While sorting, it rearranges the elements right in the same array. Therefore, its space complexity is constant (1).

1.2 Big Omega

1.21 Explanation

Big-Ω notation is used to describe the time complexity or space complexity of an algorithm in the best case. It indicates how much time or space that the algorithm takes at least for all possible inputs. Big-Ω could help to understand the best performance under the best conditions.

Time-complexity: it refers to the relationship between the execution time of algorithms and input size n.

- $\Omega(1)$: constant time. Whatever the input size, the execution time is always the same.
- Ω(n): linear time. When the input size increases, the execution time increases linearly as well.
- $\Omega(n^2)$: quadratic time. When the input size increases, the execution time increases quadratically.
- Ω(logn): logarithmic time. When the input size increases, the execution time increases logarithmically.

Space-complexity: it refers to the relationship between the required memory space and the size of the inputs by an algorithm during execution. The pattern is similar to the time complexity.

Big-Ω is often used to describe the lower bound of an algorithm, indicating that the algorithm can not be faster than Ω. While the worst case is more practical, the best case can also serve as an additional analysis of the algorithm.

1.22 Exercises

Use binary search to find the target element 13 in the following sorted array and analyze the time complexity:

[1,3,5,7,9,11,13]

1.23 Solution

Binary search is an efficient search algorithm for finding a target element in an ordered array. It divides the search interval repeatedly into two halves to narrow the possible range of the target element, until the target element be found or the search interval is empty. The specific step is as follows:

Initial array and start at the middle element of the array

- The search interval is [1,3,5,7,9,11,13], and the target element is 13

Fisrt comparison, compare current element with target number, if smaller than target element, continue to search the left half, if bigger than target element, continue to search right half, if equal to target, target element found.

- The middle element is 7, and the target element is 13, the target element should be in the right half[9,11,13].

Second comparison

- The middle element is 11, and the target element is 13, the target element should be in the right half [13].

Third comparison

- The middle element is 13, and the target element is 13, the target element found in three comparisons.

Time Complexity:

Best Case: $\Omega(1)$

- The best case for Binary Search is when the target element [13] is found as the middle element of the array on the first comparison. This means that only one comparison is required to find the target element [13]. So the time complexity is $\Omega(1)$.

Worst Case: O(logn)

- The worst case is when the target element [13] is not present in the array or is located at the far front or end of the array. In such cases, the array needs to be repeatedly divided until the interval size becomes 1, which means logn comparisons are required. So the time complexity is O(logn).

Average Case: O(logn)

- For a randomly selected target element in the array, the average number of comparisons required to find the element is proportional to log n. This is because the array is halved with each comparison, thus exponentially reducing the search space. So the time complexity is O(logn).

Space Complexity:

Iterative: O(1)

- In the iterative implementation of Binary Search, no additional space is used other than a few variables to track the left, right, and middle indices. The space requirement is independent of the input size. So the space complexity is O(1).

Recursive: O(logn)

- In the recursive implementation of Binary Search, each recursive call adds a new frame to the call stack. The depth of the recursion is logn because the array is halved at each step. So the space complexity is O(logn).

1.3 Big Theta

1.31 Explanation

Big-θ is used to describe the average time or space complexity of an algorithm. It represents the exact growth rating of an algorithm in all possible cases. That means it's both the upper and lower bounds of an algorithm's complexity. Big-θ common time and space complexities:

Time-Complexity: it refers to the relationship between the execution time of algorithms and input size n.

- $\theta(1)$: constant time. Whatever the input size, the execution time is always the same.
- θ(n): linear time. When the input size increases, the execution time increases linearly as well.
- $\theta(n^2)$: quadratic time. When the input size increases, the execution time increases quadratically.
- θ(logn): logarithmic time. When the input size increases, the execution time increases logarithmically.

Space-Complexity: it refers to the relationship between the required memory space and the size of the inputs by an algorithm during execution. The pattern is similar to the time complexity

If an algorithms's time complexity is θ(n), it means that the execution time is proportional to n in both the best and worst case.

1.32 Exercises

Use Bubble Sort to sort the following array and show the state of the array after each step:

[5,2,9,1,5,6]

1.33 Solution

Bubble sort is a simple and inefficient sorting algorithm. It works by repeatedly iterating over an array, comparing adjacent elements and swapping them if they are in the wrong order. The largest element "bubbles" into its correct position at the end of the array. In short, after each swap, the largest unsorted element is moved to its correct position.

The steps of Bubble Sort are as follows:

- Start from the first element of an array and compare adjacent elements. If the previous one is larger than the next one, swap them.
- After each move, the last element of the unsorted part is in its correct position.
- Repeat the above step until each element is in its correct position and the array is sorted.

Solution:

Initial Array:

$$[5,2,9,1,5,6]$$

first round comparison and swap:

- Compare 5 and 2, $5 > 2$, then swap:

$$[2,5,9,1,5,6]$$

- Compare 5 and 9, $5 < 9$, no need swap:
- Compare 9 and 1, $9 > 1$, then swap:

$$[2,5,1,9,5,6]$$

- Compare 9 and 5, $9 > 5$, then swap:

$$[2,5,1,5,9,6]$$

- Compare 9 and 6, $9 > 6$, then swap:

$$[2,5,1,5,6,9]$$

Second round comparison and swap:

- Compare 2 and 5, $2 < 5$, no need swap:
- Compare 5 and 1, $5 > 1$, then swap:

$$[2,1,5,5,6,9]$$

- Compare 5 and 5, $5 = 5$, then swap:

$$[2,1,5,5,6,9]$$

- Compare 5 and 6, $5 < 6$, no need swap:

$$[2,1,5,5,6,9]$$

Third round comparison and swap:

- Compare 2 and 1, $2 > 1$, then swap:

$$[1,2,5,5,6,9]$$

- Compare 2 and 5, $2 < 5$, no need swap:
- No need for swaps for the remaining elements.

The final sorted array is:

$$[1,2,5,5,6,9]$$

Time Complexity is $\theta(n^2)$

Best Case: O(n)

- The array is already sorted such as [1,2,5,5,6,9]. Bubble Sort only needs to pass through the array once without any swaps. So the best case time complexity is O(n).

Worst Case: O(n^2)

- The array is in reverse order such as [9,6,5,5,2,1]. In this case, each move requires the maximum number of swaps. So the worst case time complexity is O(n^2).

Average Case: $\theta(n^2)$

- Even though in this array [5,2,9,1,5,6], the total number of comparisons was smaller than the worst case, it still follows a quadratic growth pattern, because the quadratic nature of the algorithm remains dominant over all passes. The cumulative sum of operations will always be of the order $\theta(n^2)$.

Space Complexity is $\theta(1)$

- Bubble Sort is an in-place sorting algorithm, meaning that it does not require additional memory for extra variables. So the space complexity is $\theta(1)$.

2 Divide and Conquer

2.1 Technique Definition

2.11 Explanation

The Divide and Conquer algorithm is a powerful algorithm design paradigm, that lets a complex problem break down into smaller problems. The basic idea is to recursively divide the complex problem into two or more smaller problems with the same type until it can't be divided. and solve each of the smallest questions independently, then combine their solutions to solve the complex problems. Compared with solving the problem directly in one step, this approach can significantly reduce the computational complexity. The steps are as follows :

Divide:

- The problem is divided into smaller problems that are similar to the original problem but scale smaller. This step is usually performed recursively until the smaller problems are simple enough to be solved directly.

Conquer:

- The smaller problems are solved independently. If the smaller problems is still large, the divide and conquer strategy is applied recursively. This process continues until each smaller problem is small enough to be solved directly.

Combine:

- This step combines the results of the smaller problems to solve the original problem.

And, the Divide and Conquer has some pros: it breaks complex problems down into smaller, more manageable subproblems, which can be easier to solve. This approach often leads to more efficient algorithms, as it reduces the overall computational complexity, especially for problems that have similar optimal substructure. As well, Divide and Conquer is well-suited for parallel processing, since independent subproblems can be solved concurrently, further improving performance. Overall, this approach creates efficient and scalable solutions to complex problems.

2.12 Exercises

N/A

2.13 Solution

N/A

2.2 Divide And Conquer Multiplication

2.21 Explanation

Divide and Conquer Multiplication is a smart way to multiply big numbers by splitting them into smaller pieces, multiplying these smaller parts, and then putting everything back together. A famous example of this method is Karatsuba's Algorithm, which makes multiplying faster by reducing the number of small multiplications, only have to do compared to the regular way we multiply numbers. Karatsuba's Algorithm helps by reducing the number of multiplications to $O(n^{log_2 3}) \approx O^{1.585}$. This makes it much faster and more efficient when dealing with large numbers. This is how Karatsuba's Algorithms works:

Divide:

- Split each of the two binary strings A and B into two equal halves, m is half the length of the binary strings.
- A $= A_1 \cdot 2^m + A_0$
- B $= B_1 \cdot 2^m + B_0$

Conquer:

- Compute the following three products recursively.
- P1 $= A_1 \times B_1$
- P2 $= A_0 \times B_0$
- P3 $= (A_1 + B_1) \times (A_0 + B_0)$

Combine:

- Using the values of P1, P2, and P3, calculate the final result:
- $A \cdot B = P1 \cdot 2^{2m} + (P3 - P2 - P1) \cdot 2^m + P2$

Karatsuba's Algorithm is a method used to multiply two numbers more efficiently. It's especially helpful for multiplying really big numbers or binary strings. Instead of multiplying numbers directly, it splits the problem into smaller parts and solves them step-by-step. This way, it makes the whole process faster and easier to handle.

2.22 Exercises

Multiply the following two distinct binary strings of length 16 using Karatsuba's Algorithm. Show all steps involved.

Binary String 1: A = 1011101010111010
Binary String 2: B = 1101101010110110

2.23 Solution

Step 1: Split the binary strings

- For A = 1011101010111010
- A_1 = 10111010
- A_0 = 10111010
- For B = 1101101010110110
- B_1 = 11011010
- B_0 = 10110110

Step 2: compute the three parts

- P1 = $A_1 \times B_1$
- A_1 = 10111010
- B_1 = 11011010
- P1 = 1001110011110100

$$\begin{array}{r}
10111010 \\
\underline{\times\ 11011010} \\
00000000 \\
10111010 \\
00000000 \\
10111010 \\
10111010 \\
00000000 \\
10111010 \\
\underline{+\ 10111010} \\
1001110011110100
\end{array} \tag{1}$$

- P2 = $A_0 \times B_0$
- A_1 = 10111010
- B_1 = 10110110
- P2 = 1000010000101100

$$
\begin{array}{r}
10111010 \\
\times\ 10110110 \\
\hline
00000000 \\
10111010 \\
10111010 \\
00000000 \\
10111010 \\
10111010 \\
00000000 \\
+\ 10111010 \\
\hline
1000010000101100
\end{array}
\tag{2}
$$

- P3 = $(A_1 + A_0) \times (B_1 + B_0)$
- $A_1 + A_0$ = 10111010 + 10111010 = 101110100
- $B_1 + B_0$ = 11011010 + 10110110 = 110110100
- P3 = 101110100 $\times$ 110110100 = 100100011100000000

$$
\begin{array}{r}
101110100 \\
\times\ 110110100 \\
\hline
000000000 \\
000000000 \\
101110100 \\
000000000 \\
101110100 \\
101110100 \\
000000000 \\
101110100 \\
+\ 101110100 \\
\hline
100100011100000000
\end{array}
\tag{3}
$$

Step 3: Calculate P3 - P1 - P2

- P3 - P1 - P2 = 100100011100000000 - 1001110011110100 - 1000010000101100 = 1001000101000000

$$
\begin{array}{r}
100100011100000000 \\
1001110011110100 \\
-\ 1000010000101100 \\
\hline
1001000101000000
\end{array}
\tag{4}
$$

Step 4: Combine the parts

- The formula AB = P1 $\cdot\ 2^{2m}$ + (P3 - P1 - P2) $\cdot\ 2^m$ + P2
- m = 8 because A_1,A_0,B_1, and B_0 length is 8.
- The first part = P1 $\cdot\ 2^{16}$ = 1001110011110100 $\cdot\ 2^{16}$ = 10011100111101000000000000000000
- The second part = (P3 - P1 - P2) $\cdot\ 2^8$ = 1001000101000000 $\cdot\ 2^8$ = 100100010100000000000000
- Final AB result = 10011100111101000000000000000000 + 100100010100000000000000 + 1000010000101100 = 100111100011100110011101001010100

$$
\begin{array}{lr}
 & 10011100111101000000000000000000 \\
 & 100100010100000000000000 \\
+ & 1000010000101100 \\
\hline
\multicolumn{2}{r}{100111100011100110011101001010100}
\end{array}
\tag{5}
$$

The binary string A=1011101010111010 and B=1101101010110110 using Karatsuba's Algorithms is 10011110001110011001110100101010100

2.3 Divide And Conquer Algorithmic Design

2.31 Explanation

This section talks about the Divide and Conquer method, which is a way to tackle tough problems by breaking them down into smaller, easier ones. You solve each small problem separately and then put all the answers together at the end. This method is super helpful for problems that have a repeating pattern because splitting them up makes them way less complicated.

In section 2.32, shown Divide and Conquer strategy to find the biggest and smallest numbers in an array. First, we split the array in half. Then, we look for the biggest and smallest numbers in each half separately. Finally, we combine our results by comparing the largest and smallest numbers from both halves.

Using this method cuts down on the number of comparisons we need to make compared to just searching through the whole list one by one, which would take O(n) time. The whole idea behind this algorithm is to make things more efficient, especially when we're dealing with a lot of data, by breaking the problem into smaller pieces and avoiding unnecessary comparisons.

2.32 Exercises

Design an algorithm that uses the divide and conquer approach to find the biggest and smallest numbers in an array of integers. Explain the steps you design this algorithm, explain its correctness, and analyze its time complexity.

2.33 Solution

Given the question, we know that there is an array with integers, and we need to find the biggest and smallest numbers. One straightforward approach is to iterate through the array once and keep tracking the biggest and smallest numbers, this would take O(n) time. But in this case, we need to use the divide and conquer approach. So break it down first as shown in section 2.11.

Divide:

- Split the array into two halves, and the length of the array is n.
- Left half is arr[0...mid]
- Right half is arr[mid+1 ... n - 1]
- Middle is (0 + n - 1) / 2

Conquer:

- Recursively to find out the biggest and smallest number in each half.
- This is the core of divide and conquer that each subproblem is a smaller instance of the whole complex problem.

Combine:

- Once find out the biggest and smallest number in both halves, combine them to find the biggest and smallest number of the entire array.
- The biggest number is the greater number from the two halves's biggest number.
- The smallest number is the smaller number from the two halves's smallest number.

Summarize the breaking down before write Pseudocode:

Special Case

- If the array is empty, return null.
- If the array only has one number, that is both the biggest and smallest number.
- If the array only has two numbers, that compare them directly and determine which is the biggest and which is the smallest.

Main Case (Recursive Case)

- Once the array has more than two numbers, divide them into two halves recursively.
- Find the biggest and smallest number in both halves recursively.
- Combine the results by comparing the biggest of two halves and the smallest of the two halves.

Algorithm: 1 Divide And Conquer Psesudocode

```
function findMaxMin(array, low, high)
    if if array.length == 0 then
        return null;
    end if
    if if low = high then // only one number.
        return (array[low], array[low]);
    end if
    if if high = low + 1 then // two numbers.
        if array[low] < array[high]  then
            return (array[high], array[low]); // max, min
        else
            return (array[low], array[high]); // max, min
        end if
    end if
    min = (low + high) / 2
     // recursively find max and min for left half
    (leftMax, leftMin) = findMaxMin(array, low, mid)
     // recursively find max and min for right half
    (rightMax, rightMin) = findMaxMin(array, mid + 1, high)
     // combine the halves
    overallMax = max(leftMax, rightMax)
    overallMin = min(leftMin, ightMin)
    return (overallMax, overallMin)
end function
```

Pseudocode:

Explanation of correctness:

- The algorithm works well because it correctly divides the array into two halves and finds the biggest and smallest number in each half recursively. First, it deals with small arrays (one or two elements) by directly returning the biggest and smallest. Then, it splits the array into smaller parts to make the problem easier. Each smaller part is solved using the same steps, once the results of both halves are computed, the final maximum is simply the larger of the two maxima, and the final minimum is the smaller of the two minima. This ensures that no potential maximum or minimum is missed, making the algorithm correct.

Time Complexity: O(n)

- The divide and conquer algorithm time complexity is O(n). Because at each recursion level, the array is split into two halves, resulting in two recursive calls. The cost of splitting the array and combining the results is constant O(1) for each level
- The recurrence relation for this process is $T(n)=2T(\frac{n}{2})+O(1)$, which solves to O(n).

3 Recurrences and Master Theorem

3.1 Recurrences

3.11 Explanation

A recurrence relation or call it difference equation is a way to describe a sequence in which each term is defined based on the terms that come before it. This concept is often applied in analyzing algorithms, particularly to understand the time complexity of recursive functions. One common approach to solving a recurrence is the iterative or expansion method, which involves expanding the relation multiple times until you can identify a pattern or reach a base case.

3.12 Exercises

Solve the recurrence $\mathbf{T(n) = 2T(\frac{n}{2}) + n}$ using the expansion method.

3.13 Solution

Start with the recurrence

- $T(n) = 2T(\frac{n}{2}) + n$

Expand the recurrence that replace $\mathbf{T(\frac{n}{2})}$ with its recurrence

- $T(n) = 2\left[2T(\frac{n}{4}) + \frac{n}{2}\right] + n$

Simplify the recurrence

- $T(n) = 4T(\frac{n}{4}) + 2n$

Further expand the recurrence with $\mathbf{T(\frac{n}{4})}$

- $T(n) = 4\left[2T(\frac{n}{8}) + \frac{n}{4}\right] + 2n$

Simplify the recurrence

- $T(n) = 8T(\frac{n}{8}) + 3n$

After expanding twice, there are found some rules, so after k expansions.

- $T(n) = 2^kT(\frac{n}{2^k}) + kn$

Then the base case is when the problem size become to 1, i.e. like

- $\frac{n}{2^k} = 1$

To solve for k, we reorder the equations

- n = 2^k

Taking the logarithm

- k = log_2n

So, it means after k = log_2n expansions, the problem size reach to 1, at the point which the recurrent stops. The base case is usually T(1) = c, where c is some constant time for solving the smallest subproblem. Then, go back to the expanded recurrence

- T(n) = $2^{log_2n} \cdot T(1) + log_2n \cdot n$

Given 2^{log_2n} = n T(1) = c, then simplify

- T(n) = $n \cdot c + nlog_2n$

So after full expansion

- T(n) = $nc + nlog_2n$

The term nc represents a liner growth O(n) because c is a constant, and the term $nlog_2n$ grows faster due to it's logarithmic factor. Therefore the final solution is

- T(n) = $nlog_2n$

3.2 Master Theorem

3.21 Explanation

The Master Theorem is simply a set of formulas for quickly calculating the time complexity of a divide and conquer Algorithm with recurrence.

And also, there is a shortcut to solve recurrences of the form provided by Master Theorem.

- $T(n) = aT(\frac{n}{b}) + n^d$

Where

- a $>$ 0 and b $>$ 1
- $d \geq 0$

The solution depends on comparing d and log_ba, there are three cases:

- T(n) = O(n^d) when a $<$ b^d that $log_ba < d$
- T(n) = O(n^dlogn) when a = b^d that $log_ba = d$
- T(n) = O(n^{log_ba}) when a $>$ b^d that $log_ba > d$

3.22 Exercises

Problem 1:

- Solve the recurrence $T(n) = 2T(\frac{n}{4}) + n^2$ using the Master Theorem.

Problem 2:

- Solve the recurrence $T(n) = 4T(\frac{n}{2}) + n^2$ using the Master Theorem.

Problem 3:

- Solve the recurrence $T(n) = 9T(\frac{n}{3}) + n$ using the Master Theorem.

3.23 Solution

Problem 1 Solution:

Given the recurrence $T(n) = 2T(\frac{n}{4}) + n^2$, we know

- a = 2, b = 4, d = 2

Compute $log_b a$

- $log_4 2 = 0.5$

Compare $log_b a$ and d

- $0.5 < 2$

This matches case 1 of the Master Theorem, so the solution is

- T(n) = $O(n^d) = O(n^2)$

Thurs the solution of the recurrence $T(n) = 2T(\frac{n}{4}) + n^2$ is $O(n^2)$

Problem 2 Solution:

Given the recurrence $T(n) = 4T(\frac{n}{2}) + n^2$, we know

- a = 4, b = 2, d = 2

Compute $log_b a$

- $log_2 4 = 2$

Compare $log_b a$ and d

- $2 = 2$

This matches case 2 of the Master Theorem, so the solution is

- T(n) = $O(n^d logn) = O(n^2 logn)$

Thurs the solution of the recurrence $T(n) = 4T(\frac{n}{2}) + n^2$ is $O(n^2 logn)$

Problem 3 Solution:

Given the recurrence $T(n) = 9T(\frac{n}{3}) + n$, we know

- a = 9, b = 3, d = 1

Compute $log_b a$

- $log_3 9 = 2$

Compare $log_b a$ and d

- $2 < 1$

This matches case 3 of the Master Theorem, so the solution is

- T(n) = $O(n^{log_b a}) = O(n^2)$

Thurs the solution of the recurrence $T(n) = 4T(\frac{n}{2}) + n^2$ is $O(n^2)$

3.3 Using Master Theorem To Analyze A Recursive Algorithm

3.31 Explanation

This section expressed the recurrence relation for Merge Sort and used the Master Theorem to find its Big-O time complexity, which is O(n log n). This analysis demonstrates how the Master Theorem can be effectively applied to evaluate the performance of recursive algorithms.

3.32 Exercises

Analyze the time complexity of the Merge Sort algorithm using the Master Theorem.

3.33 Solution

Define the Merge Sort algorithm

- As shown in section 2.1, Merge Sort is a recursive sorting algorithm that works as follows:
- Divide, the array is split into two halves.
- Conquer, each half is recursively sorted using Merge Sort
- Combine, the two sorted halves are merged into a single sorted array.

Write the recurrence relation

- The recurrence for Merge sort as $T(n) = 2T(\frac{n}{2}) + O(n)$
- Where:
- a = 2, because of Merge Sort divides the array into two subproblems.
- b = 2, because of each subproblem size is $\frac{n}{2}$
- d = 1, the combine step takes liner time to merge two sorted halves.

Using Master Theorem to analyze

- Now we know the recurrence relation is $T(n) = 2T(\frac{n}{2}) + O(n)$ and a = 2, b = 2, d = 1.
- Compute $log_b a = log_2 2 = 1$
- Compare $log_b a$ and d, 1 = 1
- This matches Case 2 of the Master Theorem which is shown in section 3.21
- The overall time complexity is: T(n) = $O(n^d logn)$ = O(nlogn)

The Merge Sort algorithm works by breaking the input down into two smaller problems, each about half the size of the original $\frac{n}{2}$. It sorts these smaller problems recursively, and then combines the sorted results in a linear. We applied the Master Theorem to analyze its complexity, which showed that the main factor, its overall performance comes from both the recursive divisions, and the linear merging process. As a result, the total time complexity for Merge Sort is O(nlogn), a common characteristic of divide and conquer algorithms, particularly when the splits are balanced and the merging step runs in linear time.